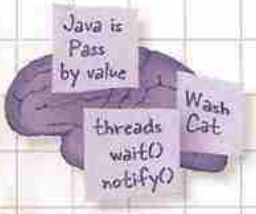
# Изучаем Java

2-е издание

Узнай, как потоки могут изменить твою жизнь

Научись программировать без ошибок

Развивай скорость соображения при помощи 42 Java-пазлов




Накрепко усвой все концепции Java

Чувствуй себя в библиотеке Java как «рыба в воде»

Закрепите полученные знания при помощи многочисленных упражнений

Кэти Сьерра и Берт Бейтс

эксмо O'REILLY®

# Head First Java

2nd Edition

Kathy Sierra,
Bert Bates

O'REILLY®

*Beijing • Cambridge • Köln • Sebastopol • Taipei • Tokyo*

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

# Изучаем Java

2-е издание

Кэти Сьерра
и Берт Бейтс

"Изучаем Java" — это не просто книга. Она не только научит вас теории языка Java и объектно-ориентированного программирования, она сделает вас программистом. В ее основу положен уникальный метод обучения на практике. В отличие от классических учебников информация дается не в текстовом, а в визуальном представлении. Из нее вы узнаете все самое нужное: синтаксис и концепции языка, работа с потоками, работа в сети, распределенное программирование. Вся теория закрепляется интереснейшими примерами и тестами.

ЭКСМО

*Москва*
*2012*

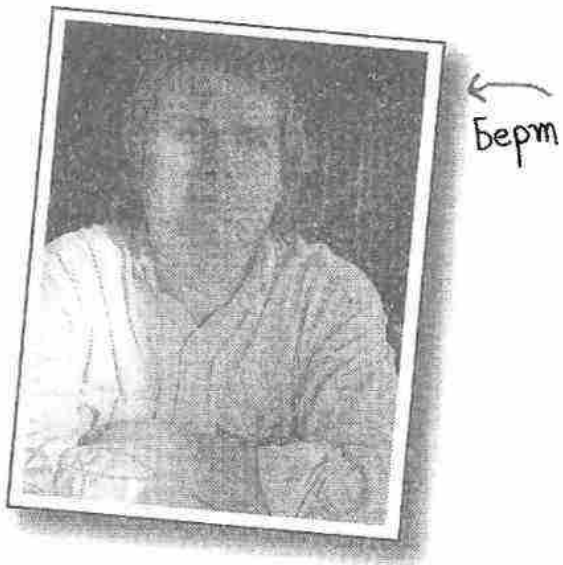
# Создатели серии Head First

**Кэти Сьерра (Kathy Sierra)** заинтересовалась теорией обучения, еще когда была разработчиком игровых приложений (для студий Virgin, MGM и Amblin). Основную концепцию оформления серии Head First она придумала, когда преподавала предмет «Разработки новых подходов в сфере распространения информации» в рамках курса «Развлекательные СМИ» в Калифорнийском университете. Позже Кэти стала главным тренером компании Sun Microsystems, где обучала других тренеров технике преподавания современных технологий, связанных с Java. В то же время она была ведущим разработчиком нескольких сертификационных экзаменов для Java-программистов. Вместе с Бертом Бейтсом активно использовала методику изложения материала, представленную в этой книге, чтобы обучить сотни преподавателей, разработчиков и обычных людей, которые не имеют отношения к программированию. Кроме того, Кэти — основатель одного из крупнейших в мире сайтов для Java-сообщества javaranch.com и блога headrush.typepad.com.

Кэти также соавтор изданий из серии Head First, посвященных сервлетам, технологии EJB и шаблонам проектирования.

В свободное от работы время Кэти занимается верховой ездой на своей исландской лошади, катается на лыжах и бегает, а также пытается преодолеть скорость света.

kathy@wickedlysmart.com




**Берт Бейтс (Bert Bates)** — разработчик программного обеспечения. Когда он трудился над проблемами искусственного интеллекта (около десяти лет), то заинтересовался теорией обучения и современными методиками преподавания. С тех пор он сам преподает программирование своим клиентам. Недавно Берт стал членом команды, разрабатывающей сертификационные экзамены для компании Sun.

Первые десять лет своей карьеры в качестве программиста Берт провел в разъездах, сотрудничал с такими вещательными компаниями, как Radio New Zealand, Weather Channel и Arts & Entertainment Network (A & E). Один из его самых любимых проектов заключался в построении полноценного симулятора сети железных дорог для компании Union Pacific Railroad.

Берт — заядлый игрок в го, и уже долгое время работает над соответствующей программой. Он прекрасный гитарист и сейчас пробует свои силы в игре на банджо. Любит кататься на лыжах, занимается бегом и тренирует свою исландскую лошадь Энди (хотя еще неизвестно, кто кого тренирует).

Берт пишет книги в соавторстве с Кэти. Они уже корпят над новой серией изданий (следите за обновлениями в блоге).

Иногда его можно застать на сервере IGC для игры в го (под псевдонимом *jackStraw*).

terrapin@wickedlysmart.com

Кэти и Берт пытаются не оставлять ни одного письма без ответа, но, учитывая количество посланий и напряженный график командировок, сделать это довольно сложно. Самый лучший (и быстрый) способ получить техническую помощь по этой книге — обратиться на очень оживленный форум для новичков в Java: javaranch.com.

# Содержание (краткое)


| | | |
|---|---|---|
| | Введение | 19 |
| 1. | Погружаемся | 31 |
| 2. | Путешествие в Объективилль | 57 |
| 3. | Свои переменные нужно знать в лицо | 79 |
| 4. | Как себя ведут объекты | 101 |
| 5. | Особо мощные методы | 125 |
| 6. | Использование библиотеки Java | 155 |
| 7. | Прекрасная жизнь в Объективилле | 195 |
| 8. | Серьезный полиморфизм | 227 |
| 9. | Жизнь и смерть объектов | 265 |
| 10. | Числа имеют значение | 303 |
| 11. | Опасное поведение | 345 |
| 12. | Очень графическая история | 383 |
| 13. | Улучшай свои навыки | 429 |
| 14. | Сохранение объектов | 459 |
| 15. | Устанавливаем соединение | 501 |
| 16. | Структуры данных | 559 |
| 17. | Выпусти свой код | 611 |
| 18. | Распределенные вычисления | 637 |
| | Приложение А. Итоговая кухня кода | 679 |
| | Приложение Б. Десять самых важных тем, которым не хватило самой малости, чтобы попасть в основную часть книги... | 689 |


# Содержание (подробное)

## Введение

**Ваш мозг по отношению к Java.** Когда *вы* стараетесь что-то изучить, ваш *мозг* пытается оказать вам услугу, убеждая, что все это не имеет никакого значения. Он думает: «Лучше сосредоточиться на том, как избежать встречи со свирепым хищником или что обнаженный сноубордист — это плохая идея». Как же *убедить* мозг в том, что от знания Java зависит ваша жизнь?


| | |
|---|---|
| Для кого эта книга | 20 |
| Мы знаем, о чем вы подумали | 21 |
| Мы знаем, о чем подумал ваш мозг | 21 |
| Метапознание: размышления о мышлении | 23 |
| Вот как вы можете подчинить себе свой мозг | 25 |
| Что необходимо для чтения этой книги | 26 |
| Технические редакторы | 28 |

# 1 Погружаемся

**Java открывает новые возможности.** Еще во времена первой (и довольно скромной) открытой версии 1.02 этот язык покорил разработчиков своим дружественным синтаксисом, объектно ориентированной направленностью, управлением памятью и, что важнее всего, перспективой переносимости на разные платформы. Соблазн написать код один раз и запускать его везде оказался слишком велик. Однако, по мере того как программистам приходилось бороться с ошибками, ограничениями и чрезвычайной медлительностью Java, первоначальный интерес к языку остывал. Но это было давно. Сегодня Java достаточно мощный и работает быстрее.

| | |
|---|---|
| Как работает Java | 32 |
| Краткая история Java | 34 |
| Структура кода в Java | 37 |
| Структура класса | 38 |
| Создание класса с методом main | 39 |
| Что можно разместить внутри главного метода | 40 |
| Зацикливаем, зацикливаем и... | 41 |
| Условное ветвление | 43 |
| Создание серьезного бизнес-приложения | 44 |
| Генератор фраз | 47 |
| Упражнения | 50 |

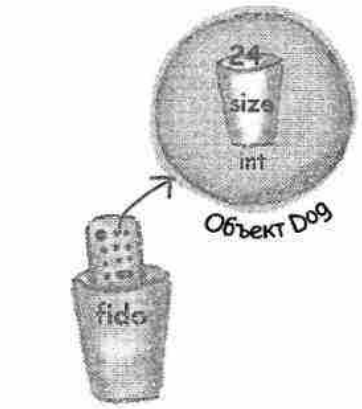

# 2 Путешествие в Объектвилль

**Мне говорили, что там будут объекты.** В главе 1 вы размещали весь свой код в методе main(), но это не совсем объектно ориентированный подход. По сути, он не имеет ничего общего с объектами. Да, вы использовали некоторые объекты, например строковые массивы для генератора фраз, но не создавали свои собственные типы объектов. Теперь вы оставите позади мир процедур, выберетесь из тесного метода main() и сами начнете разрабатывать объекты. Вы узнаете, чем же так удобно объектно ориентированное программирование (ООП) на языке Java, и почувствуете разницу между классом и объектом.

| | |
|---|---|
| Война за кресло, или Как объекты могут изменить вашу жизнь | 58 |
| Создаем первый объект | 66 |
| Создание и тестирование объектов Movie | 67 |
| Скорее! Выбирайтесь из главного метода! | 68 |
| Запускаем нашу игру | 70 |
| Упражнения | 72 |

# 3 Свои переменные нужно знать в лицо

**Переменные делятся на два вида: примитивы (простые типы) и ссылки.** В жизни должно быть что-то помимо чисел, строк и массивов. Как быть с объектом PetOwner с переменной экземпляра типа Dog? Или Cat с переменной экземпляра Engine? В этой главе мы приоткроем завесу тайны над типами в языке Java и вы узнаете, что именно можно *объявлять* в качестве переменных, какие значения *присваивать* им и как вообще с ними *работать*.



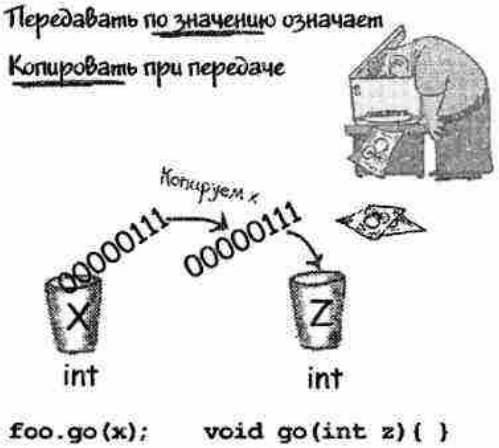
Ссылка типа Dog


| | |
|---|---|
| Объявление переменной | 80 |
| Простые типы | 81 |
| Ключевые слова в Java | 83 |
| Управление объектом | 84 |
| Управление объектом с помощью ссылки | 85 |
| Массивы | 89 |
| Упражнения | 93 |


---

# 4 Как себя ведут объекты

**Состояние влияет на поведение, а поведение — на состояние.** Вы знаете, что объекты характеризуются **состоянием** и **поведением**, которые представлены **переменными экземпляра** и **методами**. До этого момента вопрос о связи между состоянием и поведением не поднимался. Вам уже известно, что каждый экземпляр класса (каждый объект определенного типа) может иметь уникальные значения для своих переменных экземпляра. Суть объектов заключается в том, что их поведение зависит от состояния. Иными словами, ***методы используют значения переменных экземпляра***. В этой главе вы узнаете, как изменить состояние объекта.

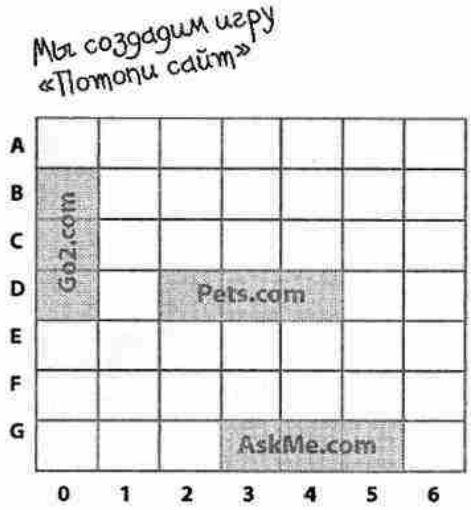




| | |
|---|---|
| Класс описывает, что объект знает и делает | 102 |
| Передаем методу разные значения | 104 |
| Получаем значения обратно из метода | 105 |
| Передаем в метод сразу несколько значений | 106 |
| Трюки с параметрами и возвращаемыми значениями | 109 |
| Инкапсуляция | 110 |
| Объекты внутри массива | 113 |
| Объявление и инициализация переменных экземпляра | 114 |
| Разница между переменными экземпляра и локальными переменными | 115 |
| Сравниваем переменные (примитивы или ссылки) | 116 |
| Упражнения | 118 |

# 5 Особо мощные методы

**Сделаем методы еще более мощными.** Вы уже поработали с переменными, несколькими объектами и написали небольшой код. Но для полноценной работы требуется гораздо больше инструментов, например **операторы** и **циклы**. Почему бы вам во время учебы не *создать* что-нибудь настоящее, чтобы собственными глазами увидеть, как с нуля пишутся (и тестируются) программы. **Может быть, это будет игра** вроде «Морского боя».





| | |
|---|---|
| Создадим аналог «Морского боя»: игра «Потопи сайт» | 126 |
| Плавное введение в игру «Потопи сайт» | 128 |
| Разработка класса | 129 |
| Записываем реализацию метода | 131 |
| Тестовый код для класса SimpleDotCom | 132 |
| Метод checkYouself() | 134 |
| Метод main() в игре | 140 |
| Поговорим о циклах for | 144 |
| Путешествия сквозь цикл | 145 |
| Улучшенный цикл for | 146 |
| Приведение простых типов | 147 |
| Упражнения | 148 |

# 6 Использование библиотеки Java

**Вместе с Java поставляются сотни готовых классов.** Можете не тратить время на изобретение собственного велосипеда, если знаете, как отыскать нужное в библиотеке Java, называемой еще **Java API**. *Думаем, у вас найдутся дела поважнее.* При написании кода сосредоточьтесь на той части, которая уникальна для вашего приложения. Стандартная библиотека Java представляет собой гигантский набор классов, готовых к применению в качестве строительных блоков.

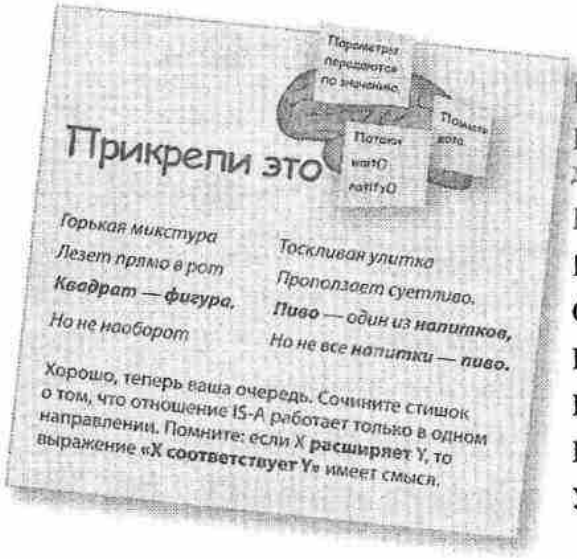
*Хорошо, когда известно, что ArrayList находится в пакете java.util. Но как бы я смогла додуматься до этого самостоятельно?*

Джулия, 31 год, модель

| | |
|---|---|
| Небольшая интрига из предыдущей главы — ошибка | 156 |
| Примеры использования ArrayList | 163 |
| Сравнение ArrayList с обычным массивом | 166 |
| Исправляем код класса DotCom | 168 |
| Новый и улучшенный класс DotCom | 169 |
| Создаем настоящую игру «Потопи сайт» | 170 |
| Что делает каждый элемент в игре DotComBust (и в какой момент) | 172 |
| Псевдокод для настоящего класса DotComBust | 174 |
| Окончательная версия класса DotCom | 180 |
| Супермощные булевы выражения | 181 |
| Использование библиотеки (Java API) | 184 |
| Как работать с API | 188 |
| Упражнения | 191 |

# 7 Прекрасная жизнь в Объективилле

**Планируйте свои программы с прицелом на будущее.** Как вы оцените способ создания Java-приложений, при котором у вас останется много свободного времени? Заинтересует ли вас создание гибкого кода, которому не страшны досадные изменения в техническом задании, возникающие в последний момент? Поверьте, вы можете получить все это лишь за три простых подхода по 60 минут каждый. Изучив принципы полиморфизма, вы узнаете о пяти шагах грамотного проектирования классов, трех приемах для достижения полиморфизма и восьми способах создания гибкого кода.

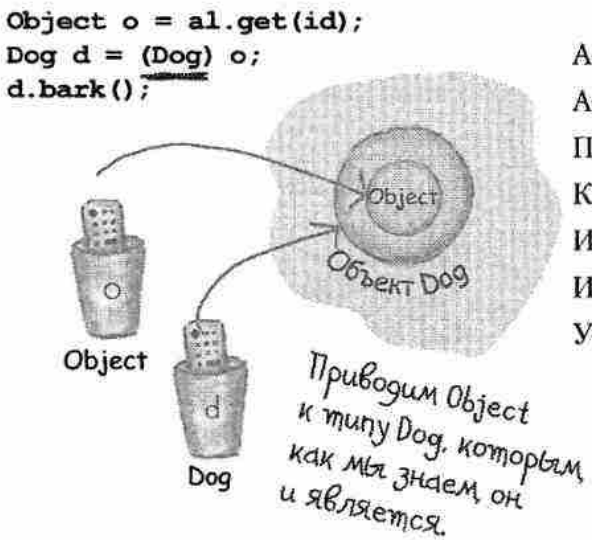




| | |
|---|---|
| Принципы наследования | 198 |
| Использование наследования для предотвращения дублирования кода в дочерних классах | 201 |
| Ищем новые возможности, которые дает наследование | 203 |
| Проектирование иерархии наследования | 206 |
| Отношения IS-A и HAS-A | 207 |
| Как узнать, что наследование оформлено правильно | 209 |
| Настоящая ценность наследования | 212 |
| Перегрузка метода | 221 |
| Упражнения | 222 |


# 8 Серьезный полиморфизм

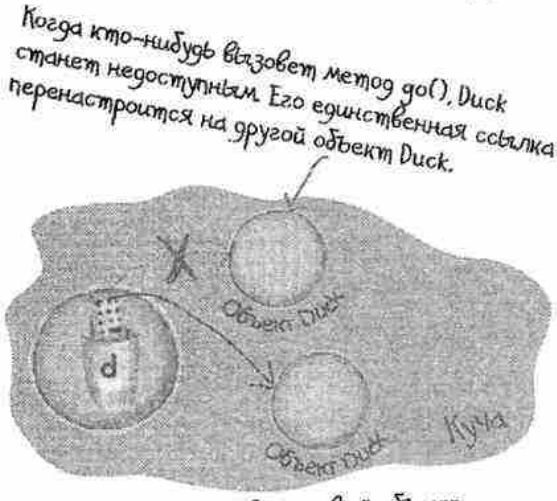
**Наследование — это только начало.** Чтобы задействовать полиморфизм, нужны интерфейсы. Новый уровень гибкости и масштабируемости может обеспечить только архитектура, основанная на интерфейсах. Мы уверены, что вы захотите их использовать. Вы удивитесь, как могли жить без них раньше. Что такое интерфейс? Это на 100 % абстрактный класс. Что такое абстрактный класс? Это класс, для которого нельзя создать экземпляр. Зачем это нужно? Читайте главу...



| | |
|---|---|
| Абстрактный против Конкретного | 232 |
| Абстрактные методы | 233 |
| Полиморфизм в действии | 236 |
| Класс Object | 238 |
| Использование полиморфических ссылок типа Object | 241 |
| Интерфейс спешит на помощь! | 254 |
| Упражнения | 260 |

# 9 Жизнь и смерть объектов

**Объекты рождаются и умирают.** Вы управляете их жизненным циклом. Вы решаете, когда и как *создавать* их. И вы решаете, когда их *уничтожать*. Но на самом деле вы не уничтожаете их, а просто делаете брошенными и недоступными. Уже после этого безжалостный **сборщик мусора** может аннулировать объекты, освобождая используемую память. В этой главе мы рассмотрим, как создаются объекты, где размещаются, как эффективно использовать их и делать недоступными.





| | |
|---|---|
| Стек и куча: где все хранится | 266 |
| Методы размещаются в стеке | 267 |
| Как работают локальные переменные, являющиеся объектами | 268 |
| Создание объекта | 270 |
| Создаем объект Duck | 272 |
| Инициализация состояния нового объекта Duck | 273 |
| Роль конструкторов родительского класса в жизни объекта | 274 |
| Как вызвать конструктор родительского класса | 283 |
| Конструкторы родительских классов с аргументами | 285 |
| Вызов одного перегруженного конструктора из другого | 286 |
| Сколько живет объект | 288 |
| Упражнения | 296 |


# 10 Числа имеют значение

**Поговорим о математике.** В Java API есть множество удобных и простых в использовании методов для работы с числами. Поскольку большинство из них статические, то сначала разберемся, какими особенностями обладают статические переменные и методы, включая константы, которые в Java считаются *статическими* финализированными переменными.

**Статические переменные универсальны для всех экземпляров класса.**

| | |
|---|---|
| Математические методы — наиболее близкие к глобальным | 304 |
| Разница между обычными (не статическими) и статическими методами | 305 |
| Что значит иметь класс со статическими методами | 306 |
| Инициализация статической переменной | 311 |
| Автоматическая упаковка: стираем границы между примитивом и объектом | 319 |
| Форматирование чисел | 324 |
| Спецификатор форматирования | 328 |
| Работа с датами | 332 |
| Работа с объектами Calendar | 335 |
| Основные методы класса Calendar | 336 |
| Статический импорт | 337 |
| Упражнения | 340 |

# 11 Опасное поведение

**Случается всякое.** То файл пропадает, то сервер падает. Не важно, насколько хорошо вы программируете, ведь невозможно контролировать *все*. Что-то может пойти не так. При создании опасного метода вам понадобится код, который будет обрабатывать возможные нестандартные ситуации. Но как узнать, опасен ли метод? И куда *поместить* код для обработки ***непредвиденной*** ситуации? В *этой* главе мы разработаем музыкальный MIDI-проигрыватель, использующий опасный JavaSound API.

Что происходит, если вы вызываете опасный метод 349
Методы в Java используют исключения, чтобы сообщить вызывающему коду: «Случилось нечто плохое. Я потерпел неудачу». 350
Исключение — объект типа Exception 352
Управление программным потоком в блоках try/catch 356
Finally: для действий, которые нужно выполнить несмотря ни на что 357
Исключения поддерживают полиморфизм 360
Приложение для проигрывания звуков 372
Создание MIDI-событий (данных о композиции) 373
MIDI-сообщение: сердце MidiEvent 374
Как изменить сообщение 375
Кухня кода 369
Упражнения 378


# 12 Очень графическая история

**Соберитесь, вам предстоит придумать GUI (graphical user interface — графический пользовательский интерфейс).** Даже если вы уверены, что всю оставшуюся жизнь будете писать код для серверных программ, где работающий на клиентской стороне пользовательский интерфейс представляет собой веб-страницу, рано или поздно вам придется создавать инструменты и вы захотите применить графический интерфейс. Работе над GUI посвящены две главы, из которых вы узнаете ключевые особенности языка Java, в том числе такие, как **обработка событий** и **внутренние классы**. В этой главе мы расскажем, как поместить на экран кнопку и заставить ее реагировать на нажатие. Кроме того, вы научитесь рисовать на экране, добавлять изображение в формате JPEG и даже создавать анимацию.

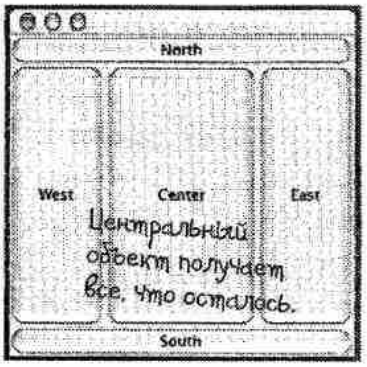
```
class MyOuter {

    class MyInner {
        void go() {
        }
    }

}
```

Ваш первый графический интерфейс: кнопка во фрейме 385
Пользовательское событие 387
Слушатели, источники и события 391
Вернемся к графике... 393
Личный виджет для рисования 394
Что интересного можно сделать в paintComponent() 395
За каждой хорошей графической ссылкой стоит объект Graphics2D 396
Компоновка графических элементов: помещаем несколько виджетов во фрейм 400
Попробуем сделать это с двумя кнопками 402
Как создать экземпляр внутреннего класса 408
Используем внутренний класс для анимации 412
Легкий способ создания сообщений/событий 418
Кухня кода 416
Упражнения 424

# 13 Улучшай свои навыки

**Swing — это просто.** Код для работы со Swing выглядит просто, но, скомпилировав его, запустив и посмотрев на экран, вы подумаете: «Эй, этот объект должен быть в другом месте». Инструмент, который *упрощает* создание *кода*, одновременно усложняет управление им — это **диспетчер компоновки**. Но, приложив небольшие усилия, вы можете подчинить диспетчер компоновки. В этой главе мы будем работать со Swing и ближе познакомимся с виджетами.

Компоненты в областях east и west вытягиваются по ширине.




Компоненты в областях north и south вытягиваются в длину.


| | |
|---|---|
| Компоненты Swing | 430 |
| Диспетчеры компоновки | 431 |
| Как диспетчер компоновки принимает решения | 432 |
| Три главных диспетчера компоновки: border, flow и box | 433 |
| Играем со Swing-компонентами | 443 |
| Кухня кода | 448 |
| Упражнения | 454 |


# 14 Сохранение объектов

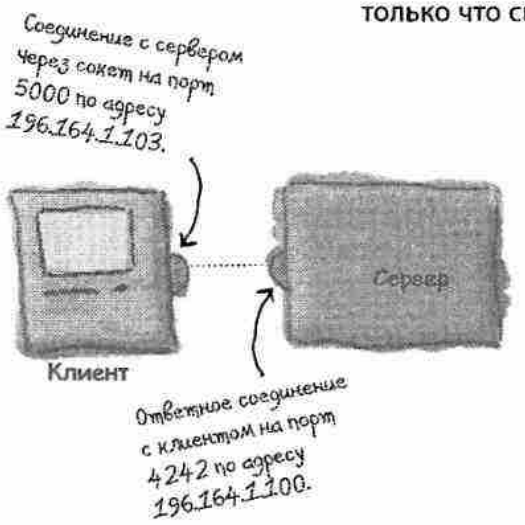
**Объекты могут быть сплющенными и восстановленными.** Они характеризуются состоянием и поведением. Поведение содержится в классе, а *состояние* определяется каждым *объектом* в отдельности. Что же происходит при сохранении состояния объекта? Если вы создаете игру, вам понадобится функция сохранения/восстановления игрового процесса. Если вы пишете приложение, которое рисует графики, вам также необходима функция сохранения/восстановления. **Вы можете сделать это легким объектно ориентированным способом** — нужно просто сублимировать/сплющить/сохранить/опустошить сам объект, а затем реконструировать/надуть/восстановить/наполнить его, чтобы получить снова.

Есть вопросы?

| | |
|---|---|
| Сохранение состояния | 461 |
| Запись сериализованного объекта в файл | 462 |
| Что на самом деле происходит с объектом при сериализации | 464 |
| Десериализация: восстановление объекта | 471 |
| Сохранение и восстановление игровых персонажей | 474 |
| Запись строки в текстовый файл | 477 |
| Пример текстового файла: электронные флеш-карты | 478 |
| Quiz Card Builder: структура кода | 479 |
| Класс java.io.File | 482 |
| Чтение из текстового файла | 484 |
| Quiz Card Player: структура кода | 485 |
| Разбор текста с помощью метода split() из класса String | 488 |
| Использование serialVersionUID | 491 |
| Сохранение схемы BeatBox | 493 |
| Восстановление схемы BeatBox | 494 |
| Кухня кода | 492 |
| Упражнения | 496 |

# 15 Устанавливаем соединение

**Свяжитесь с внешним миром.** Это просто. Обо всех низкоуровневых сетевых компонентах заботятся классы из библиотеки java.net. В Java отправка и получение данных по Сети — это обычный ввод/вывод, разве что со слегка измененным соединительным потоком в конце цепочки. Получив класс BufferedReader, вы можете считывать данные. Ему все равно, откуда они приходят — из файла или по сетевому кабелю. В этой главе вы установите связь с внешним миром с помощью сокетов. Вы создадите клиентские и серверные сокеты. В итоге у вас будут клиенты и серверы. И вы сделаете так, чтобы они смогли общаться между собой. К концу этой главы вы получите полноценный многопоточный клиент для чатов. Ой, мы только что сказали «*многопоточный*»?

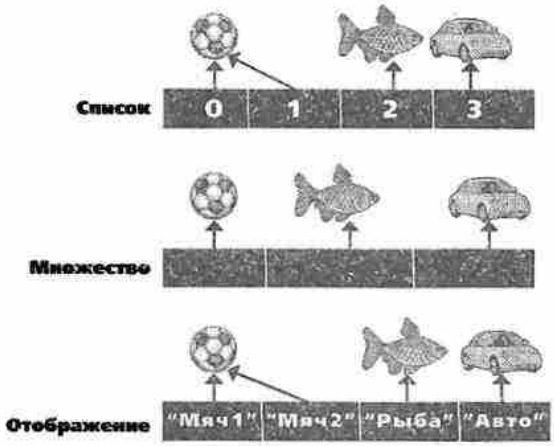




| | |
|---|---|
| Чат в BeatBox в режиме реального времени | 502 |
| Подключение, отправка и прием | 504 |
| Устанавливаем сетевое соединение с помощью сокета | 505 |
| BufferedReader для считывания данных из сокета | 508 |
| PrintWriter для записи данных в сокет | 509 |
| DailyAdviceClient | 510 |
| Код программы DailyAdviceClient | 511 |
| Создание простого сервера | 513 |
| Код приложения DailyAdviceServer | 514 |
| Создание чат-клиента | 516 |
| Несколько стеков вызовов | 521 |
| Планировщик потоков | 527 |
| Приостановление потока | 531 |
| Метод sleep() | 532 |
| Создание и запуск двух потоков | 533 |
| Использование блокировки объектов | 541 |
| Новая улучшенная версия SimpleChatClient | 548 |
| Очень-очень простой чат-сервер | 550 |
| Упражнения | 554 |

# 16 Структуры данных

**Сортировка в Java — проще простого.** У вас уже есть все необходимые инструменты для сбора данных и управления ими, поэтому не нужно писать собственные алгоритмы для сортировки. Фреймворк для работы с коллекциями в Java (Java collections framework) содержит структуры данных, которые должны подойти практически для всего, что вам может понадобиться. Хотите получить список, в который можно добавлять элементы? Вам необходимо найти что-нибудь по имени? Хотите создать список, который автоматически убирает все дубликаты? Желаете отсортировать своих сослуживцев по количеству ударов, которые они нанесли вам в спину, или домашних любимцев по количеству трюков, которые они выучили? Читайте главу...

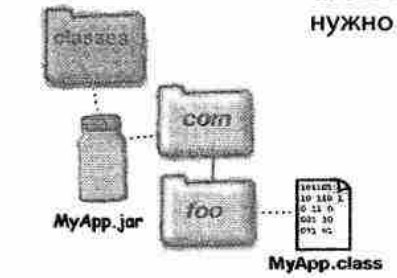


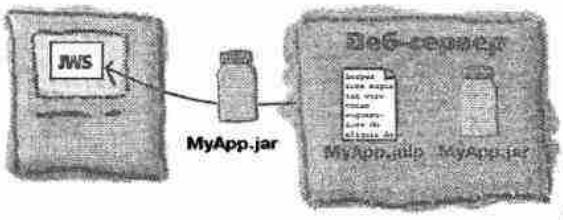


| | |
|---|---|
| Коллекции | 561 |
| Сортировка ArrayList методом Collections.sort() | 564 |
| Обобщения и безопасность типов | 570 |
| Коллекция API — List (Список), Set (Множество) и Map (Отображение, или Ассоциативный массив) | 587 |
| Переопределение hashCode() и equals() | 591 |
| Использование полиморфических аргументов и обобщений | 599 |
| Упражнения | 606 |


# 17 Выпусти свой код

**Пришло время его отпустить.** Вы написали код. Вы его протестировали и откорректировали. Вы рассказали всем знакомым, что больше не желаете видеть ни единой его строки. По большому счету вы создали произведение искусства. Это то, что действительно работает! Но что дальше? В последних двух главах мы расскажем, как организовывать, упаковывать и развертывать (внедрять или доставлять) код на языке Java. Мы рассмотрим локальный, полулокальный и удаленный варианты развертывания, включая исполняемые Java-архивы (JAR), Java Web Start, RMI и сервлеты. Большую часть этой главы мы посвятим организации и упаковыванию вашего кода — это то, что нужно знать вне зависимости от выбранного варианта доставки.



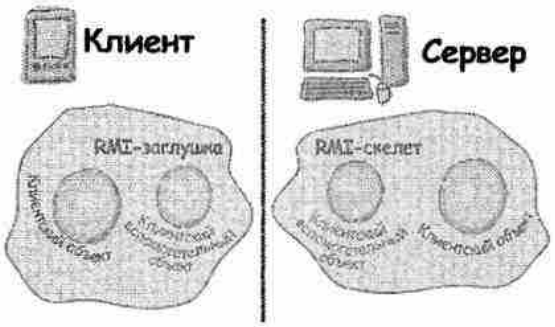





| | |
|---|---|
| Варианты развертывания | 612 |
| Отделение исходных файлов от скомпилированных | 614 |
| Помещаем программу в JAR-архив | 615 |
| Помещаем классы в пакеты | 617 |
| Предотвращение конфликтов при именовании пакетов | 618 |
| Компилируем и запускаем, используя пакеты | 620 |
| Флаг -d | 621 |
| Создание исполняемых Java-архивов с пакетами внутри | 622 |
| Java Web Start | 627 |
| JNLP-файл | 629 |
| Упражнения | 631 |

# 18 Распределенные вычисления

**Находиться далеко — это не всегда плохо.** Конечно, задача упрощается, когда все компоненты приложения собраны в одном месте и всем этим управляет одна JVM. Но это не всегда возможно. И это не всегда целесообразно. Как быть, если ваше приложение выполняет сложные вычисления и должно работать на маленьком симпатичном пользовательском устройстве? Что делать, если вашему приложению нужна информация из базы данных, но в целях безопасности к ней может получить доступ только код на вашем сервере? Мы также познакомимся с технологиями EnterpriseJavaBeans (EJB) и Jini, узнаем, в каких случаях их работа зависит от RMI. Последние страницы книги будут посвящены созданию одной из самых потрясающих программ — обозревателя универсальных сервисов.

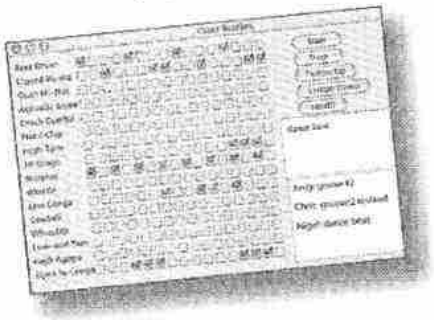




| | |
|---|---|
| Технология RMI | 644 |
| Создание удаленного сервиса | 645 |
| Сервлеты | 655 |
| Enterprise JavaBeans | 661 |
| Немного о Jini | 662 |
| Создаем обозреватель универсальных сервисов | 666 |


# А Приложение А

**Итоговая кухня кода.** Финальная версия серверной части программы BeatBox.




| | |
|---|---|
| Итоговый вариант клиентской программы BeatBox | 680 |
| Итоговый вариант серверной части программы BeatBox | 687 |


# Б Приложение Б

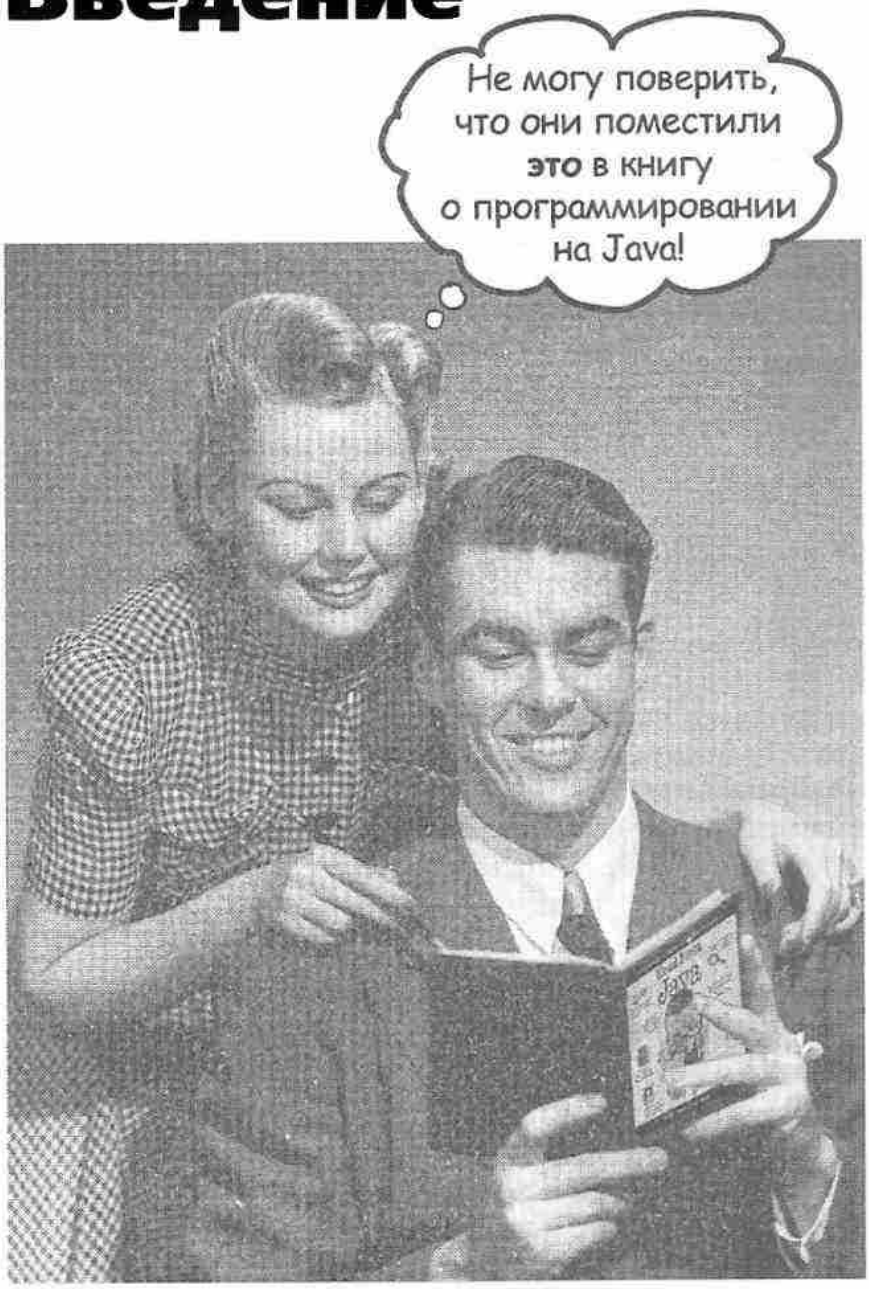
**Десять самых важных тем, которым не хватило самой малости, чтобы попасть в основную часть книги...** Вы не можете уйти, не изучив десять наиболее важных тем. И помните, что на этом книга заканчивается.


| | |
|---|---|
| Топ-десять | 689 |


# У Указатель


709

# Введение




Здесь дан ответ на животрепещущий вопрос:
«Почему же они поместили это в книгу
о программировании на Java?»

# Для кого эта книга

Если вы утвердительно ответите на все эти вопросы:

1. **Вы когда-нибудь программировали?**
2. **Вы хотите изучить язык Java?**
3. **Вы предпочитаете непринужденный разговор на званом обеде вместо сухой, скучной технической лекции?**

тогда эта книга для вас.

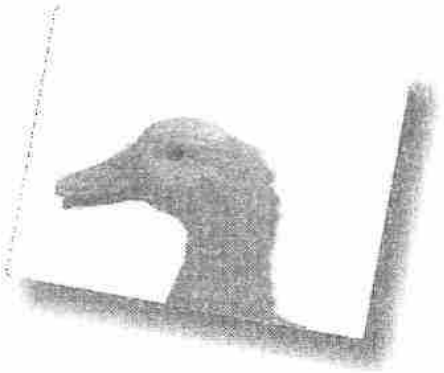
**Это не справочник. Книга предназначена для изучения Java и не содержит энциклопедических сведений об этом языке.**

# Кому лучше держаться подальше от книги

Если вы утвердительно ответите на ***любой*** из этих вопросов:

1. **Ограничивается ли ваш опыт программирования одним HTML, без сценарных языков?**
   Если вы имели дело с любыми циклами и знакомы с условиями if/then, книга вам подойдет, но одной разметки HTML может быть недостаточно.
2. **Вы опытный программист на языке C++ и ищете *справочник*?**
3. **Вы боитесь пробовать что-то новое? Вы скорее отправитесь на прием к дантисту, чем наденете полосатые брюки вместе с клетчатой рубашкой? Считаете ли вы, что книга не может быть серьезной, если в одном из ее разделов, посвященном управлению памятью, нарисован утенок?**

то эта книга ***не*** для вас.



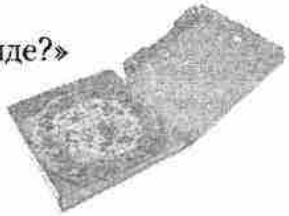
# Мы знаем, о чем вы подумали

«Насколько эта книга о программировании на Java серьезна?»

«Что это за иллюстрации?»

«Могу ли я *изучить* материал, подаваемый в таком виде?»

«Это случайно не пиццей пахнет?»



# Мы знаем, о чем подумал ваш мозг

Ваш мозг жаждет нового. Он постоянно ищет и *ожидает* необычных событий. Он так устроен, и это помогает вам в жизни.

В нынешнее время вы вряд ли станете завтраком для тигра. Но ваш мозг всегда настороже. Вы просто этого не замечаете.

Как же он поступает с обычными вещами, с которыми вы сталкиваетесь? Он делает все, чтобы они не мешали ему запоминать *действительно важную* информацию. Он не обращает внимания на скучные и очевидно бесполезные вещи, беспощадно их фильтруя.

Откуда же вашему мозгу *известно*, что важно? Предположим, вы вышли прогуляться, и тут на вас набросился тигр. Что будет твориться в вашей голове?

Нейроны накалятся. Предельная концентрация. *Химический взрыв.*

Вот откуда ваш мозг знает...

**Это очень важно! Не забудь!**

А теперь представьте, что вы находитесь дома или в библиотеке. Это безопасное и уютное место, совсем без тигров. Вы что-то читаете. Готовитесь к экзамену или пытаетесь выучить сложную техническую тему за неделю, максимум за десять дней (чего, по мнению вашего начальника, должно быть достаточно).

Но есть одна проблема. Ваш мозг хочет оказать вам услугу. Он пытается принять меры, чтобы вся эта *очевидно* ненужная информация не отнимала у вас столь дефицитные ресурсы. Ресурсы, которые лучше потратить на запоминание по-настоящему *важных* вещей. Таких как тигры, угроза пожара, обещание никогда не кататься на сноуборде в шортах.

И нельзя просто сказать мозгу: «Мозг, спасибо большое, но какой бы скучной ни была эта книга и как бы мало эмоций она у меня ни вызывала, я, правда, хочу запомнить то, что в ней написано».

*В роли читателя этой книги мы хотели бы видеть человека, который любит учиться.*

Что же нужно для того, чтобы что-нибудь *выучить*? Прежде всего вы должны *понять* прочитанное и убедиться, что не *забудете* новую информацию. Речь идет не о зубрежке фактов. Согласно последним исследованиям в области когнитивистики, нейробиологии и педагогической психологии, *процесс обучения* — это нечто намного большее, чем просто чтение текста. Мы знаем, что заставляет мозг работать.

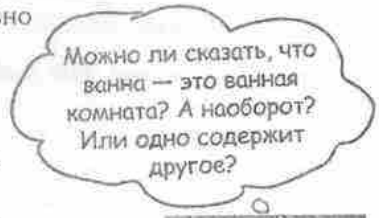
**Несколько принципов, на которых основана эта книга.**

**Визуализация.** Изображения запоминаются намного лучше, чем обычные слова, и с ними обучение становится более эффективным (увеличение запоминания и восприимчивости вплоть до 89 %). Это также делает информацию понятнее.

**При размещении слов внутри соответствующих иллюстраций или рядом с ними** вероятность возникновения у читателей проблем, связанных с содержимым книги, снижается почти *вдвое* (по сравнению с классической подачей материала, когда текст находится внизу или вовсе на другой странице).

**Разговорный стиль и персонификация.** Новейшие исследования показывают, что студенты справляются с итоговыми тестами почти на 40 % лучше, если учебники написаны от первого лица с применением разговорного стиля вместо официального. Рассказывайте истории, а не читайте лекции. Используйте неформальный язык. Не будьте слишком серьезными. Что бы вас больше заинтересовало — интересное общение на званом обеде или прослушивание лекции?

**Попытка сделать так, чтобы читатель более глубоко вник в содержимое.** Иными словами, пока вы не начнете активно напрягать свои нейроны, в вашей голове ничего не родится. Читатель должен быть достаточно мотивирован, заинтересован, заинтригован и вдохновлен, чтобы решать проблемы, делать выводы и генерировать новые знания. А для этого нужны испытания, упражнения и вопросы, заставляющие задуматься; нужно сделать так, чтобы в процессе обучения участвовали обе доли головного мозга и разные уровни сознания.



**Завоевание и удержание внимания читателя.** Все мы сталкивались с ситуацией, когда действительно нужно что-то выучить, но после первой же страницы клонит в сон. Мозг интересуется неординарными, интересными, странными, яркими и неожиданными вещами. Изучение нового сложного технического материала будет проходить намного быстрее, если вам не будет скучно.

**Эмоции.** Нам уже известно, что способность к запоминанию чего-либо во многом зависит от эмоциональной составляющей. Вы запоминаете то, что для вас важно. Вы запоминаете, когда что-то чувствуете. Нет, мы имеем в виду не душещипательные истории. Под эмоциями мы подразумеваем удивление, любопытство, веселье. Это ситуации, когда мы задумываемся над материалом и чувствуем удовлетворение от того, что решили головоломку, выучили то, что остальные считают сложным, или понимаем, что знаем нечто такое, чего *не знает* наш заносчивый приятель Боб, который якобы более подкован в технических вопросах.

# Метапознание: размышления о мышлении

Если вы действительно хотите ускорить процесс обучения, обратите внимание на то, как вы обращаете внимание. Подумайте над тем, как вы думаете. Изучите то, как вы учитесь.

Большинство из нас не оканчивали курсы метапознания и не знакомы с теорией обучения. Мы *планировали* учиться, а не «учить, как нужно учиться».

Но мы предполагаем, что, если вы держите в руках эту книгу, то хотите выучить язык Java. И, вероятно, не желаете тратить на это слишком много времени.

Чтобы получить максимальную пользу от этой (или *любой* другой) книги, возьмите на себя ответственность за свой мозг. За то, как он поступит с *новым* материалом.

Хитрость заключается в том, чтобы заставить мозг думать о новом материале как о чем-то по-настоящему важном, как будто от этого зависит ваше благосостояние. Словно перед вами тигр. В ином случае вы обречены на постоянную войну со своим мозгом в попытках заставить его запомнить новый материал.

### Как же заставить свой мозг воспринимать Java словно это не язык программирования, а голодный тигр?

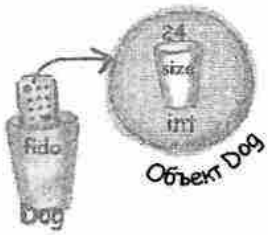
Можно пойти двумя путями: медленным и скучным или более быстрым и эффективным. Медленный путь — обычная зубрежка. Вам, конечно, известно, что упорство и труд перетрут даже самый неинтересный материал. При достаточном количестве повторений ваш мозг рано или поздно решит: «Я не *думаю*, что эта вещь для него важна, но раз он продолжает смотреть на нее *снова* и *снова*, то, наверное, в ней что-то есть».

Быстрый путь заключается ***в повышении мозговой активности***, в частности разных ее *типов*. На предыдущей странице мы уже описали большинство подходов, призванных помочь вашему мозгу работать в ваших же интересах. Например, исследования показывают, что размещение слов *внутри* изображений, которые их иллюстрируют (а не в заголовке или в основном блоке текста), заставляет мозг искать связь между текстом и картинкой, что приводит в действие больше нейронов. Больше нейронов — это больше шансов, что мозг сочтет потребляемую информацию стоящей и запомнит ее.

Помогает и разговорный стиль изложения. Люди имеют склонность больше сосредотачиваться, когда к ним кто-то обращается; они отслеживают мысль, чтобы поддержать беседу. Сложно поверить, но ваш мозг может проигнорировать тот факт, что «беседа» ведется между вами и книгой! С другой стороны, если книга выдержана в официальном и сухом стиле, ваш мозг будет вести себя так, словно вы слушаете лекцию в аудитории, наполненной сонными студентами. Нет нужды сосредотачиваться.

Но иллюстрации и разговорный стиль — это только начало.

# Вот что мы сделали




Мы использовали ***изображения***, потому что мозг приспособлен к восприятию визуальной информации, а не текста. С его точки зрения, картинка действительно *стоит* 1024 слов. Мы поместили ключевой текст *внутрь* иллюстраций, на которые он ссылается, а не в заголовок или куда-либо еще, потому что так мозг работает более эффективно.

Мы задействовали ***повторения***, выражая одни и те же вещи разными способами и с помощью разных материалов, увеличивая шансы на то, что информация будет закодирована сразу в нескольких участках вашего мозга.

Мы использовали понятия и изображения так, чтобы вас ***удивить***, потому что мозг приспособлен к восприятию чего-то нового. Мы ввели *некую* ***эмоциональную составляющую***, потому что мозг восприимчив к биохимии эмоций. Благодаря этому вы *чувствуете*, что данную информацию лучше запомнить, даже если ощущение основано лишь на ***юморе***, ***неожиданности*** или ***интересе***.

Мы применяли персонифицированный, ***разговорный стиль изложения***, так как мозг больше сосредотачивается при общении, а не во время пассивного просмотра презентации. Такой подход срабатывает даже при *чтении*.

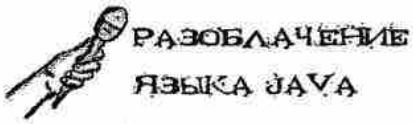
Головоломки

Мы добавили более 50 ***упражнений***, так как мозг может выучить и запомнить больше, если вы ***выполняете*** какие-то вещи, а не просто *читаете* о них. Мы сделали упражнения сложными, но выполнимыми, потому что именно этого хотят большинство *людей*.

Мы использовали ***несколько стилей подачи материала***, ведь все люди разные. Одни могут отдавать предпочтение пошаговым методикам, другие хотят сначала понять общую картину, третьим нужно увидеть пример кода. Но вне зависимости от личных пристрастий каждый читатель получает пользу от знакомства с материалом благодаря разным подходам.

КЛЮЧЕВЫЕ МОМЕНТЫ

Мы подготовили материал для ***обеих половин вашего мозга***, потому что чем больше его участков вы задействуете, тем лучше пройдет обучение и запоминание и тем дольше вы сможете оставаться сконцентрированным. Поскольку нагрузка на одну половину мозга означает, что у другой есть шанс отдохнуть, ваши занятия становятся более продуктивными и продолжительными.

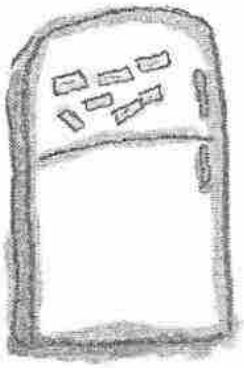



Мы также включили в книгу ***рассказы*** и упражнения, которые предоставляют ***несколько разных точек зрения***, так как ваш мозг более глубоко вникает в проблему, если заставить его давать оценки и делать выводы.

Штанга для мозга

Помимо упражнений мы добавили ***сложные задачи***, ставя в них ***вопросы***, на которые не всегда можно ответить однозначно. Это связано с тем, что мозг лучше обучается и запоминает во время какой-то *работы* (то же самое с вашим *телом* — чтобы поддерживать его в хорошей форме, недостаточно наблюдать за посетителями фитнес-клуба). Но мы сделали все от нас зависящее, чтобы направить ваши усилия в *правильное* русло. Мы старались, чтобы вы ***не потратили ни одного лишнего дендрита*** на работу с тяжелым для понимания примером или на разбор сложного либо, наоборот, чрезмерно лаконичного текста.

Мы использовали закон Парето (или принцип ***80/20***) и исходили из того, что, если вы хотите получить ученую степень в Java, это будет не единственная книга, которую вы прочтете. Поэтому мы не пытались рассказать обо *всем*. Мы рассматривали только те вещи, которые вам действительно *пригодятся*.



# Вот как вы можете подчинить себе свой мозг

Итак, свою работу мы выполнили. Дальше все зависит от вас. Эти советы — лишь отправная точка. Прислушивайтесь к своему мозгу и сами определяйте, какие пункты вам подходят, а какие — нет. Можете попробовать что-нибудь новое.

Вырежьте это и приклейте на свой холодильник.

1. **Не спешите. Чем больше вы поймете, тем меньше вам придется запоминать.**
Недостаточно просто *читать*. Нужно останавливаться и обдумывать прочитанное. Когда в книге вам задается вопрос, не пропускайте его. Представьте, что кто-то действительно вас спрашивает. Чем более глубоко вы заставите свой мозг вникать в материал, тем больше у вас шансов что-нибудь выучить и запомнить.

2. **Выполняйте упражнения. Делайте свои заметки.**
Мы включили их в книгу, но не будем делать их за вас — это как тренироваться вместо другого человека. Кроме того, недостаточно только *смотреть* на них. **Пользуйтесь карандашом.** Давно доказано, что физическая активность во время занятий способствует процессу обучения.

3. **Читайте «Это не глупые вопросы».**
Все без исключения! Это не дополнение, а часть основного материала! Иногда вопросы более полезны, чем сами ответы.

4. **Не читайте книгу в одном месте.**
Встаньте, разомнитесь, подвигайтесь, пересядьте на другое кресло, перейдите в другую комнату. Это поможет вашему мозгу что-то *почувствовать* и не даст вам слишком тесно ассоциировать учебу с конкретным местом.

5. **Постарайтесь, чтобы эта книга была последней перед сном. По крайней мере не читайте после нее ничего, что бы вас напрягало.**
Процесс изучения (особенно сохранение информации в долговременной памяти) частично происходит *после* того, как вы откладываете книгу в сторону. Мозгу нужно время, чтобы обработать прочитанное. Если во время этого процесса вы попытаетесь запомнить что-то новое, то часть изученного материала будет утрачена.

6. **Пейте воду. Много воды.**
Лучше всего мозгу работается во влажной среде. Обезвоживание (которое может наступить даже до того, как вы почувствуете жажду) снижает способность к познанию.

7. **Разговаривайте о прочитанном. Вслух.**
При разговоре активизируются разные части мозга. Если вы пытаетесь что-то понять или хотите с большей долей вероятности запомнить прочитанное, попробуйте произнести это вслух. А еще лучше попытайтесь объяснить это на словах кому-нибудь другому. Процесс изучения ускорится, и у вас могут появиться идеи, о которых вы не задумывались при чтении.

8. **Прислушивайтесь к своему мозгу.**
Следите за тем, чтобы ваш мозг не был перегружен. Если вы начинаете поверхностно воспринимать или забывать только что прочитанный текст, значит, пришло время сделать перерыв. Пройдя определенный рубеж, вы уже не сможете ускорить свое обучение, увеличивая объем материала. Более того, это может даже навредить.

9. **Чувствуйте что-нибудь!**
Вашему мозгу нужно знать, что это *важно*. Пытайтесь прочувствовать рассказы, которые читаете. Делайте собственные заголовки для рисунков. Вздохнуть от плохой шутки все же лучше, чем совсем ничего не ощутить.

10. **Набирайте и запускайте код.**
Набирайте и запускайте код из примеров. Потом вы сможете экспериментировать, изменяя и улучшая его (или переписывая полностью — это иногда лучший способ понять, что происходит на самом деле). Длинные примеры или заранее подготовленный код можно загрузить в виде исходных файлов с сайта headfirstjava.com.

# Что необходимо для чтения этой книги

Вам *не* нужны дополнительные инструменты, в том числе интегрированная среда разработки (IDE). Мы настоятельно рекомендуем вам *не* использовать ничего, кроме простого текстового редактора, пока вы не дочитаете эту книгу (*особенно* до окончания главы 16). IDE может скрыть от вас действительно важные детали, поэтому при изучении языка лучше пользоваться командной строкой, а уже потом, вникнув в происходящее, перейти к работе с инструментами, которые автоматизируют процесс разработки.

## ПОДГОТОВКА ИНСТРУМЕНТОВ ДЛЯ JAVA

- Если у вас еще нет пакета Java 2 Standard Edition SDK (Software Development Kit — комплект средств разработки) версии 1.5 или выше, придется его установить. Если вы работаете в Linux, Windows или Solaris, то можете получить его бесплатно на сайте java.sun.com (сайт компании Sun для Java-программистов). Обычно загрузка J2SE занимает не больше двух щелчков кнопкой мыши на главной странице. Вам нужна последняя стабильная (не beta) версия. SDK содержит все, что необходимо для компиляции и запуска программ на языке Java.

  Если вы работаете с Mac OS X 10.4, то Java SDK у вас уже установлен и вам не нужно выполнять дополнительных действий. Предыдущие версии OS X содержат более ранние версии Java, которые, впрочем, совместимы с 95 % кода из этой книги.

  Примечание: книга основана на Java 1.5, но по непонятным маркетинговым причинам вскоре после выпуска компания Sun переименовала эту версию в Java 5, сохранив при этом обозначение 1.5 для комплекта разработки. Поэтому, встречая такие названия как Java 1.5, Java 5, Java 5.0 или Tiger (изначальное кодовое имя пятой ветки), знайте, что это одно и то же. Java 3.0 или 4.0 никогда не существовал — был сделан скачок с версии 1.4 к 5.0, но в некоторых местах еще используют обозначение 1.5 вместо 5. Еще один интересный момент — Java 5 и Mac OS X 10.4 получили одно и то же кодовое имя — Tiger, поэтому иногда вы можете услышать, как люди говорят о «Тигре на Тигре». Это всего лишь означает «Java 5 на OS X 10.4».

- SDK не содержит документации по API, но она вам нужна! Вернитесь на сайт java.sun.com и загрузите документацию по J2SE API. Можно читать ее прямо в браузере, но это очень неудобно. Лучше все же загрузить.

- Вам нужна программа для набора текста. В сущности, подойдут любые текстовые редакторы (vi, emacs, pico), включая поставляемые вместе с операционными системами. Блокнот, WordPad, TextEdit и т. д. — все сгодятся (только следите, чтобы к названиям ваших исходных файлов не добавлялось расширение .txt).
- Загрузив и распаковав SDK (метод распаковки зависит от версии вашей ОС), нужно добавить в переменную среды PATH значение, которое указывает на директорию /bin внутри главного каталога с Java. Например, если J2SDK помещен в директорию j2sdk1.5.0 на вашем диске, то в ней вы найдете каталог bin, хранящий утилиты (инструменты). Его нужно добавить в PATH, чтобы при наборе

  ```
  % javac
  ```

  в командной строке ваш терминал знал, где искать компилятор javac.

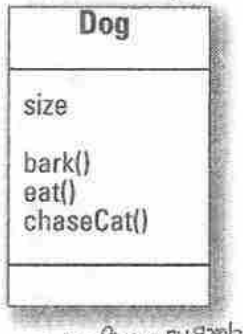
  Примечание: если у вас возникли проблемы с установкой, сходите на сайт javaranch.com и присоединитесь к форуму Java-Beginning! Хотя вам стоит сделать это в любом случае.

Еще одно примечание: большая часть кода из этой книги доступна на сайте wickedlysmart.com.

# Еще кое-что, о чем вам нужно знать

Наша книга основана на реальном опыте, это не справочник. Мы умышленно выбросили из нее все, что может помешать *изучению* подготовленного нами материала. Нужно начать с первых страниц, так как дальше информация подается исходя из того, что вы уже видели и выучили.

Мы используем упрощенный модифицированный псевдо-UML.




### Мы используем простые диаграммы, похожие на UML.

Если бы мы воспользовались *настоящим* UML, вы бы увидели нечто *похожее* на Java, но с абсолютно *неправильным* синтаксисом. Поэтому мы прибегли к упрощенной версии UML, которая не противоречит синтаксису Java. Если вы еще не знаете UML, не волнуйтесь — вы сможете изучить его *вместе* с Java.

Вы должны выполнять все упражнения под заголовком «Наточите свой карандаш».

### Мы не обращаем внимания на организацию и оформление кода вплоть до заключительных глав.

С этой книгой вы можете непосредственно изучать язык Java, не отвлекаясь на организационные и административные подробности. В реальном мире вам *придется* знать и использовать эти вещи, поэтому они тоже будут рассмотрены. Но мы приберегли их для предпоследней главы 17. Расслабьтесь и спокойно погружайтесь в мир Java.

Наточите свой карандаш

### Упражнения, которыми заканчиваются главы, обязательны для выполнения; головоломки — на ваше усмотрение. Ответы и решения размещаются в конце каждой главы.

Все, что вам нужно знать о головоломках, — над ними нужно *ломать голову*. Как и над логическими загадками, задачами с подвохом, кроссвордами и т. д. *Упражнения* нужны для того, чтобы помочь вам применить полученные знания на практике, и придется выполнить каждое из них. С головоломками все иначе, и некоторые довольно сложные в своем роде. Головоломки созданы для тех, кто *любит их решать*, и вы уже, наверное, знаете, относитесь ли вы к их числу. Но если вы не уверены, то просто попробуйте и не отчаивайтесь, если ничего *не получится*.

Упражнения, рядом с которыми нарисованы кроссовки, обязательны для выполнения! Не пропускайте их, если вы серьезно намерены выучить Java.

Упражнение

### У упражнений под заголовком «Наточите свой карандаш» нет ответов.

По крайней мере в этой книге их нет. На часть из них просто нельзя ответить правильно. Остальные же изначально подразумевают, что *вы* сами, основываясь на пройденном материале, должны решить, когда давать свой ответ (и будет ли он правильным). Некоторые наши *рекомендованные* ответы доступны на сайте wickedlysmart.com.

Головоломки, рядом с которыми нарисован кусочек мозаики, решать не обязательно. Если вы не любите мудреную логику или кроссворды, они вам вряд ли понравятся.

### Примеры кода максимально упрощены.

Неудобно, если приходится пересматривать 200 строк кода ради тех двух, которые нужны для понимания происходящего. Большинство примеров в этой книге представлены в максимально урезанном виде, чтобы изучаемая часть была простой и понятной. По этой причине не стоит ждать от кода целостности или даже завершенности. Такими свойствами будет обладать ваш код после того, как вы дочитаете книгу. Наши примеры создавались специально для *изучения*, поэтому они не всегда полноценны.

# Технические редакторы

«Заслуги приписывают всем, но за ошибки всегда отвечает автор...» Кто-нибудь действительно в это верит? Видите этих двоих? Если вы найдете в книге технические ошибки, то, скорее всего, это *их* вина :)

**Джесс** работает в компании Hewlett-Packard и вместе со своей командой занимается самовосстанавливающимися сервисами. Она имеет степень бакалавра университета Виллановы, прошла сертификации SCJP 1.4 и SCWCD и находится буквально в шаге от получения степени магистра программирования в университете Дрексела (вот так!).

В свободное время, не занятое работой, учебой, вязанием крючком (кому-нибудь нужна шляпка?) и разъездами на Мини Купере, Джесс может отвлечься и поиграть со своей кошкой. Родом она из Солт-Лейк-Сити, штат Юта (нет, она не мормон... только не делайте вид, будто не собирались об этом спросить) и сейчас проживает недалеко от Филадельфии вместе со своим мужем и двумя кошками.

Вы можете застать ее за модерированием технических форумов на сайте javaranch.com.

**Валентин Креттас** имеет степень магистра информационных и компьютерных наук Швейцарского федерального технологического института Лозанны (EPFL). Он работает программистом в исследовательском институте SRI International (Менло-Парк, Калифорния) и занимает должность главного инженера в лаборатории программного обеспечения при EPFL.

Валентин — один из основателей и технический директор компании Condris Technologies, специализирующейся на разработке программного обеспечения.

Как исследователь и разработчик он интересуется аспектно ориентированными технологиями, проектировочными шаблонами, веб-сервисами и архитектурой программного обеспечения. Валентин увлекается садоводством, любит читать и заниматься спортом. Кроме того, он часто модерирует форумы по SCBCD и SCDJWS на сайте javaranch.com. Он имеет сертификаты SCJP, SCJD, SCBCD, SCWCD и SCDJWS и является соавтором симулятора прохождения экзаменов по SCBCD от компании Whizlabs.

Мы все еще не можем поверить, что он натянул на себя *галстук*.

поблагодарить

# Другие люди, которых можно порхать:

Некоторые наши экспертные рецензенты...

***В издательстве O'Reilly***

Выражаем наивысшую признательность **Майку Лукидису** (Mike Loukides) из O'Reilly за то, что он помог оформить серию Head First. Сейчас, когда к печати готовится второе издание, у нас уже есть пять книг из этой серии, и все это время Майк был с нами.

Благодарим **Тима О'Рейлли** (Tim O'Reilly) за его готовность пропускать в печать нечто *совершенно* новое и необыкновенное. Спасибо талантливому **Кайлу Харту** (Kyle Hart) за подготовку этой серии. И, наконец, спасибо **Эдди Фридману** (Edie Freedman) за дизайн обложек для Head First, в которых «акцент сделан на *голову*».

***Бесстрашные бета-тестеры и рецензенты***

Наши высочайшие почтение и благодарность адресованы главе команды рецензентов **Йоханнесу де Йонгу** (Johannes de Jong). Это уже пятая книга из серии Head First, над которой мы вместе работали, и мы рады, что он все еще с нами. **Джефф Кампс** (Jeff Cumps) неустанно ищет неточности и некорректные фрагменты уже в третьей нашей книге.

**Кори Макглоун** (Corey McGlone), ты крут. Мы считаем, что на форумах javaranch ты даешь наиболее понятные объяснения. И ты уже, наверное, заметил, что мы стащили парочку из них. **Джейсон Менард** (Jason Menard) неоднократно нас выручал, обращая внимание на технические детали. **Томас Пол** (Thomas Paul), как обычно, предоставлял нам экспертные отзывы и находил тончайшие нестыковки, связанные с Java, которые никто другой не смог увидеть. **Джейн Гристи** (Jane Griscti), что называется, собаку съела в работе с Java (имея при этом кое-какое представление и о *писательском деле*), и нам было очень приятно получить ее помощь при написании новой книги. То же самое относится и к давнему участнику форумов javaranch **Барри Гаунту** (Barry Gaunt).

**Мэрилин де Кейруш** (Marilyn de Queiroz) оказала нам неоценимую помощь при подготовке *обоих* изданий этой книги. **Крис Джонс** (Chris Jones), **Джон Найквист** (John Nyquist), **Джеймс Кубета** (James Cubeta), **Терри Кубета** (Terri Cubeta) и **Айра Беккер** (Ira Becker) внесли огромный вклад в создание первого издания.

Особую благодарность выражаем участникам команды, которые были с нами с самого начала: **Анджело Селесте** (Angelo Celeste), **Микалаю Зайкину** (Mikalai Zaikin) и **Томасу Даффу** (Thomas Duff) (twduff.com). И спасибо нашему отличному агенту **Дэвиду Роджелбергу** (David Rogelberg) из StudioB (а если серьезно, что там с правами на съемки *фильма*?).

# И когда вы уже начали думать, что благодарности закончились*...

***Технические эксперты в области Java, которые также выручили нас при подготовке первого издания (в псевдослучайном порядке):***

Эмико Хори (Emiko Hori), Михаэль Таупиц (Michael Taupitz), Майк Галихью (Mike Gallihugh), Маниш Хатволн (Manish Hatwalne), Джеймс Чегуидден (James Chegwidden), Швета Матур (Shweta Mathur), Мохамед Мазахим (Mohamed Mazahim), Джон Певерд (John Paverd), Джозеф Бих (Joseph Bih), Скулрат Патанаванич (Skulrat Patanavanich), Сунил Палича (Sunil Palicha), Суддхазатва Гош (Suddhasatwa Ghosh), Рамки Сринивасан (Ramki Srinivasan), Альфред Роуф (Alfred Raouf), Анджело Селеста (Angelo Celeste), Джон Зутебир (John Zoetebier), Джим Пледжер (Jim Pleger), Барри Гаунт (Barry Gaunt) и Марк Дьелен (Mark Dielen).

***Команда, создавшая головоломки для первого издания:***

Дик Шрекманн (Dirk Schreckmann), Мари «Чемпион по Кроссвордам» Ленерс (Mary «JavaCross Champion» Leners), Родни Дж. Вудрофф (Rodney J. Woodruff), Гэвин Бонг (Gavin Bong) и Джейсон Менард (Jason Menard. Javaranch). Участники форумов Javaranch благодарны вам за помощь.

***Другие соучастники, которых хотелось бы поблагодарить:***

**Пол Уитон** (Paul Wheaton), участник форумов javaranch, который стал проводником в мир Java для тысяч людей.

**Сольвейг Хогленд** (Solveig Haugland), эксперт в J2EE и автор книги «Dating Design Patterns».

Писатели **Дори Смит** (Dori Smith) и **Том Негрино** (Tom Negrino) (backupbrain.com) помогли нам сориентироваться в мире технической литературы.

Наши соучастники по циклу Head First — **Эрик Фриман** (Eric Freeman) и **Бэт Фриман** (Beth Freeman) (авторы книги о шаблонах проектирования). Благодаря уверенности, которую они нам внушили, мы успели закончить эту книгу вовремя.

**Шерри Доррис** (Sherry Dorris), спасибо за вещи, которые действительно имеют значение.

***Смелые первопроходцы, которые одними из первых приступали к чтению книг серии Head First:***

Джо Литтон (Joe Litton), Росс П. Голдберг (Ross P. Goldberg), Доминик Да Силва (Dominic Da Silva), honestpuck, Дэнни Бромберг (Danny Bromberg), Стивен Лепп (Stephen Lepp), Элтон Хагс (Elton Hughes), Эрик Кристенсен (Eric Christensen), Вулин Нгуен (Vulinh Nguyen), Марк Рау (Mark Rau), Abdulhaf, Натан Олифант (Nathan Oliphant), Майкл Брэдли (Michael Bradly), Алекс Даррой (Alex Darrow), Майкл Фишер (Michael Fischer), Сара Нотингем (Sarah Nottingham), Тим Аллен (Tim Allen), Боб Томас (Bob Thomas) и Майк Бибби (Mike Bibby) (первый).

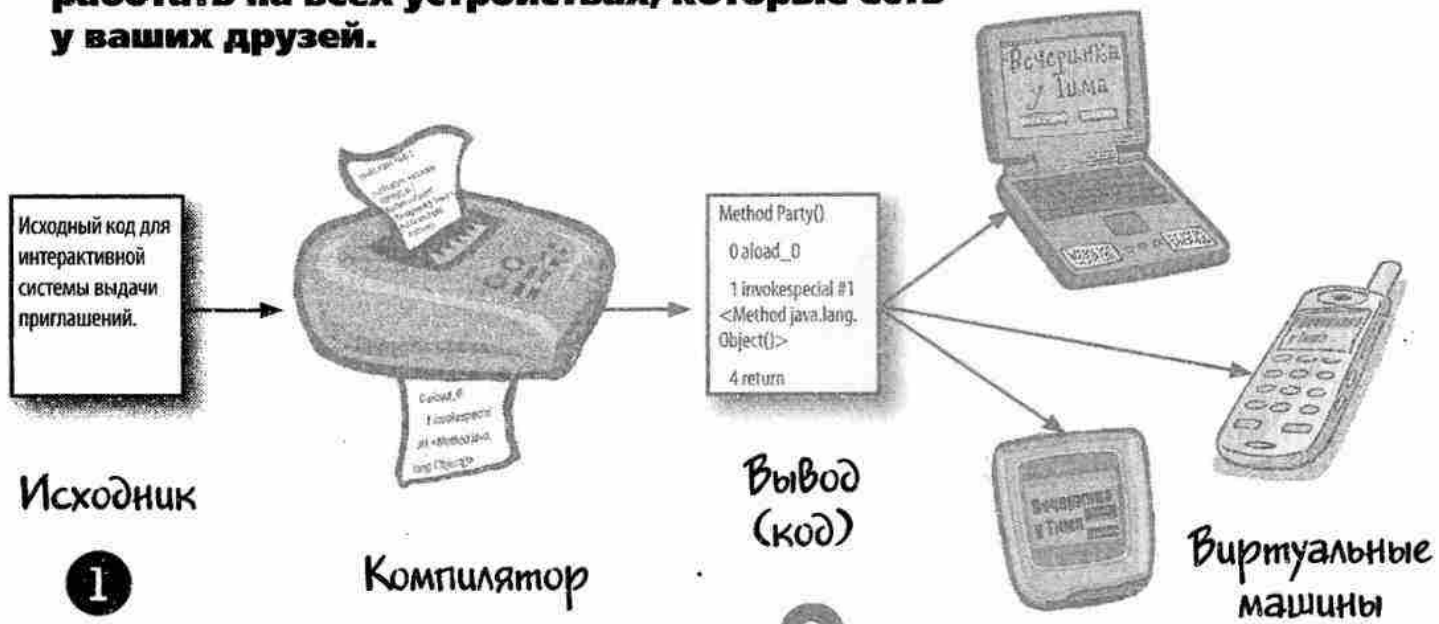
* Такое огромное количество благодарностей связано с тем, что мы хотим проверить одну теорию. А именно: каждый человек, упомянутый в книге, купит как минимум один ее экземпляр, а может, и больше (чтобы подарить родственникам и т. д.). Если вы хотите попасть в список благодарностей для нашей следующей книги и у вас большая семья, пишите нам.

# Погружаемся

**Java открывает новые возможности.** Еще во времена первой (и довольно скромной) открытой версии 1.02 этот язык покорил разработчиков своим дружественным синтаксисом, объектно ориентированной направленностью, управлением памятью и, что важнее всего, перспективой переносимости на разные платформы. Соблазн **написать код один раз и запускать его везде** оказался слишком велик. Однако, по мере того как программистам приходилось бороться с ошибками, ограничениями и чрезвычайной медлительностью Java, первоначальный интерес к языку остывал. Но это было давно. Если вы только сейчас начинаете изучать Java, вам повезло. Вашим предшественникам приходилось преодолевать целые километры босиком по заснеженным склонам лишь для того, чтобы заставить работать простой апплет. Однако сегодня в Java **гораздо меньше проблем, работает он быстрее и обладает значительной мощью**.

# Как работает Java

**Ваша цель — написать приложение (в данном случае это интерактивная система выдачи приглашений на вечеринку) и заставить его работать на всех устройствах, которые есть у ваших друзей.**




1

Создайте исходный документ. Используйте оговоренный протокол (в нашем случае это язык Java).

2

Пропустите документ через компилятор исходных кодов. Он проверит код на ошибки и выдаст конечный результат только тогда, когда будет уверен в его работоспособности и корректности.

3

Компилятор создает новый документ, закодированный с помощью байт-кода. Любое устройство, способное выполнять Java, сможет интерпретировать/транслировать этот файл в такой формат, который потом получится запустить. Скомпилированный байт-код не зависит от платформы.

4

У ваших друзей нет физической Java-машины, но есть виртуальная (реализованная программно), которая работает внутри электронных устройств. Виртуальная машина считывает и выполняет байт-код.

# Что вы будете делать с Java

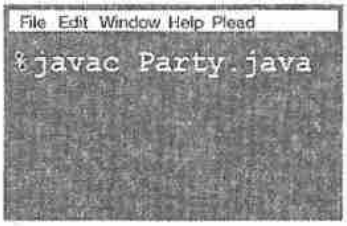
**Вы создадите файл с исходным кодом, скомпилируете его с помощью компилятора javac, а потом запустите полученный байт-код внутри виртуальной машины Java.**

## Исходник

1

Наберите свой исходный код.

Сохраните его как ***Party.java***.

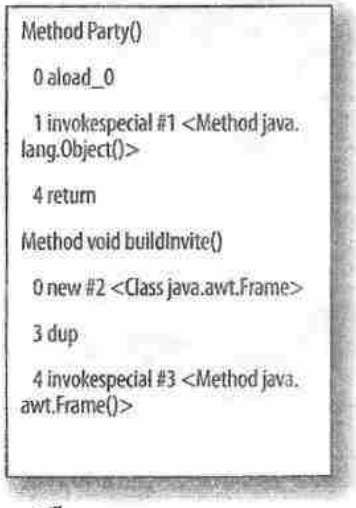



## Компилятор

2

Скомпилируйте файл ***Party.java***, запустив утилиту javac (приложение-компилятор). Если все пройдет без ошибок, вы получите еще один файл с именем ***Party.class***.

Файл **Party.class**, сгенерированный компилятором, состоит из байт-кода.




## Вывод (код)

3

Скомпилированный код: файл ***Party.class***.

## Виртуальные машины

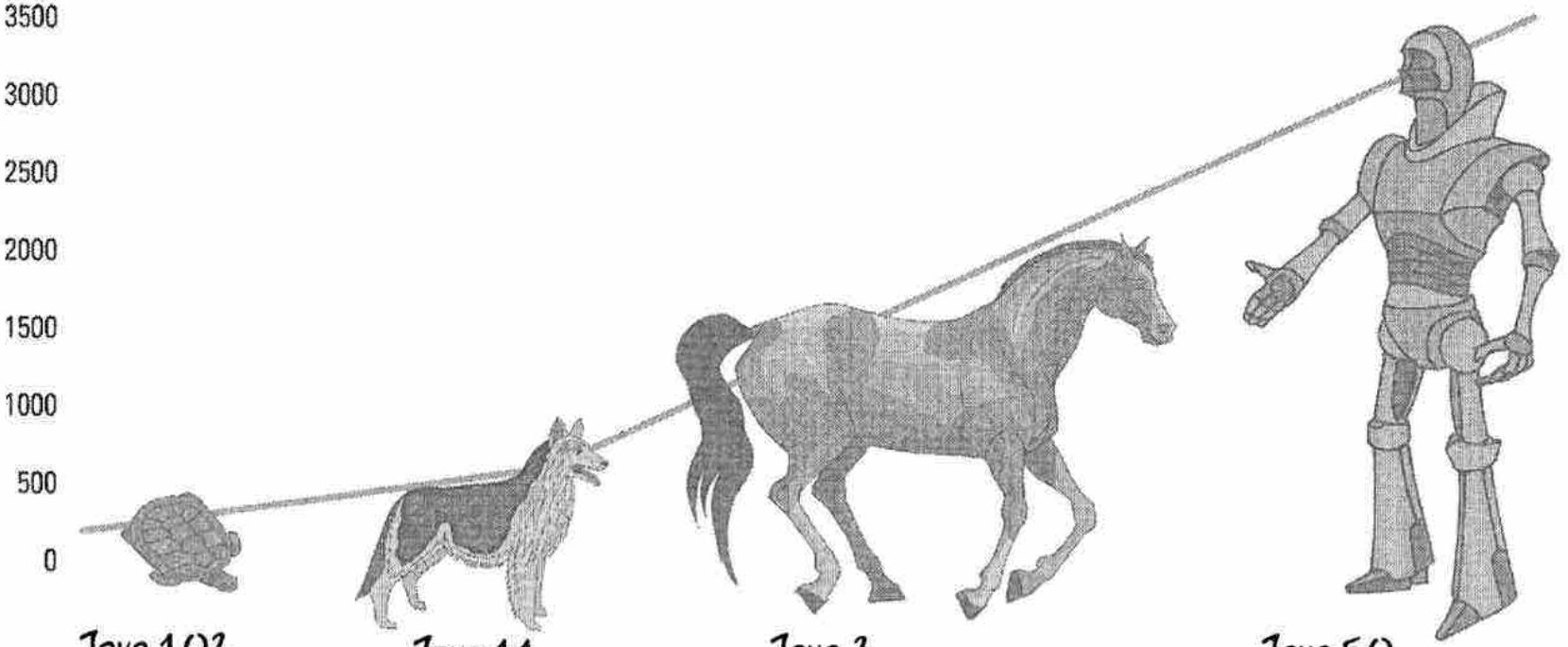
4

Выполните программу, запустив виртуальную машину Java (Java Virtual Machine, или JVM) с файлом **Party.class**. JVM транслирует байт-код в такой формат, который поймет целевая платформа, и запустит вашу программу.

Примечание: не воспринимайте это как учебное пособие. Уже совсем скоро вы начнете писать настоящий код, а пока мы хотим, чтобы вы поняли, как это работает все вместе.

# Краткая история Java

**Классы в стандартной библиотеке Java**




## Java 1.02

250 классов.

Работает ***медленно***.

Забавное имя и логотип.

Интересная в использовании версия. Множество ошибок. Достоинство — ***апплеты***.

## Java 1.1

500 классов.

Работает ***чуть*** быстрее.

Больше возможностей и ***комфортнее*** для разработчика.

Версия становится очень ***популярной***.

Улучшается код для GUI.

## Java 2 (versions 1.2–1.4)

2300 классов.

***Значительно быстрее.***

Может (иногда) соперничать с машинным кодом. Серьезная, ***мощная*** версия. Поставляется в трех вариантах: Micro Edition (J2ME), Standard Edition (J2SE) и Enterprise Edition (J2EE). В этот период Java становится ***стандартным языком*** для разработки новых промышленных систем (особенно веб-ориентированных) и мобильных приложений.

## Java 5.0 (versions 1.5 and up)

3500 классов.

Язык стал ***мощнее и проще в разработке***.

Добавлено более тысячи новых классов. Версия Java 5.0 (известная как Tiger) характеризуется изменениями в самом языке. Он стал более простым для программистов (по крайней мере в теории) и получил новые популярные возможности из других языков.

## Посмотрите, как легко писать код на Java.

Попробуйте угадать, что делает каждая строка этого кода. Ответы вы найдете на следующей странице.

```
int size = 27;
String name = "Фидо";
Dog myDog = new Dog(name, size);
x = size - 5;
if (x < 15) myDog.bark(8);

while (x > 3) {
  myDog.play();
}

int[] numList = {2,4,6,8};
System.out.print("Привет");
System.out.print("Собака: " + name);
String num = "8";
int z = Integer.parseInt(num);

try {
  readTheFile("myFile.txt");
}
catch(FileNotFoundException ex) {
  System.out.print("Файл не найден.");
}
```

| Объявляем целочисленную переменную с именем size и присваиваем ей значение 27. |
|---|
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |

## Это не глупые вопросы

**В: Я вижу версии Java 2 и 5.0, но существовали ли Java 3 и 4? И почему есть Java 5.0, но нет 2.0?**

О: При смене версии с 1.1 на 1.2 изменения в Java оказались настолько существенными, что маркетологи решили придумать совершенно новое «имя» — Java 2, хотя на самом деле это была версия 1.2. Версии 1.3 и 1.4 тоже относились к линейке Java 2, а 3 или 4 никогда не существовали. Анализируя версию 1.5, маркетологи опять решили, что изменений слишком много и понадобится новое имя (и большинство разработчиков согласились с ними), поэтому начали рассматривать разные варианты. Согласно логике следующим должно было стать название Java 3, но применительно к версии 1.5 оно еще больше сбивало с толку, поэтому маркетологи решили присвоить имя Java 5.0, где цифра 5 взята из версии 1.5.

Итак, изначально платформа имела версии с 1.02 (первый официальный выпуск) по 1.1 и называлась просто Java. Версии 1.2, 1.3 и 1.4 принадлежат линейке Java 2. Начиная с версии 1.5, Java называется Java 5.0. Можно встретить названия Java 5 (без .0) и Tiger (это оригинальное кодовое имя). Теперь мы даже не имеем понятия, каких сюрпризов следует ожидать от будущей версии…

## Наточите свой карандаш Ответы

### Посмотрите, как легко писать код на Java.

***Не переживайте, если пока не понимаете ни одной строки этого кода!***

Все, что вы здесь видите, будет подробно описано в книге (в основном на первых 40 страницах). Если Java похож на язык, с которым вы работали до этого, то кое-что покажется вам простым. В противном случае не беспокойтесь — *мы все рассмотрим.*

| | |
|---|---|
| `int size = 27;` | Объявляем целочисленную переменную с именем size и присваиваем ей значение 27. |
| `String name = "Фидо";` | Объявляем строковую переменную с именем name и присваиваем ей значение «Фидо». |
| `Dog myDog = new Dog(name, size);` | Объявляем новую переменную типа Dog с именем myDog и передаем ей параметры name и size. |
| `x = size - 5;` | Вычитаем 5 из 27 (значение size) и присваиваем результат переменной x. |
| `if (x < 15) myDog.bark(8);` | Если x (со значением 22) меньше, чем 15, приказываем собаке пролаять 8 раз. |
| | |
| `while (x > 3) {` | Выполняем цикл, пока x больше 3. |
| `  myDog.play();` | Приказываем собаке играть (что бы это ни значило для нее). |
| `}` | Это похоже на конец цикла — все, что находится внутри скобок { }, выполняется в цикле. |
| | |
| `int[] numList = {2,4,6,8};` | Объявляем целочисленный массив numList и добавляем в него значения 2, 3, 6, 8. |
| `System.out.print("Привет");` | Выводим слово «Привет». Вероятно, в командной строке. |
| `System.out.print("Собака: " + name);` | Выводим «Привет, Фидо» (переменная name имеет значение «Фидо») в командной строке. |
| `String num = "8";` | Объявляем строковую переменную num и присваиваем ей значение «8». |
| `int z = Integer.parseInt(num);` | Преобразуем строку «8» в настоящее число 8. |
| | |
| `try {` | Пытаемся что-то выполнить... Возможно, это необязательно должно работать... |
| `  readTheFile("myFile.txt");` | Читаем текстовый файл с именем myFile.txt (или по крайней мере пытаемся это сделать). |
| `}` | Это похоже на завершение нашей попытки что-то сделать... |
| `catch(FileNotFoundException ex) {` | Здесь мы распознаем, работают ли операции, которые мы пытались выполнить. |
| `  System.out.print("Файл не найден.");` | Если операции завершились неудачей, выводим в командной строке «Файл не найден». |
| `}` | Похоже, что весь код внутри { } — это наши действия при сбое, который может случиться внутри оператора try. |

# Структура кода в Java

**Добавляем класс в исходный файл.**

**Добавляем метод в класс.**

**Добавляем выражения в метод.**

## Что творится внутри исходного файла

Файл с исходным кодом (с расширением *java*) содержит определение одного ***класса***. Класс — это *часть* вашей программы; совсем маленькие приложения могут обойтись единственным классом. Содержимое класса должно находиться внутри парных фигурных скобок.

```
public class Dog {



}
```
класс

## Что происходит внутри класса

Класс может иметь один или несколько ***методов***. Метод ***bark*** из класса Dog будет хранить инструкции о том, каким образом собака должна лаять. Методы должны быть объявлены *внутри* класса (иными словами, между его фигурными скобками).

```
public class Dog {
   void bark() {


   }
}
```
метод

## Что происходит внутри метода

Инструкции для метода должны быть размещены между его фигурными скобками. Если говорить упрощенно, *код* метода — это набор выражений. Думайте о методе как о функции или процедуре.

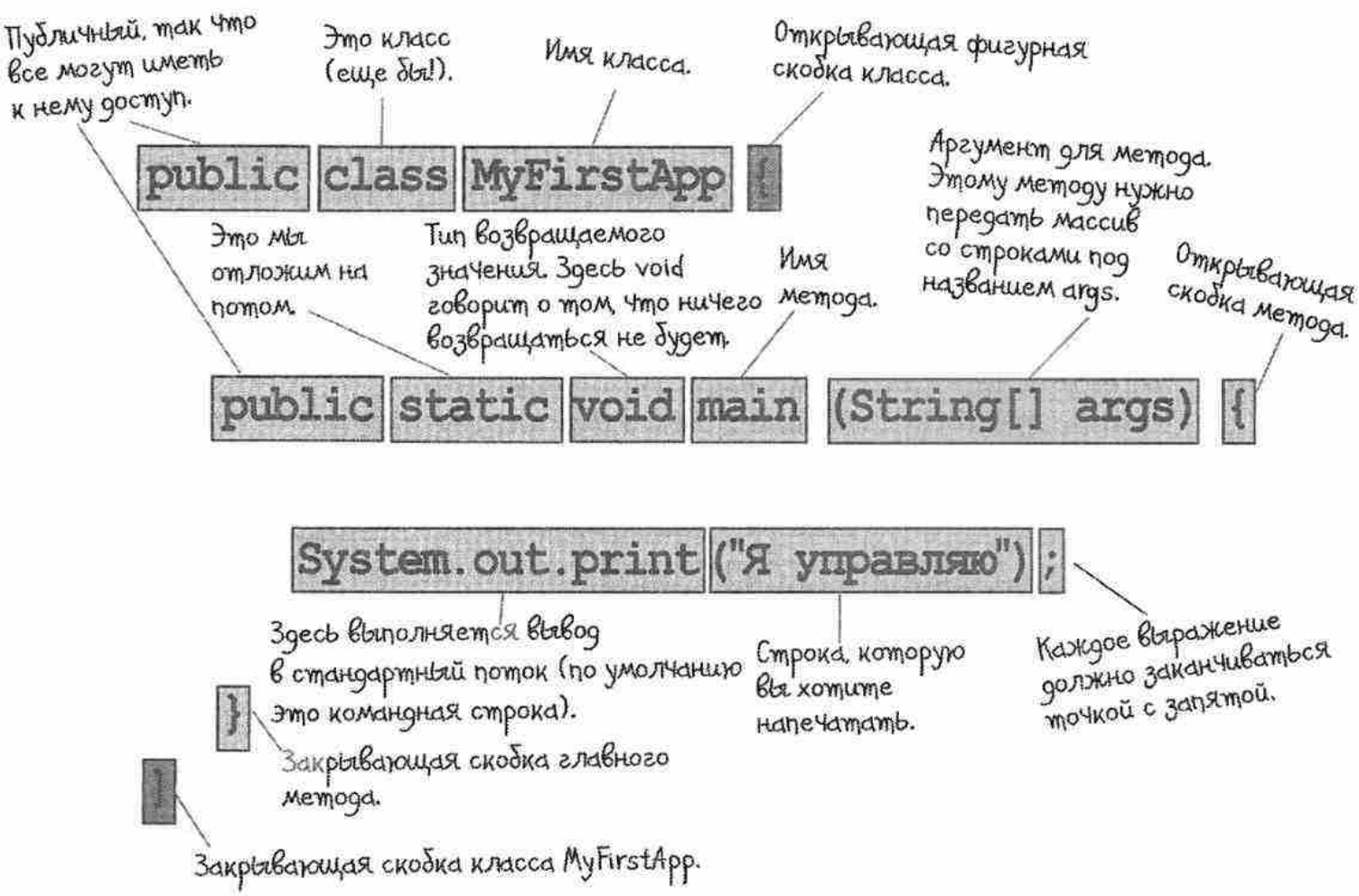
```
public class Dog {
   void bark() {
    statement1;
    statement2;
   }
}
```
выражения

# Структура класса

Когда JVM начинает свою работу, она ищет класс, который ей передали через командную строку. Затем она ищет метод, записанный особым образом, например так:

```
public static void main (String[] args) {
  // Здесь размещается ваш код
}
```

Далее JVM выполняет все, что находится между фигурными скобками { } главного метода. Любая программа на языке Java содержит по меньшей мере один **класс** и как минимум один метод **main** (один для всего *приложения*, а не для каждого *класса*).




Не пытайтесь запомнить все это прямо сейчас. В этой главе вы лишь делаете свои первые шаги.

# Создание класса с методом main

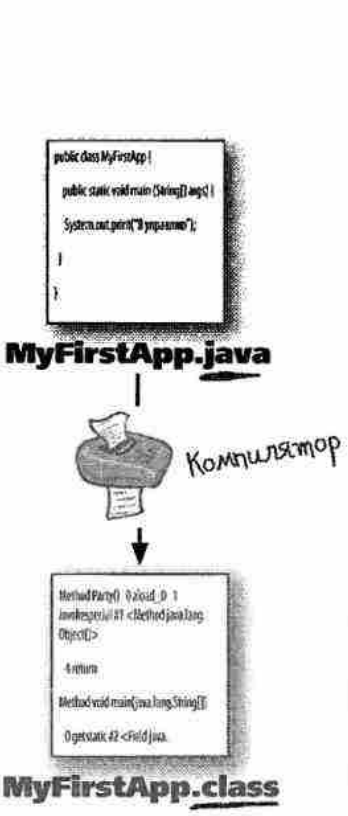
В языке Java весь код хранится в виде **классов**. Вы набираете свой исходный файл (с расширением *.java*) и компилируете его в новый файл с байт-кодом (с расширением *.class*). Запуская программу, вы на самом деле запускаете *класс*.

Запустить программу — значит сказать виртуальной машине Java (JVM): «Загрузи класс `Hello`, после чего запусти его метод `main()`. Продолжай выполнять, пока не закончится весь код в методе main».

Во второй главе мы подробнее рассмотрим все, что связано с классами, но сейчас вас должно интересовать следующее: ***как написать код на языке Java***, чтобы он запустился?

Все начинается с метода **main()**. Именно с него программа приступает к выполнению.

Неважно, насколько велика ваша программа (то есть неважно, сколько классов она содержит), всегда должен быть метод **main()**, который приведет в движение остальной код.




```
public class MyFirstApp {

   public static void main (String[] args) {
     System.out.println("Я управляю");
     System.out.println("миром");
   }

}
```

**1 Сохраняем**

`MyFirstApp.java`

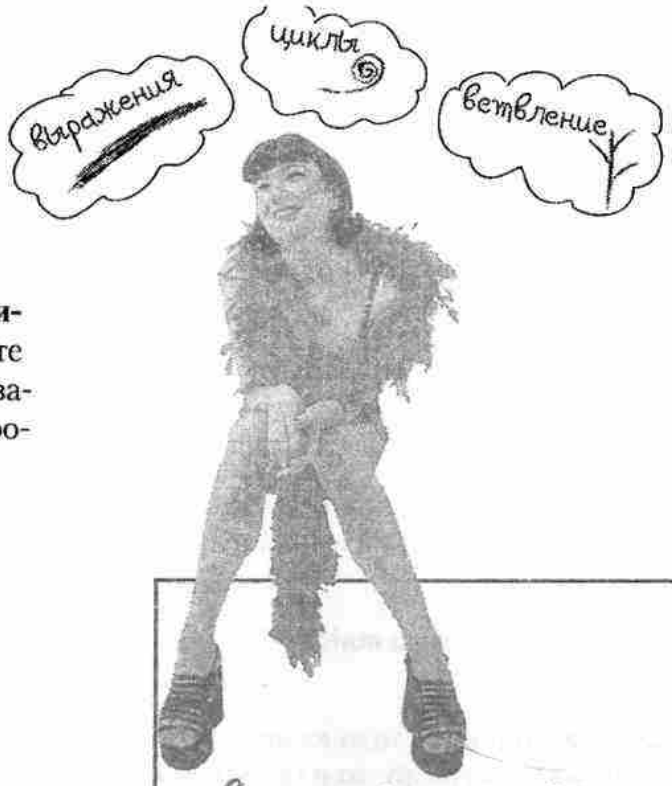
**2 Компилируем**

```
javac MyFirstApp.java
```

**3 Запускаем**

```
File Edit Window Help Scream
%java MyFirstApp
Я управляю
миром
```

# Что можно разместить внутри главного метода




Веселье начинается, как только вы переходите внутрь главного (или *любого* другого) метода. **Приказывая компьютеру выполнить что-либо,** можете пользоваться обычными выразительными средствами, знакомыми вам из других языков программирования.

В своем коде вы можете сказать JVM следующее.

## 1 Сделать что-то

**Выражения:** объявления, присваивания, вызовы методов и т. д.

```
int x = 3;
String name = "Кинжал";
x = x * 17;
System.out.print("x равен " + x);
double d = Math.random();
// Это комментарий
```

## 2 Делать что-то снова и снова

**Циклы:** *for* и *while*

```
while (x > 12) {
  x = x -1;
}
for (int x = 0; x < 10; x = x + 1) {
  System.out.print("Теперь x равен " + x);
}
```

## 3 Сделать что-то при условии

**Ветвление:** условия *if/else*.

```
if (x == 10) {
  System.out.print("x должен быть равен 10");
} else {
  System.out.print("x не равен 10");
}
if ((x < 3) & (name.equals("Кинжал"))) {
  System.out.println("Осторожно");
}
System.out.print("Эта строка выполняется
в любом случае");
```

## Синтаксические забавы

★ Каждое выражение должно заканчиваться точкой с запятой.

```
x = x + 1;
```

★ Однострочный комментарий начинается двумя слешами.

```
x = 22;
// Эта строка меня волнует
```

★ Большинство пробелов, табуляций, символов переноса строки и т. д. игнорируются.

```
x      =      3  ;
```

★ Переменные объявляются с помощью **имени** и **типа** (в главе 3 вы изучите все типы, доступные в языке Java).

```
int weight;
//Тип: int, имя: weight
```

★ Классы и методы должны объявляться внутри парных фигурных скобок.

```
public void go() {
   // Здесь будет
   // восхитительный код
}
```

# Зацикливаем, зацикливаем и...

В Java есть три стандартные конструкции для циклов: *while*, *do-while* и *for*. Далее мы рассмотрим каждую из них, но пока остановимся на *while*.

Синтаксис (не говоря уже о логике) этого оператора чрезвычайно прост. Пока некоторое условие верно, выполняется все, что находится внутри *блока*. Этот блок ограничен парными фигурными скобками, поэтому все, что нужно повторять, должно располагаться именно там.

Ключевое свойство цикла состоит в *проверке условия*. В Java проверка условия — это выражение, которое возвращает булево значение, то есть **true** либо **false**.

Если вы напишете нечто вроде «Пока *мороженое внутри упаковки равно true*, продолжай копать», то получите четкое булево условие. На вопрос, *находится* мороженое в упаковке *или нет*, может быть два однозначных ответа. Но если вы напишете «Пока *Боб* продолжает копать», то не получите настоящего условия. Чтобы выражение заработало, придется изменить его на что-то вроде «Пока у Боба насморк...» или «Пока Боб *не* завернулся в плед...».

## Простые логические проверки

Вы можете выполнять простые логические проверки, используя такие *операции сравнения*:

< (меньше чем);

> (больше чем);

== (равенство) (здесь *два* символа).

Обратите внимание на разницу между оператором *присваивания* (*одиночный* знак =) и оператором *сравнения* (*два* знака ==). Многие программисты случайно пишут = *вместо* == (но не вы).

```
int x = 4; // Присваиваем x значение 4
while (x > 3) {
  // Код в цикле будет работать, так как
  // x больше, чем 3
  x = x - 1;
  // или цикл будет выполняться вечно
}
int z = 27; //
while (z == 17) {
  // Код цикла не будет запущен, так как
  // z не равно 17
}
```

## Это не глупые вопросы

**В: Почему все нужно добавлять внутрь класса?**

О: Java — объектно ориентированный (ОО) язык. Это раньше были древние компиляторы, для которых вы писали монолитные исходные файлы с множеством процедур. В главе 2 вы узнаете, что класс — это шаблон для объекта и почти все в языке Java относится к объектам.

**В: Нужно ли добавлять метод main в каждый класс?**

О: Нет. Java-программа может иметь десятки (и даже сотни) классов, но вам нужен только один метод main — тот самый, с которого программа начинает работу. При этом у вас могут быть проверочные классы с методами main, предназначенные для тестирования других ваших классов.

**В: В языке программирования, которым я пользуюсь, разрешено делать логическую проверку с целочисленным значением. Можно ли в Java написать так:**

```
int x = 1;
while (x) { }
```

О: Нет. Типы **boolean** и integer в языке Java несовместимы между собой. Поскольку результатом проверки условия должно быть булево значение, единственная переменная, которую вы можете проверять напрямую (без использования оператора сравнения), будет иметь тип ***boolean***. Например, вы можете написать:

```
boolean isHot = true;
while(isHot) { }
```

## Пример цикла `while`

```
public class Loopy {
  public static void main (String[] args) {
  int x = 1;
  System.out.println("Перед началом цикла");
  while (x < 4) {
    System.out.println("Внутри цикла");
    System.out.println("Значение x равно " + x);
    x = x + 1;
  }
  System.out.println("После окончания цикла");
  }
}
```

Это результат работы цикла.

```
% java Loopy
Перед началом цикла
Внутри цикла
Значение x равно 1
Внутри цикла
Значение x равно 2
Внутри цикла
Значение x равно 3
После окончания цикла
```

### КЛЮЧЕВЫЕ МОМЕНТЫ

- Выражения заканчиваются точкой с запятой ;.
- Блоки кода задаются парными фигурными скобками { }.
- Целочисленная переменная объявляется с типом и именем: **int x;**.
- Оператор **присваивания** состоит из одинарного символа: =.
- В операторе сравнения используется два таких символа: ==.
- Цикл *while* выполняет все, что находится внутри его блока (заданного фигурными скобками), пока *проверка условия* возвращает значение **true**.
- Если проверка условия вернула **false**, то блок кода внутри цикла *while* не выполнится, а JVM проследует вниз по коду и начнет выполнять выражение, находящееся сразу *после* блока с циклом.
- Условие размещается внутри скобок:

```
while (x == 4) { }
```

# Условное ветвление

В языке Java проверка условия с оператором *if* ничем не отличается от логической проверки в цикле *while*, но вместо «***Пока*** у нас все еще есть пиво...» мы говорим «***Если*** у нас все еще есть пиво...»

```
class IfTest {
  public static void main (String[] args) {
  int x = 3;
  if (x == 3) {
    System.out.println("x должен равняться 3");
  }
  System.out.println("Эта строка выполняется
в любом случае");
  }
}
```

Результат работы программы

```
% java IfTest
x должен равняться 3
Эта строка выполняется в любом случае
```

Приведенный выше код напечатает «*x* должен равняться 3», только если соблюдено условие (*x* равен 3). Предложение «Эта строка выполняется в любом случае» будет выведено в любом случае. Таким образом, в зависимости от значения переменной *x* будет напечатано либо две строки, либо одна.

Но можно добавить к условию оператор *else* и получить возможность сказать нечто вроде: «*Если* у нас еще есть пиво, пишем код дальше, *иначе* (в ином случае) берем еще пива и только потом продолжаем...»

```
class IfTest2 {
  public static void main (String[] args) {
  int x = 2;
  if (x == 3) {
    System.out.println("x должен равняться 3");
  } else {
    System.out.println("x не равен 3");
  }
  System.out.println("Эта строка выполняется в любом
случае");
  }
}
```

Новый результат работы программы

```
% java IfTest2
x не равен 3
Эта строка выполняется в любом случае
```

## System.out.print против System.out.println

Если вы были внимательны (а мы в этом не сомневаемся), то должны были заметить, что мы начали использовать **println** вместо **print**.

**Увидели ли вы разницу?**

System.out.***println*** вставляет перенос строки (print***ln*** расшифровывается как **print*newline*** — «напечатать новую строку»), тогда как System.out.**print** продолжает выводить текст в той же строке. Если вы хотите, чтобы текст каждый раз печатался в новой строке, используйте **println**. Метод **print** подходит для тех случаев, когда текст нужно выводить в одну строку.

## Наточите свой карандаш

***Текст, который нужно вывести:***

```
% java DooBee
DooBeeDooBeeDo
```

***Впишите недостающий код:***

```
public class DooBee {
  public static void main (String[] args) {
    int x = 1;
    while (x < _____ ) {
      System.out._____("Doo");
      System.out._____("Bee");
      x = x + 1;
    }
    if (x == _____ ) {
      System.out.print("Do");
    }
  }
}
```

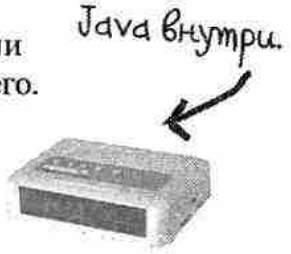
# Создание серьезного бизнес-приложения

Теперь попробуйте собрать воедино все приобретенные сведения и применить их на практике. Понадобится класс с методом *main()*, переменными типа *int* и *String*, циклом *while* и условным оператором *if*. Добавьте немного лоска — и бизнес-система будет готова. Но, *прежде* чем посмотреть на код, размещенный на этой странице, подумайте, как бы вы запрограммировали детскую песенку про 99 бутылок пива.

```
public class BeerSong {
   public static void main (String[] args) {
     int beerNum = 99;
     String word = "бутылок (бутылки)";

     while (beerNum > 0) {

       if (beerNum == 1) {
         word = "бутылка"; // В единственном числе - ОДНА бутылка.
       }

       System.out.println(beerNum + " " + word + " пива на стене");
      System.out.println(beerNum + " " + word + " пива.");
      System.out.println("Возьми одну.");
      System.out.println("Пусти по кругу.");
      beerNum = beerNum - 1;
      if (beerNum > 0) {
        System.out.println(beerNum + " " + word + " пива на стене");
      } else {
        System.out.println("Нет бутылок пива на стене");
      } // Конец else
    } // Конец цикла while
  } // Конец метода main
} // Конец класса
```

**В коде есть одна ошибка. Он скомпилируется и запустится, но результат будет не таким, как ожидается. Поищите эту ошибку и попробуйте ее исправить.**

## Утро понедельника у Боба




Будильник зазвонил в 8:30 утра, как и в любой другой день недели. Но у Боба были очень насыщенные выходные, поэтому он дотянулся до будильника и выключил его. С этого все и началось — ожила Java-машина.

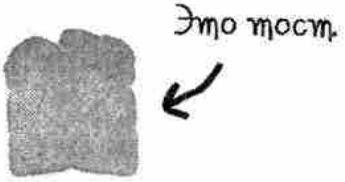
Сначала будильник отправил сообщение кофеварке*: «Эй, наш чокнутый опять уснул, отложи готовку кофе на 12 минут».

Кофеварка отправила сообщение тостеру: «Не спеши готовить тосты, Боб еще спит».

Затем будильник передал сообщение сотовому телефону Nokia: «Позвони Бобу в 9 часов и скажи ему, что мы слегка опаздываем».

И, наконец, будильник отправил сообщение беспроводному ошейнику Сэма (Сэм — собака) с помощью уже знакомого сигнала, который означает «Сходи за газетой, но не жди, что тебя будут выгуливать».

Пару минут спустя Боб снова выключил будильник и задремал. Наконец звонок раздался в третий раз. Но, как только Боб нажал спасительную кнопку, будильник отправил собачьему ошейнику сигнал: «Прыгай и лай». Взбудораженный и окончательно проснувшийся Боб поднялся с кровати. Благодаря изучению языка Java и удачной поездке на радиорынок привычный режим жизни заметно улучшился.




***Тост готов.***

***Кофе дымится.***

***Газета на столе.***

Еще одно чудесное утро в ***доме, которым управляет Java***.

**Вы тоже можете иметь такой дом.** Воспользуйтесь разумным решением на основе Java, Ethernet и технологии Jini. Остерегайтесь подделок, которые применяют так называемую технологию plug and play (что на самом деле означает «включи и мучайся с этим следующие три дня, пытаясь заставить работать») или «переносимые» платформы. Бетти, сестра Боба, как-то раз попробовала одну из них, и результат оказался, как бы помягче сказать, не совсем удовлетворительным и надежным. Хотя ее собака тоже отчасти виновата...

Правдива ли эта история? И да и нет. Хотя различные версии Java работают в таких устройствах, как PDA, мобильные телефоны (*особенно* мобильные телефоны), пейджеры, будильники, смарт-карты и т. д. — вы не встретите ее в тостерах или собачьих ошейниках. Но, даже если вы не находите модель своего любимого устройства с поддержкой Java, вы все еще можете управлять им через другой интерфейс (например, из ноутбука), на котором работает Java. Такой подход реализуется с помощью Jini *Surrogate Architecture* (сетевая архитектура для создания распределенных систем). Таким образом, вы можете воплотить в жизнь мечту об «умном» доме.

** IP multicast, если вас интересует конкретный протокол.*

Попробуйте мой новый генератор фраз, и вы начнете говорить так же красиво, как ваш начальник или парни из отдела продаж.

Хорошо, песенка про пиво — это не *совсем* серьезное бизнес-приложение. Вам все еще нужно что-то более практичное, что можно показать начальству? Тогда взгляните на код генератора фраз.

Примечание: набирая этот код в редакторе, делайте правильный перенос слов/строк! Никогда не нажимайте Enter при вводе строки (текста между кавычками), так как программа может не скомпилироваться. Переносы, которые вы видите на этой странице, настоящие, и вы можете смело их набирать, но не нажимайте Enter, пока не напечатаете текст полностью.

```
public class PhraseOMatic {
  public static void main (String[] args) {

1   // Создайте три набора слов для выбора. Добавляйте собственные слова!
    String[] wordListOne = {"круглосуточный", "трех-звенный",
"30000-футовый", "взаимный", "обоюдный выигрыш", "фронтэнд",
"на основе веб-технологий", "проникающий", "умный", "шесть
сигм", "метод критического пути", "динамичный"};

    String[] wordListTwo = {"уполномоченный", "трудный",
"чистый продукт", "ориентированный", "центральный",
"распределенный", "кластеризованный", "фирменный",
"нестандартный ум", "позиционированный", "сетевой",
"сфокусированный", "использованный с выгодой", "выровненный",
"нацеленный на", "общий", "совместный", "ускоренный"};

    String[] wordListThree = {"процесс", "пункт разгрузки",
"выход из положения", "тип структуры", "талант", "подход",
"уровень завоеванного внимания", "портал", "период времени",
"обзор", "образец", "пункт следования"};

2   // Вычисляем, сколько слов в каждом списке
    int oneLength = wordListOne.length;
    int twoLength = wordListTwo.length;
    int threeLength = wordListThree.length;

3   // Генерируем три случайных числа
    int rand1 = (int) (Math.random() * oneLength);
    int rand2 = (int) (Math.random() * twoLength);
    int rand3 = (int) (Math.random() * threeLength);

4   // Теперь строим фразу
    String phrase = wordListOne[rand1] + " " +
wordListTwo[rand2] + " " + wordListThree[rand3];

5   // Выводим фразу на экран
    System.out.println("Все, что нам нужно, — это " + phrase);
  }
}
```

# Генератор фраз

## Как он работает

Программа формирует три списка со словами, потом случайным образом выбирает по одному слову из каждого списка и выводит результат. Не переживайте, если вам не удалось понять, что происходит в каждой строке, ведь вы только начали читать книгу. Это лишь беглый взгляд с высоты 30 000 футов на нестандартный ум, нацеленный на образец, использованный с выгодой.

1. Сначала нужно создать три строковых массива — контейнера, которые будут хранить все слова. Объявлять и создавать массивы просто; вот небольшой пример:

```
String[] pets = {"Фидо", "Зевс", "Бин"};
```

Все слова заключены в кавычки (как и подобает приличным строкам) и разделены запятыми.

2. Все три списка (массива) требуются для того, чтобы выбирать из них случайные слова, поэтому нужно знать, сколько слов хранится в каждом списке. Если в списке содержится 14 слов, нам подойдет случайное число между 0 и 13 (нумерация массивов в Java начинается с нуля, поэтому первое слово расположено под индексом 0, второе — под индексом 1, а последнее — под индексом 13). Нам повезло, так как массив в Java всегда готов сообщить о своей длине. Нужно только спросить. Имея массив pets, мы напишем:

```
int x = pets.length
```

и переменная **x** примет значение 3.

**Все, что нам нужно, — это...**

проникающий нацеленный процесс

взаимный фирменный портал

уполномоченный трехзвенный подход

круглосуточный сетевой тип структуры

динамичный совместный обзор

умный ускоренный выход из положения

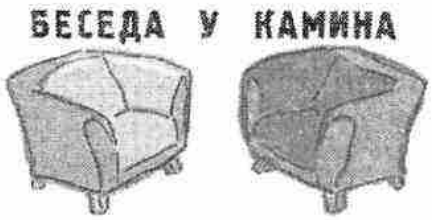
3. Нам нужны три случайных числа. Вместе с Java поставляется стандартный пакет с набором математических методов (пока вы можете думать о них как о функциях). Метод **random()** возвращает случайное число между 0 и 1 (не включительно), поэтому мы должны умножить это число на количество элементов (длину массива) в нашем списке. Нужно сделать так, чтобы результат был целочисленным (дробные значения недопустимы!); для этого мы используем приведение типов (подробно об этом процессе вы прочтете в главе 4). Это то же самое, что и преобразование числа с плавающей точкой в целое число:

```
int x = (int) 24.6;
```

4. Теперь нам нужно построить фразу, выбирая по слову из каждого списка и склеивая их вместе (не забывая вставлять между ними пробелы). Мы используем оператор +, который *конкатенирует* (или просто соединяет) объекты типа String. Чтобы получить элемент массива, мы даем ему индекс (позицию) той сущности, которую хотим использовать:

```
String s = pets[0]; // s — теперь строка со значением "Фидо"
s = s + " " + "— собака"; // s теперь имеет значение "Фидо — собака"
```

5. Наконец мы выводим фразу в командной строке и... вуаля! *Теперь мы в рекламном бизнесе.*




Сегодня в эфире: **Компилятор и JVM выясняют, кто из них важнее**

**Виртуальная машина Java**

Привет, я Java! Я тот, кто на самом деле выполняет программу. Компилятор просто выдает вам *файл*. Вы можете распечатать его и наклеить на стену, сжечь, завернуть в него рыбу и т. д. Но файл абсолютно *бесполезен*, пока я его не выполню.

Вот еще один факт: у компилятора нет чувства юмора. Конечно, если вам приходится проводить дни напролет, выискивая мелкие синтаксические погрешности...

Я не говорю, что вы абсолютно бесполезны. Но чем же вы на самом деле занимаетесь? Серьезно. Я даже не представляю. Программист может собственноручно написать байт-код, и я его приму. Вы, должно быть, скоро лишитесь своей работы, дружище.

К слову, об отсутствии чувства юмора. Вы еще не ответили на мой вопрос: чем вы занимаетесь на самом деле?

**Компилятор**

Мне неприятен ваш тон.

Извините, но если бы не было меня, что бы вы запускали? К вашему сведению, Java спроектирован для выполнения байт-кода не просто так. Если бы Java был интерпретатором и транслировал исходный код на лету, то программы, написанные на этом языке, работали бы с удручающе низкой скоростью. Разработчики Java потратили достаточно времени, чтобы убедить людей в том, что скорости и мощи этого языка хватает для большинства задач.

Извините, но это абсолютно невежественный взгляд на вещи (не говоря уже о вашем высокомерии). Если бы это было правдой (теоретически), то вы смогли бы выполнять любой правильно сформированный байт-код, даже если бы он не был продуктом компилятора. На практике это полнейший абсурд. Писать байт-код вручную — все равно что создавать офисные документы, напрямую используя язык PostScript. И я буду благодарен, если вы перестанете называть меня «дружище».

**Виртуальная машина Java**

Тем не менее некоторые из них до меня доходят! Я могу выбросить исключение ClassCastException и иногда замечаю, как люди пытаются добавить объект одного типа в массив, предназначенный для хранения совсем другого, и...

Ладно, это понятно. Но что насчет безопасности? Это же моя прерогатива! А что делаете вы — смотрите, правильно ли расставлены точки с запятыми? Ух, какая серьезная опасность! Как хорошо, что у нас есть вы!

Как скажете. Но мне приходится делать то же самое, просто чтобы убедиться, что никто не схитрил и не изменил байт-код прямо перед запуском.

О, можете на меня рассчитывать, ***дружище***.

**Компилятор**

Не забывайте, что Java — строго типизированный язык. Это означает, что я не могу позволить переменным хранить данные не того типа. Это чрезвычайно важная особенность для обеспечения безопасности, и я могу отловить большую часть ошибок, прежде чем код дойдет до вас. И я также...

Извините, но я не закончил. Да, бывают исключительные ситуации, связанные с типами и возникающие при выполнении программы. Но некоторые из них должны быть разрешены, чтобы обеспечить поддержку другой важнейшей особенности Java — динамического связывания. Во время своей работы программа может включать в себя новые объекты, о которых ее разработчик даже не знает, поэтому я должен обеспечивать некую гибкость. Однако моя работа заключается в том, чтобы не пропустить код, который никогда не сможет успешно выполниться. Как правило, я точно могу сказать, когда какой-то участок кода не будет работать. Например, если программист по ошибке попытается использовать объект Button в качестве сокета для сетевого соединения, я это увижу и уберегу его программу от неправильной работы.

Извините, но я, как говорится, нахожусь на передовой линии обороны. Описанные ранее ошибки, связанные с типами, могут привести к разрушительным последствиям, если их не отловить. Я также предотвращаю ошибки доступа. Например, код пытается выполнить приватный метод или изменить такой метод, который (по соображениям безопасности) никогда не должен изменяться. Я не позволяю людям получить доступ к коду, который они не должны видеть, и пресекаю попытки извлечь критически важные данные из другого класса. Можно потратить часы и, возможно, даже дни, чтобы описать важность моей работы.

Конечно, но, как я уже говорил, если бы мне не удавалось отловить примерно 99 % всех проблем, вы бы далеко не уехали. И, похоже, что мы уже слишком долго болтаем. Предлагаю вернуться к этой теме в следующем эфире.

# Упражнение Магнитики с кодом

Части рабочего Java-приложения разбросаны по всему холодильнику. Можете ли вы сгруппировать фрагменты кода, чтобы итоговая программа выводила текст, приведенный ниже? Некоторые фигурные скобки упали на пол; они настолько маленькие, что их нельзя поднять. Можете добавлять столько скобок, сколько понадобится.

```
if (x == 1) {
  System.out.print("d");
  x = x - 1;
}
```

```
if (x == 2) {
  System.out.print("b c");
}
```

```
class Shuffle1 {
  public static void main(String [] args) {
```

```
if (x > 2) {
  System.out.print("a");
}
```

```
int x = 3;
```

```
x = x - 1;
System.out.print("-");
```

```
while (x > 0) {
```

Результат:

```
File Edit Window Help Sleep
% java Shuffle1
a-b c-d
```

Упражнение

# Поработайте компилятором

Каждый из Java-файлов на этой странице — это полноценный исходник. Ваша задача — притвориться компилятором и определить, все ли из них скомпилируются. Если компиляция не сможет пройти успешно, как вы это исправите?

**A**

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      if ( x > 3) {
        System.out.println("большой икс");
      }
    }
  }
}
```

**B**

```
public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("маленький икс");
      }
    }
}
```

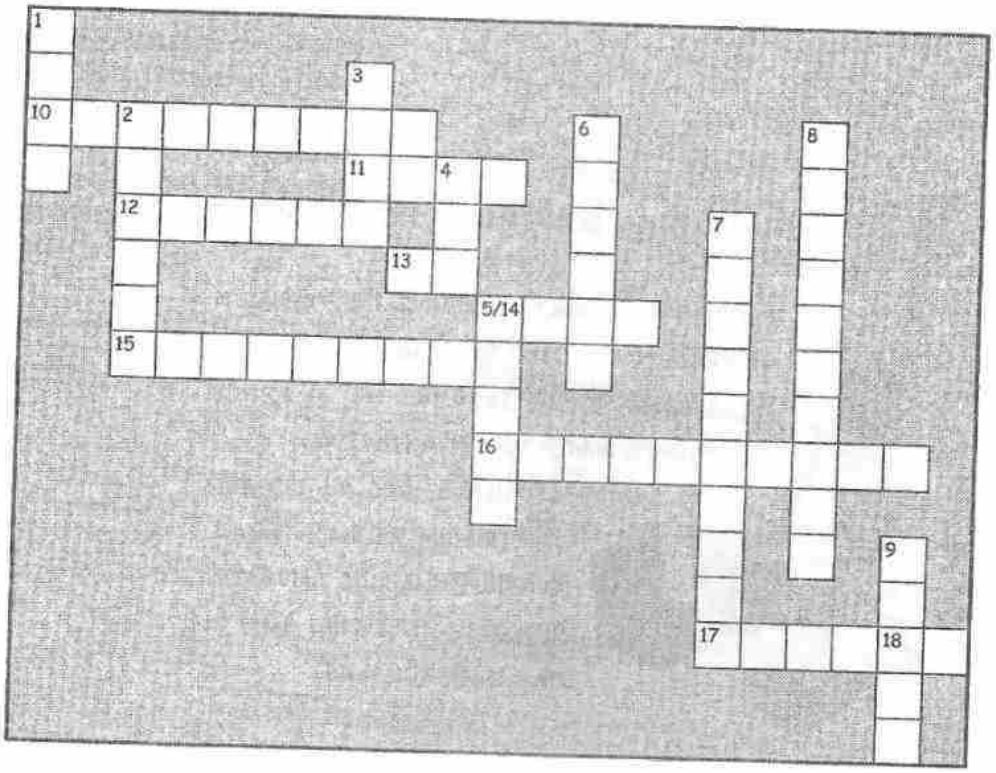
**C**

```
class Exercise1b {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("маленький икс");
      }
    }
}
```

## Кроссворд

# JavaCross 7.0

Немного разомнем мозги. Это обычный кроссворд, но почти все ответы для него взяты из этой главы. И чтобы вы не теряли бдительность, мы также добавили несколько слов, не связанных с Java, которые относятся к миру высоких технологий.

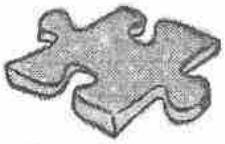



### По вертикали

1. Опять повторить?
2. Хранитель «сущностей».
3. Запускается из командной строки.
4. Числовой тип переменной.
5. Улаживает проблемы.
6. Загадочный модификатор.
7. Потребитель исходного кода.
8. Ее нельзя задать раз и навсегда.
9. Пока положение не улучшится.

### По горизонтали

10. Какой бывает строка?
11. Возвращается с пустыми руками.
12. Набор символов.
13. Отдел сетевых администраторов.
14. Вы просто должны иметь один такой метод.
15. Невозможность выбрать два пути сразу.
16. Создание нового класса или метода.
17. Проходной двор.
18. Аббревиатура для чипа.



## Смешанные сообщения

Ниже приведен код небольшой программы на языке Java. Но один ее блок пропал. Ваша задача — **найти блоки** (слева), которые выведут соответствующий результат, если их вставить в код. Не все строки с результатами будут использованы, а некоторые из них будут задействованы несколько раз. Соедините линиями блоки кода и подходящие для них результаты (ответы можно найти в конце главы).

```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while ( x < 5 ) {
      [          ]
      System.out.print(x + "" + y +" ");
      x = x + 1;
    }
  }
}
```

Сюда нужно вставлять возможные варианты.

**Возможные блоки кода:**

Соедините каждый блок с одним из возможных результатов.

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
  y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
  x = x + 1;
  if ( y < 3 ) {
    x = x - 1;
  }
}
y = y + 2;
```

**Возможный программный вывод:**

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

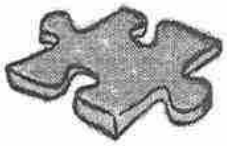
```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```



# Головоломка у бассейна

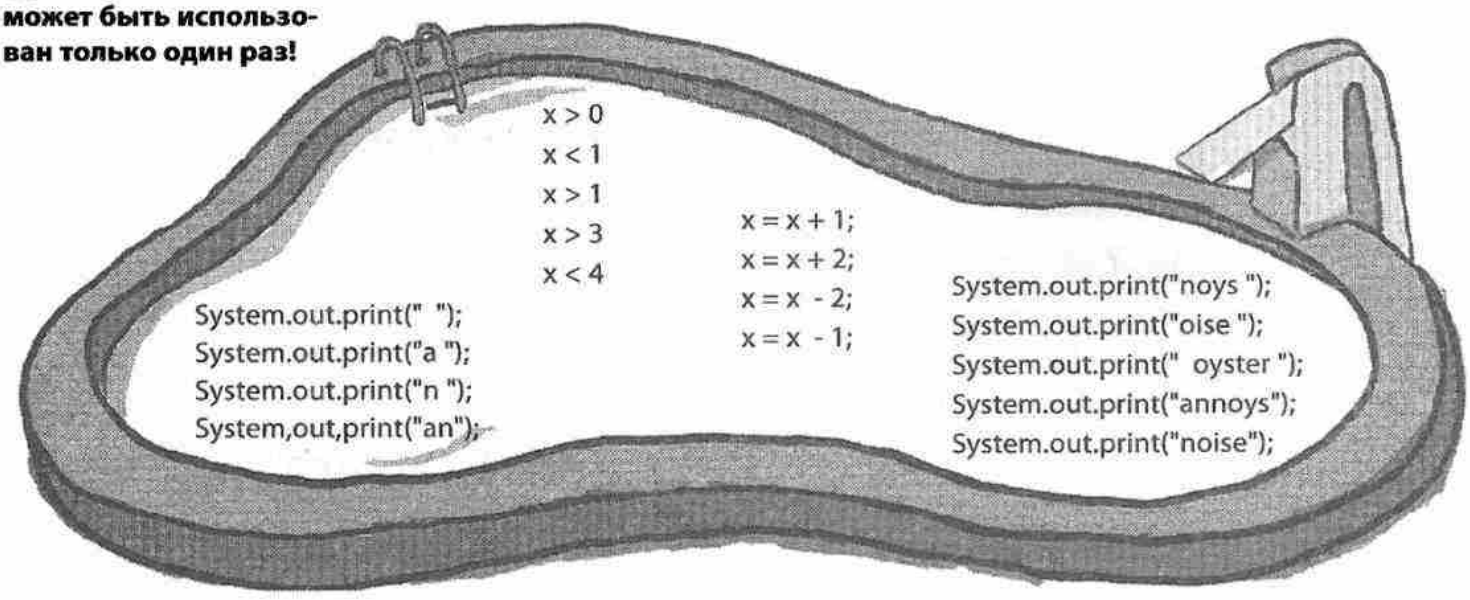
Ваша **задача** — взять фрагменты кода со дна бассейна и заменить ими пропущенные участки программы. **Нельзя** использовать один фрагмент несколько раз, и не все из них вам пригодятся. **Цель** — создать класс, который скомпилируется, запустится и выведет приведенный ниже текст. Не обольщайтесь — это сложнее, чем вы думаете.

**Результат:**

```
File Edit Window Help Cheat
%java PoolPuzzleOne
a noise
annoys
an oyster
```

```
class PoolPuzzleOne {
  public static void main(String [] args) {
    int x = 0;

    while ( _____ ) {

      _____
      if ( x < 1 ) {
        _____
      }
      _____

      if ( _____ ) {

        _____

        _____
      }
      if ( x == 1 ) {

        _____
      }
      if ( _____ ) {

        _____
      }
      System.out.println("");

      _____
    }
  }
}
```

**Примечание: каждый фрагмент из бассейна может быть использован только один раз!**



# Ответы

## Магнитики с кодом

```
class Shuffle1 {
  public static void main(String [] args)
{

    int x = 3;
    while (x > 0) {

      if (x > 2) {
        System.out.print("a");
      }

      x = x - 1;
      System.out.print("-");

      if (x == 2) {
        System.out.print("b c");
      }

      if (x == 1) {
        System.out.print("d");
        x = x - 1;
      }
    }
  }
}
```

```
File Edit Window Help Poet
% java Shuffle1
a-b c-d
```

## Поработайте компилятором

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      x = x + 1;
      if ( x > 3) {
        System.out.println("большой икс");
      }
    }
  }
}
```

**A**

Этот код скомпилируется и запустится (без вывода в командной строке), но если не изменить его, он будет работать вечно благодаря бесконечному циклу while.

---

```
class Foo {
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
  }
}
```

**B**

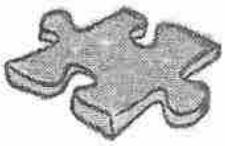
Этот файл не скомпилируется без **объявления класса. И не забывайте** о соответствующих фигурных скобках!

---

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
  }
}
```

**C**

Код из цикла while должен размещаться внутри метода. Он не может находиться в самом классе.

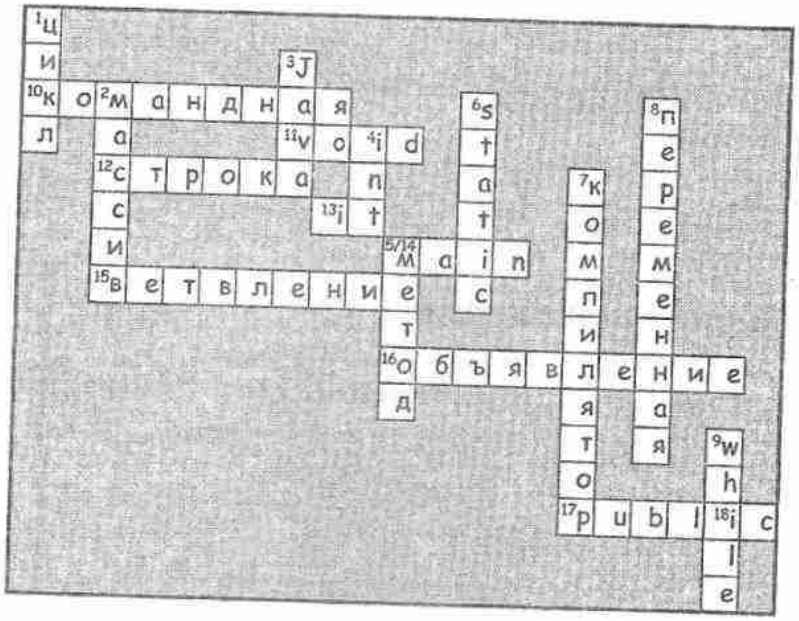


## Головоломка у бассейна

```
class PoolPuzzleOne {
  public static void main(String [] args) {
    int x = 0;

    while ( X < 4 ) {

      System.out.print("a");
      if ( x < 1 ) {
        System.out.print(" ");
      }
      System.out.print("n"

      if ( X > 1 ) {

        System.out.print(" oyster");
        x = x + 2;
      }
      if ( x == 1 ) {

        System.out.print("noys");
      }
      if ( X < 1 ) {

        System.out.print("oise");
      }
      System.out.println("");

      X = X + 1;
    }
  }
}
```

```
File Edit Window Help Cheat
%java PoolPuzzleOne
a noise
annoys
an oyster
```

## JavaCross 7.0



## Смешанные сообщения

```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while ( x < 5 ) {
      [                                    ]

      System.out.print(x + "" + y +" ");
      x = x + 1;
    }
  }
}
```

**Возможные блоки кода:**

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
  y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
  x = x + 1;
  if ( y < 3 ) {
    x = x - 1;
  }
}
y = y + 2;
```

**Возможный программный вывод:**

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

# Путешествие в Объектвилль

**Мне говорили, что там будут объекты.** В главе 1 вы размещали весь свой код в методе main(), но это не совсем объектно ориентированный подход. По сути, он *не имеет ничего общего* с объектами. Да, вы использовали несколько объектов, например строковые массивы для генератора фраз, но не создавали свои собственные типы объектов. Теперь вы оставите позади мир процедур, выберетесь из тесного метода main() и сами начнете разрабатывать объекты. Вы узнаете, чем же так удобно объектно ориентированное программирование (ООП) на языке Java, и почувствуете разницу между *классом* и *объектом*. Кроме того, вы убедитесь, что объекты могут сделать вашу жизнь лучше (по крайней мере ее профессиональную часть; привить вам чувство стиля мы не можем). Осторожно: попав однажды в Объектвилль, вы уже никогда не сможете вернуться обратно. Пришлите нам открытку!

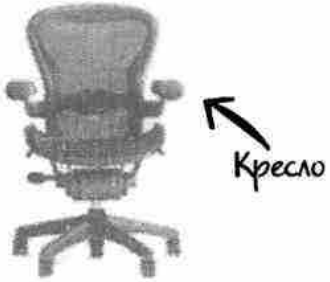
# Война за кресло,

## или Как объекты могут изменить вашу жизнь

Однажды в магазине программного обеспечения двум разработчикам вручили одно и то же задание. Надоедливый управляющий проектами начал подгонять программистов, обещая выдать крутое кресло Aeron (одно из тех, на которых сидят все парни из Кремниевой долины) тому, кто первым закончит работу. Ларри, занимающийся процедурным программированием, и Брэд, отдающий предпочтение ООП, с энтузиазмом восприняли перспективу завладеть столь лакомым кусочком.

Ларри, сидя за компьютером, спросил себя: «Что именно должна *делать* эта программа? Какие **процедуры** мне понадобятся?» И сам тут же ответил: «***rotate*** и ***playSound***». Таким образом, он принялся создавать процедуры. В конце концов, что это за программа, если в ней нет процедур?

Тем временем, сидя в кафе, Брэд лениво зевнул и спросил себя: «**Из чего** состоит эта программа? Какие ее ключевые *компоненты*?» Сначала он подумал о **фигурах**. Конечно, он помнил и о других объектах, таких как пользователь, звук и событие нажатия. Но у него уже была библиотека для этих частей программы, поэтому он сосредоточился на создании фигур. Посмотрите, как Брэд и Ларри реализовали свои программы, и попытайтесь догадаться, ***кто же выиграл Aeron***.



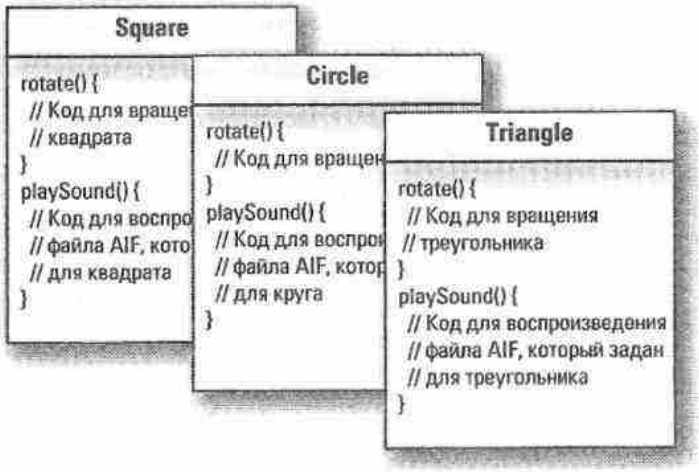

## На рабочем месте Ларри

Как и много раз до этого, Ларри сосредоточился на написании своих важных процедур. В мгновение ока он создал две **процедуры**: **rotate** и **playSound**.

```
   rotate(shapeNum) {
     // Поворачиваем фигуру на 360°
}
playSound(shapeNum) {
   // Используем переменную shapeNum
   // для определения того, какой звуковой файл
   // нужно воспроизвести, и воспроизводим его
}
```

## В кафе, где находится Брэд с ноутбуком

Брэд написал ***классы*** для каждой из трех фигур:



## Ларри уже считал себя победителем и почти ощущал стальное основание нового кресла...

### Постойте! Задание изменилось.

«Ладно, Ларри, *формально* ты был первым, — сказал управляющий, — но необходимо добавить в программу небольшую деталь. Для таких первоклассных программистов, как вы оба, это не составит труда».

«Ох, если бы мне давали монетку каждый раз, когда я слышу нечто подобное, — подумал Ларри, прекрасно понимая, что изменение в задании, не вызывающее никаких проблем, — это фантастика. — Да и Брэд выглядит каким-то подозрительно спокойным. К чему бы это?»

Ларри все еще верил, что объектно ориентированный подход хоть и интересный, но слишком громоздкий. В этом он был абсолютно убежден и не собирался менять свою позицию.

Добавлено к заданию.

## Ларри вернулся на рабочее место

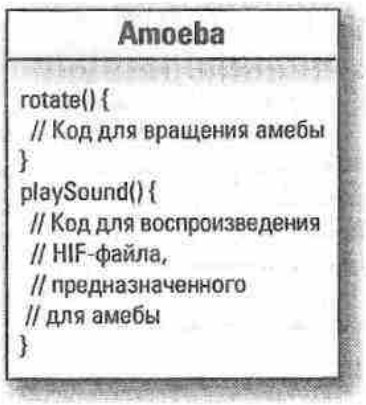
Процедура для вращения фигур по-прежнему должна работать; в коде используется таблица для нахождения графической фигуры, на которую указывает переменная shapeNum. Но процедуру ***playSound придется изменить***. И что это за файл в формате HIF?

```
playSound(shapeNum) {
  // если фигура – не амеба,
    // используем shapeNum для определения,
    // какой AIF-файл нужно проигрывать,
    // и воспроизводим его
  // иначе
    // воспроизводим файл HIF для амебы
}
```

Повернуть фигуру не сложно, но ***из-за необходимости трогать ранее проверенный код*** Ларри становится плохо. Уж он-то должен был знать, что, несмотря на заверения управляющего проектами, ***задание всегда меняется***.

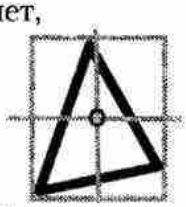
## А в это время Брэд на пляже...

Брэд улыбнулся, хлебнул немного коктейля и *написал новый класс*. В такие моменты в ООП ему больше всего нравилось то обстоятельство, что не нужно трогать код, который уже был протестирован. «Гибкость, расширяемость», — приговаривал он, размышляя о достоинствах ООП.



## Ларри успел опередить Брэда лишь на секунду.

«Ха-ха! Куда там нелепому ООП». Но улыбка на лице Ларри быстро растаяла, как только надоедливый управляющий проектами сказал (с нотками разочарования в голосе): «О нет, амеба должна вращаться совсем не так...»

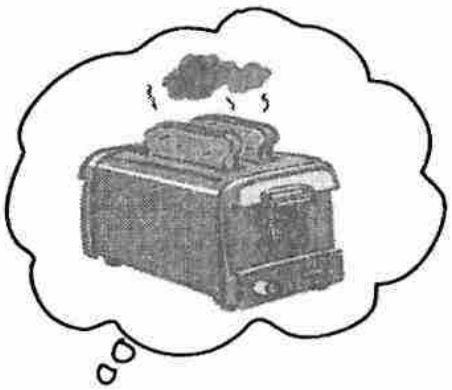


Чуть ранее оба программиста использовали следующий алгоритм для вращения.

1. **Определить прямоугольник, который вмещает в себя фигуру.**
2. **Вычислить центр прямоугольника и повернуть фигуру вокруг этой точки.**

Но амебоподобная фигура должна вращаться вокруг точки на ее конце, как стрелка в часах.

«Все, я готов, — подумал Ларри, представляя себя в виде обуглившейся гренки. — Хотя я могу добавить еще одно условие if/else в процедуру для вращения и затем написать код специально для амебы. Думаю, из-за этого ничего не сломается». Но тоненький голосок где-то у него внутри прошептал: «Наивный. Неужели ты действительно думаешь, что задание больше не будет меняться?»



Об этом в задании деликатно умалчивалось.

## Вернувшись на рабочее место, Ларри...

...решил, что лучше добавить аргумент для точки вращения в соответствующую процедуру. ***Придется изменить много кода.*** Тестирование, перекомпиляция — все это придется делать заново. Части кода, которые работали ранее, больше не работают.

```
rotate(shapeNum, xPt, yPt) {
// Если фигура – не амеба,
   // вычисляем центральную точку,
   // основываясь на прямоугольнике,
   // затем вращаем
// иначе
   // используем xPt и yPt
   // как сдвиг для точки вращения
   // и затем вращаем
}
```

## Тем временем Брэд, лежа с ноутбуком на раскладушке на фестивале народной музыки в Теллерайде...

...ни капли не сомневаясь, отредактировал метод rotate, но только в классе Amoeba. Он никогда не трогал проверенный, работающий и скомпилированный код, который относится к другим частям программы. Чтобы задать точку вращения для амебы, Брэд добавил атрибут, который должен быть обязательным для всех амеб. Он изменил, проверил и отправил по Wi-Fi итоговую программу быстрее, чем успела доиграть одна-единственная композиция.

## Итак, кресло получил Брэд, отдающий предпочтение ООП, правильно?

***Не спешите.*** Ларри нашел недостаток в коде Брэда. И, поскольку он был уверен, что благодаря новому креслу сможет произвести впечатление на симпатичную бухгалтершу, то решил указать на этот недостаток.

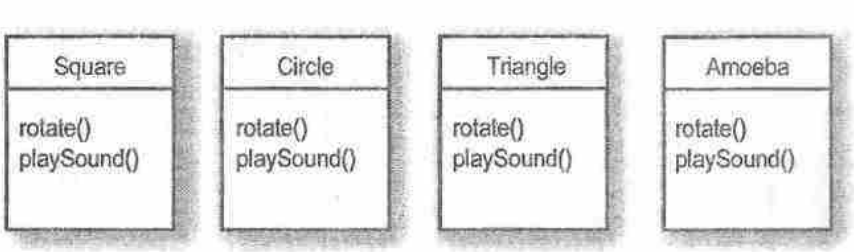
**Ларри:** Ты дублируешь код! Процедура rotate находится во всех четырех штуковинах.

**Брэд:** Это ***метод***, а не *процедура*. И он находится в *классах*, а не «штуковинах».

**Ларри:** Какая разница? Это глупый подход. Ты должен поддерживать *четыре* разных метода rotate. Что в этом может быть хорошего?

**Брэд:** Похоже, ты не видел конечный результат. Давай я покажу тебе, как в ООП работает **наследование**.

Благодаря креслу Ларри надеялся произвести на нее впечатление

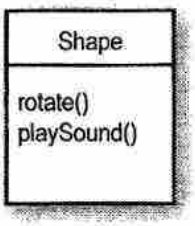



1

Я выяснил, что общего между всеми этими классами.

2

Это фигуры, которые вращаются и воспроизводят звуки. Я абстрагировал их общие черты и поместил в новый класс с именем Shape.

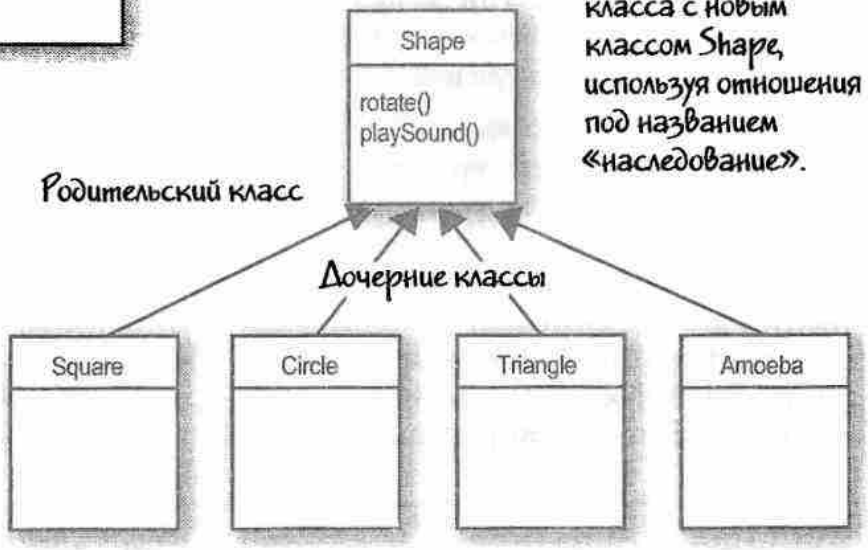



3

Затем я связал остальные четыре класса с новым классом Shape, используя отношения под названием «наследование».

Можете читать это как **«квадрат унаследован от фигуры»**, **«круг унаследован от фигуры»** и т. д. Я убрал методы rotate() и playSound() из других фигур, поэтому теперь нужно поддерживать всего один экземпляр каждого из них.

Класс Shape **родительский** для остальных четырех классов, которые, в свою очередь, являются дочерними (**потомками**, или **подклассами**) для Shape. Дочерние классы наследуют методы родительского класса. Иными словами, *если класс Shape обладает какой-то функциональностью, то его подклассы получают те же возможности.*



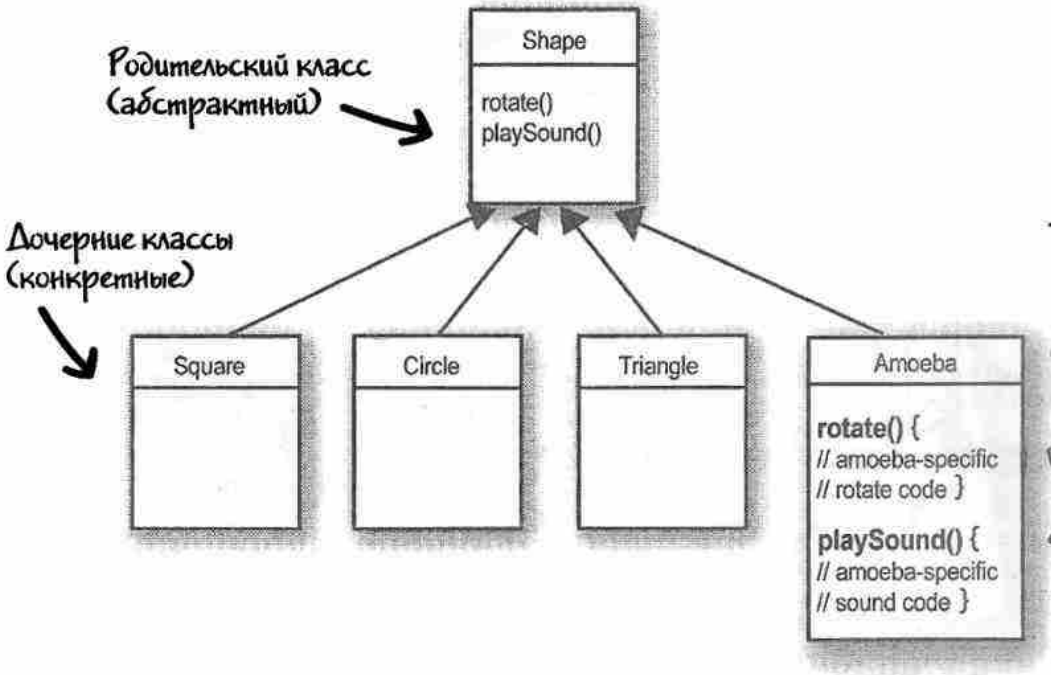
## А как быть с методом rotate для фигуры Amoeba?

**Ларри:** Не кроется ли вся проблема в том, что амеба имеет совершенно другие процедуры rotate и playSound?

**Брэд: Методы!**

**Ларри:** Не имеет значения. Как же амеба сможет делать что-то по-своему, если она наследует свою функциональность от класса Shape?

**Брэд:** Класс Amoeba **переопределяет** методы класса Shape. При выполнении программы JVM будет точно знать, какой именно метод rotate нужно вызвать, когда придется вращать амебу.



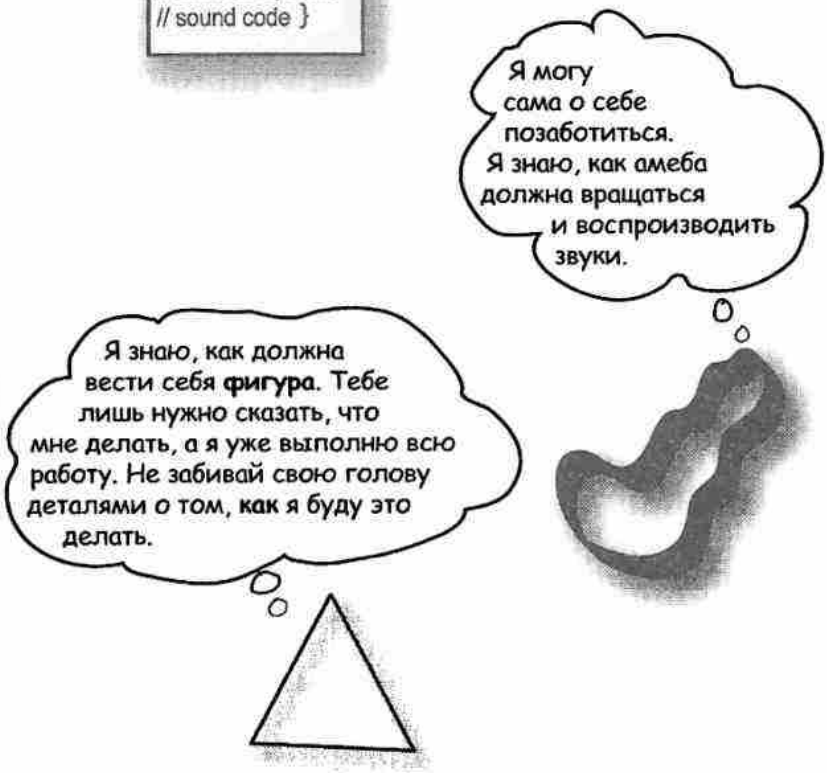

4

Я сделал так, чтобы класс Amoeba переопределил методы rotate() и playSound(), принадлежащие родительскому классу Shape.

Переопределение означает, что подкласс заново описывает унаследованные им методы, если ему нужно изменить или расширить их поведение.

**Ларри:** Как ты указываешь амебе сделать что-нибудь? Вызываешь ли ты при этом процедуру, прости, — *метод*, и говоришь ему, какую именно фигуру нужно вращать?

**Брэд:** Именно за это я люблю ООП. Когда нужно повернуть, например, треугольник, в коде вызывается метод rotate() *из объекта Triangle*. Остальная часть программы даже не знает о том, *как* треугольник это делает (и ей все равно). Если же нужно добавить в программу что-то новое, ты просто создаешь новый класс для **нового типа объектов, которые будут вести себя по-своему**.



## Я умираю от любопытства! Кто получил кресло?

**Эми со второго этажа.**

Держа все в тайне, управляющий проектами дал задание *трем* программистам.

### Что вам нравится в ООП?

«Оно помогает мне разрабатывать программы более естественным путем. Предметы имеют возможность развиваться».

Джой, 27 лет, архитектор программного обеспечения

«Если нужно добавить новую функциональность, мне не приходится возвращаться к коду, который я уже протестировал».

Брэд, 32 года, программист

«Мне нравится, что данные и методы, которые этими данными оперируют, находятся в одном классе».

Джош, 22 года, любитель пива

«Привлекает возможность повторного использования кода в других приложениях. Создавая новый класс, я могу сделать его достаточно гибким, чтобы потом применять в своих проектах».

Крис, 39 лет, руководитель проектов

«Не могу поверить, что Крис мог сказать такое. За последние 5 лет он не написал ни строчки кода».

Дэрил, 44 года, работает на Криса

«Помимо кресла?»

Эми, 34 года, программист

## Сила мозга

**Пора размять извилины**

Вы только что прочли историю о том, как поклонник процедур встретился лицом к лицу с программистом, предпочитающим ООП. Вы получили общее представление о таких объектно ориентированных концепциях, как классы, методы и атрибуты. Оставшуюся часть главы мы потратим на изучение классов и объектов (а в следующих главах вернемся к наследованию и переопределению).

Исходя из того, что вы только что увидели (а также узнали из других объектно ориентированных языков, с которыми работали раньше), уделите несколько минут следующим вопросам.

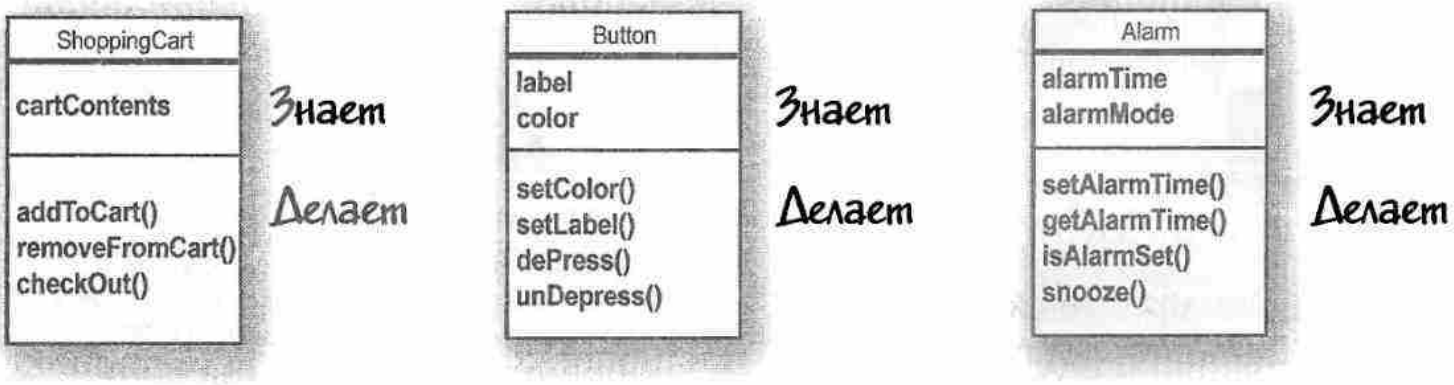
О каких фундаментальных вещах вы обязаны помнить, проектируя класс на языке Java? О чем вы должны себя спросить? Что бы вы внесли в список, который должен использоваться при проектировании класса?

### Общепознавательный совет

Если при решении упражнения вы зашли в тупик, попробуйте поговорить о нем вслух. Во время разговора активизируются различные участки мозга. Хотя лучше всего это работает при общении с другим человеком, домашние животные тоже подойдут. Именно так наша собака изучила полиморфизм.

## Проектируя класс, думайте об объектах, которые будут созданы на его основе. Думайте:

- о вещах, которые объект **знает;**
- вещах, которые объект **делает.**




То, что объект о себе **знает**, называется

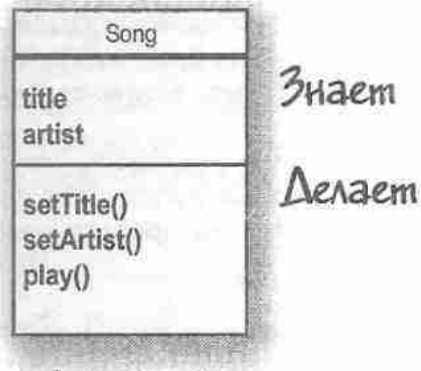
- **переменной экземпляра.**

То, что объект может **делать**, называется

- **методом.**

Переменные экземпляра (состояние)

Методы (поведение)




Вещи, которые объект о себе *знает*, называются **переменными экземпляра**. Они отражают состояние объекта (данные) и могут иметь уникальные значения для каждого объекта выбранного типа.

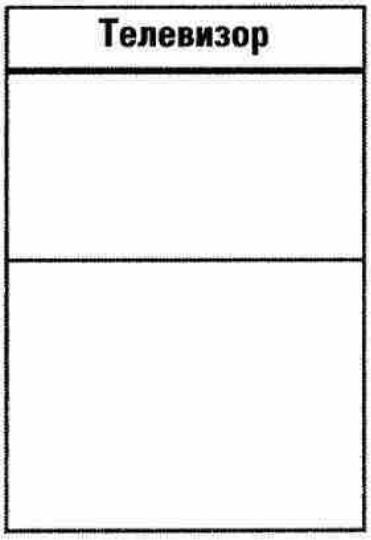
Думайте об **экземпляре** класса как о синониме слова **объект**.

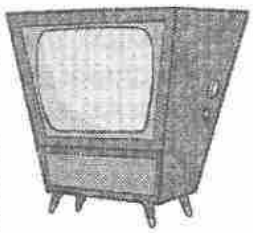
**Методы** — это действия, которые объект способен ***выполнять***. Проектируя класс, нужно думать о данных, которые должен знать объект. Кроме того, необходимо создавать методы, которые будут работать с этими данными. Часто объекты содержат методы для чтения и записи значений, относящихся к переменным экземпляра. Например, объект Alarm включает в себя переменную экземпляра для хранения времени срабатывания (alarmTime) и два метода для чтения/записи значений этой переменной.

Итак, у объектов есть переменные экземпляра и методы, но эти сущности создаются как часть класса.

### Наточите свой карандаш

Напишите, о чем должен знать и что должен уметь объект Television.



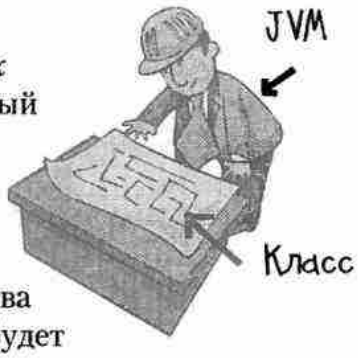
Переменные экземпляра

Методы

# В чем разница между классом и объектом?

## Класс — это не объект.

**Класс — это *шаблон* для объекта.** Он говорит виртуальной машине, *как* сделать объект заданного типа. Каждый объект, созданный из этого класса, может иметь собственные значения для переменных экземпляра класса. Например, вы можете использовать класс Button для получения множества различных кнопок, и каждая из них будет иметь свои цвет, размер, форму, метку и т. д.




## Смотрите на это так:

**Объект — это что-то вроде одной записи в телефонной книге.**

Еще одной аналогией для объектов может служить набор чистых карточек Rolodex. Они имеют одни и те же пустые поля (как и переменные экземпляра у объекта). Когда вы заполняете карточку, вы создаете экземпляр объекта, а записи на карточке описывают его состояние.

Методы класса — это действия, которые вы выполняете с конкретной карточкой. Например, getName(), changeName(), setName() могут быть методами класса Rolodex.

Итак, все карточки могут делать одни и те же вещи (getName(), changeName() и т. д.), но каждая из них обладает уникальной информацией.

# Создаем первый объект

Итак, что необходимо для создания и использования объекта? Вам понадобятся *два* класса. Один класс нужен для описания типа применяемого объекта (Dog, AlarmClock, Television и т. д.), другой — для *тестирования* нового класса. *Проверочный* класс содержит метод main(), в котором вы создаете объекты нового типа и получаете к ним доступ. Этот класс позволяет *опробовать* методы и переменные экземпляра объекта, созданного из вашего нового класса.

Далее в наших примерах вы неоднократно встретитесь с такими парными классами. Один класс будет *настоящим* — из него мы будем создавать объекты для последующего использования, а другой будет проверочным и носить название <*ИмяВашегоПервогоКласса*>**TestDrive**. Например, если мы создали класс **Bungee**, то тестовый класс будет называться **BungeeTestDrive**. Класс с именем <*ИмяВашегоПервогоКласса*>**TestDrive** будет содержать метод main(), и его единственная цель будет заключаться в создании объектов вашего нового типа (не проверочного класса) и доступе к его методам и переменным экземпляра через оператор доступа «точка» (.). Все это будет подробно рассмотрено на следующих примерах.

### Оператор доступа «точка» (.)

**Этот оператор предоставляет доступ к состоянию объекта и его поведению (к переменным экземпляра и методам).**

```
// Создаем новый объект
Dog d = new Dog();

// Приказываем ему подать голос,
// получая доступ к методу bark()
// через оператор «точка»
d.bark();

// Устанавливаем его размер,
// используя оператор доступа
d.size = 40;
```

## 1 Создаем класс.

```
class Dog {

  int size;
  String breed;
  String name;

  void bark() {
    System.out.println("Гав! Гав!");
  }
}
```

Переменные экземпляра.

Метод.

## 2 Создаем проверочный класс (TestDrive).

Главный метод (на следующем шаге мы добавим в него код).

```
class DogTestDrive {
  public static void main (String[] args) {
    // Проверочный код для класса Dog
  }
}
```

## 3 Создаем внутри тестового класса объект и получаем доступ к его переменным экземпляра и методам.

```
class DogTestDrive {
  public static void main (String[] args) {

    Dog d = new Dog();
    d.size = 40;
    d.bark();
  }
}
```

Создаем объект класса Dog.

Используем оператор доступа. Устанавливаем значение поля size.

Вызываем метод bark().

Оператор доступа «точка».

Если у вас уже сложилось определенное представление об ООП, вы должны были заметить, что в этом примере не используется инкапсуляция. Эту тему мы рассмотрим в главе 4.

# Создание и тестирование объектов Movie

```
class Movie {
  String title;
  String genre;
  int rating;
  void playIt() {
    System.out.println("Проигрывание фильма");
  }
}
public class MovieTestDrive {
  public static void main(String[] args) {
    Movie one = new Movie();
    one.title = "Как Прогореть на Акциях";
    one.genre = "Трагедия";
    one.rating = -2;
    Movie two = new Movie();
    two.title = "Потерянные в Офисе";
    two.genre = "Комедия";
    two.rating = 5;
    two.playIt();
    Movie three = new Movie();
    three.title = "Байт-Клуб";
    three.genre = "Трагедия, но в целом веселая";
    three.rating = 127;
  }
}
```

## Наточите свой карандаш

Класс MovieTestDrive создает объекты (экземпляры) класса Movie и с помощью оператора доступа (.) присваивает определенные значения его переменным экземпляра. Он также вызывает методы этих объектов. Справа показана схема. Заполните ее значениями из трех объектов, полученными к концу метода main().

object 1
- title
- genre
- rating

object 2
- title
- genre
- rating

object 3
- title
- genre
- rating

# Скорее! Выбирайтесь из главного метода!

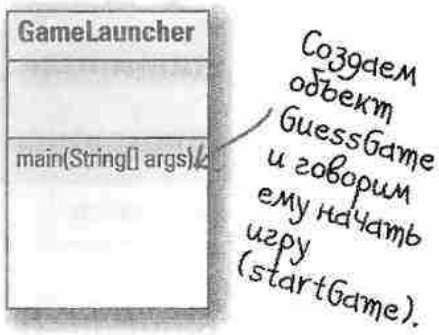
Будучи в методе main(), вы все еще находитесь за пределами Объективилля. Хранить весь код в главном методе — это нормально для тестовой программы, но при создании настоящих объектно ориентированных приложений ваши объекты должны будут общаться с другими объектами. Создавать и тестировать их внутри статического метода main() — не самая лучшая идея.

## Два варианта применения метода main:

- **тестирование** настоящего класса;
- **запуск/старт Java-приложения.**

Настоящее приложение на языке Java целиком состоит из объектов, общающихся между собой. *Общение* в данном случае означает, что объекты вызывают методы друг друга. На предыдущей странице, а также в главе 4 применение метода main() рассматривается в контексте отдельного класса TestDrive с целью создания и тестирования методов и переменных экземпляра другого класса. В главе 6 мы рассмотрим вариант использования класса с главным методом для запуска настоящего Java-приложения (создавая объекты и давая им возможность взаимодействовать друг с другом).

В следующем примере мы попытаемся показать, как может вести себя настоящее приложение на языке Java. Находясь на ранней стадии изучения Java, мы используем небольшое количество инструментов. По этой причине приложение покажется вам немного нескладным и неэффективным. Вероятно, вам захочется улучшить его, и именно этим мы займемся в следующих главах. Не волнуйтесь, если какие-то части кода останутся для вас непонятными; смысл примера — показать, как объекты общаются между собой.




## Игра в угадывание

**Краткое описание**

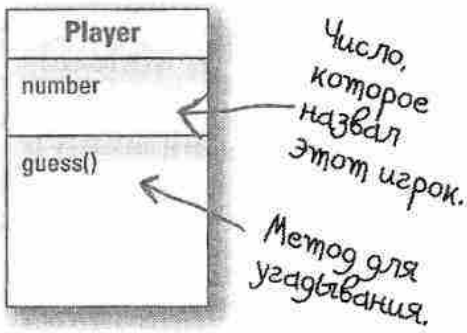
Игра предусматривает один игровой объект и три объекта-игрока. Генерируются случайные числа от 0 до 9, а три объекта-игрока пытаются их угадать (мы и не обещали, что это будет захватывающая игра).

**Классы**

`GuessGame.class` `Player.class` `GameLauncher.class`

**Логика**

1. Класс GameLauncher — это точка, из которой стартует приложение; он содержит метод main().
2. В методе main() создается объект GuessGame, из которого вызывается метод startGame().
3. В методе startGame() объекта GuessGame происходит весь игровой процесс. Он создает трех игроков, затем «придумывает» случайные числа (которые игроки должны угадывать). После того как каждого из игроков просят угадать число, проверяется результат и либо выводится информация о победителях, либо игроков просят угадать еще раз.



```
public class GuessGame {
   Player p1;
   Player p2;
   Player p3;

   public void startGame() {
     p1 = new Player();
     p2 = new Player();
     p3 = new Player();

     int guessp1 = 0;
     int guessp2 = 0;
     int guessp3 = 0;

     boolean p1isRight = false;
     boolean p2isRight = false;
     boolean p3isRight = false;

     int targetNumber = (int) (Math.random() * 10);
     System.out.println("Я загадываю число от 0 до 9...");

     while(true) {
       System.out.println("Число, которое нужно угадать, — " + targetNumber);

       p1.guess();
       p2.guess();
       p3.guess();

       guessp1 = p1.number;
       System.out.println("Первый игрок думает, что это " + guessp1);

       guessp2 = p2.number;
       System.out.println("Второй игрок думает, что это " + guessp2);

       guessp3 = p3.number;
       System.out.println("Третий игрок думает, что это " + guessp3);

       if (guessp1 == targetNumber) {
          p1isRight = true;
       }
       if (guessp2 == targetNumber) {
          p2isRight = true;
       }
       if (guessp3 == targetNumber) {
         p3isRight = true;
       }

       if (p1isRight || p2isRight || p3isRight) {

         System.out.println("У нас есть победитель!");
         System.out.println("Первый игрок угадал?" + p1isRight);
         System.out.println("Второй игрок угадал?" + p2isRight);
         System.out.println("Третий игрок угадал?" + p3isRight);
         System.out.println("Конец игры.");
         break; // Игра окончена, так что прерываем цикл

       } else {
         // Мы должны продолжать, так как никто не угадал!
         System.out.println("Игроки должны попробовать еще раз.");
       } // конец if/else
     } // конец цикла
   } // конец метода
} // конец класса
```

GuessGame содержит три переменных экземпляра для трех объектов Player.

Создаем три объекта Player и присваиваем их трем переменным экземпляра.

Объявляем три переменные для хранения вариантов от каждого игрока.

Объявляем три переменные для хранения правильности или неправильности (true или false) ответов игроков.

Создаем число, которое игроки должны угадать.

Вызываем метод guess() из каждого объекта Player.

Извлекаем варианты каждого игрока (результаты работы их методов guess()), получая доступ к их переменным number.

Проверяем варианты каждого из игроков на соответствие загаданному числу. Если игрок угадал, то присваиваем соответствующей переменной значение true (помните, что по умолчанию она хранит значение false).

Если первый игрок, ИЛИ второй игрок, ИЛИ третий игрок угадал (оператор || означает ИЛИ)...

Иначе остаемся в цикле и просим игроков сделать еще одну попытку.

# Запускаем нашу игру

```
public class Player {
  int number = 0; // Здесь хранится вариант числа
  public void guess() {
    number = (int) (Math.random() * 10);
    System.out.println("Думаю, это число " + number);
  }
}
public class GameLauncher {
  public static void main (String[] args) {
    GuessGame game = new GuessGame();
    game.startGame();
  }
}
```

## Java выносит мусор

Каждый раз, когда в Java создается объект, он отправляется в область памяти под названием **куча (heap)**. Все объекты, независимо от того, когда, где или как они были созданы, расположены в куче. Но это не просто классическая куча в памяти — в Java она управляется сборщиком мусора. Когда вы создаете объект, Java выделяет участок памяти в куче такого размера, который необходим этому объекту. Например, объект с 15 переменными экземпляра, вероятно, потребует больше места, чем объект с двумя переменными. Но что происходит, когда вам нужно освободить это пространство? Каким образом можно убрать объект из кучи после того, как он больше не нужен? Java сделает это за вас!

Когда JVM видит, что объект больше не понадобится, она делает его **пригодным для сборки мусора**. И если у вас заканчивается память, сборщик мусора запускается, выбрасывает недоступные объекты и освобождает место, чтобы вы снова могли его использовать. В следующих главах вы больше узнаете об этом процессе.

**Программный вывод (будет отличаться от запуска к запуску)**

```
File Edit Window Help Explode
%java GameLauncher
Я загадываю число от 0 до 9..
Число, которое нужно угадать, — 7
Думаю, это число 1
Думаю, это число 9
Думаю, это число 9
Первый игрок думает, что это 1
Второй игрок думает, что это 9
Третий игрок думает, что это 9
Игроки должны попробовать еще раз
Число, которое нужно угадать, — 7
Думаю, это число 3
Думаю, это число 0
Думаю, это число 9
Первый игрок думает, что это 3
Второй игрок думает, что это 0
Третий игрок думает, что это 9
Игроки должны попробовать еще раз
Число, которое нужно угадать, — 7
Думаю, это число 7
Думаю, это число 5
Думаю, это число 0
Первый игрок думает, что это 7
Второй игрок думает, что это 5
Третий игрок думает, что это 0
У нас есть победитель!
Первый игрок угадал? true
```

## Это не глупые вопросы

**В: А вдруг мне понадобятся глобальные переменные и методы? Как я смогу реализовать их, если все должно храниться в классах?**

О: В объектно ориентированных программах на языке Java нет понятия глобальных переменных и методов. Однако иногда нужно, чтобы какие-то методы или константы были доступны в любой части программы. Вспомните о методе `random()` из приложения для генерирования фраз; он должен быть доступен везде. А как насчет такой константы, как число пи? В главе 10 вы узнаете, что ключевые слова `public` и `static` делают метод почти глобальным. Вы можете вызывать публичные статические методы из любого кода, из любого класса своей программы. А пометив поле как `public`, `static` и `final`, вы получите глобальную *константу*.

**В: Как это связано с ООП, если вы все равно можете создавать глобальные функции и данные?**

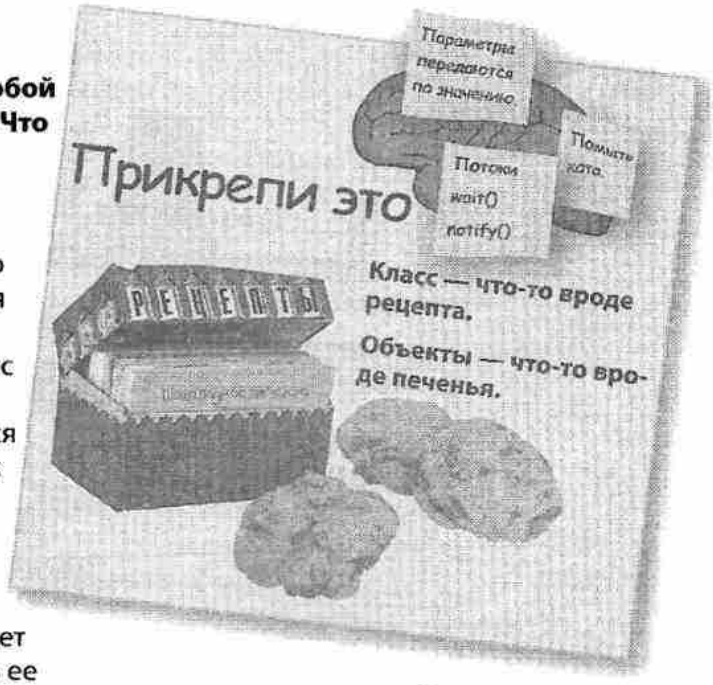
О: Прежде всего все в языке Java разбито по классам. Число пи и метод `random()`, несмотря на ключевые слова `public` и `static`, объявлены внутри класса `Math`. И вы должны помнить, что эти статические (как бы глобальные) сущности считаются для Java скорее исключением, чем правилом. Они уместны в особых случаях, когда у вас нет нескольких экземпляров/объектов.

**В: Что представляет собой программа на языке Java? Что мы *получаем* на выходе?**

О: Java-программа — это набор классов (где есть хотя бы *один* класс). В приложении на языке Java один класс должен содержать главный метод, который используется для запуска программы. Как программист вы создаете один или несколько классов, которые и будут продуктом вашего труда. Если у конечного пользователя нет JVM, вам придется добавить ее в свое приложение, чтобы запустить его. Существует несколько установщиков, которые позволяют упаковывать классы вместе с различными версиями JVM (например, для разных платформ) и размещать все это на компакт-диске. Таким образом, конечный пользователь сможет установить корректную версию JVM (конечно, если он этого еще не сделал).

**В: А если у меня есть сто классов? Или тысяча? Разве удобно поставлять столько отдельных файлов? Могу ли я упаковать их в единственный *исполняемый файл*?**

О: Да, поставлять столько отдельных файлов конечному пользователю очень неудобно, но вам и не придется этого делать. Вы можете упаковать все свои программные файлы в единый Java-архив — *файл JAR*, который основывается на формате PKZIP. В такой архив вы можете добавить простой текстовый файл, отформатированный в виде *манифеста*. В манифесте определяется, какой класс в архиве содержит метод main(), предназначенный для запуска приложения.




### КЛЮЧЕВЫЕ МОМЕНТЫ

- Объектно ориентированное программирование позволяет расширять приложение, не затрагивая проверенный ранее и работающий код.
- Весь код в Java находится внутри **классов**.
- Класс описывает, как создавать объект определенного типа. **Класс — это что-то вроде шаблона.**
- Объект может сам о себе позаботиться; вам не нужно знать, *как* именно он это делает.
- Объект кое-что **знает** и **умеет** делать.
- Сведения об объекте называются **переменными экземпляра**. Они определяют состояние объекта.
- Действия объекта называются **методами**. Они обуславливают *поведение* объекта.
- Для каждого класса можно предусмотреть отдельный проверочный класс, который будет использоваться для создания объектов нового класса.
- Класс может **наследовать** поля и методы от более абстрактного **родительского класса**.
- Работающая программа на языке Java — это не что иное, как набор объектов, которые общаются между собой.

# Поработайте компилятором

Каждый Java-файл на этой странице представляет собой полноценный исходник. Ваша задача — притвориться компилятором и определить, все ли из них скомпилируются. Если компиляция не сможет пройти успешно, как вы исправите файлы? Если же они скомпилируются, какой результат на экране вы получите?

**A**

```
class TapeDeck {

  boolean canRecord = false;

  void playTape() {
    System.out.println("пленка проигрывается");
  }

  void recordTape() {
    System.out.println("идет запись на пленку");
  }
}

class TapeDeckTestDrive {
  public static void main(String [] args) {

    t.canRecord = true;
    t.playTape();

    if (t.canRecord == true) {
      t.recordTape();
    }
  }
}
```

**B**

```
class DVDPlayer {

  boolean canRecord = false;

  void recordDVD() {
    System.out.println("идет запись DVD");
  }
}

class DVDPlayerTestDrive {
  public static void main(String [] args) {

    DVDPlayer d = new DVDPlayer();
    d.canRecord = true;
    d.playDVD();

    if (d.canRecord == true) {
      d.recordDVD();
    }
  }
}
```

# Магнитики с кодом

Части рабочего Java-приложения разбросаны по всему холодильнику. Можете ли вы восстановить из фрагментов кода работоспособную программу, которая будет выводить приведенный ниже текст? Некоторые фигурные скобки упали на пол. Они настолько маленькие, что их нельзя поднять, поэтому разрешено добавлять столько скобок, сколько понадобится.

```
d.playSnare();
```

```
DrumKit d = new DrumKit();
```

```
boolean topHat = true;
boolean snare = true;
```

```
void playSnare() {
  System.out.println("бах бах ба-бах");
}
```

```
public static void main(String [] args) {
```

```
if (d.snare == true) {
  d.playSnare();
}
```

```
d.snare = false;
```

```
class DrumKitTestDrive {
```

```
d.playTopHat();
```

```
class DrumKit {
```

```
void playTopHat () {
  System.out.println("динь динь ди-динь");
}
```

```
File Edit Window Help Dance
% java DrumKitTestDrive
бах бах ба-бах
динь динь ди-динь
```

# Головоломка у бассейна

Ваша **задача** — взять фрагменты кода со дна бассейна и заменить ими пропущенные участки программы. Вы **можете** использовать один фрагмент несколько раз, но не все из них вам пригодятся. Ваша **цель** — создать класс, который скомпилируется, запустится и выведет приведенный ниже текст.

**Результат:**

```
File Edit Window Help Implode
%java EchoTestDrive
привееееет...
привееееет...
привееееет...
привееееет...
10
```

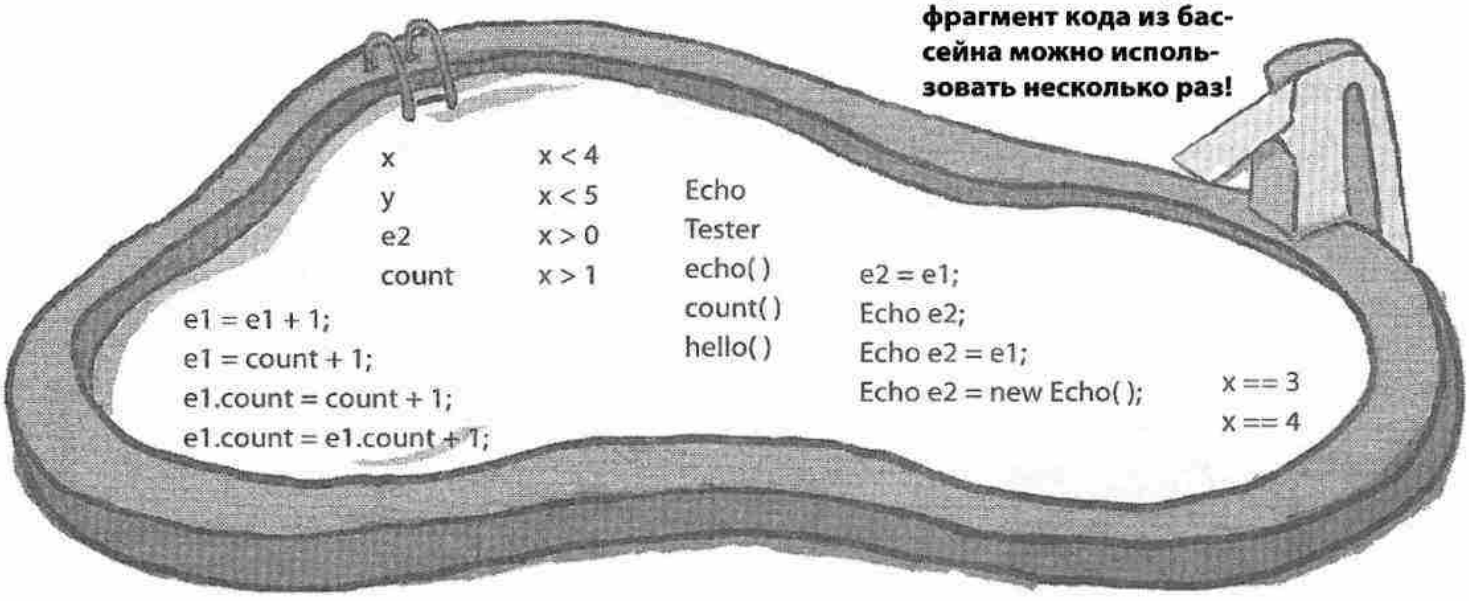
```
public class EchoTestDrive {
  public static void main(String [] args) {
    Echo e1 = new Echo();
    _____________________________
    int x = 0;
    while ( _____________ ) {
      e1.hello();
      _____________________________
      if ( _____________ ) {
        e2.count = e2.count + 1;
      }
      if ( _____________ ) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
    }
    System.out.println(e2.count);
  }
}
```

```
class _____________ {
  int _____________ = 0;
  void _____________ {
    System.out.println("привееееет...");
  }
}
```

**Призовой вопрос!**

Как бы вы завершили эту головоломку, если бы в последней строке было не **10,** а **24**?

**Примечание: каждый фрагмент кода из бассейна можно использовать несколько раз!**



# Кто я такой?

Команда звезд из мира Java хочет сыграть с вами в игру «Кто я такой?». Они дают вам подсказку, а вы пытаетесь угадать их имена. Считайте, что они никогда не врут. Если говорят что-нибудь подходящее сразу для нескольких участников, то записывайте всех, к кому применимо данное утверждение. Заполните пустые строки именами одного или нескольких участников рядом с утверждениями. Первого мы взяли на себя.

**Сегодня участвуют:**

**Класс Метод Объект Поле**

Я компилируюсь из файла JAVA. Класс.

Значения моих переменных экземпляра могут отличаться от значений моих полей ввода. _____

Я выступаю в роли шаблона. _____

Я люблю что-нибудь делать. _____

У меня может быть много методов. _____

Я представляю собой состояние. _____

Я могу вести себя по-разному. _____

Я нахожусь в объектах. _____

Я обитаю в куче. _____

Меня используют для создания экземпляров объекта. _____

Мое состояние может меняться. _____

Я объявляю методы. _____

Я могу изменяться во время выполнения программы. _____

# Ответы

## Магнитики с кодом

```
class DrumKit {

  boolean topHat = true;
  boolean snare = true;

  void playTopHat() {
    System.out.println("динь динь ди-динь");
  }

  void playSnare() {
    System.out.println("бах бах ба-бах");
  }
}

class DrumKitTestDrive {
  public static void main(String [] args) {

    DrumKit d = new DrumKit();
    d.playSnare();
    d.snare = false;
    d.playTopHat();

    if (d.snare == true) {
      d.playSnare();
    }
  }
}
```

```
File Edit Window Help Dance
% java DrumKitTestDrive
бах бах ба-бах
динь динь ди-динь
```

## Поработайте компилятором

A

```
class TapeDeck {
  boolean canRecord = false;
  void playTape() {
    System.out.println("пленка проигрывается");
  }
  void recordTape() {
    System.out.println("идет запись на пленку");
  }
}

class TapeDeckTestDrive {
  public static void main(String [] args) {

    TapeDeck t = new TapeDeck( );
    t.canRecord = true;
    t.playTape();

    if (t.canRecord == true) {
      t.recordTape();
    }
  }
}
```

У нас есть шаблон, теперь мы можем создать объект!

B

```
class DVDPlayer {
  boolean canRecord = false;
  void recordDVD() {
    System.out.println("идет запись DVD");
  }
  void playDVD ( ) {
    System.out.println("DVD проигрывается");
  }
}

class DVDPlayerTestDrive {
  public static void main(String [] args) {
    DVDPlayer d = new DVDPlayer();
    d.canRecord = true;
    d.playDVD();
    if (d.canRecord == true) {
      d.recordDVD();
    }
  }
}
```

Строка d.playDVD( ); не будет скомпилирована без метода!

# Ответы

## Головоломка у бассейна

```
public class EchoTestDrive {
  public static void main(String [] args) {
    Echo e1 = new Echo();
    Echo e2 = new Echo(); // правильный ответ
          - или -
    Echo e2 = e1;  // бонусный ответ!
    int x = 0;
    while ( x < 4 ) {
      e1.hello();
      e1.count = e1.count + 1;
      if ( x == 3 ) {
        e2.count = e2.count + 1;
      }
      if ( x > 0 ) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
    }
    System.out.println(e2.count);
  }
}
```

---

```
class Echo {
  int count = 0;
  void hello( ) {
    System.out.println("привееееет...");
  }
}
```

**Результат:**

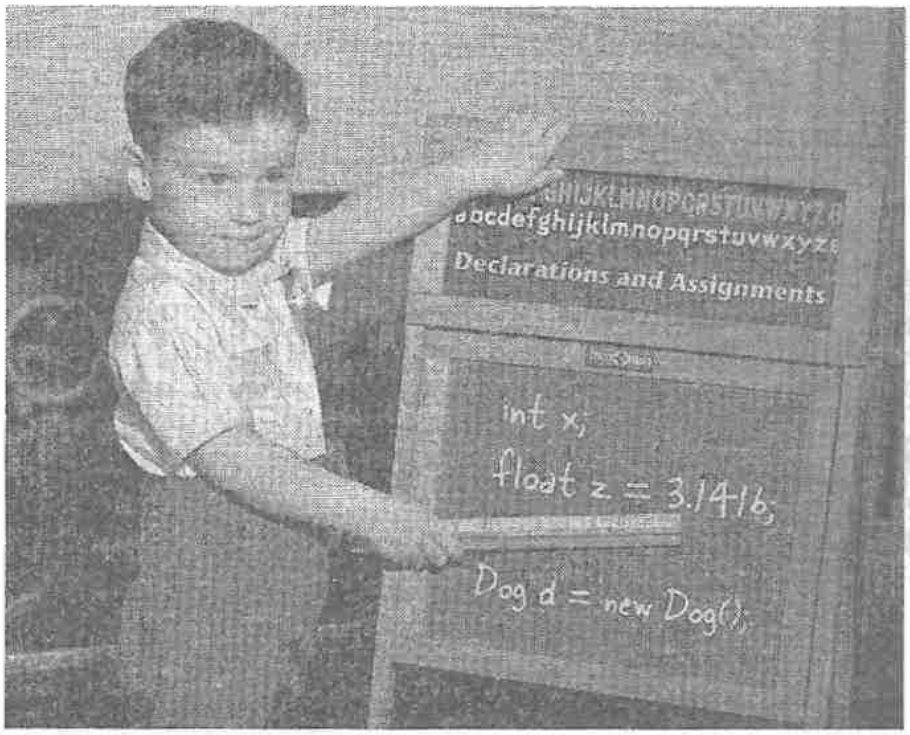
```
File Edit Window Help Assimilate
%java EchoTestDrive
привееееет...
привееееет...
привееееет...
привееееет...
10
```

## Кто я такой?

| | |
|---|---|
| Я компилируюсь из файла JAVA. | Класс. |
| Значения моих переменных экземпляра могут отличаться от значений моих полей ввода. | Объект. |
| Я выступаю в роли шаблона. | Класс. |
| Я люблю что-нибудь делать. | Объект, метод. |
| У меня может быть много методов. | Класс, объект. |
| Я представляю собой состояние. | Переменная экземпляра. |
| Я могу вести себя по-разному. | Объект, класс. |
| Я нахожусь в объектах. | Метод, переменная экземпляра. |
| Я обитаю в куче. | Объект. |
| Меня используют для создания экземпляров объекта. | Класс. |
| Мое состояние может меняться. | Объект, поле. |
| Я объявляю методы. | Класс. |
| Я могу изменяться во время выполнения программы. | Объект, поле. |

Примечание: Состояние и поведение свойственны как классам, так и объектам. Они определяются внутри класса, но объект ими также обладает. Пока нам все равно, где они на самом деле обитают.

# Свои переменные нужно знать в лицо




**Переменные делятся на два вида:** примитивы (простые типы) **и** ссылки. Пока вы использовали их либо как переменные **экземпляра** (которые описывают состояние объекта), либо как **локальные** переменные (которые объявляются внутри метода). Далее мы будем применять их в **качестве аргументов** (значений, передающихся методу из вызываемого кода) и возвращаемых значений (которые возвращаются методом в вызывающий код). Вы уже видели переменные, объявленные в качестве **простых** целочисленных значений (тип `int`). Вы также видели более сложные **элементы** — строки или массивы. Но **в жизни должно быть что-то помимо** чисел, строк и массивов. Как быть с объектом PetOwner с переменной экземпляра типа Dog? Или Cat с переменной экземпляра Engine? В этой главе мы приоткроем завесу тайны над типами в языке Java и вы узнаете, что именно можно *объявлять* в качестве переменных, какие значения *присваивать* им и как вообще с ними *работать*.

# Объявление переменной

**Java заботится о типах.** Он не позволит вам сделать что-либо странное, например засунуть ссылку типа Giraffe в переменную Rabbit. Java также не разрешит присвоить число с плавающей точкой целочисленной переменной, если только вы *не скажете компилятору*, что сознательно решили обойтись без точных результатов (например, убрав дробную часть).

Компилятор может распознать большинство проблем:

```
Rabbit hopper = new Giraffe();
```

Даже не надейтесь, что это скомпилируется.

Чтобы все эти меры безопасности работали, нужно указать тип своей переменной. Это целое число? Тип Dog? Одиночный символ? Как уже говорилось, переменные делятся на два вида: ***примитивы*** (простые типы) и ***ссылки***. Первые хранят основные значения (иначе говоря, простые битовые структуры), включая целые числа, булевы значения и числа с плавающей точкой. Вторые хранят *ссылки* на *объекты*.

Первым делом мы рассмотрим простые типы, а затем попытаемся разобраться, что на самом деле представляют собой ссылки на объекты. Но вне зависимости от типа переменных при их объявлении вы должны соблюдать два правила.

---

## Переменная должна иметь тип.

---

Помимо типа у переменной должно быть имя, чтобы можно было работать с ней внутри кода.

---

## Переменная должна иметь имя.

---

```
int count;
```

Тип. Имя.

Примечание: когда вы видите выражение **«Объект типа X»**, считайте, что *тип* и *класс* — синонимы (мы рассмотрим это в следующих главах).

# «Мне, пожалуйста, двойной мокко. Хотя нет, сделайте его цельным»

Думайте о переменных в Java как о чашках. Кофейных, чайных, больших пивных кружках, стаканах для попкорна, чашечках с симпатичными изогнутыми ручками, бокалах с металлической обводкой, которые ни при каких обстоятельствах нельзя ставить в микроволновую печь.

**Переменная — это просто посуда. Контейнер. Она предназначена для *хранения*.**

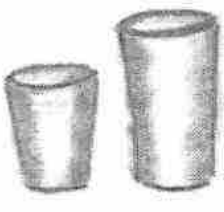
У нее есть размер и тип. В этой главе мы сначала рассмотрим переменные (чашки), предназначенные для ***простых типов***, а чуть позже перейдем к изучению посуды, которая хранит *ссылки на объекты*. Будем придерживаться такой аналогии с чашками, что поможет нам, когда речь пойдет о более сложных вещах.

Примитивы похожи на стаканчики в кофейнях. Если вы бывали в Старбаксе, то знаете, о чем идет речь. Стаканчики бывают разных размеров, и каждый называется по-своему — «короткий», «высокий» и «большое мокко пополам с воздушным кремом».

Эти стаканчики выставлены на прилавке, так что можно заказать:

Так и в Java: простые типы имеют разные размеры и названия. Объявляя переменную, вы должны указать для нее конкретный тип. Следующие четыре контейнера предназначены для четырех числовых типов Java.

Каждый контейнер содержит значение, поэтому для простых типов в Java вместо «Я буду большой кофе по-французски» вы говорите компилятору: «Мне, пожалуйста, целочисленную переменную с числом 90». Но есть одна небольшая особенность. Вы также должны дать своей чашке *имя*, поэтому итоговая фраза будет звучать так: «Мне, пожалуйста, целочисленную переменную со значением 2486 и именем ***height***». Каждый простой тип имеет фиксированное количество бит (размер чашки). Далее перечислены размеры шести простых числовых типов в Java.




## Простые типы

| Тип | Количество бит | Диапазон значений |
|---|---|---|
| **boolean и char** | | |
| boolean | (зависит от JVM) | ***true*** или ***false*** |
| char | 16 бит | от 0 до 65 535 |
| **Числовые (все числа со знаком)** | | |
| ***Целочисленные*** | | |
| byte | 8 бит | от –128 до 127 |
| short | 16 бит | от –32 768 до 32 767 |
| int | 32 бит | от –2 147 483 648 до 2 147 483 647 |
| long | 64 бит | от $-9 \times 10^{18}$ до $9 \times 10^{18}$ |
| ***Типы с плавающей точкой*** | | |
| float | 32 бит | от $-3{,}4 \times 10^{38}$ до $3{,}4 \times 10^{38}$ |
| double | 64 бит | от $\pm 5{,}0 \times 10^{-324}$ до $\pm 1{,}7 \times 10^{308}$ |

**Объявление переменных простых типов и присваивание им значений**

```
int x;
x = 234;
byte b = 89;
boolean isFun = true;
double d = 3456.98;
char c = 'f';
int z = x;
boolean isPunkRock;
isPunkRock = false;
boolean powerOn;
powerOn = isFun;
long big = 3456789;
float f = 32.5f;
```

Обратите внимание на символ «f». Его нужно применять вместе с типом float, так как в ином случае все значения с точкой в Java автоматически относятся к типу double.

# Постарайтесь ничего не рассыпать...

**Проверяйте, помещается ли значение в переменную.**

Вы не можете поместить большое значение в маленькую «чашку».

Честно говоря, это можно сделать, но тогда кое-что потеряется. Произойдет своего рода переполнение. Компилятор попытается предотвратить такую ситуацию, если увидит, что в вашем коде какая-то сущность не вмещается в контейнер (переменную/чашку), который вы используете.

Например, вы не можете «залить» значение типа int в контейнер размера byte, как показано ниже.

```
int x = 24;
byte b = x;
//Не сработает!
```

Вы спросите, почему это не работает? Значение *x* равно 24, значит, оно полностью совместимо с типом byte. *Нам* это известно, но компилятора волнует только тот факт, что вы пытаетесь поместить большую сущность в маленькую, из-за чего *может* произойти переполнение. Не надейтесь, что компилятор будет знать значение переменной *x*, даже если вы прямо указали его в своем коде.

**Вы можете присвоить значение переменной несколькими способами:**

- добавить литерал после знака «равно» (x = ***12***, isGood = ***true*** и т. д.);
- присвоить переменной значение, которое принадлежит другой переменной (x = y);
- использовать выражение, сочетая оба подхода (x = y + ***43***).

В следующих примерах значения-литералы выделены полужирным шрифтом.

```
int size = 32;
```
Объявляем переменную типа int с именем *size*, присваиваем ей значение *32*.

```
char initial = 'j';
```
Объявляем переменную типа char с именем *initial*, присваиваем ей значение *'j'*.

```
double d = 456.709;
```
Объявляем переменную типа double с именем *d*, присваиваем ей значение *456.709*.

```
boolean isCrazy;
```
Объявляем переменную типа boolean с именем *isCrazy* (ничего ей не присваиваем).

```
isCrazy = true;
```
Присваиваем значение *true* объявленной ранее переменной *isCrazy*.

```
int y = x + 456;
```
Объявляем переменную типа int с именем y, присваиваем ей значение, которое является суммой *x* (чему бы он ни равнялся) и *456*.

## Наточите свой карандаш

Компилятор не позволит вам поместить значение из большой «чашки» в маленькую. Но можно ли перелить содержимое маленькой чашки в большую? **Без проблем.**

На основании сведений о размере и типе переменной подумайте, сможете ли вы определить, какое из выражений допустимо, а какое — нет. Мы пока не рассмотрели все правила, поэтому в некоторых случаях вам придется угадывать. **Подсказка:** компилятор всегда заботится о безопасности.

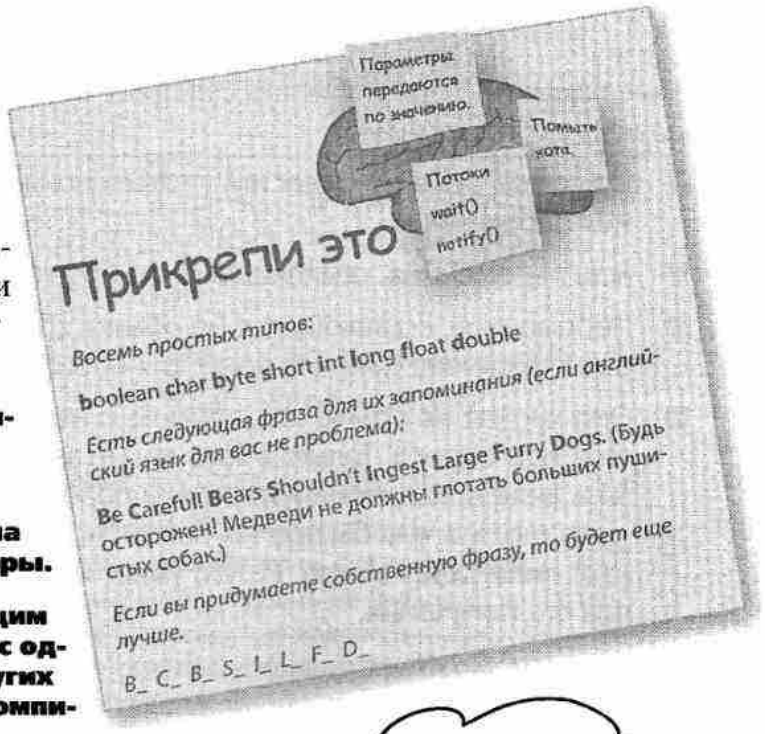
В следующем списке выделите допустимые выражения, исходя из того, что они используются в одном и том же методе.

```
1. int x = 34.5;
2. boolean boo = x;
3. int g = 17;
4. int y = g;
5. y = y + 10;
6. short s;
7. s = y;
8. byte b = 3;
9. byte v = b;
10. short n = 12;
11. v = n;
12. byte k = 128;
13. int p = 3 * g + y;
```

# Держитесь подальше от этого ключевого слова!

Вы знаете, что переменным нужны тип и имя.
Вы уже изучили простые типы.

***Но что вы вправе использовать в качестве имен?***
Можно подбирать имена для классов, методов или переменных исходя из следующих правил (в реальности все не настолько строго, но их знание убережет вас от ошибок).




- **Имя должно начинаться с буквы, знака подчеркивания (_) или знака доллара ($). Нельзя начинать имя с цифры.**
- **После первого символа можно использовать числа без ограничений. Главное, не начинать имя с цифры.**
- **Вы вправе выбрать любое имя, следуя предыдущим двум правилам, но только если оно не совпадает с одним из зарезервированных ключевых слов (и других названий) в языке Java, которые распознаются компилятором.**

Если вы действительно хотите запутать компилятор, попробуйте указать в качестве имени зарезервированное слово.

```
public   static   void
```

Не используйте эти слова в качестве имен

Точно так же зарезервированы названия простых типов:

```
boolean char byte short int long float double
```

Ключевых слов гораздо больше, чем перечислено здесь. Даже если вам не нужно знать, что они означают, вы все равно должны понимать, что их нельзя использовать при именовании переменных. ***Не пытайтесь запомнить их сейчас.*** Чтобы найти в своей памяти место для этого, вам, вероятно, придется забыть что-то другое, например где припаркован ваш автомобиль. Не волнуйтесь — к концу книги вы будете прекрасно знать большинство из этих слов.

## Эти слова зарезервированы.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| boolean | byte | char | double | float | int | long | short | public | private |
| protected | abstract | final | native | static | strictfp | synchronized | transient | volatile | if |
| else | do | while | switch | case | default | for | break | continue | assert |
| class | extends | implements | import | instanceof | interface | new | package | super | this |
| catch | finally | try | throw | throws | return | void | const | goto | enum |

Ключевые слова языка Java и другие зарезервированные сочетания символов (в случайном порядке). Если вы укажете их в качестве имен, компилятор очень огорчится.

# Управляем объектом Dog

Вы знаете, как объявить переменную простого типа и присвоить ей значение. Но что делать с непростыми переменными, *то есть с объектами*?

- На самом деле не существует такого понятия, как объектная переменная.
- Есть переменная, ссылающаяся на объект.
- Переменная, ссылающаяся на объект, хранит биты, которые описывают путь для доступа к объекту.
- Она хранит не объект как таковой, а нечто вроде указателя или адреса. Только в Java мы не знаем, что на самом деле находится внутри такой переменной. Но точно известно, что чем бы она ни была, она всегда представляет лишь один объект. И JVM умеет использовать ссылку для его получения.

Нельзя вложить объект в переменную. Мы часто думаем об этом что-то вроде «я передал объект String методу System.out.println()». Или «метод возвращает объект Dog», или «я поместил новый объект Foo в переменную под названием myFoo».

На самом деле все не так. Нет гигантских расширяемых контейнеров, которые могут подстроиться под размер любого объекта. Объект существует лишь в одном месте — в куче, управляемой сборщиком мусора (более подробно читайте об этом далее)!

Хотя переменные простых типов состоят из битов и действительно представляют свои ***значения***, ссылочные переменные содержат биты, которые описывают ***способ получения объекта***.

Вы использовали оператор доступа «точка» (.) в сочетании со ссылочной переменной, чтобы сказать: «Используй элемент *перед* точкой, чтобы дать мне элемент, находящийся *после* точки». Например, команда:

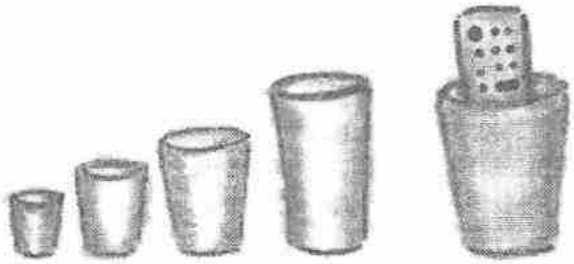
```
myDog.bark();
```

означает: «Используй объект, на который ссылается переменная myDog, чтобы вызвать метод bark()». Когда вы используете оператор «точка» (.) в сочетании со ссылкой на объект, думайте об этом, как о нажатии кнопок на пульте дистанционного управления объектом.

**Dog d = new Dog();**
**d.bark();**

Думайте об этом.

Думайте о ссылочной переменной типа Dog как о пульте дистанционного управления. Вы используете его, чтобы приказывать объекту что-то сделать (вызывая методы).




# Ссылка на объект — всего лишь еще одно значение переменной.

**Нечто такое, что хранится внутри «чашки». Только в данном случае значение — это пульт дистанционного управления.**

## Переменная простого типа

```
byte x = 7;
```

Биты, представляющие число 7, хранятся в переменной.

## Ссылочная переменная

```
Dog myDog = new Dog();
```

Биты, описывающие способ получить объект Dog, хранятся в переменной.

***Но сам по себе объект Dog не в переменной!***

Значение переменной простого типа представляет собой значение (5, –26,7, 'a').

Значение ссылочной переменной — это биты, описывающие способ получения конкретного объекта.

Вы не знаете (и вам должно быть все равно), как конкретная версия JVM реализует ссылки на объекты. Конечно, это могут быть указатели на указатели на... Но даже если вы знаете, то все равно не можете использовать эти биты ни для чего другого, кроме доступа к объекту.

Нам не важно, сколько нулей и единиц хранится в ссылочной переменной. Это может зависеть от конкретной JVM или от фазы Луны.

# Три шага на пути к объявлению, созданию и инициализации объекта.

```
Dog myDog = new Dog();
```
1 — Dog myDog; 3 — =; 2 — new Dog();

### 1 Объявляем ссылочную переменную.

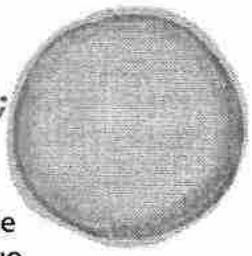
```
Dog myDog = new Dog();
```

Это говорит JVM, что она должна выделить место в памяти для ссылочной переменной и назвать ее *myDog*. Ссылочная переменная навсегда получает тип Dog. Иными словами, это пульт управления, у которого есть кнопки для работы именно с объектом Dog, а не Cat, Button или Socket.

### 2 Создаем объект.

```
Dog myDog = new Dog();
```

Здесь JVM получает приказ выделить память в куче для объекта Dog (позже вы значительно больше узнаете об этом процессе, особенно в главе 9).




### 3 Связываем объект и ссылку.

```
Dog myDog = new Dog();
```

Присваиваем ссылочной переменной myDog новый объект типа Dog. Другими словами, **программируем пульт управления**.

## Это не глупые вопросы

**В: Насколько велика ссылочная переменная?**

О: Это неизвестно. Где-то внутри нее есть указатели, но к ним нельзя получить доступ. Вам это просто не нужно. Но если вы очень хотите понять, как выглядит ссылка, можете представить ее в виде 64-битного значения. Однако, рассуждая о вопросах выделения памяти, вы должны беспокоиться о том, сколько *объектов* (а не *ссылок* на объекты) создаете и насколько они большие.

**В: Означает ли это, что все ссылки имеют одинаковый размер независимо от величины объектов, на которые они ссылаются?**

О: Да. Все ссылки одной версии JVM будут иметь одинаковый размер независимо от объектов, на которые они указывают. Но каждая JVM может использовать собственный способ представления ссылок, поэтому от версии к версии размеры ссылки могут изменяться.

**В: Можно ли выполнять арифметические операции со ссылочными переменными, инкрементировать их, как в языке C?**

О: Нет. Java — это не C.

# РАЗОБЛАЧЕНИЕ ЯЗЫКА JAVA

## Интервью этой недели: Объектная ссылка

**HeadFirst:** Итак, расскажите нам, как поживает объектная ссылка.

**Ссылка:** У меня довольно простая жизнь. Я — дистанционный пульт управления, и меня могут запрограммировать для манипуляции разными объектами.

**HeadFirst:** Вы имеете в виду разные объекты на протяжении одной сессии? Например, можете ли вы ссылаться на объект Dog, а через пять минут на объект Car?

**Ссылка:** Конечно, нет. Меня объявляют лишь для одного объекта. Если я управляю объектом Dog, я уже никогда не смогу указывать (ой, мы ведь не должны говорить «указывать»!), то есть ссылаться ни на что другое, кроме объектов типа Dog.

**HeadFirst:** Значит ли это, что вы можете ссылаться только на один объект Dog?

**Ссылка:** Нет. Я могу ссылаться на один объект Dog, а через пять минут — уже на другой. Если объект имеет тип Dog, я могу быть перенаправлена (как пульт управления перепрограммируется для другого телевизора) на него. Кроме случаев... Хотя нет, не обращайте внимания.

**HeadFirst:** Нет уж, не останавливайтесь. Что вы собирались сказать?

**Ссылка:** Вам вряд ли захочется обсуждать это сейчас, но я попробую рассказать в двух словах. Если я помечена как final и мне присвоили объект Dog, то я уже не смогу ни с чем работать, кроме этого объекта Dog. Иначе говоря, мне нельзя будет присвоить никакой другой объект.

**HeadFirst:** Вы правы, не будем говорить об этом сейчас. Значит, кроме тех случаев, когда вы помечены как final, вы можете ссылаться сначала на один объект типа Dog, а позже на другой. А можете ли вы не ссылаться *вообще ни на что*? Можете ли не быть ни для чего запрограммированной?

**Ссылка:** Да, но мне неприятно об этом говорить.

**HeadFirst:** Почему?

**Ссылка:** Потому что в таких случаях я полный ноль (`null`), и это меня расстраивает.

**HeadFirst:** По той причине, что у вас нет значения?

**Ссылка:** Ох, `null` — это и есть значение. Я остаюсь пультом управления, но это то же самое, что купить меня и принести в дом, где нет ни одного телевизора. Я не запрограммирована и не могу ничем управлять. Пользователи сутками напролет будут нажимать мои кнопки, но ничего хорошего из этого не выйдет. Я чувствую себя такой бесполезной и зря трачу свои биты. К счастью, их не так много, но все же. Меня устанавливают в `null` (распрограммируют), и значит, *никто* не может получить тот объект, на который я ссылалась раньше.

**HeadFirst:** И это плохо, потому что...

**Ссылка:** Вы еще *спрашиваете*? Сначала я выстраиваю отношения с объектом, налаживаю связь, а затем нас внезапно безжалостно разлучают. И я никогда больше не увижу его, потому что теперь он первый кандидат [режиссер, включите трагическую музыку] на свидание со *сборщиком мусора*. (Всхлип.) Думаете, программист хотя бы осознает, *что* натворил? (Всхлип.) *Почему* я не могу быть примитивом? *Я ненавижу быть ссылкой.* Ответственность, все эти разорванные связи...

## Жизнь в куче под управлением сборщика мусора

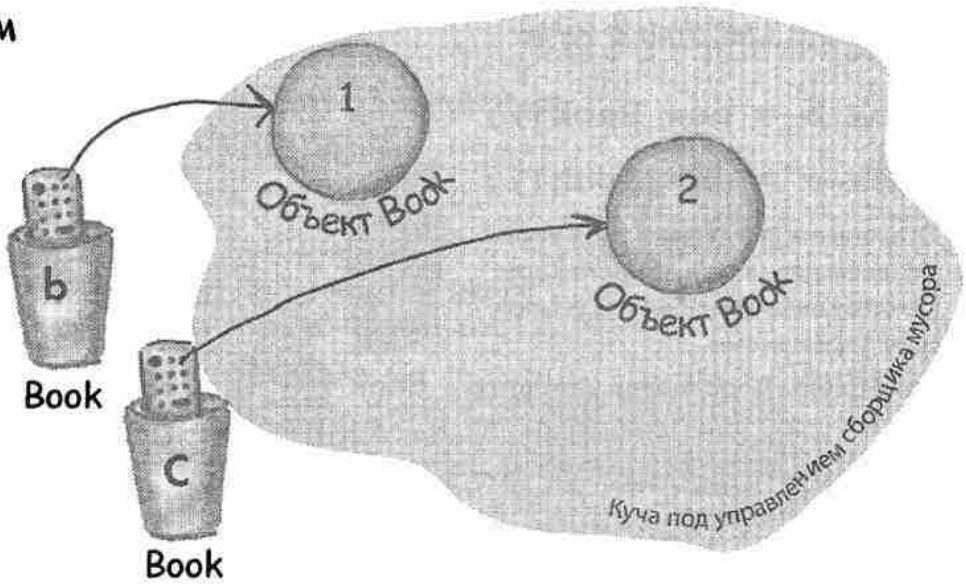
```
Book b = new Book();
Book c = new Book();
```

Объявляем две ссылки типа Book. Создаем два новых объекта Book. Присваиваем ссылочным переменным объекты типа Book.

Два объекта Book теперь находятся в куче.

Ссылок: 2.

Объектов: 2.




---

```
Book d = c;
```

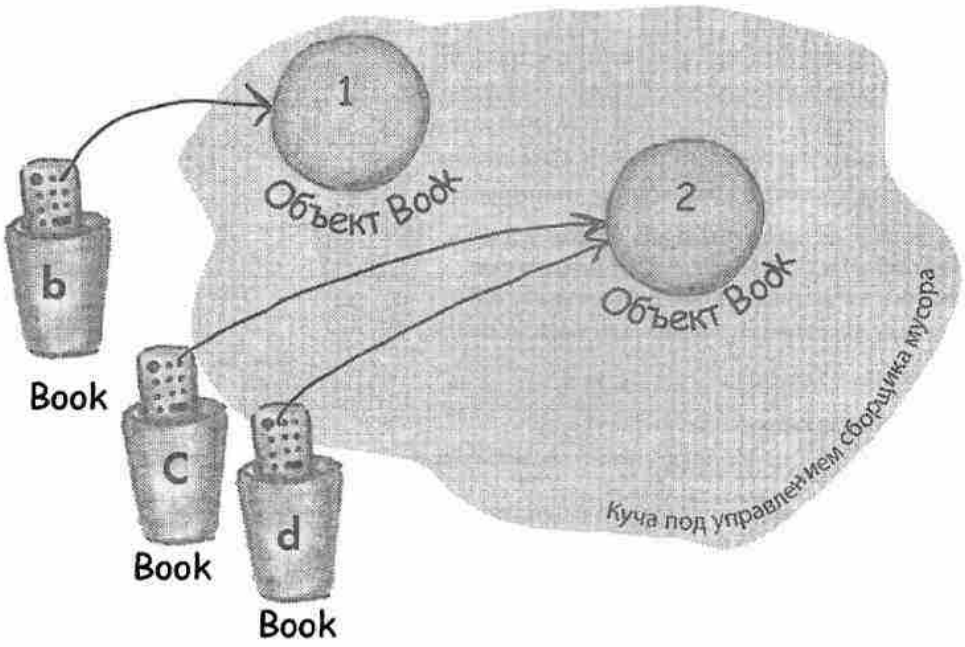
Объявляем новую ссылку типа Book. Вместо того чтобы создавать новый, третий по счету, объект Book, присваиваем ссылке значение переменной ***c***. Мы как бы говорим: «Возьми биты из ***c***, скопируй их и помести эту копию в ***d***».

**Переменные c и d ссылаются на один и тот же объект.**

**Ссылки c и d хранят две разные копии одного и того же значения. Два пульта управления для одного телевизора.**

Ссылок: 3.

Объектов: 2.




---

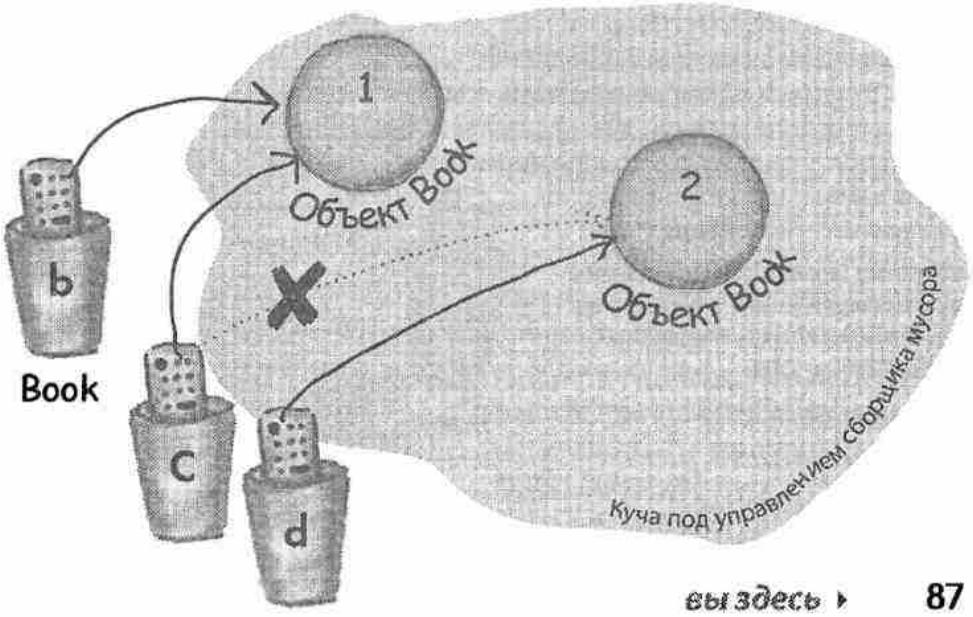
```
c = b;
```

Берем значение переменной ***b*** и присваиваем его переменной ***c***. Сейчас вам уже известно, что это означает. Биты внутри переменной ***b*** копируются, и новая копия помещается в переменную ***c***.

**И b, и c ссылаются на один и тот же объект.**

Ссылок: 3.

Объектов: 2.



## Жизнь и смерть в куче

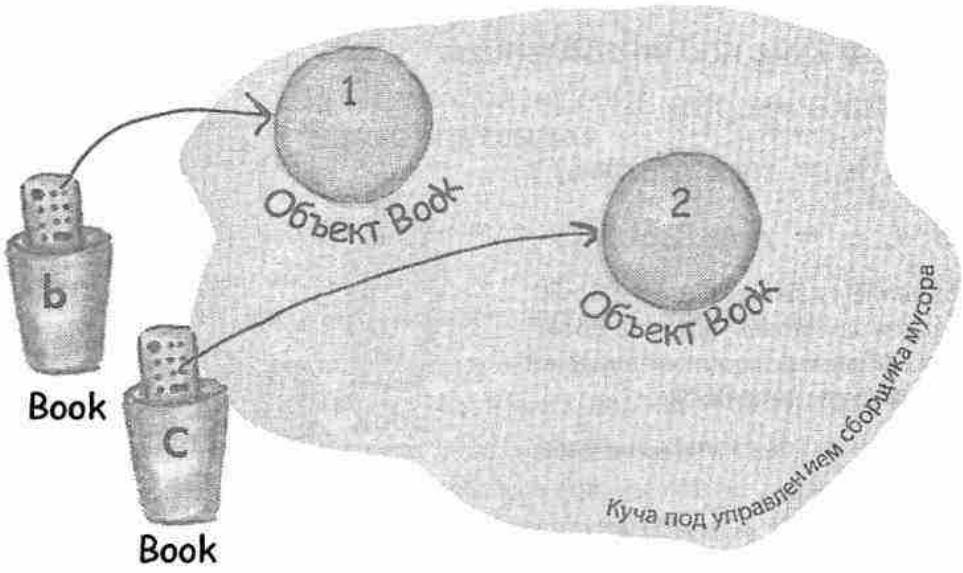
```
Book b = new Book();
Book c = new Book();
```

Объявляем две ссылки типа Book. Создаем два новых объекта Book. Присваиваем эти объекты ссылочным переменным.

Два объекта Book теперь находятся в куче.

Активных ссылок: 2.

Активных объектов: 2.




---

```
b = c;
```

Берем значение переменной ***c*** и присваиваем его переменной ***b***. Биты внутри переменной ***c*** копируются, и новая копия помещается в переменную ***b***. Обе переменные хранят одинаковые значения.

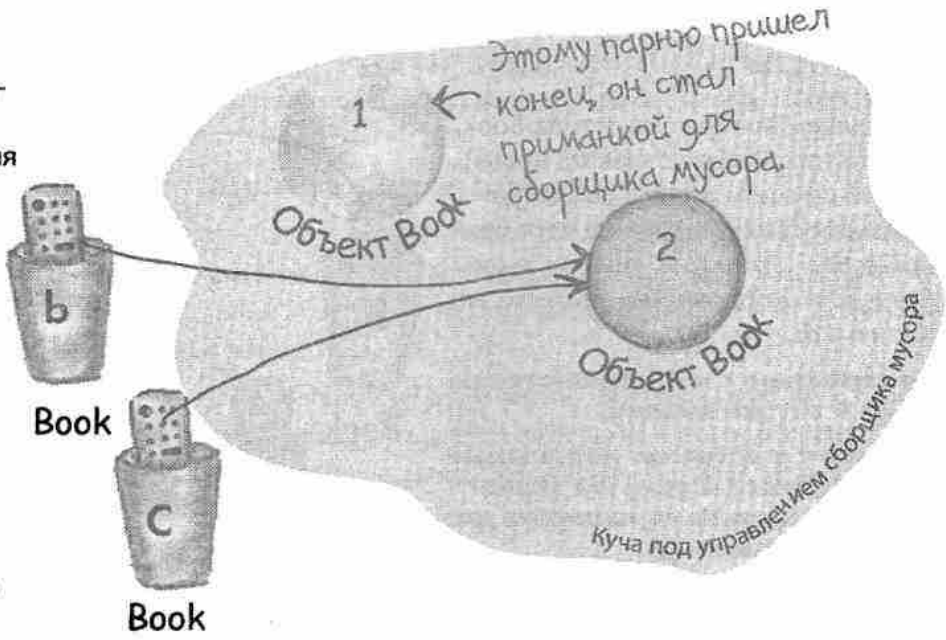
**И *b*, и *c* ссылаются на один и тот же объект. Объект 1 становится недоступным и пригоден для обработки сборщиком мусора (СМ).**

Активных ссылок: 2.

Доступных объектов: 1.

Недоступных объектов: 1.

Первый объект, на который ссылалась переменная ***b***, больше не имеет ссылок. Он *недоступен*.




---

```
c = null;
```

Присваиваем переменной ***c*** значение `null`. Это превращает ее в *нулевую ссылку*, то есть она больше ни на что не ссылается. Но это по-прежнему ссылочная переменная, и ей все еще можно присвоить другой объект типа Book.

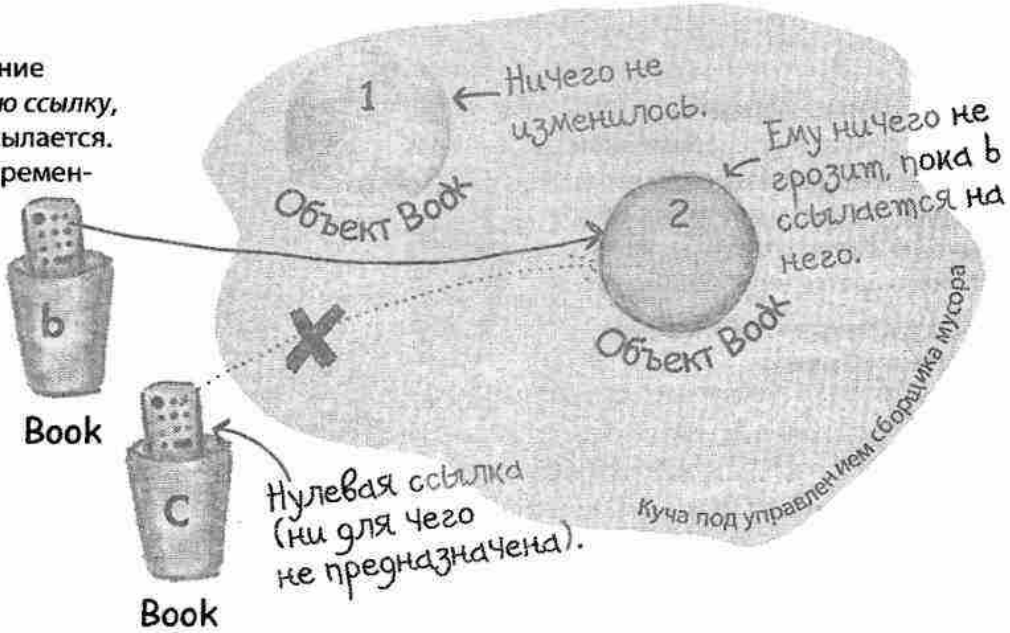
**Объект 2 еще имеет активную ссылку (b), значит, он не может быть отдан на обработку СМ.**

Активных ссылок: 1.

*Нулевых* ссылок: 1.

Доступных объектов: 1.

Недоступных объектов: 1.



# Массив как подставка для стаканов

1. Объявляем переменную целочисленного массива. Переменная массива — это пульт управления объектом Array.

```
int[] nums;
```

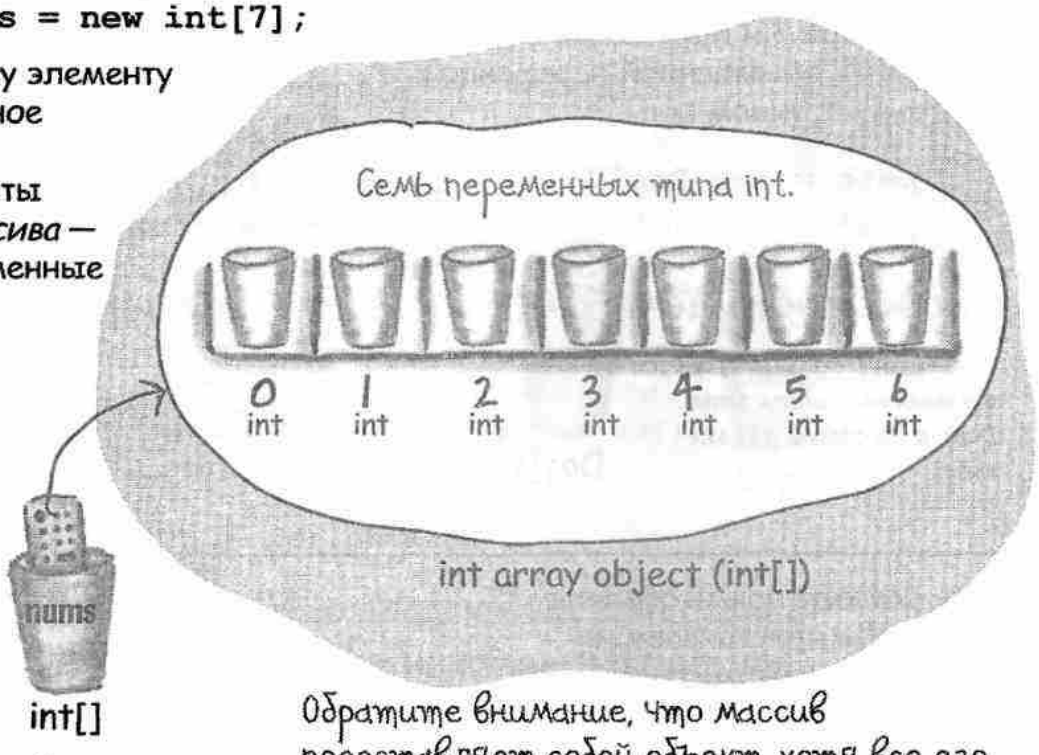
2. Создаем новый целочисленный массив с семью элементами и присваиваем его ранее объявленной переменной `nums` с типом `int[]`.

```
nums = new int[7];
```

3. Присваиваем каждому элементу массива целочисленное значение. Помните, что элементы целочисленного *массива* — это всего лишь переменные типа *int*.

Семь переменных типа int.

```
nums[0] = 6;
nums[1] = 19;
nums[2] = 44;
nums[3] = 42;
nums[4] = 10;
nums[5] = 20;
nums[6] = 1;
```




Обратите внимание, что массив представляет собой объект, хотя все его семь элементов имеют простой тип.

# Массивы — тоже объекты

Стандартная библиотека в Java содержит множество сложных структур данных, включая ассоциативные массивы, деревья и множества (см. Приложение Б). Массивы лучше всего подходят для создания упорядоченного, эффективного списка элементов. Они предоставляют быстрый произвольный доступ, позволяя использовать индекс (позицию) для получения элемента списка.

Каждый элемент в массиве — всего лишь переменная. Иначе говоря, это один из восьми простых типов, или ссылочная переменная. Все, что вы можете поместить в *переменную*, может быть присвоено *элементу массива* того же типа. В массиве типа int (int[]) каждый элемент имеет тип int. В массиве типа Dog (Dog[]) каждый элемент может хранить... объект Dog? Нет, вспомните, что ссылочные переменные хранят всего лишь ссылки (пульты управления), а не сами объекты. Поэтому в массиве типа Dog каждый элемент может хранить *пульт для управления* объектом Dog. Конечно, нам все еще нужно создать этот объект, и на следующей странице вы увидите, как это делается.

Рассматривая рисунок, приведенный выше, не упустите один ключевой момент: массив — *это всегда объект, даже если он хранит элементы простых типов.*

**У вас может быть массив, который объявлен для хранения простых значений.** Иными словами, объект массива может *содержать* элементы, которые имеют простой тип, но сам не может быть примитивом. Вне зависимости от содержимого массив всегда остается объектом!

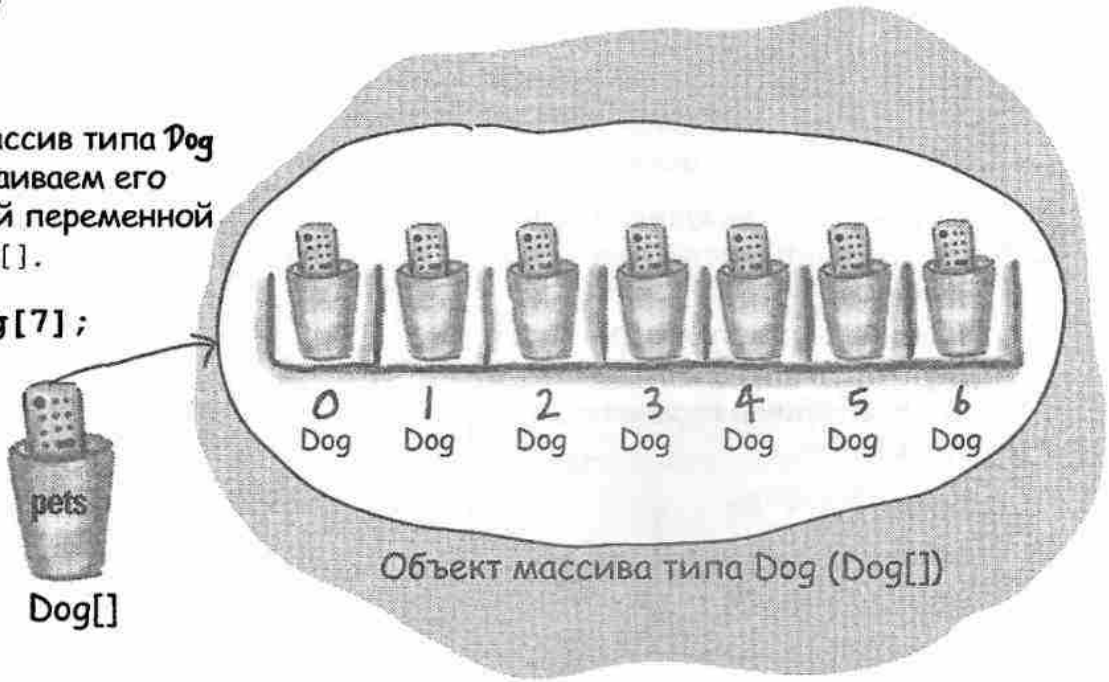
# Создадим массив объектов Dog

1. Объявляем переменную-массив Dog.

```
Dog[] pets;
```

2. Создаем новый массив типа Dog длиной 7 и присваиваем его ранее объявленной переменной pets с типом Dog[].

```
pets = new Dog[7];
```

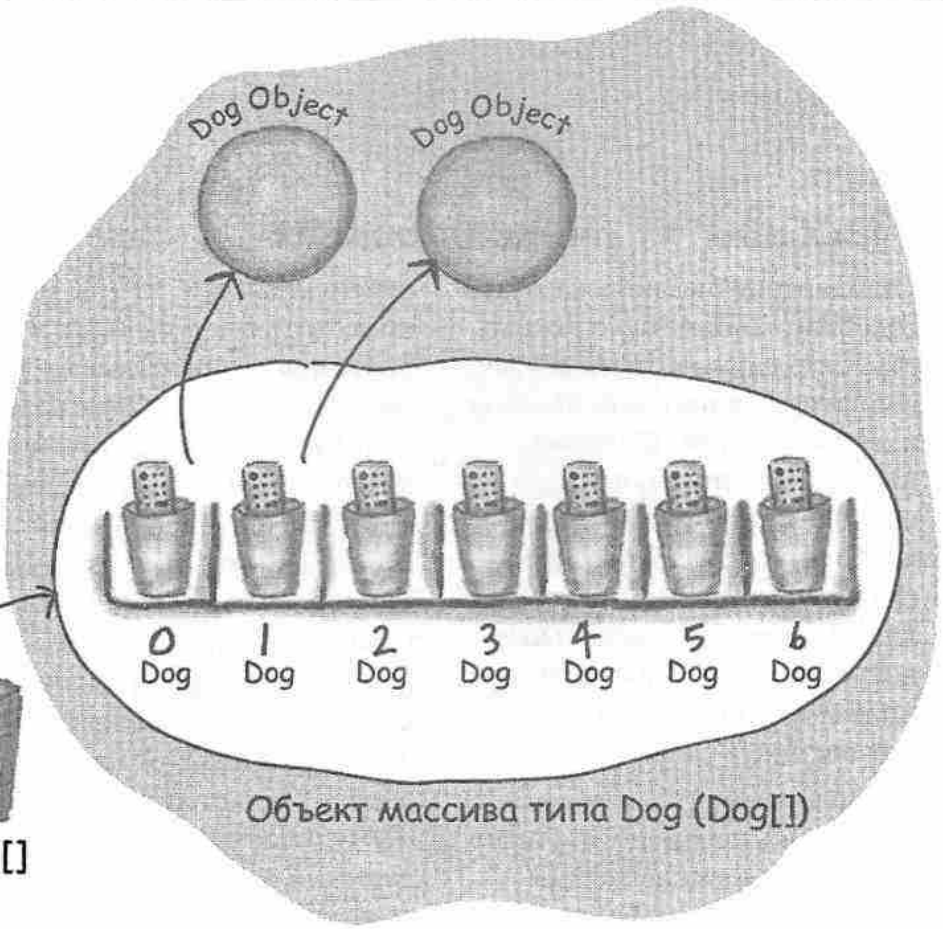



**Что мы упустили?**

**Объекты Dog! Мы имеем массив *ссылок* типа Dog, а не самих *объектов*!**

3. Создаем новые объекты Dog и присваиваем их элементам массива. Помните, элементы в массиве — это всего лишь ссылки типа Dog. Нам все еще нужны сами объекты!

```
pets[0] = new Dog();
pets[1] = new Dog();
```

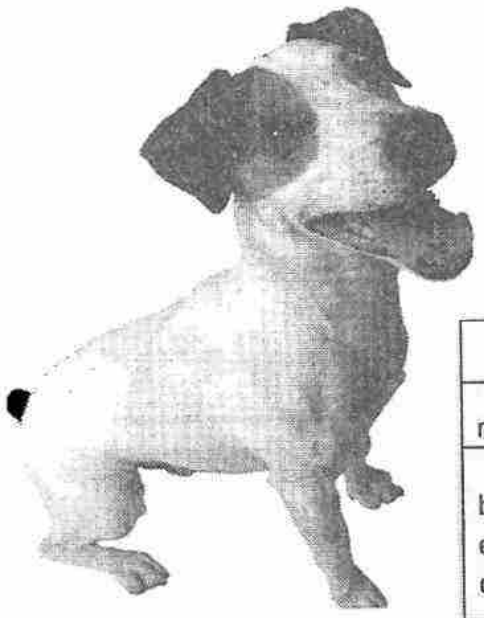


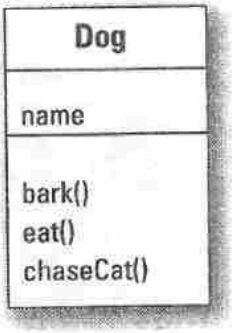
Наточите свой карандаш

Какое значение имеет элемент pets[2]? _____

С помощью каких команд можно заставить элемент pets[3] ссылаться на один из двух существующих объектов Dog?

_____

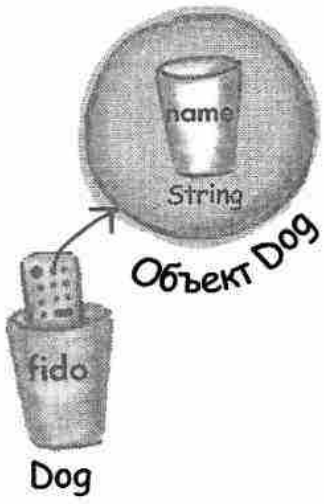





# Управляем объектом Dog с помощью ссылки

```
Dog fido = new Dog();
fido.name = "Фидо";
```

Мы создали объект Dog и использовали оператор доступа в сочетании со ссылочной переменной ***fido*** для получения доступа к переменной name.*




Можно задействовать ссылку ***fido*** для доступа к методам bark(), eat() или chaseCate().

```
fido.bark();
fido.chaseCat();
```

## Что происходит, если ссылка на Dog находится в массиве

Мы знаем, что можем получить доступ к переменным экземпляра и методам объекта Dog с помощью оператора доступа, *но как*?

Если Dog находится в массиве, у нас нет имени переменной (вроде ***fido***). Вместо этого мы используем обозначение массива и нажимаем кнопку на пульте (оператор «точка»), получая доступ к объекту по указанному индексу (позиции):

```
Dog[] myDogs = new Dog[3];
myDogs[0] = new Dog();
myDogs[0].name = "Фидо";
myDogs[0].bark();
```

*Да, мы знаем, что здесь не демонстрируются возможности инкапсуляции, но пытаемся рассказать все максимально просто. А инкапсуляцией мы займемся в главе 4.

## Java заботится о типах

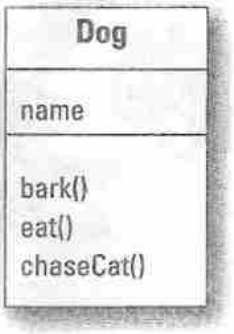
**Объявив массив, вы можете разместить в нем элементы только такого же типа, что и у самого массива.**

К примеру, вы не вправе поместить ссылку на объект Cat в массив типа Dog (пользователь решит, что в массиве могут находиться только ссылки типа Dog, попытается вызвать у каждого элемента метод bark и очень удивится, обнаружив затаившийся объект Cat). Нельзя также поместить переменную `double` в массив типа `int` (это переполнение, помните?). Однако можно поместить в такой массив переменную типа `byte`, так как этот тип всегда помещается в контейнере размера `int`. Это неявное приведение типов. Подробности мы рассмотрим позже, а пока запомните, что компилятор не позволит вам поместить неподходящий элемент в массив исходя из объявленного типа.

```
class Dog {
  String name;
  public static void main (String[] args) {
    // Создаем объект Dog и получаем к нему доступ
    Dog dog1 = new Dog();
    dog1.bark();
    dog1.name = "Барт";

    // Теперь создадим массив типа Dog
    Dog[] myDogs = new Dog[3];
    // и поместим в него несколько элементов
    myDogs[0] = new Dog();
    myDogs[1] = new Dog();
    myDogs[2] = dog1;

    // Теперь получаем доступ к объектам Dog
    // с помощью ссылок из массива
    myDogs[0].name = "Фред";
    myDogs[1].name = "Джордж";

    // Хммммм... какое имя у myDogs[2]?
    System.out.print("Имя последней собаки - ");
    System.out.println(myDogs[2].name);

    // Теперь переберем все элементы массива
    // и вызовем для каждого метод bark()
    int x = 0;
    while(x < myDogs.length) {
      myDogs[x].bark();
      x = x + 1;
    }
  }

  public void bark() {
    System.out.println(name + "сказал Гав!");
  }
  public void eat() { }
  public void chaseCat() { }
}
```

Массивы содержат переменную length, которая предоставляет количество хранимых элементов.

## Пример класса Dog




**Результат:**

```
File Edit Window Help Howl
%java Dog
null сказал Гав!
Имя последней собаки - Барт
Фред сказал Гав!
Джордж сказал Гав!
Барт сказал Гав!
```

### КЛЮЧЕВЫЕ МОМЕНТЫ

- Переменные разделяются на два вида: примитивы (простые типы) и ссылки.
- При объявлении переменных всегда нужно указывать имя и тип.
- Переменные простого типа содержат биты, представляющие значение (5, 'a', true, 3.1416 и т. д.).
- Ссылки включают биты, описывающие способ получить объект в куче.
- Ссылочная переменная похожа на дистанционный пульт управления. Используя оператор «точка» (.) в сочетании со ссылочной переменной, вы как бы нажимаете кнопки на пульте для получения доступа к методам или переменным экземпляра.
- Ссылочная переменная имеет значение `null`, если не ссылается ни на какой объект.
- Массив — это всегда объект, даже если он объявлен для хранения простых значений. Не бывает массивов простого типа, есть массивы, которые *хранят* примитивы.

# Поработайте компилятором

Каждый из Java-файлов на этой странице — полноценный исходник. Ваша задача — притвориться компилятором и определить, все ли файлы скомпилируются. Если компиляция не сможет пройти успешно, как вы их исправите?

**A**

```
class Books {
  String title;
  String author;
}

class BooksTestDrive {
  public static void main(String [] args) {

    Books [] myBooks = new Books[3];
    int x = 0;
    myBooks[0].title = "Плоды Java";
    myBooks[1].title = "Java Гэтсби";
    myBooks[2].title = "Сборник рецептов
                                    по Java";
    myBooks[0].author = "Боб";
    myBooks[1].author = "Сью";
    myBooks[2].author = "Ян";

    while (x < 3) {
      System.out.print(myBooks[x].title);
      System.out.print(", автор");
      System.out.println(myBooks[x].author);
      x = x + 1;
    }
  }
}
```

**B**

```
class Hobbits {

  String name;

  public static void main(String [] args) {

    Hobbits [] h = new Hobbits[3];
    int z = 0;

    while (z < 4) {
      z = z + 1;
      h[z] = new Hobbits();
      h[z].name = "Бильбо";
      if (z == 1) {
        h[z].name = "Фродо";
      }
      if (z == 2) {
        h[z].name = "Сэм";
      }
      System.out.print(h[z].name + " - ");
      System.out.println("имя хорошего
                                    хоббита");
    }
  }
}
```

# Магнитики с кодом

Части рабочего Java-приложения разбросаны по всему холодильнику. Можете ли вы восстановить из фрагментов работоспособную программу, которая выведет на экран текст, приведенный ниже? Некоторые фигурные скобки упали на пол; они настолько маленькие, что их нельзя поднять. Можете добавлять столько скобок, сколько понадобится.

```
File  Edit  Window  Help  Bikini
% java TestArrays
Фиджи
Косумель
Бермуды
Азорские острова
```

```
int y = 0;
```

```
ref = index[y];
```

```
islands[0] = "Бермуды";
islands[1] = "Фиджи";
islands[2] = "Азорские острова";
islands[3] = "Косумель";
```

```
int ref;
while (y < 4) {
```

```
System.out.println(islands[ref]);
```

```
index[0] = 1;
index[1] = 3;
index[2] = 0;
index[3] = 2;
```

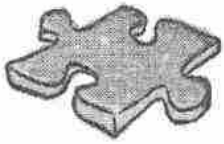
```
String [] islands = new String[4];
```

```
System.out.print("острова = ");
```

```
int [] index = new int[4];
```

```
y = y + 1;
```

```
class TestArrays {

  public static void main(String [] args) {
```



# Головоломка у бассейна

Ваша **задача** — взять фрагменты кода со дна бассейна и заменить ими пропущенные участки программы. **Можно** использовать один фрагмент несколько раз, но не все из них вам пригодятся. Ваша **цель** — создать класс, который скомпилируется, запустится и выведет приведенный ниже текст.

**Результат:**

```
File Edit Window Help Bermuda
%java Triangle
треугольник 0, зона = 4.0
треугольник 1, зона = 10.0
треугольник 2, зона = 18.0
треугольник 3, зона = _____
y = _____________________________
```

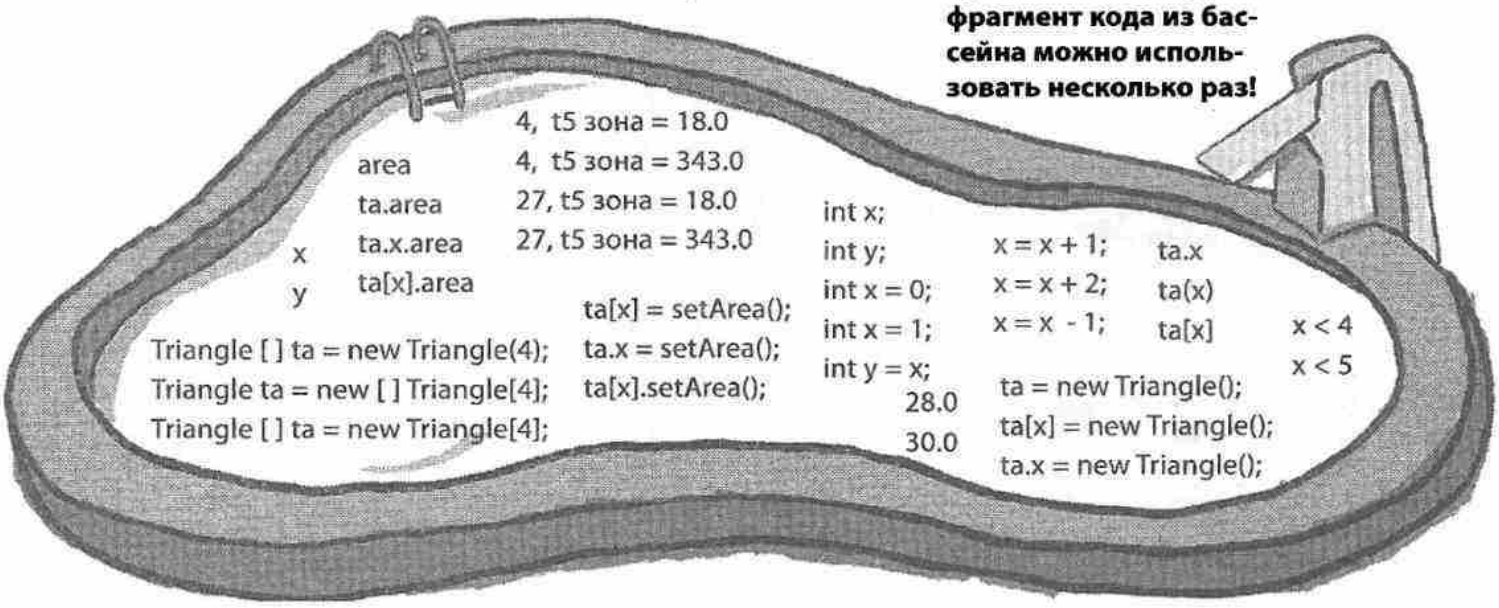
**Призовой вопрос!**

Если хотите получить призовые баллы, используйте фрагменты из бассейна, чтобы заполнить пропуски в показанном выше результате работы программы.

```
class Triangle {
  double area;
  int height;
  int length;
  public static void main(String [] args) {

    ____________

    ____________________

    while ( __________ ) {

      ____________________

      ________.height = (x + 1) * 2;

      ________.length = x + 4;

      ____________________

     System.out.print("треугольник "+x+", зона");

      System.out.println(" = " + ________.area);

      ____________

    }

    ____________

    x = 27;

    Triangle t5 = ta[2];

    ta[2].area = 343;

    System.out.print("y = " + y);

    System.out.println(", зона t5 = "+ t5.area);

  }

  void setArea() {

    ____________ = (height * length) / 2;

  }

}
```

Иногда мы пренебрегаем отдельным классом для тестирования, чтобы сохранить место на странице.

**Примечание: каждый фрагмент кода из бассейна можно использовать несколько раз!**



# Куча проблем

Справа представлена небольшая Java-программа. В коде до строки с комментарием создаются объекты и ссылочные переменные. Ваша задача — определить, какая переменная ссылается на какой объект. Не все ссылки будут использованы, а на некоторые объекты нужно ссылаться несколько раз. Нарисуйте линии, соединяющие ссылки с соответствующими объектами.

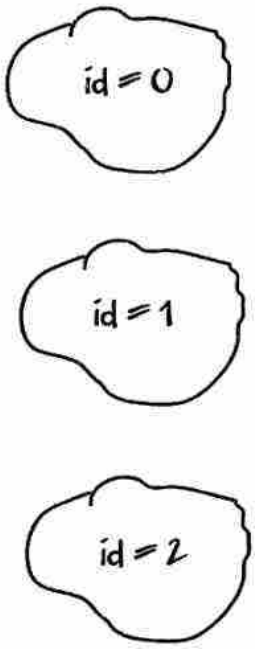
***Подсказка:*** вероятно, вам придется рисовать, как на с. 87 и 88 (хотя, может, вам это не нужно). Вы можете рисовать карандашом и затем стирать ластиком линии, соединяющие ссылку (пульт управления) с объектом.

```
class HeapQuiz {
  int id = 0;
  public static void main(String [] args)
    int x = 0;
    HeapQuiz [ ] hq = new HeapQuiz[5];
    while ( x < 3 ) {
      hq[x] = new HeapQuiz();
      hq[x].id = x;
      x = x + 1;
    }
    hq[3] = hq[1];
    hq[4] = hq[1];
    hq[3] = null;
    hq[4] = hq[0];
    hq[0] = hq[3];
    hq[3] = hq[2];
    hq[2] = hq[0];
    // Делаем что-то
  }
}
```

Соедините каждую ссылку с соответствующим объектом или объектами. Возможно, вам понадобятся не все ссылки.

**Ссылочные переменные:**

**Объекты HeapQuiz:**



## Дело о похищении ссылок

Темная ночь. Шторм. Тавни расхаживала по офису, в котором работали программисты. Она знала, что все разработчики очень заняты, и ей хотелось им помочь. Нужно было добавить новый метод в главный класс, который необходимо загрузить на новый совершенно секретный мобильный телефон с поддержкой Java, разрабатываемый для клиента. Динамической памяти (кучи) в телефоне чрезвычайно мало — об этом знали все. Привычный офисный гул быстро стих, как только Тавни подошла к доске для рисования схем. Она кратко описала возможности метода и обвела взглядом всех своих подчиненных. «Ну что ж, парни, пришло время истины, — промурлыкала она себе под нос. — Кто сможет придумать наименее требовательный к памяти вариант этого метода, того я возьму с собой на вечеринку в Майами, которую наш клиент устраивает завтра. Придется помочь мне установить новое программное обеспечение».

Пятиминутный детектив

На следующее утро Тавни пришла в офис в своем коротком гавайском платье. «Господа, — улыбнулась она, — самолет вылетает через несколько часов, покажите же мне результаты ваших стараний!» Первым вышел Боб. Как только он начал изображать свои идеи на доске, Тавни сказала: «Ближе к делу, Боб, покажи мне, как ты управляешь обновлением списка объектов с контактами». Боб быстро набросал фрагмент кода на доске:

```
Contact [] ca = new Contact[10];
while ( x < 10 ) {   // Создаем 10 объектов Contact
  ca[x] = new Contact();
  x = x + 1;
}
// Делаем сложное обновление списка
// с объектами Contact с помощью массива ca
```

«Тавни, я понимаю, что у нас ограниченный размер памяти, но в твоем задании говорится, что мы должны иметь возможность работать с информацией каждого из десяти допустимых контактов. Это лучшее, что я смог придумать», — сказал Боб. Кент, вышедший следующим, уже представлял, как будет пить кокосовые коктейли вместе с Тавни. «Боб, — сказал он, — твое решение слегка нелепое, не находишь?» И с ехидной улыбкой добавил: «Взгляни на это, крошка».

```
Contact refc;
while ( x < 10 ) {   // Создаем десять объектов Contact
  refc = new Contact();
  x = x + 1;
}
// Делаем сложное обновление списка
// с объектами Contact с помощью переменной refc
```

«Я сохранил ценную память, сэкономив на ссылочных переменных, Боб, так что можешь спрятать свои солнцезащитные очки», — насмешливо сказал Кент. «Попридержи лошадей, Кент! — ответила Тавни. — Ты сохранил немного памяти, это правда. Но со мной едет Боб».

***Почему Тавни предпочла метод Боба, хотя Кент использовал меньше памяти?***

# Ответы

## Магнитики с кодом

```
class TestArrays {
  public static void main(String [] args) {
    int [] index = new int[4];
    index[0] = 1;
    index[1] = 3;
    index[2] = 0;
    index[3] = 2;
    String [] islands = new String[4];
    islands[0] = "Бермуды";
    islands[1] = "Фиджи";
    islands[2] = "Азорские острова";
    islands[3] = "Косумель";
    int y = 0;
    int ref;
    while (y < 4) {
      ref = index[y];
      System.out.print("острова = ");
      System.out.println(islands[ref]);
      y = y + 1;
    }
  }
}
```

```
File Edit Window Help Bikini
% java TestArrays
острова = Фиджи
острова = Косумель
острова = Бермуды
острова = Азорские острова
```

## Поработайте компилятором

```
class Books {
  String title;
  String author;
}
class BooksTestDrive {
  public static void main(String [] args) {
    Books [] myBooks = new Books[3];
    int x = 0;
A   myBooks[0] = new Books();
    myBooks[1] = new Books();
    myBooks[2] = new Books();
    myBooks[0].title = "Плоды Java";
    myBooks[1].title = "Java Гэтсби";
    myBooks[2].title = "Сборник рецептов по Java";
    myBooks[0].author = "Боб";
    myBooks[1].author = "Сью";
    myBooks[2].author = "Ян";
    while (x < 3) {
      System.out.print(myBooks[x].title);
      System.out.print(" by ");
      System.out.println(myBooks[x].author);
      x = x + 1;
    }
  }
}
```

Помните: на самом деле мы должны создать объекты Book!

---

```
class Hobbits {
  String name;
  public static void main(String [] args) {
    Hobbits [] h = new Hobbits[3];
    int z = -1;
    while (z < 2) {
      z = z + 1;
      h[z] = new Hobbits();
B     h[z].name = "Бильбо";
      if (z == 1) {
        h[z].name = "Фродо";
      }
      if (z == 2) {
        h[z].name = "Сэм";
      }
      System.out.print(h[z].name + " - ");
      System.out.println("имя хорошего хоббита");
    }
  }
}
```

Помните: массив начинается с нулевого элемента!

# Ответы

## Головоломка у бассейна

```
class Triangle {
  double area;
  int height;
  int length;
  public static void main(String [] args) {
    int x = 0;
    Triangle [ ] ta = new Triangle[4];
    while ( x < 4 ) {
      ta[x] = new Triangle();
      ta[x].height = (x + 1) * 2;
      ta[x].length = x + 4;
      ta[x].setArea();
      System.out.print("треугольник "+x+", зона");
      System.out.println(" = " + ta[x].area);
      x = x + 1;
    }
    int y = x;
    x = 27;
    Triangle t5 = ta[2];
    ta[2].area = 343;
    System.out.print(" y = " + y);
    System.out.println(", зона t5 = " + t5.area);
  }
  void setArea() {
    area = (height * length) / 2;
  }
}
```

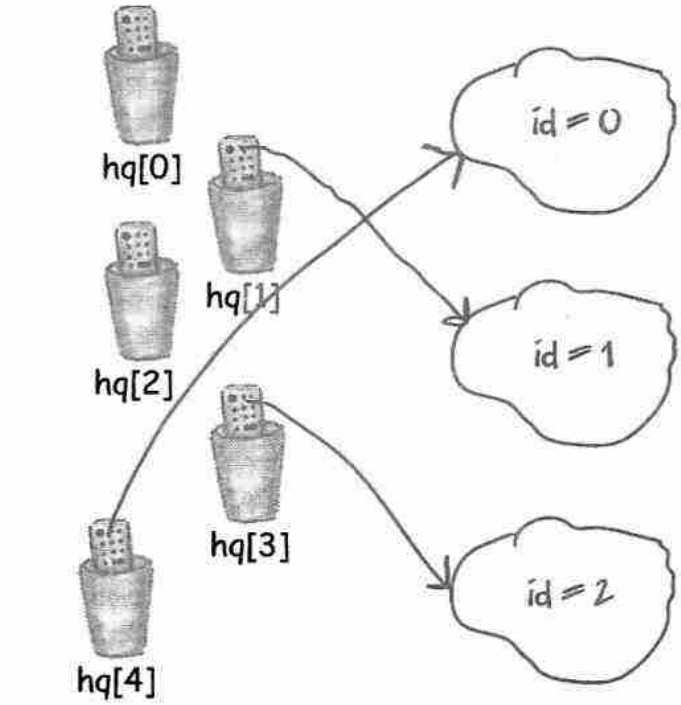
```
File Edit Window Help Bermuda
%java Triangle
треугольник 0, зона = 4.0
треугольник 1, зона = 10.0
треугольник 2, зона = 18.0
треугольник 3, зона = 28.0
y = 4, t5 зона = 343
```

## Разгадка дела о похищении ссылок

Тавни смогла увидеть в методе Кента серьезный недостаток. Он на самом деле использовал меньше ссылочных переменных, чем Боб, но у него не было возможности получить доступ ни к одному объекту контактов, кроме последнего созданного. При каждой итерации цикла он присваивал новый объект одной и той же переменной, а объект, на который эта переменная ссылалась ранее, становился *недоступен*. Не имея шансов получить доступ к девяти из десяти созданных объектов, метод Кента оказался бесполезным.

(Разработанную программу ждал большой успех, а клиент выдал Тавни и Бобу дополнительную неделю отпуска на Гавайях. И мы рады сообщить, что вы получите нечто похожее, прочитав эту книгу.)

## Куча проблем



# Как себя ведут объекты

## Состояние влияет на поведение, а поведение — на состояние.

Вы знаете, что объекты характеризуются **состоянием** и **поведением**, которые представлены **переменными экземпляра** и **методами**. До этого момента вопрос о связи между состоянием и поведением не поднимался. Вам уже известно, что каждый экземпляр класса (каждый объект определенного типа) может иметь уникальные значения для своих переменных экземпляра. У первого объекта типа Dog есть имя (переменная *name*) Фидо и вес (переменная *weight*) 30 килограммов. Второй объект этого типа носит имя Киллер и весит 4 килограмма. И если класс Dog содержит метод makeNoise(), будет ли 4-килограммовая собака лаять более убедительно, чем 30-килограммовая (при условии, что это тявканье можно назвать лаем)? К счастью, в этом и состоит суть объектов — их поведение зависит от состояния. Иными словами, ***методы используют значения переменных экземпляра***. Можно сказать что-то вроде «если собака весит меньше 6 килограммов, издаем тявканье, иначе...» или «увеличиваем вес на 2». ***В этой главе вы узнаете, как изменить состояние объекта.***

# Помните: класс описывает, что объект знает и делает

**Класс — это шаблон для объекта.** Разрабатывая класс, вы описываете, как JVM должна создать объект указанного типа. Вам уже известно, что каждый объект определенного типа может иметь разные значения *переменных экземпляра*. Но как насчет методов?

## Может ли метод каждого из объектов определенного типа иметь собственное поведение?

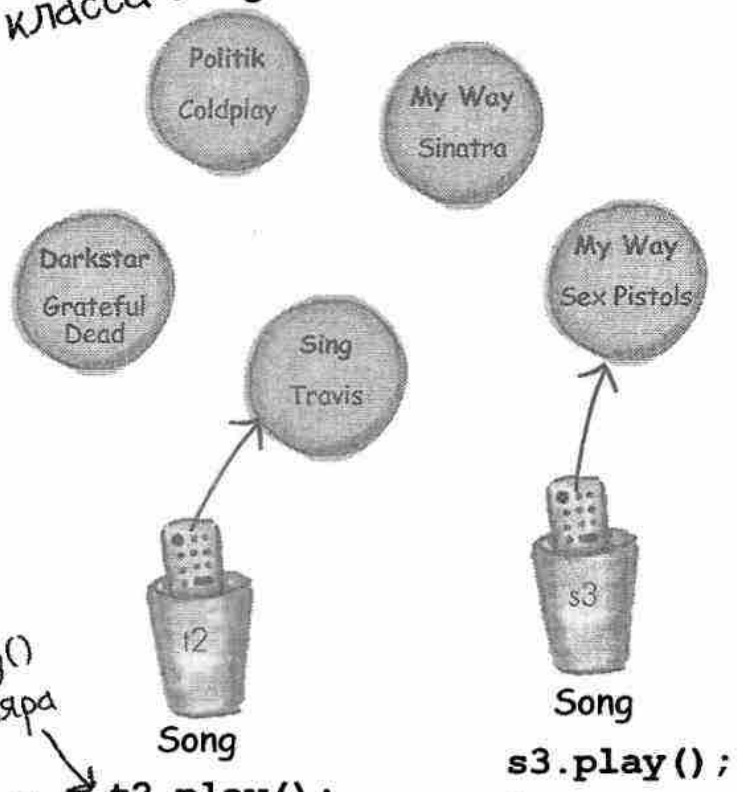
Ну... **что-то вроде того.***

Все экземпляры конкретного класса имеют одни и те же методы, которые могут *вести себя* по-разному, в зависимости от значений переменных экземпляра.

Класс Song содержит два поля — *title* и *artist*. Метод play() проигрывает композицию, выбор которой будет зависеть от поля *title* для конкретного экземпляра. Вызывая метод play() из разных экземпляров, вы можете услышать различные композиции — «Politik», «Darkstar» или другую. Однако код метода остается прежним.

```
void play() {
    soundPlayer.playSound(title);
}
```

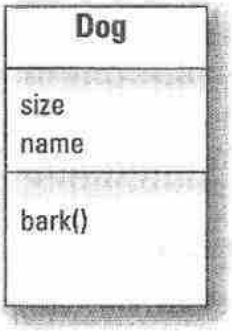
```
Song t2 = new Song();
t2.setArtist("Travis");
t2.setTitle("Sing");
Song s3 = new Song();
s3.setArtist("Sex Pistols");
s3.setTitle("My Way");
```



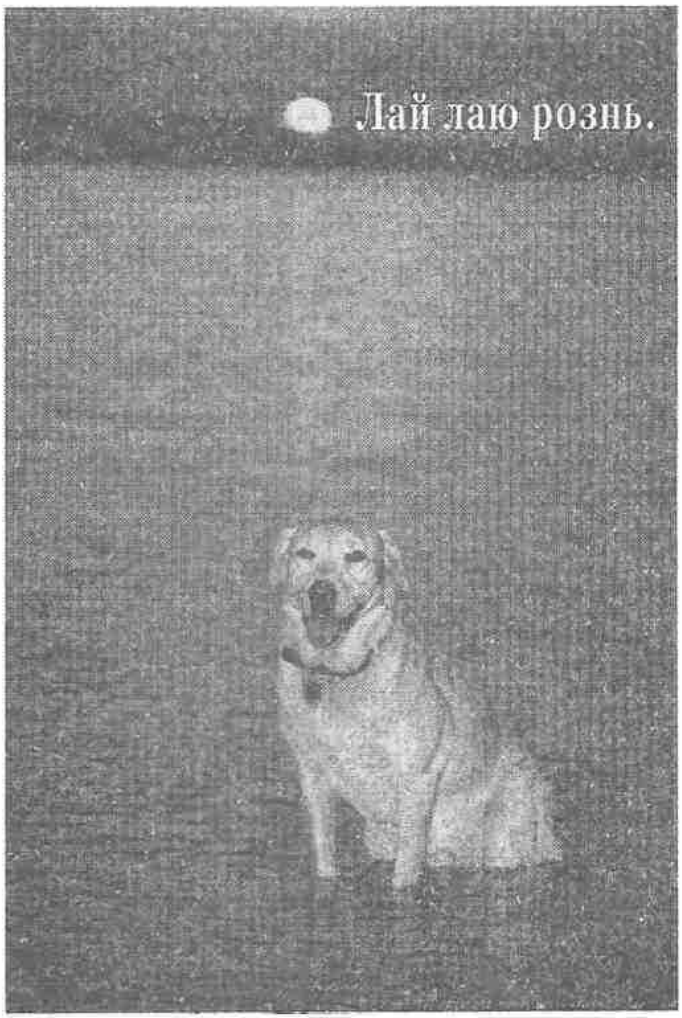

*Еще один поразительно четкий ответ!

# Размер влияет на громкость лая

Маленькие собаки лают не так, как большие. Класс Dog содержит переменную экземпляра size, которая используется методом bark() для определения вида лая.




```
class Dog {
  int size;
  String name;

  void bark() {
    if (size > 60) {
      System.out.println("Гав Гав!");
    } else if (size > 14) {
      System.out.println("Вуф Вуф!");
    } else {
      System.out.println("Тяф Тяф!");
    }
  }
}
```




---

```
class DogTestDrive {

  public static void main (String[] args) {
    Dog one = new Dog();
    one.size = 70;
    Dog two = new Dog();
    two.size = 8;
    Dog three = new Dog();
    three.size = 35;

    one.bark();
    two.bark();
    three.bark();
  }
}
```

```
File Edit Window Help Playdead
%java DogTestDrive
Гав Гав!
Тяф Тяф!
Вуф Вуф!
```

# Вы можете передавать методу разные значения

Как и в любом другом языке программирования, в Java вы можете передавать значения своим методам. К примеру, укажите объекту Dog, сколько именно раз нужно пролаять:

```
d.bark(3);
```

В зависимости от опыта и личных предпочтений для обозначения передаваемых в метод значений вы можете использовать два термина — *аргументы* или *параметры*. Хотя с точки зрения компьютерных наук эти понятия не тождественны, мы не будем отвлекаться на такие мелочи. Можете называть их, как вам вздумается (аргументами, пончиками, комками шерсти и т. д.), но мы будем придерживаться следующих правил.

**Метод использует параметры. Вызывающий код передает аргументы.**

Аргументы — это значения, которые вы передаете методам. ***Аргумент*** (значение наподобие 2, "Foo" или ссылки на объект Dog) превращается в ***параметр***. В свою очередь, это не что иное, как локальная переменная, то есть переменная с именем и типом, которая может быть использована внутри метода.

Но есть один важный момент: **если метод принимает параметр, вы обязаны ему что-нибудь передать.** При этом передаваемое значение должно быть соответствующего типа.

# Вы можете получать значения обратно из метода

Методы могут возвращать значения. Каждый метод объявляется с указанием типа возвращаемого значения, но до сих пор в этом качестве мы использовали только тип void; то есть метод ничего не отдавал.

```
void go() {
}
```

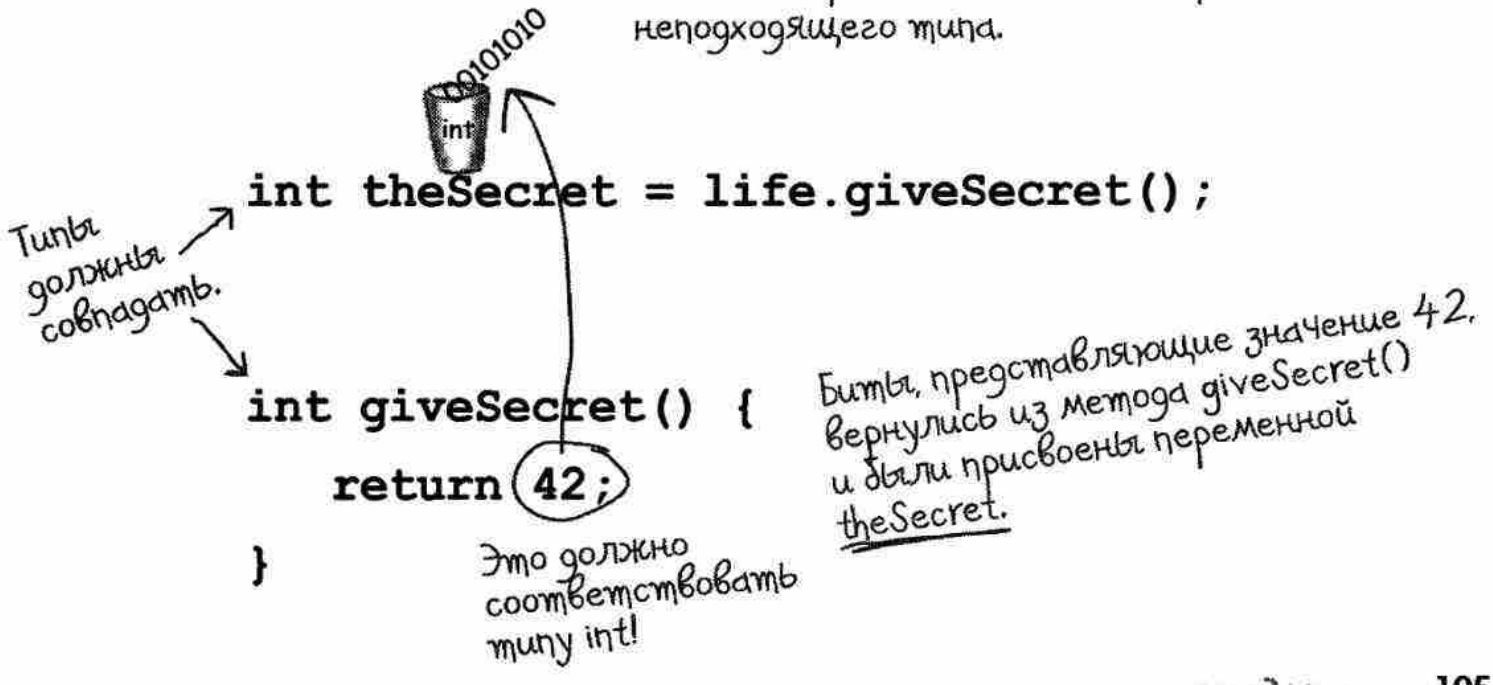
Можно объявить метод, указав конкретный тип значения, возвращаемого вызывающему коду:

```
int giveSecret() {
   return 42;
}
```

Если вы объявили метод, который возвращает значение, то *обязаны* вернуть значение указанного типа (или *совместимое* с ним). Мы вернемся к этому в главах 7 и 8, когда будем изучать полиморфизм.

## Что бы вы ни *пообещали*, вам *лучше* это вернуть!

Компилятор не позволит вам вернуть значение неподходящего типа.

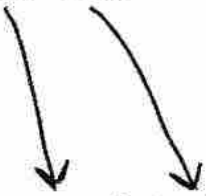


# Вы можете передать в метод сразу несколько значений

Методы могут содержать несколько параметров. Объявляемые параметры и передаваемые аргументы нужно разделять запятыми. Что еще более важно, если метод имеет параметры, то *необходимо* передавать ему аргументы соответствующего типа и в правильном порядке.

## Вызов метода с двумя параметрами и передача ему двух аргументов.

```
void go() {
    TestStuff t = new TestStuff();
    t.takeTwo(12, 34);
}

void takeTwo(int x, int y) {
    int z = x + y;
    System.out.println("Сумма равна " + z);
}
```



Аргументы передаются в том порядке, в котором принимаются параметры. Первый аргумент передается в первый параметр, второй аргумент — во второй параметр и т. д.

## Можно передавать переменные в метод, если их типы совпадают с типами параметров.

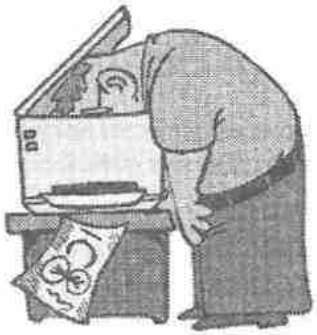
```
void go() {
    int foo = 7;
    int bar = 3;
    t.takeTwo(foo, bar);
}

void takeTwo(int x, int y) {
    int z = x + y;
    System.out.println("Total is " + z);
}
```

Значения переменных foo и bar находятся в параметрах x и y, поэтому биты в переменной x идентичны битам в foo (битовая последовательность, представляющая число 7), а биты в y идентичны битам в bar.

Какое значение хранит переменная z? Результат будет таким же, как от сложения переменных foo и bar на момент их передачи в метод takeTwo.

# В Java все передается по значению.

## Имеется в виду, что значение копируется при передаче

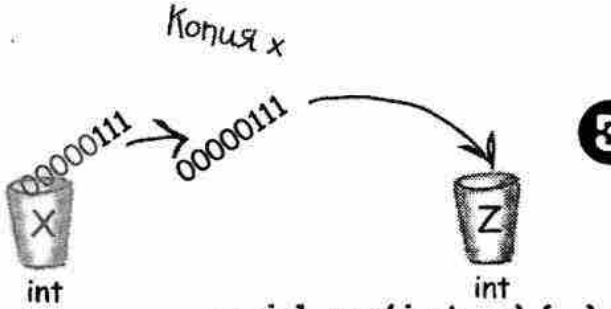


```
int x = 7;
```

1 Объявляем целочисленную переменную и присваиваем ей значение 7. Последовательность битов для числа 7 отправляется в переменную x.

```
void go(int z){ }
```

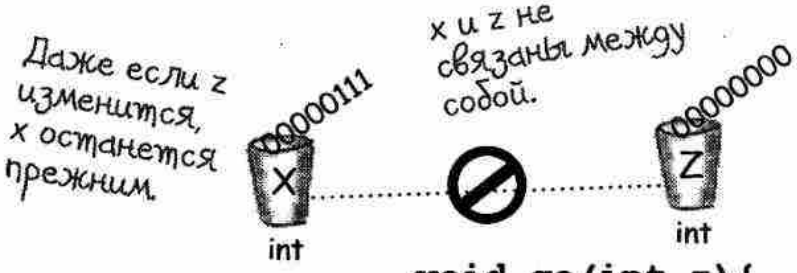
2 Объявляем метод с целочисленным параметром z.




```
foo.go(x);
```

```
void go(int z){ }
```

3 Вызываем метод go(), передавая ему в качестве аргумента переменную x. Биты из x копируются и помещаются в z.




```
void go(int z){
    z = 0;
}
```

4 Изменяем значение z внутри метода. Оно не изменилось! Аргумент, ставший параметром z, — это всего лишь копия x.

Метод не может изменить биты, хранящиеся в программе (в виде переменной x), из которой он был вызван.

## Это не глупые вопросы

**В: Что происходит, если аргумент, который вы хотите передать, представляет собой объект, а не примитив?**

О: Это мы рассмотрим в следующих главах, но ответ вам уже известен. В Java *все* передается по значению. Все. Но... *значение* — это *биты внутри переменной*. Помните, что вы не помещаете объекты в переменные; переменная — это дистанционный пульт управления, *или ссылка на объект*. Таким образом, если вы передаете в метод ссылку на объект, то тем самым передаете *копию пульта для управления* объектом. Оставайтесь с нами, и вы узнаете еще много интересного на эту тему.

**В: Может ли метод объявить несколько возвращаемых значений? Есть ли способ вернуть их одновременно?**

О: Метод может объявить только одно возвращаемое значение. Но если нужно вернуть, скажем, три целочисленных значения, это можно сделать с помощью *массива*. Поместите числа в массив и верните их. С возвращением значений разного типа все обстоит несколько сложнее — об этом мы поговорим в следующей главе, когда речь пойдет об ArrayList.

**В: Должен ли я возвращать тот самый тип, который указал при объявлении?**

О: Разрешено вернуть любое значение, которое можно *неявно* привести к этому типу. Поэтому вы можете вернуть byte там, где ожидается int. Вызывающему коду все равно, так как значения byte отлично согласуются с типом int, который применяется для присваивания результата. Если объявленный тип *меньше*, чем тот, который вы пытаетесь вернуть, необходимо использовать *явное* приведение.

**В: Обязательно ли что-то делать со значением, которое возвращает метод? Можно ли его просто проигнорировать?**

О: Java не требует, чтобы вы подтверждали получение возвращаемого значения. Возможно, вам придется вызвать метод, который что-то возвращает, даже если вас не интересует его значение. В этом случае метод вызывается ради работы, которую он проделывает внутри себя, а не ради *возвращаемого* значения. В языке Java не обязательно присваивать или использовать возвращенное значение.

## Напоминание: Java заботится о типах!

**Нельзя вернуть объект Giraffe (жираф), если объявленный тип возвращаемого значения — Rabbit (кролик). Тот же принцип работает с параметрами. Вы не можете передать Giraffe в метод, который принимает объекты типа Rabbit.**

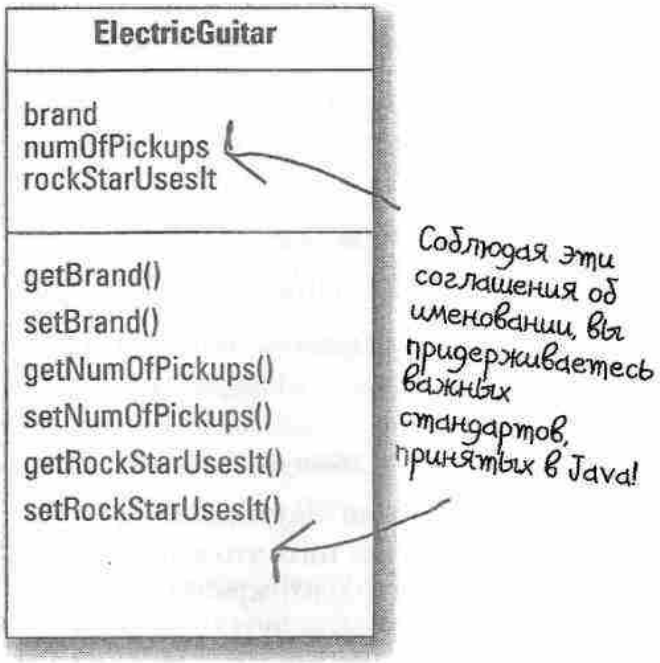
### КЛЮЧЕВЫЕ МОМЕНТЫ

- Классы определяют, что объект знает и что он умеет.
- Информация, которой объект владеет, — это **переменные экземпляра** (состояние).
- Действия, которые объект может выполнять, — это его **методы** (поведение).
- Методы могут использовать значения переменных экземпляра, поэтому объекты одного типа могут вести себя по-разному.
- Методы могут содержать параметры, то есть разрешено передавать им одно или несколько значений.
- Количество и тип передаваемых значений должны соответствовать параметрам, объявленным для метода (включая порядок их следования).
- Значения, принимаемые и возвращаемые методом, могут быть неявно приведены к более вместительному типу (или явно приведены к менее вместительному).
- Значения, передаваемые в метод в качестве аргументов, могут быть литералами (2, 'c' и т. д.) или переменными тех типов, которые были объявлены для параметров (например, переменная x, которая имеет тип int). Есть и другие объекты, которые можно передавать в качестве аргументов, но об этом мы поговорим позже.
- Для метода должен быть объявлен тип возвращаемого значения. Тип void говорит о том, что метод ничего не возвращает.
- Если объявленный для метода тип возвращаемого значения не равен void, то этот метод должен вернуть значение, совместимое с указанным типом.

# Разные трюки с параметрами и возвращаемыми значениями

Теперь, когда вы увидели, как работают параметры и возвращаемые значения, пришло время найти им достойное применение: рассмотрим **геттеры** и **сеттеры**. В других источниках встречаются названия *accessors* и *mutators*. Можете называть их так, если угодно, но именно термины «геттеры» и «сеттеры» хорошо вписываются в соглашение по именованию, принятое в Java, поэтому мы будем использовать эти слова.

Геттеры и сеттеры позволяют *получать и устанавливать значения*. Как правило, это значения переменных экземпляра. Главная задача геттера — вернуть значение (по аналогии с возвращаемым значением). Вероятно, вы уже не удивитесь тому, что сеттеры принимают аргумент и *устанавливают* его значение для переменной экземпляра.




```
class ElectricGuitar {

   String brand;
   int numOfPickups;
   boolean rockStarUsesIt;

   String getBrand() {
      return brand;
   }

   void setBrand(String aBrand) {
      brand = aBrand;
   }

   int getNumOfPickups() {
      return numOfPickups;
   }

   void setNumOfPickups(int num) {
      numOfPickups = num;
   }

   boolean getRockStarUsesIt() {
      return rockStarUsesIt;
   }

   void setRockStarUsesIt(boolean yesOrNo) {
      rockStarUsesIt = yesOrNo;
   }
}
```

# Инкапсуляция

## Игнорируя ее, вы рискуете быть осмеянными

До этого знаменательного момента мы придерживались одного из самых неестественных стилей в ООП.

В чем же мы провинились?

Мы открыли наши данные!

Вот так мы насвистываем себе что-то под нос, не заботясь о том, что наши данные брошены на произвол судьбы и любой может их увидеть и даже потрогать.

Вы уже должны были ощутить смутную тревогу от осознания того, что ваши переменные экземпляра открыты *всем* подряд.

Открыты — значит доступны при использовании оператора «точка», как показано ниже:

```
theCat.height = 27;
```

Представьте, что любой человек может использовать пульт управления, чтобы изменить поле size нашего объекта Cat. Попав в руки не того человека, ссылочная переменная (пульт управления) может превратиться в опасное оружие. Этого не должно произойти.

```
theCat.height = 0;
```

О нет! Мы не можем этого допустить!

Это будет большой ошибкой. Необходимо создать методы-сеттеры для всех переменных экземпляра и найти способ заставить весь остальной код обращаться к ним именно так, а не напрямую.

Заставляя всех вызывать методы-сеттеры, мы можем защитить наш объект Cat от недопустимых изменений.

```
public void setHeight(int ht) {
    if (ht > 9) {
        height = ht;
    }
}
```

Мы добавляем условие, чтобы гарантировать минимальное значение поля height.

## Прячем данные

На самом деле можно легко уйти от реализации, которая провоцирует прием плохих данных, к подходу, защищающему ваши поля и учитывающему ваше право вносить изменения спустя некоторое время.

Но как именно спрятать данные? Это можно сделать с помощью модификаторов public и private. С первым вы уже знакомы, так как использовали его в каждом главном методе.

Первое правило инкапсуляции, которое опирается на опыт многих программистов, гласит (все оговорки, присущие таким правилам, остаются в силе): помечайте свои переменные экземпляра модификатором private, добавляя публичные (public) геттеры и сеттеры для контроля за доступом. Получив больше навыков в проектировании и кодировании на языке Java, вы, вероятно, начнете делать это немного иначе, но пока такой подход вас обезопасит.

**Помечайте переменные экземпляра как private.**

**Помечайте геттеры и сеттеры как public.**

**«К сожалению, Билл забыл инкапсулировать свой класс Cat и получил плоскую кошку».**

(Было подслушано возле кулера с водой.)

## РАЗОБЛАЧЕНИЕ ЯЗЫКА JAVA

**Интервью этой недели:**
**Непредвзятый взгляд на инкапсуляцию глазами Объекта.**

**HeadFirst:** Чем так важна инкапсуляция?

**Объект:** Знаете, иногда человеку снится, что он выступает перед полным залом людей и внезапно осознает, что *голый*.

**HeadFirst:** Да, нам тоже такое снилось. Значит, вы чувствуете себя голым. Но есть ли в этом какая-то опасность, кроме риска быть обнаженным?

**Объект:** Есть ли какая-то *опасность*? (Смеется.) Эй, коллеги-экземпляры, слышали, как он спросил: *«Есть ли какая-то опасность?»* (Падает на пол от смеха.)

**HeadFirst:** Что в этом смешного? По-моему, это вполне корректный вопрос.

**Объект:** Ладно, я объясню. Это... (Опять взрывается от смеха, не может остановиться.)

**HeadFirst:** Может, вам что-то принести? Воды, например?

**Объект:** Ух! Вот это да. Нет, я в порядке, правда. Попытаюсь быть серьезным. Глубокий вдох. Хорошо, приступим.

**HeadFirst:** Итак, от чего же вас защищает инкапсуляция?

**Объект:** Она создает защитный барьер вокруг моих переменных экземпляра, поэтому никто не сможет передать им, скажем, что-то *неподходящее*.

**HeadFirst:** Можете привести пример?

**Объект:** Большинство переменных экземпляра создаются с расчетом на то, что их значения будут ограничены определенными рамками. Представьте, сколько вещей сломается, если будут разрешены отрицательные значения. Количество уборных в офисе. Скорость самолета. Дни рождения. Вес штанги. Телефонные номера. Мощность микроволновой печи.

**HeadFirst:** Я понимаю, что вы имеете в виду. Как же инкапсуляция помогает установить эти рамки?

**Объект:** Она вынуждает остальной код использовать сеттеры. В таком случае метод-сеттер может проверить параметр и решить, подходит ли он. Возможно, метод отвергнет его и на этом все закончится, а может, будет сгенерировано исключение (например, если номер социального страхования в программе для работы с кредитными картами равен null) или параметр будет округлен до минимально допустимого значения. Суть в том, что внутри сеттера вы можете делать *все что угодно*, тогда как при наличии публичных переменных экземпляра вы не можете сделать ничего.

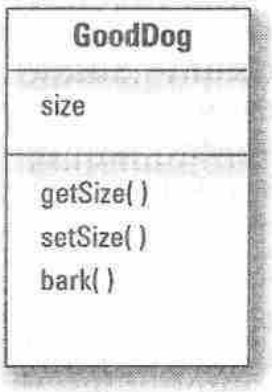
**HeadFirst:** Но иногда мне попадаются сеттеры, которые просто устанавливают значения, ничего не проверяя. Будет ли такой сеттер лишним, если у вас есть переменная экземпляра без рамок? Не повлияет ли это на производительность?

**Объект:** Смысл сеттеров (и геттеров тоже) в том, что позже ***можно передумать и что-то поменять, но при этом не придется нарушать чужой код***! Представьте, что половина сотрудников компании используют ваш класс с его публичными переменными экземпляра, и однажды вы внезапно понимаете: «Ой, с этим значением можно сделать то, чего я не предусмотрел. Заменю-ка я его сеттером». В итоге вы сломаете весь чужой код. Самое замечательное в инкапсуляции то, что *вы всегда можете передумать*, и из-за этого никто не пострадает. Выгода же от непосредственного использования значений слишком несущественна и проявляется *(если такое вообще происходит)* очень редко.

# Инкапсуляция на примере класса GoodDog

Помечаем переменную экземпляра как private.

Помечаем геттеры и сеттеры как public.




```
class GoodDog {

   private int size;

   public int getSize() {
     return size;
   }

   public void setSize(int s) {
     size = s;
   }

   void bark() {
     if (size > 60) {
        System.out.println("Гав Гав!");
     } else if (size > 14) {
        System.out.println("Вуф Вуф!");
     } else {
        System.out.println("Тяф Тяф!");
     }
   }
}

class GoodDogTestDrive {

   public static void main (String[] args) {
     GoodDog one = new GoodDog();
     one.setSize(70);
     GoodDog two = new GoodDog();
     two.setSize(8);
     System.out.println("Первая собака: " + one.getSize());
     System.out.println("Вторая собака: + two.getSize());
     one.bark();
     two.bark();
   }
}
```

Даже несмотря на то, что метод не добавляет новых возможностей, он полезен, так как позволяет в дальнейшем что-то поменять. Можно вернуться к этому методу и сделать его безопаснее, быстрее и лучше.

**Везде, где можно использовать конкретное значение, допускается вызывать *метод, который возвращает значение соответствующего типа.***

**Вместо:**

```
int x = 3 + 24;
```

**вы можете написать:**

```
int x = 3 + one.getSize();
```

# Как себя ведут объекты внутри массива

Как и любые другие элементы. Единственное отличие заключается в *способе доступа* к ним. Другими словами, важно, как вы получаете пульт управления. Попытаемся вызвать методы из объектов Dog, которые находятся внутри массива.

**1** Объявляем и создаем массив для хранения семи ссылок типа Dog.

```
Dog[] pets;
pets = new Dog[7];
```

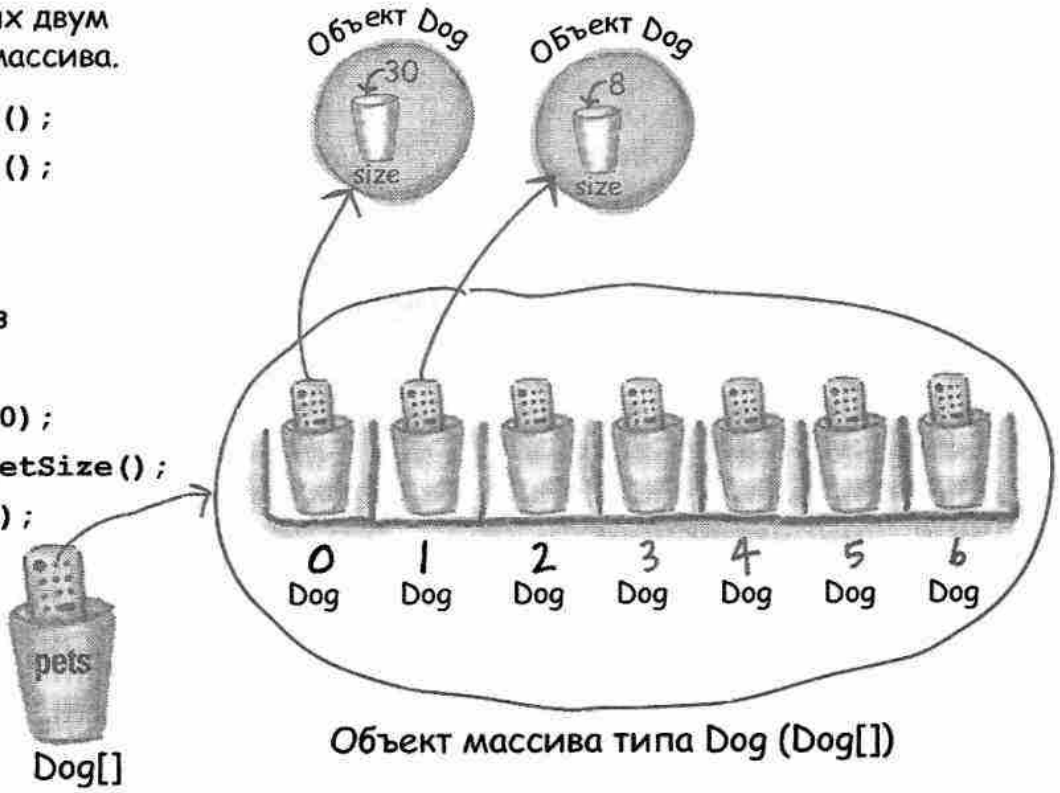
---

**2** Создаем два новых объекта Dog и присваиваем их двум первым элементам массива.

```
pets[0] = new Dog();
pets[1] = new Dog();
```

**3** Вызываем методы из объектов Dog.

```
pets[0].setSize(30);
int x = pets[0].getSize();
pets[1].setSize(8);
```



# Объявление и инициализация переменных экземпляра

Вы уже знаете, что при объявлении переменных необходимо указывать их имя и тип:

```
int size;
String name;
```

Вы также знаете, что одновременно с этим можно инициализировать переменную (присвоить ей значение):

```
int size = 420;
String name = "Donny";
```

Но что получится, если вы вызовете геттер, не инициализировав соответствующую переменную экземпляра? Иными словами, какое *значение* переменная экземпляра содержит *до* своей инициализации?

**Переменные экземпляра всегда получают значения по умолчанию. Если вы явно не присвоите переменной значение или не вызовете сеттер, она все равно будет хранить значение!**

| | |
|---|---|
| Целые | 0 |
| С плавающей точкой | 0.0 |
| Булевые | false |
| Ссылки | null |

```
class PoorDog {

   private int size;
   private String name;

   public int getSize() {
      return size;
   }
   public String getName() {
      return name;
   }

}

public class PoorDogTestDrive {
   public static void main (String[] args) {
      PoorDog one = new PoorDog();
      System.out.println("Размер собаки - " + one.getSize());
      System.out.println("Имя собаки - " + one.getName());
   }
}
```

Объявляем две переменные, но не присваиваем им значение.

Что вернут эти методы?

Как вы думаете, скомпилируется ли этот код?

```
File Edit Window Help CallVet
% java PoorDogTestDrive
Размер собаки - 0
Имя собаки - null
```

Необязательно инициализировать переменные экземпляра, так как они всегда имеют значения по умолчанию. Числовые примитивы (включая char) получают 0, boolean — false, а ссылкам на объекты присваивается null.

Значение null говорит о том, что пульт управления ничем не управляет (не запрограммирован). Ссылка есть, но нет объекта.

# Разница между переменными экземпляра и локальными переменными

**1** **Переменные экземпляра** объявляются внутри класса, но за пределами метода.

```
class Horse {
   private double height = 15.2;
   private String breed;
   // еще код...
}
```

**2** **Локальные переменные** объявляются внутри метода.

```
class AddThing {
   int a;
   int b = 12;

   public int add() {
      int total = a + b;
      return total;
   }
}
```

**3** **Локальные переменные** перед использованием нужно инициализировать!

```
class Foo {
   public void go() {
      int x;
      int z = x + 3;
   }
}
```

Не скомпилируется! Можно объявить x без значения, но как только вы попытаетесь его использовать, компилятор сойдет с ума.

```
File Edit Window Help Yikes
% javac Foo.java
Foo.java:4: variable x might
not have been initialized
      int z = x + 3;
1 error          ^
```

**Локальные переменные не содержат значение по умолчанию! Компилятор будет возмущаться, если вы попытаетесь использовать локальную переменную до того, как инициализируете ее.**

## Это не глупые вопросы

**В: Как работают правила о локальных переменных по отношению к параметрам методов?**

О: Параметры методов фактически ничем не отличаются от локальных переменных — они объявлены *внутри* метода (формально они объявлены в *списке аргументов* метода, а не в его теле, но являются локальными переменными, а не переменными экземпляра). Однако они всегда содержат значения, поэтому вы никогда не получите от компилятора ошибку, в которой говорится, что параметр не инициализирован.

Если вы попытаетесь вызвать метод, не передав ему необходимые аргументы, компилятор выдаст вам другую ошибку. Таким образом, параметры *всегда* инициализированы, так как компилятор гарантирует, что методы вызываются с аргументами, совпадающими с объявленными для них параметрами. Таким образом, аргументы присваиваются параметрам автоматически.

# Сравниваем переменные (примитивы или ссылки)

**Используйте оператор ==, чтобы сравнить примитивы или проверить, ссылаются ли переменные на *один и тот же* объект.**

**Используйте метод equals() для проверки идентичности двух *разных* объектов.**

**Например, можно проверить два объекта типа String, содержащих одинаковую последовательность символов.**

Иногда нужно узнать, одинаковы ли два *примитива*. Это довольно просто сделать, используя оператор ==. Порой разработчика интересует, указывают ли две ссылки на один и тот же объект в куче. Это тоже легко выяснить благодаря тому же оператору ==. Но иногда нужно узнать, идентичны ли два *объекта*. И для этого понадобится метод `equals()`. Идентичность (или эквивалентность) объектов зависит от их типа. К примеру, если два объекта типа `String` содержат одинаковые символы (скажем, слово «проворный»), они явно эквивалентны, несмотря на то что представляют два разных объекта в куче. Но как насчет типа `Dog`? Будете ли вы рассматривать два объекта этого типа как эквивалентные, если им посчастливилось иметь одинаковые значения переменных `size` и `weight`? Вряд ли. Таким образом, два объекта должны считаться эквивалентными, если это имеет смысл для конкретного типа. К идее эквивалентности объектов мы вернемся в следующих главах (и в Приложении Б), а пока вы должны четко уяснить, что оператор == используется *только* для сравнения битов в двух переменных. Совершенно не важно, что эти биты собой представляют. Они либо одинаковые, либо нет.

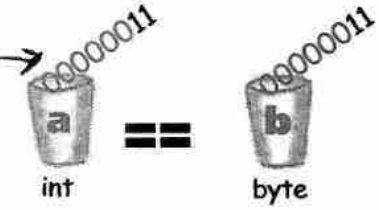
### Для сравнения двух примитивов применяйте оператор ==

Оператор == может быть использован для сравнения двух переменных любого типа — он просто проверяет на соответствие их биты.

Выражение `if (a == b) {...}` смотрит на биты внутри `a` и `b` и возвращает `true`, если они совпадают (при этом размер переменных не имеет значения, начальные нули не учитываются).

```
int a = 3;
byte b = 3;
if (a == b) { // true }
```

В переменной типа int слева больше нулей, но здесь это не имеет значения.




Последовательность битов совпадает, поэтому оператор == говорит, что они идентичны.

### Чтобы проверить, идентичны ли две ссылки (ссылаются ли они на один объект в куче), используйте оператор ==

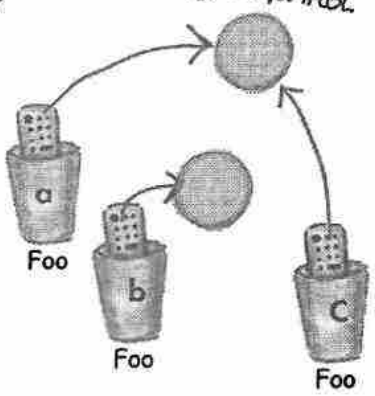
Помните, оператора == интересует только последовательность битов в переменной. Это правило работает как для примитивов, так и для ссылок. Таким образом, оператор == возвращает `true`, если ссылочные переменные указывают на один и тот же объект! В данном случае мы не знаем, какие именно биты там содержатся (эта информация спрятана от нас из-за особенностей JVM), но нам точно известно, что *две ссылки на один объект будут хранить одинаковые данные*.

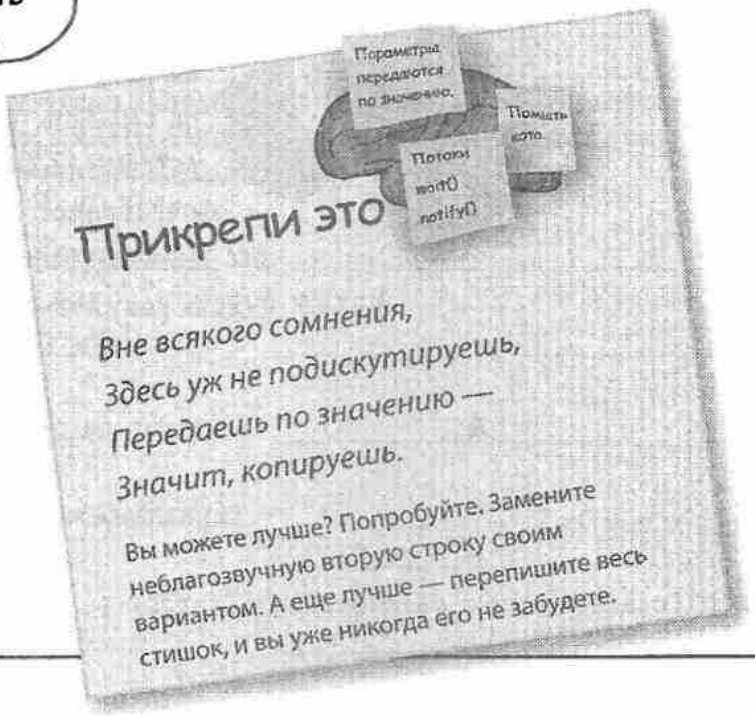
```
Foo a = new Foo();
Foo b = new Foo();
Foo c = a;
if (a == b) { // false }
if (a == c) { // true }
if (b == c) { // false }
```

a == c вернет true.

a == b вернет false.

Последовательность битов для a и c совпадает, поэтому оператор == говорит, что они идентичны.






Наточите свой карандаш

## Что допустимо?

Ниже приведен метод. Какие его вызовы, представленные справа, разрешены?

Установите напротив правильного вызова флажок (некоторые выражения используются для присвоения значений, передаваемых в дальнейшем в метод).

```
int calcArea(int height, int width) {
   return height * width;
}
```

```
int a = calcArea(7, 12);
short c = 7;
calcArea(c,15);
int d = calcArea(57);
calcArea(2,3);
long t = 42;
int f = calcArea(t,17);
int g = calcArea();
calcArea();
byte h = calcArea(4,20);
int j = calcArea(2,3,5);
```

# Поработайте компилятором

Каждый Java-файл на этой странице — полноценный исходник. Ваша задача — притвориться компилятором и определить, все ли файлы скомпилируются. Если компиляция не сможет пройти успешно, как вы их исправите? А если они все же скомпилируются, каким будет результат?

**A**

```
class XCopy {

  public static void main(String [] args) {

    int orig = 42;

    XCopy x = new XCopy();

    int y = x.go(orig);

    System.out.println(orig + " " + y);
  }

  int go(int arg) {

    arg = arg * 2;

    return arg;
  }
}
```

**B**

```
class Clock {
  String time;

  void setTime(String t) {
    time = t;
  }

  void getTime() {
    return time;
  }
}

class ClockTestDrive {
  public static void main(String [] args) {

    Clock c = new Clock();

    c.setTime("1245");
    String tod = c.getTime();
    System.out.println("время: " + tod);
  }
}
```

Упражнение

# Кто я такой?

Команда звезд из мира Java хочет сыграть с вами в игру «Кто я такой?». Они дают вам подсказку, и вы пытаетесь угадать их имена, основываясь на полученной информации. Считайте, что они никогда не врут. Если они говорят что-нибудь подходящее сразу для нескольких участников, то записывайте всех, к кому применимо данное утверждение. Заполните пустые строки именами одного или нескольких участников рядом с утверждениями.

**Сегодня участвуют:**

**переменная экземпляра, аргумент, return, геттер, сеттер, инкапсуляция, public, private, передача по значению, метод.**

Класс может содержать лишь один такой элемент. _____

Метод может хранить только один такой элемент. _____

Это может быть неявно приведено к другому типу. _____

Я предпочитаю, чтобы мои переменные экземпляра были закрытыми. _____

Это означает «сделать копию». _____

Только сеттер должен изменять это. _____

Метод может иметь много таких элементов. _____

Я что-то возвращаю, и это ясно из моего названия. _____

Я не должен использоваться вместе с переменными экземпляра. _____

У меня может быть много аргументов. _____

По определению я принимаю один аргумент. _____

Это помогает создавать инкапсуляцию. _____

Я всегда беру на борт только одного пассажира. _____

# Смешанные сообщения

Справа приведена небольшая Java-программа. Два ее блока потерялись. Ваша задача — **сопоставить варианты блоков кода** *(ниже)* **с результатами вывода**, которые вы увидите, если эти блоки будут добавлены.

Не все строки с результатами будут использованы, а некоторые пригодятся несколько раз. Соедините линиями варианты кода с соответствующим результатом в командной строке.

```
public class Mix4 {
  int counter = 0;
  public static void main(String [] args) {
    int count = 0;
    Mix4 [] m4a  =new Mix4[20];
    int x = 0;
    while (          ) {
      m4a[x] = new Mix4();
      m4a[x].counter = m4a[x].counter + 1;
      count = count + 1;
      count = count + m4a[x].maybeNew(x);
      x = x + 1;
    }
    System.out.println(count + " "
                       + m4a[1].counter);
  }

  public int maybeNew(int index) {
    if (          ) {
      Mix4 m4 = new Mix4();
      m4.counter = m4.counter + 1;
      return 1;
    }
    return 0;
  }
}
```

**Варианты кода:**

```
x < 9
index < 5
```

```
x < 20
index < 5
```

```
x < 7
index < 7
```

```
x < 19
index < 1
```

**Возможный результат:**

```
14 7
```

```
9 5
```

```
19 1
```

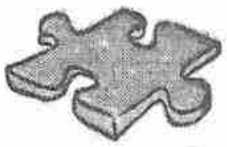
```
14 1
```

```
25 1
```

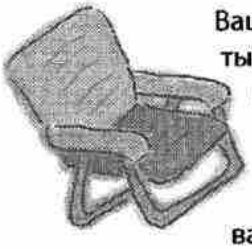
```
7 7
```

```
20 1
```

```
20 5
```



# Головоломка у бассейна



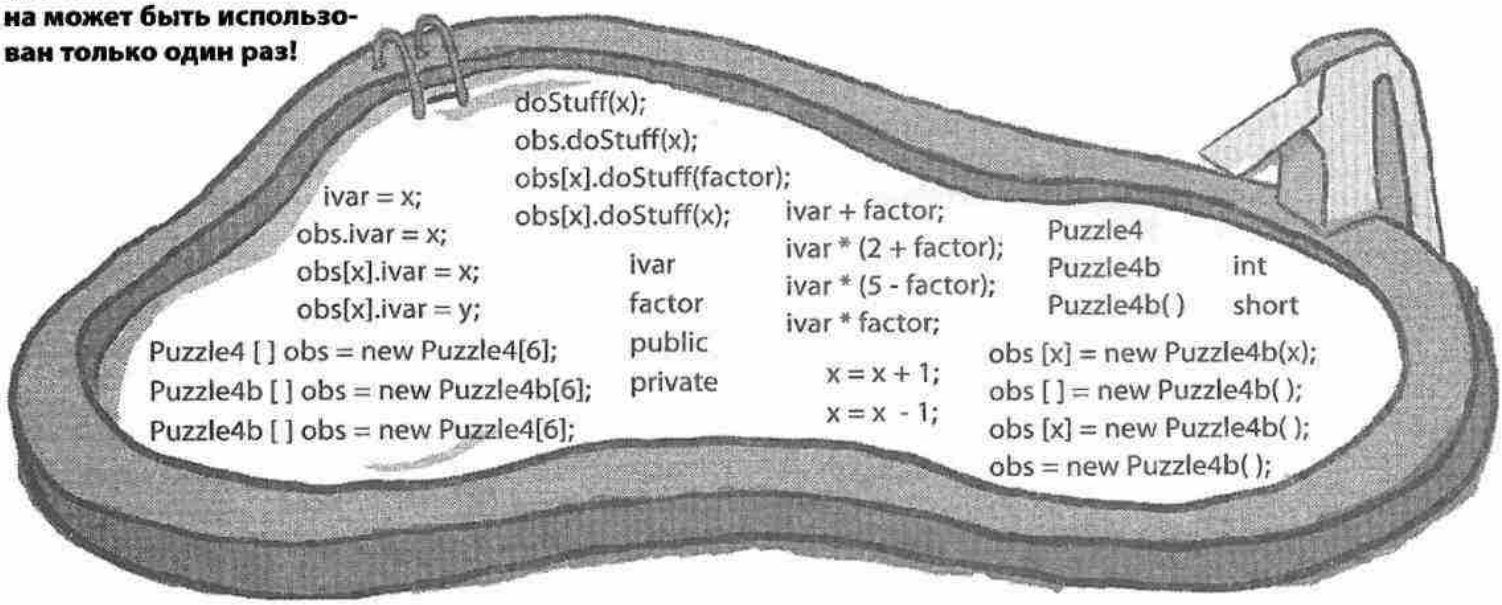
Ваша **задача** — взять фрагменты кода со дна бассейна и заменить ими пропущенные участки программы. Нельзя использовать один фрагмент несколько раз, и не все из них вам пригодятся. Ваша **цель** — создать класс, который скомпилируется, запустится и выведет приведенный ниже текст.

**Результат:**

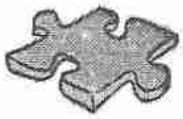
```
File Edit Window Help BellyFlop
%java Puzzle4
Результат 543345
```

```
public class Puzzle4 {
  public static void main(String [] args) {
    _____________________________________
    int y = 1;
    int x = 0;
    int result = 0;
    while (x < 6) {
      _____________________________
      _____________________________
      y = y * 10;
      ___________________
    }
    x = 6;
    while (x > 0) {
      ___________________
      result = result + _____________________
    }
    System.out.println("Результат" + result);
  }
}
class ____________ {
  int ivar;
  _________ ______ doStuff(int _________) {
    if (ivar > 100) {
      return _____________________________
    } else {
      return _____________________________
    }
  }
}
```

**Примечание: каждый фрагмент кода из бассейна может быть использован только один раз!**



## Время перемен в Стим Сити



Когда Бьюкенен направил свой пистолет в сторону Джайя, тот застыл. Джай знал, что глупость Бьюкенена могла сравниться только с его уродством, и не хотел первировать здоровяка. Бьюкенен вызвал Джайя в офис своего начальника, но Джай не сделал ничего плохого (по крайней мере за последнее время), поэтому он решил, что небольшая беседа с боссом Бьюкенена, господином Леверелом, не сулит больших неприятностей... В последнее время он продал приличное количество нейростимуляторов, и надеялся, что Леверел будет доволен. Работа подпольным распространителем препаратов не приносила много денег, но была спокойной. Большинство клиентов, с которыми он имел дело, спустя некоторое время оказывались на мели и возвращались к прежней жизни, разве что после этого они были чуть менее сосредоточенными, чем раньше.

«Офис» Леверела на первый взгляд представлял собой потрепанный автомобиль, но как только Бьюкенен затолкал его внутрь, Джай увидел, что эта колымага тюнингована настолько, чтобы обеспечить скорость и безопасность, какие мог себе позволить такой мелкий начальник, как Леверел. «Джай, мальчик мой, — выдавил из себя Леверел, — рад снова тебя видеть». «Взаимно, я полагаю, — ответил Джай, чувствуя неладное. — Мы вроде в расчете, или я что-то упустил?» «Ха! С виду все выглядит неплохо, твой оборот увеличился, но недавно я наткнулся на, скажем так, небольшую брешь...» — сказал Леверел.

Джай непроизвольно вздрогнул — в свое время он был одним из лучших хакеров-взломщиков. Каждый раз, когда кто-нибудь умудрялся сломать защиту одного из уличных авторитетов, Джай получал порцию ненужного ему внимания. «Да ладно, это же я, дружище, — залепетал Джай, — я завязал с хакерством. Я просто оставил это позади и занимаюсь своим бизнесом». «Да-да, — улыбнулся Леверел, — не сомневаюсь, что на этот раз ты чист, но я понесу большие убытки, пока новый хакер не исчезнет!» «Может быть, ты лучше высадишь меня здесь, и я продам еще парочку "упаковок" для тебя, прежде чем закончится мой рабочий день», — предложил Джай.

«Боюсь, все не так просто, Джай. Бьюкенен только что рассказал мне, чем ты приторговываешь на своем углу», — вкрадчиво произнес Леверел. «Нейролептики? Я немного с ними побаловался, что с того?» — ответил Джай, испытывая легкую тошноту. «С помощью нейролептиков я сообщаю клиентам, где будет следующая точка распространения, — пояснил Леверел. — Проблема в том, что некоторые наркоманы держатся достаточно долго, чтобы понять, как пролезть в базу данных моего склада». «Нужно, чтобы такой смышленый парень, как ты, Джай, взглянул на мой класс StimDrop — методы, переменные экземпляра и т. д. — и выяснил, как они проникают внутрь. Это должно...» «Эй! — крикнул Бьюкенен. — Я не хочу, чтобы такие грязные хакеры, как Джай, крутились возле моего кода!» «Спокойно, начальник, — Джай понял, что у него появился шанс. — Я уверен, что ты проделал первоклассную работу со своими модификаторами до...» «Молчи, бездельник! — заорал Бьюкенен. — Все методы для клиентов я оставил публичными, чтобы те могли заходить на сайт, но все важные методы класса, работающего со складскими данными, я сделал закрытыми. Никто не может получить доступ к ним извне, приятель, никто!»

«Думаю, я могу найти твою утечку, Леверел. Что скажешь, если мы высадим Бьюкенена здесь на углу и прокатимся вокруг квартала?» — предложил Джай. Бьюкенен выхватил свой револьвер, но Леверел уже прислонил дуло к его шее. «Оставь его, — усмехнулся Леверел, — брось ствол и выходи, похоже, у нас с Джайем есть дела».

***Какие подозрения возникли у Джайя?***

***Смог ли он выбраться из машины Леверела целым и невредимым?***

# Ответы

## Поработайте компилятором

**A** Класс XCopy компилируется и запускается в первоначальном виде! Программный вывод: 42.84. Помните, что Java передает параметры по значению (то есть путем копирования), переменная orig не изменится в методе go().

**B**

```
class Clock {
  String time;
  void setTime(String t) {
    time = t;
  }
  String getTime() {
    return time;
  }
}

class ClockTestDrive {
  public static void main(String [] args) {
    Clock c = new Clock();
    c.setTime("1245");
    String tod = c.getTime();
    System.out.println("время: " + tod);
  }
}
```

**Примечание: метод-геттер должен содержать в своем объявлении тип возвращаемого значения.**

---

## Кто я такой?

| | |
|---|---|
| Класс может содержать лишь один такой элемент. | **Переменные экземпляра, геттер, сеттер, метод.** |
| Метод может хранить только один такой элемент. | **return** |
| Это может быть неявно приведено к другому типу. | **return, аргумент.** |
| Я предпочитаю, чтобы мои переменные экземпляра были закрытыми. | **Инкапсуляция** |
| Это означает «сделать копию». | **Передача по значению** |
| Только сеттер должен изменять это. | **Переменные экземпляра.** |
| Метод может иметь много таких элементов. | **Аргумент.** |
| Я что-то возвращаю и это ясно из моего названия. | **Геттер.** |
| Я не должен использоваться вместе с переменными экземпляра. | **public.** |
| У меня может быть много аргументов. | **Метод.** |
| По определению я принимаю один аргумент. | **Сеттер.** |
| Это помогает создавать инкапсуляцию. | **Геттер, сеттер, public, private.** |
| Я всегда беру на борт только одного пассажира. | **return** |

# Ответы на головоломки

## Головоломка у бассейна

```
public class Puzzle4 {
  public static void main(String [] args) {
    Puzzle4b [ ] obs = new Puzzle4b[6];
    int y = 1;
    int x = 0;
    int result = 0;
    while (x < 6) {
      obs[x] = new Puzzle4b( );
      obs[x] . ivar = y;
      y = y * 10;
      x = x + 1;
    }
    x = 6;
    while (x > 0) {
      x = x - 1;
      result = result + obs[x].doStuff(x);
    }
    System.out.println("результат" + result);
  }
}
class Puzzle4b {
  int ivar;
  public int doStuff(int factor) {
    if (ivar > 100) {
      return ivar * factor;
    } else {
      return ivar * (5 - factor);
    }
  }
}
```

**Результат:**

```
File Edit Window Help BellyFlop
%java Puzzle4
Результат 543345
```

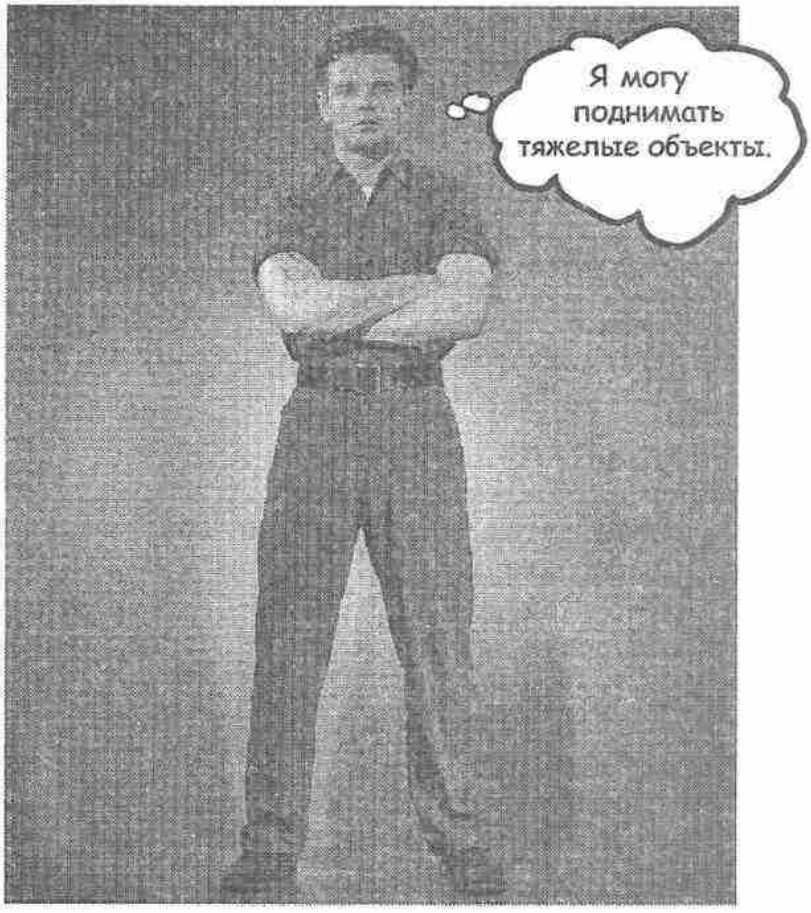
## Разгадка пятиминутного детектива

Джай знал, что Бьюкенен не самый смышленый парень на свете. Рассказывая о своем коде, тот ни разу не упомянул переменные экземпляра. Джай предположил, что, защитив методы, Бьюкенен забыл пометить переменные экземпляра модификатором `private`. Эта ошибка запросто могла стоить Леверелу больших денег.

## Смешанные сообщения

| Варианты кода: | Возможный результат: |
|---|---|
| `x < 9`<br>`index < 5` | 14 7 |
| | 9 5 |
| `x < 20`<br>`index < 5` | 19 1 |
| | 14 1 |
| `x < 7`<br>`index < 7` | 25 1 |
| | 7 7 |
| `x < 19`<br>`index < 1` | 20 1 |
| | 20 5 |

# Особо мощные методы




**Сделаем методы еще более мощными.** Вы уже поработали с переменными, несколькими объектами и написали небольшой код. Но для полноценной работы потребуется больше инструментов. Например, **операторы** нужны для того, чтобы сделать нечто более интересное, чем воспроизведение *лая* собаки или программа с **циклами**. Конечно, циклы нужны, но не только такие простые, как *while*. Если вы на самом деле решительно настроены, придется прибегнуть к циклам ***for***. Они пригодятся для **генерирования случайных чисел** и **преобразования строк в тип int** — поверьте, это будет интересно. Почему бы вам во время учебы не *создать* что-нибудь настоящее, чтобы собственными глазами увидеть, как с нуля пишутся (и тестируются) программы? **Может быть**, это будет игра вроде «Морского боя». Тяжелая задача, поэтому на ее выполнение придется потратить *две* главы. В этой главе мы рассмотрим упрощенную версию игры, а потом сделаем ее более мощной и интересной.

# Создадим аналог «Морского боя»: игра «Потопи сайт»

Здесь вы будете играть против компьютера. Основное отличие от оригинального «Морского боя» состоит в том, что тут не нужно размещать свои корабли. Вместо этого придется потопить корабли компьютера за минимальное количество ходов.

Кроме того, вы будете топить не корабли, а «сайты» (повышая тем самым свою бизнес-грамотность и окупая расходы на эту книгу).

**Цель:** потопить все «сайты» компьютера за минимальное количество попыток. Вы будете получать баллы в зависимости от того, насколько хорошо играете.

**Подготовка:** при загрузке программы компьютер разместит три «сайта» **на виртуальной доске (7×7)**. После этого вас попросят сделать первый ход.

**Как играть:** вы пока не научились создавать GUI (графический пользовательский интерфейс), поэтому данная версия будет работать в командной строке. Компьютер предложит вам ввести предполагаемую ячейку в виде A3, C5 и т. д. В ответ на это в командной строке вы получите результат — либо «Попал», либо «Мимо», либо «Вы потопили Pets.com» (или любой другой «сайт», который вам посчастливилось потопить в этот день). Как только вы отправите на корм рыбам все три «сайта», игра завершится выводом на экран вашего рейтинга.

**Вам предстоит создать игру «Потопи сайт» с доской 7x7 и тремя «сайтами», каждый из которых занимает три ячейки.**

## Фрагмент игрового диалога

```
File Edit Window Help Sell
%java DotComBust
Сделайте ход  A3
Мимо
Сделайте ход  B2
Мимо
Сделайте ход  C4
Мимо
Сделайте ход  D2
Попал
Сделайте ход  D3
Попал
Сделайте ход  D4
Ой! Вы потопили Pets.com  :(
Потопил
Сделайте ход  B4
Мимо
Сделайте ход  G3
Попал
Сделайте ход  G4
Попал
Сделайте ход  G5
Ой! Вы потопили AskMe.com  :(
```

# В первую очередь высокоуровневое проектирование

Понятно, что нужны классы и методы, но какими они должны быть? Чтобы ответить на этот вопрос, необходимо получить больше информации о том, как игра должна себя вести.

Прежде всего нужно определиться с игровым процессом. Рассмотрим общую идею.

**1** Пользователь запускает игру.

- **A** Игра создает три «сайта».
- **B** Игра размещает «сайты» на виртуальной игровой доске.

**2** Игра начинается.

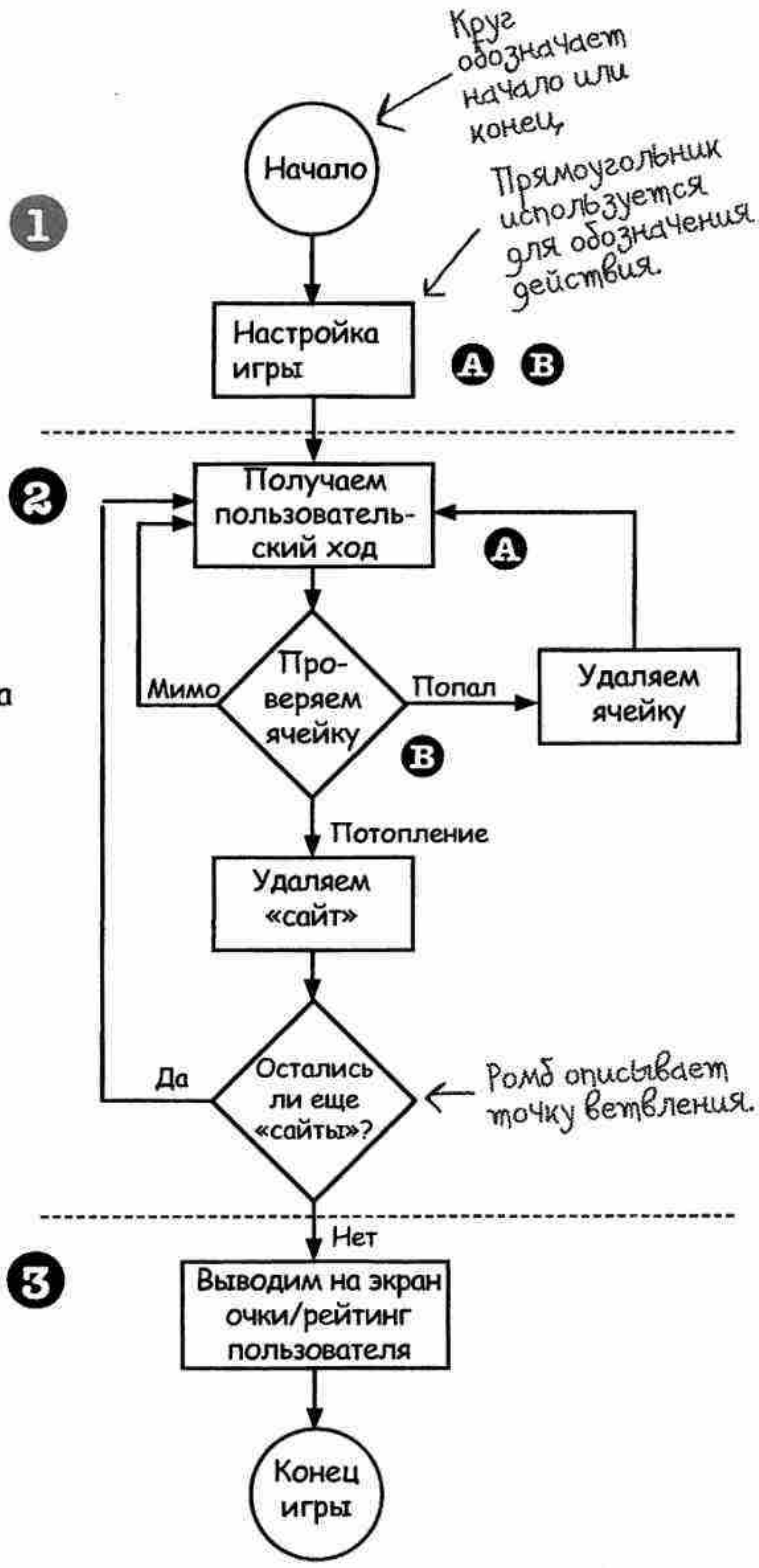
Повторять следующие действия, пока не останется ни одного «сайта».

- **A** Предложить пользователю сделать ход (A2, C0 и т. д.).
- **B** Проверить все «сайты» на попадание, промах и потопление. Выбрать подходящее действие: если попадание — удалить ячейку (A2, D4 и т. д.). Если потопление — удалить «сайт».

**3** Игра заканчивается.

Показать пользователю рейтинг, основываясь на количестве попыток.

Итак, вы получили представление о том, что должна делать программа. Далее необходимо выяснить, какие для этого понадобятся **объекты**. Думайте об этом, как Брэд, а не как Ларри; в первую очередь сосредотачивайтесь на ***элементах***, из которых состоит программа, а не на ***процедурах***.




Ух ты! Настоящая блок-схема.

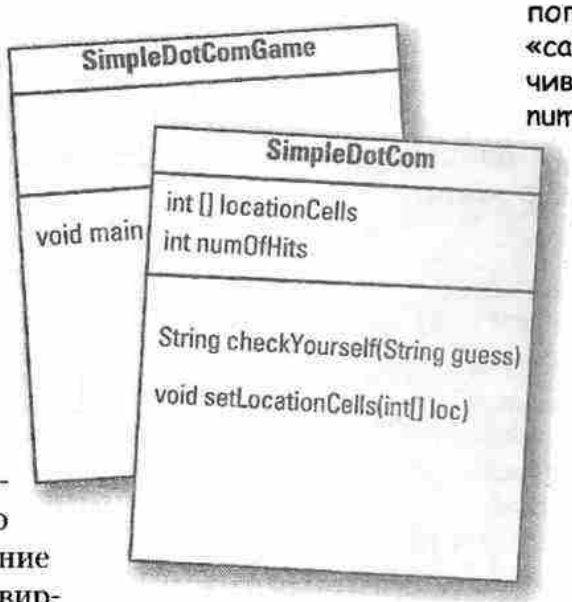
# Плавное введение в игру «Потопи сайт»

Похоже, потребуется минимум два класса — Game и DotCom. Но прежде чем написать полноценную игру, создадим ее упрощенную версию. Этим мы и займемся в текущей главе, а полноценную версию оставим для главы 6.

В этой версии все просто. Вместо двумерной сетки мы будем использовать единственный *ряд*. И вместо *трех* «сайтов» у нас будет только *один*.

Тем не менее принцип игры остается тем же: нужно создать экземпляр класса DotCom, присвоить ему положение где-нибудь в ряду, получить пользовательский ввод и, когда все три ячейки «сайта» будут поражены, закончить игру.

Упрощенная версия дает хороший старт для создания полноценной игры. Создав что-то маленькое, позже можно расширить и усложнить это.

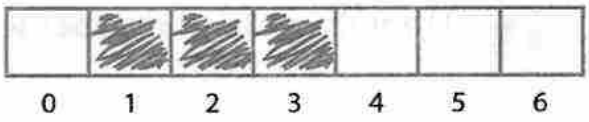



В текущем варианте класс игры не содержит переменных экземпляра, а весь игровой код находится в методе main(). Иными словами, после запуска игры и вызова главного метода будет создан только один экземпляр класса DotCom, после чего для него будет выбрано положение (три последовательные ячейки виртуального ряда). Далее программа предложит пользователю сделать ход, проверит его вариант, и предыдущие два действия станут повторяться, пока все три клетки не будут поражены.

Не забывайте, что виртуальный ряд остается *виртуальным*, то есть не существует нигде в программе. Пока игре и пользователю известно, что «сайт» спрятан в трех последовательных ячейках из семи возможных (начиная с нулевой), ряд как таковой не нуждается в программном представлении. Может возникнуть соблазн создать массив из семи элементов и присвоить трем из них числа, представляющие «сайт», но это необязательно делать. Нужен массив, который хранит лишь три клетки, занимаемые «сайтом».

1. **Игра запускается и создает один «сайт»**, присваивая ему адрес из трех ячеек в ряду.

   Вместо A2, C4 и подобных обозначений положение сайта представлено числами. Например, местоположение 1, 2, 3 показано на следующей картинке:




2. **Начинается игровой процесс.** Предлагаем пользователю сделать ход, после чего проверяем, попал ли он в одну из трех ячеек «сайта». Если попал, то увеличиваем значение переменной numOfHits на 1.

3. **Игра завершается,** если все три ячейки поражены (значение переменной numOfHits достигло 3), а пользователю сообщается, сколько ходов ему потребовалось для потопления «сайта».

Полная версия игрового диалога.

```
File Edit Window Help Destroy
%java SimpleDotComGame
Введите число  2
Попал
Введите число  3
Попал
Введите число  4
Мимо
Введите число  1
Потопил
Вам потребовалось 4 попыток(и)
```

# Разработка класса

Как у каждого программиста, у вас, вероятно, есть своя методология написания кода. Собственно, как и у нас. Список, приведенный ниже, создан для того, чтобы помочь вам увидеть и понять, о чем мы думали при разработке класса. Необязательно использовать такой подход при написании настоящего кода. Несомненно, на практике вы будете основываться на собственных предпочтениях, требованиях проекта или работодателя. Однако перед вами простирается обширное поле для действий. Например, когда мы создаем «учебный» класс на языке Java, обычно делаем это так.

- ☐ **Выясняем, что должен *делать* класс.**
- ☐ **Перечисляем переменные экземпляра и методы.**
- ☐ **Пишем псевдокод для методов (очень скоро вы увидите, о чем речь).**
- ☐ **Пишем тестовый код для методов.**
- ☐ **Реализуем класс.**
- ☐ **Тестируем методы.**
- ☐ **Отлаживаем и при необходимости корректируем.**
- ☐ **Радуемся, что не нужно проверять нашу так называемую *учебную программу* на реальных пользователях.**

## Три кода, которые мы напишем для каждого класса:

Эта полоса будет размещена на следующих страницах, чтобы напоминать вам, над какой частью вы в данный момент работаете. Например, если вы видите ее вверху страницы, это означает, что вы пишете псевдокод для класса SimpleDotCom.

### Псевдокод

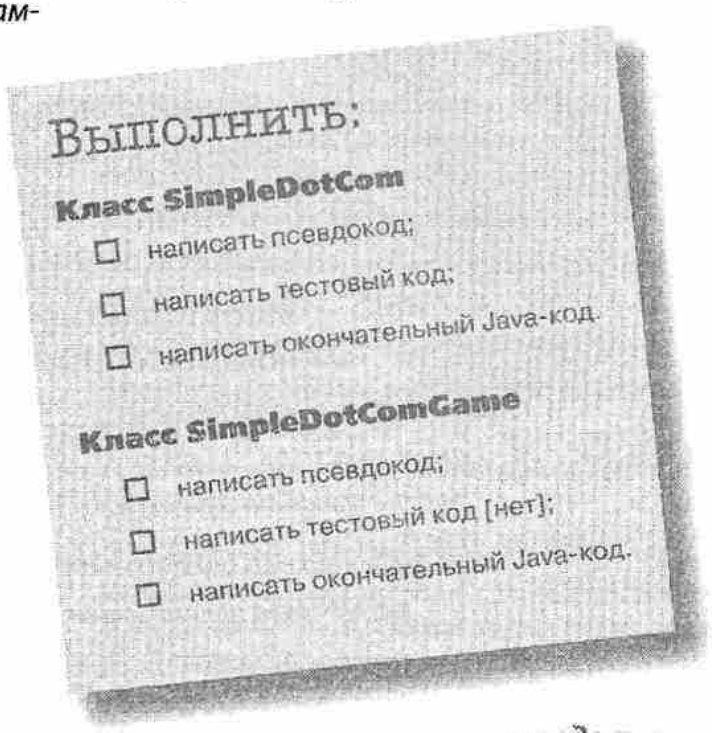
Алгоритм, который поможет вам сосредоточиться на логике, не вникая в синтаксис.

### Тестовый код

Класс или метод, с помощью которого можно проверять реальный код и подтверждать, что он работает правильно.

### Реальный код

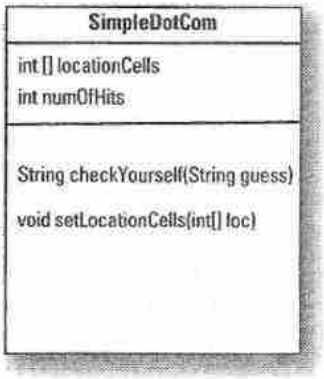
Непосредственная реализация класса. Это рабочий код на языке Java.



**Пора размять извилины.**

**Как решить, какой класс (или классы) создавать в *первую* очередь? С чего начать, если учитывать, что любая программа приличных размеров состоит из нескольких классов (мы предполагаем, что вы придерживаетесь объектно ориентированного стиля и *один* класс у вас не выполняет сразу несколько действий)?**

Псевдокод | Тестовый код | Реальный код




Ознакомившись с этим примером, вы поймете, как работает наш вариант псевдокода. Это что-то среднее между реальным кодом на языке Java и описанием класса на человеческом языке. Большая часть псевдокода содержит три раздела: объявление переменных экземпляра, объявление методов, логика методов. Из этих трех составляющих наиболее важна последняя, так как в ней описывается, *что* должно произойти.

**ОБЪЯВЛЯЕМ** *целочисленный массив* для хранения адреса ячеек. Даем ему имя *locationCells*.

**ОБЪЯВЛЯЕМ** переменную типа *int* для хранения количества попаданий. Называем ее *numOfHits* и **ПРИСВАИВАЕМ** ей 0.

---

**ОБЪЯВЛЯЕМ** метод *checkYourself()*, который принимает ход пользователя в качестве параметра *String* (1, 3 и т. д.), проверяет его и возвращает результат: «Попал», «Мимо» или «Потопил».

**ОБЪЯВЛЯЕМ** сеттер *setLocationCells()*, который принимает *целочисленный массив* (хранящий адрес трех ячеек в виде переменных типа *int* — 2, 3, 4 и т. д.).

---

**МЕТОД:** ***String checkYourself(String userGuess)***

**ПОЛУЧАЕМ** ход пользователя в виде строкового параметра.

**ПРЕОБРАЗУЕМ** полученные данные в тип *int*.

**ПОВТОРЯЕМ** это с каждой ячейкой массива.

***// СРАВНИВАЕМ*** ход пользователя с местоположением клетки.

**ЕСЛИ** пользователь угадал,

**ИНКРЕМЕНТИРУЕМ** количество попаданий.

***// ВЫЯСНЯЕМ,*** была ли это последняя ячейка.

**ЕСЛИ** количество попаданий равно 3, **ВОЗВРАЩАЕМ** «Потопил».

**ИНАЧЕ** потопления не произошло, значит, **ВОЗВРАЩАЕМ** «Попал».

КОНЕЦ ВЕТВЛЕНИЯ

**ИНАЧЕ** пользователь не попал, значит, **ВОЗВРАЩАЕМ** «Мимо».

КОНЕЦ ВЕТВЛЕНИЯ

КОНЕЦ ПОВТОРЕНИЯ

КОНЕЦ МЕТОДА

**МЕТОД:** ***void setLocationCells(int[] cellLocations)***

**ПОЛУЧАЕМ** адреса ячеек в виде параметра с *целочисленным массивом*.

**ПРИСВАИВАЕМ** полученный параметр полю, хранящему адреса ячеек.

КОНЕЦ МЕТОДА

## Записываем реализацию метода

### Теперь напишем настоящий код и заставим его работать.

Но прежде чем начать, перестрахуемся и напишем код для *тестирования*. Все верно, мы напишем проверочный код еще *до того*, как у нас появится что проверять!

Предварительное создание тестового кода — одна из концепций экстремального программирования (Extreme Programming, или XP) — позволяет упростить (и ускорить) написание программы. Это не значит, что вы должны использовать XP, но подход с предварительным написанием тестов нам импонирует. Кроме того, XP — это здорово *звучит*!

## Экстремальное программирование (XP)

Экстремальное программирование — это новое веяние в мире веб-технологий. Появившись в конце 1990-х, эта концепция была опробована многими компаниями: от мелких магазинов с двумя работниками до корпорации Ford. Экстремальное программирование часто рассматривается как «подход, с которым программисты действительно хотят работать». Суть XP — клиент всегда получает то, что хочет и когда хочет, даже если для этого необходимо постфактум менять техническое задание.

Экстремальное программирование основано на наборе проверенных правил и рекомендаций, которые дополняют друг друга, хотя многие люди выбирают и адаптируют для себя только часть из них. Эти правила и рекомендации включают в себя следующее.

Вносите в новые версии небольшие изменения, но делайте это часто.

Придерживайтесь пошаговой циклической разработки.

Не добавляйте в код элементы, которых нет в техническом задании (независимо от того, насколько сильно вам хочется сделать это «на будущее»).

*В первую очередь* пишите тестовый код.

Никаких сверхурочных; используйте обычное рабочее время.

Улучшайте код при каждом удобном случае.

Не выпускайте новую версию, пока она не пройдет все тесты.

Устанавливайте реалистичные сроки, привязанные к выпуску минорных версий.

Не усложняйте (принцип Keep it simple).

Разбивайте разработчиков на пары и постоянно меняйте их местами, чтобы каждый знал о коде практически все.

# Тестовый код для класса SimpleDotCom

Нужно написать тестовый код, который позволит создать объект SimpleDotCom и запустить его методы. В данном случае нам интересен только метод *chackYourSelf()*, но для его правильной работы нам придется также реализовать метод *setLocationCells()*.

Внимательно изучите псевдокод для метода *chackYourSelf()*, приведенный ниже. Здесь метод *setLocationCells()* — обычный сеттер, поэтому мы не обращаем на него внимания. Однако в реальном приложении *может* понадобиться более сложный метод с необходимостью тестирования.

Теперь спросите себя: «Если метод checkYourSelf() уже реализован, какой тестовый код я могу написать, чтобы убедиться в его корректной работе?»

## Исходя из этого псевдокода:

**МЕТОД:** ***String checkYourself(String userGuess)***

**ПОЛУЧАЕМ** ход пользователя в виде строкового параметра.

**ПРЕОБРАЗУЕМ** полученные данные в *int*.

**ПОВТОРЯЕМ** это с каждой ячейкой массива.

// ***СРАВНИВАЕМ*** ход пользователя с адресом ячейки.

**ЕСЛИ** ход пользователя совпал,

**ИНКРЕМЕНТИРУЕМ** количество попаданий.

***// ВЫЯСНЯЕМ***, была ли это последняя ячейка.

**ЕСЛИ** количество попаданий равно 3, **ВОЗВРАЩАЕМ** «Потопил».

**ИНАЧЕ** потопления не произошло, то **ВОЗВРАЩАЕМ** «Попал».

КОНЕЦ ВЕТВЛЕНИЯ

**ИНАЧЕ** пользователь не попал, **ВОЗВРАЩАЕМ** «Мимо».

КОНЕЦ ВЕТВЛЕНИЯ

КОНЕЦ ПОВТОРЕНИЯ

КОНЕЦ МЕТОДА

## Как происходит тестирование.

1. Создаем экземпляр класса SimpleDotCom.
2. Присваиваем ему местоположение (массив из трех чисел, например {2,3,4}).
3. Создаем строку для представления хода пользователя («2», «0» и т. д.).
4. Вызываем метод checkYourSelf(), передавая ему вымышленный ход пользователя.
5. Выводим на экран результат, чтобы увидеть, корректно ли сработал код («Пройден» или «Неудача»).

Псевдокод | Тестовый код | Реальный код

## Это не глупые вопросы

**В: Может быть, я что-то упустил, но объясните, как именно вы тестируете элементы, которых даже не существует?**

О: Вы ничего не упустили. Мы никогда и не говорили, что начинать нужно с *выполнения* тестов; вы начинаете с их *написания*. При создании тестового кода у вас нет ничего, на чем можно его опробовать, так что вы, вероятно, не сумеете его скомпилировать, пока не напишете формально работающую «заглушку», но это всегда будет приводить к провалу теста (например, возвращать null).

**В: Я все еще не понимаю, в чем суть. Почему бы не подождать, пока код будет написан, и только затем добавлять тесты?**

О: Тщательное обдумывание тестового кода (и его написания) помогает более четко представить, какие действия требуются от метода.

После реализации метода вам останется только проверить его с помощью уже готового тестового кода. Кроме того, вы *знаете*, что если не напишете его сейчас, то не напишете уже *никогда*. Всегда найдется более интересное занятие.

Старайтесь писать небольшой тестовый код, после чего создавайте реализацию лишь в том объеме, который позволит ей пройти этот тест. Затем напишите *чуть больше* тестового кода и создайте новую реализацию для его прохождения. На каждом этапе запускаются все уже написанные тесты, поэтому вы всегда можете быть уверены, что последние изменения не нарушают код, протестированный ранее.

## Тестовый код для класса SimpleDotCom

```
public class SimpleDotComTestDrive {

   public static void main (String[] args) {

      SimpleDotCom dot = new SimpleDotCom();

      int[] locations = {2,3,4};

      dot.setLocationCells(locations);

      String userGuess = "2";
      String result = dot.checkYourself(userGuess);

      String testResult = "Неудача";

      if (result.equals("Попал") ) {

          testResult = "Пройден";

       }

       System.out.println(testResult);

   }

}
```

Создаем экземпляр класса SimpleDotCom.

Создаем массив для местоположения «сайта» (три последовательных числа из семи).

Вызываем сеттер «сайта».

Делаем ход от имени пользователя.

Вызываем метод checkYourself() объекта SimpleDotCom.

Если ход (2) возвращает строку «Попал», значит, все работает.

Напечатайте результат прохождения теста («Пройден» или «Неудача»).

### Наточите свой карандаш

На нескольких следующих страницах мы реализуем класс SimpleDotCom, а позже вернемся к тестовому классу. Взгляните на тестовый код, приведенный выше. Как его можно дополнить? Какие моменты, которые *следовало бы* проверить, в нем упущены? Изложите свои идеи (или строки кода) здесь:

# Метод checkYourself()

Не существует идеального перехода от псевдокода к коду на языке Java — всегда будут вноситься некоторые изменения. Псевдокод позволил лучше понять, что должен делать метод, и теперь нужно подобрать Java-код, с помощью которого можно выразить, как ему это следует делать.

Посмотрите на этот код и подумайте, что бы вам хотелось в нем улучшить. Числами (1) обозначаются такие особенности синтаксиса и языка, с которыми вы еще не знакомы. Они будут подробно разобраны на следующей странице.

**ПОЛУЧАЕМ** ход пользователя в виде строкового параметра.

**ПРЕОБРАЗУЕМ** полученные данные в *int*.

**ПОВТОРЯЕМ** это с каждой ячейкой массива.

**ЕСЛИ** догадка пользователя совпала,

**ИНКРЕМЕНТИРУЕМ** количество попаданий.

*// **ВЫЯСНЯЕМ**, была ли это последняя ячейка.*
**ЕСЛИ** количество попаданий равно 3,

**ВОЗВРАЩАЕМ** «Потопил» в качестве результата.

**ИНАЧЕ** потопления не произошло, значит, **ВОЗВРАЩАЕМ** «Попал».

**ИНАЧЕ** пользователь не попал, значит,

**ВОЗВРАЩАЕМ** «Мимо».

```
public String checkYourself(String stringGuess) {
    int guess = Integer.parseInt(stringGuess);   (1)
    String result = "Мимо";
    for (int cell : locationCells) {   (2)
        if (guess == cell) {
            result = "Попал";
            numOfHits++;   (3)
            break;   (4)
        } // Конец if
    } // Конец for
    if (numOfHits == locationCells.length) {
        result = "Потопил";
    } // Конец if
    System.out.println(result);
    return result;
} // Конец метода
```

Преобразуем тип String в int.

Создаем переменную для хранения результата, который будем возвращать. Присваиваем по умолчанию строковое значение «Мимо» (то есть подразумеваем промах).

Повторяем с каждой ячейкой из массива locationCells (местоположение ячейки объекта).

Сравниваем ход пользователя с этим элементом (ячейкой) массива.

Мы обнаружили попадание!

Выбираемся из цикла: другие ячейки проверять не нужно.

Мы вышли из цикла, но посмотрим, не потопили ли нас (три попадания), и при необходимости изменим результат на «Потопил».

Выводим пользователю результат («Мимо», если он не был изменен на «Попал» или «Потопил»).

Возвращаем результат в вызывающий метод.

Псевдокод | Тестовый код | Реальный код

## Только новое

На этой странице собрана информация, с которой вы еще не знакомы. Только не волнуйтесь! Ее вполне достаточно, чтобы вы вошли в курс дела. Подробности ждут вас в конце этой главы.

### 1 Преобразование String в int.

Класс, встроенный в Java.

Метод из класса Integer, который «знает», как преобразовывать строку в число.

Принимает строку.

```
Integer.parseInt("3")
```

### 2 Цикл for.

Читайте это объявление цикла for так: «Повторять для каждого элемента в массиве locationCells: взять следующий элемент массива и присвоить его целочисленной переменной cell».

Двоеточие (:) означает «в», поэтому все выражение читается так: «Для каждого значения int в locationCells...»

```
for (int cell : locationCells) {  }
```

Переменная, которая хранит один элемент массива. При каждой итерации она (в данном случае это переменная типа int с именем cell) получает следующий элемент массива, пока элементы не закончатся (или не выполнится оператор break... (см. № 4 чуть ниже)).

Массив, перебор элементов которого происходит в цикле. При каждой новой итерации следующий элемент массива присваивается переменной cell (более подробно об этом в конце главы).

### 3 Постинкрементный оператор.

Оператор ++ добавляет единицу к значению переменной (то есть это инкремент).

```
numOfHits++
```

Выражение numOfHits++ — то же самое (в данном случае), что numOfHits = numOfHits + 1, но более эффективно.

### 4 Оператор break.

```
break;
```

Мгновенно выбрасывает вас из цикла. Прямо здесь. Никаких итераций, условий — сразу на выход!

Псевдокод | Тестовый код | Реальный код

## Это не глупые вопросы

**В:** **Что случится с методом Integer.parseInt(), если передать ему не число? Распознает ли он цифру, записанную словом, например «три»?**

**О:** Integer.parseInt() работает только со строками, которые представляют собой ASCII-значения для цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Если вы попытаетесь разобрать строки вроде «два» или «слоник», возникнет ошибка (то есть будет *запущено исключение*, но мы не будем затрагивать эту тему, пока не дойдем до соответствующей главы; пока же словосочетания «возникнет ошибка» будет достаточно).

**В:** **В самом начале книги приводился пример цикла *for*, который заметно отличался от представленного здесь. Это два разных стиля для циклов *for*?**

**О:** Да! Начиная с первой версии, в Java поддерживался единственный вариант цикла for (мы рассмотрим его позже в этой главе), который выглядел так:

```
for(int i = 0; i < 10; i++) {
  // Выполняем что-нибудь 10 раз
}
```

Вы можете применять этот формат для любых нужных циклов. Но с версии Java 5.0 (Tiger) вы также можете пользоваться *улучшенным* циклом *for* (это официальное описание), когда нужно перебирать элементы массива (или *другого* вида коллекции, как будет показано в следующей главе). Для этих целей сгодится и старый цикл *for*, но его улучшенная версия предоставляет больше возможностей.

## Окончательный вариант кода для SimpleDotCom и SimpleDotComTester

```
public class SimpleDotComTestDrive {

   public static void main (String[] args) {
      SimpleDotCom dot = new SimpleDotCom();
      int[] locations = {2,3,4};
      dot.setLocationCells(locations);
      String userGuess = "2";
      String result = dot.checkYourself(userGuess);
    }
}
```

---

```
public class SimpleDotCom {

   int[] locationCells;
   int numOfHits = 0;

   public void setLocationCells(int[] locs) {
      locationCells = locs;
   }

   public String checkYourself(String stringGuess) {
      int guess = Integer.parseInt(stringGuess);
      String result = "Мимо";
      for (int cell : locationCells) {
         if (guess == cell) {
            result = "Попал";
            numOfHits++;
            break;
         }
      } // Выходим из цикла

      if (numOfHits ==
          locationCells.length) {
          result = "Потопил";
      }
      System.out.println(result);
         return result;
    } // Завершаем метод
} // Завершаем класс
```

Здесь есть небольшая ошибка. Программа компилируется и запускается, но иногда... Сейчас не будем об этом думать, но *придется* разобраться с проблемой чуть позже.

**Что мы должны увидеть при запуске этого кода?**

Тестовый код создает объект SimpleDotCom и передает ему местоположение под номерами 2, 3, 4. Затем он шлет вымышленный пользовательский код 2 в метод checkYourSelf(). Если код работает правильно, то мы должны увидеть следующий результат:

```
java SimpleDotComTestDrive
Попал
```

Псевдокод | Тестовый код | Реальный код

## Наточите свой карандаш

Мы создали тестовый класс и класс SimpleDotCom. Но у нас все еще нет самой *игры*. Пользуясь кодом с предыдущей страницы и техническим заданием для игры, напишите свой вариант псевдокода для игрового класса. Мы добавили несколько готовых строк, чтобы вам было с чего начать. Полная версия псевдокода игры находится на следующей странице, поэтому ***не подглядывайте, пока не закончите это упражнение!***

**МЕТОД** *public static void main (String [] args)*

**ОБЪЯВЛЯЕМ** переменную *numOfGuesses* типа int для хранения количества ходов пользователя.

**ВЫЧИСЛЯЕМ** случайное число от 0 до 4 для местоположения начальной ячейки.

**ПОКА** «сайт» не потоплен:

**ПОЛУЧАЕМ** пользовательский ввод из командной строки.

**Класс SimpleDotComGame делает следующее.**

1. Создает объект SimpleDotCom.
2. Придумывает для него местоположение (три последовательные ячейки в одном ряду).
3. Предлагает пользователю сделать ход.
4. Проверяет введенные данные.
5. Повторяет, пока «сайт» не будет потоплен.
6. Сообщает пользователю, сколько ходов он сделал.

### Полный игровой диалог

```
File Edit Window Help Runaway
%java SimpleDotComGame
Введите число  2
Попал
Введите число  3
Попал
Введите число  4
Мимо
Введите число  1
Потопил
Вам потребовалось 4 попыток(и)
```

# Псевдокод для класса SimpleDotComGame

## Все это находится внутри main()

Есть некоторые вещи, которые нужно принять на веру. Например, одна из строк псевдокода гласит: «ПОЛУЧАЕМ пользовательский ввод из командной строки». На текущий момент это немного больше, чем нам нужно реализовать. К счастью, мы используем ООП, то есть можно попросить *другой* класс/объект о выполнении определенного действия и не задумываться, *как именно* он это сделает. При написании псевдокода вы должны понимать, что *когда-нибудь* у вас будет возможность сделать что угодно, а сейчас нужно направить все умственные усилия на продумывание логики.

```
public static void main (String [] args)
    ОБЪЯВЛЯЕМ переменную numOfGuesses типа int для хранения количества ходов пользователя.
    СОЗДАЕМ новый экземпляр класса SimpleDotCom.
    ВЫЧИСЛЯЕМ случайное число от 0 до 4 для местоположения начальной ячейки.
    СОЗДАЕМ целочисленный массив с тремя элементами, используя сгенерированное случайным
    образом число и увеличивая его на 1, а затем на 2 (например, 3, 4, 5).
    ВЫЗЫВАЕМ метод setLocationCells() из экземпляра SimpleDotCom.
    ОБЪЯВЛЯЕМ булеву переменную isAlive для хранения состояния игры. ПРИСВАИВАЕМ ей значение true.

    ПОКА «сайт» не потоплен (isActive == true):
        ПОЛУЧАЕМ пользовательский ввод из командной строки.
        // ПРОВЕРЯЕМ полученную информацию.
        ВЫЗЫВАЕМ метод checkYourSelf() из экземпляра SimpleDotCom.
        ИНКРЕМЕНТИРУЕМ переменную numOfGuesses.
        // ПРОВЕРЯЕМ, не потоплен ли «сайт».
        ЕСЛИ результат равен Потопил,
            ПРИСВАИВАЕМ переменной isAlive значение false (это значит, что мы не хотим снова заходить в цикл).
            ВЫВОДИМ количество попыток.
        КОНЕЦ ВЕТВЛЕНИЯ
    КОНЕЦ ЦИКЛА
КОНЕЦ МЕТОДА
```

### Общепознавательный совет

Не загружайте одну половину мозга на протяжении длительного времени. Использовать только левую часть более 30 минут — то же самое, что полчаса работать одной левой *рукой*. Устраивайте каждой половине мозга перерыв, меняя их через равные промежутки времени. Загрузив одну сторону, вы даете другой отдохнуть и восстановиться. Левая часть мозга отвечает за такие вещи, как последовательности, решение логических задач и анализ; правая же сторона больше нацелена на метафоры, креативный подход, распознавание и визуализацию.

## КЛЮЧЕВЫЕ МОМЕНТЫ

- Начинать работу над Java-программой нужно с высокоуровневого проектирования.
- Как правило, в процессе создания нового класса следует написать три вещи:

  ***псевдокод;***

  ***тестовый код;***

  ***реальный код (на языке Java).***

- Псевдокод должен описывать, что делать, а не как это делать. Реализация начинается позже.
- Используйте псевдокод для упрощения разработки тестового кода.
- Пишите тестовый код перед реализацией методов.
- Выбирайте цикл *for* вместо *while*, если точно знаете, сколько итераций необходимо выполнить.
- Используйте оператор *пре-/постинкремента* для добавления к переменной единицы (x++;).
- Применяйте оператор *пре-/постдекремента* для вычитания из переменной единицы (x--;).
- Используйте метод `Integer.parseInt()` для получения целого числа из строки.
- `Integer.parseInt()` работает только тогда, когда цифры представлены в строковом значении («0», «1», «2» и т. д.).
- Используйте оператор *break* для преждевременного завершения цикла (даже если проверка условия все еще возвращает true).

Псевдокод | Тестовый код | Реальный код

# Метод main() в игре

Как и в случае с классом SimpleDotCom, подумайте о фрагментах кода, которые вам хочется (или нужно) улучшить. Отметки (1) предназначены для элементов, на которые нам хотелось бы обратить ваше внимание. Они рассматриваются на следующей странице. Мы пропустили создание тестового класса для игры, потому что он просто здесь не нужен. Класс SimpleDotComGame состоит лишь из одного метода, поэтому нет смысла писать для него проверочный код. Делать *отдельный* класс, который будет вызывать метод *main()* из *этого* класса? Нет необходимости.

**ОБЪЯВЛЯЕМ** переменную numOfGuesses типа int для хранения количества ходов пользователя; присваиваем ей 0.

**СОЗДАЕМ** новый экземпляр класс SimpleDotCom.

**ВЫЧИСЛЯЕМ** случайное число от 0 до 4 для местоположения начальной ячейки.

**СОЗДАЕМ** целочисленный массив с местоположением трех ячеек и

**ВЫЗЫВАЕМ** метод setLocationCells() из экземпляра.

**ОБЪЯВЛЯЕМ** булеву переменную isAlive.

**ПОКА** «сайт» не потоплен.

**ПОЛУЧАЕМ** пользовательский ввод.

*// **ПРОВЕРЯЕМ** его.*

**ВЫЗЫВАЕМ** метод checkYourSelf() из SimpleDotCom.

**ИНКРЕМЕНТИРУЕМ** переменную numOfGuesses.

**ЕСЛИ** результат равен «Потопил».

**ПРИСВАИВАЕМ** переменной isAlive значение false.

**ВЫВОДИМ** количество попыток.

```
public static void main(String[] args) {
   int numOfGuesses = 0;
   GameHelper helper = new GameHelper();
   SimpleDotCom theDotCom = new SimpleDotCom();
   int randomNum = (int) (Math.random() * 5);
   int[] locations = {randomNum, randomNum+1, randomNum+2};
   theDotCom.setLocationCells(locations);
   boolean isAlive = true;
   while(isAlive == true) {
      String guess = helper.getUserInput("Введите число");
      String result = theDotCom.checkYourself(guess);
      numOfGuesses++;
      if (result.equals("Потопил")) {
         isAlive = false;
         System.out.println("Вам потребовалось" + numOfGuesses + "попыток(и)");
      } // Завершаем if
   } // Завершаем while
} // Завершаем метод main
```

Создаем переменную, чтобы следить за количеством ходов пользователя.

Это специальный класс, который содержит метод для приема пользовательского ввода. Пока сделаем вид, что это часть Java.

Создаем объект «сайт».

Генерируем случайное число для первой ячейки и используем его для формирования массива ячеек.

Передаем «сайту» местоположение его ячеек (массив).

Создаем булеву переменную, чтобы проверять в цикле, не закончилась ли игра.

Получаем строку, вводимую пользователем.

Просим «сайт» проверить полученные данные; сохраняем возвращенный результат в переменную типа String.

Инкрементируем количество попыток.

Потоплен ли «сайт»? Если да, то присваиваем isAlive значение false (так как не хотим продолжать цикл) и выводим на экран количество попыток.

Псевдокод | Тестовый код | Реальный код

## randomO и getUserInputO

Эта страница посвящена методам random() и getUserInput(). Здесь приведен лишь краткий обзор, позволяющий получить общее представление об их работе.
О классе GameHelper вы подробнее прочтете в конце этой главы.

**1 Генерируем случайное число.**

Это приведение типов, в результате которого последующий элемент меняет свой тип на заданный в скобках. Math.random возвращает double, поэтому необходимо привести его к int (нам нужна последовательность целых чисел между 0 и 4). В процессе отсекается дробная часть double.

Метод Math.random возвращает число от 0 до 1 (не включительно), так что эта формула (включая приведение типов) возвращает число от 0 до 4 (то есть 0–4.999, приведенное к int).

```
int randomNum = (int) (Math.random() * 5)
```

Мы объявляем переменную типа int для хранения случайного числа.

Класс, встроенный в Java.

Метод из класса Math.

**2 Получаем пользовательский ввод с помощью класса GameHelper.**

Ранее созданный нами экземпляр вспомогательного класса. Он называется GameHelper, и вы пока с ним не знакомы (но еще познакомитесь).

Метод принимает строковый аргумент, который применяется для обращения к пользователю в командной строке. Что бы вы ему ни передали, это будет выведено в терминале до того, как программа начнет принимать пользовательский ввод.

```
String guess = helper.getUserInput("Введите число");
```

Мы объявили строковую переменную для хранения пользовательского ввода, который получаем обратно («3», «5» и т. д.).

Метод класса GameHelper, предлагающий пользователю ввести данные. Он читает их после того, как пользователь нажал кнопку Enter, и возвращает результат в виде строки.

# Последний класс: GameHelper

**Мы создали классы *SimpleDotCom* и *SimpleDotComGame*.**

**Остался *вспомогательный* класс,** который содержит метод getUserInput(). Код для получения пользовательского ввода включает в себя элементы, которые мы пока не рассматривали. Сейчас вам многое будет непонятно, поэтому оставим его на потом (до главы 14).

Просто скопируйте* код, приведенный ниже, и скомпилируйте его в класс GameHelper. Поместите все три класса (SimpleDotCom, SimpleDotComGame, GameHelper) в один каталог, который будет рабочим.

Каждый раз, когда вы видите значок

, знайте, что находящийся рядом с ним код нужно просто принять на веру и точно перепечатать. О том, как этот код работает, вы узнаете *позже*.

## Код, готовый к употреблению

```
import java.io.*;
public class GameHelper {
   public String getUserInput(String prompt) {
      String inputLine = null;
      System.out.print(prompt + "  ");
      try {
        BufferedReader is = new BufferedReader(
        new InputStreamReader(System.in));
       inputLine = is.readLine();
       if (inputLine.length() == 0 )  return null;
      } catch (IOException e) {
        System.out.println("IOException: " + e);
      }
      return inputLine;
   }
}
```

*Мы не сомневаемся, что вы обожаете набирать текст, но если у вас будут более важные дела, можете заглянуть на сайт wickedlysmart.com и скопировать код, который мы уже приготовили.

## Сыграем

Вот что произойдет, если мы запустим игру и введем числа 1, 2, 3, 4, 5, 6. Выглядит неплохо.

### Полный игровой диалог

Количество попыток может изменяться

```
File Edit Window Help Smile
%java SimpleDotComGame
Введите число  1
Мимо
Введите число  2
Мимо
Введите число  3
Мимо
Введите число  4
Попал
Введите число  5
Попал
Введите число  6
Потопил
Вам потребовалось 6 попыток(и)
```

## Что такое? Ошибка?

### Только не это!

Вот что будет, если мы введем 1, 1, 1.

### Другой пример игрового диалога

Не может быть!

```
File Edit Window Help Faint
%java SimpleDotComGame
Введите число  1
Попал
Введите число  1
Попал
Введите число  1
Потопил
Вам потребовалось 3 попыток(и)
```

Наточите свой карандаш

**Вот так интрига!**

***Найдем*** ли мы ошибку?

***Исправим*** ли мы ее?

Потерпите до следующей главы, и вы получите ответы на эти вопросы и не только...

А пока попробуйте разобраться, что пошло не так и как это исправить.

# Поговорим о циклах for

В *этой* главе мы смогли описать весь код игры (но в следующей главе мы вернемся к нему, чтобы закончить полную версию). Ранее мы не хотели отвлекать вас подробностями и справочной информацией, поэтому все это собрано здесь. Начнем с особенностей циклов for. Но, если вы программировали на C++, можете смело пропускать следующие несколько страниц.

## Обычные (не улучшенные) циклы for

```
for(int i = 0; i < 100; i++){ }
```

Инициализация.

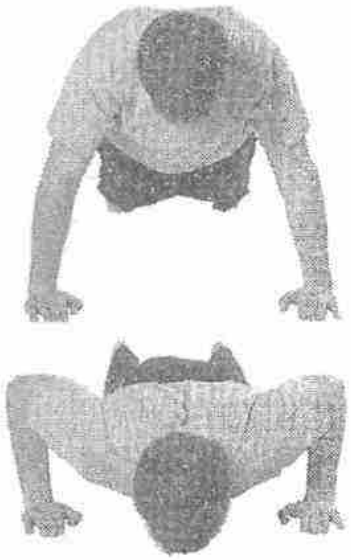
Условие.

Итерационное выражение.

Оператор постинкремента.

Здесь находится код, который будет повторяться (тело цикла).

Повторяйте 100 раз



**На русском языке это будет звучать так:** «Повторять 100 раз».

**Как это выглядит с точки зрения компилятора:**

* создать переменную *i* и присвоить ей 0;
* повторять, пока *i* меньше 100;
* в конце каждой итерации прибавлять к *i* единицу.

**Часть первая: *инициализация.***

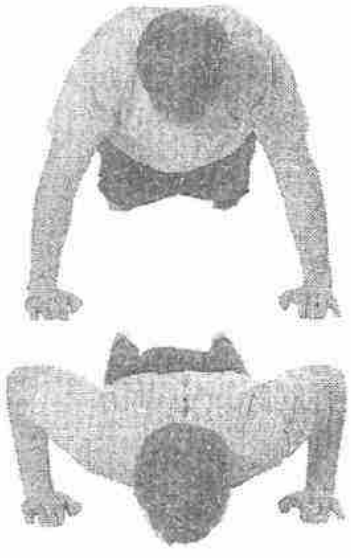
Используйте эту часть для объявления и инициализации переменной, которая задействована внутри тела цикла. Чаще всего она выступает в роли счетчика. На самом деле здесь можно инициализировать сразу несколько переменных, но об этом мы поговорим позже.

**Часть вторая: *условие.***

Это место, где проверяется условие. Что бы здесь ни находилось, оно должно возвращать булево значение (***true*** или ***false***). Вы можете предусмотреть условие вроде (*x* >= 4) или даже вызвать метод, который возвращает значение типа boolean.

**Часть третья: *итерационное выражение.***

Сюда нужно поместить одно или несколько выражений, которые вы хотите выполнять при каждом проходе цикла. Помните, что они выполняются в конце каждой итерации.



# Путешествия сквозь цикл

```
for (int i = 0; i < 8; i++) {
   System.out.println(i);
}
System.out.println("Готово");
```

Результат:

```
File Edit Window Help Repeat
%java Test
0
1
2
3
4
5
6
7
Готово
```

## Разница между for и while

Цикл *while* содержит только условие и не предусматривает встроенной инициализации или итерационного выражения. Он хорош в тех случаях, когда неизвестно, сколько раз нужно повторить тело цикла, и следует продолжать, пока истинно какое-то выражение. Но если вы знаете, сколько раз необходимо пройти через цикл (например, известна длина массива, нужно выполнить действие 7 раз и т. д.), *for* подходит больше. Рассмотрим предыдущий цикл, переписанный с использованием оператора *while*:

```
int i = 0;                    <- Мы должны объявить и инициализировать счетчик.
while (i < 8) {
   System.out.println(i);
   i++;                       <- Нужно инкрементировать счетчик.
}
System.out.println("Готово");
```

## ++ --

**Оператор Пре- и Пост- Инкремента/Декремента**

**Сокращение для добавления или вычитания единицы к/из переменной**

```
x++;
```

**то же самое, что:**

```
x = x + 1;
```

**Оба выражения означают одно и то же в таком контексте: «Добавить 1 к текущему значению x» или «*Инкрементировать* x на 1».**

**И:**

```
x--;
```

**то же самое, что:**

```
x = x - 1;
```

**Но, как всегда, этим все не ограничивается. Расположение оператора (перед или после переменной) может повлиять на результат. Добавляя оператор перед переменной (например, ++x), мы говорим: «*Сначала* инкрементируем x на 1, а потом используем это новое значение». Это важно только в тех случаях, когда ++x применяется не отдельно, а внутри более объемного выражения.**

```
int x = 0;      int z = ++x;
```

**В итоге: x равно 1, z равно 1.**

**Но разместив ++ *после* x, вы получите другой результат:**

```
int x = 0;      int z = x++;
```

**В итоге: x равно 1, но *z равно 0*! Переменная z принимает значение x до того, как та увеличилась на единицу.**

# Улучшенный цикл for

Начиная с версии 5.0 (Tiger), язык Java приобрел еще один вариант цикла `for`, который называют *улучшенным*. Он создавался для упрощенного перебора элементов в массиве или другом виде коллекций (о *других* коллекциях вы узнаете в следующей главе). Улучшенный цикл `for` вскоре стал частью языка, так как по сути делал то же самое, что и обычный `for`, только эффективнее. Мы снова вернемся к нему в следующей главе, когда речь пойдет о коллекциях, отличных от массивов.

**На русском языке это будет звучать так:** «Для каждого элемента в массиве nameArray: присвоить элемент переменной name и запустить тело цикла».

Примечание: некоторые люди в зависимости от используемого ранее языка программирования относятся к улучшенному циклу for как к циклам «for each» или «for in», потому что читают это именно так: «for each thing in the collection...» («для каждого элемента в коллекции...»).

**Как это выглядит с точки зрения компилятора.**

* Создать строковую переменную с именем *name*, присвоить ей значение null.
* Присвоить переменой name значение первого элемента из массива *nameArray*.
* Запустить тело цикла (блок кода, заключенный в фигурные скобки).
* Присвоить переменной name значение следующего элемента из массива *nameArray*.
* Повторять, пока *в массиве остаются элементы*.

**Часть первая: объявление переменной для итераций.**

Используйте эту часть для объявления и инициализации переменной, чтобы работать с ней в теле цикла. С каждой итерацией цикла переменная будет хранить следующий элемент коллекции. Ее тип должен быть совместим с элементами массива! К примеру, вы не можете объявить итерационную переменную типа int для перебора массива типа *String[]*.

**Часть вторая: текущая коллекция.**

Это должна быть ссылка на массив или другую коллекцию. Не волнуйтесь о *других* видах коллекций, отличных от массивов, — вы познакомитесь с ними в следующей главе.

### Преобразование String в int

```
int guess = Integer.parseInt(stringGuess);
```

Пользователь вводит номер ячейки в командной строке, когда программа предлагает ему это сделать. Номер приходит в виде строки («2», «0» и т. д.), которую игра передает в метод checkYourSelf().

Но адреса ячеек представлены в виде целых чисел в массиве, и вы не можете сравнивать int со String.

Например, ***это не будет работать***:

String num = «2»;

int x = 2;

if (x == num) // Ужасный взрыв!

Увидев такой код, компилятор просто посмеется над вами:

```
operator == cannot be applied
      to int,java.lang.String
    if (x == num) { }
         ^
```

Чтобы не попасть впросак, мы должны превратить *строку* «2» в *число* 2. В библиотеке классов Java предусмотрен класс Integer (именно *класс* Integer, а не *простой тип* int), и одна из его функций — принимать строки, *представляющие* числа, и преобразовывать их в *настоящие* числа.

В главе 3 мы говорили о размерах различных примитивов и о том, что нельзя поместить большую вещь в маленькую:

```
long y = 42;
int x = y;     // Не скомпилируется
```

Тип *long* больше int, и компилятор не может знать, где этот тип был раньше. Может, он выпивал вместе с другими переменными типа *long* и «принял» по-настоящему большую дозу. Чтобы заставить компилятор сжать переменную до размеров менее вместительного типа, можно задействовать оператор **приведения**. Он выглядит так:

```
long y = 42;      // Пока все хорошо
int x = (int) y; // x = 42 Отлично!
```

Появившееся здесь приведение типов говорит компилятору взять значение переменной y, обрезать его до размеров int и присвоить переменной x оставшееся. Если значение y оказалось больше, чем максимальное значение x, то мы получим странное (но допустимое*) число:

```
long y = 40002;
// 40002 превышает лимит для типа short в 16 бит
short x = (short) y;  // x теперь равен -25534!
```

Суть в том, что компилятор позволяет это сделать. Допустим, у вас есть число с плавающей точкой и вы просто хотите получить его целую часть (*int*):

```
float f = 3.14f;
int x = (int) f;   //  x теперь равен -25534!
```

Даже не думайте о том, чтобы приводить переменные к типу boolean или наоборот.

** Здесь появляются знаковые разряды, «дополнительный код» и другая компьютерная ерунда, речь о которой пойдет в Приложении Б.*

# Поработайте виртуальной машиной

Java-файл на этой странице — это полноценный исходник. Ваша задача — притвориться JVM и определить, что программа выведет на экран, если ее запустить.

```
class Output {

  public static void main(String [] args) {
    Output o = new Output();
    o.go();
  }

  void go() {
    int y = 7;
    for(int x = 1; x < 8; x++) {
      y++;
      if (x > 4) {
         System.out.print(++y + " ");
      }
      if (y > 14) {
        System.out.println(" x = " + x);
        break;
      }
    }
  }
}
```

```
File Edit Window Help OM
% java Output
12 14
```

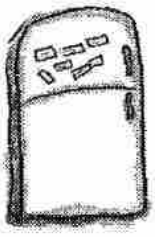
**-ИЛИ-**

```
File Edit Window Help Incense
% java Output
12 14 x = 6
```

**-ИЛИ-**

```
File Edit Window Help Believe
% java Output
13 15 x = 6
```

# Магнитики с кодом



Части рабочего Java-приложения разбросаны по всему холодильнику. Можете ли вы восстановить из фрагментов кода работоспособную программу, которая выведет на экран текст, приведенный ниже? Некоторые фигурные скобки упали на пол. Они настолько малы, что их нельзя поднять. Можете добавлять столько скобок, сколько понадобится!

```
x++;
```

```
if (x == 1) {
```

```
System.out.println(x + " " + y);
```

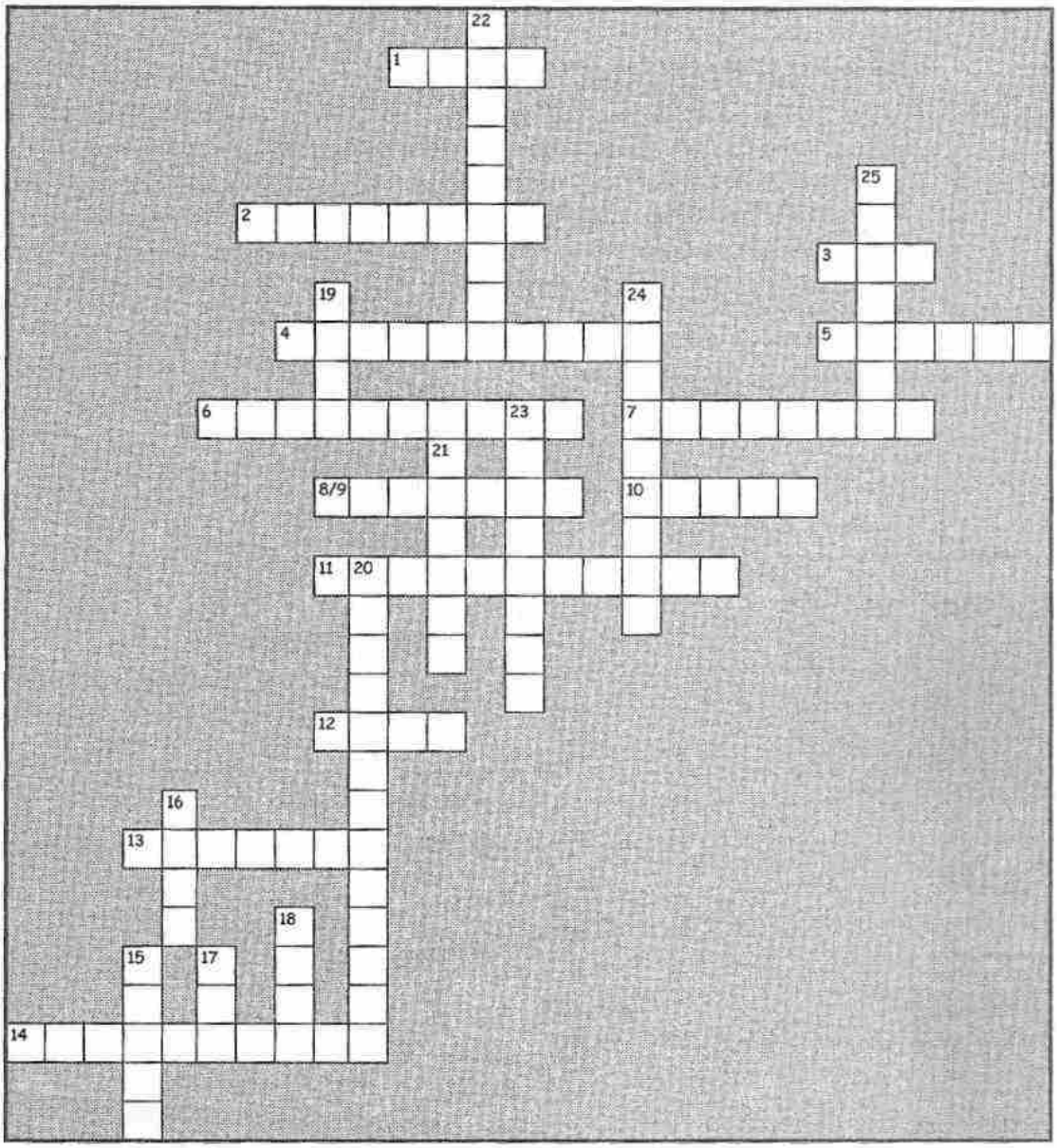
```
class MultiFor {
```

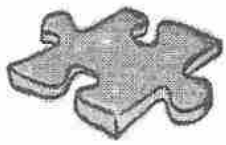
```
for(int  y = 4; y > 2; y--) {
```

```
for(int x = 0; x < 4; x++) {
```

```
public static void main(String [] args) {
```

```
File  Edit  Window  Help  Raid
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```





# JavaCross

Каким образом кроссворд поможет вам в изучении Java? В нем все слова связаны с этим языком. К тому же подсказки содержат метафоры, игру слов и т. п. Такая интеллектуальная зарядка открывает новые подходы к освоению Java.

## По горизонтали

1. Инкремент, который следует после.

2. Инкремент — это один из его видов.

3. Цикл, состоящий из нескольких частей.

4. Изменение.

5. Ответ метода.

6. Причудливое компьютерное слово для создания чего-либо.

7. Метод из класса Math.

8. 32-битный.

9. Выглядит как простой тип, но…

10. Рабочая лошадка у класса.

11. Невсамделишный.

12. While или for.

13. Скомпилируй это и _____.

14. Может быть локальной, а может принадлежать классу.

## По вертикали

15. Помогает уйти раньше времени.

16. Пристанище числа пи.

17. Вид инкремента.

18. То же самое, что ++ .

19. Большой набор инструментов.

20. Установить первое значение.

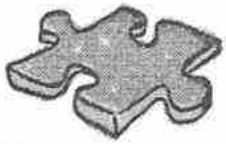
21. Обновляет значение поля.

22. Предварительный код.

23. Цикл.

24. Уменьшает на единицу.

25. Не участвует в приведении типов.



## Смешанные сообщения

**Ниже представлен код небольшой программы на языке Java. Но один ее блок пропал. Ваша задача — найти блоки (слева), которые выведут соответствующий программный результат, если их вставить в код. Не все строки с выводом будут использованы, а некоторые из них могут применяться несколько раз. Соедините линиями блоки кода и подходящий для них результат.**

```
class MixFor5 {
  public static void main(String [] args) {
    int x = 0;
    int y = 30;
    for (int outer = 0; outer < 3; outer++) {
      for(int inner = 4; inner > 1; inner--) {

        [          ]   <-- Сюда нужно вставлять возможные варианты.

        y = y - 2;
        if (x == 6) {
          break;
        }
        x = x + 3;
      }
      y = y - 2;
    }
    System.out.println(x + " " + y);
  }
}
```

**Возможные блоки:**

```
x = x + 3;
x = x + 6;
x = x + 2;
x++;
x--;
x = x + 0;
```

**Возможный результат:**

```
45 6
36 6
54 6
60 10
18 6
6 14
12 14
```

Соедините каждый блок с одним из возможных результатов.

# Ответы

## Поработайте виртуальной машиной

```
class Output {

  public static void main(String [] args) {
    Output o = new Output();
    o.go();
  }
  void go() {
    int y = 7;
    for(int x = 1; x < 8; x++) {
      y++;
      if (x > 4) {
         System.out.print(++y + " ");
      }
      if (y > 14) {
        System.out.println(" x = " + x);
        break;
      }
    }
  }
}
```

**Вы не забыли про оператор break? Как он влияет на результат работы программы?**

```
File Edit Window Help MotorcycleMaintenance
% java Output
13 15 x = 6
```

## Магнитики с кодом

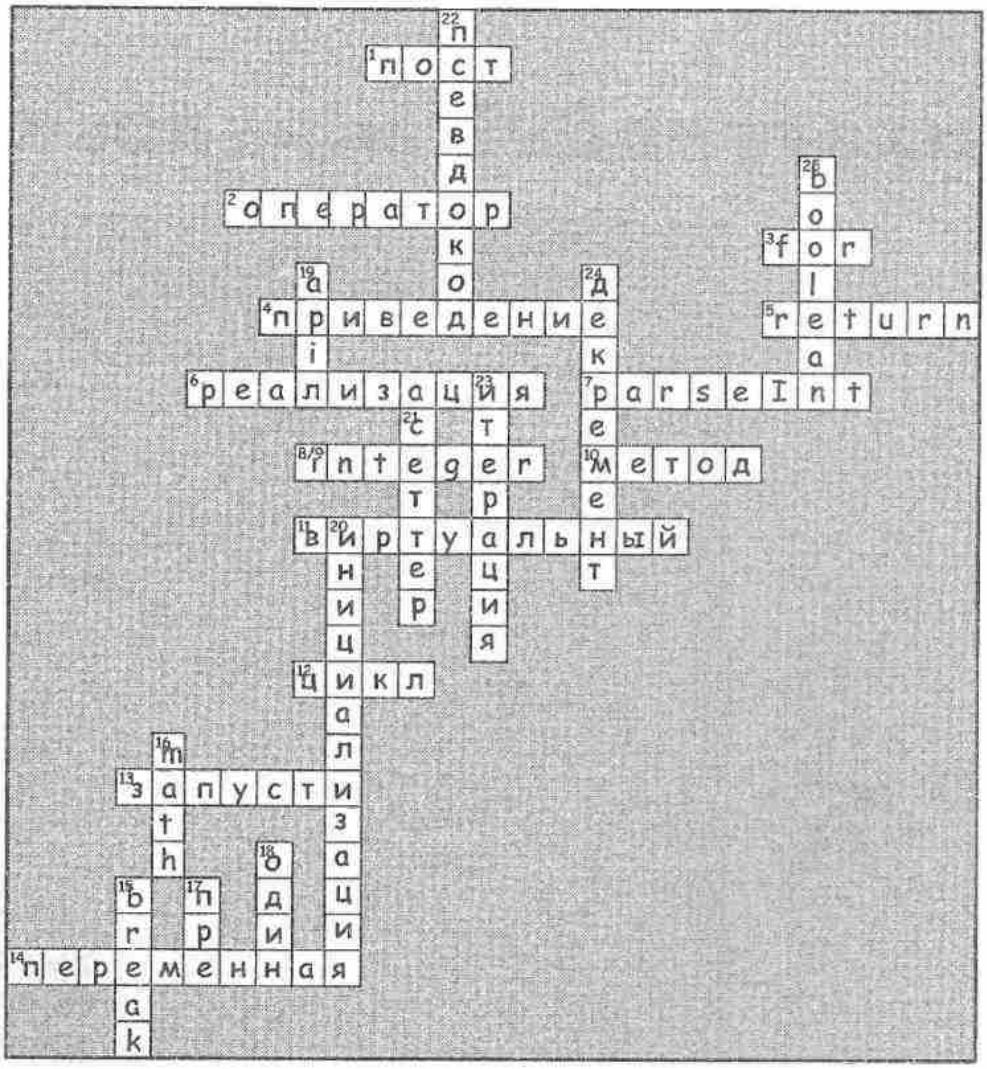
```
class MultiFor {

  public static void main(String [] args) {

    for(int x = 0; x < 4; x++) {

      for(int y = 4; y > 2; y--) {
        System.out.println(x + " " + y);
      }

      if (x == 1) {
        x++;
      }
    }
  }
}
```

**Что бы случилось, если бы этот блок кода шел перед вторым циклом for?**

```
File Edit Window Help Monopole
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```

# Ответы

## JavaCross




## Смешанные сообщения

**Возможные блоки:**

```
x = x + 3;
x = x + 6;
x = x + 2;
x++;
x--;
x = x + 0;
```

**Возможный результат:**

```
45 6
36 6
54 6
60 10
18 6
6 14
12 14
```

# Использование библиотеки Java

**Вместе с Java поставляются сотни готовых классов.** Можете не тратить время на изобретение собственного велосипеда, если знаете, как отыскать нужное в библиотеке Java, называемой еще **Java API**. *Думаем, что у вас найдутся более важные дела.* При написании кода сосредоточьтесь на той части, которая уникальна для вашего приложения. Наверняка вы знаете программистов, которые приходят на работу не раньше 10 часов утра и уходят ровно в 5 часов вечера. **Они используют Java API.** На следующих страницах вы займетесь тем же. Стандартная библиотека Java представляет собой гигантский набор классов, готовых к применению в качестве «строительных блоков». Они позволят вам создавать приложения преимущественно из готового кода. Заранее подготовленный код, который мы приводим в книге, не нужно писать с нуля, хотя набрать его все же придется. Java API содержит большое количество кода, который даже *набирать* не нужно. Все, что от вас требуется, — научиться его использовать.

# В предыдущей главе мы оставили небольшую интригу, а именно — ошибку

## Как это должно выглядеть

Посмотрите, что произойдет, если мы запустим программу и введем числа 1, 2, 3, 4, 5, 6. Выглядит нормально.

**Полная версия игрового диалога**
Количество попыток может изменяться

File Edit Window Help Smile

```
%java SimpleDotComGame
Введите число  1
Мимо
Введите число  2
Мимо
Введите число  3
Мимо
Введите число  4
Попал
Введите число  5
Попал
Введите число  6
Потопил
Вам потребовалось 6 попыток(и)
```

## Где закралась ошибка

Вот что случится, если мы введем 2, 2, 2.

**Другой игровой диалог**
Не может быть!

File Edit Window Help Faint

```
%java SimpleDotComGame
Введите число  2
Попал
Введите число  2
Попал
Введите число  2
Потопил
Вам потребовалось 3 попыток(и)
```

В текущей версии при попадании вы можете просто повторить два раза удачный ход и потопить «сайт»!

# Что же случилось?

**Здесь кроется ошибка. Мы засчитываем попадание каждый раз, когда пользователь угадывает адрес ячейки, *даже если она уже была поражена!***

**Нужно научиться узнавать, попадал ли пользователь в конкретную ячейку. При повторном попадании в эту же ячейку не надо засчитывать ход.**

```
public String checkYourself(String stringGuess) {

    int guess = Integer.parseInt(stringGuess);
    String result = " Мимо";
    for (int cell : locationCells) {
      if (guess == cell) {
        result = "Попал";
        numOfHits++;
        break;
      } // Конец if
    } // Конец for
    if (numOfHits == locationCells.length) {
      result = "Потопил";
    } // Конец if
    System.out.println(result);
    return result;
} // Конец метода
```

Преобразуем тип String в int.

Создаем переменную для хранения результата, который будем возвращать. Присваиваем по умолчанию строку «Мимо» (то есть подразумеваем промах).

Повторяем это с каждым элементом массива.

Сравниваем ход пользователя с этим элементом (ячейкой) массива.

Мы попали!

Выходим из цикла. Нет необходимости проверять другие ячейки.

Выходим из цикла, но проверяем, «потоплены» ли мы (три попадания), и изменяем результат на «Потопил».

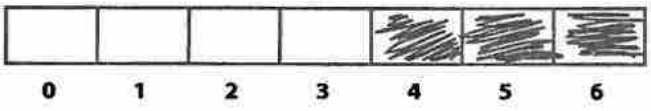
Выводим пользователю результат («Мимо», если он не был изменен на «Попал» или «Потопил»).

Возвращаем результат в вызывающий метод.

# Как мы это исправим?

Нужно найти способ узнавать, попал ли игрок в конкретную ячейку. Рассмотрим возможные варианты, но перед этим сформулируем, что уже известно.

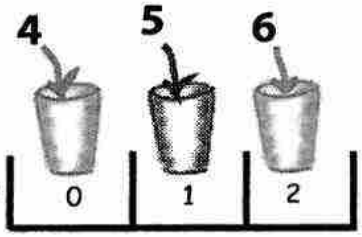
У нас есть виртуальный ряд, состоящий из семи ячеек, три из которых будет занимать объект DotCom. В этом виртуальном ряду показан «сайт», состоящий из ячеек 4, 5 и 6.




Виртуальный ряд с тремя ячейками для объекта DotCom.

Объект DotCom содержит переменную экземпляра — целочисленный массив, хранящий адреса занимаемых ячеек.

**locationCells** (переменная экземпляра объекта DotCom).



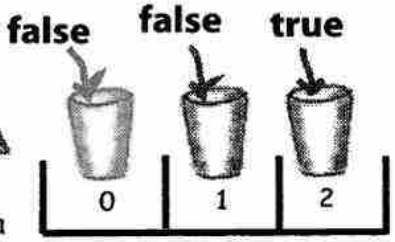

Переменная-массив, которая хранит местоположение ячеек «сайта». Объект DotCom размещен в ячейках 4, 5 и 6. Это числа, которые пользователь должен угадать.

### 1 Вариант первый

Можно создать второй массив и при каждом попадании делать в нем соответствующую запись, а затем проверять, стреляли ли в эту ячейку раньше.

Значение true в ячейке с выбранным индексом говорит о том, что в ячейку с этим же индексом, но в другом массиве (locationCells), уже стреляли.

**Массив hitCalls** (это новая переменная экземпляра в классе DotCom, хранящая булев массив).



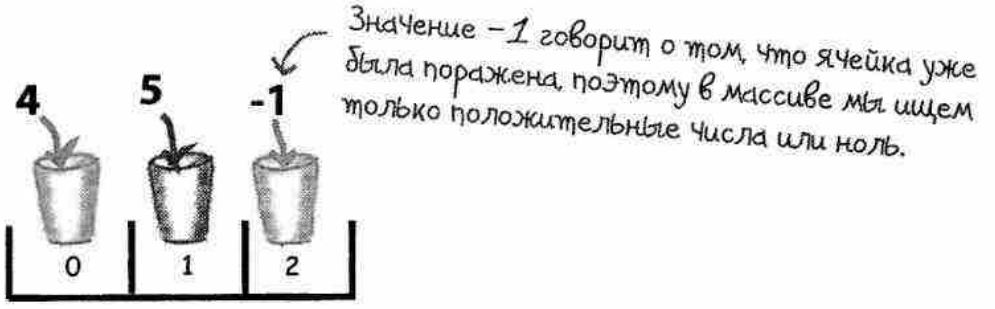

Массив содержит три значения — «состояния» каждой ячейки в массиве с местоположением «сайта». Например, если ячейка с индексом 2 поражена, то элементу с индексом 2 в массиве hitCells присваивается значение true.

# Первый вариант слишком усложнен

В первом варианте выполняется больше действий, чем можно ожидать. При каждом попадании необходимо изменять состояние второго массива (hitCells), но перед этим нужно проверить его и выяснить, не стреляли ли в эту ячейку раньше. Конечно, это будет работать, но должно существовать более изящное решение.

### ② Вариант второй

Можно ограничиться оригинальным массивом, изменяя значение пораженной ячейки на -1. При таком подходе мы будем использовать для проверки и изменений один массив.



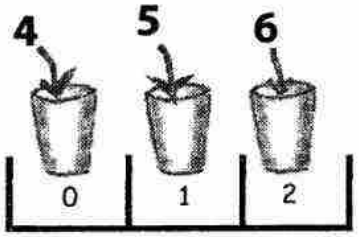

# Второй вариант немного лучше, но все еще сложноват

Второй вариант не такой сложный, как первый, но тоже не отличается особой эффективностью. Вам по-прежнему придется перебирать в цикле три элемента массива, даже если в один или два из них уже попали (и они имеют значение -1). Должно существовать лучшее решение...

## ③ Вариант третий

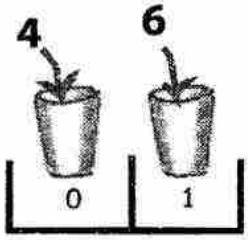
Можно удалить местоположение пораженной ячейки, сделав массив меньше. Однако такие объекты не могут изменять свой размер, поэтому придется создать **новый** массив с меньшей вместимостью и копировать в него оставшиеся ячейки.

**Массив locationCells** до того, как произошло первое попадание.




Сначала массив состоит из трех элементов. Мы перебираем три ячейки (их позиции в массиве), чтобы проверить, совпадает ли пользовательский ход со значением одной из ячеек (4, 5, 6).

**Массив locationCells** после того, как ячейка 5 с индексом 1 была поражена.




Когда пользователь попадает в ячейку 5, мы создаем новый массив меньшего размера для хранения оставшихся ячеек и присваиваем ему ссылку на оригинальный массив locationCells.

**Третий вариант будет куда лучше, если массив сможет уменьшаться. Нам не придется создавать новый массив меньшего размера, копировать оставшиеся значения и переопределять ссылку.**

### Оригинальный псевдокод для части метода checkYourself():

**ПОВТОРЯЕМ** то же самое с каждой ячейкой массива.

*// СРАВНИВАЕМ* ход пользователя с местоположением клетки.

**ЕСЛИ** ход пользователя совпал,

**ИНКРЕМЕНТИРУЕМ** количество попаданий.

*// ВЫЯСНЯЕМ*, была ли это последняя ячейка.

**ЕСЛИ** количество попаданий равно 3, **ВОЗВРАЩАЕМ** «Потопил».

**ИНАЧЕ** потопления не произошло, значит, **ВОЗВРАЩАЕМ** «Попал».

КОНЕЦ ВЕТВЛЕНИЯ

**ИНАЧЕ** пользователь не попал, значит, **ВОЗВРАЩАЕМ** «Мимо».

КОНЕЦ ВЕТВЛЕНИЯ

КОНЕЦ ПОВТОРЕНИЯ

### Жизнь сразу наладится, если мы изменим псевдокод:

**ПОВТОРЯЕМ** то же самое с *оставшимися* ячейками.

*// СРАВНИВАЕМ* ход пользователя с местоположением клетки.

**ЕСЛИ** ход пользователя совпал,

**УДАЛЯЕМ** эту ячейку из массива.

*// ВЫЯСНЯЕМ*, была ли это последняя ячейка.

**ЕСЛИ** массив теперь пустой, **ВОЗВРАЩАЕМ** «Потопил».

**ИНАЧЕ** потопления не произошло, значит, **ВОЗВРАЩАЕМ** «Попал».

КОНЕЦ ВЕТВЛЕНИЯ

**ИНАЧЕ** пользователь не попал, значит, **ВОЗВРАЩАЕМ** «Мимо».

КОНЕЦ ВЕТВЛЕНИЯ

КОНЕЦ ПОВТОРЕНИЯ

Если бы мне удалось найти массив, который будет **сжиматься**, когда я из него что-либо **удалю**. И такой, чтобы не приходилось перебирать все его элементы, а можно было просто **спросить, содержит ли он нужное мне значение**. И пусть бы он позволял забирать элементы, даже если мне неизвестно, в какой ячейке они находятся. Это было бы просто сказочно. Ах, мечты, мечты...

# Проснитесь и почувствуйте дух библиотеки

**Словно по волшебству, такой массив действительно появился.**

**Но это не совсем массив — это *ArrayList*.**

**Это класс из стандартной библиотеки Java (API).**

Java Standard Edition (версия Java, с которой вы сейчас работаете; поверьте, *вы бы знали*, если бы это была Micro Edition, предназначенная для небольших устройств) поставляется с сотнями готовых классов. Это похоже на код, который мы подготовили для вас, где стандартные классы уже скомпилированы.

***Это значит, что их не нужно перепечатывать.***

Просто используйте их.

Один из множества классов в библиотеке Java. Можете применять его в своем коде как собственный класс.

**ArrayList**

**add(Object elem)**
Добавляет в список параметр Object.

**remove(int index)**
Удаляет объект по переданному индексу.

**remove(Object elem)**
Удаляет указанный объект (если он находится внутри ArrayList).

**contains(Object elem)**
Возвращает true, если нашлось совпадение для переданного объекта.

**isEmpty()**
Возвращает true, если список не содержит элементов.

**indexOf(Object elem)**
Возвращает либо индекс объекта, переданного в параметре, либо –1.

**size()**
Возвращает количество элементов в списке на текущий момент.

**get(int index)**
Возвращает объект, который сейчас находится по индексу, переданному в параметре.

Это лишь пример некоторых методов ArrayList.

Примечание: метод add(Object elem) на самом деле выглядит не так просто, как мы здесь показали. Позже мы еще вернемся к его реальной версии. А пока думайте о нем как о методе add(), принимающем объект, который вы хотите добавить.

# Несколько примеров использования ArrayList

1. **Создаем один такой:**

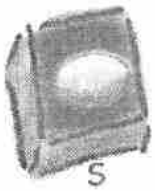
```
ArrayList<Egg> myList = new ArrayList<Egg>();
```

Пока не обращайте внимания на новый синтаксис с угловыми скобками. Это означает «сделай из этого список объектов Egg».

В куче создан новый объект ArrayList. Он маленький, потому что пустой.

2. **Кладем в него что-нибудь:**

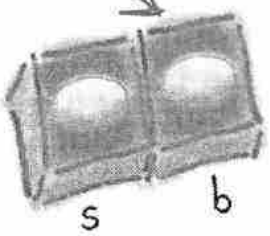
```
Egg s = new Egg();

myList.add(s);
```




Теперь ArrayList «подрос», чтобы вместить объект Egg.

3. **Кладем в него еще что-либо:**

```
Egg b = new Egg();

myList.add(b);
```




ArrayList опять увеличился, чтобы вместить еще один объект Egg.

4. **Выясняем, сколько элементов в нем хранится:**

```
int theSize = myList.size();
```

ArrayList хранит два объекта, поэтому метод size() возвращает 2.

5. **Выясняем, содержит ли он что-либо:**

```
boolean isIn = myList.contains(s);
```

ArrayList действительно содержит объект Egg, на который ссылается s, поэтому contains() возвращает true.

6. **Выясняем, где хранится элемент (то есть его индекс):**

```
int idx = myList.indexOf(b);
```

Индексация элементов в ArrayList начинается с нуля, поэтому indexOf() возвращает 1, хотя объект, на который ссылается b, идет вторым в списке.

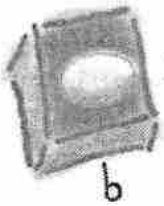
7. **Выясняем, не пустой ли он:**

```
boolean empty = myList.isEmpty();
```

Он точно не пустой, поэтому isEmpty() возвращает false.

8. **Удаляем из него что-нибудь:**

```
myList.remove(s);
```




Эй, смотрите — он уменьшился!

## Наточите свой карандаш

Заполните таблицу, глядя на код слева и подбирая его аналог для обычных массивов. Скорее всего, у вас не получится избежать ошибок, поэтому просто попытайтесь угадать.

| ArrayList | Обычный массив |
|---|---|
| `ArrayList<String> myList = new ArrayList<String>();` | String [] myList = new String[2]; |
| | |
| `String a = new String("Ура");` | String a = new String("Ура"); |
| `myList.add(a);` | |
| | |
| `String b = new String("Лягушка");` | String b = new String("Лягушка"); |
| `myList.add(b);` | |
| | |
| `int theSize = myList.size();` | |
| | |
| `Object o = myList.get(1);` | |
| | |
| `myList.remove(1);` | |
| | |
| `boolean isIn = myList.contains(b);` | |

## Это не глупые вопросы

**В: Да, ArrayList — это круто. Но как я мог знать о его существовании?**

**О:** На самом деле вопрос должен звучать так: «Как мне *узнать*, что находится в API?» Умение работать с библиотекой — ключ к вашему успеху в качестве Java-программиста. Кроме того, она позволяет создавать программное обеспечение без особых трудозатрат. Вы удивитесь, сколько времени можно сэкономить, если кто-то уже сделал за вас тяжелую работу, и остается самая интересная часть.

Но мы немного отвлеклись. Если говорить коротко, то некоторое время придется потратить на изучение стандартных API. Если вам нужен подробный ответ, то вы найдете его в конце этой главы.

**В: Это серьезная проблема. Мало того что я должен знать о существовании ArrayList внутри библиотеки Java, но и, что еще важнее, мне необходимо помнить, что ArrayList выполняет именно те задачи, которые нужны! Как же мне перейти от необходимости что-то сделать к способу выполнить это с помощью API?**

**О:** Теперь вы действительно уловили суть. Дочитав книгу, вы уже будете хорошо ориентироваться в языке и понимать, как пройти путь от постановки задачи до ее решения, создавая минимум кода. Потерпите еще немного — мы начнем обсуждать этот вопрос в конце текущей главы.

## РАЗОБЛАЧЕНИЕ ЯЗЫКА JAVA

### Интервью этой недели: ArrayList о массивах

**HeadFirst:** Итак, объекты ArrayList — это что-то вроде массивов, правильно?

**ArrayList:** Только в их мечтах! Слава Богу, **я** *объект*.

**HeadFirst:** Но если я не ошибаюсь, массивы — это тоже объекты. Как и другие объекты, они обитают в куче.

**ArrayList:** Конечно, массивы находятся в куче, но они лишь пытаются быть похожими на ArrayList. Они подражатели. Объекты обладают состоянием *и* поведением, правильно? Здесь не может быть двух мнений. Но пробовали ли вы когда-нибудь вызывать метод из массива?

**HeadFirst:** Теперь, когда вы об этом упомянули, я понимаю, что никогда так не делал. Но, в любом случае, какой метод мне вызывать? Меня волнуют только методы тех объектов, которые я поместил в массив. И я могу использовать синтаксис массива, чтобы добавлять и удалять из него разные элементы.

**ArrayList:** Разве? Вы хотите сказать, что на самом деле *удаляете* что-либо из массива? Где они вас *готовят*, в MacJava?

**HeadFirst:** *Конечно*, я могу взять что-либо из массива. Я пишу Dog d = dogArray[1] и получаю из него объект Dog по индексу 1.

**ArrayList:** Хорошо, я попытаюсь говорить медленно, чтобы вы могли уследить за ходом моей мысли. Вы *не* удаляете тот объект Dog из массива. Все, что вы делаете, — создаете копию *ссылки на объект Dog* и присваиваете ее другой переменной типа Dog.

**HeadFirst:** Теперь я понимаю, к чему вы клоните. Нет, я фактически не удаляю объект Dog из массива. Он все еще там. Но, мне кажется, я могу легко присвоить его ссылке null.

**ArrayList:** Я первоклассный объект, поэтому поддерживаю методы и на самом деле могу, например, удалить из себя ссылку Dog, а не просто присвоить ей null. И я могу *динамически* изменить свой размер. Попробуйте заставить *массив* сделать то же самое!

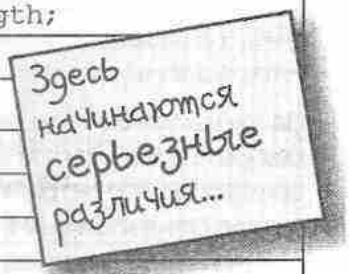
**HeadFirst:** Ничего себе! Не хотелось поднимать эту тему, но ходят слухи, что ваши заслуги очень преувеличены и вы лишь менее эффективный массив. Говорят, что вы простая обертка для массива, добавляющая ему такие способности, как изменение размера, которые я должен писать сам. И раз уж об этом зашла речь, отмечу, что *вы даже не можете хранить примитивы*! Разве это не существенное ограничение?

**ArrayList:** Не могу *поверить*, что вы купились на это. Нет, я *не* просто менее эффективный массив. Иногда бывают ситуации, в которых массив может оказаться слегка *быстрее* меня для одной и той же задачи. Но стоит ли отказываться от всех моих достоинств, чтобы получить этот незначительный выигрыш? А что касается примитивов, конечно же, можно помещать их внутрь ArrayList, используя простые классы-обертки (более подробно об этом вы узнаете в главе 10). В Java 5.0 такое упаковывание (и распаковывание, когда вы получаете примитив обратно) происходит автоматически. Я знаю, что с примитивами быстрее работает массив, так как ему не нужно их упаковывать и распаковывать, но... кто в наши дни всерьез пользуется примитивами?

Ох, вы только взгляните на время! *Я опаздываю на сеанс пилатеса.* Поговорим об этом еще в следующий раз.

# Сравнение ArrayList с обычным массивом

| ArrayList | Обычный массив |
|---|---|
| `ArrayList<String> myList = new ArrayList<String>();` | `String [] myList = new String[2];` |
| | |
| `String a = new String("Ура");` | `String a = new String("Ура");` |
| `myList.add(a);` | `myList[0] = a;` |
| | |
| `String b = new String("Лягушка");` | `String b = new String("Лягушка");` |
| `myList.add(b);` | `myList[1] = b;` |
| | |
| `int theSize = myList.size();` | `int theSize = myList.length;` |
| | |
| `Object o = myList.get(1);` | `String o = myList[1];` |
| | |
| `myList.remove(1);` | `myList[1] = null;` |
| | |
| `boolean isIn = myList.contains(b);` | `boolean isIn = false;`<br>`for (String item : myList) {`<br>`if (b.equals(item)) {`<br>`isIn = true;`<br>`break;`<br>`}`<br>`}` |




Обратите внимание, что с ArrayList вы работаете, как с объектом соответствующего типа, вызывая старые добрые методы из старого доброго объекта и используя старый добрый оператор доступа.

Работая с *массивом*, вы применяете *специальный синтаксис* (такой как miList[0] = foo), который больше нигде не употребляется. Хотя массив и является объектом, он обитает в собственном необычном мире. Нельзя вызвать ни один его метод, вам доступна всего одна переменная экземпляра — *length*.

# Сравнение ArrayList с обычным массивом

**1 Обычный массив в момент создания должен знать свой размер.**

Используя ArrayList, вы каждый раз просто создаете объект соответствующего типа. Объекту все равно, большим он должен быть или маленьким, потому что он увеличивается и уменьшается по мере добавления и удаления объектов.

```
new String[2]
```

Нужно указать размер.

```
new ArrayList<String>()
```

Нет необходимости задавать размер (хотя при желании вы можете это сделать).

**2 При добавлении объекта в обычный массив нужно присвоить ему конкретный индекс.**

Индекс должен быть в промежутке от нуля до числа, меньшего, чем длина массива.

```
myList[1] = b;
```

Требует указания индекса.

Если этот индекс выходит за границы массива (например, массив объявлен с размером 2, а теперь вы пытаетесь присвоить что-нибудь индексу 3), то во время выполнения программы будет выдана ошибка.

С ArrayList вы можете указать индекс через метод *add(anInt, anObject)* или просто написать *add(anObject)*, и объект увеличит свой размер, чтобы вместить новый элемент.

```
myList.add(b);
```

Без индекса.

**3 Массивы используют специальный синтаксис, который в Java больше нигде не применяется.**

Но в случае с ArrayList мы работаем с обычными Java-объектами, которые не имеют особого синтаксиса.

```
myList[1]
```

Квадратные скобки [] — элемент специального синтаксиса, используемого только для массивов.

**4 В Java 5.0 класс ArrayList параметризован.**

Мы сказали, что, в отличие от массивов, ArrayList не имеет специального синтаксиса. Но у него *есть* кое-что необычное, появившееся в Java с версией 5.0 Tiger — это ***параметризованные типы***.

```
ArrayList<String>
```

<String> в угловых скобках — это типовой аргумент. ArrayList<String> означает «список объектов String», а ArrayList<Dog> читается как «список объектов Dog».

До версии 5.0 в Java не было возможности объявлять *тип* сущностей, которые должны храниться в ArrayList, поэтому с точки зрения компилятора все экземпляры ArrayList представляли собой разнородные коллекции объектов. Но теперь, используя синтаксис <типУказываетсяЗдесь>, можно объявить и создать ArrayList, знающий (и ограничивающий) типы объектов, которые он способен хранить. Мы подробно обсудим параметризованные типы, когда дойдем до коллекций, а пока не забивайте себе голову синтаксисом с угловыми скобками <>, который вам будет встречаться при использовании ArrayList. Просто помните, что этот способ заставляет компилятор допускать для ArrayList только конкретный тип объектов *(в угловых скобках)*.

# Исправим код класса DotCom

**Вспомните, как выглядит версия с ошибкой.**

Мы изменили название класса на DotCom (вместо DocComSimple) для новой, улучшенной версии, но это тот же код, который вы могли видеть в предыдущей главе.

```
public class DotCom {

   int[] locationCells;
   int numOfHits = 0;

   public void setLocationCells(int[] locs) {
      locationCells = locs;
   }

   public String checkYourself(String stringGuess) {
      int guess = Integer.parseInt(stringGuess);
      String result = "Мимо";

      for (int cell : locationCells) {
        if (guess == cell) {

          result = "Попал";
          numOfHits++;

          break;
        }
      } // Выходим из цикла

      if (numOfHits == locationCells.length) {
          result = "Потопил";
      }
      System.out.println(result);
         return result;
    } // Конец метода
} // Конец класса
```

Здесь все пошло не так. Мы засчитывали каждый ход как попадание, не проверяя, была ли поражена эта ячейка ранее.

Псевдокод | Тестовый код | Реальный код

# Новый и улучшенный класс DotCom

```
import java.util.ArrayList;

public class DotCom {

   private ArrayList<String> locationCells;
   // private int numOfHits;
   // сейчас это нам не нужно

   public void setLocationCells(ArrayList<String> loc) {
      locationCells = loc;
   }

   public String checkYourself(String userInput) {

      String result = "Мимо";

         int index = locationCells.indexOf(userInput);

         if (index >= 0) {

            locationCells.remove(index);


            if (locationCells.isEmpty()) {
               result = "Потопил";
            } else {
               result = "Попал";
            } // Конец if

         } // Конец внешнего if

      return result;
   } // Конец метода
} // Конец класса
```

Пока не обращайте внимания на эту строку: мы поговорим о ней в конце главы.

Изменяем строковый массив на ArrayList, чтобы хранить объекты String.

Новое и усовершенствованное имя аргумента.

Проверяем, содержится ли загаданная пользователем ячейка внутри ArrayList, запрашивая ее индекс. Если ее нет в списке, то indexOf() возвращает -1.

Если индекс больше или равен нулю, то загаданная пользователем ячейка определенно находится в списке, поэтому удаляем ее.

Если список пустой, значит, это было роковое попадание!

# Создадим настоящую игру «Потопи сайт»

Мы работали над упрощенной версией, но теперь создадим полноценную игру. Вместо единственного ряда мы будем использовать сетку и вместо одного объекта DotCom задействуем сразу три.

**Цель:** потопить все «сайты» компьютера за минимальное количество попыток. Вы будете получать баллы в зависимости от того, насколько хорошо играете.

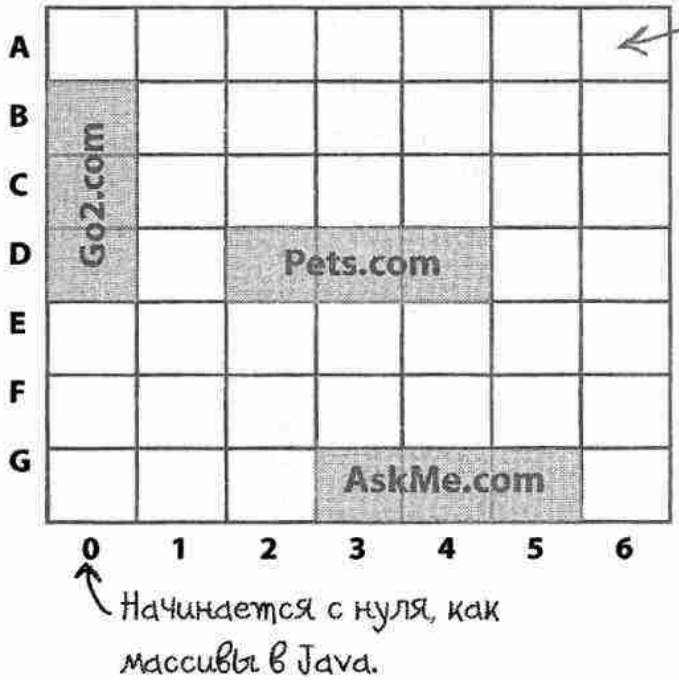
**Подготовка:** при загрузке программы компьютер разместит три «сайта» на **виртуальной доске (7×7)**. После этого вас попросят сделать первый ход.

**Как играть:** вы пока не научились создавать GUI (графический пользовательский интерфейс), поэтому данная версия будет работать в командной строке. Компьютер предложит вам ввести предполагаемую ячейку в виде A3, C5 и т. д. В ответ на это в командной строке вы получите результат — либо «Попал», либо «Мимо», либо «Вы потопили Pets.com» (или любой другой «сайт», который вам посчастливилось потопить в этот день). Как только вы отправите на корм рыбам все три «сайта», игра завершится выводом на экран вашего рейтинга.

**Вам предстоит создать игру «Потопи сайт» с доской 7×7 и тремя «сайтами», каждый из которых занимает три ячейки.**

Доска 7x7.

Каждая секция — это «ячейка».




Начинается с нуля, как массивы в Java.

## Фрагмент игрового диалога

```
File Edit Window Help Sell
%java DotComBust
Сделайте ход  A3
Мимо
Сделайте ход  B2
Мимо
Сделайте ход  C4
Мимо
Сделайте ход  D2
Попал
Сделайте ход  D3
Попал
Сделайте ход  D4
Ой! Вы потопили Pets.com  :(
Потопил
Сделайте ход  B4
Мимо
Сделайте ход  G3
Попал
Сделайте ход  G4
Попал
Сделайте ход  G5
Ой! Вы потопили AskMe.com  :(
```

# Что нужно изменить?

У нас есть три класса, в которые нужно внести изменения: DotCom (раньше назывался DotComSimple), игровой класс (DotComBust) и вспомогательный игровой класс (о нем мы на время забудем).

## A Класс DotCom

- **Добавим переменную name** для хранения имен «сайтов» («Pets.com», «Go2.com» и т. д.). При потоплении каждый «сайт» сможет вывести на экран свое название (см. результат на предыдущей странице).

## B Класс DotComBust (игра)

- **Создадим *три* «сайта» вместо одного.**
- **Дадим каждому из них *имя*.** Вызовем сеттер для каждого экземпляра DotCom и присвоим с его помощью имя переменной экземпляра name.

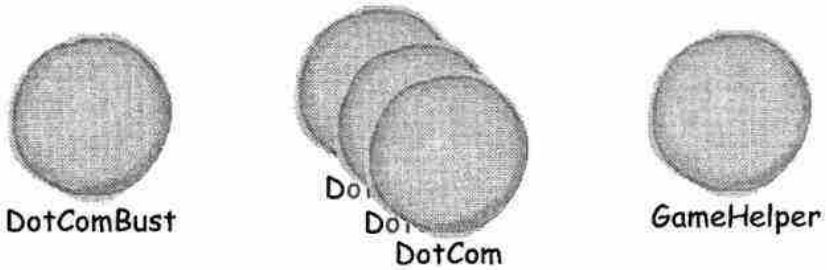
## Продолжим улучшать класс DotComBust...

- **Разместим все три «сайта» на сетке вместо единственного ряда.** Этот этап заметно усложнится, если понадобится размещать объекты DotCom случайным образом. Не углубляясь в математику, добавим в заранее подготовленный класс GameHelper алгоритм для задания «сайтам» местоположения.
- **Будем проверять ход пользователя на *всех трех «сайтах»*, а не на одном.**
- **Продолжим играть** (то есть принимать ходы пользователя и проверять их на оставшихся объектах DotCom) до тех пор, ***пока не останется ни одного живого «сайта»*.**
- **Выйдем из метода main.** Мы не усложняли метод main, просто чтобы... сохранить его простым. Но в *реальной* игре хотелось бы сделать его другим.

---

**Три класса:**

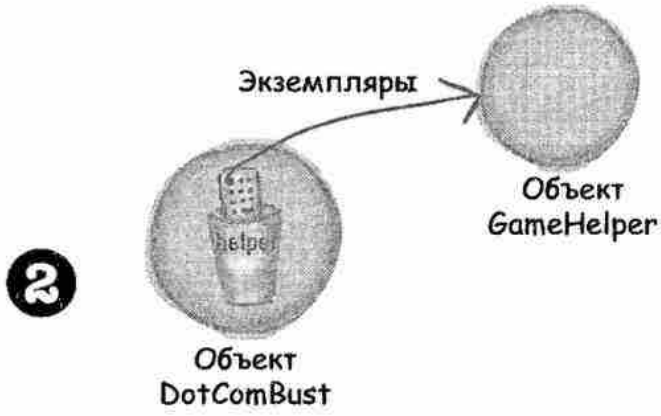
**Пять объектов:**



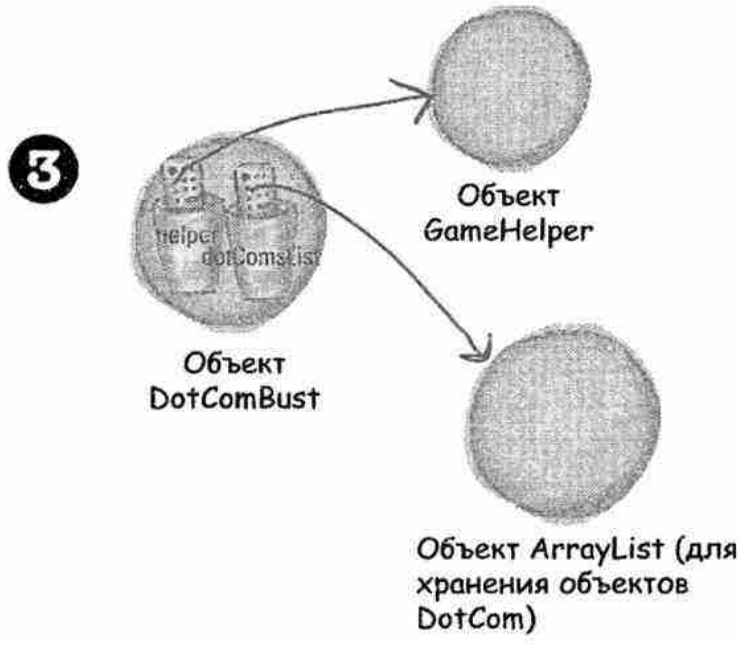

Кроме того, созданы четыре объекта ArrayList: один для DotComBust и по одному для каждого объекта DotCom.

# Что делает каждый элемент в игре DotComBust (и в какой момент)

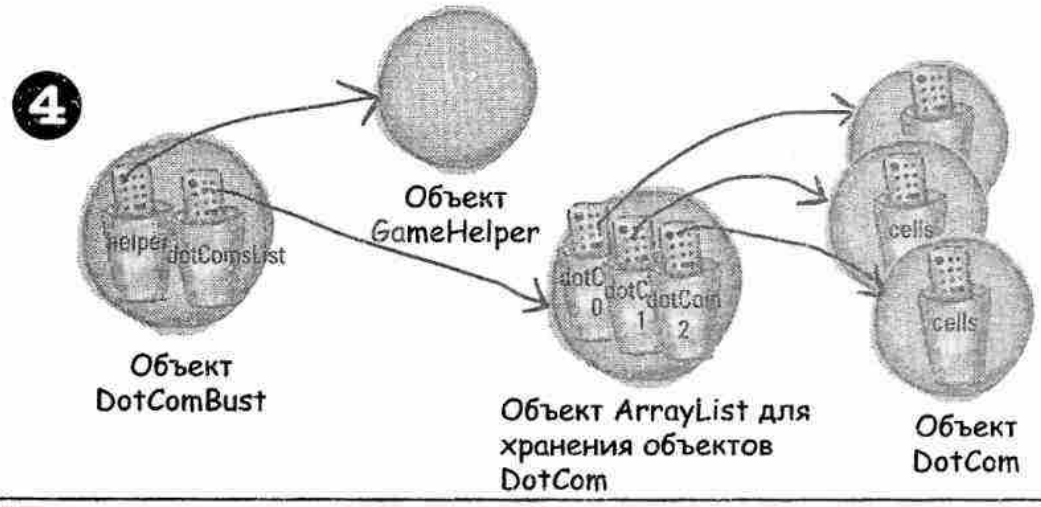
Метод main() класса DotComBust создает объект DotComBust, отвечающий за весь игровой процесс.




DotComBust (игра) создает экземпляр класса GameHelper — объект, который поможет игре выполнять свою работу.



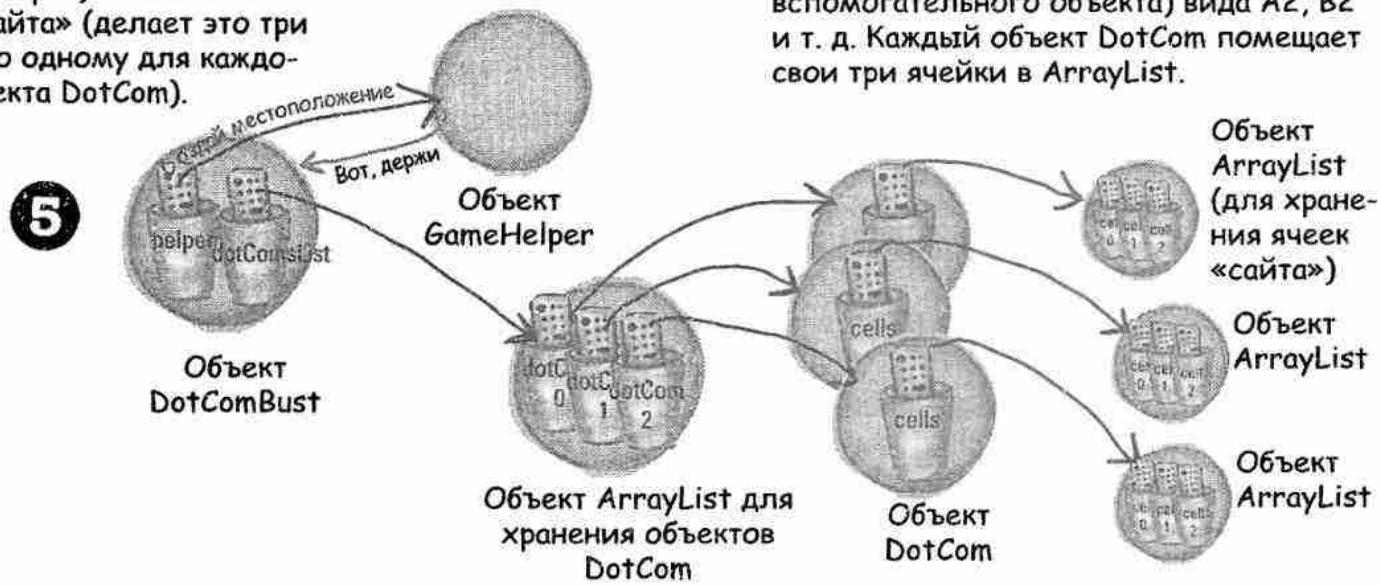

Объект DotComBust создает экземпляр ArrayList, который будет хранить три объекта DotCom.




Объект DotComBust создает три объекта DotCom (и помещает их в ArrayList)

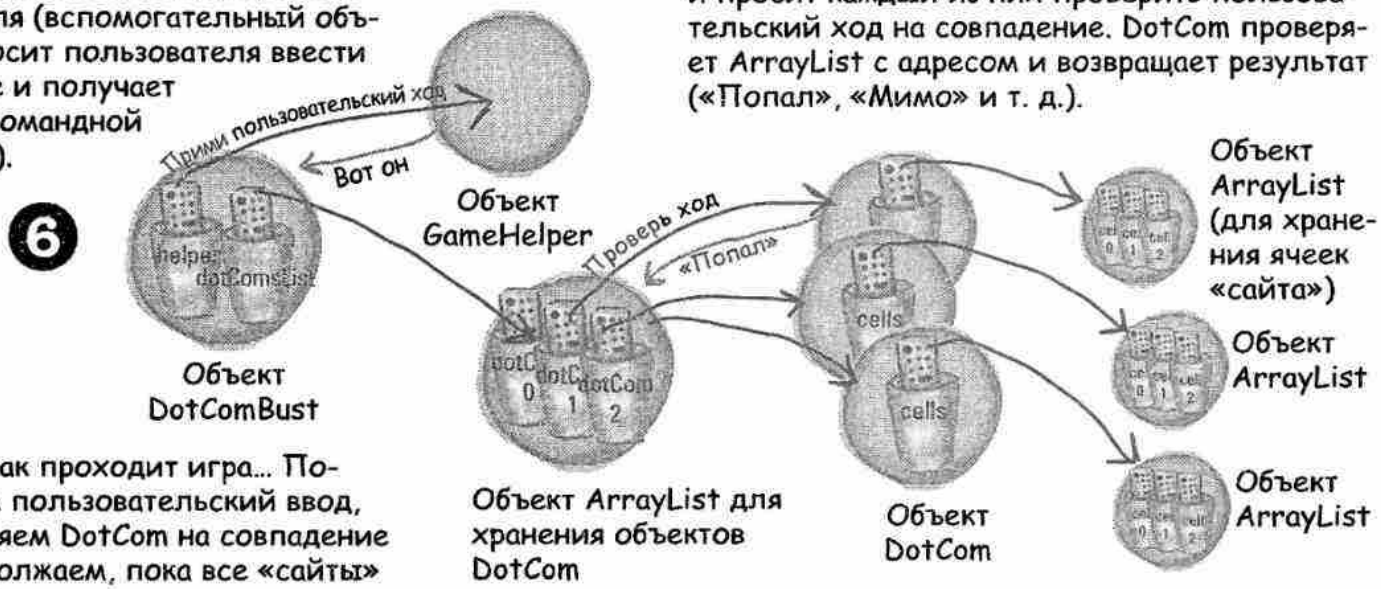
DotComBust запрашивает у вспомогательного объекта (GameHelper) местоположение «сайта» (делает это три раза, по одному для каждого объекта DotCom).

DotComBust выдает каждому объекту DotCom отдельный адрес (с помощью вспомогательного объекта) вида A2, B2 и т. д. Каждый объект DotCom помещает свои три ячейки в ArrayList.

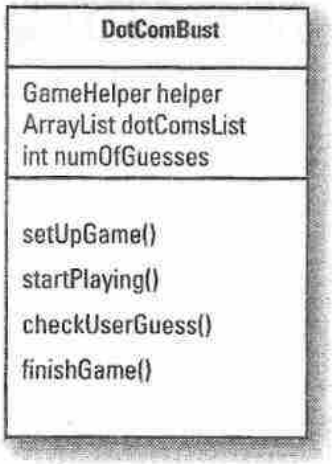



DotComBust запрашивает у вспомогательного объекта ход пользователя (вспомогательный объект просит пользователя ввести данные и получает их из командной строки).

DotComBust перебирает список «сайтов» и просит каждый из них проверить пользовательский ход на совпадение. DotCom проверяет ArrayList с адресом и возвращает результат («Попал», «Мимо» и т. д.).




И вот как проходит игра... Получаем пользовательский ввод, проверяем DotCom на совпадение и продолжаем, пока все «сайты» не будут потоплены.




# Псевдокод для настоящего класса DotComBust

Класс DotComBust выполняет три главные задачи: подготавливает игру, играет с пользователем, пока не будут потоплены все «сайты», и завершает игру. Хотя для этих задач можно было назначить три отдельных метода, мы разбили второй пункт (игра с пользователем) на *два* метода, чтобы повысить модульность. Методы с меньшими размерами лучше поддаются тестированию, отладке и редактированию кода.

## Объявление переменных

**ОБЪЯВЛЯЕМ** переменную ***helper*** и присваиваем ей экземпляр класса ***GameHelper***.

**ОБЪЯВЛЯЕМ** переменную ***dotComList*** и создаем для нее экземпляр ***ArrayList***, который будет хранить список объектов DotCom (изначально три).

**ОБЪЯВЛЯЕМ** целочисленную переменную для хранения количества ходов пользователя (чтобы иметь возможность вывести счет в конце игры). Назовем ее ***numOfGuesses*** и присвоим ей 0.

## Объявление методов

**ОБЪЯВЛЯЕМ** метод ***setUpGame()***, предназначенный для создания объектов DotCom и присвоения им имен и адресов. Он также выводит пользователю краткие инструкции.

**ОБЪЯВЛЯЕМ** метод ***startPlaying()***, который запрашивает у пользователя ход и вызывает метод checkUserGuess(), пока все объекты DotCom не выведены из игры.

**ОБЪЯВЛЯЕМ** метод checkUserGuess(), который просматривает все остальные объекты DotCom и вызывает каждый объект DotCom метода checkYourself().

**ОБЪЯВЛЯЕМ** метод ***finishGame()***, который выводит на экран сообщение об успехах пользователя, основываясь на том, за сколько ходов тот потопил все объекты DotCom.

## Реализация методов

**МЕТОД:** ***void setUpGame()***

*// Создаем три объекта DotCom и назначаем им имена.*

**СОЗДАЕМ** три объекта DotCom.

**УСТАНАВЛИВАЕМ** имя для каждого экземпляра DotCom.

**ДОБАВЛЯЕМ** объекты DotCom в ***dotComList*** (ArrayList).

**ПОВТОРЯЕМ** это с каждым объектом DotCom массива ***dotComList***.

**ВЫЗЫВАЕМ** метод ***placeDotCom()*** из вспомогательного объекта, чтобы получить случайно выбранное местоположение для этого «сайта» (три ячейки на доске 7×7, расположенные вертикально или горизонтально).

**УСТАНАВЛИВАЕМ** местоположение каждого «сайта» исходя из результатов вызова метода ***placeDotCom()***.

КОНЕЦ ПОВТОРЕНИЯ

КОНЕЦ МЕТОДА

Псевдокод | Тестовый код | Реальный код

## Продолжение реализации методов:

**МЕТОД:** ***void startPlaying()***

**ПОВТОРЯЕМ** до тех пор, пока существует хоть один объект DotCom.

**ПОЛУЧАЕМ** пользовательский ввод, вызвав вспомогательный метод ***getUserInput()***.

**ОЦЕНИВАЕМ** ход пользователя с помощью метода ***checkUserGuess()***.

КОНЕЦ ПОВТОРЕНИЯ

КОНЕЦ МЕТОДА

**МЕТОД:** ***void checkUserGuess(String userGuess)***

*// Выясняем, было ли попадание в один из «сайтов» (или его потопление).*

**ИНКРЕМЕНТИРУЕМ** количество ходов пользователя в переменной ***numOfGuesses.***

**УСТАНАВЛИВАЕМ** значение «Мимо» для локальной ***строковой*** переменной ***result***, предполагая, что пользователь промахнулся.

**ПОВТОРЯЕМ** это с каждым объектом DotCom в массиве ***dotComList***.

**ОЦЕНИВАЕМ** ход пользователя, вызывая метод ***checkYourself()*** из объекта DotCom.

**УСТАНАВЛИВАЕМ** результат значения «Попал» или «Потопил», если потребуется.

**ЕСЛИ** результат равен «Потопил», то **УДАЛЯЕМ** DotCom из ***dotComList***.

КОНЕЦ ПОВТОРЕНИЯ

**ВЫВОДИМ** пользователю значение переменной ***result***.

КОНЕЦ МЕТОДА

**МЕТОД:** ***void finishGame()***

**ВЫВОДИМ** сообщение о конце игры, затем:

**ЕСЛИ** количество ходов маленькое,

**ВЫВОДИМ** поздравительное сообщение.

**ИНАЧЕ**

**ВЫВОДИМ** насмешливый вариант сообщения.

КОНЕЦ ВЕТВЛЕНИЯ

КОНЕЦ МЕТОДА

### Наточите свой карандаш

Как мы перейдем от псевдокода к реальной программе? Напишем тестовый код, а затем начнем проверять и достраивать методы бит за битом. В этой книге мы больше не будем показывать вам тестовый код, поэтому вам самим придется подумать, какими сведениями нужно обладать для проверки этих методов. Какой метод вы протестируете и напишете в первую очередь? Попробуйте составить псевдокод для набора тестов. Простого алгоритма или даже списка ключевых моментов будет достаточно для этого упражнения. Но, если вы хотите попробовать написать настоящий тестовый код (на языке Java), смело делайте это.

## Код класса DotComBust (игра)

Псевдокод | Тестовый код | Реальный код

```
import java.util.*;
public class DotComBust {

  (1) private GameHelper helper = new GameHelper();
      private ArrayList<DotCom> dotComsList = new ArrayList<DotCom>();
      private int numOfGuesses = 0;

  private void setUpGame() {
     // Создадим несколько "сайтов" и присвоим им адреса
     DotCom one = new DotCom();
     one.setName("Pets.com");
     DotCom two = new DotCom();
     two.setName("eToys.com");
     DotCom three = new DotCom();
     three.setName("Go2.com");
     dotComsList.add(one);
     dotComsList.add(two);
     dotComsList.add(three);                                   (2)

     System.out.println("Ваша цель — потопить три "сайта".");
     System.out.println("Pets.com, eToys.com, Go2.com");
     System.out.println("Попытайтесь потопить их за минимальное количество ходов");   (3)

     for (DotCom dotComToSet : dotComsList) {                  (4)
        ArrayList<String> newLocation = helper.placeDotCom(3); (5)
        dotComToSet.setLocationCells(newLocation);             (6)
     } // Конец цикла
  } // Конец метода setUpGame

  private void startPlaying() {
     while(!dotComsList.isEmpty()) {                           (7)
       String userGuess = helper.getUserInput("Сделайте ход"); (8)
       checkUserGuess(userGuess);                              (9)
     } // Конец while
     finishGame();                                             (10)
  } // Конец метода startPlaying method
```

Наточите свой карандаш

**Добавьте примечания к коду самостоятельно!**

**Подберите их внизу страницы для соответствующих участков кода, помеченных цифрами. Укажите цифры рядом с подходящими примечаниями.**

**Каждое примечание будет использоваться один раз.**

---

Объявляем и инициализируем переменные, которые нам понадобятся.

Получаем пользовательский ввод.

Запрашиваем у вспомогательного объекта адрес «сайта».

Повторяем с каждым объектом DotCom в списке.

Выводим краткие инструкции для пользователя.

Вызываем сеттер из объекта DotCom, чтобы передать ему местоположение, которое только что получили от вспомогательного объекта.

Вызываем наш метод checkUserGuess.

Вызываем наш метод finishGame.

Создаем три объекта DotCom, даем им имена и помещаем в ArrayList.

До тех пор, пока список объектов DotCom не станет пустым

Псевдокод | Тестовый код | Реальный код

**Что бы ни случилось, не переворачивайте страницу, пока не закончите это упражнение!**

**На следующей странице наша версия.**

```
  private void checkUserGuess(String userGuess) {
      numOfGuesses++; (11)
      String result  = "Мимо"; (12)
      for (DotCom dotComToTest : dotComsList) { (13)
         result = dotComToTest.checkYourself(userGuess); (14)
         if (result.equals("Попал")) {
             break; (15)
         }
         if (result.equals("Потопил")) {
             dotComsList.remove(dotComToTest); (16)
             break;
         }
      } // Конец for
      System.out.println(result); (17)
  } // Конец метода

  private void finishGame() {
      System.out.println("Все "сайты" ушли ко дну! Ваши акции теперь ничего не стоят.");
      if (numOfGuesses <= 18) {
         System.out.println("Это заняло у вас всего" + numOfGuesses + "попыток.");
         System.out.println("Вы успели выбраться до того, как ваши вложения утонули.");
      } else {
         System.out.println("Это заняло у вас довольно много времени. "+ numOfGuesses + "попыток.");
         System.out.println("Рыбы водят хороводы вокруг ваших вложений.");
      }
  } // Конец метода

  public static void main (String[] args) {
      DotComBust game = new DotComBust(); (19)
      game.setUpGame();  (20)
      game.startPlaying(); (21)
  } // Конец метода
}
```

(13) — Повторяем это для всех объектов DotCom в списке.

(14) — Просим DotCom проверить ход пользователя, ищем попадание (или потопление).

(15) — Выбираемся из цикла раньше времени, нет смысла проверять другие «сайты».

(16) — Ему пришел конец, так что удаляем его из списка «сайтов» и выходим из цикла.

(17) — Выводим для пользователя результат.

(11) — Инкрементируем количество попыток, которые сделал пользователь.

(12) — Подразумеваем промах, пока не выяснили обратного.

(13) — Выводим сообщение о том, как пользователь прошел игру.

(19) — Создаем игровой объект.

(20) — Говорим игровому объекту подготовить игру.

(21) — Говорим игровому объекту начать главный игровой цикл (продолжаем запрашивать пользовательский ввод и проверять полученные данные).

## Код класса DotComBust (игра)

Псевдокод | Тестовый код | Реальный код

```
import java.util.*;
public class DotComBust {

    private GameHelper helper = new GameHelper();
    private ArrayList<DotCom> dotComsList = new ArrayList<DotCom>();
    private int numOfGuesses = 0;

  private void setUpGame() {
     // Создадим несколько "сайтов" и присвоим им адреса
     DotCom one = new DotCom();
     one.setName("Pets.com");
     DotCom two = new DotCom();
     two.setName("eToys.com");
     DotCom three = new DotCom();
     three.setName("Go2.com");
     dotComsList.add(one);
     dotComsList.add(two);
     dotComsList.add(three);

     System.out.println("Ваша цель — потопить три "сайта".");
     System.out.println("Pets.com, eToys.com, Go2.com");
     System.out.println("Попытайтесь потопить их за минимальное количество ходов");

      for (DotCom dotComToSet : dotComsList) {

        ArrayList<String> newLocation = helper.placeDotCom(3);

        dotComToSet.setLocationCells(newLocation);

     } // Конец цикла
  } // Конец метода setUpgame

  private void startPlaying() {

    while(!dotComsList.isEmpty()) {

       String userGuess = helper.getUserInput("Сделайте ход");
       checkUserGuess(userGuess);

    } // Конец while
     finishGame();
  } // Конец метода startPlaying
```

Объявляем и инициализируем переменные, которые нам понадобятся.

Создайте ArrayList ТОЛЬКО для объектов DotCom.

Создаем три объекта DotCom, даем им имена и помещаем в ArrayList.

Выводим краткие инструкции для пользователя.

Повторяем с каждым объектом DotCom в списке.

Запрашиваем у вспомогательного объекта адрес «сайта».

Вызываем сеттер из объекта DotCom, чтобы передать ему местоположение, которое только что получили от вспомогательного объекта.

До тех пор, пока список объектов DotCom не станет пустым.

Получаем пользовательский ввод.

Вызываем наш метод checkUserGuess.

Вызываем наш метод finishGame.

Псевдокод | Тестовый код | Реальный код

```
private void checkUserGuess(String userGuess) {
    numOfGuesses++;                                  // Инкрементируем количество попыток, которые сделал пользователь.
    String result = "Мимо";                          // Подразумеваем промах, пока не выяснили обратного.
    for (DotCom dotComToTest : dotComsList) {        // Повторяем это для всех объектов DotCom в списке.
       result = dotComToTest.checkYourself(userGuess); // Просим DotCom проверить ход пользователя, ищем попадание (или потопление).
       if (result.equals("Попал")) {
            break;                                   // Выбираемся из цикла раньше времени, нет смысла проверять другие «сайты».
       }
       if (result.equals("Потопил")) {
            dotComsList.remove(dotComToTest);        // Ему пришел конец, так что удаляем его из списка «сайтов» и выходим из цикла.
            break;
       }
    } // Конец for
    System.out.println(result);                      // Выводим для пользователя результат.
} // Конец метода

private void finishGame() {
    System.out.println("Все "сайты" ушли ко дну! Ваши акции теперь ничего не стоят.");
    if (numOfGuesses <= 18) {
       System.out.println("Это заняло у вас всего" + numOfGuesses + "попыток.");
       System.out.println("Вы успели выбраться до того, как ваши вложения утонули.");
    } else {
       System.out.println("Это заняло у вас довольно много времени. "+ numOfGuesses + "попыток.");
       System.out.println("Рыбы водят хороводы вокруг ваших вложений.");
    }
} // Конец метода

public static void main (String[] args) {
    DotComBust game = new DotComBust();              // Создаем игровой объект
    game.setUpGame();                                // Говорим игровому объекту подготовить игру.
    game.startPlaying();                             // Говорим игровому объекту начать главный игровой цикл (продолжаем запрашивать пользовательский ввод и проверять полученные данные).
} // Конец метода
}
```

Выводим сообщение о том, как пользователь прошел игру.

Псевдокод | Тестовый код | Реальный код

# Окончательная версия класса DotCom

```
import java.util.*;

public class DotCom {
   private ArrayList<String> locationCells;
   private String name;

   public void setLocationCells(ArrayList<String> loc) {
      locationCells = loc;
   }

      public void setName(String n) {
      name = n;
  }

   public String checkYourself(String userInput) {
      String result = "Мимо";
      int index = locationCells.indexOf(userInput);
      if (index >= 0) {
          locationCells.remove(index);

         if (locationCells.isEmpty()) {
             result = "Потопил";
             System.out.println("Ой! Вы потопили" + name + "    : ( ");
         } else {
             result = "Попал";
         }  // Конец if
      } // Конец if
       return result;

   } // Конец метода
} // Конец класса
```

Переменные экземпляра класса DotCom:
- ArrayList с ячейками, описывающими местоположение;
- имя «сайта».

Сеттер, который обновляет местоположение «сайта» (случайный адрес, предоставляемый методом placeDotCom() из класса GameHelper).

Ваш простой сеттер.

Метод indexOf() из ArrayList в действии! Если ход пользователя совпал с одним из элементов ArrayList, то метод indexOf() вернет его местоположение. Если нет, то indexOf() вернет -1.

Применяем метод remove() из ArrayList для удаления элемента.

Используем метод isEmpty(), чтобы проверить, все ли адреса были разгаданы.

Сообщаем пользователю о потоплении «сайта».

Возвращаем «Мимо», «Попал» или «Потопил».

# Супермощные булевы выражения

До этого момента мы работали с довольно простыми булевыми (логическими) выражениями, которые применялись в циклах и условиях. Далее мы воспользуемся более мощными булевыми выражениями в некоторых фрагментах подготовленного кода, который вы еще увидите. Конечно, вы могли бы их и не заметить, но нам кажется, что сейчас самое время поразмыслить над тем, как сделать выражения более мощными.

## Операторы «И» и «ИЛИ» (&&, ||)

Допустим, вы пишете метод chooseCamera(), содержащий множество правил о том, какую камеру нужно выбирать. Может быть, вас устроит ассортимент камер с ценой в пределах $50–1000, но иногда хочется более тонко ограничить ценовой диапазон. Например, нужно указать нечто вроде:

«Если цена находится *в пределах* между $300 ***и*** $400, выбираем X».

```
if (price >= 300 && price < 400) {
    camera = "X";
}
```

Допустим, вам доступно десять марок камер и по некоторым условиям вы можете выбрать *несколько* из них.

```
if (brand.equals("A") || brand.equals("B") ) {
   // Выполняем действия только для марок A или B }
```

Логические выражения могут стать очень большими и сложными:

```
if ((zoomType.equals("Оптический") &&
    (zoomDegree >= 3 && zoomDegree <= 8)) ||
    (zoomType.equals("Цифровой") &&
    (zoomDegree >= 5 && zoomDegree <= 12))) {
   // Выполняем соответствующее масштабирование
}
```

Если вы хотите разобраться в этом *более детально*, то начните с изучения приоритетов операторов. Пока же мы рекомендуем вам ***использовать круглые скобки***, которые позволят сделать код понятным.

## Неравенства (!= и !)

Допустим, есть утверждение вида «Определенное свойство верно для *всех десяти доступных моделей камер, кроме одной*».

```
if (model != 2000) {
   //  Выполняем действия, не относящиеся к модели 2000
}
```

Или нужно сравнить такие объекты, как строки...

```
if (!brand.equals("X")) {
   // Выполняем действия, не относящиеся к марке X}
}
```

## Укороченные логические операторы (&&, ||)

Операторы && и ||, которые мы до сих пор изучали, также известны как ***укороченные*** операторы. В случае с && выражение будет истинным, только если обе его части истинны. Таким образом, если JVM видит, что левая часть выражения && возвращает false, она тут же останавливается! Она даже не подумает взглянуть на правую сторону.

То же самое происходит с оператором ||: выражение будет истинно, если любая из его частей истинна. Если JVM видит, что левая сторона возвращает true, то она объявляет все выражение истинным и не проверяет правую часть.

Почему это хорошо? Допустим, у вас есть ссылочная переменная и вы не знаете, связана ли она с объектом. Если вы попытаетесь вызвать метод из этой нулевой ссылки (которая не указывает на объект), то получите исключение NullPointerException. Поэтому попробуйте сделать так:

```
if (refVar != null &&
     refVar.isValidType() ) {
   // Выполняем действие "получить допустимый тип"
}
```

## Простые логические операторы (&, |)

При использовании с булевыми выражениями операторы & и | ведут себя так же, как их братья-близнецы && и ||, за исключением того, что они заставляют JVM всегда проверять обе части выражения. Как правило, операторы & и | применяются в других ситуациях, например для работы с битами.

## Готовый к употреблению: класс GameHelper

### Код, готовый к употреблению

Это вспомогательный класс для игры. Если не считать метода для приема пользовательского ввода (который предлагает пользователю ввести данные и считывает их из командной строки), то его главная задача состоит в создании ячеек с адресами объектов DotCom. На вашем месте мы бы набрали и скомпилировали этот код и больше к нему не прикасались. Мы старались сделать его как можно проще, чтобы вам не пришлось набирать слишком много текста, но это ухудшило читаемость кода. Помните, что вы не сможете скомпилировать игровой класс DotComBust, пока не наберете этот код.

```
import java.io.*;
import java.util.*;

public class GameHelper {

  private static final String alphabet = "abcdefg";
  private int gridLength = 7;
  private int gridSize = 49;
  private int [] grid = new int[gridSize];
  private int comCount = 0;

  public String getUserInput(String prompt) {
     String inputLine = null;
     System.out.print(prompt + "  ");
     try {
       BufferedReader is = new BufferedReader(
          new InputStreamReader(System.in));
       inputLine = is.readLine();
       if (inputLine.length() == 0 )  return null;
     } catch (IOException e) {
       System.out.println("IOException: " + e);
     }
     return inputLine.toLowerCase();
  }

 public ArrayList<String> placeDotCom(int comSize) {
    ArrayList<String> alphaCells = new ArrayList<String>();
    String [] alphacoords = new String [comSize];      // Хранит координаты типа f6
    String temp = null;                                // Временная строка для конкатенации
    int [] coords = new int[comSize];                  // Координаты текущего "сайта"
    int attempts = 0;                                  // Счетчик текущих попыток
    boolean success = false;                           // Нашли подходящее местоположение?
    int location = 0;                                  // Текущее начальное местоположение

    comCount++;                                        // Энный "сайт" для размещения
    int incr = 1;                                      // Устанавливаем горизонтальный инкремент
    if ((comCount % 2) == 1) {                         // Если нечетный (размещаем вертикально)
      incr = gridLength;                               // Устанавливаем вертикальный инкремент
    }

    while ( !success & attempts++ < 200 ) {            // Главный поисковый цикл (32)
         location = (int) (Math.random() * gridSize);    // Получаем случайную стартовую точку
        //System.out.print("пробуем" + location);
         int x = 0;                                    // Энная позиция в "сайте", который нужно разместить
        success = true;                                // Предполагаем успешный исход
        while (success && x < comSize) {               // Ищем соседнюю неиспользованную ячейку
          if (grid[location] == 0) {                   // Если еще не используется
```

Примечание: чтобы заработать «дополнительные очки», можете попробовать «раскомментировать» строки с System.out.print(ln) в методе placeDotCom(). Это позволит убедиться в том, что метод работает! На экране вы увидите выражения с координатами «сайтов», которые позволят вам «схитрить» при игре, а также помогут проверить работоспособность данного метода.

**Код, готовый к употреблению**

## Продолжение кода GameHelper...

```
        coords[x++] = location;                    // Сохраняем местоположение
        location += incr;                          // Пробуем "следующую" соседнюю ячейку
        if (location >= gridSize){                 // Вышли за рамки – низ
          success = false;                         // Неудача
        }
        if (x>0 && (location % gridLength == 0)) { // Вышли за рамки – правый край
          success = false;                         // Неудача
        }
      } else {                                     // Нашли уже использующееся местоположение
        // System.out.print("используется" + location);
        success = false;                           // Неудача
      }
    }
  }                                                // Конец while

  int x = 0;                                       // Переводим местоположение в символьные координаты
  int row = 0;
  int column = 0;
  // System.out.println("\n");
  while (x < comSize) {
    grid[coords[x]] = 1;                           // Помечаем ячейки на главной сетке как "использованные"
    row = (int) (coords[x] / gridLength);          // Получаем значение строки
    column = coords[x] % gridLength;               // Получаем числовое значение столбца
    temp = String.valueOf(alphabet.charAt(column)); // Преобразуем его в строковый символ

    alphaCells.add(temp.concat(Integer.toString(row)));
    x++;
    // System.out.print("  coord "+x+" = " + alphaCells.get(x-1));
  }

  // System.out.println("\n");

  return alphaCells;
 }
}
```

Это выражение говорит вам о том, где именно находится «сайт».

# Использование библиотеки (Java API)

Итак, не без помощи ArrayList вы прошли все этапы создания игры «Потони сайт». Теперь, как и было обещано, пришло время научиться работать с библиотекой Java.

**В Java API классы сгруппированы в пакеты.**

**Чтобы использовать класс из API, нужно знать, в каком пакете он находится.**

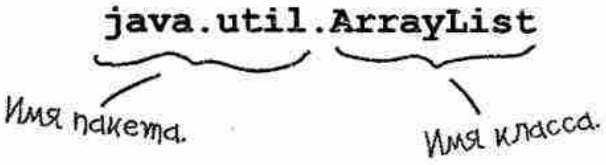
Любой класс из библиотеки Java принадлежит определенному пакету. У пакета есть имя, например **javax.swing** (пакет с классами пользовательского графического интерфейса (GUI) Swing, с которыми вы скоро познакомитесь). Класс ArrayList находится в пакете **java.util**, который содержит набор *классов-утилит*. В главе 16 вы узнаете много нового о пакетах, в частности научитесь помещать собственные классы в *собственные* пакеты. На этом пока остановимся, ведь мы хотим просто *применять* классы, которые поставляются вместе с Java.

Использовать в своем коде классы из API легко. Вы просто работаете с ними так, как будто сами их написали, скомпилировали, и теперь они ждут, когда вы их задействуете. Но есть одно существенное отличие: где-то в своем коде вы должны указать *полное* имя нужного библиотечного класса в виде «имя пакета + имя класса».

Сами того не подозревая, ***вы уже используете классы из пакета***. Классы System (System.out.println), String и Math (Math.random()) находятся в пакете **java.lang**.

## Необходимо знать полное имя* класса, который вы хотите использовать в своем коде.

ArrayList — это не *полное* имя класса ArrayList, как и Катя — сокращение от имени Екатерина. Полное имя ArrayList на самом деле:




**Вы должны сказать Java, какой именно ArrayList хотите использовать. У вас есть два варианта:**

### A ИМПОРТИРОВАТЬ

Поместить оператор import в начале своего исходного файла:

```
import java.util.ArrayList;
public class MyClass {... }
```

**ИЛИ**

### B НАБРАТЬ

Набрать полное имя класса в любом месте своего кода. Каждый раз, когда вы его используете. *Везде*, где вы его используете.

При объявлении и/или создании его экземпляра:

```
java.util.ArrayList<Dog> list = new java.util.ArrayList<Dog>();
```

При использовании его в качестве типа аргумента:

```
public void go(java.util.ArrayList<Dog> list) { }
```

При использовании его в качестве типа возвращаемого значения:

```
public java.util.ArrayList<Dog> foo() {...}
```

*Если только этот класс не из пакета java.lang.

## Это не глупые вопросы

**В: Почему нужно указывать полное имя? Это единственное предназначение пакетов?**

О: Пакеты важны по трем основным причинам. Во-первых, они помогают при общей организации проекта или библиотеки. Вместо того чтобы смешивать классы в одну большую кучу, их группируют в пакеты исходя из функций (например, GUI, структуры данных или базы данных и т. д.).

Во-вторых, благодаря пакетам вы получаете области видимости, которые помогут предотвратить конфликт, если вы и еще 12 других программистов вашей компании вдруг решите создать класс с одним и тем же именем. Если у вас создан класс Set и где-нибудь еще (включая Java API) имеется класс Set, то вам нужен способ сообщить JVM о том, *какой* именно из этих классов вы пытаетесь использовать.

В-третьих, пакеты предоставляют еще один уровень безопасности, так как вы можете ограничить доступ к своему коду только теми классами, которые находятся в одном пакете. Всю информацию об этом вы получите в главе 16.

**В: Вернемся к конфликтам имен. Чем здесь поможет полное имя? Что помешает двоим разработчикам дать пакету одно имя?**

О: В Java включено соглашение об именовании. Если разработчики его придерживаются, то таких ситуаций, как правило, не возникает. В главе 16 мы изучим этот вопрос более подробно.

## Откуда появилась буква «x», или Что значит, когда пакет начинается с javax?

В первых двух версиях Java (1.02 и 1.1) все поставляемые классы (другими словами, стандартная библиотека) находились в пакетах, начинающихся со слова ***java***. Среди них, конечно же, всегда присутствовал ***java.lang*** — пакет, который не нужно импортировать. Кроме того, там были ***java.net***, ***java.io***, ***java.util*** (хотя там не было класса ArrayList) и несколько других пакетов, включая ***java.awt*** с классами, связанными с GUI.

Были и иные пакеты, не включенные в стандартную библиотеку. Их классы известны как ***расширения*** и делятся на две группы: *стандартные* и нестандартные. К первым принадлежат классы, которые компания Sun считает официальными, в отличие от экспериментальных пакетов, пребывающих на ранней стадии разработки или в статусе «бета». Последние могут так никогда и не попасть в официальную поставку.

Имена всех стандартных расширений, исходя из соглашения, начинаются со слова ***java***, к которому добавляется буква «x». Прародителем всех стандартных расширений считается библиотека Swing. Она включает несколько пакетов, названия которых начинаются с ***javax.swing***.

Но стандартные расширения могут быть приравнены к первоклассным, поставляемым вместе с Java и входящим в стандартную библиотеку пакетам. Именно это и произошло со Swing начиная с версии 1.2 (которая в итоге превратилась в первую версию из ветки Java 2).

«Здорово, — подумали разработчики. — Теперь все пользователи Java будут иметь классы Swing и нам не придется думать об их установке».

Однако здесь таилась одна проблема, так как пакет с новым статусом, конечно же, должен был иметь имя, начинающееся с ***java***, а не с ***javax***. Все знают, что пакеты в стандартной библиотеке не содержат в своем названии букву «x», которая бывает только у расширений. И вот перед самым выходом версии 1.2 компания Sun наряду с другими корректировками изменила имена пакетов и убрала «x». В свежих книгах, лежавших на полках магазинов, описывался код Swing с использованием новых имен. Соглашения об именовании были соблюдены. В мире Java царил порядок.

Все было хорошо, если не считать примерно 20 тысяч программистов, рвущих на себе волосы. Они вмиг осознали, что это невинное изменение в именах ведет к катастрофе! Необходимо было исправить весь код с использованием Swing! Кошмар! Представьте себе все операторы import, которые начинались с ***javax***...

В итоге, отчаявшись и потеряв надежду, разработчики убедили компанию Sun «наплевать на соглашения и спасти их код». Остальное — уже история. Поэтому, когда вы видите в библиотеке пакет, название которого начинается с ***javax***, знайте, что он появился на свет как расширение, а потом повысил свой статус.

### КЛЮЧЕВЫЕ МОМЕНТЫ

- **ArrayList** — класс из Java API.
- Чтобы добавить элемент в ArrayList, используйте метод **add()**.
- Для удаления элемента из ArrayList применяйте **remove()**.
- Чтобы выяснить, где внутри ArrayList находится (и находится ли вообще) элемент, используйте **indexOf()**.
- Чтобы выяснить, пуст ли ArrayList, используйте **isEmpty()**.
- Если хотите получить размер (количество элементов) ArrayList, используйте метод **size()**.
- Помните, что для получения длины (количества элементов) обычного массива следует применять **переменную length**.
- ArrayList при необходимости **динамически изменяет** свой размер. Он увеличивается при добавлении объектов и **уменьшается** при их удалении.
- Тип массива объявляется с помощью **типового параметра**, который представляет собой имя типа в угловых скобках. Например, ArrayList<Button> означает, что ArrayList способен хранить только объекты типа Button (или его дочерних классов, о чем будет рассказано в следующих главах).
- Несмотря на то что ArrayList хранит объекты, а не примитивы, компилятор автоматически «упакует» (и «распакует», когда вы попытаетесь его достать) примитив в объект и поместит его в ArrayList (подробнее об этой возможности будет рассказано далее в книге).
- Классы группируются в пакеты.
- У класса есть полное имя, которое состоит из его обычного имени и имени пакета. Класс ArrayList в действительности называется java.util.ArrayList.
- Чтобы использовать класс из пакета (если это не java.lang), нужно сообщить Java его полное имя.
- Вы можете либо указывать оператор import в начале исходного кода, либо набирать полное имя класса при каждом его упоминании.

## Это не глупые вопросы

**В: Увеличивает ли оператор `import` размер моего класса? Добавляет ли он в *мой* код при компиляции импортируемые классы и пакеты?**

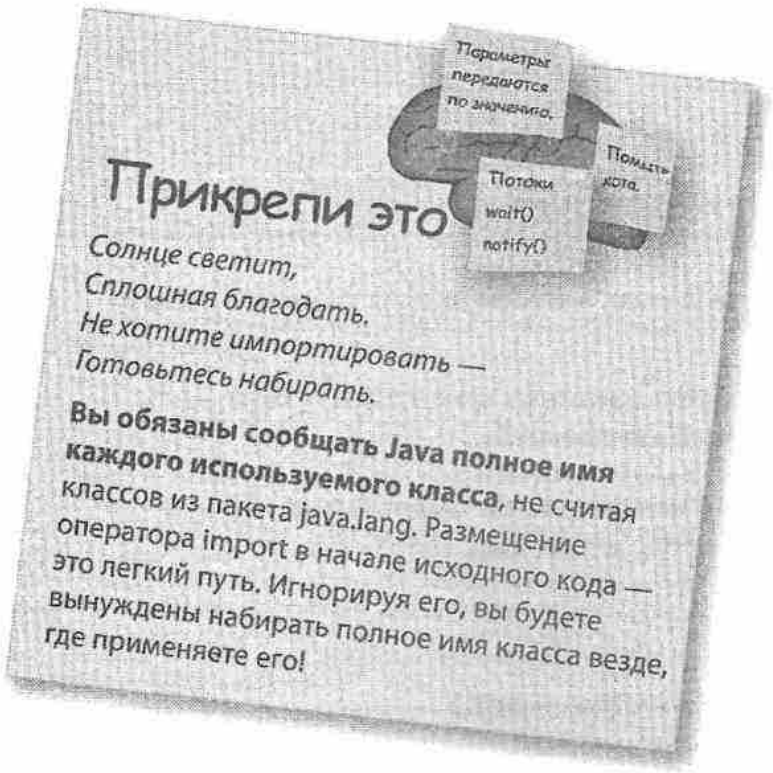
О: Вы, должно быть, программируете на языке C. Оператор `import` отличается от директивы `include`, поэтому на оба вопроса ответ будет один — нет. Оператор `import` избавляет от набора лишнего кода. Вам не нужно беспокоиться о том, что из-за слишком большого количества выражений импорта код получится раздутым или будет медленно работать. Оператор `import` позволяет передать Java *полное имя класса*.

**В: Хорошо, тогда почему я никогда не импортирую класс String? Или System?**

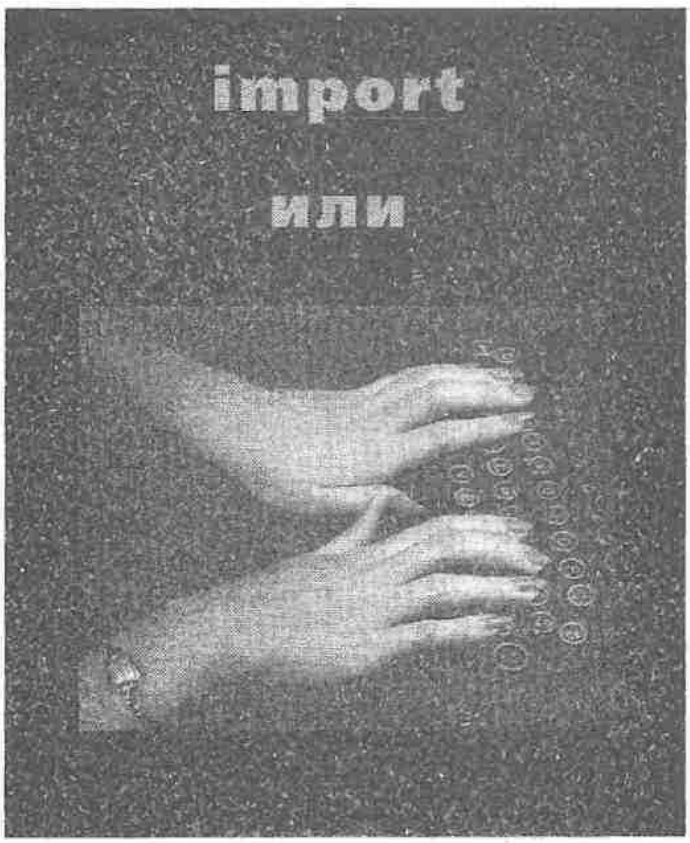
О: Запомните, пакет java.lang импортируется как бы заранее, без вашего участия. Поскольку классы в java.lang весьма основательны, вам не нужно использовать для них полные имена. Классы java.lang.String и java.lang.System существуют лишь в одном экземпляре, и Java очень хорошо осведомлен о том, где их искать.

**В: Должен ли я размещать свои классы внутри пакетов? Как мне это сделать? И могу ли я это сделать?**

О: В реальном мире у вас *будет* необходимость размещать классы в пакетах. Более детально мы поговорим об этом в главе 16. А пока классы из наших примеров будут существовать вне пакетов.




**Если вы все еще этого не поняли:**



*Хорошо, когда известно, что ArrayList находится в пакете java.util. Но как бы **я** смогла додуматься до этого самостоятельно?*

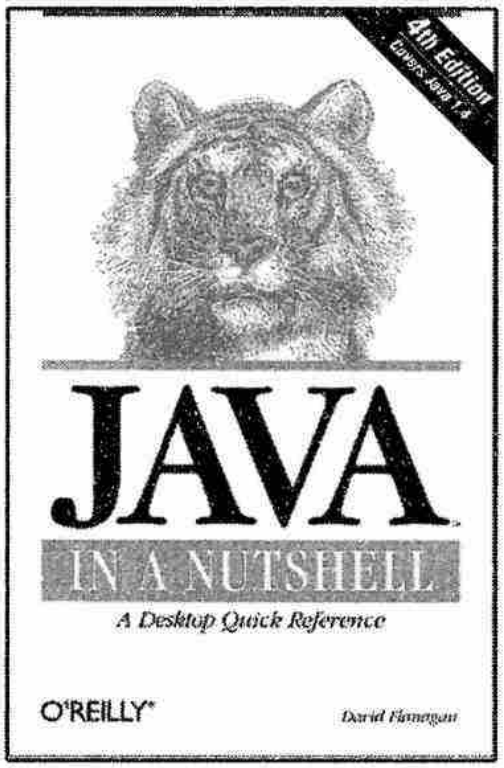
Джулия, 31 год, модель

# Как работать с API

**Две вещи, которые вам следует знать.**

1. **Какие классы содержатся в библиотеке?**
2. **Как вы узнаете о возможностях класса, отыскав его?**

## 1 Читаем книгу




## 2 Используем документацию по API в формате HTML

## 1 Читаем книгу

**Листать справочник — лучший способ изучения содержимого библиотеки Java. Просматривая страницы, вы легко можете обнаружить класс, который покажется вам полезным.**

*java.util.Currency*

*Returned By:* java.text.DecimalFormat.getCurrency(), java.text.DecimalFormatSymbols.getCurrency(), java.text.NumberFormat.getCurrency(), Currency.getInstance()

**Date** Java 1.0

java.util *cloneable serializable comparable*

This class represents dates and times and lets you work with them in a system-independent way. You can create a Date by specifying the number of milliseconds from the epoch (midnight GMT, January 1st, 1970) or the year, month, date, and, optionally, the hour, minute, and second. Years are specified as the number of years since 1900. If you call the Date constructor with no arguments, the Date is initialized to the current time and date. The instance methods of the class allow you to get and set the various date and time fields, to compare dates and times, and to convert dates to and from string representations. As of Java 1.1, many of the date methods have been deprecated in favor of the methods of the Calendar class.

```
public class Date implements Cloneable, Comparable, Serializable {
// Public Constructors
    public Date();
    public Date(long date);
#   public Date(String s);
#   public Date(int year, int month, int date);
#   public Date(int year, int month, int date, int hrs, int min);
#   public Date(int year, int month, int date, int hrs, int min, int sec);
// Property Accessor Methods (by property name)
    public long getTime();
    public void setTime(long time);
// Public Instance Methods
    public boolean after(java.util.Date when);
    public boolean before(java.util.Date when);
1.2 public int compareTo(java.util.Date anotherDate);
// Methods Implementing Comparable
1.2 public int compareTo(Object o);
// Public Methods Overriding Object
1.2 public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
// Deprecated Public Methods
#   public int getDate();
#   public int getDay();
#   public int getHours();
#   public int getMinutes();
#   public int getMonth();
#   public int getSeconds();
#   public int getTimezoneOffset();
#   public int getYear();
#   public static long parse(String s);
#   public void setDate(int date);
#   public void setHours(int hours);
#   public void setMinutes(int minutes);
#   public void setMonth(int month);
```

Имя класса.

Имя пакета.

Описание класса.

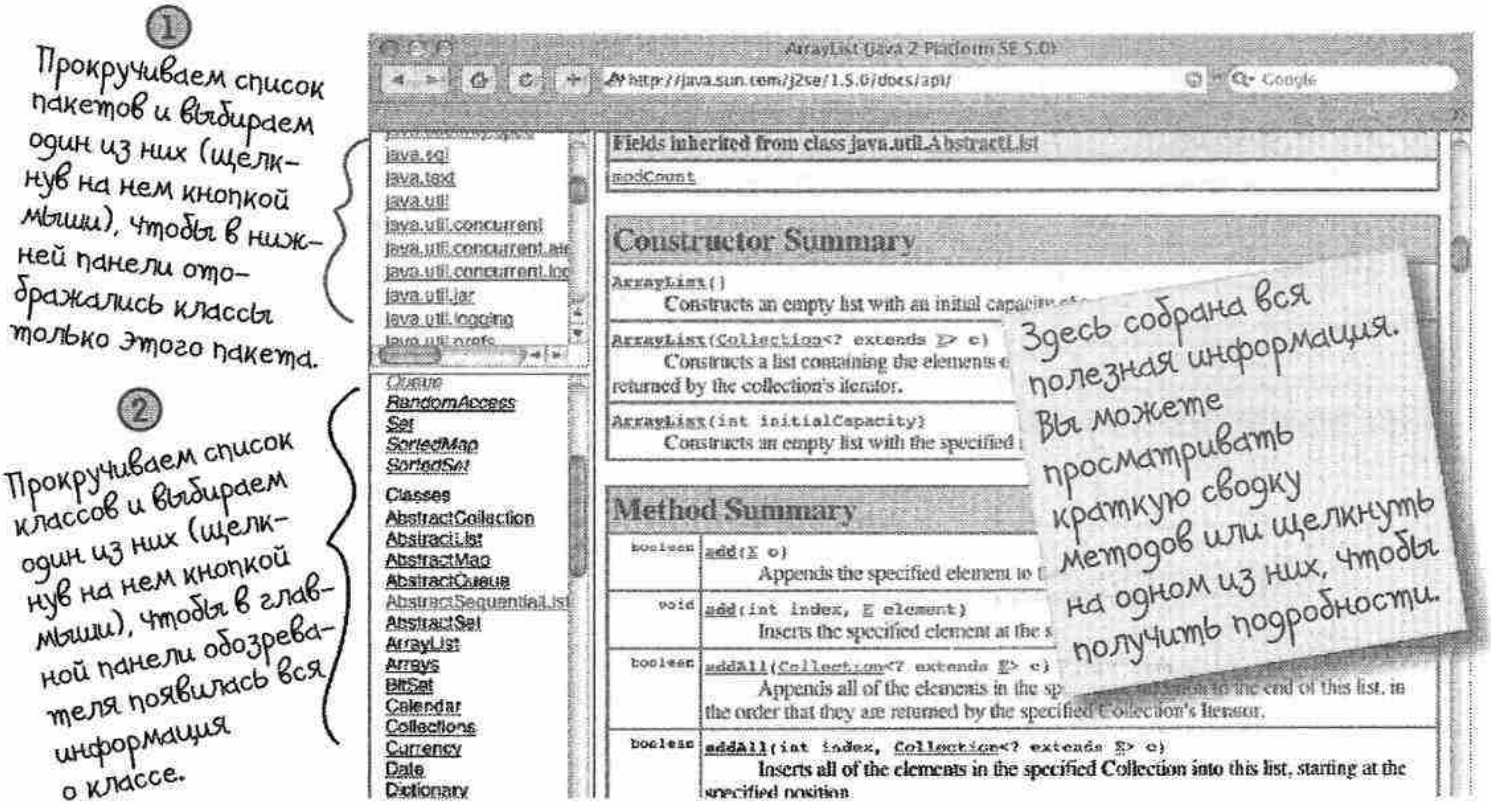
Методы (и другие элементы, о которых мы поговорим позже).

## 2 Используем документацию по API в формате HTML

В поставку Java входит потрясающий набор онлайн-документов под общим названием Java API. Они представляют собой часть более объемной коллекции документов, известной как Java 5 Standard Edition Documentation (которая может именоваться компанией Sun как Java 2 Standard Edition 5.0). Эту документацию нужно загружать отдельно, так как она не входит в архив с Java 5. Если у вас есть высокоскоростное интернет-подключение и недюжинное терпение, вы также можете просматривать ее на сайте java.sun.com. Но поверьте, скорее всего, вам захочется загрузить ее на свой жесткий диск.

Документация по API — лучший кладезь информации с подробностями о классах и их методах. Допустим, вы просматривали справочник и нашли в пакете java.util класс под названием Calendar. В книге о нем написано совсем немного — достаточно для того, чтобы захотеть его использовать, но вам нужно больше информации о методах.

Например, в справочнике указано, что именно принимает метод в качестве аргументов и что он возвращает. Вернемся к ArrayList. В справочнике вы можете найти метод indexOf(), который мы использовали в классе DotCom. Вам известно только, что есть метод indexOf(), который принимает объект и возвращает индекс (типа int) этого объекта. Однако остается одна важная вещь, которую вам необходимо знать: что случится, если объект не находится внутри ArrayList? Одна лишь сигнатура метода не скажет вам, как это работает. И вот здесь пригодится документация по API (в большинстве случаев). Из нее вы узнаете, что если объект находится вне ArrayList, то метод indexOf() возвращает –1. Вот откуда мы знали, что его можно использовать и для проверки нахождения объекта внутри ArrayList, и для получения его индекса, если он действительно там. Но без документации по API мы бы думали, что метод indexOf(), не найдя объект в списке, выбрасывает исключение.



# Магнитики с кодом

Можете ли вы восстановить из фрагментов кода работоспособную программу, которая выведет на экран текст, приведенный ниже? **Примечание:** чтобы выполнить это упражнение, вам необходимо знать еще кое-что — если вы отыщете ArrayList в документации, то увидите второй метод add, который принимает два аргумента:

**add(int index, Object o)**

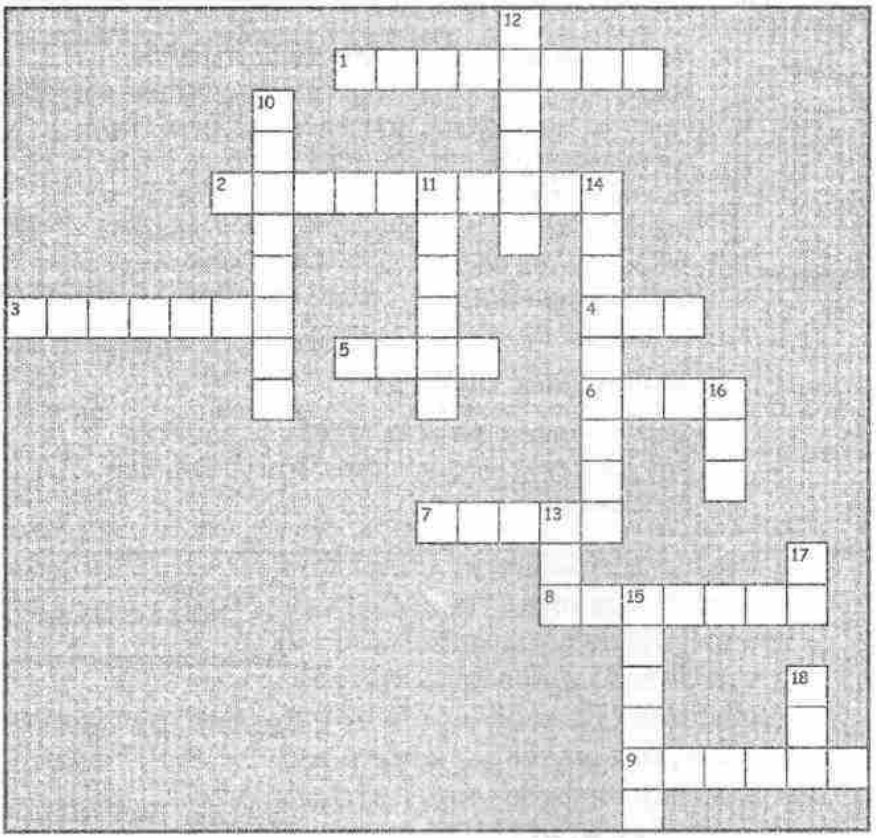
С его помощью вы можете сказать объекту ArrayList, *куда* именно поместить объект при добавлении.

```
a.remove(2);
printAL(a);
printAL(a);
a.add(0,"ноль");
a.add(1,"один");
public static void printAL(ArrayList<String> al) {
if (a.contains("два")) {
  a.add("2.2");
}
a.add(2,"два");
public static void main (String[] args) {
System.out.print(element + "  ");
}
System.out.println(" ");
if (a.contains("три")) {
  a.add("четыре");
}
public class ArrayListMagnet {
if (a.indexOf("четыре") != 4) {
  a.add(4, "4.2");
}
}
}
import java.util.*;
}
printAL(a);
ArrayList<String> a = new ArrayList<String>();
for (String element : al) {
a.add(3,"три");
printAL(a);
```

```
File Edit Window Help Dance
% java ArrayListMagnet
ноль  один  два  три
ноль  один  три  четыре
ноль  один  три  четыре  4.2
ноль  один  три  четыре  4.2
```

## Кроссворд

# JavaCross 7.0



Как кроссворд поможет вам в изучении Java? Все слова в нем связаны с этим языком.

***Подсказка:*** Когда начнете сомневаться, вспомните ArrayList.

### По горизонтали

1. Он где-то там.
2. Место обитания пакетов.
3. Индексируемый модуль.
4. Увеличивает ArrayList.
5. Ходит кругами.
6. Очень крупный.
7. Должен иметь низкую плотность.
8. Где же он, крошка?
9. У массива тоже есть свой размер.

### По вертикали

10. Я не могу себя как-то вести.
11. Как субъект, только наоборот.
12. Объем.
13. Краткое название библиотеки.
14. Массив на стероидах.
15. Тип по умолчанию для дробных чисел.
16. Копия значения.
17. Начало ветвления.
18. Не объект.

### Больше подсказок

По горизонтали
3. Подумайте об ArrayList.
7. Принципиа.
8. Подумайте об ArrayList.
9. Объем массива.

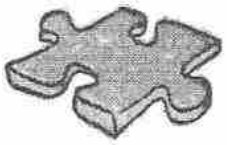
По вертикали
10. Всегда выдов.
18. Часто используемый простой тип.

## Магнитики с кодом

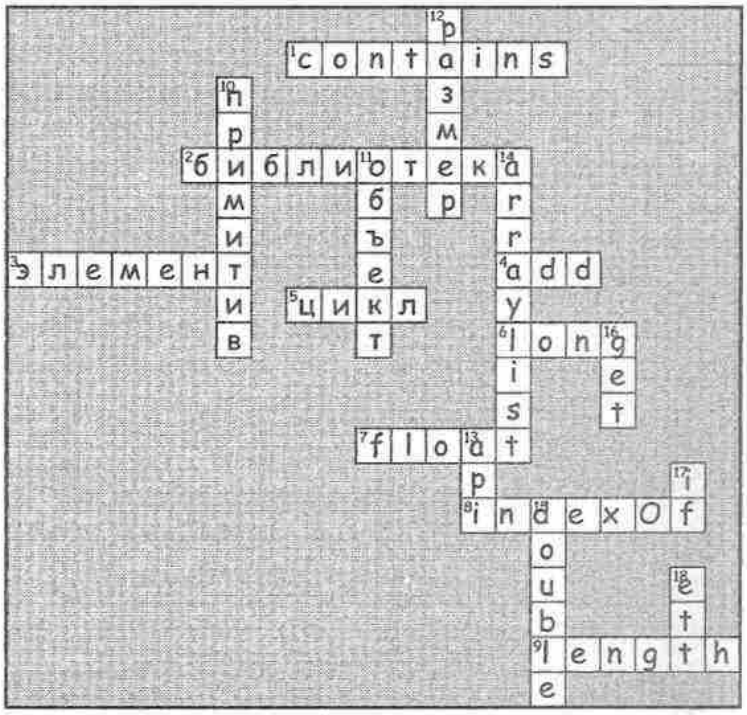
Ответы

```
import java.util.*;
public class ArrayListMagnet {
  public static void main (String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    a.add(0,"ноль");
    a.add(1,"один");
    a.add(2,"два");
    a.add(3,"три");
    printAL(a);
    if (a.contains("три")) {
      a.add("четыре");
    }
    a.remove(2);
    printAL(a);
    if (a.indexOf("четыре") != 4) {
      a.add(4, "4.2");
    }
    printAL(a);
    if (a.contains("два")) {
      a.add("2.2");
    }
    printAL(a);
  }
  public static void printAL(ArrayList<String> al) {
    for (String element : al) {
      System.out.print(element + "  ");
    }
    System.out.println(" ");
  }
}
```

```
File Edit Window Help Dance
% java ArrayListMagnet
ноль  один  два  три
ноль  один  три  четыре
ноль  один  три  четыре  4.2
ноль  один  три  четыре  4.2
```



# Ответы на кроссворд



## Наточите свой карандаш

Напишите собственный набор подсказок! Глядя на каждое слово, попытайтесь подобрать к нему ключ. Старайтесь делать подсказки легкими, добавляйте в них больше технических подробностей, чем это сделали мы.

### По горизонтали

1. _____
2. _____
3. _____
4. _____
5. _____
6. _____
7. _____
8. _____
9. _____

### По вертикали

10. _____
11. _____
12. _____
13. _____
14. _____
15. _____
16. _____
17. _____
18. _____

# Прекрасная жизнь в Объективилле

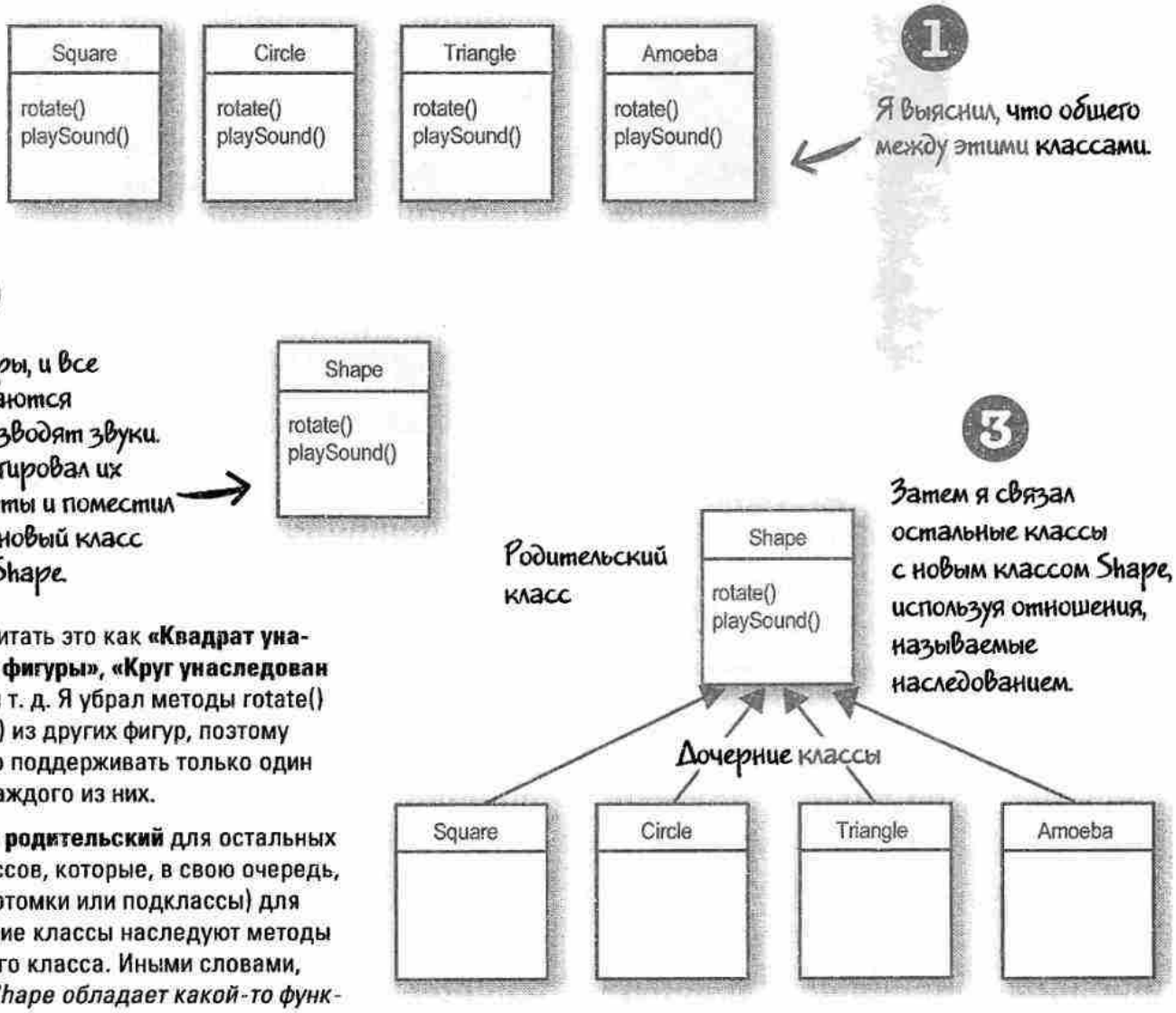
**Планируйте свои программы с прицелом на будущее.** Как вы оцените способ создания Java-приложений, при котором у вас останется много свободного времени? Представьте, что вы можете написать такой код, который **легко** сумеет дополнить *другой* программист. Заинтересует ли вас создание гибкого кода, которому не страшны досадные изменения в техническом задании, возникающие в последний момент? Поверьте, вы можете получить все это лишь за три простых подхода по 60 минут каждый. Изучив принципы полиморфизма, вы узнаете, о пяти шагах грамотного проектирования классов, трех приемах для достижения полиморфизма и восьми способах создания гибкого кода. Если вы начнете прямо сейчас, то получите призовой урок из четырех советов по использованию наследования. Не медлите! Приняв наше предложение, вы обретете то, что заслуживаете, — свободу и гибкость при проектировании программ. Это быстро, просто и доступно прямо сейчас. Начните сегодня — и вам покорятся невиданные уровни абстракции!

# И снова война за кресло...

***Надеемся, вы не забыли, как в главе 2 Ларри (любитель процедурного стиля) и Брэд (приверженец ООП) сошлись лицом к лицу в борьбе за кресло Aeron? Вспомним несколько отрывков из той истории, чтобы повторить основные принципы наследования.***

**Ларри:** Ты дублируешь код! Процедура rotate находится во всех четырех объектах. Это глупый подход. Ты должен поддерживать четыре разных «метода» rotate. Что в этом может быть хорошего?

**Брэд:** Похоже, что ты не видел конечный результат. Сейчас я тебе покажу, как работает **наследование** в ООП.




Вы можете читать это как **«Квадрат унаследован от фигуры»**, **«Круг унаследован от фигуры»** и т. д. Я убрал методы rotate() и playSound() из других фигур, поэтому теперь нужно поддерживать только один экземпляр каждого из них.

Класс Shape **родительский** для остальных четырех классов, которые, в свою очередь, **дочерние** (потомки или подклассы) для него. Дочерние классы наследуют методы родительского класса. Иными словами, *если класс Shape обладает какой-то функциональностью, то его подклассы получают те же возможности.*

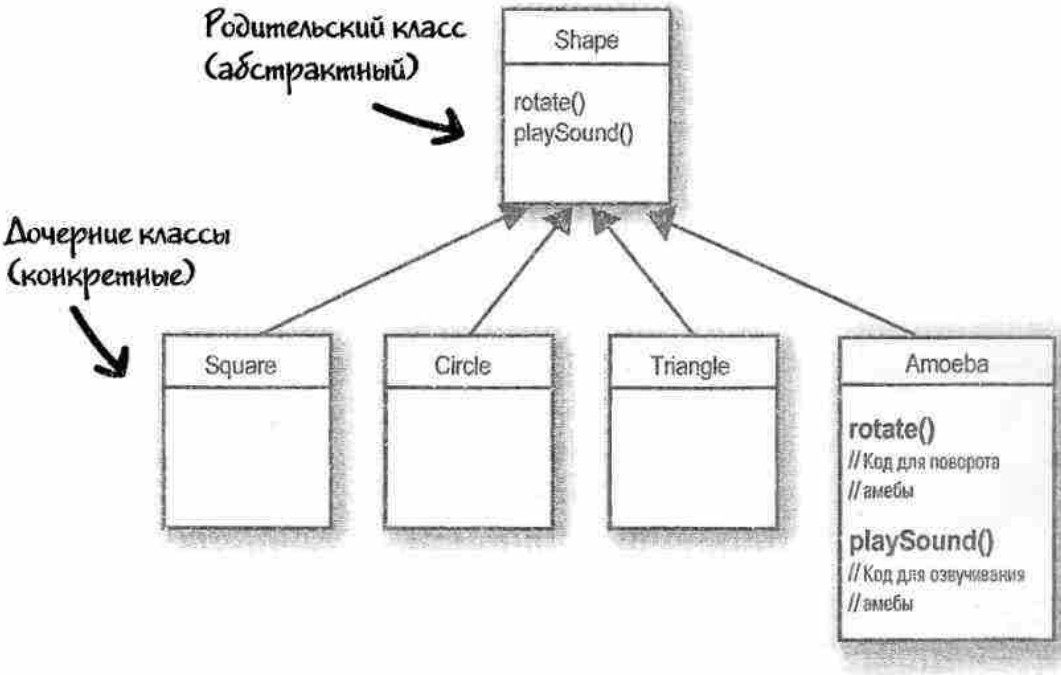
## Как быть с методом rotate фигуры Amoeba?

**Ларри:** Не кроется ли вся проблема в том, что амеба имеет совершенно другие процедуры rotate и playSound?

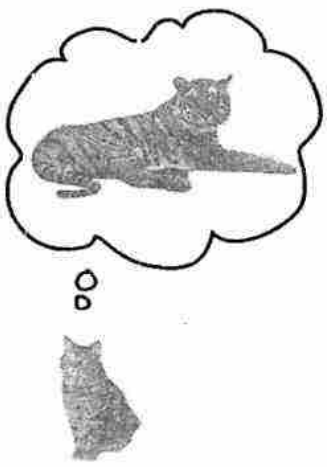
Как же амеба сможет делать что-то по-своему, если она *наследует* свою функциональность от класса Shape?

**Брэд:** Это завершающий штрих. Класс Amoeba ***переопределяет*** методы класса Shape. Потом, при выполнении программы, JVM будет точно знать, какой именно метод *rotate* нужно вызвать, когда понадобится вращать амебу.

Я сделал так, чтобы класс Amoeba переопределил методы rotateO и playSoundO, принадлежащие родительскому классу Shape. Переопределение означает, что подкласс заново описывает унаследованные им методы, если ему нужно изменить или расширить их поведение.




---



## Сила мозга

Как вы опишете в структуре наследования домашнего кота и тигра? Может ли обычный кот рассматриваться как отдельный вариант тигра? Кто из них станет родительским классом, а кто — дочерним? Или они оба будут наследниками какого-то *другого* класса?

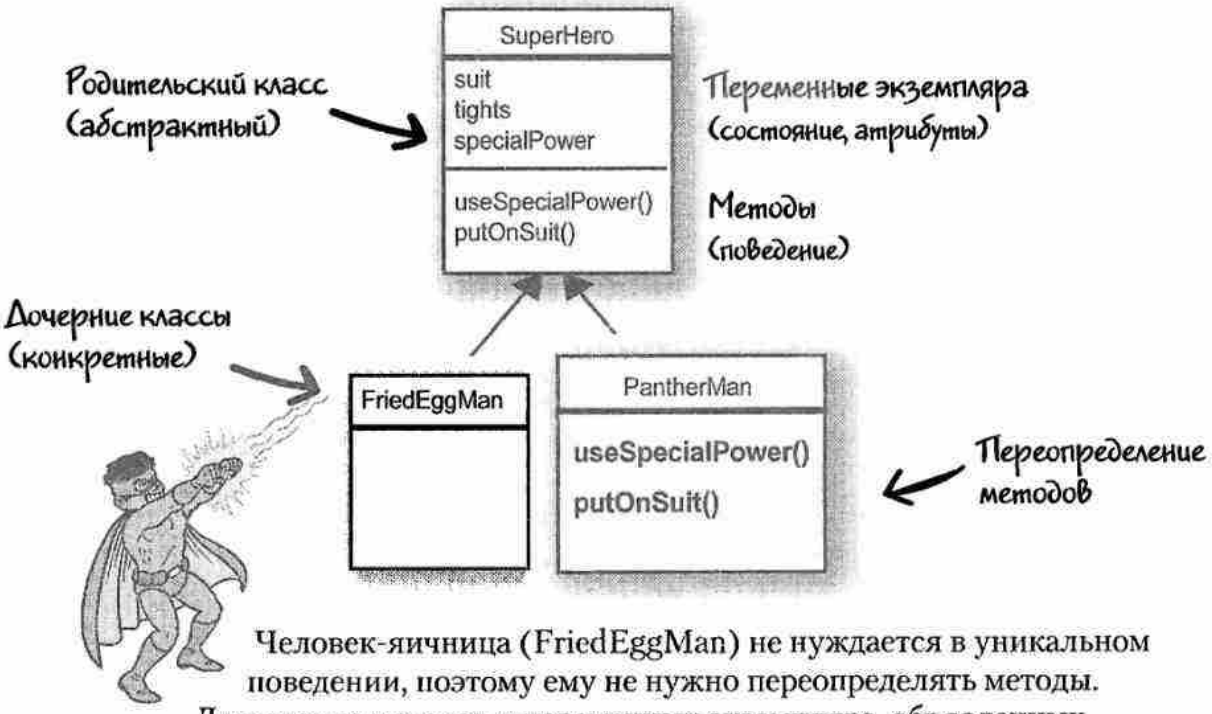
Как вы спроектируете такую структуру наследования? Какие методы при этом переопределите? Подумайте об этом, *прежде* чем перевернете страницу.

# Принципы наследования

При проектировании с использованием наследования вы помещаете общий код в один абстрактный класс и объявляете его родителем для других, более специфических классов. Когда один класс наследует другой, это означает, **что дочерний класс наследует родительский**.

В Java мы говорим, что **дочерний класс *расширяет* родительский**. Подобные отношения подразумевают, что дочерний класс наследует **члены** родительского класса. Когда мы говорим «член класса», мы имеем в виду переменные экземпляра и методы.

Например, если Человек-пантера (PantherMan) — потомок Супергероя (SuperHero), то класс PantherMan автоматически наследует переменные экземпляра и методы, общие для всех супергероев, включая костюм (`suit`), трико (`tights`), суперсилу (`specialPower`), умение использовать суперсилу (`useSpecialPower()`) и т. д. Но дочерний класс PantherMan **может иметь собственные переменные экземпляра и методы** и способен **переопределить методы,** унаследованные от SuperHero.




Человек-яичница (FriedEggMan) не нуждается в уникальном поведении, поэтому ему не нужно переопределять методы. Достаточно методов и переменных экземпляра, объявленных в классе SuperHero.

А вот у PantherMan есть особые требования к костюму и суперсиле, поэтому в его классе переопределяются методы `useSpecialPower()` и `putOnSuit()`.

**Переменные экземпляра не переопределяются**, так как в этом нет необходимости. Они не описывают специфическое поведение, поэтому дочерний класс может присвоить им любые значения. PantherMan может задать переменной `tights` пурпурный цвет, тогда как FriedEggMan сделает его белым.

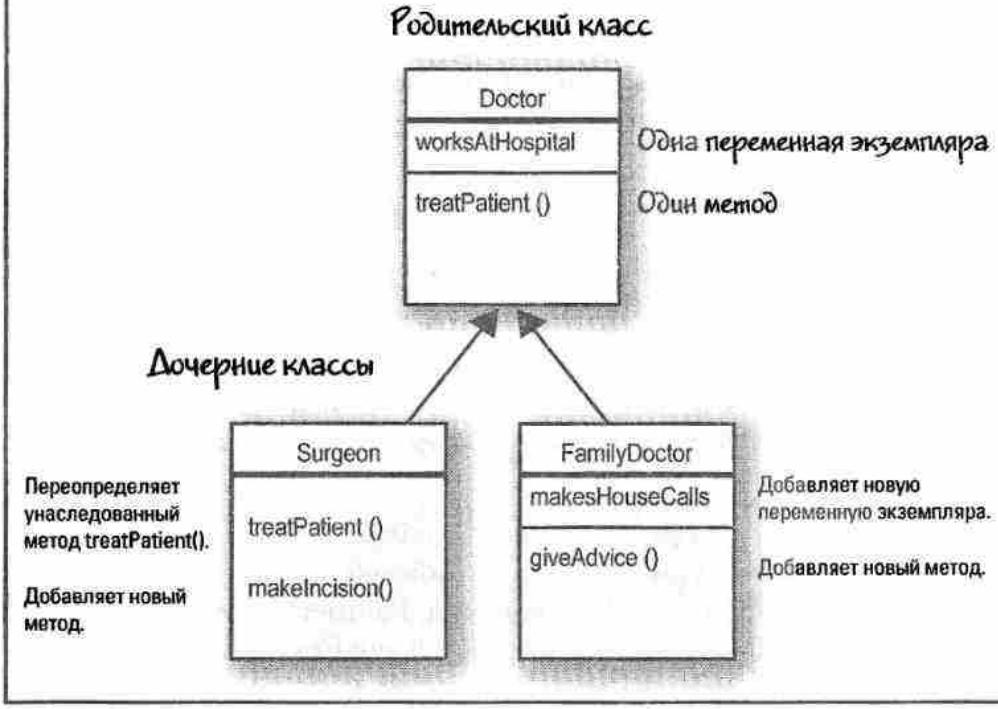
## Пример наследования:

```
public class Doctor {

    boolean worksAtHospital;

    void treatPatient() {
      // Проводим проверку

    }
}
```

```
public class FamilyDoctor extends Doctor {

    boolean makesHouseCalls;
    void giveAdvice() {
      // Даем простой совет
    }

}
```

```
public class Surgeon extends Doctor{

   void treatPatient() {
      // Проводим операцию
   }

   void makeIncision() {
      // Делаем надрез (ой!)
   }
}
```

### Наточите свой карандаш




Сколько переменных экземпляра у класса Surgeon? _____

Сколько переменных экземпляра у класса FamilyDoctor? _____

Сколько методов у класса Doctor? _____

Сколько методов у класса Surgeon? _____

Сколько методов у класса FamilyDoctor? _____

Может ли FamilyDoctor выполнить метод treatPatient()? _____

Может ли FamilyDoctor выполнить метод makeIncision()? _____

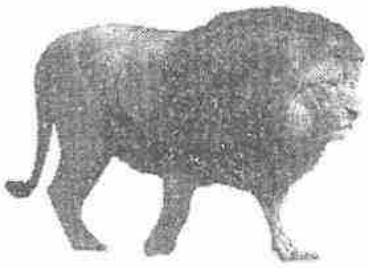
# Построим иерархию наследования на примере программы, моделирующей жизнь животных

Допустим, нужно создать программу для моделирования поведения животных. Она должна позволять пользователю выбирать несколько разных животных, помещать их в окружающую среду и наблюдать за тем, что произойдет. Сейчас не нужно писать код, нас больше интересует проектирование.

У нас есть список животных, которые должны быть в программе, но он пока неполный. Мы знаем, что каждое животное будет представлено объектом, объекты будут перемещаться по среде обитания и вести себя так, как они запрограммированы.

***Еще мы хотим, чтобы другие программисты могли в любой момент добавить в программу новые виды животных.***

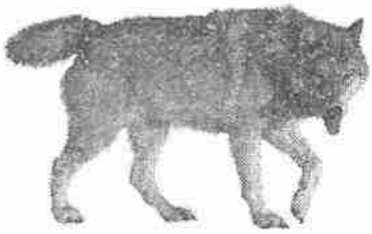
Прежде всего необходимо выделить общие, абстрактные характеристики, присущие всем животным, и воплотить их в виде класса, который будут наследовать все животные.



**1** Нужно найти объекты, которые имеют общие черты и поведение.

**Что общего у этих шести типов? Эта информация поможет выделить поведение (шаг 2).**

**Как эти типы связаны между собой? Эти сведения помогут определить иерархию наследования (шаги 4–5).**



# Использование наследования для предотвращения дублирования кода в дочерних классах

Разработаем класс, который воплощает общие черты и поведение.

**Все эти объекты представляют животных, поэтому мы создадим общий родительский класс с названием Animal.**

**Мы добавим в него методы и переменные экземпляра, которые могут понадобиться всем животным.**

У нас есть пять ***переменных экземпляра***.

***picture*** — имя JPEG-файла, который ассоциируется с данным животным.

***food*** — тип еды, которую это животное употребляет. Пока допускаются два значения: *мясо* и *трава*.

***hunger*** — целое число, описывающее уровень аппетита животного. Оно изменяется в зависимости от того, когда и сколько животное ело.

***boundaries*** — значение, описывающее длину и ширину «участка» (например, 640×480), по которому животное будет бродить.

***location*** — координаты X и Y, определяющие местоположение животного.

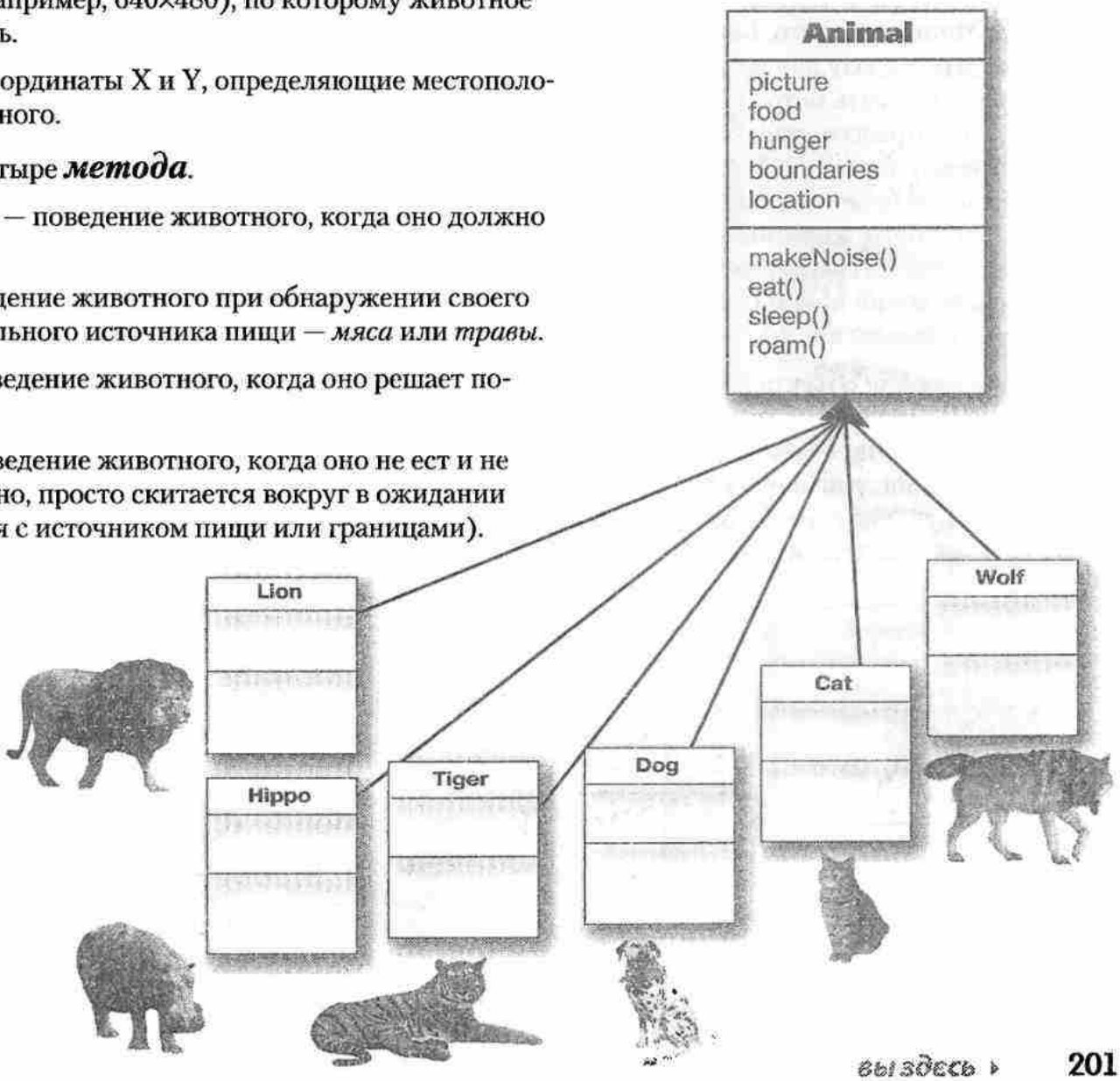
У нас есть четыре ***метода***.

***makeNoise()*** — поведение животного, когда оно должно издать звук.

***eat()*** — поведение животного при обнаружении своего предпочтительного источника пищи — *мяса* или *травы*.

***sleep()*** — поведение животного, когда оно решает поспать.

***roam()*** — поведение животного, когда оно не ест и не спит (вероятно, просто скитается вокруг в ожидании столкновения с источником пищи или границами).



# Все ли животные питаются одинаково?

Допустим, мы решили, что переменные экземпляра будут уместны для *всех* типов животных. Лев будет иметь собственные значения для переменных picture, food (вероятно, это *мясо*), hunger, boundaries и location. У гиппопотама будут свои *значения*, но набор переменных у них будет такой же, как и у других типов животных. То же самое с собакой, тигром и т. д. Но что насчет *поведения*?

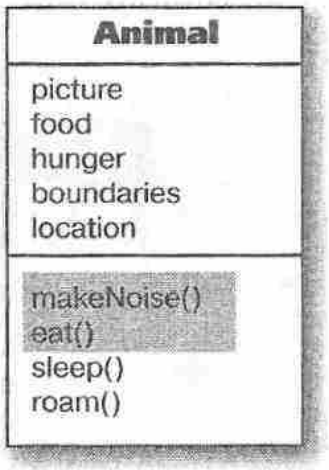
## Какие методы нужно переопределить

Издает ли лев такие же **звуки**, как собака? Одинаково ли питаются кот и гиппопотам? В *вашей* версии — может быть, но у нас питание и звуки зависят от типа животного. Как заставить эти методы работать сразу для всех животных? К примеру, можно написать метод makeNoise(), который будет проигрывать звуковой файл, заданный в виде значения переменной конкретного типа, но такое поведение будет не совсем специализированным. Некоторые животные в различных ситуациях могут издавать разные звуки (например, во время приема пищи или при встрече с врагом и т. д.).

Вспомните, как для объекта Amoeba мы переопределяли метод rotate класса Shape, чтобы получить более специфичное (другими словами, уникальное) для нее поведение. Теперь нужно сделать то же самое и для наших потомков класса Animal.

**3** Следует установить, нуждается ли дочерний класс в поведении (реализации методов), характерном именно для этого типа.

**Взглянув на класс Animal, мы решили, что методы eat() и makeNoise() нужно переопределить на уровне отдельных дочерних классов.**




Мы переопределим методы eat() и makeNoise(), чтобы животное каждого типа могло определять свое уникальное поведение при приеме пищи и издавании звуков. Методы sleep() и roam() пока могут остаться общими.

# Ищем новые возможности, которые дает наследование

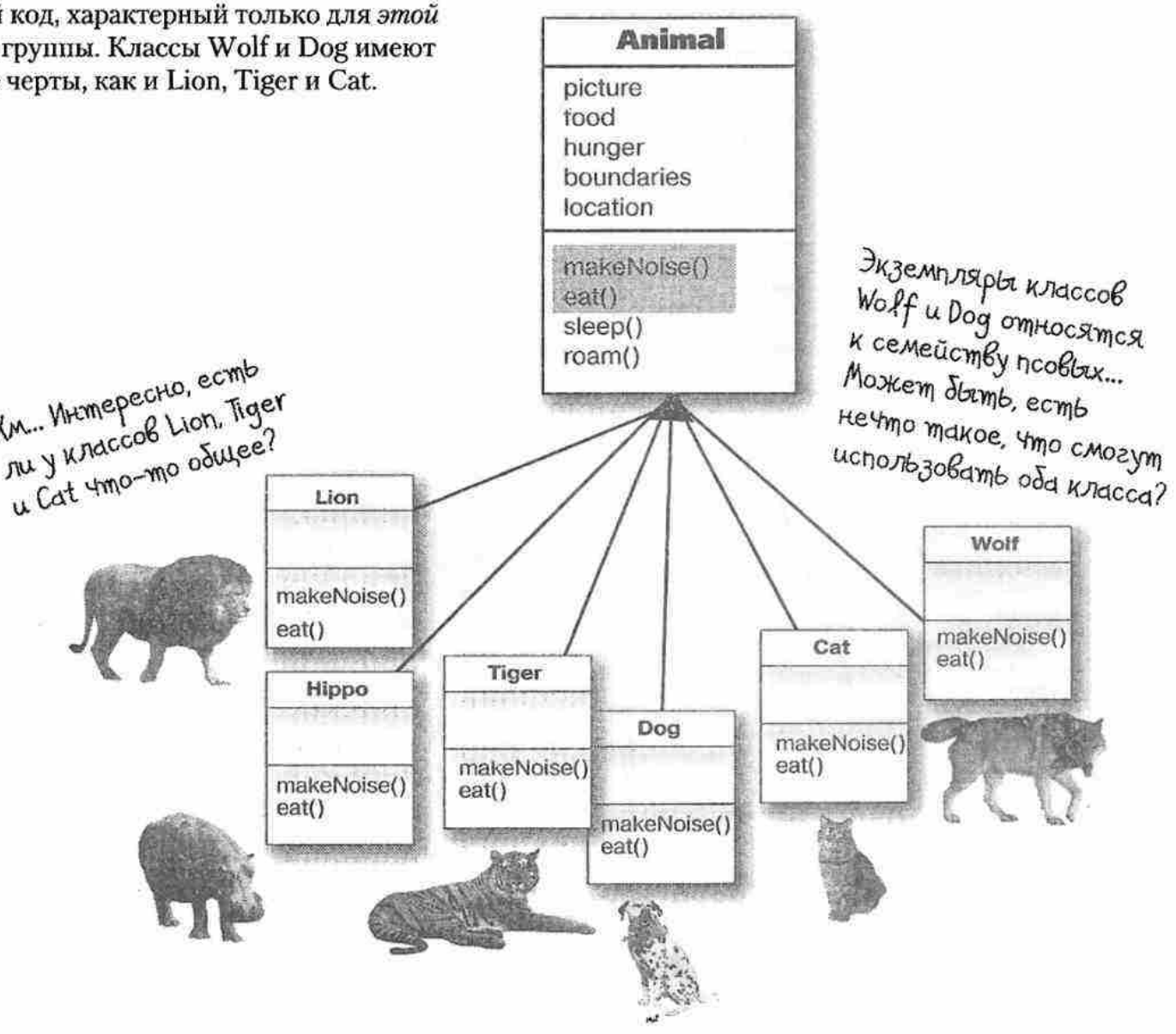
Иерархия классов начинает прорисовываться более четко. Каждый дочерний класс должен переопределить методы *makeNoise()* и *eat()*, чтобы лай объекта Dog нельзя было спутать с мяуканьем объекта Cat (это было бы оскорбительно для обоих). Кроме того, объект Hippo питается совсем не так, как объект Lion.

Но, возможно, мы можем сделать кое-что еще. Взглянем на потомков класса Animal и подумаем, можно ли каким-то образом сгруппировать несколько из них, выделив общий код, характерный только для *этой* новой группы. Классы Wolf и Dog имеют общие черты, как и Lion, Tiger и Cat.

4

Нужно попробовать найти больше возможностей для использования абстракции, выделив несколько дочерних классов, которые могут иметь общее поведение.

**Взглянув на классы, мы понимаем, что у Wolf и Dog есть общие черты. То же самое относится к Lion, Tiger и Cat.**

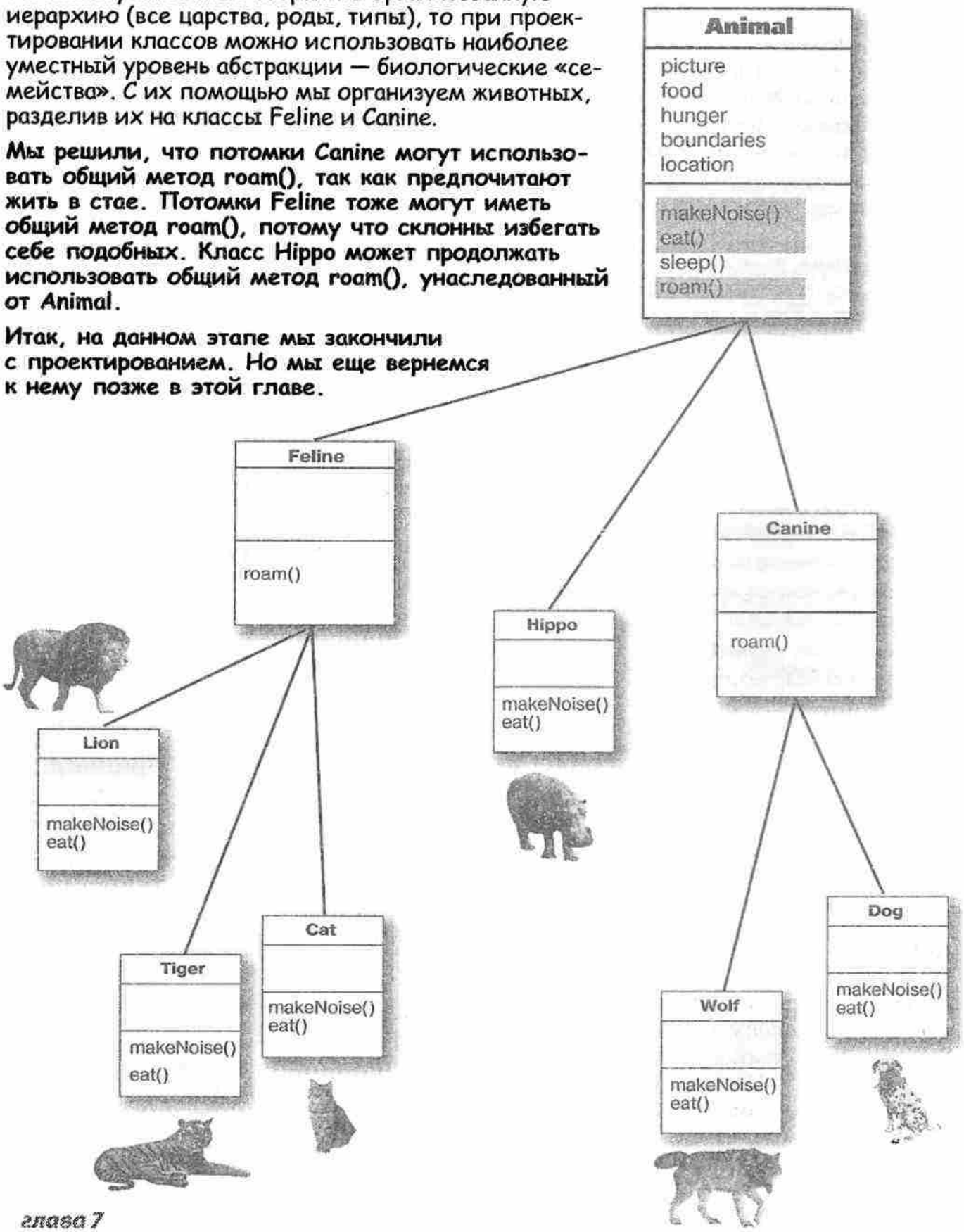


## 5 Завершение работы над иерархией классов.

Поскольку животные собраны в организованную иерархию (все царства, роды, типы), то при проектировании классов можно использовать наиболее уместный уровень абстракции — биологические «семейства». С их помощью мы организуем животных, разделив их на классы Feline и Canine.

**Мы решили, что потомки Canine могут использовать общий метод roam(), так как предпочитают жить в стае. Потомки Feline тоже могут иметь общий метод roam(), потому что склонны избегать себе подобных. Класс Hippo может продолжать использовать общий метод roam(), унаследованный от Animal.**

**Итак, на данном этапе мы закончили с проектированием. Но мы еще вернемся к нему позже в этой главе.**



## Какой метод вызывается?

У класса Wolf есть четыре метода. Один унаследован от Animal, другой — от Canine (это, по сути, переопределенная версия метода из класса Animal) и еще два переопределенных. Создав объект Wolf и присвоив его переменной, вы можете использовать оператор «точка» в сочетании со ссылкой, чтобы вызывать все четыре метода. Но какие их *версии* будут при этом вызваны?

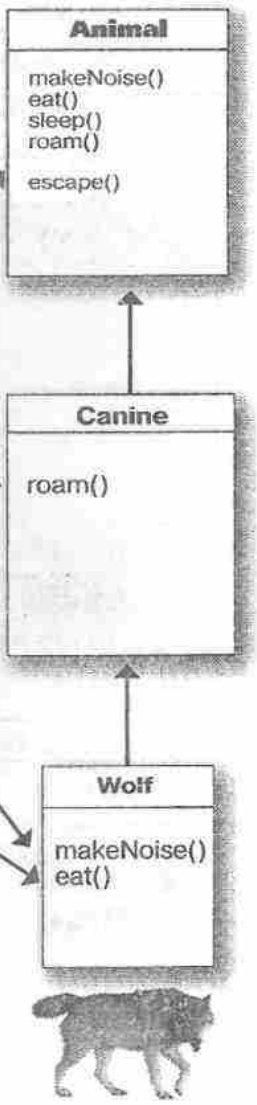
Создаем новый объект Wolf. `Wolf w = new Wolf();`

Вызывается версия из Wolf. `w.makeNoise();`

Вызывается версия из Canine. `w.roam();`

Вызывается версия из Wolf. `w.eat();`

Вызывается версия из Animal. `w.sleep();`




Вызывая метод из ссылки на объект, вы получаете его версию, наиболее характерную для этого типа объектов.

Иными словами, ***берется метод самого низкого уровня***.

Это «самый низкий» метод в контексте иерархии наследования. Класс Canine находится ниже, чем Animal, а Wolf — ниже, чем Canine, поэтому вызов метода из ссылки, связанной с объектом Wolf, заставляет JVM искать его в первую очередь в классе Wolf. Не найдя метод там, JVM начинает подниматься по иерархии наследования, пока не найдет совпадения.

# Проектирование иерархии наследования

| Класс | Родительские классы | Дочерние классы |
|---|---|---|
| Одежда | – | Трусы, Рубашка |
| Трусы | Одежда | – |
| Рубашка | Одежда | – |

Таблица наследования

Диаграмма наследования классов

## Наточите свой карандаш

Найдите отношения, которые имеют смысл. Заполните последние два столбца.

| Класс | Родительские классы | Дочерние классы |
|---|---|---|
| Музыкант | | |
| Рок-звезда | | |
| Фанат | | |
| Бас-гитарист | | |
| Концертный пианист | | |
| | | |

*Подсказка: не каждый элемент может быть связан с другим элементом.*
*Подсказка: можно добавлять или изменять перечисленные здесь классы.*

Нарисуйте здесь диаграмму наследования.

## Это не глупые вопросы

**В:** **Вы говорите, что JVM поднимается по иерархии наследования, начиная с класса, из которого был вызван метод (как в примере с классом Wolf на предыдущей странице). Но что случится, если JVM так и не найдет совпадения?**

**О:** Хороший вопрос! Не волнуйтесь, компилятор гарантирует, что конкретный метод можно вызвать из ссылки данного типа, но ему все равно, из какого класса на самом деле это произойдет.
В случае с классом Wolf компилятор ищет метод sleep(), но его не волнует, что тот был объявлен в классе Animal (и унаследован от него). Помните, если класс *наследует* метод, значит он содержит его.
Компилятору не важно, *где именно* был объявлен унаследованный метод (то есть в каком родительском классе он определен). Но во время выполнения программы **JVM всегда выбирает правильную версию**. Здесь имеется в виду ***наиболее специфичная версия конкретного объекта***.

# Использование отношений IS-A и HAS-A

Когда один класс наследует другой, мы говорим, что дочерний класс расширяет родительский. Если вы хотите узнать, *расширяет* ли одна сущность другую, проведите проверку на соответствие — IS-A (является).

Треугольник — это фигура. Все правильно.

Кошка — представитель семейства кошачьих. Это тоже нормально.

Ванна расширяет ванную комнату. Звучит логично.

*Это все будет верным, пока вы не проведете проверку на соответствие.*

Чтобы узнать, правильно ли вы спроектировали свои типы, спросите себя: «Можно ли сказать, что тип X представляет собой тип Y?» Если ответ отрицательный, значит, вы что-то не так сделали. Иными словами, проверив на соответствие ванну (класс Tub) и ванную комнату (класс Bathroom), вы определенно получите false.

А если поменять их местами и сказать: «Ванная комната расширяет ванну»? Все еще неверно, так как ванная комната — это не ванна.

Эти понятия связаны между собой, но не через наследование. Такая связь выражается отношением HAS-A (содержит). Разве не логично сказать, что ванная комната содержит ванну? Если ответ положительный, значит, класс Bathroom имеет переменную типа Tub. Иными словами, класс Bathroom содержит *ссылку* на класс Tub, но не *расширяет* его (и наоборот).

Ванная комната содержит ванну, а ванна — пену (класс Bubbles).

Но ни один из этих классов не наследует (расширяет) другой.

## Но погодите! Это еще не все!

Проверка на соответствие IS-A работает в *любой части* иерархии наследования. Если ваша иерархия спроектирована должным образом, то проверка на соответствие будет иметь смысл в контексте *любого* родительского класса и *любых* его подклассов.

**Если класс B расширяет класс A, то класс B соответствует классу A. Это утверждение истинно на любом участке иерархии наследования. Если класс C расширяет класс B, то он проходит проверку на соответствие обоим классам — B и A.**

Canine расширяет Animal.

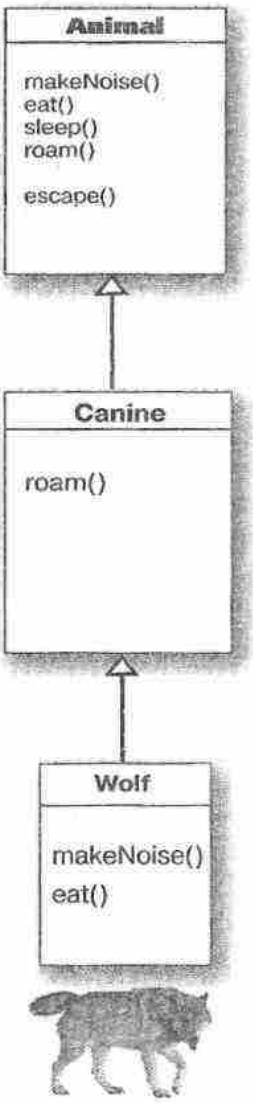
Wolf расширяет Canine.

Wolf расширяет Animal.

Canine соответствует Animal.

Wolf соответствует Canine.

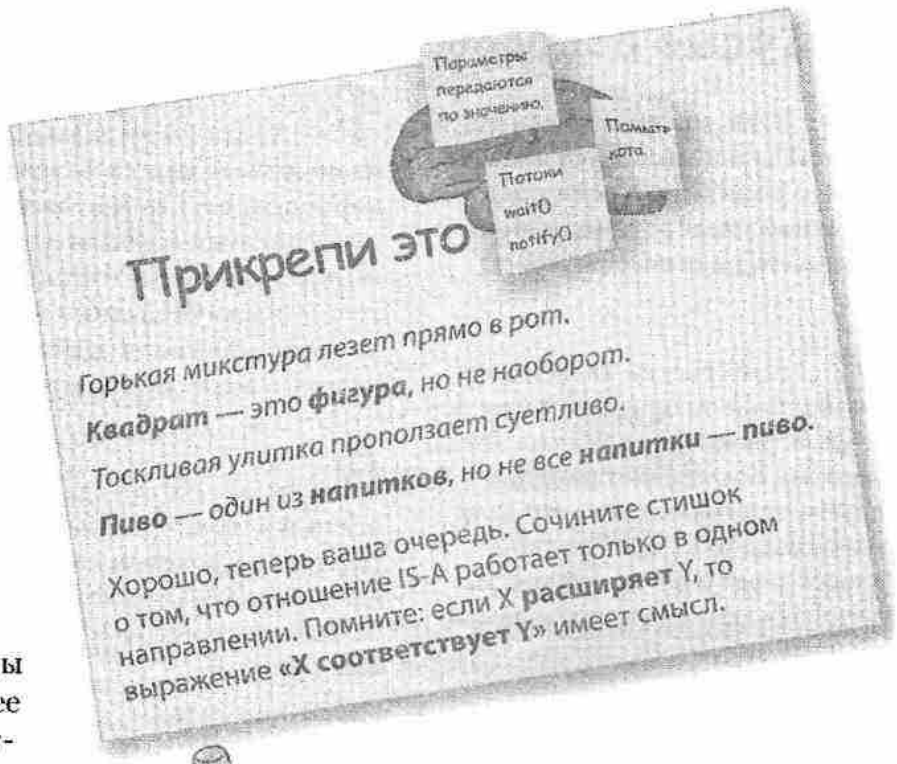
Wolf соответствует Animal.




При такой иерархии наследования, как показана здесь, вы всегда можете сказать: **«Wolf расширяет Animal»** или **«Wolf соответствует Animal»**. Не важно, будет ли Animal для Wolf дочерним или родительским классом. В сущности, **пока Animal находится *где-либо* в иерархии наследования относительно Wolf, тот всегда будет соответствовать Animal.**

Структура иерархии наследования класса Animal гласит: «Wolf соответствует Canine, поэтому может делать все, что может делать Canine. Класс Wolf также соответствует Animal, поэтому он может делать все, на что способен класс Animal».

Не важно, переопределяет ли Wolf методы из классов Animal или Canine. Пока окружающий мир (другой код) в этом уверен, Wolf способен выполнять все четыре метода. *Как* он будет это делать или *в каком классе они переопределены*, не играет никакой роли. У Wolf есть методы makeNoise(), eat(), sleep() и roam(), так как он унаследовал их у класса Animal.




# Как узнать, что наследование оформлено правильно

Конечно, эта тема гораздо обширнее и мы рассмотрели ее далеко не всю. Подробнее проблемы ООП будут раскрыты в следующей главе (где мы в итоге улучшим кое-что из того, что спроектировали в *этой* главе).

Старайтесь использовать проверку на соответствие. Если X соответствует Y, значит, оба класса (X и Y), скорее всего, должны находиться в одной иерархии наследования. Вероятно, они ведут себя одинаково или дублируют функции друг друга.

## Не забывайте, что отношение IS-A работает только в одном направлении!

Треугольник — это фигура. Выражение имеет смысл, поэтому треугольник расширяет фигуру.

Но если поменять их местами: фигура — это треугольник, то будет нелогично, поэтому класс Shape *не* должен расширять класс Triangle. Помните, отношение IS-A подразумевает, что если X соответствует Y, то X может делать все, что может делать Y (а иногда и больше).

### Наточите свой карандаш

Отметьте отношения, которые имеют смысл.

- ☐ **Печь расширяет кухню.**
- ☐ **Гитара расширяет инструмент.**
- ☐ **Человек расширяет служащего.**
- ☐ **Феррари расширяет двигатель.**
- ☐ **Яичница расширяет еду.**
- ☐ **Гончая расширяет домашнее животное.**
- ☐ **Контейнер расширяет емкость.**
- ☐ **Металл расширяет титан.**
- ☐ **GratefulDead расширяет музыкальную группу.**
- ☐ **Блондинка расширяет ум.**
- ☐ **Напиток расширяет мартини.**

*Подсказка: выполните проверку на соответствие.*

## Это не глупые вопросы

**В: Итак, мы видим, как дочерний класс наследует методы своего родителя. А если родительский класс захочет использовать методы своего потомка?**

О: Родительский класс может не *знать* ни об одном из своих потомков. Вы можете написать класс, который позже будет расширен другим программистом. Но даже если создатель родительского класса знает о дочерней версии метода (и хочет ее использовать), он не сможет этого сделать, так как не существует *реверсивного*, или *обратного*, наследования. Подумайте, ведь только дети наследуют что-либо от родителей, по-другому не бывает.

**В: А если я хочу вызывать из дочернего класса не только переопределенный им метод, но и оригинальную версию этого метода, указанную в классе-родителе? Иными словами, я хочу не *заменить* версию родительского класса, а лишь дополнить ее.**

О: Вы можете это сделать! И это важный элемент дизайна языка. Думайте о ключевом слове extends как о выражении «я хочу *расширить* функциональность родительского класса».

Вы можете разрабатывать методы своего родительского класса так, чтобы они содержали реализацию для любых дочерних классов, даже если методам придется «добавить» к ним собственный код. Переопределяя метод, вы можете вызвать его родительскую версию с помощью ключевого слова `super`. Это то же самое, что сказать: «Сначала запусти версию родительского класса, потом вернись и закончи выполнение моего кода».

```
public void roam() {
  super.roam();
  // Мое дополнение к методу roam
}
```

Здесь вызывается унаследованная версия метода roam(), после чего выполняется код, относящийся только к дочернему классу.

## Кто получает «Порш», а кому достается фарфоровая ваза, или Как узнать, что дочерний класс может унаследовать от своего родителя

Дочерний класс наследует члены класса-родителя. На текущий момент это относится к переменным и методам, но позже мы рассмотрим и другие наследуемые члены класса. Родительский класс может выбирать, хочет ли он, чтобы кто-то наследовал его конкретный метод. Это делается с использованием уровня доступа, который выдается методу.

В книге будут рассмотрены четыре уровня доступа. Перечислим их, начиная с самого ограниченного:

private default protected public

**Применение уровней доступа** — необходимое условие для создания хорошо спроектированного, устойчивого к ошибкам Java-кода. Пока мы сосредоточимся на двух уровнях — public и private, так как они предусматривают самые простые правила:

**публичные** (public) члены *наследуются;*

**приватные** (private) члены *не наследуются.*

Наследуя член родительского класса, потомок ***как бы сам объявляет его***. Рассмотрим пример с фигурами. Класс Square наследует методы `rotate()` и `playSound()`, и внешне (для остального кода) это выглядит так, как будто `Square` *содержит* эти методы.

Члены класса включают в себя переменные экземпляра и методы, объявленные в классе, а также все, что было унаследовано от класса-родителя.

Примечание: больше информации об уровнях default и protected можно получить в Приложении Б.

# Наследование при проектировании — хорошо или плохо?

Вы сможете лучше проектировать свои приложения, зная некоторые правила. Пока просто примите их на веру, а более подробно вы познакомитесь с ними позже в этой книге.

**Используйте** наследование для класса, который представляет собой более специфичный вид родительского класса. Пример: Willow (ива) — это тип Tree (дерево), поэтому Willow на самом деле *расширяет* Tree.

**Применяйте** наследование, если у вас есть поведение (реализованный код), которое должно быть общим для разных классов одного типа. Пример: Square (квадрат), Circle (окружность) и Triangle (треугольник) должны уметь поворачиваться и воспроизводить звук, поэтому есть смысл добавить такое поведение в родительский класс Shape (фигура). Это также улучшит сопровождаемость и расширяемость кода. Наследование — одна из ключевых возможностей объектно ориентированного программирования, но оно не всегда должно рассматриваться как наилучший способ повторного использования кода. Оно поможет вам на первых порах и часто будет правильным выбором при разработке, но шаблоны проектирования предоставят более изысканные и гибкие варианты.

**Не применяйте** наследование только для того, чтобы иметь возможность повторно использовать код из других классов, если отношения между родительским и дочерним классами нарушают хотя бы одно из двух правил, приведенных выше. Представьте, что вы написали специальный код для печати в классе Alarm, и теперь этот код нужен вам в классе Piano. Вы наследуете Piano от Alarm, чтобы получить доступ к коду для печати. В этом нет смысла! Piano *не* является специфичным типом класса Alarm. Код для печати должен находиться в классе Printer, чтобы все объекты, которые хотят печатать, могли получить его возможности через отношение HAS-A.

**Не используйте** наследование, если дочерний и родительский классы не проходят проверку на соответствие. Всегда спрашивайте себя, является ли дочерний класс частным случаем родительского класса. Например, выражение «Чай — это напиток» имеет смысл, а «Напиток — это чай» — нет.

**КЛЮЧЕВЫЕ МОМЕНТЫ**

- Дочерний класс *расширяет* класс-родитель.
- Дочерний класс наследует все *публичные* переменные и методы своего родителя, но не наследует его приватные переменные и методы.
- Унаследованные методы *могут* быть переопределены. Переменные экземпляра *не могут* быть переопределены (хотя их можно *заново объявить* в классе, но этого почти никогда не требуется).
- Используйте проверку на соответствие (IS-A), чтобы подтвердить правильность своей иерархии наследования. Если X *расширяет* Y, значит, X — это Y.
- Отношение IS-A работает только в одну сторону. Гиппопотам — это животное, но не все животные — гиппопотамы.
- Когда метод, переопределенный в дочернем классе, вызывается из экземпляра дочернего класса, в ответ будет выполнена переопределенная версия *(которая находится в самом низу иерархии)*.
- Если класс B расширяет A и C расширяет B, то класс B — это A, класс C — это B, а класс C — это A.

# В чем же настоящая ценность наследования?

Используя при проектировании наследование, вы получаете множество возможностей ООП. Вы избавляетесь от дублирования кода, абстрагируя общее поведение для группы классов и помещая его в один класс-родитель. Если вам понадобится изменить такой код, то придется сделать это только один раз и в одном месте, *и изменения магическим образом отразятся на всех классах, наследующих это поведение*. Конечно, магия здесь не при чем, но это действительно очень просто: внести изменения и повторно скомпилировать класс. И все. **Не нужно трогать дочерние классы!**

**Просто добавьте свежеизмененный родительский класс, и все расширяющие его классы будут автоматически использовать новую версию.**

Программа на языке Java — не что иное, как набор классов, поэтому дочерние классы не должны перекомпилироваться вслед за классом-родителем. Пока родительский класс не *разрушает* ничего, что связано с его потомками, все будет хорошо.

Позже в этой книге мы обсудим, что в данном контексте означает слово «разрушать». Пока думайте об этом как об изменениях в родительском классе, затрагивающих аргументы методов, тип возвращаемого значения, имя метода и т. д., от которых зависят классы-потомки.

1. **Вы избавляетесь от повторения кода.**

   Размещайте общий код в одном месте и позволяйте дочерним классам его наследовать. Если вы захотите его измени придется исправить только один участок кода, и все остальные (то есть дочерние классы) увидят эти изменения.

2. **Вы определяете общи протокол для группы классов.**

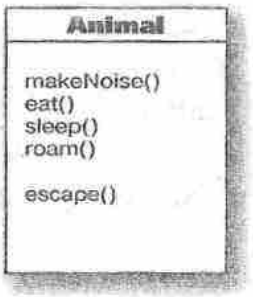
## Благодаря наследованию вы можете гарантировать, что все классы, сгруппированные под одним родительским типом, содержат все методы этого типа*.

**Другими словами, вы определяете общий протокол для набора классов, связанных наследованием.**

Определяя в родительском классе методы, которые могут быть унаследованы дочерними классами, вы сообщаете другому коду о существовании протокола, в котором говорится: «Все мои подтипы (то есть дочерние классы) могут делать это благодаря методам, которые выглядят вот так...»

Если говорить кратко, вы заключаете *контракт*.

Класс Animal устанавливает общий протокол для всех своих подтипов:




Вы говорите всем, что любой Animal может выполнить эти четыре действия. Нужно указать аргументы методов и типы возвращаемых значений.

Запомните, когда мы говорим *«любой Animal»*, то имеем в виду класс Animal и *любой другой класс, который его наследует*. Что также означает любой класс, *относящийся к Animal в каком-либо участке иерархии наследования*.

Но самое интересное — *полиморфизм* — мы приберегли напоследок.

Когда вы объявляете общий тип для группы классов, *вы можете заменять родительский класс дочерним везде, где предполагается его присутствие*.

Простите, что?

Не волнуйтесь, мы только начали объяснять. Через две страницы вы во всем разберетесь.

* Когда мы говорим *«все методы»*, то имеем в виду *«все наследуемые методы»*, что на данном этапе означает *«все публичные методы»*, хотя позже мы еще уточним это определение.

### И меня это должно волновать, потому что...

... вы получаете преимущества полиморфизма.

### Это важно для меня, потому что...

...вы получаете возможность ссылаться на объект дочернего типа через ссылку, объявленную с родительским типом.

### И это означает, что...

...вы можете писать универсальный код. Более чистый, эффективный и простой. Подобный код не только легче *писать*, но и гораздо проще *расширять*; причем такими способами, о которых вы в самом начале и подумать не могли.

Это означает, что, пока вы отдыхаете на тропических островах, ваши коллеги могут обновлять программу, даже не заглядывая в исходники. Как это работает, вы увидите на следующей странице.

Не знаем, как вы, но мы считаем, что перспектива отпуска на тропических островах очень вдохновляет.

**Чтобы понять работу полиморфизма, нужно немного вернуться назад и вспомнить, как обычно объявляют ссылки и создают объекты...**

## Три шага при объявлении и инициализации

```
1        3  2
Dog myDog = new Dog();
```

### 1 Объявляем ссылочную переменную.

```
Dog myDog = new Dog();
```

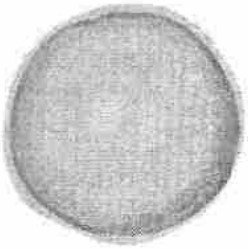
Говорим JVM выделить место для ссылочной переменной. Ссылка навсегда получает тип Dog. Иными словами, создаем пульт дистанционного управления, у которого есть кнопки для работы с объектом Dog, но не Cat, Button или Socket.

Dog

### 2 Создаем объект.

```
Dog myDog = new Dog();
```

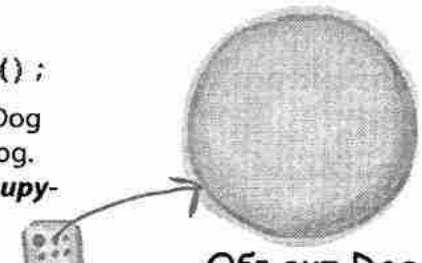
Говорим JVM выделить в куче, управляемой сборщиком мусора, место для нового объекта Dog.


Объект Dog

### 3 Связываем объект и ссылку.

```
Dog myDog = new Dog();
```

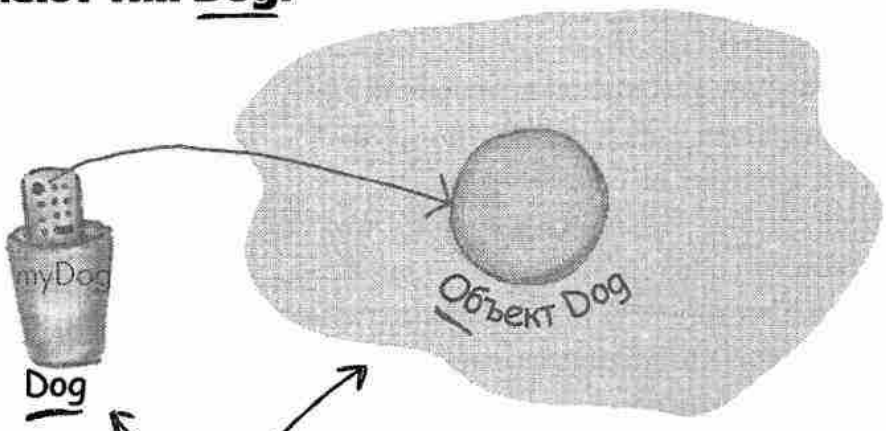
Присваиваем новый объект Dog ссылочной переменной myDog. Другими словами, ***программируем пульт управления***.


Объект Dog

Dog

**Важно отметить, что типы ссылки и объекта совпадают.**

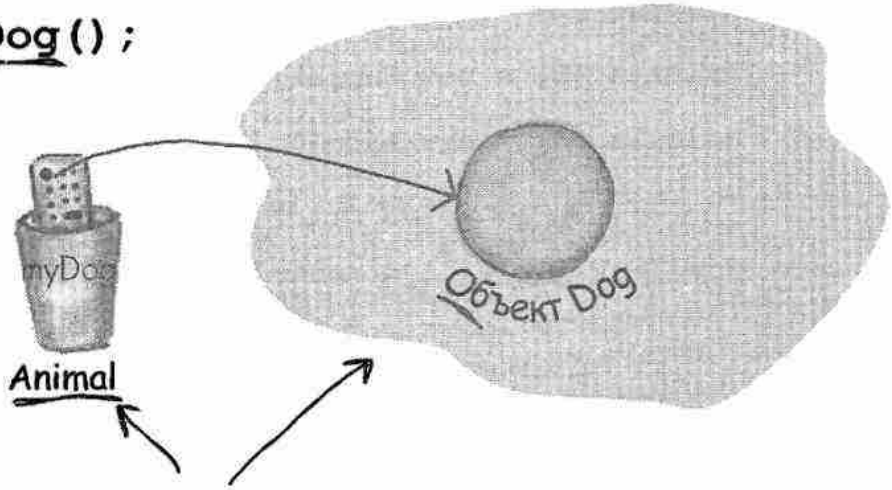
**В этом примере оба имеют тип Dog.**




Оба имеют один и тот же тип. Тип ссылочной переменной объявлен как Dog, а объект создан как новый экземпляр класса Dog.

**Однако полиморфизм допускает, что ссылка и объект могут иметь *разные* типы.**

```
Animal myDog = new Dog();
```




Они имеют разные типы. Тип ссылочной переменной объявлен как Animal, но объект создан как новый экземпляр класса Dog.

## Благодаря полиморфизму тип ссылки может быть родительским для типа самого объекта.

При объявлении ссылочной переменной ей можно присвоить любой объект, который проходит проверку на соответствие для типа этой ссылки. Иными словами, все, что *расширяет* тип объявленной ссылочной переменной, может быть ей присвоено. ***Это позволяет создавать такие объекты, как полиморфные массивы.***

## Хорошо, возможно, пример все прояснит.

Объявляем массив типа Animal, то есть массив, который будет хранить объекты типа Animal.

```
Animal[] animals = new Animal[5];
animals [0] = new Dog();
animals [1] = new Cat();
animals [2] = new Wolf();
animals [3] = new Hippo();
animals [4] = new Lion();

for (int i = 0; i < animals.length; i++) {

    animals[i].eat();

    animals[i].roam();
}
```

Но смотрите, что происходит... Вы можете поместить экземпляр любого потомка Animal в массив типа Animal!

Вот и самая интересная часть истории о полиморфизме (суть примера): вы проходитесь в цикле по массиву, вызываете методы класса Animal, и каждый объект делает все как положено!

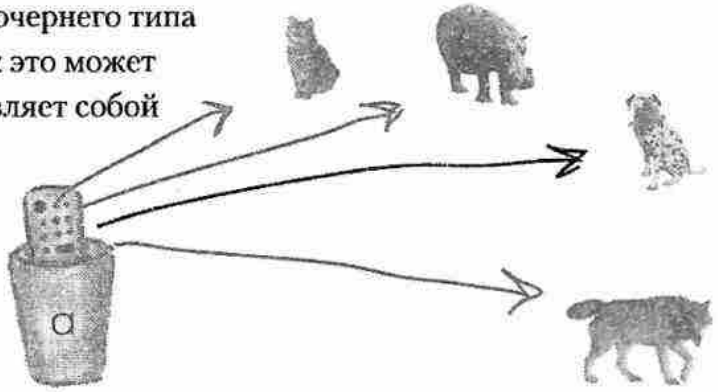
Когда i равен 0, по индексу 0 в массиве находится объект Dog, поэтому вы получаете его метод eat(). Когда i равен 1, вы получаете метод eat(), принадлежащий объекту Cat.

То же самое происходит с roam().

## Но погодите! Это еще не все!

### У вас могут быть полиморфные аргументы и типы возвращаемых значений.

Если вы можете объявить ссылочную переменную родительского типа (например, Animal) и присвоить ей объект дочернего типа (скажем, Dog), то подумайте, как это может сработать, когда ссылка представляет собой аргумент для метода...




```
class Vet {

   public void giveShot(Animal a) {

      // Управляем животными

      // по другую сторону параметра a

      a.makeNoise();

   }

}
```

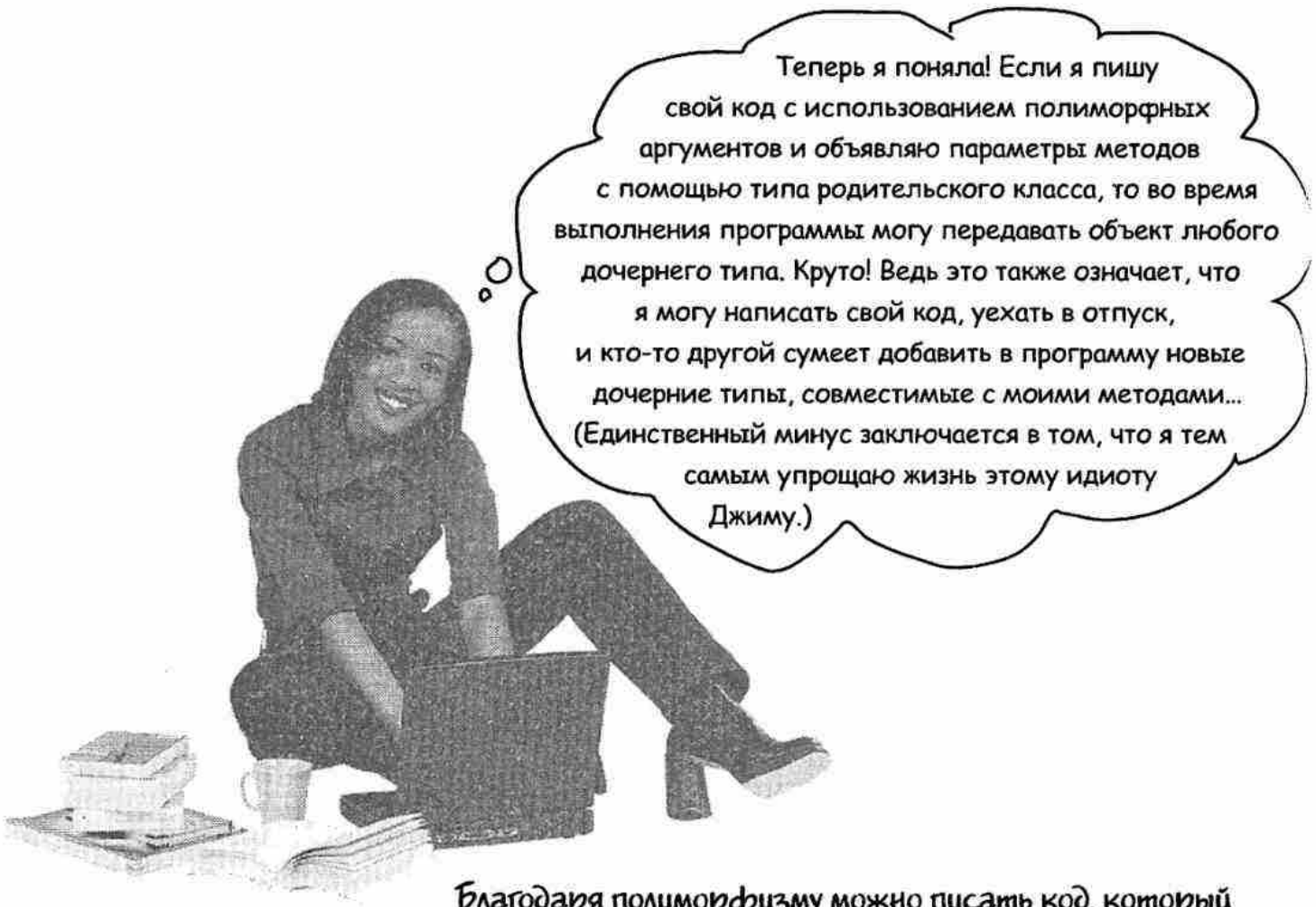
Параметр Animal может принимать в качестве аргумента любой тип. Когда класс Vet заканчивает выполнение метода giveShot(), он вызывает из Animal метод makeNoise(), и какой бы потомок Animal ни находился в куче, он запускает свою версию makeNoise().

```
class PetOwner {

   public void start() {

      Vet v = new Vet();

      Dog d = new Dog();

      Hippo h = new Hippo();

      v.giveShot(d);

      v.giveShot(h);

   }

}
```

Метод giveShot() из класса Vet может принимать любой объект Animal. Все будет хорошо, пока тип передаваемого объекта представляет собой дочерний класс Animal.

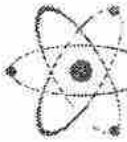
Запускается метод makeNoise() из Dog.

Запускается метод makeNoise() из Hippo.




### Благодаря полиморфизму можно писать код, который не придется менять с появлением в программе новых типов дочерних классов.

Помните класс Vet? Если при его создании вы будете использовать аргументы типа *Animal*, то ваш код сможет работать с любым *потомком* Animal. Другими словами, если кто-то со стороны захочет получить возможности вашего класса Vet, ему нужно будет лишь убедиться, что *его* новые типы расширяют класс Animal. Методы, принадлежащие Vet, будут продолжать работать несмотря на то, что он был написан без малейшего представления о новых подтипах Animal.



## Сила мозга

Есть ли гарантия, что полиморфизм всегда будет так работать? Почему можно утверждать, что любой *дочерний класс* будет содержать методы, которые, как вы предполагаете, вызываются из его *родителя*?

## Это не глупые вопросы

**В: Существуют ли на практике ограничения для уровней наследования? Насколько глубоко можно зайти?**

О: Если вы посмотрите на Java API, то увидите, что большинство иерархий наследования разрастаются в ширину, то есть они не очень глубокие. Многие из них достигают одного-двух уровней, хотя бывают исключения (особенно в классах, связанных с GUI). Вскоре вы поймете, что более разумно сохранять незначительную глубину иерархий наследования, несмотря на то что строгих ограничений нет.

**В: Если у меня нет доступа к коду класса, но хочется изменить поведение его метода, могу ли я использовать для этого наследование? Можно расширить «плохой» класс и переписать метод, добавив собственный улучшенный код?**

О: Да. Это одна из замечательных особенностей ООП, и в отдельных случаях она избавляет от необходимости переписывать класс с нуля или искать программиста, который «потерял» исходный код.

**В: *Любой* ли класс можно расширить? Или, как и в случае с членами класса, нельзя наследовать класс, если он приватный?**

О: Приватных классов не существует, если не считать особых случаев с *вложенными* классами, которые мы пока не рассматривали. Но существует три способа запретить наследовать класс.

Во-первых, это контроль доступа. Хотя класс и *нельзя* пометить словом `private`, он все равно *может* не быть публичным (так происходит, когда вы не используете при объявлении слово `public`). Непубличные классы могут наследоваться только внутри одного пакета. Классы из других пакетов не могут наследовать (или даже использовать, если уж на то пошло) непубличные классы.

Во-вторых, это использование ключевого слова `final`. Оно будет означать, что данный класс — это конечная точка в иерархии наследования. Никто и никогда не может расширить такой класс.

В-третьих, это наличие у класса только приватных конструкторов (мы поговорим о них в главе 9), что тоже предотвращает возможность наследования.

**В: Для чего может понадобиться финализированный класс? Каковы преимущества предотвращения наследования класса?**

О: Как правило, вам не придется делать свои классы финализированными. Но если нужна гарантия, что методы будут всегда работать так, как вы изначально задумывали (так как они не могут быть переопределены), то можете получить ее, используя финализированный класс. Именно по этой причине множество классов в Java API финализированы. Например, String — один из таких классов. Представьте, что может произойти, если кто-то со стороны изменит поведение строк!

**В: Можно ли финализировать отдельный *метод*, а не весь *класс*?**

О: Если вы хотите защитить метод от переопределения, пометьте его модификатором `final`. Обозначив таким образом весь *класс*, вы гарантируете, что *ни один* из его методов никогда не будет переопределен.

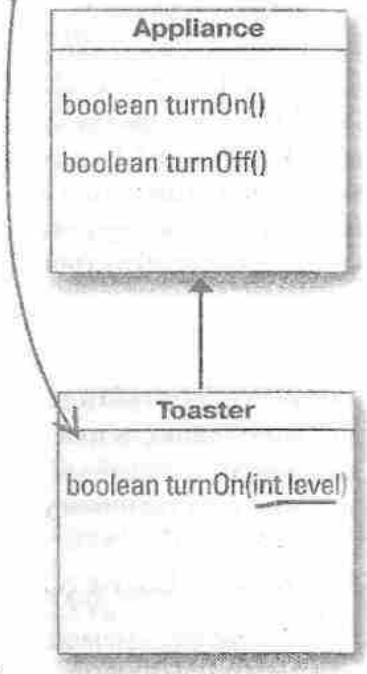
# Заключение контракта: правила переопределения

Переопределяя метод, принадлежащий родительскому классу, вы подписываете контракт. Например, в нем говорится: «Я не принимаю никаких аргументов и возвращаю булево значение». Иными словами, аргументы и возвращаемые значения переопределяемых методов внешне должны выглядеть *точно так же*, как в оригинальном методе родительского класса.

**Методы — это и *есть* контракт.**

Если учесть полиморфизм, то при выполнении программы будет работать переопределенная классом Toaster версия метода turnOn(), которая унаследована от Appliance. Прежде чем решить, можете ли вы вызвать конкретный метод из ссылки, компилятор смотрит на ее тип. Если переменная *Appliance* ссылается на объект Toaster, то компилятор смотрит, есть ли у класса Appliance метод, который вызывается из *ссылки* типа Appliance. Но при выполнении программы JVM интересует не тип ссылки (Appliance), а сам объект *Toaster*, находящийся в куче. Поэтому, если компилятор уже *одобрил* вызов метода, переопределенный метод имеет те же аргументы и тип возвращаемого значения. Иначе кто-нибудь может вызвать из ссылки типа Appliance метод turnOn() без аргументов, несмотря на то что в классе Toaster этот метод принимает целочисленное значение. Какой же метод будет вызван? Метод, который находится в Appliance. Таким образом, ***метод turnOn(int level) в классе Toaster не переопределен***!

Это не переопределение! Нельзя менять аргументы в переопределяемом методе!




На самом деле это вполне допустимая перегрузка, но не переопределение.

### ① Аргументы должны совпадать, а типы возвращаемых значений должны быть совместимы.

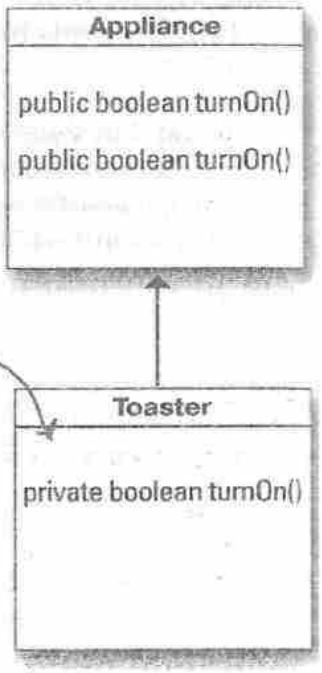
Контракт для родительского класса определяет, как остальной код сможет использовать метод. Что бы он ни принимал в качестве аргументов, дочерний класс, переопределяя его, должен применять такие же аргументы. И что бы он ни возвращал, переопределенный метод должен возвращать значение того же или дочернего типа. Помните, объект дочернего класса обязан уметь все, о чем объявил его родитель, поэтому безопасно возвращать дочерний класс в ситуациях, когда ожидается класс-родитель.

### ② Метод не может быть менее доступным.

Это означает, что уровень доступа должен быть таким же или менее строгим. Например, вы не можете переопределить публичный метод, сделав его приватным. Представьте, какой сюрприз получит внешний код, вызывая, как он *считает* на уровне компиляции, публичный метод, если внезапно во время выполнения программы JVM закроет дверь перед его носом, потому что вызванная переопределенная версия приватна!

Пока мы изучили два уровня доступа: приватный (private) и публичный (public). Два других описываются в главе 17 и в Приложении Б. Существует еще одно правило переопределения, связанное с обработкой исключений, но мы рассмотрим его в главе 11.

Недопустимо! Так переопределять нельзя, потому что уровень доступа становится более строгим. Это также нельзя назвать допустимой перегрузкой, так как аргументы не изменяются.



# Перегрузка метода

Перегрузка метода — это не что иное, как наличие нескольких методов с одним именем, но разными наборами аргументов. И все. В процессе перегрузки методов полиморфизм не задействуется!

Перегрузка позволяет создавать несколько версий одного метода с разным списком аргументов, делая их более удобными для вызова. Например, если у вас есть метод, который принимает только тип int, то перед вызовом этого метода код должен преобразовать, скажем, double в int. Перегрузив метод и добавив новую версию, принимающую double, вы упростите его вызов для внешнего кода. Подробнее об этом мы поговорим, когда дойдем до конструкторов в главе 9, посвященной жизненному циклу объекта.

Хотя перегруженные методы и не пытаются соблюдать контракт для выполнения условий полиморфизма, они обладают гораздо большей гибкостью.

Перегруженный метод — просто другой метод с тем же именем. Это не имеет ничего общего ни с наследованием, ни с полиморфизмом. Перегруженный метод — это не то же самое, что переопределенный метод.

1. **Типы возвращаемых значений могут быть разными.**

   Вы можете изменять типы возвращаемых значений в перегруженных методах. При этом список аргументов тоже должен отличаться.

2. **Вы не можете изменить только тип возвращаемого значения.**

   Если изменить лишь тип возвращаемого значения, это будет некорректная *перегрузка* — компилятор решит, что вы пытаетесь *переопределить* метод. Кроме того, это считается недопустимым, так как типы возвращаемых значений в дочернем классе и его родителе должны совпадать. Чтобы перегрузить метод, нужно изменить набор аргументов. При этом можно заменить тип возвращаемого значения любым другим.

3. **Вы можете как угодно изменять уровни доступа.**

   При перегрузке есть возможность добавлять новые версии метода с более ограниченным доступом. Это не имеет значения, так как новый метод не обязан выполнять контракт перегружаемого метода.

**Пример корректной перегрузки метода:**

```
public class Overloads {

   String uniqueID;

   public int addNums(int a, int b) {
      return a + b;
   }

   public double addNums(double a, double b) {
       return a + b;
   }

   public void setUniqueID(String theID) {
      // Много разного проверочного кода
      // и затем:
      uniqueID = theID;
   }

   public void setUniqueID(int ssNumber) {
      String numString = "" + ssNumber;
      setUniqueID(numString);
   }
}
```

**Упражнение:** смешанные сообщения

# Смешанные сообщения

```
a = 6;      56
b = 5;      11
a = 5;      65
```

Внизу приведена небольшая Java-программа. Но один ее блок пропал! Ваша задача — сопоставить варианты блоков кода (слева) с результатом, который можно будет увидеть, если вставить их в программу. Не все строки с результатами будут задействованы, хотя некоторые могут использоваться несколько раз. Соедините линиями варианты блоков кода с соответствующим результатом в командной строке.

## Программа:

```
class A {
  int ivar = 7;
  void m1() {
    System.out.print("A's m1, ");
  }
  void m2() {
    System.out.print("A's m2, ");
  }
  void m3() {
    System.out.print("A's m3, ");
  }
}

class B extends A {
  void  m1() {
    System.out.print("B's m1, ");
  }
}

class C extends B {
  void m3() {
    System.out.print("C's m3, "+(ivar + 6));
  }
}

public class Mixed2 {
  public static void main(String [] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    A a2 = new C();
    [                    ]   <-- Сюда вставляются варианты кода (три строки)
  }
}
```

## Варианты кода:

```
b.m1();
c.m2();
a.m3();
```

---

```
c.m1();
c.m2();
c.m3();
```

---

```
a.m1();
b.m2();
c.m3();
```

---

```
a2.m1();
a2.m2();
a2.m3();
```

## Результат:

```
A's m1, A's m2, C's m3, 6

B's m1, A's m2, A's m3,

A's m1, B's m2, A's m3,

B's m1, A's m2, C's m3, 13

B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13
```

# Поработайте компилятором

Какие пары методов (A-B) при добавлении в класс, приведенный слева, позволят скомпилировать программу и выведут на экран представленный текст? Метод A помещается в класс Monster, а метод B — в класс Vampire.

```
public class MonsterTestDrive {
  public static void main(String [] args) {
    Monster [] ma = new Monster[3];
    ma[0] = new Vampire();
    ma[1] = new Dragon();
    ma[2] = new Monster();
    for(int x = 0; x < 3; x++) {
      ma[x].frighten(x);
    }
  }
}

class Monster {
  A
}

class Vampire extends Monster {
  B
}

class Dragon extends Monster {
  boolean frighten(int degree) {
    System.out.println("Огненное дыхание");
    return true;
  }
}
```

```
File Edit Window Help SaveYourself
% java MonsterTestDrive
Укусить?
Огненное дыхание
Гррррррр
```

1

A
```
boolean frighten(int d) {
  System.out.println("Гррррррр");
  return true;
}
```

B
```
boolean frighten(int x) {
  System.out.println("Укусить?");
  return false;
}
```

2

A
```
boolean frighten(int x) {
  System.out.println("Гррррррр");
  return true;
}
```

B
```
int frighten(int f) {
  System.out.println("Укусить?");
  return 1;
}
```

3

A
```
boolean frighten(int x) {
  System.out.println("Гррррррр");
  return false;
}
```

B
```
boolean scare(int x) {
  System.out.println("Укусить?");
  return true;
}
```

4

A
```
boolean frighten(int z) {
  System.out.println("Гррррррр");
  return true;
}
```

B
```
boolean frighten(byte b) {
  System.out.println("Укусить?");
  return true;
}
```

# Головоломка у бассейна

Ваша ***задача*** — взять фрагменты кода со дна бассейна и заменить ими пропущенные участки программы. Можете использовать один фрагмент несколько раз, но при этом необязательно задействовать все фрагменты. Ваша ***цель*** — создать набор классов, которые скомпилируются и запустятся как единая программа. Но будьте внимательны — на самом деле все сложнее, чем кажется.

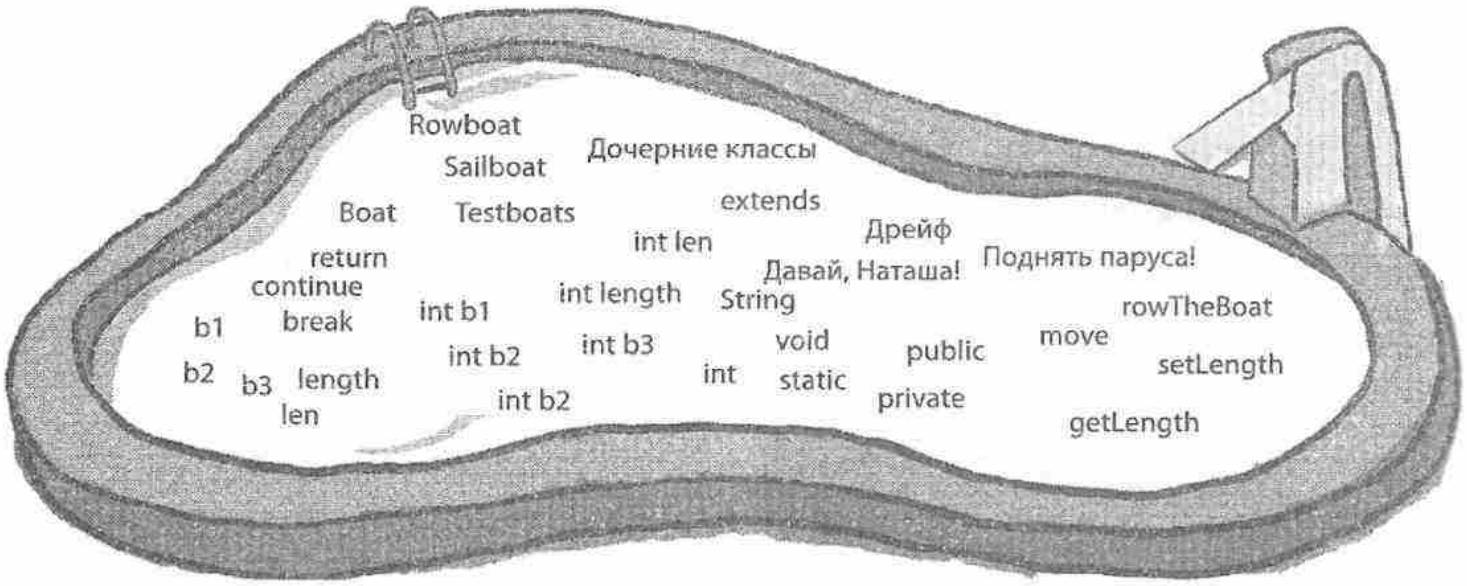
```
public class Rowboat _______ _______ {
   public __________ rowTheBoat() {
      System.out.print("Давай, Наташа!");
   }
}
```

```
public class _______ {
   private int _________ ;
   _______ void _________ ( ______ ) {
      length = len;
   }
   public int getLength() {
      _______ _________ ;
   }
   public __________ move() {
      System.out.print("__________");
   }
}
```

```
public class TestBoats {
   ______ ______ _______ main(String[] args
      _________ b1 = new Boat();
      Sailboat b2 = new _________();
      Rowboat _______ = new Rowboat();
      b2.setLength(32);
      b1._________();
      b3._________();
      _______.move();
   }
}
```

```
public class _________ _______ Boat {
   public ______ _________() {
      System.out.print("__________");
   }
}
```

**Результат:** `Дрейф Дрейф Поднять паруса!`



## Поработайте компилятором

Набор 1 будет работать.

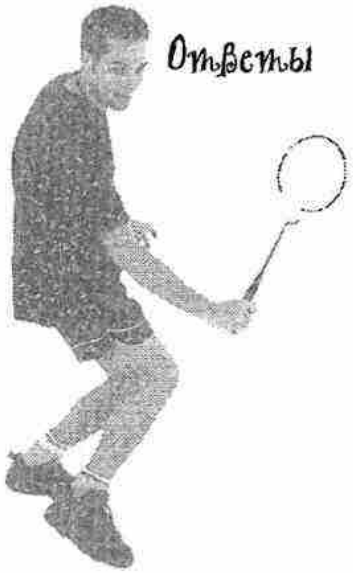
Набор 2 не скомпилируется из-за типа возвращаемого значения в классе Vampire (int).

Метод frighten() класса Vampire в варианте B некорректно переопределяет или перегружает свой аналог из класса Monster. Недостаточно изменить тип возвращаемого значения, чтобы получить корректную перегрузку. Поскольку тип int несовместим с boolean, это нельзя считать правильным переопределением. Если вы изменяете только тип возвращаемого значения, то он должен быть совместим с аналогичным типом из родительского класса — тогда это будет переопределение.

Наборы 3 и 4 скомпилируются, но выведут на экран:

```
Грррррррр
Огненное дыхание
Грррррррр
```

Помните, что класс Vampire не *переопределяет* метод frighten() из класса Monster. Метод frighten() в классе Vampire из четвертого набора принимает значение типа byte, а не int.




---

## Варианты кода:

```
b.m1();
c.m2();
a.m3();
```

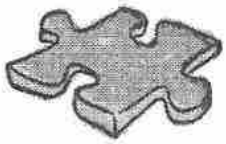
```
c.m1();
c.m2();
c.m3();
```

```
a.m1();
b.m2();
c.m3();

a2.m1();
a2.m2();
a2.m3();
```

## Смешанные сообщения

### Результат:

```
A's m1, A's m2, C's m3, 6
B's m1, A's m2, A's m3,
A's m1, B's m2, A's m3,
B's m1, A's m2, C's m3, 13
B's m1, C's m2, A's m3,
B's m1, A's m2, C's m3, 6
A's m1, A's m2, C's m3, 13
```



## Головоломка у бассейна

```
public class Rowboat extends Boat {
    public void rowTheBoat() {
        System.out.print("Давай, Наташа!");
    }
}
public class Boat {
    private int length ;
    public void setLength ( int len ) {
        length = len;
    }
    public int getLength() {
        return length ;
    }
    public void move() {
        System.out.print("Дрейф");
    }
}
```

```
public class TestBoats {
    public static void main(String[] args){
        Boat b1 = new Boat();
        Sailboat b2 = new Sailboat();
        Rowboat b3 = new Rowboat();
        b2.setLength(32);
        b1.move();
        b3.move();
        b2.move();
    }
}
public class Sailboat extends Boat {
    public void move() {
        System.out.print("Поднять паруса!");
    }
}
```

**Результат:**

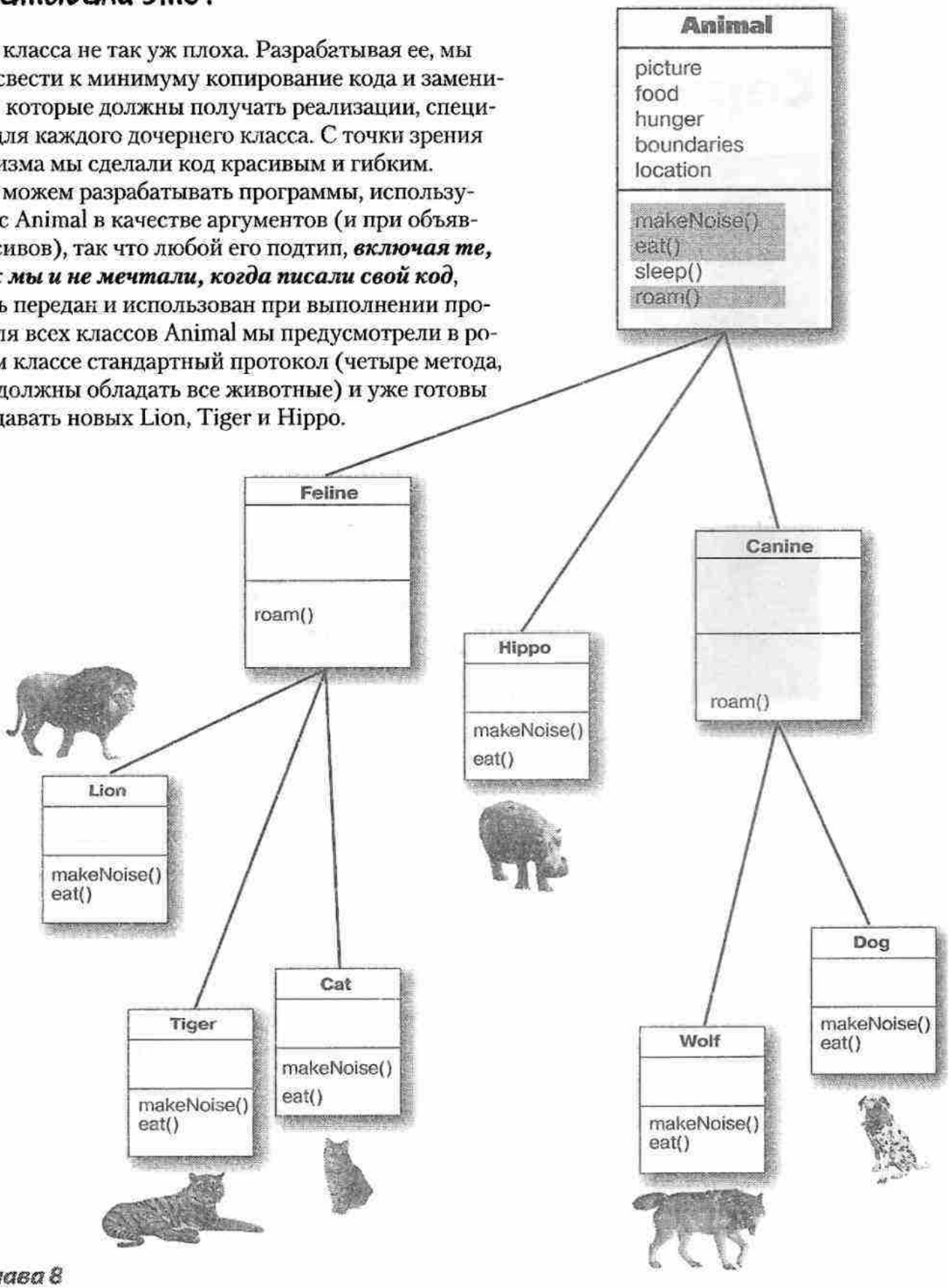
```
Дрейф Дрейф Поднять паруса!
```

# Серьезный полиморфизм

**Наследование — это только начало.** Чтобы задействовать полиморфизм, нужны интерфейсы (нет, не те, что предназначены для пользователя). Пора оставить позади обычное наследование и выйти на новый уровень гибкости и масштабируемости. Его может обеспечить только архитектура, основанная на интерфейсах. Самые замечательные возможности Java существуют лишь благодаря интерфейсам, поэтому, даже если вы не используете их в работе, все равно *придется* столкнуться с ними. Но мы уверены, что вы *захотите* использовать их. ***Вы удивитесь, как могли жить без них раньше.*** Что такое интерфейс? Это на 100 % абстрактный класс. Что такое абстрактный класс? Это класс, для которого нельзя создать экземпляр. Зачем это нужно? Вы скоро увидите. Если вы думаете об окончании предыдущей главы, где мы использовали полиморфические аргументы так, чтобы единственный метод Vet мог принимать дочерние классы Animal всех типов, то это было только начало. Интерфейсы — это ***«поли»*** в слове «полиморфизм», ***«аб»*** в слове «абстрактный» и ***«кофеин»*** в Java.

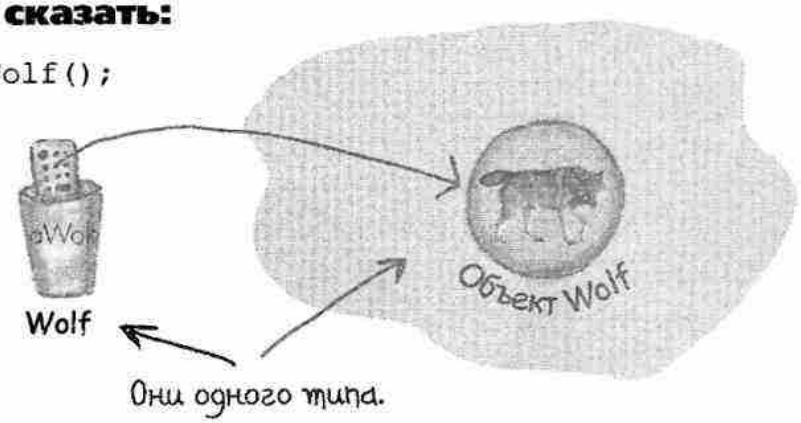
# Мы что-то забыли, когда разрабатывали это?

Структура класса не так уж плоха. Разрабатывая ее, мы старались свести к минимуму копирование кода и заменили методы, которые должны получать реализации, специфические для каждого дочернего класса. С точки зрения полиморфизма мы сделали код красивым и гибким. Теперь мы можем разрабатывать программы, использующие класс Animal в качестве аргументов (и при объявлении массивов), так что любой его подтип, ***включая те, о которых мы и не мечтали, когда писали свой код***, может быть передан и использован при выполнении программы. Для всех классов Animal мы предусмотрели в родительском классе стандартный протокол (четыре метода, которыми должны обладать все животные) и уже готовы начать создавать новых Lion, Tiger и Hippo.



**Мы знаем, что можем сказать:**

```
Wolf aWolf = new Wolf();
```

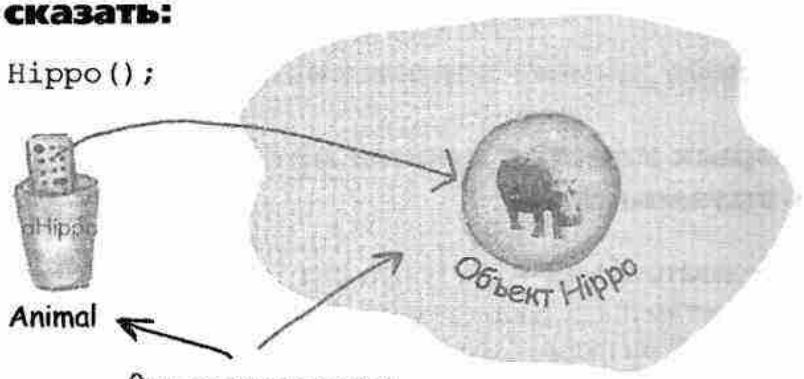



Ссылка Wolf на объект Wolf.

Они одного типа.

**И мы знаем, что можем сказать:**

```
Animal aHippo = new Hippo();
```

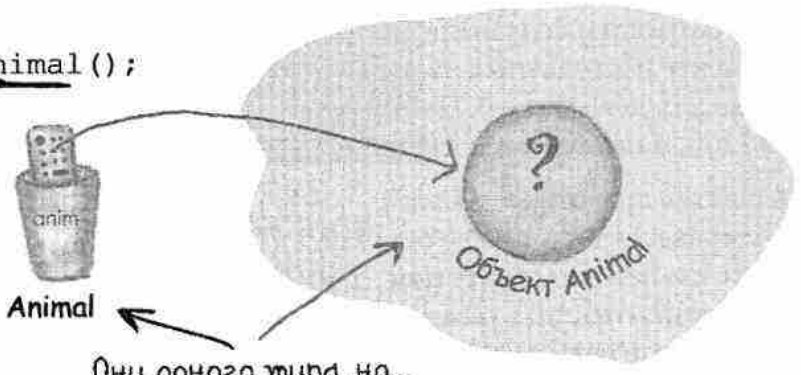



Ссылка типа Animal на объект Hippo.

Они разного типа.

**Но вот что странно:**

```
Animal anim = new Animal();
```

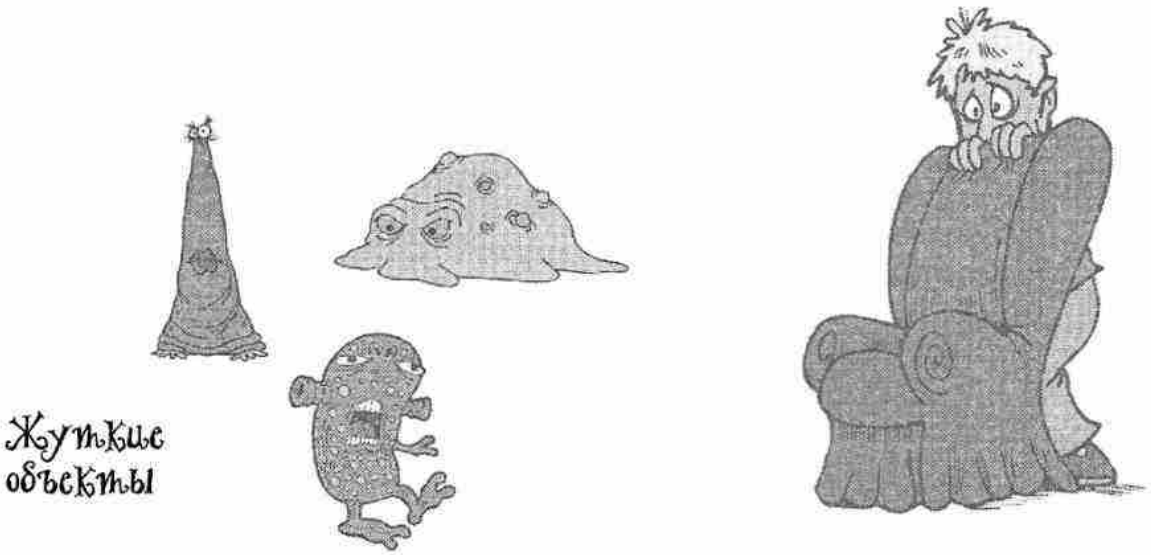



Ссылка типа Animal на объект класса Animal.

Они одного типа, но...

Как же выглядит этот объект Animal?

## Как выглядит объект new Animal()?




### Какие значения хранит переменная экземпляра?

### Для некоторых классов просто *не* следует создавать экземпляры!

Имеет смысл создавать объект Wolf, Hippo или Tiger, но что конкретно представляет собой объект Animal? Какой он формы? Какого цвета, размера, сколько у него ног?..

Попытка создать объект типа Animal похожа на **кошмарную аварию с транспортером в Star Trek™**, когда что-то плохое случилось с буфером во время телепортирования с помощью луча.

Но что с этим делать? Нам *нужен* класс Animal для наследования и полиморфизма. Однако мы хотим, чтобы программисты создавали экземпляр только для менее абстрактных *подклассов* класса Animal, а не для него самого. Нам нужны объекты Tiger и Lion, ***а не объекты Animal***.

К счастью, есть способ, предотвращающий создание экземпляров класса, а точнее останавливающий любого, кто захочет задать **new** для этого типа. Нужно лишь отметить класс словом **abstract**, и компилятор остановит процесс создания экземпляра этого типа в любой части кода.

Вы по-прежнему сможете использовать такой абстрактный тип в качестве ссылочного. Но на самом деле это большой вопрос — зачем вам вообще абстрактный класс (для использования в качестве полиморфического аргумента, в виде типа возвращаемого значения или для создания полиморфического массива).

Разрабатывая структуру наследования класса, вы должны решить, какие классы будут *абстрактными*, а какие — *конкретными*. Конкретные классы должны быть достаточно специфическими, чтобы имело смысл создавать для них экземпляр. Делая класс *конкретным*, вы лишь говорите, что создавать объекты данного типа — это нормально.

Сделать класс абстрактным легко — поместите ключевое слово **abstract** перед объявлением класса:

```
abstract class Canine extends Animal
{
   public void roam() { }
}
```

# Компилятор не позволит вам создать экземпляр для абстрактного класса

Абстрактный класс означает, что никто и никогда не сможет создать для него новый экземпляр. Вы все еще сможете использовать этот класс как объявленный ссылочный тип, например для полиморфизма, но вам не придется переживать, что кто-то создаст объекты данного типа. Компилятор *позаботится* об этом.

```
abstract public class Canine extends Animal
{
   public void roam() { }
}
-------------------------------------------------
public class MakeCanine {
   public void go() {
      Canine c;
      c = new Dog();
      c = new Canine();
      c.roam();
   }
}
```

Это нормально, так как вы всегда можете присвоить объект дочернего класса ссылке на родительский класс, даже если тот является абстрактным.

Класс Canine абстрактный, поэтому компилятор не даст вам этого сделать.

```
File Edit Window Help BeamMeUp
% javac MakeCanine.java
MakeCanine.java:5: Canine is abstract;
cannot be instantiated
      c = new Canine();
          ^
1 error
```

**Абстрактный класс** практически* не используется, у него нет значения, нет цели в жизни, если только он не ***расширенный***.

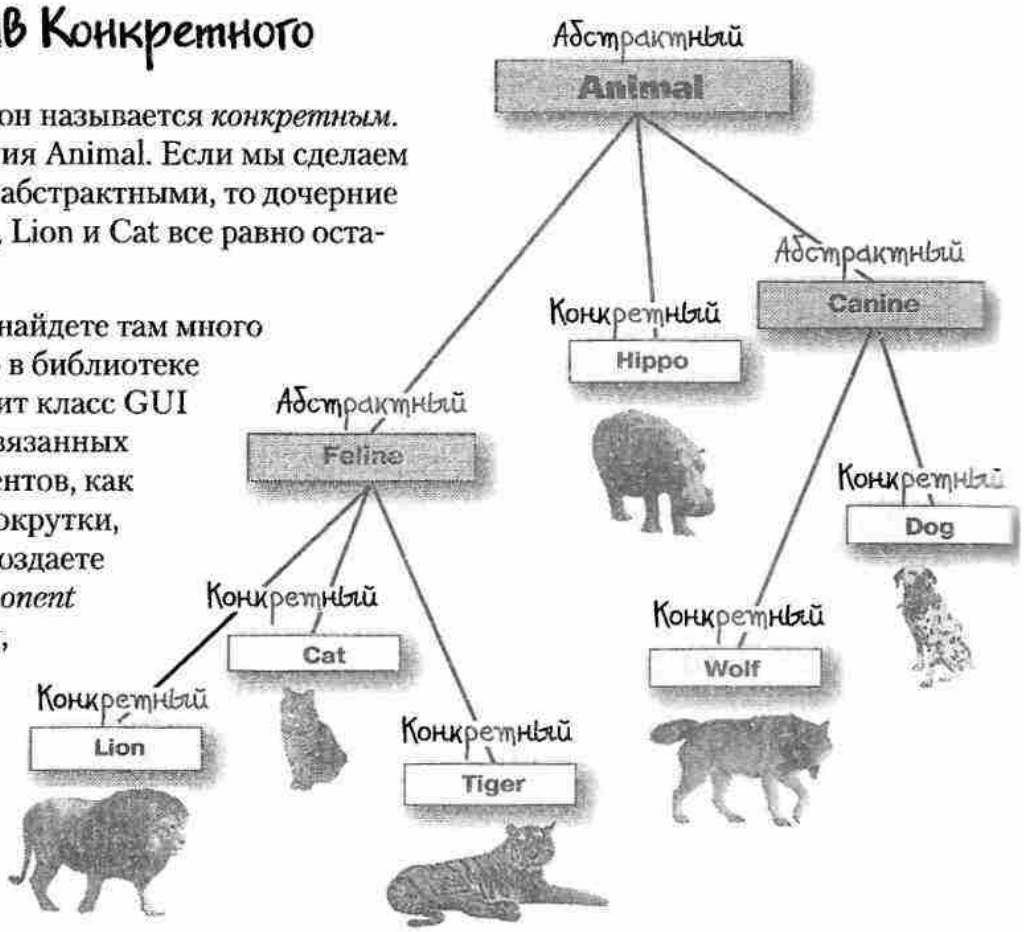
Элементы абстрактного класса, делающие работу при выполнении программы, являются **экземплярами** его ***дочернего класса***.

* Здесь есть исключение — абстрактный класс может содержать статические элементы (смотрите главу 10).

# Абстрактный против Конкретного

Если класс не абстрактный, то он называется *конкретным*. Рассмотрим дерево наследования Animal. Если мы сделаем классы Animal, Canine и Feline абстрактными, то дочерние классы Hippo, Wolf, Dog, Tiger, Lion и Cat все равно останутся конкретными.

Пробегитесь по Java API, и вы найдете там много абстрактных классов, особенно в библиотеке для создания GUI. Как выглядит класс GUI Component? Он базовый для связанных с GUI классов: для таких элементов, как кнопки, поля ввода, полосы прокрутки, диалоговые окна и т. п. Вы не создаете экземпляр общего класса *Component* и затем помещаете его на экран, а создаете, например, объект JButton. Другими словами, вы создаете экземпляр только для *конкретных наследников* класса Component, но никогда не делаете этого для него самого.




## Сила мозга

### Абстрактный или конкретный?

Как вы узнаете, когда классу надлежит быть абстрактным? ***Вино*** — это, вероятно, абстрактное понятие. А если мы говорим о ***красном*** и ***белом***? Возможно, это тоже абстрактность (по крайней мере для некоторых из нас). Но при каких условиях объекты иерархии действительно становятся конкретными?

Каким вы сделаете класс ***PinotNoir*** — конкретным или абстрактным? Похоже, что Camelot Vineyards 1997 Pinot Noir — конкретный экземпляр в любом случае. Но как узнать наверняка?

Посмотрите на дерево наследования Animal. Правильно ли мы выбрали, кого сделать абстрактным, а кого — конкретным? Хотите ли вы изменить что-нибудь в дереве наследования (не считая, конечно, добавления новых видов Animal)?

# Абстрактные методы

Кроме классов вы можете делать абстрактными и *методы*. Если класс абстрактный, значит, его необходимо *расширить*; если же абстрактным является метод, он должен быть *переопределен*. Вы можете решить, что поведение абстрактного класса (полностью или частично) не имеет смысла, если оно не реализовано более специфическими подклассами. Другими словами, не получится представить реализацию общего метода, которая может оказаться полезной для дочерних классов. Как бы выглядел общий метод *eat()*?

## У абстрактных методов нет тела!

Поскольку вы уже поняли, что в абстрактном методе нет кода, который имел бы смысл, то не станете добавлять тело метода. Поэтому нет и фигурных скобок — просто закрываем объявление точкой с запятой.

```
public abstract void eat();
```

У метода нет тела!

Закрываем его точкой с запятой.

**Если вы объявляете абстрактный *метод*, то должны сделать абстрактным и его *класс*. Нельзя иметь абстрактный метод в неабстрактном классе.**

Даже помещая один абстрактный метод в класс, вы обязаны сделать этот класс абстрактным. Однако *можно* совмещать как абстрактные, так и неабстрактные методы в абстрактном классе.

## Это не глупые вопросы

**В: Какое предназначение у абстрактного метода? Я думал, что основная его цель — иметь общий код, который может быть унаследован дочерними классами.**

О: Реализации наследуемых методов (другими словами, методы *с телом*) можно и нужно помещать в родительские классы, *когда это имеет смысл*. В абстрактном классе это действие часто *лишено* смысла, потому что нельзя придумать общий код, который окажется полезен для подклассов. Смысл абстрактного метода в том, что хоть вы и не написали код для метода, вы все же определили часть протокола для группы дочерних типов (классов).

**В: И это хорошо, потому что...**

О: Полиморфизм! Не забывайте, что вам нужна способность использовать тип родительского класса (часто абстрактного) в качестве аргумента метода, типа возвращаемого значения или массива. Таким образом, вы можете добавлять новые подтипы (как новый подкласс Animal) в свою программу, не переписывая (или добавляя) для них новые методы. Представьте, как бы вы изменили класс Vet, если бы он не использовал Animal в качестве типа аргумента для методов. Вам бы пришлось создавать отдельный метод для каждого подкласса Animal! Один, принимающий подкласс Lion, другой, принимающий Wolf, еще один, принимающий... ну, вы поняли. Поэтому с абстрактными методами вы объявляете: «Все подтипы данного типа обладают этим методом».

# Вы обязаны реализовывать все абстрактные методы

**Реализация абстрактного метода происходит так же, как обычное переопределение.**

Абстрактные методы не имеют тела; они существуют только для полиморфизма. Это значит, что первый конкретный класс в дереве наследования должен реализовать все абстрактные методы.

Однако вы можете переложить ответственность на кого-то еще, если сами абстрактны. Например, если оба класса Animal и Canine абстрактны и содержат абстрактные методы, то класс Canine не обязан реализовывать методы класса Animal. Но как только мы доберемся до первого конкретного подкласса, например Dog, этот подкласс должен будет реализовать *все* абстрактные методы из обоих классов Animal и Canine.

Однако не забывайте, что абстрактный класс может включать как абстрактные, так и *не*-абстрактные методы, поэтому, например, Canine мог бы реализовать абстрактные методы Animal, чтобы этого не пришлось делать подклассу Dog. Но если Canine не скажет ничего про абстрактные методы Animal, то Dog должен будет реализовать их все.

Когда мы говорим: «Вы должны реализовать абстрактный метод», это означает, что вы *должны дать методу тело*. Иными словами, нужно создать неабстрактный метод в классе с такой же сигнатурой метода (именем и аргументами) и возвращаемым типом, совместимым с объявленным возвращаемым типом абстрактного метода. Что вы поместите *в* этот метод, зависит от вас. Java волнует только то, что метод находится *там*, в вашем конкретном дочернем классе.

## Абстрактные классы против Конкретных

Найдем этим абстрактным словам конкретное применение. В центральном столбце мы поместили несколько классов. Ваша задача — придумать такие приложения, где эти классы могли бы стать конкретными, а также приложения, где они могли бы стать абстрактными. Мы уже привели пару примеров, чтобы вам было легче начать. Например, класс Tree был бы абстрактным в программе лесного питомника, где разница между Oak и Aspen имеет значение. Однако в симуляторе гольфа Tree может быть конкретным классом (возможно, даже наследником класса Obstacle), потому что в этой программе не важно, чем отличаются разные типы деревьев.

Здесь нет правильных ответов; все зависит от дизайна вашей программы.

| Конкретные | Примеры классов | Абстрактные |
|---|---|---|
| Моделирование площадки для игры в гольф | Дерево | Приложение для лесного питомника |
| _____ | Дом | Архитектурное приложение |
| Приложение для спутниковой фотографии | Город | _____ |
| _____ | Футболист | Приложение для тренировки |
| _____ | Стул | _____ |
| _____ | Клиент | _____ |
| _____ | Заказ на закупку | _____ |
| _____ | Книга | _____ |
| _____ | Магазин | _____ |
| _____ | Поставщик | _____ |
| _____ | Гольф-клуб | _____ |
| _____ | Карбюратор | _____ |
| _____ | Духовка | _____ |

# Полиморфизм в действии

А если мы хотим написать *свой* класс списка, который будет содержать объекты Dog, но предположительно ничего не знаем о классе ArrayList? На первом этапе мы создадим для него метод *add()*. Мы используем простой массив Dog (Dog[]) для хранения добавленных объектов Dog и присвоим ему длину 5. Добавив пять объектов, мы по-прежнему сможем вызывать метод *add()*, но он не будет ничего делать. Если предел еще не достигнут, то метод *add()* поместит Dog в следующую доступную ячейку массива, а затем увеличит значение следующего доступного индекса (nextIndex).

## Создаем свой список для объектов Dog.

**Возможно, это худший в мире способ создания с нуля собственного класса вроде ArrayList.**

```
public class MyDogList {

    private Dog [] dogs = new Dog[5];

    private int nextIndex = 0;


    public void add(Dog d) {

       if (nextIndex < dogs.length) {

          dogs[nextIndex] = d;

          System.out.println("Dog добавлен в ячейку " + nextIndex)

          nextIndex++;

       }

    }

}
```

Внутри используем обычный массив объектов Dog.

Мы будем увеличивать его при каждом добавлении объекта Dog.

Если мы еще не дошли до предела массива, то добавляем объект Dog и выводим сообщение.

Увеличиваем индекс для доступа к следующей ячейке.

# Эх! Теперь нам нужно хранить еще и объекты типа Cat.

Здесь есть три варианта.

1. Создаем отдельный класс MyCatList для хранения объектов Cat. Это не очень грамотно.
2. Создаем один класс DogAndCatList, который будет содержать два разных массива в виде переменных экземпляра и иметь два разных метода add(): addCat(Cat c) и addDog(Dog d). Еще одно неудачное решение.
3. Создаем разнотипный класс AnimalList, который будет принимать *любой* тип подкласса Animal (если нужно менять спецификацию при добавлении объектов Cat, то рано или поздно придется добавлять *другое* животное). Нам больше всего нравится такой вариант, поэтому изменим наш класс и сделаем его более универсальным, чтобы он мог принимать объекты Animal, а не только Dog. Мы выделили ключевые изменения (логика осталась той же, однако тип изменился с Dog на Animal во всем коде).

## Создание собственного списка для объектов Animal.

Версия 2

```
public class MyAnimalList {

    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public void add(Animal a) {
       if (nextIndex < animals.length) {
          animals[nextIndex] = a;
          System.out.println("Animal добавлен в ячейку" + nextIndex);
          nextIndex++;

       }
    }
}
```

Без паники. Мы создаем не новый объект Animal, а новый объект массива для типа Animal. Помните, что нельзя создавать новые экземпляры абстрактного типа, но можно создавать массивы для хранения данного типа.

---

```
public class AnimalTestDrive{
   public static void main (String[] args) {
      MyAnimalList list = new MyAnimalList();
      Dog a = new Dog();
      Cat c = new Cat();
      list.add(a);
      list.add(c);
   }
}
```

```
File Edit Window Help Harm
% java AnimalTestDrive
Animal добавлен в ячейку 0
Animal добавлен в ячейку 1
```

# А что с объектами, которые не относятся к Animal? Почему бы не сделать класс настолько универсальным, чтобы он мог принимать любые объекты?

Мы хотим изменить тип массива вместе с аргументом метода *add()* на нечто, стоящее *выше*, чем Animal. Что-то *более* универсальное, *более* абстрактное, чем Animal. Но как нам устроить это? У нас *нет* родительского класса для Animal.

Хотя, с другой стороны, может нам сделать...

Помните методы класса ArrayList? Посмотрите, как методы remove, contains и indexOf используют объект типа... ***Object!***

**Каждый класс в языке Java — наследник класса Object.**

Класс Object — прародитель всех классов; это родительский класс для *всего*.

Даже если вы используете полиморфизм, нужно создавать класс с методами, которые будут принимать и возвращать *ваш* полиморфический тип. Без общего родительского класса для всего, что есть в языке Java, разработчики не имели бы возможности создавать классы с методами, принимающими ваши пользовательские типы — *типы, о которых они и не знали, когда писали класс ArrayList.*

Раньше вы создавали подклассы класса Object с самого начала и даже не знали об этом. ***Каждый ваш класс расширяет Object*** и даже не спрашивает вас об этом. Но вы можете считать, что класс выглядит так:

```
public class Dog extends Object { }
```

Погодите, Dog *уже* кое-что расширяет — класс *Canine*.

Это нормально. Компилятор сделает так, что *Canine* будет расширять Object. Кроме того, класс *Canine* унаследован от Animal. Нет проблем, тогда компилятор просто сделает так, что *Animal* будет расширять класс Object.

**Любой класс, который *явно* не расширяет другой класс, на самом деле *неявно* унаследован от Object.**

По этой причине, поскольку Dog — потомок класса Canine, он не будет *напрямую* расширять Object (хотя он делает это косвенно). Это же верно для Canine, а вот Animal *будет* унаследован от Object напрямую.

версия 3

Это всего лишь несколько методов ArrayList. На самом деле их гораздо больше.

**ArrayList**

**boolean remove(Object elem)**
Удаляет объект, соответствующий индексу в переданном параметре. Возвращает true, если элемент был в списке.

**boolean contains(Object elem)**
Возвращает true, если для объекта, переданного в параметре, нашлось совпадение.

**boolean isEmpty()**
Возвращает true, если в списке нет элементов.

**int indexOf(Object elem)**
Возвращает либо индекс переданного объекта, либо –1.

**Object get(int index)**
Возвращает элемент на указанной позиции в списке.

**boolean add(Object elem)**
Добавляет элемент в список (возвращает true).

**// дальше...**

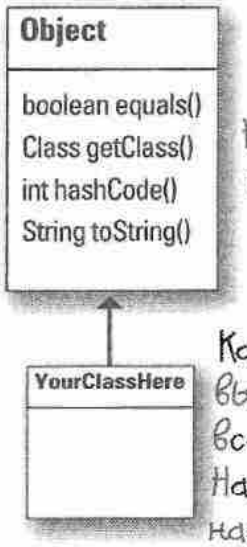
Множество методов класса ArrayList используют основной полиморфический тип Object. Поскольку каждый класс в языке Java — потомок класса Object, эти методы могут принимать все что угодно!

Примечание: начиная с Java 5.0, методы get() и add() немного отличаются от тех, что здесь продемонстрированы, но пока это все, что мы можем вам сообщить. Мы подробнее рассмотрим это немного позже.

# Что же находится в этом ультра-супер-мегаклассе Object?

Будучи языком Java, какое поведение вы бы предпочли для *каждого* своего объекта? Хм, посмотрим... Как насчет метода, который позволит проверять объекты на идентичность? А что насчет метода, который сможет сообщать действительный тип класса объекта? Может быть, метод, который даст хэш-код для объекта, чтобы использовать его в хэш-таблицах (мы поговорим о них в главе 17 и Приложении Б). О, вот то, что нужно, — метод, который выведет на экран строковое сообщение для объекта.

И что вы думаете? Словно по волшебству, класс Object действительно содержит такие методы. И хотя это не все его методы, они сейчас нам наиболее интересны.




Некоторые методы класса Object.

Каждый класс, который вы создаете, наследует все методы класса Object. Написанные вами классы наследуют методы, о которых вы даже не знали.

### 1 equals(Object o)

```
Dog a = new Dog();
Cat c = new Cat();

if (a.equals(c)) {
    System.out.println("true");
} else {
    System.out.println("false");
}
```

```
File Edit Window Help Stop
% java TestObject
false
```

Сообщает вам, считаются ли два объекта идентичными (что означает идентичность, мы расскажем в Приложении Б).

### 2 getClass()

```
Cat c = new Cat();
System.out.println(c.getClass());
```

```
File Edit Window Help Faint
% java TestObject
class Cat
```

Возвращает класс, экземпляром которого является объект.

### 3 hashCode()

```
Cat c = new Cat();
System.out.println(c.hashCode());
```

```
File Edit Window Help Drop
% java TestObject
8202111
```

Выводит на экран хэш-код (думайте о нем, как об уникальном идентификаторе (ID)).

### 4 toString()

```
Cat c = new Cat();
System.out.println(c.toString());
```

```
File Edit Window Help LapseIntoComa
% java TestObject
Cat@7d277f
```

Выводит сообщение с именем класса и числами, на которые обычно можно не обращать внимания.

**В: Класс Object абстрактный?**

О: Нет. По крайней мере в понятии языка Java. Object — это неабстрактный класс, потому что у него есть код реализации методов, которые могут наследоваться и использоваться всеми классами, без переопределения данных методов.

**В: А *можно* ли переопределять эти методы в классе Object?**

О: Некоторые можно. Однако отдельные методы отмечены ключевым словом `final`, а это означает, что их нельзя заменить. Мы настоятельно рекомендуем вам переопределять в своих классах методы hashCode(), equals() и toString(), и немного позже вы узнаете, как это делать. Но методы наподобие getClass() выполняют такие вещи, которые гарантированно должны работать именно так, как задумано.

**В: Если методы класса ArrayList достаточно универсальны для применения Object, то что будет означать запись ArrayList<DotCom>? Мне кажется, что так я ограничиваю ArrayList в использовании для хранения лишь объектов DotCom.**

О: Вы действительно ограничиваете его. До выхода Java 5.0 ArrayList нельзя было ограничить. Результат, получаемый при записи ArrayList<Object>, имеет очень большое значение в Java 5.0. Другими словами, ***ArrayList будет ограничен только для Object***, а это *любой* объект в Java — экземпляр *любого* типа класса! Позже мы рассмотрим подробности нового синтаксиса <тип>.

**В: Если так полезно использовать полиморфические типы, почему бы вам не сделать так, чтобы все ваши методы принимали и возвращали тип Object?**

О: Подумайте, к чему это приведет. Прежде всего вы разрушите весь смысл «безопасности типов» — один из наиболее мощных механизмов защиты кода в языке Java. Используя правила безопасности типов, Java следит, чтобы вы не сделали с каким-нибудь объектом то, что *хотели* сделать с объектом другого типа. Например, не попросили бы объект *Ferrari* (думая, что это *Toaster*) поджариться.

Вам *не* нужно беспокоиться о таком развитии событий, как с огненным Ferrari, даже если вы *действительно* используете для всего ссылки Object. Когда на объекты ссылаются с помощью ссылок типа Object, Java *считает*, что они указывают на экземпляр типа Object. А это означает, что единственные разрешенные для вызова методы данных объектов — те, что объявлены в классе Object! Если вы напишете:

```
Object o = new Ferrari();
o.goFast(); //Недопустимо!
```

компилятор не сможет это пропустить.

Поскольку Java — язык со строгим контролем типов, компилятор проверит, что вы вызываете метод объекта, который действительно способен *ответить*. Другими словами, вы можете вызвать метод, принадлежащий ссылке объекта, *только* если класс ссылочного типа на самом деле *содержит* этот метод. Мы рассмотрим это подробнее немного позже, поэтому не волнуйтесь, если еще не все понятно.

## Это не глупые вопросы

**В: Хорошо, вернемся к неабстрактности класса Object (по-моему, это означает, что он конкретный). Как вы можете позволить кому-то создать объект класса Object? Не будет ли это таким же странным, как создание объекта Animal?**

О: Хороший вопрос! Почему создание нового экземпляра Object считается приемлемым? Потому что иногда хочется использовать общий объект как… объект. *Небольшой* объект. Пока экземпляр типа Object наиболее часто применяют для синхронизации потоков (что вы узнаете в главе 15). На данный момент просто имейте это в виду и считайте, что редко будете создавать объекты типа Object, даже если и *можете* это делать.

**В: Получается, главный смысл типа Object состоит в том, что его можно использовать в качестве полиморфического аргумента и типа возвращаемого значения? Как в ArrayList?**

О: Класс Object служит для *двух* целей. Во-первых, он должен действовать как полиморфический тип для методов, которым нужно работать с любым классом, созданным вами или кем-то еще. Во-вторых, он обязан обеспечивать *реальный* код методов, необходимых всем объектам при выполнении программы (а добавление их в класс Object означает, что они наследуются всеми классами). Некоторые наиболее важные методы в Object относятся к потокам, и мы рассмотрим их позже.

# Использование полиморфических ссылок типа Object имеет свою цену...

Пока вы не забежали вперед и не начали использовать тип Object для всех своих ультрагибких аргументов и возвращаемых типов, вам следует узнать кое-что о применении типа Object в качестве ссылки. И имейте в виду, что мы говорим не о создании экземпляров типа Object, а о создании экземпляров других типов, но с использованием ссылки типа Object.

Когда вы добавляете объект в ArrayList<**Dog**>, то он помещается как Dog и выходит оттуда как Dog.

```
ArrayList<Dog> myDogArrayList = new ArrayList<Dog>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
Dog d = myDogArrayList.get(0);
```

Создаем ArrayList и объявляем его для хранения объектов Dog.

Создаем объект класса Dog.

Добавляем Dog в список.

Присваиваем объект Dog из списка новой переменной, ссылка на которую имеет тип Dog. Думайте об этом так, будто для метода get() объявлен тип возвращаемого значения Dog, потому что использовался ArrayList<Dog>.

Но что случится, когда вы объявите его как ArrayList<**Object**>? Если вы хотите, чтобы ArrayList принимал *любой* тип Object, нужно объявить его так:

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
```

Создаем ArrayList и объявляем его для хранения объектов типа Object.

Создаем объект класса Dog.

Добавляем Dog в список.

Эти два шага остаются прежними.

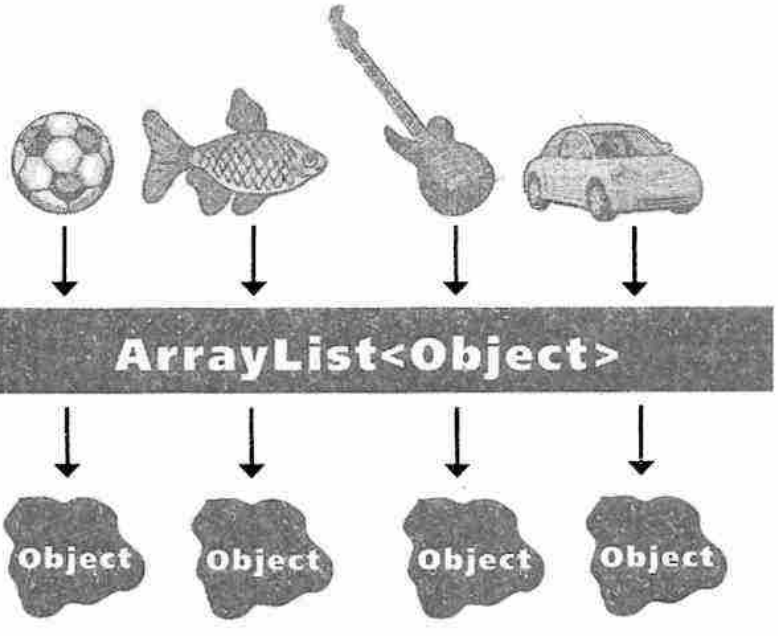
Но что произойдет, когда вы попытаетесь получить объект Dog и присвоить его ссылке класса Dog?

```
Dog d = myDogArrayList.get(0);
```

Нет! Это не скомпилируется! Когда вы используете ArrayList<Object>, метод get() возвращает тип Object. Компилятор знает только, что объект наследует класс Object (где-то в его иерархии наследования), но он не знает, что это Dog!

***Каждый объект, выходящий из ArrayList<Object>, представляет собой ссылку типа Object, независимо от назначения объекта или ссылочного типа в тот момент, когда вы добавляли его в список.***

**Объекты помещаются внутрь как SoccerBall, Fish, Guitar и Car.**




**Но выходят они так, будто были типом Object.**

**Объекты выходят из ArrayList<Object> так, будто они — универсальные экземпляры класса Object. Компилятор не допустит, чтобы полученный объект имел тип, отличающийся от типа Object.**

# Когда Dog ведет себя не как Dog

Проблема полиморфического использования всего в качестве Object заключается в том, что объекты *теряют* (но не навсегда) свою истинную сущность. *Dog теряет свои качества.* Посмотрим, что будет, если мы передадим Dog методу, который возвращает ссылку тому же объекту Dog, но при этом объявляет возвращаемый тип как Object, а не Dog.

Плохо ☹

```
public void go() {
    Dog aDog = new Dog();
    Dog sameDog = getObject(aDog);
}

public Object getObject(Object o) {
    return o;
}
```

Эта строка не сработает! Возвращаемый тип Object означает, что компилятор не позволит вам присвоить возвращенную ссылку чему-нибудь, кроме Object, несмотря на то что метод выдает ссылку на тот же объект Dog, на который ссылается аргумент.

Мы выдаем ссылку на тот же объект Dog, но как возвращаемый тип Object. Эта часть кода абсолютно допустима. Примечание: это похоже на работу метода get(), когда указано ArrayList<Object>, а не ArrayList<Dog>.

```
File Edit Window Help Remember
DogPolyTest.java:10: incompatible types
found   : java.lang.Object
required: Dog
      Dog sameDog = takeObjects(aDog);
                                ^
1 error
```

Компилятор не знает, что элемент, возвращенный из метода, — это фактически объект Dog, поэтому он не позволит вам присвоить его ссылке с типом Dog. На следующей странице вы узнаете, почему.

---

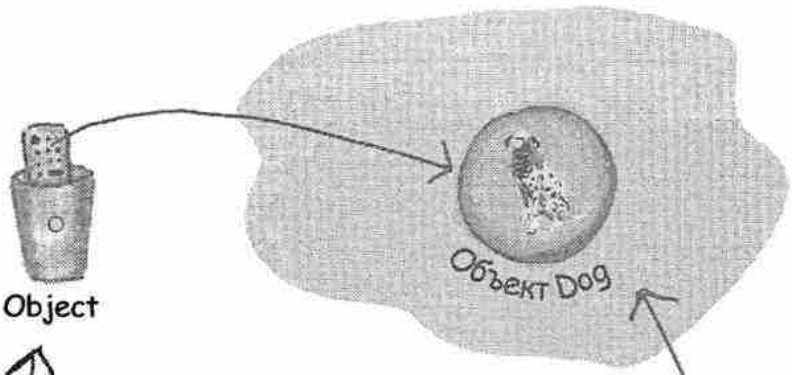
Хорошо ☺

```
public void go() {
    Dog aDog = new Dog();
    Object sameDog = getObject(aDog);
}

public Object getObject(Object o) {
    return o;
}
```

Это работает (хотя может оказаться бесполезным, что вы скоро увидите), так как можно присвоить ссылке типа Object все что угодно, ведь все классы проходят проверку на соответствие ему. Как вы помните, каждый объект в Java — это экземпляр типа Object, и на вершине иерархии наследования каждого класса в Java стоит именно этот тип.

# Объекты не лают

Итак, теперь вы знаете, что, когда на объект ссылается переменная типа Object, он не может быть присвоен переменной своего фактического типа. Вы также знаете, что возвращаемый тип или аргумент может принадлежать типу Object, например, когда объект помещен в ArrayList типа Object с помощью ArrayList<Object>. Но какой в этом смысл? Неужели сложно использовать переменную ссылочного типа Object в качестве ссылки на объект Dog? Попытаемся вызвать методы нашего объекта Dog, который воспринимается компилятором как Object:




Когда вы получаете ссылку на объект из ArrayList<Object> (или любого метода, возвращающего значение типа Object), она выдается в виде полиморфической ссылки типа Object. Поэтому у вас будет ссылка типа Object на (в данном случае) экземпляр класса Dog.

```
Object o = al.get(index);
int i = o.hashCode();
o.bark();
```

Это допустимо. Класс Object содержит метод hashCode(), поэтому вы сможете вызвать его для любого объекта в Java.

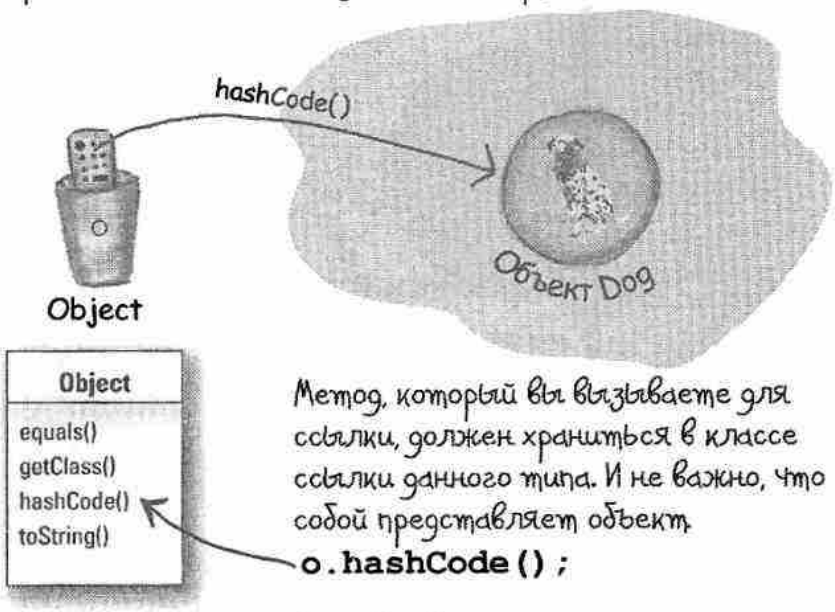
Не скомпилируется!

Так делать нельзя! Класс Object ничего не знает о методе bark() (bark — «лаять»). Даже если вы знаете, что по данному индексу действительно расположен объект Dog, компилятору это неизвестно.

**Компилятор решает, сможете ли вы вызвать метод, основываясь на типе ссылки, а не на фактическом типе объекта.**

Даже если вы *знаете*, что объект способен на такое («...это Dog, честное слово...»), компилятор воспринимает его лишь как общий объект Object. Компилятору известно, что объект может быть кнопкой. Или объектом Microwave. Или другим объектом, который действительно не знает, как лаять.

Компилятор проверяет класс *ссылочного*, а не *объектного* типа, чтобы убедиться в том, что вы можете вызывать метод с помощью этой ссылки.







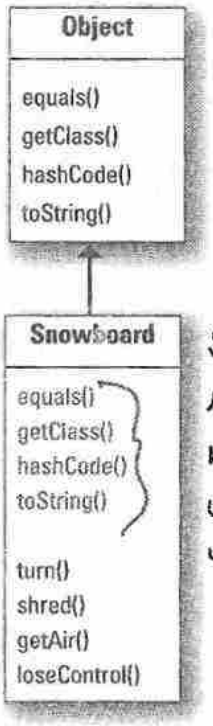
Метод, который вы вызываете для ссылки, должен храниться в классе ссылки данного типа. И не важно, что собой представляет объект

```
o.hashCode();
```

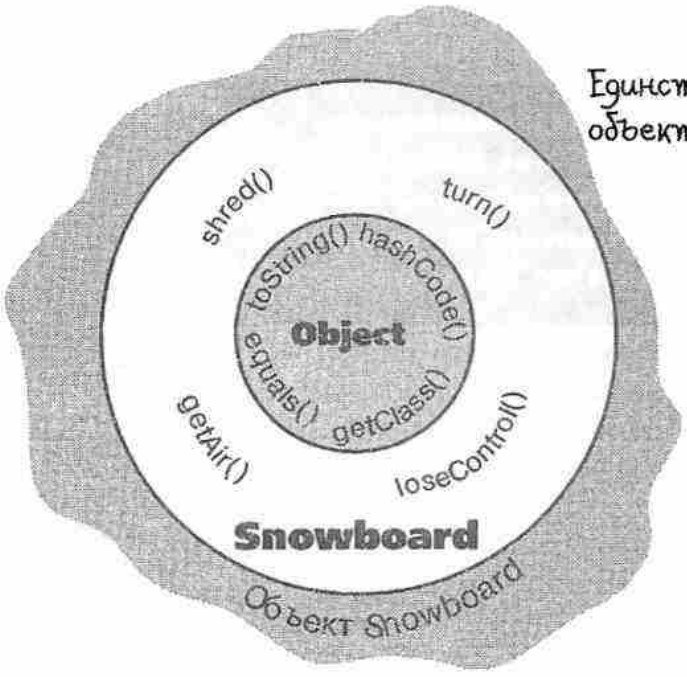
При объявлении ссылка «o» получила тип Object, поэтому вы можете вызывать методы, если они находятся в классе Object.

## Установите связь со своим внутренним Object.

Объект содержит *все*, что наследует от каждого родительского класса. Это означает, что *любой* объект, независимо от его фактического класса, *также* представляет собой экземпляр класса Object. Это, в свою очередь, говорит о том, что к любому объекту в языке Java можно относиться не только как к Dog, Button или Snowboard, но и как к Object. Когда вы говорите **`new Snowboard()`**, то получаете один объект Snowboard в куче, но он заключает в себе внутреннее ядро, представляющее часть самого Object.




Snowboard наследует методы родительского класса Object и добавляет еще четыре.




В куче только один объект — Snowboard. Но он содержит части обоих классов Snowboard и Object.

## Полиморфизм означает «много форм».

### Вы можете относиться к Snowboard как к Snowboard или как к Object.

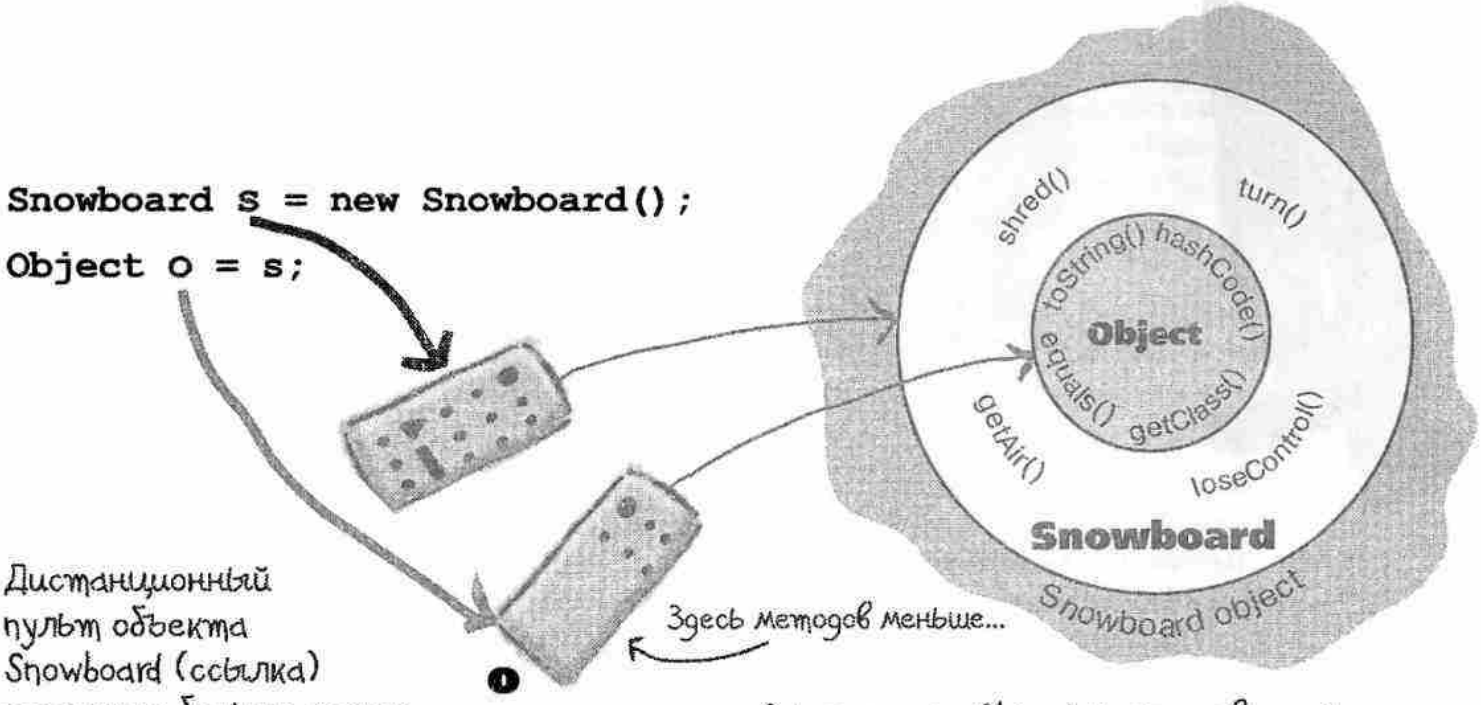
Ссылка — это что-то вроде пульта дистанционного управления, и он принимает все больше и больше кнопок, когда вы двигаетесь вниз по иерархии наследования. Пульт (ссылка) типа Object имеет несколько кнопок — для открытых методов Object. Однако пульт типа Snowboard включает в себя все кнопки класса Object, а также любые новые кнопки (для новых методов) класса Snowboard. Чем больше у класса особенностей, тем больше у него может быть кнопок.

Конечно, дочерний класс может и не добавлять новых методов, а просто заменять методы класса-родителя. Даже если *объект* принадлежит типу Snowboard, *ссылка* Object на такой объект не сможет увидеть методы для Snowboard.

**Поместив объект в ArrayList<Object>, вы можете относиться к нему только как к Object, независимо от его изначального типа.**

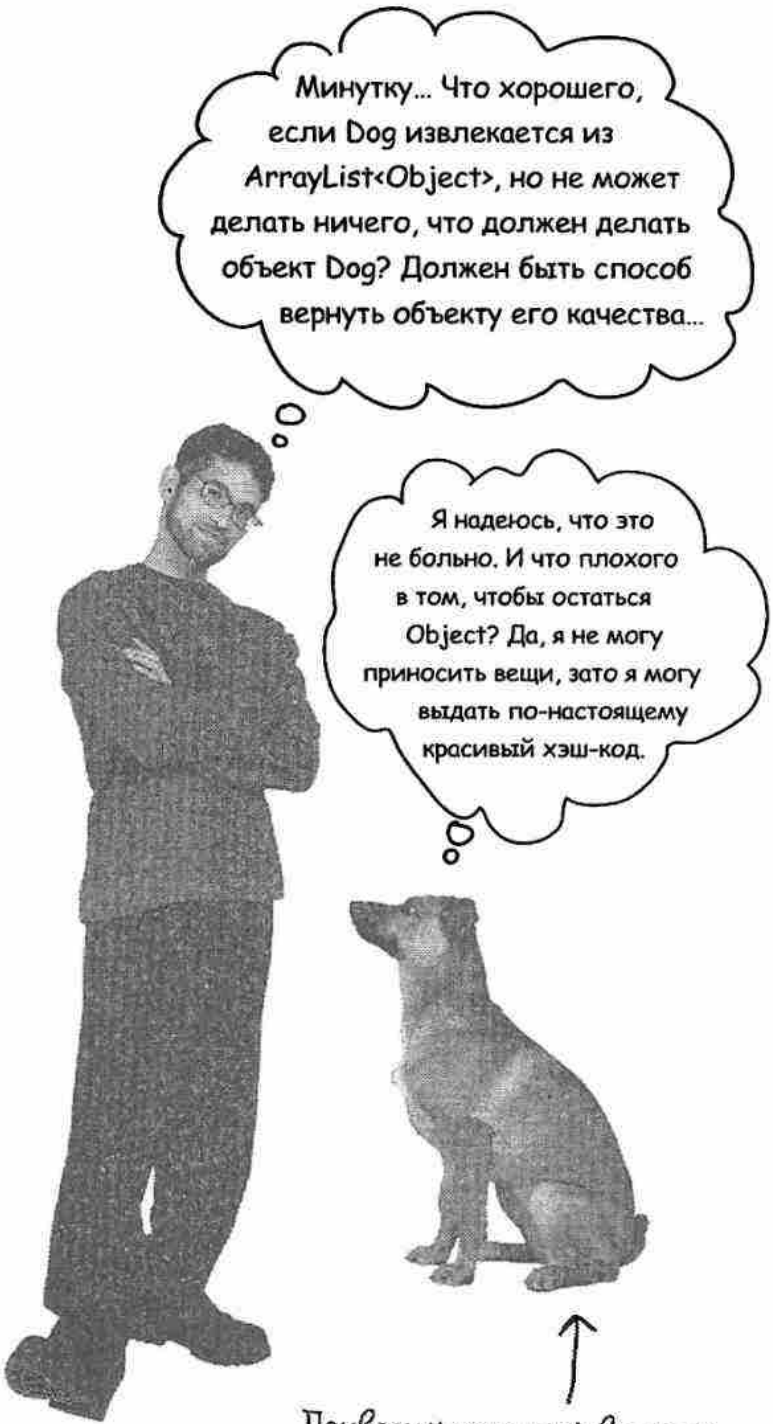
**Когда вы получите ссылку из ArrayList<Object>, она всегда будет иметь тип Object.**

**Это значит, что вы получите пульт от Object.**

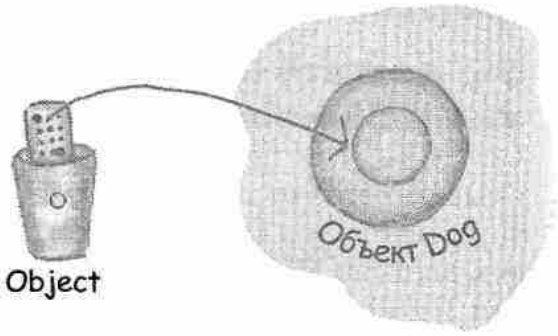



Дистанционный пульт объекта Snowboard (ссылка) содержит больше кнопок, чем пульт Object. Пульт Snowboard может распознать все, что относится к одноименному объекту. Он имеет доступ ко всем методам Snowboard, включая унаследованные методы обоих классов — Object и Snowboard.

Ссылка типа Object может увидеть только ту часть объекта Snowboard, которая принадлежит Object. Она имеет доступ лишь к методам класса Object. У нее меньше кнопок, чем у пульта Snowboard.



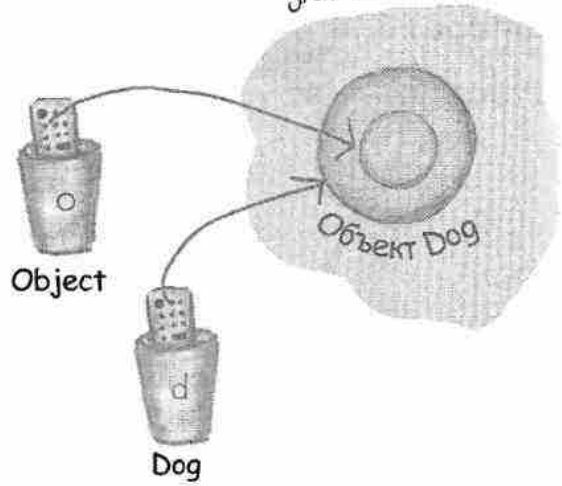

## Приведение ссылки на объект к ее *действительному* типу.




Это все еще *объект* Dog, но если вы хотите вызвать характерные для него методы, вам нужна *ссылка* с таким типом. Если вы *уверены**, что этот объект действительно Dog, то можете создать для него новую одноименную ссылку, скопировав ссылку Object и заставив копию переместиться в переменную ссылочного типа *Dog*, используя приведение (Dog). Вы сможете применять новую ссылку для вызова методов объекта *Dog*.

```
Object o = al.get(index);
Dog d = (Dog) o;
d.roam();
```

← Приводим Object к Dog, который, как мы знаем, им и является.



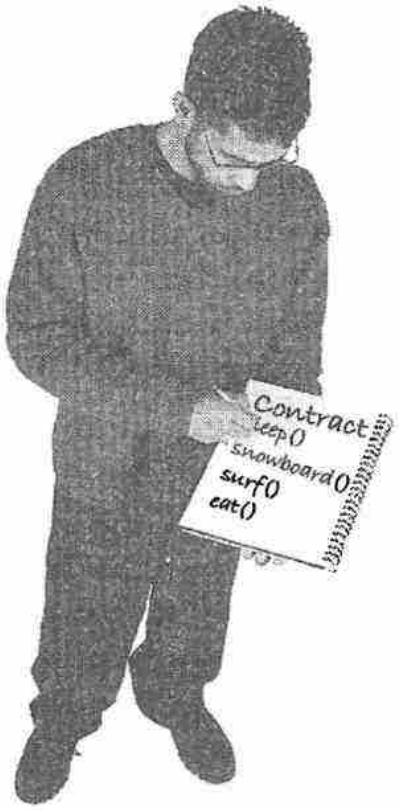

*Если вы *не* уверены, что это Dog, то можете вызвать для проверки оператор **instanceof**. Если вы ошибетесь, приводя типы, то получите ошибку ClassCastException, что вызовет преждевременное завершение программы.

```
if (o instanceof Dog) {
   Dog d = (Dog) o;
}
```

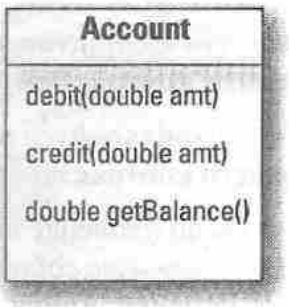
## Теперь вы знаете, как сильно Java заботится о методах в классе ссылочной переменной.

**Метод объекта можно вызвать, *только* когда он есть у класса ссылки.**

**Считайте публичные методы в классе своим контрактом, обещанием внешнему миру о том, что вы можете делать.**




Создавая класс, вы почти всегда открываете некоторые методы коду вне класса. *Открытие* метода означает, что вы делаете его *доступным*, обычно отмечая как public.




Представьте себе такой сценарий. Вы пишете код для небольшой бухгалтерской программы — заказное приложение для фирмы «Simon's Surf Shop». Вы находите класс Account, который идеально подходит, по крайней мере согласно его документации. Каждый экземпляр класса представляет собой индивидуальный счет клиента с магазином. И вот вы занимаетесь своим делом, вызываете методы *credit()* и *debit()* для объекта и понимаете, что вам нужно получить баланс по счету. Без проблем — есть метод *getBalance()*, который отлично подойдет для этого.

Только если вы вызовете метод *getBalance()*, это приведет к катастрофической ошибке. Забудьте о документации, на самом деле у класса нет такого метода. Ой!

Но этого не случится, потому что каждый раз, когда вы используете оператор «точка» для *ссылки* (a.doStuff()), компилятор смотрит на ссылочный тип (a был объявлен типом) и проверяет, чтобы у класса был этот метод и он действительно принимал передаваемый вами аргумент, а также возвращал значение того типа, который вы ожидаете.

**Просто запомните, что компилятор проверяет класс ссылки, а не класс фактического объекта на другом ее конце.**

## Что делать, если нужно изменить контракт

Хорошо, представьте, что вы — Dog. Ваш класс Dog не *единственный* контракт, определяющий, кто вы такой. Помните, что вы наследуете доступные (что обычно означает *public*) методы из всех родительских классов.

Действительно, ваш класс Dog определяет контракт.

Но не *все* ваши контракты.

**Все, что находится в классе *Canine*, — часть вашего контракта.**

**Все, что находится в классе *Animal*, — часть вашего контракта.**

**Все, что находится в классе *Object*, — часть вашего контракта.**

Согласно проверке на соответствие (IS-A), вы *представляете* собой каждый из этих объектов — Canine, Animal и Object.

Но если человек, разработавший ваш класс, думал о программе-симуляторе Animal и теперь хочет использовать вас (класс Dog) для учебной научной ярмарки объектов Animal?

Все в порядке, скорее всего, вы сможете повторно использоваться и для этого.

А если позже он захочет задействовать вас для программы PetShop (Магазин домашних животных)? *У вас нет навыков домашнего животного*. Объект Pet должен иметь методы вроде *beFriendly()* и *play()*.

Теперь представьте, что вы разрабатываете класс Dog. Просто добавьте к нему несколько методов. Вы не нарушите этим ничей код, так как не будете трогать *существующие* методы, которые могут быть вызваны другим кодом для объектов Dog.

Видите ли вы какие-нибудь препятствия для добавления методов Pet в класс Dog?

Подумайте, что бы *вы* сделали, если бы разрабатывали класс Dog и пришлось бы его изменить, чтобы он смог выполнять действия класса Pet. Если просто добавить новые свойства (методы) Pet к классу Dog, это будет работать и не нарушит другой код.

Но... это программа PetShop. У нее есть не только объекты Dog! А если кто-нибудь захочет использовать ваш класс Dog для программы, в которой предусмотрены *дикие* собаки? Какие у вас будут варианты? Не беспокоясь о том, как Java обрабатывает элементы, просто попробуйте представить, как можно решить проблему изменения некоторых классов-наследников Animal для добавления свойств Pet.

Подумайте об этом сейчас, **не заглядывая на следующую страницу**, где мы начнем открывать *все* секреты.

В ином случае вы сделаете упражнение абсолютно бесполезным, лишив себя большого шанса сжечь несколько калорий.

# Рассмотрим варианты повторного использования существующих классов в программе PetShop.

На данный момент не важно, сможет ли Java *сделать* то, что мы придумаем. Мы будем решать эту проблему, как только у нас появится наилучший выбор из нескольких вариантов.

## 1 Вариант первый

Выберем легкий путь и поместим методы для домашних животных в класс Animal.

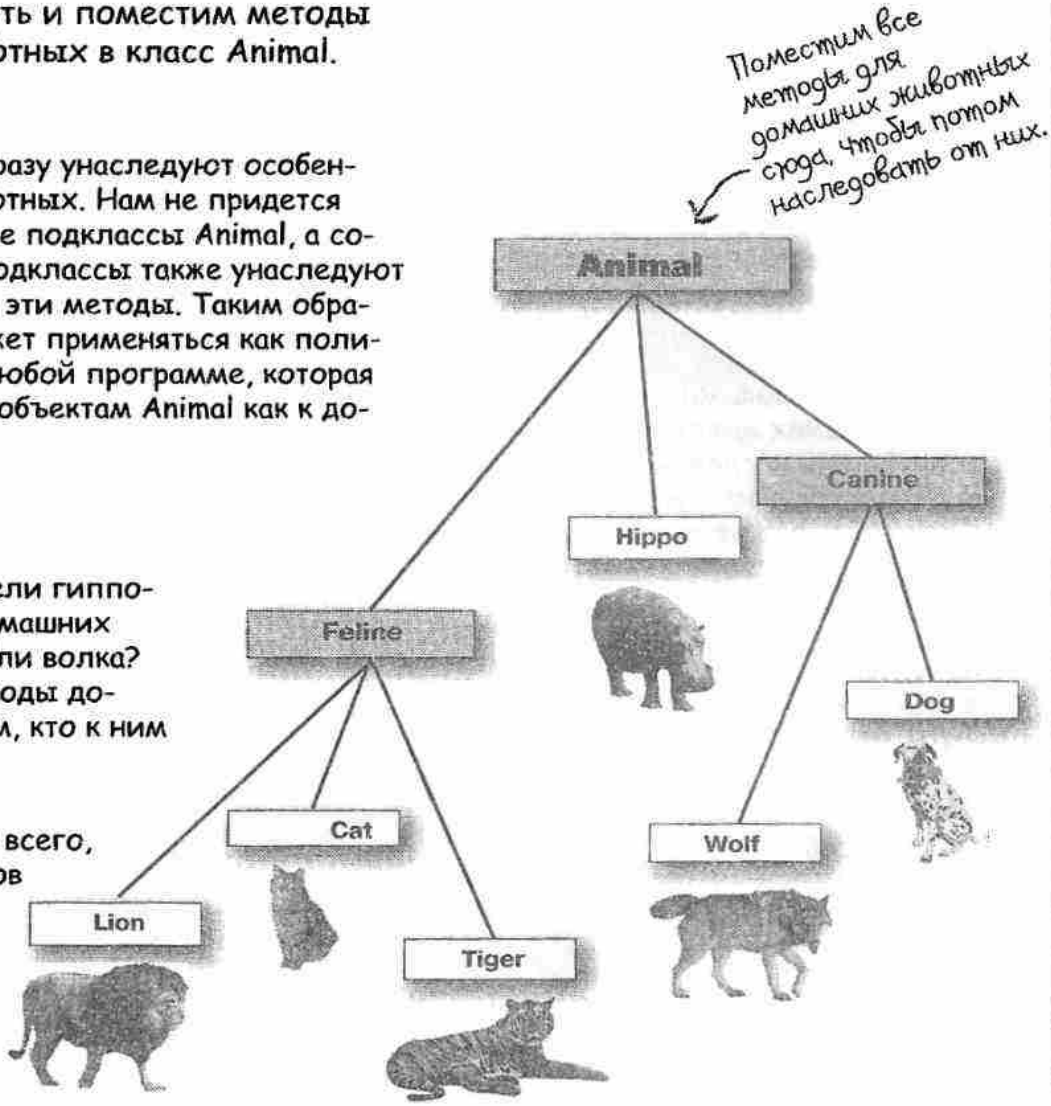
**За:**

Все объекты Animal сразу унаследуют особенности домашних животных. Нам не придется трогать существующие подклассы Animal, а созданные в будущем подклассы также унаследуют и будут использовать эти методы. Таким образом, класс Animal может применяться как полиморфический тип в любой программе, которая захочет относиться к объектам Animal как к домашним животным.

**Против:**

Вы когда-нибудь видели гиппопотама в магазине домашних животных? А льва? Или волка? Опасно раздавать методы домашних животных тем, кто к ним не относится.

К тому же мы, скорее всего, будем касаться классов для домашних животных, например Dog и Cat, так как (по крайней мере в наших домах) собакам и кошкам свойственно разное поведение.



## ② Вариант второй

Мы начнем с первого варианта, поместим методы для домашних животных в класс Animal, но сделаем их абстрактными, заставив наследников Animal переопределять эти методы.

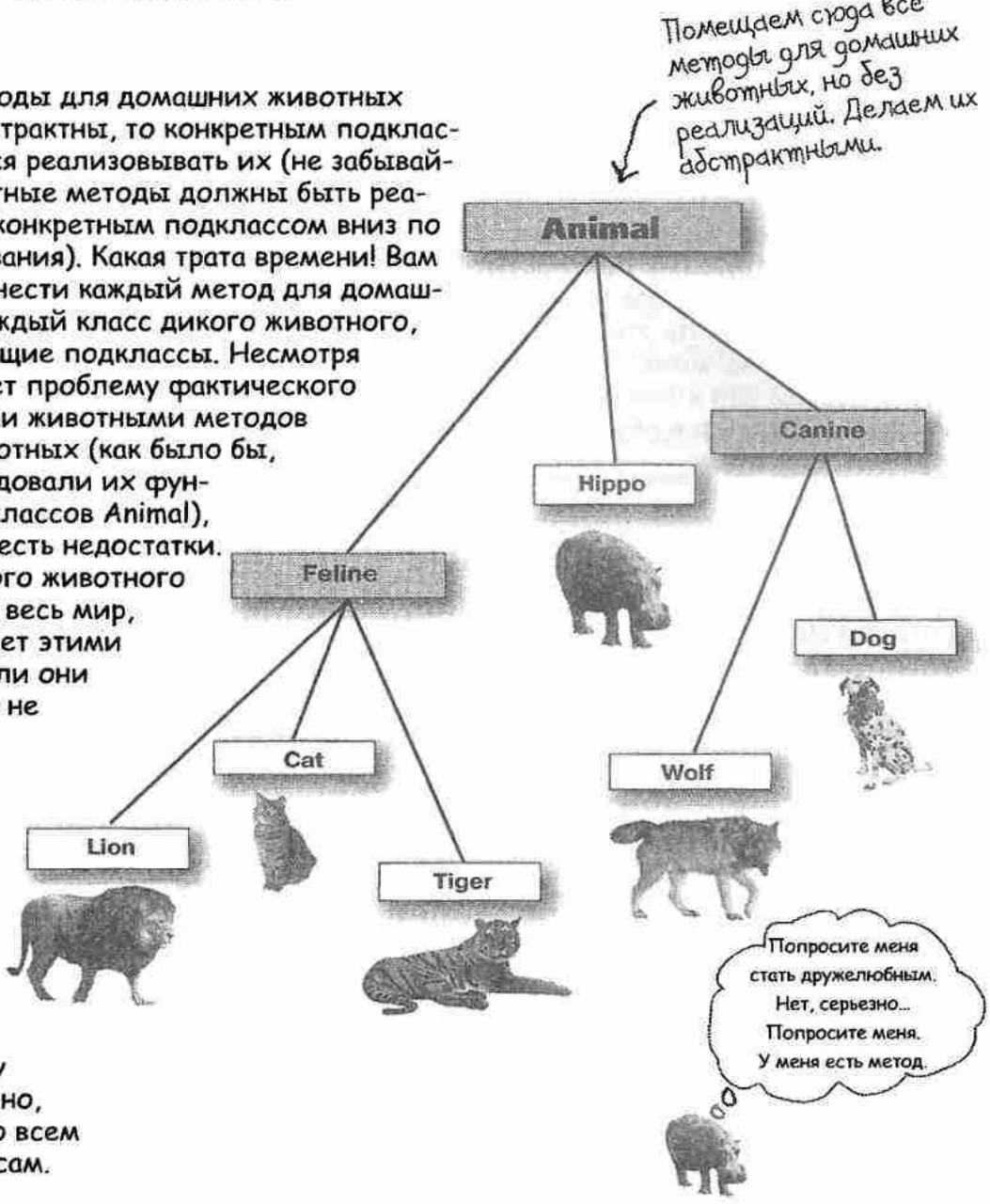
### За:

Это даст нам все преимущества первого варианта, но без негативных последствий, при которых дикие животные имели бы свойства домашних (например, beFriendly()). Все наследники Animal будут содержать эти методы (потому что они будут находиться в классе Animal). Однако, поскольку методы будут абстрактными, классы Pet не унаследуют их функции. Все классы должны будут заменять методы, но смогут сделать их такими, чтобы они «ничего не делали».

### Против:

Поскольку все методы для домашних животных в классе Animal абстрактны, то конкретным подклассам Animal придется реализовывать их (не забывайте, что все абстрактные методы должны быть реализованы первым конкретным подклассом вниз по иерархии наследования). Какая трата времени! Вам придется сесть и внести каждый метод для домашних животных в каждый класс дикого животного, а также во все будущие подклассы. Несмотря на то что это решает проблему фактического выполнения дикими животными методов для домашних животных (как было бы, если бы они наследовали их функциональность из классов Animal), в контракте все же есть недостатки. Каждый класс дикого животного будет объявлять на весь мир, что он тоже обладает этими методами, даже если они фактически ничего не будут делать при вызове.

Этот способ нам не подходит. Не нужно запихивать в класс Animal больше, чем требуется одному типу Animal, если, конечно, это не относится ко всем его дочерним классам.



### ③ Вариант третий

Помещаем методы для домашних животных только в те классы, для которых они подходят.

**За:**

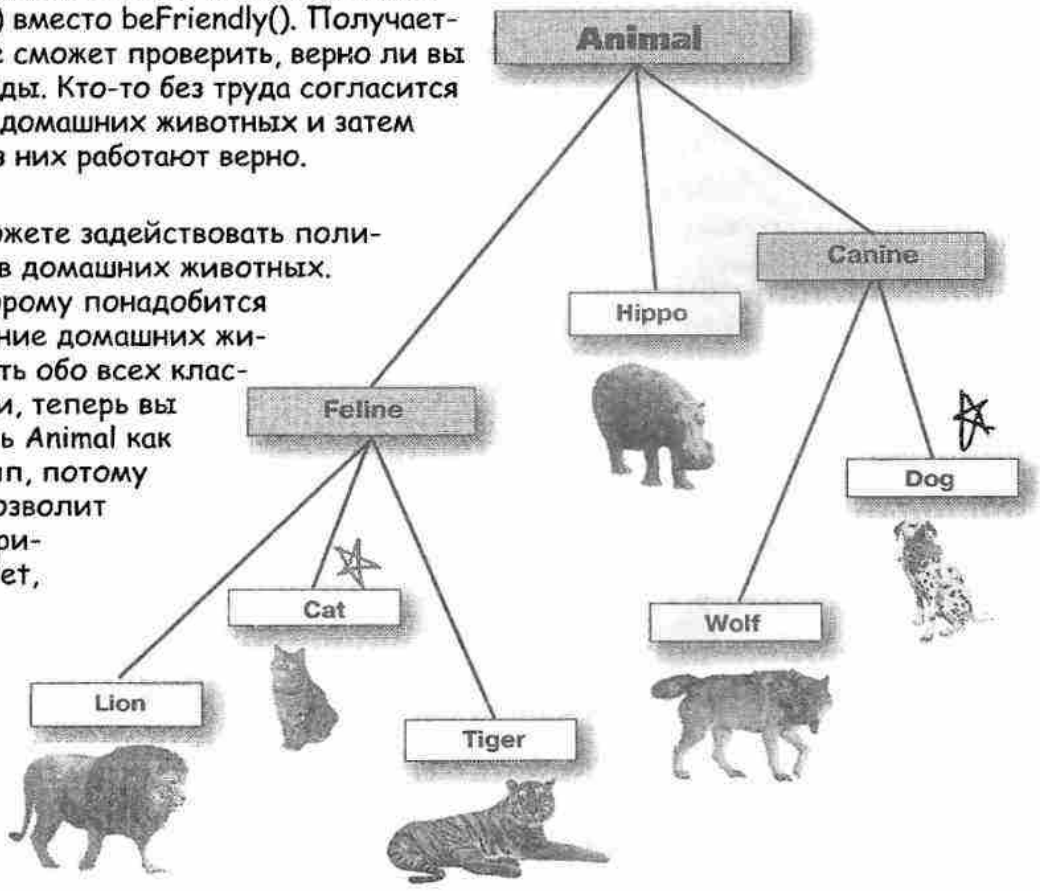
Больше не нужно переживать о гиппопотамах, встречающих вас у дверей и облизывающих вам лицо. Методы будут находиться там, где должны быть, и только там. Объекты Dog и Cat смогут реализовать эти методы, и никто больше не будет о них знать.

**Против:**

У этого способа есть две проблемы. Во-первых, вам нужно согласиться с протоколом, и все разработчики классов домашних животных с этого момента будут знать о нем. Под протоколом мы подразумеваем точные методы, которые, как мы решили, будут содержать все домашние животные, то есть это контракт для домашних животных без права его отмены. Но если кто-то из программистов неправильно поймет его? Например, метод будет принимать строку, тогда как должен принимать целое число. Или назовет метод doFriendly() вместо beFriendly(). Получается, что компилятор не сможет проверить, верно ли вы реализуете свои методы. Кто-то без труда согласится использовать классы домашних животных и затем выяснит, что не все из них работают верно.

Во-вторых, вы не сможете задействовать полиморфизм для методов домашних животных. Каждому классу, которому понадобится использовать поведение домашних животных, придется знать обо всех классах! Другими словами, теперь вы не сможете применять Animal как полиморфический тип, потому что компилятор не позволит вам вызвать метод, принадлежащий классу Pet, в контексте ссылки типа Animal (даже если он действительно является объектом Dog), ведь у класса Animal не будет этого метода.

★ Помещаем методы для домашних животных только в те дочерние классы Animal, которые могут быть домашними животными, вместо того чтобы помещать их в общий класс Animal.

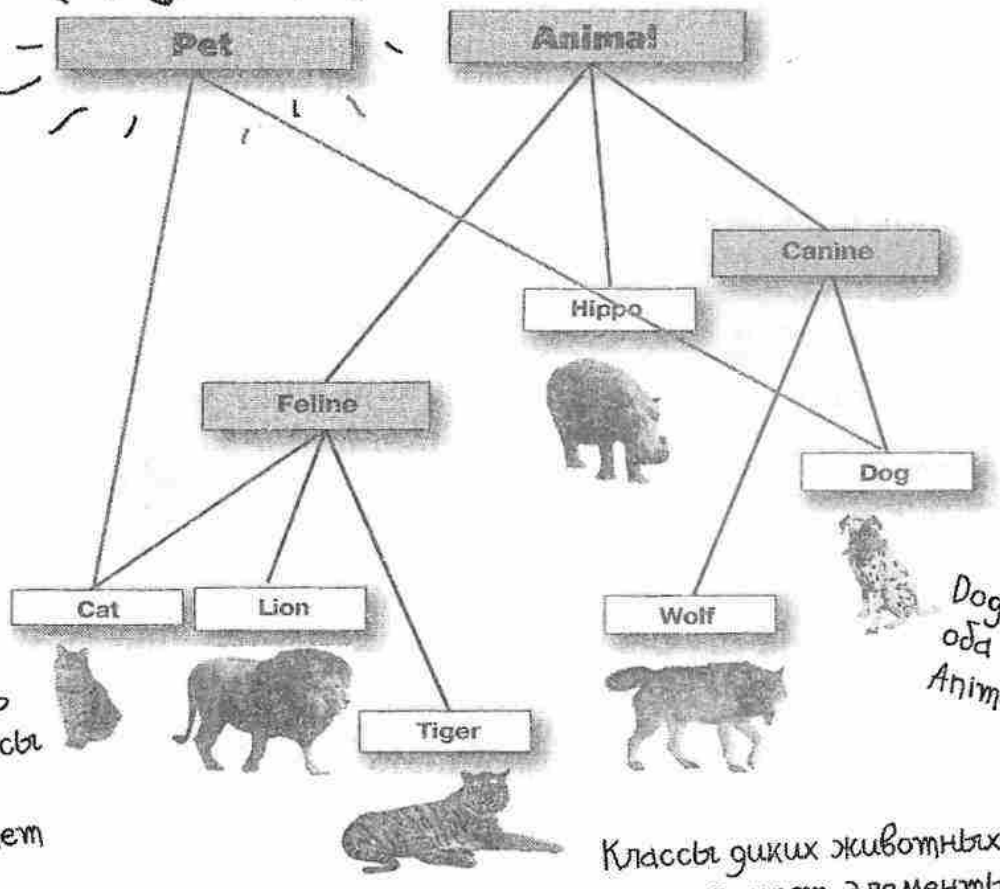


## Итак, что нам действительно необходимо.

- Способ, при котором свойства домашних животных будут описаны **только** в классах для домашних животных.
- Способ, гарантирующий, что для всех классов домашних животных будут определены одинаковые методы (одно имя, одни аргументы, одинаковые типы возвращаемых значений, не будет отсутствующих методов и т. д.). Это позволит надеяться, что все программисты поймут их так, как нужно.
- Способ, при котором можно получить пользу от полиморфизма, чтобы все домашние животные вызывали свои методы, без необходимости использовать для каждого класса аргументы, типы возвращаемых значений и массивы.

## Похоже, нам понадобятся два родительских класса на вершине иерархии.

Мы создаем новый абстрактный родительский класс Pet и даем ему все методы для домашних животных.




Класс Cat теперь расширяет классы Animal и Pet, поэтому получает методы обоих.

Dog расширяет оба класса: Animal и Pet.

Классы диких животных не наследуют элементы класса Pet.

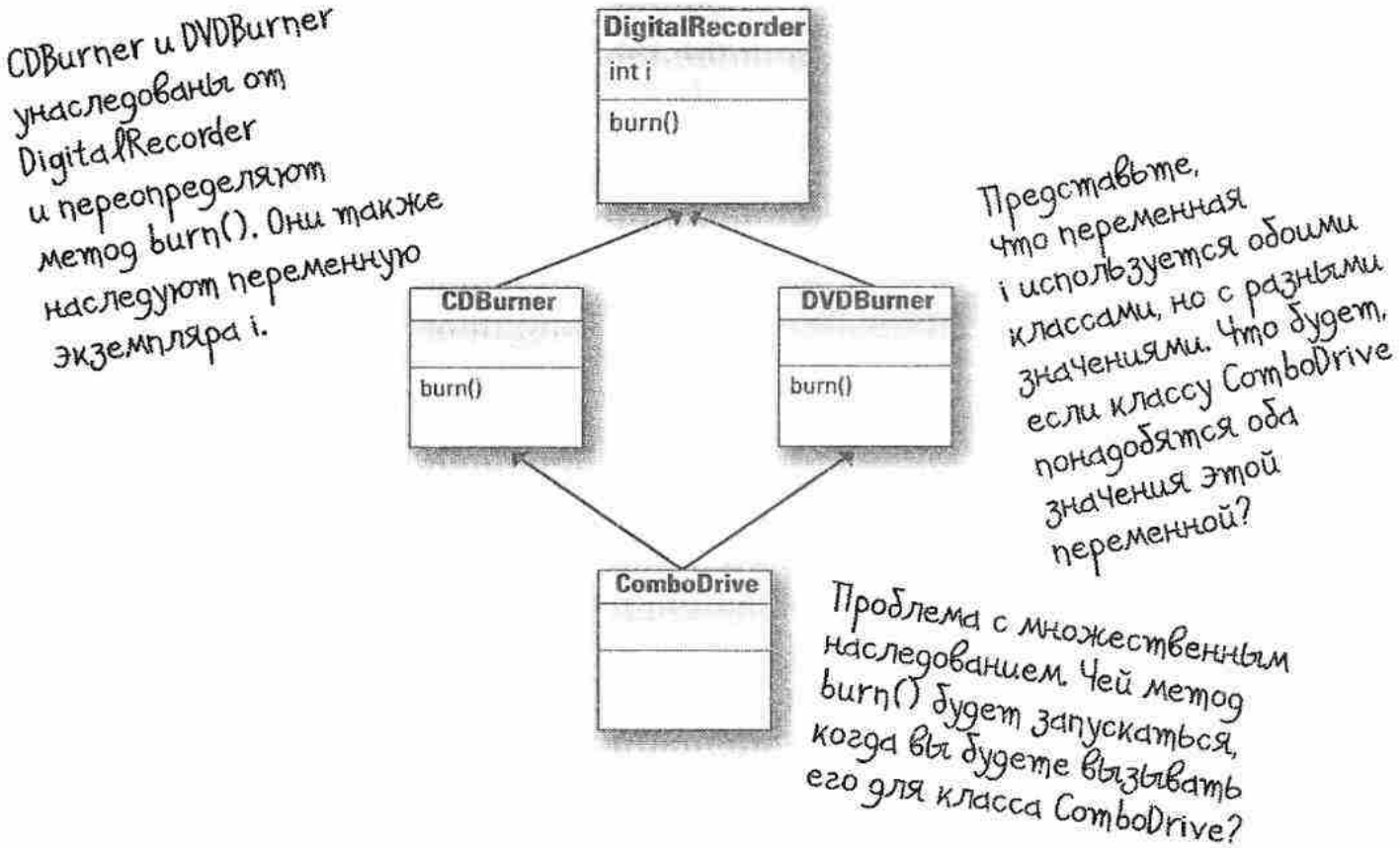
Только в таком способе «двух родительских классов» кроется одна проблема...

**Это называется множественным наследованием и может привести к действительно большим проблемам.**

Вернее, могло бы привести, если бы это понятие поддерживалось языком Java.

Но это не так, потому что у множественного наследования есть проблема, известная под названием «смертоносный ромб смерти» (Deadly Diamond of Death).

## «Смертоносный ромб смерти»



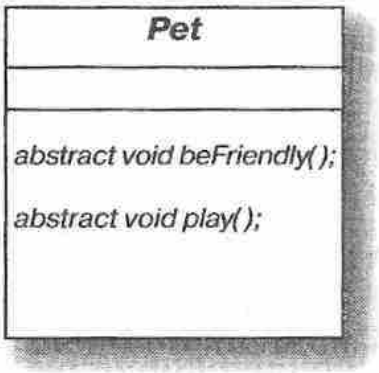

В языке, который позволяет реализовать «смертоносный ромб смерти», могут возникать запутанные ситуации, так как придется иметь специальные правила для разрешения возможных недоразумений. А дополнительные правила означают лишнюю работу как в *изучении* этих правил, так и в наблюдении за «особыми случаями». Язык Java должен быть *простым*, с согласованными правилами, которые не приведут к катастрофической ошибке при разном развитии событий. Поэтому Java (в отличие от C++) позволяет не думать о «Смертоносном Ромбе Смерти». Но это возвращает нас к первоначальной проблеме! *Что же делать с этими Animal и Pet?*

# Интерфейс спешит на помощь!

Java дает вам решение — *интерфейс*. Не *графический интерфейс*, не общее *понятие* «интерфейс», как в выражении «Это открытый интерфейс для API класса Button», а *ключевое слово* **`interface`**.

Интерфейс в Java решает проблему множественного наследования, разрешая вам пользоваться *преимуществами* этого наследования без негативных последствий от «Смертоносного Ромба Смерти» (CPC).

Способ, с помощью которого интерфейсы обходят CPC, на удивление прост: они ***делают все методы абстрактными***! Таким образом, подкласс обязан переопределять методы (помните, абстрактные методы *должны* быть реализованы первым конкретным подклассом), чтобы при выполнении программы JVM не запуталась, выбирая вызов из двух наследованных версий.




**Интерфейс в Java практически на 100 % представляет собой чистый абстрактный класс.**

Все методы в интерфейсе абстрактные, поэтому любой класс, соответствующий классу Pet, должен реализовывать (то есть переопределять) его методы.

**Для определения интерфейса:**

```
public interface Pet {...}
```

Используем ключевое слово interface вместо class.

**Для реализации интерфейса:**

```
public class Dog extends Canine implements Pet {...}
```

Используем ключевое слово implements вслед за именем интерфейса. Заметьте, даже когда вы реализуете интерфейс, вам все еще нужно наследовать класс.

# Создание и реализация интерфейса Pet

Здесь вы пишете слово interface вместо class.

Методы интерфейса неявно считаются публичными и абстрактными, поэтому писать слова public и abstract необязательно (в действительности их добавление не говорит о «хорошем стиле» программирования, но мы сделали это для акцентирования внимания, а еще потому, что никогда не были рабами моды...).

```
public interface Pet {

    public abstract void beFriendly();
    public abstract void play();

}
```

Все методы интерфейса абстрактные, поэтому должны заканчиваться точкой с запятой. Не забывайте, что у них нет тела!

Dog соответствует классу Animal и интерфейсу Pet.

Вы пишете implements вслед за именем интерфейса.

```
public class Dog extends Canine implements Pet {

    public void beFriendly() {...}
    public void play() {..}

    public void roam() {...}
    public void eat() {...}

}
```

Вы указали, что вы — Pet, поэтому нужно реализовать методы данного класса. Это ваш контракт. Заметьте, что вместо точки с запятой указаны фигурные скобки.

Это обычные переопределяемые методы.

## Это не глупые вопросы

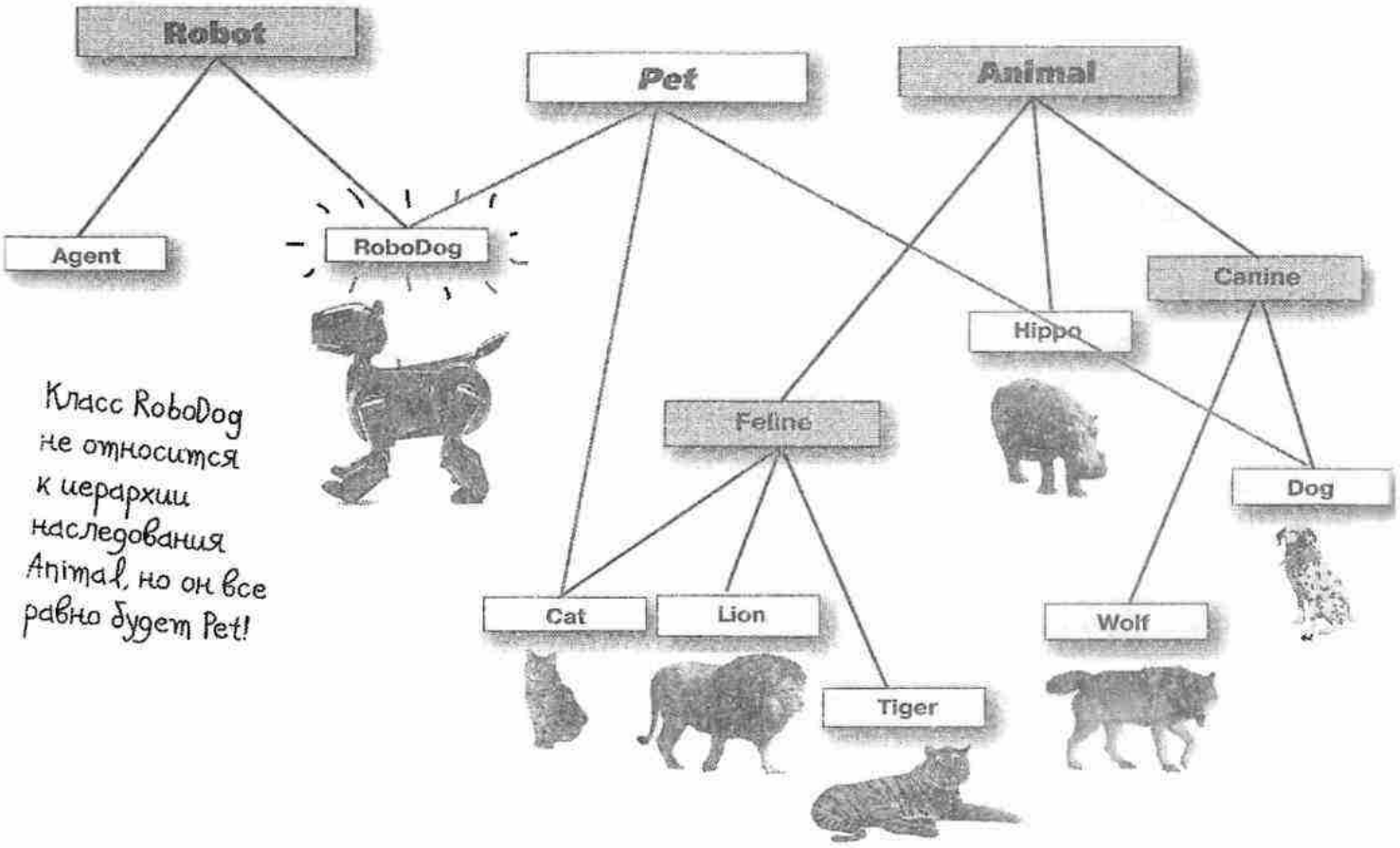
**В: Погодите, интерфейсы на самом деле не предоставляют множественного наследования, так как нельзя поместить в них код реализации. Если все методы абстрактны, то что дают нам эти интерфейсы?**

О: Полиморфизм. Интерфейсы крайне гибкие — если вы используете один из них вместо конкретного подкласса (или даже типа абстрактного родительского класса) в качестве аргумента и типа возвращаемого значения, то сможете передать все, что реализует этот интерфейс. Задумайтесь — благодаря интерфейсу класс не обязан выходить только из одной иерархии наследования. Он может наследовать один класс и реализовать интерфейс. Но другой класс может переопределить тот же самый интерфейс, но уже из иной иерархии наследования! Нужно относиться к объекту так, как он этого заслуживает, то есть согласно его роли, а не по типу класса, экземпляром которого он считается.

Между прочим, если ваш код будет использовать интерфейсы, вам не придется никому предоставлять родительский класс, который затем будет расширяться. Вы сможете просто предоставить интерфейс и сказать: «Вот, меня не волнует, из какой структуры наследования классов ты вышел, просто реализуй этот интерфейс, и будет тебе счастье».

Да, вы не можете добавлять код реализации, но это не проблема для хороших разработок, так как многие методы интерфейса не будут иметь смысла, если реализовывать их в общем виде. Другими словами, большинство методов интерфейса придется переопределять, даже если они не были принудительно объявлены абстрактными.

## Классы из *разных* иерархий наследования могут реализовывать *один и тот же* интерфейс.



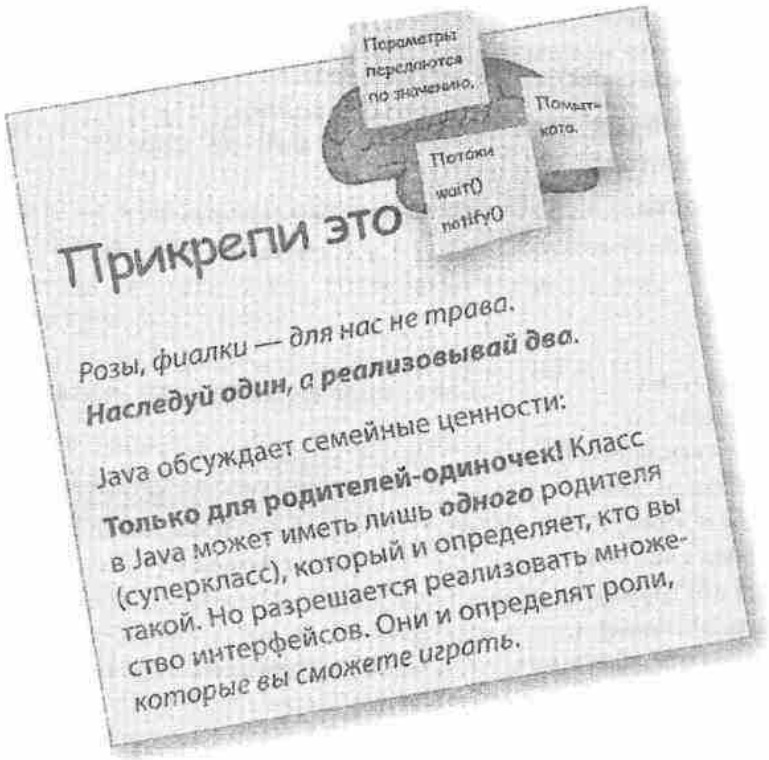

Когда вы используете класс как полиморфический тип (например, массив типа Animal или метод, принимающий аргумент типа Canine), объекты, которые вы можете прикрепить к этому типу, должны быть из одной иерархии наследования. Кроме того, они должны содержать класс, который будет дочерним классом полиморфического типа. Аргумент типа Canine может принимать Wolf и Dog, но не Cat или Hippo.

Но, когда вы используете интерфейс как полиморфический тип (например, массив для объектов Pet), объекты могут быть откуда угодно. Единственное требование — они должны содержать класс, который реализует этот интерфейс. Возможность классов из разных иерархий наследования переопределять общий интерфейс — ключевая особенность в Java API. Хотите, чтобы объект сохранял свое состояние в файл? Реализуйте интерфейс Serializable. Хотите, чтобы объекты запускали свои методы в отдельных потоках? Реализуйте Runnable. Надеемся, вы поняли идею. В следующих главах вы больше узнаете о Serializable и Runnable, а сейчас запомните, что классам из любого места в иерархии наследования, возможно, понадобится реализация этих интерфейсов. Практически каждый класс хочет быть сохраняемым или запускаемым.

### Более того, класс может реализовывать несколько интерфейсов!

Объект Dog соответствует классам Canine, Animal и Object, и все благодаря наследованию. Однако Dog соответствует интерфейсу Pet, так как реализует его. Кроме того, он может реализовывать другие интерфейсы. Просто перечислите их через запятую:

```
public class Dog extends Animal implements
Pet, Saveable, Paintable { ... }
```




## Как узнать, когда нужно создавать класс, подкласс, абстрактный класс или интерфейс

- Создавайте класс, который ничего не наследует (не считая Object), когда ваш новый класс не проходит проверку на соответствие другим типам.

- Создавайте дочерний класс, только когда вам нужно сделать **более специфичную** версию родительского класса и заменить или добавить новое поведение.

- Используйте абстрактный класс, когда хотите определить **шаблон** для группы подклассов и у вас есть хоть *какой-нибудь* код реализации, который смогут применять все подклассы. Делайте класс абстрактным, когда хотите получить гарантию того, что никто не сможет создать объекты данного типа.

- Пользуйтесь интерфейсом, когда хотите определить ***роль***, которую смогут играть другие классы, невзирая на то, где они находятся в иерархии наследования.

## Вызываем родительскую версию метода

**В: Что делать, если создан конкретный подкласс и нужно заменить метод, но требуется родительская версия этого метода? Другими словами, нет необходимости *заменять* весь метод, нужно лишь *добавить* к нему дополнительный код.**

О: Ах... Подумайте о значении слова extends («расширяет»). Рекомендации по проектированию объектно ориентированных систем частично касаются способов разработки конкретного кода, который *подразумевался* бы как заменяемый. Другими словами, вы пишете код метода, скажем, в абстрактном классе, способном поддерживать обычные конкретные реализации. Но, чтобы обработать *всю* специфичную для дочерних классов работу, конкретного кода недостаточно, поэтому подкласс переопределяет метод и *расширяет* его добавлением нужного кода. Ключевое слово super позволяет вызывать родительскую версию замененного метода из самого дочернего класса.

Родительская версия метода делает важные вещи, которые могут применяться в дочерних классах

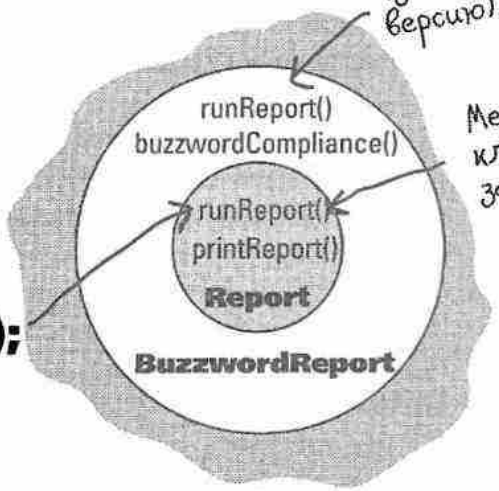
```
abstract class Report {
    void runReport() {
        // set-up report
    }
    void printReport() {
       // generic printing
    }
}

class BuzzwordsReport extends Report {

    void runReport() {
        super.runReport();
        buzzwordCompliance();
        printReport();

    }
    void buzzwordCompliance() {...}
}
```

Вызовите родительскую версию, затем вернитесь и создайте дочерние классы с конкретными задачами.

Если код метода внутри подкласса BuzzwordReport содержит:

```
super.runReport();
```

то запустится метод runReport() внутри родительского класса Report.




### super.runReport();

Ссылка на объект подкласса (BuzzwordReport) всегда будет вызывать дочернюю версию переопределенного метода. Это полиморфизм. Однако код подкласса может вызвать super.runReport() для запуска родительской версии.

Ключевое слово super — это на самом деле ссылка к родительской части объекта.

Когда код подкласса использует super, как в super.runReport(), запускается родительская версия метода.

## КЛЮЧЕВЫЕ МОМЕНТЫ

- Если вы не хотите, чтобы создавались экземпляры какого-нибудь класса (то есть кто-то создавал для него объекты), отметьте его ключевым словом **abstract**.
- Абстрактный класс может содержать и абстрактные, и неабстрактные методы.
- Если класс содержит хотя бы один абстрактный метод, то он должен быть отмечен как абстрактный.
- У абстрактного метода нет тела, и его объявление заканчивается точкой с запятой (без фигурных скобок).
- Все абстрактные методы должны быть реализованы в первом конкретном подклассе в иерархии наследования.
- Каждый класс в Java считается либо прямым, либо косвенным наследником класса **Object** (java.lang.Object).
- Возвращаемые значения и аргументы методов могут иметь тип Object.
- Вы можете вызывать методы объекта, *только* если они присутствуют в классе (или интерфейсе), который использовался в качестве типа ссылки, несмотря на фактический тип *объекта*. По этой причине ссылочная переменная типа Object может применяться только для вызова методов, определенных в классе Object, несмотря на указываемый ссылкой тип объекта.
- Ссылочная переменная типа Object не может присваиваться ссылке другого типа без *приведения*. Приведение часто применяется для присвоения ссылке одного типа ссылки, имеющей дочерний тип. Однако во время выполнения программы приведение не сработает, если тип объекта в куче окажется несовместимым с типом приведения.

  Пример: `Dog d = (Dog) x.getObject(aDog);`
- Все объекты берутся из ArrayList<Object> как объекты типа Object (это означает, что на них могут указывать только ссылки этого типа, если вы не станете использовать *приведение*).
- Множественное наследование в Java запрещено из-за проблемы, связанной со «Смертоносным Ромбом Смерти». Это значит, что вы можете наследовать лишь один класс (то есть можете иметь только одного непосредственного родителя).
- Интерфейс — практически на 100 % чистый абстрактный класс. Он определяет только абстрактные методы.
- Создавайте интерфейс, используя ключевое слово **interface** вместо слова class.
- Реализуйте интерфейс, используя ключевое слово **implements**.

  Пример: `Dog implements Pet`
- Ваш класс *может* реализовывать много интерфейсов.

  Класс, реализующий интерфейс, должен реализовывать все его методы, так как ***все методы интерфейсов неявно считаются публичными и абстрактными.***
- Для вызова родительской версии метода из заменяющего его дочернего класса используйте ключевое слово **super**.

  Пример: `super.runReport();`

**В: Все-таки здесь есть что-то не то... Вы так и не объяснили, почему ArrayList<Dog> возвращает ссылку на объект Dog, которую не нужно приводить, однако класс ArrayList использует в своих методах Object, а не Dog (или DotCom или что-то еще). Что происходит, когда вы пишете ArrayList<Dog>?**

**О:** Это особый трюк, который состоит в том, что ArrayList<Dog> возвращает Dog без необходимости выполнять какое-либо приведение, так как методы класса ArrayList ничего не знают ни о Dog, ни о других типах, кроме Object.

Если говорить кратко, то *компилятор выполняет приведение за вас!* Указывая ArrayList<Dog>, вы не добавляете специальный класс, который бы содержал методы для приема и возврата объектов Dog. Вместо этого есть запись <Dog>. Она сообщает компилятору, что вы хотите, чтобы он позволил вам помещать только объекты Dog и останавливал вас при попытке добавить в список объект другого типа. А поскольку компилятор предостерегает вас от добавления всего, кроме Dog в ArrayList, то он также знает, что безопаснее приводить все, что выходит из этого ArrayList в ссылку Dog. Другими словами, использование ArrayList<Dog> избавляет вас от необходимости приводить объекты обратно к типу Dog. Но все гораздо серьезнее, чем кажется, потому что во время выполнения программы процесс приведения может завершиться неудачей. Вы, скорее всего, предпочтете, чтобы ошибки произошли на этапе компиляции, а не в тот важный момент, когда ваш клиент будет использовать программу.

Но этот вопрос гораздо шире, и мы подробнее рассмотрим его в главе 16.

## Упражнение: каким будет изображение?

**Упражнение** Это ваш шанс продемонстрировать свои художественные способности. Слева перечислены объявления классов и интерфейсов. Ваша задача — изобразить справа соответствующие схемы классов. Первую мы сделали за вас. Используйте пунктирную линию для «реализует» и сплошную для «расширяет».

### Исходные данные:

1)
```
public interface Foo { }
public class Bar implements Foo { }
```

2)
```
public interface Vinn { }
public abstract class Vout implements Vinn { }
```

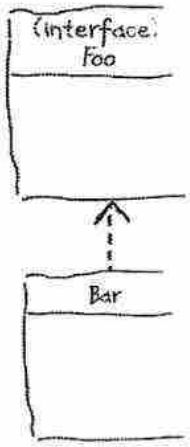
3)
```
public abstract class Muffie implements Whuffie { }
public class Fluffie extends Muffie { }
public interface Whuffie { }
```

4)
```
public class Zoop { }
public class Boop extends Zoop { }
public class Goop extends Boop { }
```

5)
```
public class Gamma extends Delta implements Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alpha extends Gamma implements Beta { }
public class Delta { }
```

### Каким будет изображение?

1)




2)

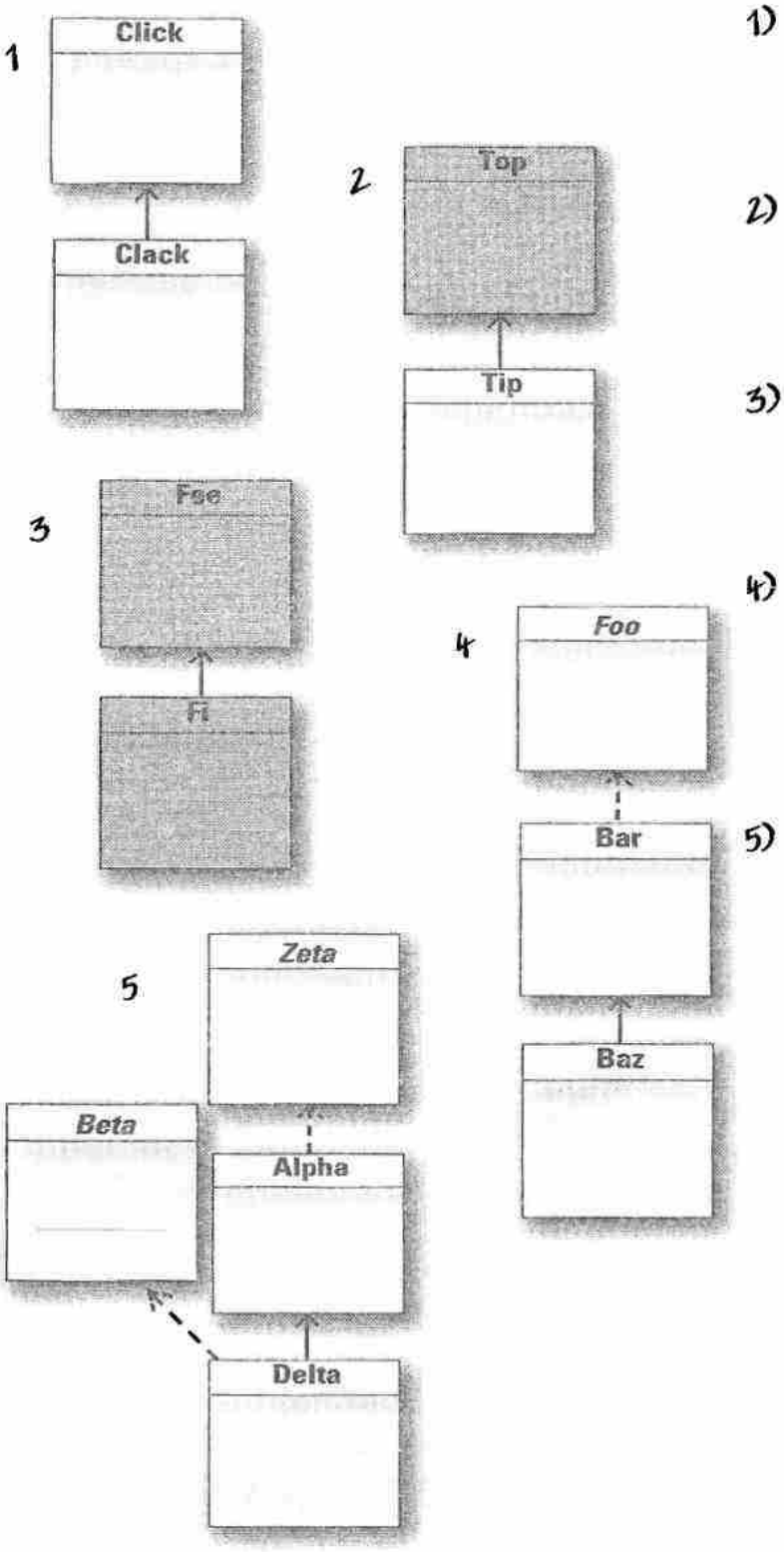
3)

4)

5)

Слева расположены схемы классов. Ваша задача — написать к ним правильный код на языке Java. Первый пункт мы уже добавили (и он был довольно трудный).

Исходные данные:




Каким будет объявление?

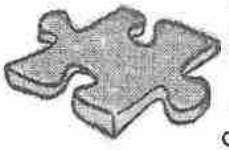
1) public class Click { }
public class Clack extends Click { }

2)

3)

4)

5)



# Головоломка у бассейна

Ваша ***задача*** — взять фрагменты кода из бассейна и поместить их на свободные места. Вы **можете** использовать один фрагмент несколько раз. Кроме того, необязательно применять все фрагменты. Ваша ***цель*** — создать класс, который скомпилируется, запустится и выдаст приведенный ниже результат.

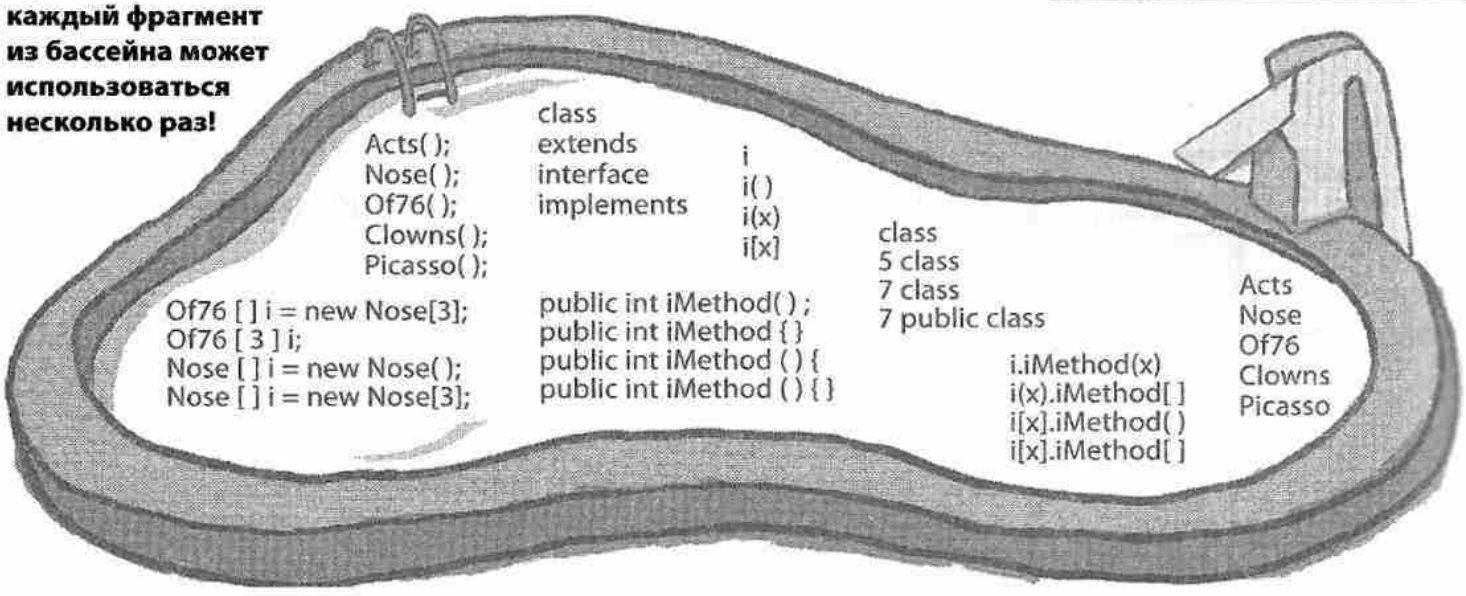
```
__________ Nose {
  ____________________
}

abstract class Picasso implements _____{
  ____________________
_____ return 7;
  }
}

class _______ _______ _________ { }

class _______ _______ _________ {
  ____________________
    return 5;
  }
}
```

```
public _________ _______ extends Clowns

  public static void main(String [] args)

    ____________________________
    i[0] = new _________
    i[1] = new _________
    i[2] = new _________
    for(int x = 0; x < 3; x++) {
      System.out.println(_____________
              + " " + _______.getClass( )
    }
  }
}
```

**Результат:**

```
File Edit Window Help BeAfraid
%java _____________
5 class Acts
7 class Clowns
_____ Of76
```

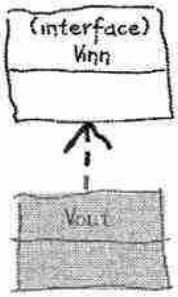
**Примечание: каждый фрагмент из бассейна может использоваться несколько раз!**



# Ответы

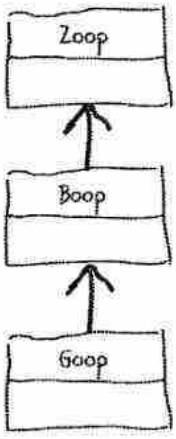
## Каким будет изображение?

2)

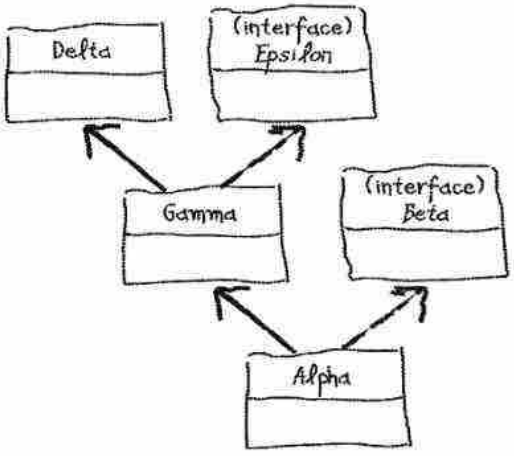



3)

4)




5)



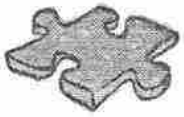

2) public abstract class Top { }
public class Tip extends Top { }

3) public abstract class Fee { }
public abstract class Fi extends Fee { }

4) public interface Foo { }
public class Bar implements Foo { }
public class Baz extends Bar { }

5) public interface Zeta { }
public class Alpha implements Zeta { }
public interface Beta { }
public class Delta extends Alpha implements Beta { }

## Ответы



### Головоломка у бассейна

```
interface Nose {
  public int iMethod( );
}
abstract class Picasso implements Nose {
  public int iMethod( ) {
    return 7;
  }
}
class Clowns extends Picasso { }

class Acts extends Picasso {
  public int iMethod( ) {
    return 5;
  }
}
```

```
public class Of76 extends Clowns {
  public static void main(String [] args) {
    Nose [ ] i = new Nose [3];
    i[0] = new Acts( );
    i[1] = new Clowns( );
    i[2] = new Of76( );
    for(int x = 0; x < 3; x++) {
      System.out.println( i [x] . iMethod( )
            + " " + i [x].getClass( ) );
    }
  }
}
```

**Результат:**

```
File Edit Window Help KillTheMime
%java Of76
5 class Acts
7 class Clowns
7 class  Of76
```

# Жизнь и смерть объектов

**Объекты рождаются и умирают.** Вы управляете их жизненным циклом. Вы решаете, когда и как **создавать** их. И вы решаете, когда их **уничтожать**. Но на самом деле вы не *уничтожаете* их, а просто делаете *брошенными* и *недоступными*. Уже после этого безжалостный **сборщик мусора** может аннулировать объекты, освобождая используемую память. Если вы собираетесь писать на Java, то нужно будет создавать объекты. Рано или поздно от некоторых из них вам придется избавиться, иначе возникнет риск перерасхода оперативной памяти. В этой главе мы рассмотрим, как создаются объекты, где размещаются, как эффективно использовать их и делать недоступными. Это означает, что речь пойдет о куче, стеке, областях видимости, конструкторах (обычных и родительских), нулевых ссылках и многом другом. Но сразу предупреждаем: глава содержит описание смертей объектов, а это зрелище не для слабонервных. Не принимайте все близко к сердцу.

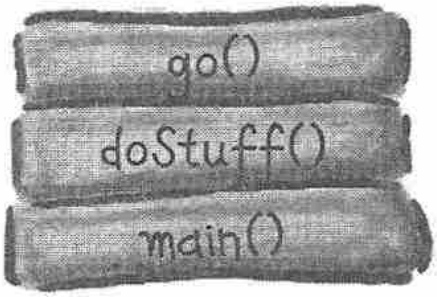
# Стек и куча: где все хранится

Прежде чем понять, что на самом деле происходит при создании объекта, нужно вернуться немного назад. Следует узнать больше о том, где и как долго Java хранит данные. Это означает, что мы должны более подробно изучить стек и кучу. При работе с Java нас как программистов волнуют две области памяти: в одной находятся объекты (куча), а в другой хранятся вызовы методов и локальные переменные (стек). При запуске JVM получает часть памяти от исходной операционной системы и использует ее для выполнения Java-программ. Сколько именно выделяется памяти и можно ли управлять ее объемом, зависит от версии JVM (и от платформы). Но, как правило, такие подробности будут от вас скрыты. Кроме того, при должном подходе к программированию они, скорее всего, не будут вас интересовать (подробнее мы это рассмотрим чуть позже).

Мы знаем, что все *объекты* размещаются в куче, управляемой сборщиком мусора, но нам пока не известно, где обитают *переменные*. Зависит ли место обитания переменных от их *типа*? Надо отметить, что под типами мы подразумеваем не примитивы или ссылки на объекты. Типы, которые нас сейчас интересуют, разделяют переменные на *локальные* и принадлежащие классу *(переменные экземпляра)*. Локальные переменные также известны как *стековые*, что указывает на место их размещения.

### Стек

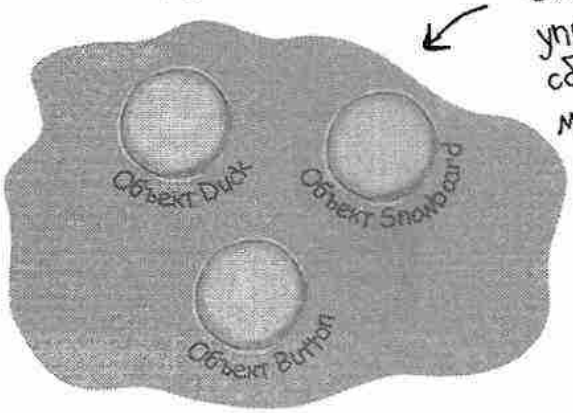
Место, где обитают вызовы методов и локальные переменные.




### Куча

Место, где обитают **все** объекты.

Известна еще как куча, управляемая сборщиком мусора.




## Переменные экземпляра

**Объявляются внутри класса, но *не* внутри метода.** Они представляют собой «поля», которые содержатся в каждом конкретном объекте (и могут иметь разные значения для каждого экземпляра одного класса). Переменные экземпляра обитают внутри объектов, которым принадлежат.

```
public class Duck {

    int size;

}
```

← Каждый экземпляр Duck содержит переменную экземпляра size.

## Локальные переменные

**Объявляются внутри *метода* (это касается и параметров метода).** Они временные и существуют до тех пор, пока метод находится в стеке (иными словами, пока метод не достиг закрывающей фигурной скобки).

```
public void foo(int x) {

    int i = x + 3;

    boolean b = true;

}
```

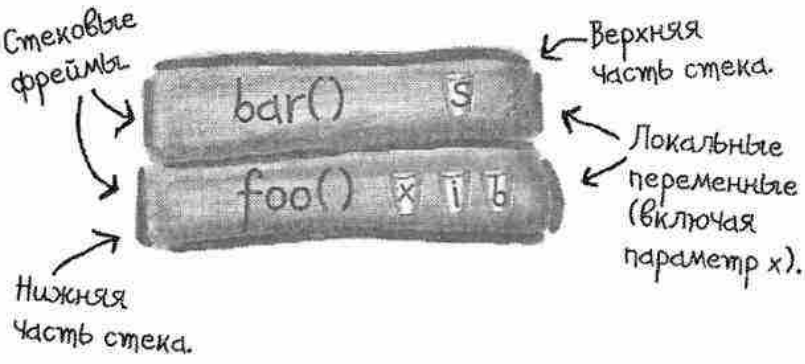
Параметр x и переменные i и b — это локальные переменные.

# Методы размещаются в стеке

Вызывая метод, вы берете его из вершины стека. Эта новая для вас сущность находится в стековом *фрейме* и хранит состояние метода, включая строку кода, которая выполняется в текущий момент, а также значения всех локальных переменных.

На *вершине* стека всегда расположен метод, который выполняется в данный момент (будем считать, что существует всего один стек). Метод остается в стеке до тех пор, пока при выполнении программа не дойдет до закрывающей фигурной скобки (это будет означать, что метод завершился). Если метод *foo()* вызывает метод *bar()*, то метод *bar()* помещается сразу над методом *foo()*.

Стек вызовов с двумя методами:




**Метод, выполняющийся в данный момент, всегда находится на вершине стека.**

---

```
public void doStuff() {
   boolean b = true;
   go(4);
}
public void go(int x) {
   int z = x + 24;
   crazy();
   // Представьте, что здесь
   // тоже находится код
}
public void crazy() {
   char c = 'a';
}
```

## Поведение стека

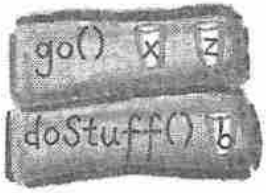
В коде из фрагмента, размещенного слева (не важно, как выглядит остальная часть класса), содержится три метода. Первый метод *doStuff()* вызывает второй метод *go()*, а тот, в свою очередь, — третий *crazy()*. Каждый объявляет в своем теле локальную переменную, а метод *go()* также определяет параметр (это означает, что у *go()* две локальные переменные).

1. Код из другого класса вызывает метод **doStuff()**, помещая его во фрейм на вершине стека. Булева переменная **b** размещается внутри стекового фрейма **doStuff()**.

2. **doStuff()** вызывает метод **go()**, который *выталкивается* на вершину стека. Переменные **x** и **z** находятся в стековом фрейме **go()**.

3. **go()** вызывает метод **crazy()**, который теперь находится на вершине стека и хранит в своем стековом фрейме переменную **c**.

4. Метод **crazy()** завершается, и его фрейм *покидает* стек. Управление возвращается методу **go()** и продолжается со строки, следующей за вызовом **crazy()**.

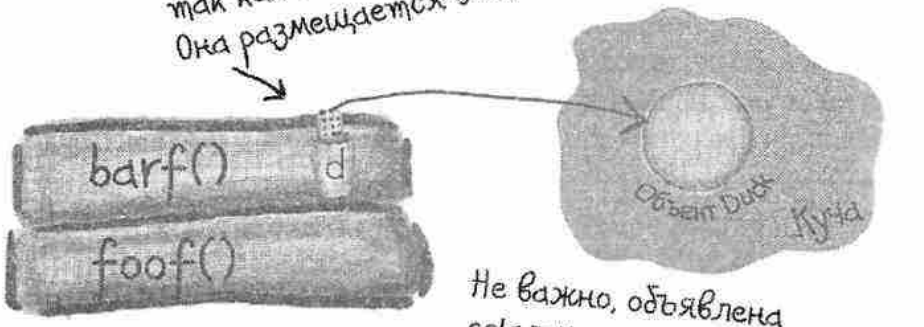


# Как работают локальные переменные, являющиеся объектами

Как вы помните, переменные непростых типов хранят не сами объекты, а только *ссылки* на них. Вы уже знаете, что объекты обитают в куче. При этом не важно, где они были объявлены или созданы. ***Если локальная переменная ссылается на объект, то в стек помещается только она (то есть ссылка, или пульт управления).***

*Сам объект по-прежнему находится в куче.*

```
public class StackRef {
   public void foof() {
      barf();
   }

   public void barf() {
      Duck d = new Duck(24);
   }
}
```




## Это не глупые вопросы

**В: Спрошу еще раз: зачем мы изучаем эти стеки и кучи? Как они мне помогут? Мне на самом деле нужно это знать?**

О: Знание основных принципов работы стека и кучи в Java чрезвычайно важно для понимания областей видимости переменных, особенностей создания объектов, управления памятью, потоков и обработки исключений. Последние две темы мы рассмотрим позже, но все остальное описано в этой главе. Вам не нужно ничего знать о том, *как* в конкретной версии JVM и/или платформе устроены стек и куча. Все, что вам должно быть о них известно, изложено на этой и предыдущей страницах. Если вы внимательно их прочитаете, то все следующие темы, связанные с этим материалом, будут намного проще усваиваться. Однажды вы еще скажете спасибо за то, что мы заставили вас вызубрить стеки и кучи.

## КЛЮЧЕВЫЕ МОМЕНТЫ

- У Java есть две области памяти, о которых нам нужно знать: стек и куча.
- Переменные экземпляра объявляются внутри класса, но за пределами метода.
- Локальные переменные объявляются внутри метода или задаются в качестве параметров.
- Все локальные переменные обитают в стеке, во фрейме соответствующего метода, в котором были объявлены.
- Объектные ссылки работают точно так же, как переменные простых типов — если они объявляются локально, то уходят в стек.
- Все объекты находятся в куче, независимо от того, объявлена ли их ссылка локально или в качестве переменной экземпляра.

## Если локальные переменные обитают в стеке, то где находятся переменные экземпляра?

Когда вы задаете объект new CellPhone(), Java приходится выделить для него место в куче. Но *сколько именно* места? Достаточно для объекта, а точнее — для всех его переменных экземпляра. Именно так переменные экземпляра класса размещаются в куче, внутри объекта, которому принадлежат.

Помните, что *значения* переменных объекта находятся внутри его. Если все переменные будут примитивами, Java выделит для них место, основываясь на их простых типах. Типу int нужно 32 бита, long — 64 бита и т. д. Java не интересуют значения внутри переменных простых типов; переменная типа int всегда будет занимать свои 32 бита, независимо от того, какое значение она хранит — 32 000 000 или 32.

А если переменные экземпляра — *объекты*? Например, CellPhone содержит объект Antenna, то есть у CellPhone есть переменная, ссылающаяся на объект Antenna.

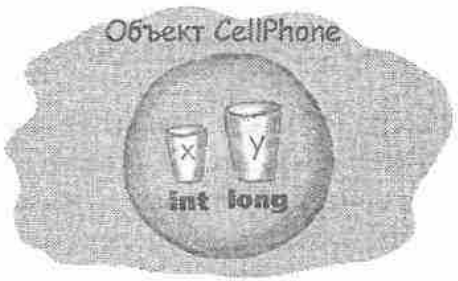
Если новый объект содержит переменные экземпляра в виде объектных ссылок, а не примитивов, то возникает вопрос: нужно ли выделять место для объектов, ссылки на которые вмещает в себя главный объект? На самом деле Java выделяет место для *значений* переменных. Но помните, что значение ссылки — это не *объект*, а всего лишь *пульт для управления* им. Поэтому, если CellPhone содержит переменную не простого типа Antenna, то Java выделит место внутри объекта CellPhone только для *ссылки* на Antenna, но не для самого *объекта*.

Когда же *объект* Antenna получит место в куче? Прежде всего нужно выяснить, *когда* он создается. Это зависит от того, каким образом объявлена переменная экземпляра. Если при объявлении ей не был присвоен объект, то будет выделено место только для ссылочной переменной (пульта дистанционного управления).

```
private Antenna ant;
```

В куче не будет никакого объекта Antenna до тех пор, пока ссылочной переменной не будет присвоен новый объект.

```
private Antenna ant = new Antenna();
```

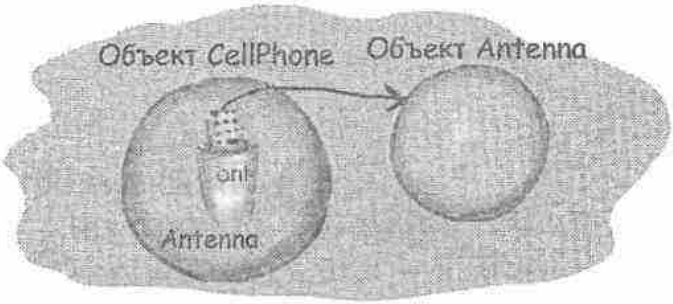



Объект с двумя переменными экземпляра простого типа.

Место для переменных находится внутри объекта.

Объект с переменной экземпляра не простого типа, ссылающейся на Antenna (ссылка, но не сам объект). Вот что вы получите, если объявите переменную, но не присвоите ей объект Antenna.

```
public class CellPhone {
  private Antenna ant;
}
```




Объект с переменной экземпляра не простого типа и переменная Antenna, которой присвоили соответствующий объект.

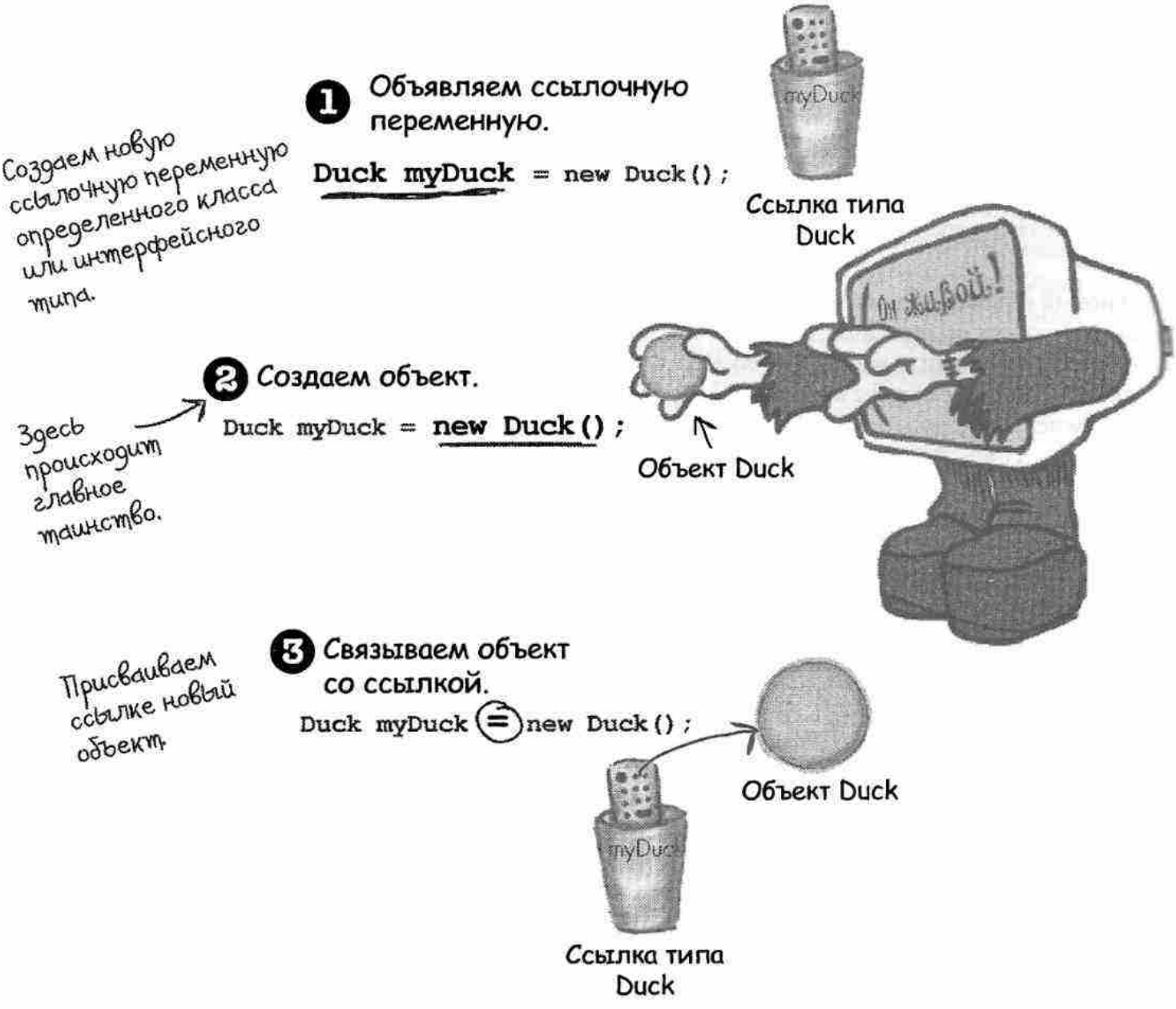
```
public class CellPhone {
  private Antenna ant = new Antenna();
}
```

# Таинство создания объекта

Теперь, когда вы знаете, где размещаются переменные и объекты, мы можем погрузиться в таинственный процесс их создания. Вспомним три шага, которые нужно выполнить для объявления и присвоения объекта: объявление ссылочной переменной, создание объекта и присвоение объекта ссылке.

До этого момента второй шаг, где происходило чудо «рождения» нового объекта, оставался для вас большой загадкой. Приготовьтесь изучить факты из жизни объекта. *Надеемся, у вас крепкие нервы.*

**Три шага для объявления, создания и присвоения объекта.**



**Мы вызываем метод под названием Duck()? Это *выглядит* именно так.**

```
Duck myDuck = new Duck();
```

Учитывая круглые скобки, мы, похоже, вызываем метод с именем Duck().

**Нет.**

**Мы вызываем *конструктор* объекта Duck.**

Конструктор действительно выглядит и ведет себя как мстод, но это все же не метод. Он отвечает за код, который выполняется, когда вы пишете ключевое слово **new**. Иными словами, это *код, который запускается при создании экземпляра класса.*

Единственный способ вызвать конструктор — указать перед именем класса ключевое слово **new**. JVM найдет этот класс и запустит его конструктор. Хотя, если говорить начистоту, формально это не *единственный* способ. С некоторыми ограничениями можно вызвать конструктор внутри другого конструктора, но об этом мы поговорим позже.

**Конструктор** содержит код, который выполняется при создании объекта. Это код, который запускается, когда вы пишете слово new перед типом класса.

Любой ваш класс включает в себя конструктор, даже если вы сами его не создавали.

**Но откуда берется конструктор?**

**Если его написали не мы, то кто?**

Вы можете создать конструктор для своего класса (скоро мы этим займемся), но даже в ином случае ***компилятор сам создаст его для вас***!

Вот как выглядит конструктор по умолчанию, создаваемый компилятором:

```
public Duck() {

}
```

**Заметили, что кое-чего не хватает? Чем конструктор отличается от метода?**

Его имя совпадает с именем класса. Это обязательное условие.

```
public Duck() {
    // Здесь размещается конструктор
}
```

Где тип возвращаемого значения? Будь это метод, вам пришлось бы указать его тип между public и Duck().

# Создаем объект Duck

Ключевая особенность конструктора — он выполняется *до того*, как объект может быть присвоен ссылке. Это означает, что у вас есть шанс вмешаться в процесс и подготовить объект для дальнейшего применения. Другими словами, прежде чем кто-либо захочет использовать пульт управления объектом, этот объект может помочь себя создать. В конструкторе для объекта Duck мы не делаем ничего полезного. Тем не менее можно выделить такую последовательность событий.

```
public class Duck {

    public Duck() {
        System.out.println("Кря");
    }
}
```

Код конструктора.

**Конструктор позволяет вам вмешаться в процесс создания на этапе выполнения оператора new.**

```
public class UseADuck {

    public static void main (String[] args) {
        Duck d = new Duck();
    }
}
```

Здесь вызывается конструктор объекта Duck.

```
File Edit Window Help Quack
% java UseADuck
Кря
```

## Наточите свой карандаш

Конструктор позволяет вам вмешаться в код на этапе создания объекта, а именно при выполнении оператора new. Можете ли вы представить ситуацию, когда это окажется полезным? Какие действия могут пригодиться в конструкторе класса Car, созданного для гоночной игры? Отметьте варианты, которые будут уместны в данном контексте.

- Инкрементировать счетчик для отслеживания количества созданных объектов класса.
- Присвоить состояние, которое может меняться при каждом выполнении программы (информация о том, что происходит сейчас).
- Присвоить значение важной переменной экземпляра объекта.
- Получить и сохранить ссылку на объект, который сам создает новый объект.
- Добавить объект в ArrayList.
- Создать объекты внутри класса.
- _____(ваш вариант).

## Инициализация состояния нового объекта Duck

Большинство людей применяют конструктор для инициализации состояния объекта, то есть для создания значений и присвоения их переменным объекта.

```
public Duck() {
   size = 34;
}
```

Нет ничего плохого, если *разработчик* класса Duck знает, насколько большим должен быть его объект. А если мы хотим, чтобы программист, *использующий* Duck, определял размеры его конкретного экземпляра?

Представьте, что у Duck есть переменная экземпляра size, и вы хотите, чтобы программист, работающий с вашим классом Duck, установил ее значение для нового объекта. Как это сделать?

Можно добавить в класс сеттер setSize(). Однако объект Duck на какое-то время останется без значения для переменной size, а программисту, создавшему объект, придется написать *два* выражения: одно — для создания экземпляра Duck, другое — для вызова метода setSize(). В коде, приведенном ниже, начальный размер нового объекта Duck устанавливается с помощью сеттера.

```
public class Duck {
   int size;                          ← Переменная экземпляра.

   public Duck() {
      System.out.println("Кря");       ← Конструктор.
   }

   public void setSize(int newSize) {   ← Сеттер.
      size = newSize;
   }
}
```

---

```
public class UseADuck {

   public static void main (String[] args) {
      Duck d = new Duck();
      d.setSize(42);
   }
}
```

Здесь возникает проблема. В этой точке кода объект Duck оживает, но у него нет размера!* Таким образом, приходится положиться на пользователя, применяющего ваш класс. Вам остается надеяться, что он знает о двух шагах для создания объекта Duck: вызове конструктора и последующем вызове сеттера.

* Переменная экземпляра имеет значение по умолчанию. Это 0 или 0.0 для числовых примитивов, false для булевых переменных и null для ссылок.

## Это не глупые вопросы

**В: Зачем писать свой конструктор, если компилятор сам создает его?**

О: Если вам нужен код, чтобы инициализировать объект и сделать его готовым для использования, придется написать собственный конструктор. Кроме того, вы можете зависеть от пользовательского ввода, прежде чем получите возможность завершить создание объекта. Это еще одна причина, по которой понадобится написать конструктор, даже если его код вам не нужен. В коде можно лишь вызывать конструктор родительского класса (мы поговорим об этом совсем скоро).

**В: Как можно вызвать конструктор из метода? И разрешено ли иметь метод, имя которого совпадает с именем класса?**

О: Java позволяет вам иметь метод, имя которого совпадает с именем вашего класса. Но это не делает его конструктором. Конструктор отличается от метода отсутствием типа возвращаемого значения. Метод *должен* иметь свой тип, тогда как у конструктора его быть *не может*.

**В: Наследуются ли конструкторы? Получу ли я конструктор родительского класса, если нет своего?**

О: Нет. Конструкторы не наследуются. Мы рассмотрим это через несколько страниц.

# Использование конструктора для инициализации важного состояния* объекта Duck

Если объект не должен использоваться, пока часть его состояния (переменной экземпляра) не будет инициализирована, не позволяйте никому с ним работать, не завершив инициализацию! Слишком рискованно разрешать пользователям создавать объект Duck и получать на него ссылки, если он не готов. Не делайте этого, пока кто-нибудь не вызовет метод setSize().

Но как человек со стороны сможет *узнать*, что после создания нового объекта необходимо вызвать сеттер? Лучшее место для размещения инициализационного кода — конструктор. И вам нужно лишь создать конструктор с аргументами.

```
public class Duck {
   int size;

   public Duck(int duckSize) {
      System.out.println("Кря");

      size = duckSize;

      System.out.println("Размер равен" + size);
   }
}
```

Добавляем в конструктор Duck параметр типа int.

Используем значение аргумента для инициализации переменной size.

---

```
public class UseADuck {

  public static void main (String[] args) {
     Duck d = new Duck(42);
  }
}
```

Передаем значение в конструктор.

На этот раз использовано только одно выражение, в котором мы создаем новый экземпляр Duck и устанавливаем его размер.

```
File Edit Window Help Honk
% java UseADuck
Кря
Размер равен 42
```

* Это не означает, что какие-то состояния Duck не важны.

# Упростите процесс создания объекта Duck

## Убедитесь, что у вас есть конструктор без аргументов

Подумайте, что случится, если конструктор Duck будет принимать аргумент. На предыдущей странице показан лишь *один* конструктор для объекта Duck, и он принимает для своей переменной *size* целочисленный аргумент. Это не большая проблема, но программисту становится сложнее создавать новые объекты Duck, особенно если он не *знает*, какое значение должно быть у переменной. Не поможет ли в этой ситуации значение по умолчанию, чтобы пользователь, который ничего о нем не знает, мог создать рабочий объект Duck?

***Допустим, вы хотите, чтобы у пользователей класса Duck было два способа создать соответствующий объект: один — с указанием размера (в качестве аргумента конструктора), другой — без такового, при этом переменной size присваивается значение по умолчанию.***

Вы не можете этого добиться при наличии только одного конструктора. Помните, что если у метода (или у конструктора — принцип тот же) есть параметр, то при вызове вы *обязаны* передать ему соответствующий аргумент. Нельзя просто сказать: «Если пользователь ничего не передал в конструктор, то применяем значение по умолчанию». Такой код даже не скомпилируется, если при вызове конструктора не передать целочисленный аргумент. Вы *могли бы* сделать следующее:

```
public class Duck {
   int size;

   public Duck(int newSize) {
      if (newSize == 0) {
         size = 27;
      } else {
         size = newSize;
      }
   }
}
```

Если значение параметра равно нулю, то задаем для Duck размер по умолчанию. В ином случае используем значение параметра. Это не очень хорошее решение.

Таким образом, при создании объекта Duck программист должен *знать*, что передача значения 0 — это условие для получения значения по умолчанию. Довольно неудобно. А если другие программисты не знают об этом или *действительно* хотят установить нулевое значение для переменной объекта Duck? Предполагается, что нулевое значение допустимо. Если вы не хотите, чтобы объекты Duck имели нулевые размеры, разместите в конструкторе проверочный код. Проблема в том, что не всегда можно понять, действительно ли программист хотел присвоить переменной size значение 0 или он надеялся получить размер по умолчанию.

**Вам нужно знать два способа создания нового объекта Duck:**

```
public class Duck2 {
   int size;

   public Duck2() {
      // Устанавливаем
      // размер по умолчанию
      size = 27;
   }

   public Duck2(int duckSize) {
      // Используем параметр
      // duckSize
      size = duckSize;
   }
}
```

**Чтобы создать объект Duck, если вам известен размер:**

```
Duck2 d = new Duck2(15);
```

**Чтобы создать объект Duck, если размер *не* известен:**

```
Duck2 d2 = new Duck2();
```

Итак, для реализации идеи с *двумя* вариантами создания объекта Duck нужно написать два конструктора. Один будет принимать число, другой — нет. ***Если в вашем классе содержится несколько конструкторов, это означает, что они перегружены.***

## Всегда ли компилятор создает конструктор без аргументов? Нет!

Вы, должно быть, думаете, что, если создадите *один* конструктор с аргументами, то компилятор увидит, что у вас нет конструктора без аргументов, и создаст его для вас. Но это совсем не так. Компилятор сделает так *только в том случае, если вы сами не создали ни одного конструктора.*

**Если вы написали конструктор, который принимает аргументы, и вам нужен вариант конструктора без аргументов, то придется создать его самостоятельно!**

Как только у *вас* появляется любой вид конструктора, компилятор останавливается и говорит: «Похоже, теперь ты сам можешь позаботиться о конструкторах».

**Если в вашем классе присутствуют сразу несколько конструкторов, то списки их аргументов должны различаться.**

Это касается как порядка следования аргументов, так и их типов. Пока они различны, вы можете иметь несколько конструкторов. То же самое относится и к методам, но об этом мы поговорим в другой главе.

**Наличие перегруженных конструкторов означает, что в вашем классе несколько конструкторов.**

**Чтобы скомпилироваться, каждый конструктор должен иметь *уникальный* список аргументов!**

Класс, представленный ниже, будет работать, так как все его четыре конструктора имеют разные списки аргументов. Если у вас, к примеру, есть два конструктора, которые принимают по одному значению типа int, такой класс не скомпилируется. Имена параметров в данном случае не учитываются. Имеют значение *типы* параметров (int, Dog и т. д.) и *порядок* их следования. У вас может быть два конструктора с параметрами идентичных типов, ***но эти параметры должны размещаться в разном порядке***. Конструктор, принимающий тип String, за которым следует int, отличается от конструктора, который принимает сначала int, а потом String.

Четыре разных конструктора — четыре способа создать новый объект Mushroom.

```
public class Mushroom {

  public Mushroom(int size) { }

  public Mushroom( ) { }

  public Mushroom(boolean isMagic) { }

  public Mushroom(boolean isMagic, int size) { }

  public Mushroom(int size, boolean isMagic) { }

}
```

Если вы знаете размер, но не уверены насчет волшебства.

Если вам вообще ничего не известно.

Если у вас есть информация о волшебстве, но вы ничего не знаете о размере.

У них одинаковые аргументы, но они размещены в разном порядке, так что все хорошо.

Если у вас есть сведения и о размере, и о волшебстве.

## КЛЮЧЕВЫЕ МОМЕНТЫ

- Переменные экземпляра находятся внутри объектов, которым они принадлежат (в куче).
- Если переменная — это ссылка на объект, то она будет храниться вместе с ним в куче.
- Конструктор — это код, который выполняется вместе с оператором new.
- Конструктор обязан иметь то же имя, что и его класс, и не должен содержать тип возвращаемого значения.
- Вы можете использовать конструктор для инициализации состояния (то есть переменных) создаваемого объекта.
- Если вы не разместите в своем классе ни одного конструктора, то компилятор создаст конструктор по умолчанию.
- Стандартный конструктор никогда не имеет аргументов.
- Если вы разместите в своем классе какой-либо конструктор, то компилятор не будет создавать для вас конструктор по умолчанию.
- Если вам нужен конструктор без аргументов и у вас уже есть один (но с аргументами), то придется создать его самостоятельно.
- По возможности всегда создавайте конструктор без аргументов, чтобы облегчить программистам работу с объектом. Предусматривайте значения по умолчанию.
- Наличие перегруженных конструкторов означает, что в вашем классе их несколько.
- Перегруженные конструкторы должны иметь разные списки аргументов.
- Вы не можете создать два конструктора с одинаковыми списками аргументов. Это касается порядка их следования и/или типов.
- Переменным экземпляра присваиваются значения по умолчанию, даже если вы явно их не определили. Для примитивов это 0/0.0/false, для ссылок — null.

## Перегруженные конструкторы

Наточите свой карандаш

Сопоставьте вызовы конструкторов класса Duck с конструкторами, которые будут запускаться при создании экземпляра этого класса. Мы выполнили самый простой вариант, чтобы вам было легче начать.

```
public class TestDuck {

  public static void main(String[] args){

    int weight = 8;
    float density = 2.3F;
    String name = "Дональд";
    long[] feathers = {1,2,3,4,5,6};
    boolean canFly = true;
    int airspeed = 22;

    Duck[] d = new Duck[7];

    d[0] = new Duck();

    d[1] = new Duck(density, weight);

    d[2] = new Duck(name, feathers);

    d[3] = new Duck(canFly);

    d[4] = new Duck(3.3F, airspeed);

    d[5] = new Duck(false);

    d[6] = new Duck(airspeed, density);
  }
}
```

```
class Duck {

  int pounds = 6;
  float floatability = 2.1F;
  String name = "Универсальная";
  long[] feathers = {1,2,3,4,5,6,7};
  boolean canFly = true;
  int maxSpeed = 25;

  public Duck() {
    System.out.println("Утка типа 1"
  }

  public Duck(boolean fly) {
    canFly = fly;
    System.out.println("Утка типа 2"
  }

  public Duck(String n, long[] f) {
    name = n;
    feathers = f;
    System.out.println("Утка типа 3")
  }

  public Duck(int w, float f) {
    pounds = w;
    floatability = f;
    System.out.println("Утка типа 4")
  }

  public Duck(float density, int max)
    floatability = density;
    maxSpeed = max;
    System.out.println("Утка типа 5")
  }
}
```

**В: Раньше вы говорили, что полезно иметь конструктор без аргументов, так как при его вызове можно предоставить для «пропущенных» аргументов значения по умолчанию. Но бывают ли ситуации, когда нельзя обойтись значениями по умолчанию? Случается ли такое, что нельзя иметь конструктор без аргументов?**

О: Да, бывают моменты, когда конструкторы без аргументов бессмысленны. Вы сможете увидеть это на примере Java API — некоторые классы не содержат такого конструктора. Так, класс Color представляет собой... цвет. Его объекты используются, например, для задания или изменения цвета экранного шрифта или графической кнопки. Когда вы создаете экземпляр класса Color, он описывает конкретный цвет (такой убийственно-коричневый или синий, как на экране смерти, и т. д.). При создании объекта Color нужно как-нибудь указать цвет.

```
Color c = new Color(3,45,200);
```

Здесь мы используем три целых числа для значений в формате RGB. Как работать с классом Color, мы поговорим позже, в главе 13.

Что бы вы получили, вызывая конструктор без аргументов? Разработчики Java API *могли* бы решить, что нужно установить милый светло-розовый оттенок. Но хорошее чувство вкуса перевесило.

Если вы попытаетесь создать экземпляр Color без аргументов:

```
Color c = new Color();
```

компилятор сойдет с ума, так как не сможет найти соответствующий конструктор для класса Color.

```
File Edit Window Help StopBeingStupid
cannot resolve symbol
:constructor Color()
location: class java.
awt.Color
Color c = new Color();
          ^
1 error
```

# Нанообзор: четыре вещи, которые необходимо помнить о конструкторах

1. Конструктор — это код, который выполняется, когда кто-нибудь пишет слово new перед типом класса.

```
Duck d = new Duck();
```

2. Конструктор обязан иметь то же имя, что и его класс, и **не** должен содержать тип возвращаемого значения.

```
public Duck(int size) { }
```

3. Если вы не разместите в своем классе ни одного конструктора, то компилятор создаст конструктор по умолчанию, который никогда не содержит аргументов.

```
public Duck() { }
```

4. Вы можете иметь в своем классе несколько конструкторов, но списки их аргументов должны различаться. Присутствие более одного конструктора в классе означает, что у вас есть перегруженные конструкторы.

```
public Duck() { }

public Duck(int size) { }

public Duck(String name) { }

public Duck(String name, int size) { }
```

Все эти мозговые упражнения приводят к увеличению нейронов на 42 %.

**Как насчет родительских классов?**

**Должен ли конструктор Canine запускаться при создании объекта Dog?**

**Обязан ли абстрактный класс *содержать* конструктор?**

Мы ответим на эти вопросы на следующих страницах, а пока остановитесь и подумайте, как ведут себя конструкторы из родительских классов.

## Это не глупые вопросы

**В: Должны ли конструкторы быть публичными?**

О: Нет. Конструкторы могут быть публичными, приватными или без модификатора доступа вообще. О третьем виде доступа (который используется по умолчанию) мы поговорим в главе 16 и Приложении Б.

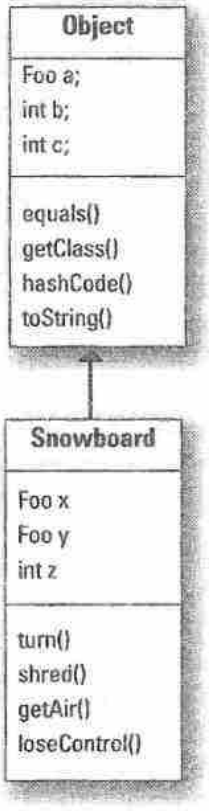
**В: Чем может быть полезен приватный конструктор? Ведь никто никогда не сможет его вызвать и создать с его помощью объект!**

О: Это не совсем так. Делая объект приватным, вы запрещаете доступ к нему не *всем*, а только тем, кто *находится за пределами класса*. Наверняка вам это кажется заколдованным кругом. Только код из класса с приватным конструктором может создать объект из этого же класса, но как вы запустите код, не имея объекта? Как вообще можно получить что-либо из такого класса? *Наберитесь терпения*, мы вернемся к этому в следующей главе.

# Минутку... Мы никогда не говорили о том, как родительские классы и наследование согласуются с конструкторами.

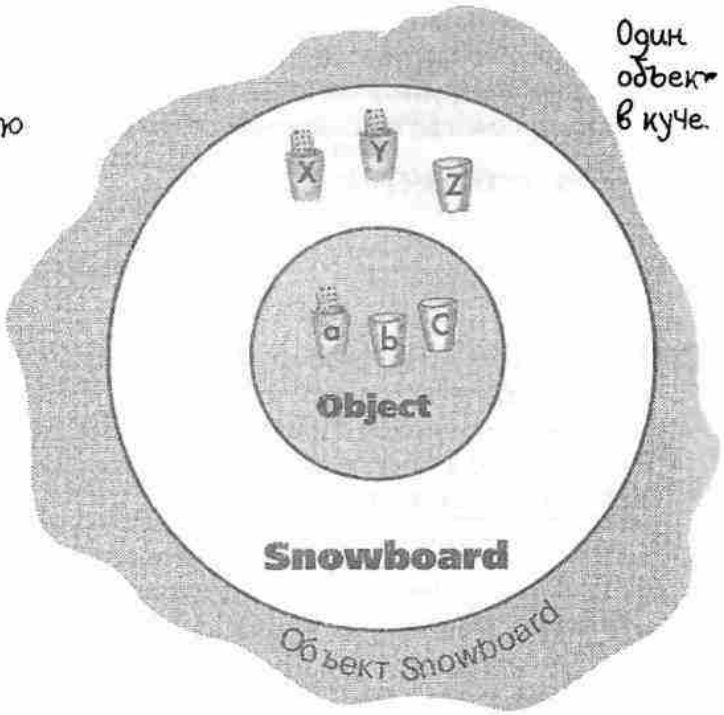
Вот где начинается веселье. Вспомните раздел прошлой главы, где мы рассматривали обертку Snowboard вокруг внутреннего ядра, представляющего часть класса Object. Ее суть заключалась в том, что любой объект содержит не только объявленные им переменные экземпляра, но и *все унаследованные от родительских классов* (это как минимум класс Object, так как *все* классы — его потомки).

Поэтому, когда объект создается (кто-нибудь использует оператор **new**, ведь ***других способов*** создания объектов не существует), он получает место в памяти для *всех* переменных, которые находятся выше его в иерархии наследования. Родительский класс может иметь сеттер, инкапсулирующий приватную переменную. Но переменная должна *где-то* находиться. Создание объекта подобно рождению сразу *нескольких* объектов: один — только что написанный, и по одному объекту на каждый родительский класс. Удобнее представить это так, как показано на рисунке ниже, где создаваемый объект состоит из *слоев*, представляющих все родительские классы.




Класс Object содержит переменные, инкапсулированные с помощью методов для доступа к ним. Такие переменные создаются при появлении экземпляра любого дочернего класса. Это ненастоящие переменные класса Object, но для нас это не имеет значения, так как они инкапсулированы.

У Snowboard есть собственные переменные экземпляра, поэтому для создания этого объекта нам понадобится место для переменных из обоих классов.



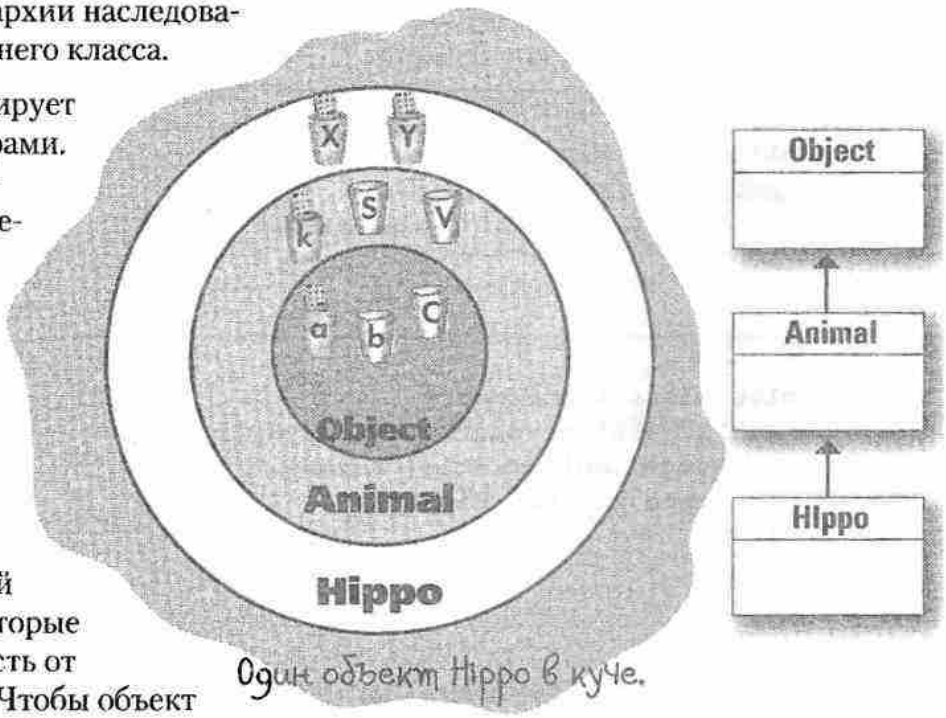

В куче всего один объект — Snowboard. Но помимо кода из собственного класса он содержит элементы класса Object. Здесь будут размещаться переменные обоих классов.

# Роль конструкторов родительского класса в жизни объекта

***При создании нового объекта должны быть запущены конструкторы со всей иерархии наследования этого объекта.***

Это означает, что каждый родительский класс содержит конструктор (так как он есть у каждого класса) и каждый конструктор вверх по иерархии наследования запускается при создании дочернего класса.

**new** — важный оператор. Он провоцирует всю цепочку действий с конструкторами. Ведь конструкторы есть даже у абстрактных классов. И хотя вы не сможете использовать для них оператор new, это все-таки родительские классы, поэтому их конструкторы тоже запускаются при создании экземпляров их потомков.




Один объект Hippo в куче.

Родительские конструкторы запускаются для построения частей объекта, принадлежащих родительскому классу. Помните, что дочерний класс может наследовать методы, которые зависят от состояния родителя (то есть от переменных родительского класса). Чтобы объект был полностью сформирован, нужно сформировать все его части, унаследованные от родительского класса. Именно поэтому родительский конструктор *должен быть* запущен. Все переменные экземпляра из иерархии наследования должны быть объявлены и инициализированы. Даже если класс Animal содержит переменные, которые не наследуются классом Hippo (например, они приватные), то Hippo все равно зависит от методов из Animal, *использующих* эти переменные.

Начиная свою работу, конструктор немедленно вызывает конструктор родительского класса, и так по цепочке вверх, пока не достигнет конструктора класса Object.

На следующих страницах вы узнаете, каким образом вызываются родительские конструкторы и как вы сами можете это делать. Вы также увидите, что нужно делать, если у конструктора родительского класса есть аргументы.

**Новый объект Hippo связан отношением IS-A с классами Animal и Object. Если вы хотите создать экземпляр Hippo, придется также создавать его части, принадлежащие Animal и Object.**

**Этот процесс называется связыванием (формированием цепочки) конструкторов.**

# Создание объекта Hippo ведет к созданию Animal и Object, из которых он состоит...

```
public class Animal {
   public Animal() {
      System.out.println("Создание Animal");
   }
}
```

---

```
public class Hippo extends Animal {
   public Hippo() {
      System.out.println("Создание Hippo");
   }
}
```

---

```
public class TestHippo {
   public static void main (String[] args) {
      System.out.println("Начало…");
      Hippo h = new Hippo();
   }
}
```

## Наточите свой карандаш

Каким будет реальный результат работы программы? Что напечатает TestHippo из кода, приведенного слева? A или B? Ответ находится внизу страницы.

A

```
File Edit Window Help Swear
Начало…
Создание Animal
Создание Hippo
```

B

```
File Edit Window Help Swear
% java TestHippo
Начало…
Создание Hippo
Создание Animal
```

1. В коде из другого класса написано `new Hippo()`, поэтому конструктор **Hippo()** помещается во фрейм на вершине стека.

2. **Hippo()** вызывает конструктор родительского класса **Animal()** и помещает его на вершину стека.

3. **Animal()** вызывает конструктор родительского класса **Object()** и помещает его на вершину стека, так как Animal — это потомок Object.

4. Работа **Object()** завершается, и его стековый фрейм *удаляется* из стека. Выполнение программы переходит к конструктору **Animal()** и продолжается со строки следующей за вызовом родительского конструктора.

Ответ: верный вариант — A. Сначала вызывается конструктор Hippo(), который первым завершит свою работу конструктор Animal.

# Как вызвать конструктор родительского класса

Вам могло показаться, что, например, если конструктор Duck расширяет Animal, то в нем находится вызов Animal(). Но это не так.

```
public class Duck extends Animal {
   int size;

   public Duck(int newSize) {
      Animal();
      size = newSize;
   }
}
```

Неправильно!

Нет! Так делать нельзя!

Единственный способ вызвать ***родительский конструктор*** — использовать выражение super().

```
public class Duck extends Animal {
   int size;

   public Duck(int newSize) {
      super();
      size = newSize;
   }
}
```

Вы просто пишете super().

Вызов *super()* внутри вашего конструктора автоматически помещает конструктор родительского класса на вершину стека. Как вы думаете, чем занимается родительский конструктор? *Вызывает конструктор своего родительского класса.* И так продолжается до тех пор, пока на вершине стека не оказывается конструктор класса Object. Как только *Object()* завершаст свою работу, он покидаст стек, а на его место помещается следующий конструктор (принадлежащий потомку *Object*, из которого вызывался его конструктор). Потом завершает свою работу *этот* конструктор, и так продолжается до тех пор, пока на вершине стека не оказывается изначальный конструктор, получающий возможность закончить вызов.

**И как это все получилось, несмотря на то что мы ничего не сделали?**

Вы, вероятно, уже догадались.

**Наш хороший друг компилятор добавляет вызов *super()* автоматически, если мы сами этого не сделали.**

Итак, компилятор *дважды* вовлечен в процесс создания конструкторов:

**1 Если вы *не создали* ни одного конструктора.**

Компилятор сам его добавит следующим образом:

```
public ClassName() {
      super();
}
```

**2 Если вы *предусмотрели* конструктор, но *не использовали* вызов super().**

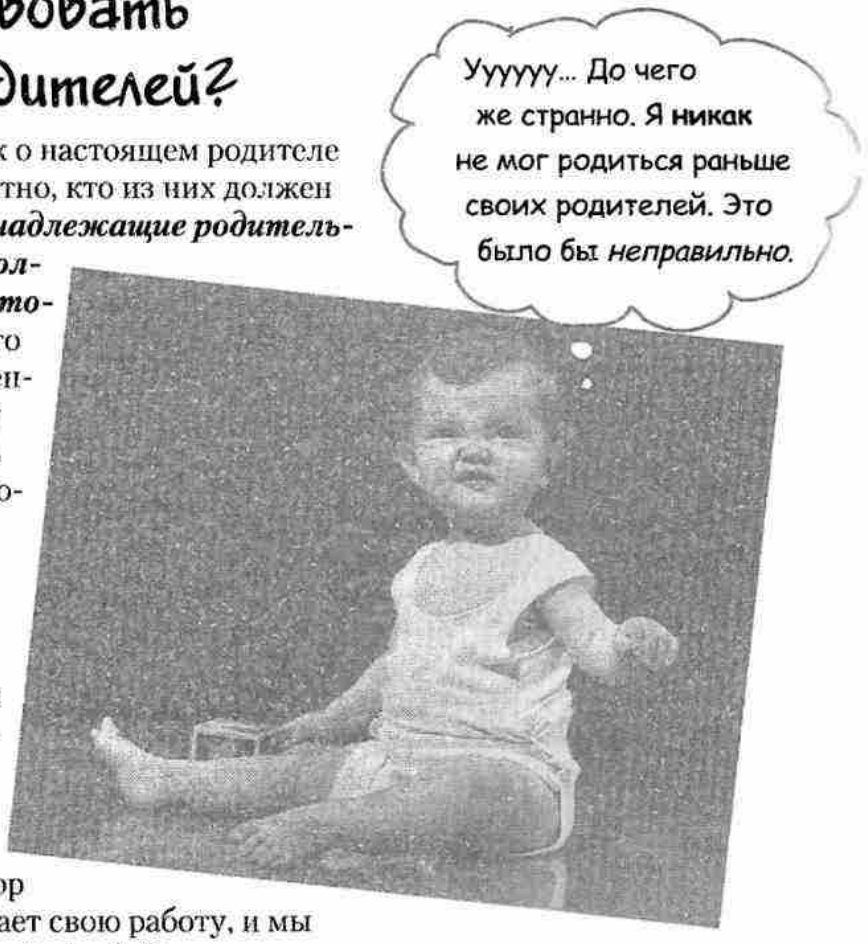
Компилятор поместит вызовы super() в каждый ваш перегруженный конструктор.* Эти вызовы всегда будут выглядеть так:

```
super();
```

Компилятор всегда добавляет вызовы super() без передачи аргументов. Если у родительского класса есть перегруженные конструкторы, то будет вызван лишь конструктор без аргументов.

* Если только конструктор не вызывает другой перегруженный конструктор (о чем идет речь, вы поймете через несколько страниц).

# Могут ли дети существовать до появления своих родителей?




Если подумать о родительском классе как о настоящем родителе своих дочерних классов, становится понятно, кто из них должен появиться первым. ***Части объекта, принадлежащие родительскому классу, будут сформированы (полностью созданы) до того, как будут готовы части дочернего класса.*** Помните, что дочерний объект может зависеть от элементов своего родительского класса, поэтому важно, чтобы они были завершены. Здесь все однозначно. Конструктор родительского класса должен завершить свою работу раньше, чем конструктор дочернего.

Взгляните еще раз на стековые последовательности на странице 278. Хотя конструктор класса Hippo вызван *первым* (он первый в стеке), завершается он в последнюю очередь! Конструктор каждого дочернего класса немедленно вызывает конструктор класса-родителя, пока на вершине стека не оказывается конструктор Object. Затем конструктор Object завершает свою работу, и мы опускаемся вниз по стеку до конструктора Animal. И только когда он завершается, мы наконец можем вернуться к выполнению конструктора Hippo. Рассмотрим, почему.

### Вызов super() должен быть в самом *начале* каждого конструктора!*

Возможные конструкторы для класса Boop

☑
```
public Boop() {
    super();
}
```
Это нормально, так как программист явно закодировал вызов super() в качестве начального выражения.

☑
```
public Boop(int i) {
    super();
    size = i;
}
```
Это нормально, так как компилятор сам помещает вызов super() в качестве начального выражения.

☑
```
public Boop() {
}
```

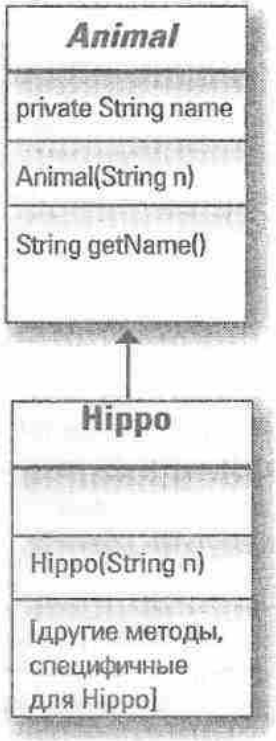
☑
```
public Boop(int i) {
    size = i;
}
```

⊘
```
public Boop(int i) {
    size = i;
    super();
}
```
Неправильно!!! Это не скомпилируется! Вы не можете разместить явный вызов super() после других выражений.

* Это исключение из правила. Подробности вы найдете на странице 282.

# Конструкторы родительских классов с аргументами

Как быть, если у конструктора родительского класса есть аргументы? Можно ли передать что-либо в вызов *super()*? Конечно. Без такой возможности вы бы никогда не смогли расширить класс, у которого нет конструктора без аргументов. Представьте следующую ситуацию: у всех животных есть имена (*name*). В классе Animal создан метод *getName()*, который возвращает значение переменной name. Переменная помечена модификатором private, но дочерний класс (в данном случае Hippo) наследует метод *getName()*. Возникает такая проблема: у Hippo есть метод *getName()* (благодаря наследованию), но нет переменной name. Класс Hippo должен зависеть от Animal, чтобы хранить переменную и возвращать ее значение, если кто-нибудь вызовет метод *getName()*. Но как часть этого класса, принадлежащая Animal, получает свое имя? Единственная связь между Hippo и Animal заключается в конструкции *super()*. Это место, куда Hippo передает имя, чтобы Animal мог сохранить его в виде переменной *name*.




```
public abstract class Animal {
   private String name;
   public String getName() {
      return name;
   }
   public Animal(String theName) {
      name = theName;
   }
}
```

Все потомки Animal (включая их дочерние классы) имеют имя.

Геттер, который унаследовал Hippo.

Конструктор, который принимает имя и присваивает его переменной экземпляра name.

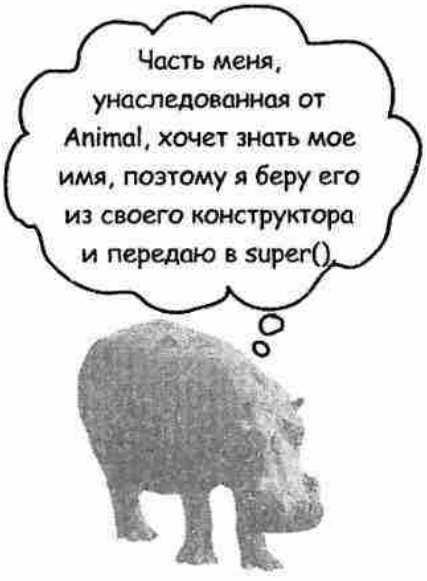
```
public class Hippo extends Animal {
   public Hippo(String name) {
      super(name);
   }
}
```

Конструктор Hippo принимает имя.

Он передает имя вверх по стеку в конструктор Animal.

```
public class MakeHippo {
   public static void main(String[] args) {
      Hippo h = new Hippo("Баффи");
      System.out.println(h.getName());
   }
}
```

Создаем Hippo, передаем его конструктору имя Баффи. Затем вызываем метод getName(), унаследованный классом Hippo.




```
File Edit Window Help Hide
%java MakeHippo
Баффи
```

# Вызов одного перегруженного конструктора из другого

Как быть с перегруженными конструкторами, которые различаются только набором аргументов? Вы знаете, что *дублировать* код в каждом конструкторе нежелательно (сложно поддерживать и т. д.). По этой причине лучше поместить некий объем кода (включая вызов super()) только в один из перегруженных конструкторов. Вы хотите, чтобы любой конструктор, который будет вызван первым, инициировал вызов настоящего конструктора и позволил ему завершить все работы по конструированию объекта? Это просто: нужно лишь написать *this()*. Или *this(aString)*. Или *this(27, x)*. Иными словами, думайте о ключевом слове *this* как о ссылке на ***текущий объект***.

Вы можете использовать выражение *this()* только внутри конструктора, и оно должно быть в самом его начале!

Возникает проблема, не так ли? Ранее мы говорили, что вызов *super()* должен размещаться в самом начале конструктора. Что ж, это означает, что у вас появился выбор.

*Используйте this() для вызова одного перегруженного конструктора из другого (в рамках одного класса).*

*Вызов this() может быть указан только в конструкторе и в виде его первого выражения.*

*Конструктор может иметь либо вызов this(), либо super(), но не оба сразу!*

**Каждый конструктор может иметь либо вызов this(), либо super(), но не оба сразу!**

```
class Mini extends Car {

    Color color;

    public Mini() {
        this(Color.Red);
    }

    public Mini(Color c) {
        super("Mini");
        color = c;
        // Дальнейшая инициализация
    }

    public Mini(int size) {
        this(Color.Red);
        super(size);
    }
}
```

Конструктор без аргументов предоставляет цвет по умолчанию и вызывает перегруженный настоящий конструктор (который вызывает super()).

Это настоящий конструктор, который делает большую работу, инициализируя объект (включая вызов super()).

Это не будет работать!!! Нельзя размещать в одном конструкторе и super(), и this(), потому что оба должны указываться в первом выражении!

```
File Edit Window Help Drive
javac Mini.java

Mini.java:16: call to super must
be first statement in constructor
        super(size);
        ^
```

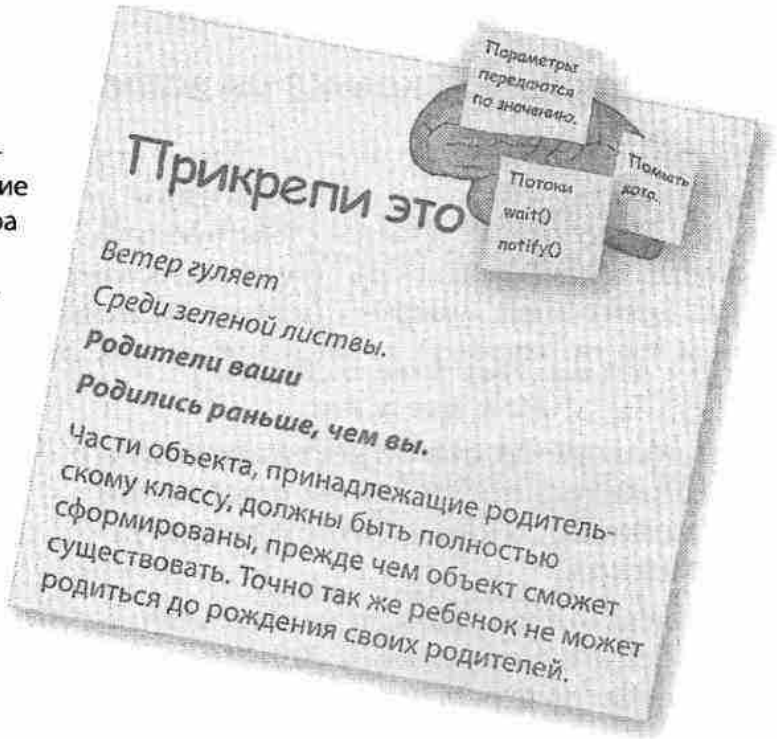
## Наточите свой карандаш

Некоторые конструкторы класса SonOfBoo не скомпилируются. Посмотрим, сможете ли вы определить, какие из них недопустимы. Сопоставьте ошибки компилятора с вызывающими их конструкторами класса SonOfBoo. Для этого соедините линиями ошибку с «плохим» конструктором.

```
public class Boo {
   public Boo(int i) { }
   public Boo(String s) { }
   public Boo(String s, int i) { }
}
```

---

```
class SonOfBoo extends Boo {

   public SonOfBoo() {
      super("boo");
   }

   public SonOfBoo(int i) {
      super("Fred");
   }

   public SonOfBoo(String s) {
      super(42);
   }

   public SonOfBoo(int i, String s) {
   }

   public SonOfBoo(String a, String b, String c) {
      super(a,b);
   }

   public SonOfBoo(int i, int j) {
      super("man", j);
   }

   public SonOfBoo(int i, int x, int y) {
      super(i, "star");
   }
}
```




```
File Edit Window Help
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(java.lang.String,java.
lang.String)
```

```
File Edit Window Help Yadayadayada
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(int,java.lang.String)
```

```
File Edit Window Help ImNotListening
%javac SonOfBoo.java
cannot resolve symbol
symbol:constructor Boo()
```

# Теперь мы знаем, как объект рождается. Но сколько он живет?

Долговечность *объекта* полностью зависит от продолжительности жизни связанных с ним ссылок. Если ссылка считается «действующей», то и объект будет продолжать свое существование в куче. Если ссылка «умирает» (и очень скоро мы узнаем, что это означает), то объект ожидает та же участь.

## Если долговечность объекта зависит от продолжительности жизни ссылочной переменной, то как долго существует *переменная*?

Это зависит от того, принадлежит переменная *классу* или является *локальной*. Код, приведенный ниже, демонстрирует продолжительность жизни локальной переменной. В примере это примитив, но то же самое происходит и со ссылочными переменными.

```
public class TestLifeOne {

   public void read() {
      int s = 42;
      sleep();
   }

   public void sleep() {
      s = 7;
   }
}
```

Переменная s находится в области видимости метода read(), поэтому не может быть использована за его пределами.

Неправильно! Нельзя использовать s здесь!

Метод sleep() не видит переменную s. Пока ее нет в стековом фрейме sleep(), этот метод ничего о ней не знает.

Переменная s продолжает существовать, но только в области видимости метода read(). Он сможет видеть ее после завершения метода sleep() и выхода на вершину стека. Когда read() закончит свою работу и выйдет из стека, s исчезнет. Еще одна смерть в цифровом мире.

**1 Локальная переменная обитает только внутри метода, в котором была объявлена.**

```
public void read() {
   int s = 42;
     // Переменная s может
     // быть использована
     // внутри этого метода.
     // Когда этот метод
     // завершается, s бесследно
     // исчезает.
}
```

Переменная *s* может быть использована только внутри метода *read()*. Иными словами, ***она находится в области видимости только своего метода***. Никакой другой код в классе (или за его пределами) не может видеть **s**.

**2 Переменная экземпляра существует до тех пор, пока существует объект. Если объект все еще «жив», то живы и его переменные.**

```
public class Life {
   int size;

   public void setSize(int s) {
      size = s;
        // Переменная s исчезает
        // по завершении этого
        // метода, но size может
        // быть использована
        // в любом месте класса
   }
}
```

Переменная **s** (на этот раз параметр метода) находится исключительно в области видимости метода setSize(). Но переменная экземпляра size существует, пока существует *объект* — она не зависит от *метода*.

# Разница между **существованием** и **видимостью** локальных переменных

### Существование

Локальные переменные *существуют* до тех пор, пока их фрейм находится в стеке. Другими словами, *пока метод не завершит свою работу.*

### Видимость

Локальные переменные находятся в области *видимости* только того метода, в котором были объявлены. Когда он вызывает другой метод, переменные продолжают существовать, но находятся за пределами видимости (пока метод не продолжит свою работу). ***Вы можете использовать переменную только тогда, когда она находится в текущей области видимости.***

```
public void doStuff() {
   boolean b = true;
   go(4);
}
public void go(int x) {
   int z = x + 24;
   crazy();
   // Представьте, что здесь
   // еще много кода
}
public void crazy() {
   char c = 'a';
}
```

1. **Метод *doStuff()* помещается в стек. Переменная 'b' существует и находится в области видимости.**

2. ***go()* «всплывает» на вершину стека. Переменные 'x' и 'z' существуют и находятся в области видимости; b существует, но *вне* области видимости.**

3. ***crazy()* выходит на вершину стека, а 'c' начинает свой жизненный цикл и попадает в область видимости. Остальные три переменные продолжают существовать, но за пределами области видимости.**

4. ***crazy()* завершает свою работу и покидает стек, поэтому 'c' уходит из области видимости и перестает существовать. Когда go() продолжит свою работу с того места, на котором она была прервана, 'x' и 'z' все еще будут существовать и опять станут видимыми. Переменная 'b' также существует, но ее не видно, пока go() не завершится.**

Пока локальная переменная существует, ее состояние сохраняется. Например, во время пребывания метода doStuff() на вершине стека переменная b хранит свое значение. Но она может быть использована только тогда, когда стековый фрейм метода doStuff() находится в самом верху. Иными словами, разрешается задействовать локальную переменную, *только* если ее метод в этот момент выполняется (в противном случае она ждет, пока завершатся все стековые фреймы, находящиеся выше).

# Что насчет ссылочных переменных?

Для примитивов и ссылок действуют одни и те же правила. Ссылочная переменная может быть использована, только если она находится в области видимости. Это означает, что нельзя управлять объектом, пока ссылка не вернется в текущую область видимости. *Главный* вопрос звучит так:

## Как жизнь переменной влияет на жизнь объекта

Объект живет до тех пор, пока существуют связанные с ним ссылки. Если ссылка перестает быть видимой, но все еще существует, то объект, на который она указывает, остается в куче. И вот здесь вам следует спросить: «Что происходит, когда фрейм, содержащий ссылку, покидает стек по завершении работы метода?»

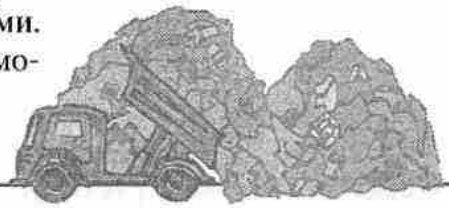
Если это была *единственная* живая ссылка на объект, то он делается недоступным в куче. Ссылочная переменная уничтожается вместе со стековым фреймом, поэтому о таком объекте можно официально забыть. Самое сложное — узнать, когда объект становится ***лакомым куском для сборщика мусора.***

Как только объект делается доступным для сборщика мусора, можно не беспокоиться об освобождении памяти, которую он занимал. Если вашей программе будет не хватать памяти, сборщик мусора удалит некоторые (или все) подходящие для этого объекты. Вы можете израсходовать всю память, но предварительно все подходящие объекты будут выброшены. Ваша цель — делать объекты недоступными (пригодными для сборщика мусора) после работы с ними. Если вы удерживаете объекты, то сборщик мусора не сможет ничем помочь, а ваша программа рискует погибнуть мучительной смертью, израсходовав всю память.

Жизнь объекта не стоит ломаного гроша, не имеет ни смысла, ни цели, пока на него кто-нибудь не сошлется.

Без этого вы не можете просить его что-нибудь сделать. Он будет лишь большой тратой драгоценных битов.

Но если объект недоступен, сборщик мусора узнает об этом. Рано или поздно такой объект исчезнет.



**Объект становится доступным для сборщика мусора, когда исчезает его последняя ссылка.**

**Три способа избавиться от ссылки на объект.**

1. Ссылка навсегда перестает быть видимой.

```
void go() {
  Life z = new Life();
}
```

Ссылка z исчезает в конце метода.

2. Ссылке присваивается другой объект.

```
Life z = new Life();
z = new Life();
```

Первый объект становится недоступным, когда z «перенастраивается» на новый объект.

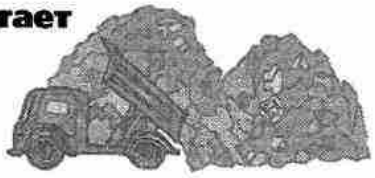
3. Ссылке явно присваивается null.

```
Life z = new Life();
z = null;
```

Первый объект становится недоступным, когда z «распрограммируется».

## Убийца объектов № 1

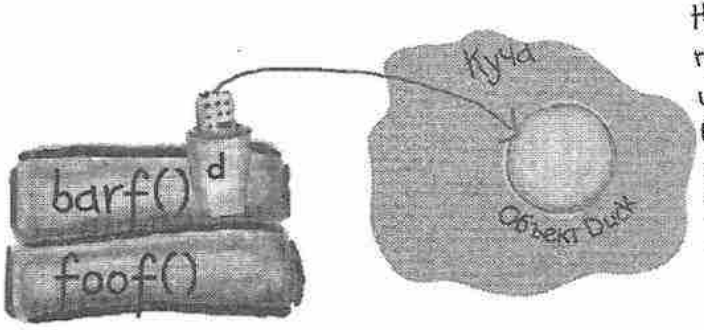
**Ссылка навсегда перестает быть видимой.**



```
public class StackRef  {
   public void foof() {
      barf();
   }

   public void barf() {
      Duck d = new Duck();
   }
}
```

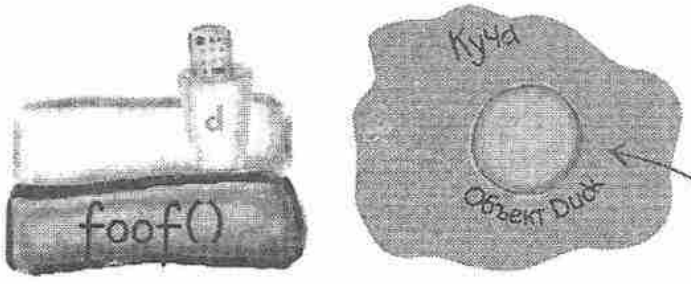
1. ***foof()*** **помещается в стек, переменные не объявляются.**

2. ***barf()*** **помещается в стек, где объявляется ссылочная переменная и новый объект, который с ней связывается. Объект создан в куче, а ссылка действительна и находится в области видимости.**




Новый объект Duck помещается в кучу, и, пока метод barf() выполняется, ссылка d работает и находится в области видимости, поэтому Duck считается «живым» (доступным).

3. ***barf()*** **завершает свою работу и покидает стек. Его фрейм разрушен, поэтому d исчезает. Выполнение программы возвращается к методу *foof()*, но он не использует ссылку d.**

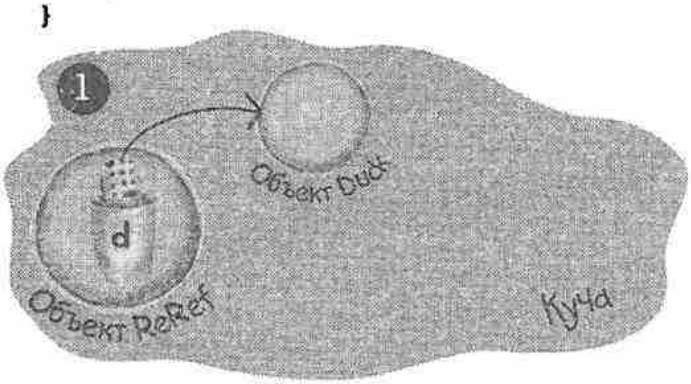



Ого! Переменная d исчезает, когда фрейм метода barf() покидает стек, поэтому объект Duck становится недоступным (и лакомым куском для сборщика мусора).

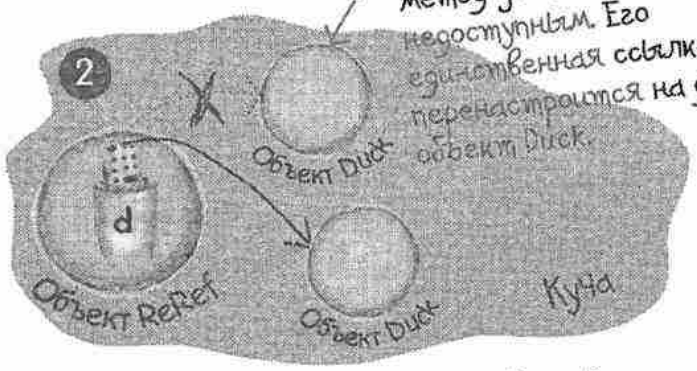
## Убийца объекта № 2

**Ссылке присваивается другой объект.**

```
public class ReRef {

    Duck d = new Duck();

    public void go() {
      d = new Duck();
    }
}
```




Новый объект Duck, на который ссылается d, отправляется в кучу. Поскольку d — переменная, Duck будет жить до тех пор, пока существует породивший его объект ReRef. Если только не...



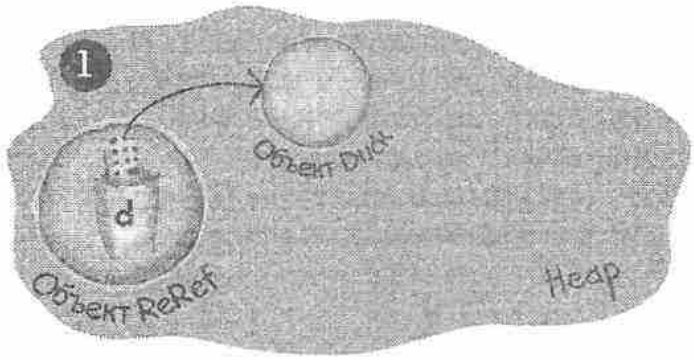

Когда кто-нибудь вызовет метод go(), Duck станет недоступным. Его единственная ссылка перенастроится на другой объект Duck.

Переменной d присваивается новый объект Duck, в результате чего первый объект становится недоступным.

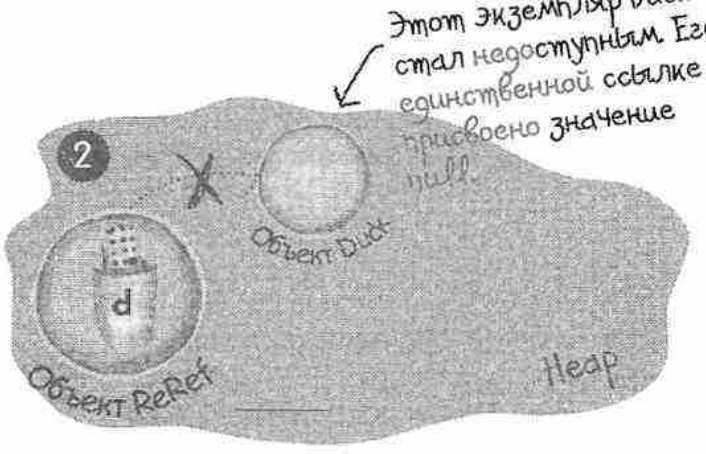
## Убийца объекта № 3

**Ссылке явно присваивается значение null.**

```
public class ReRef {

    Duck d = new Duck();

    public void go() {
      d = null;
    }
}
```




Новый объект Duck, на который ссылается d, отправляется в кучу. Поскольку d — переменная экземпляра, Duck будет жить до тех пор, пока существует породивший его объект ReRef. Если только не...




Переменной d присвоено значение null. Это то же самое, что получить пульт дистанционного управления, который ничем не управляет. Вам даже нельзя использовать оператор «точка» в сочетании с d, пока этот пульт не будет повторно запрограммирован (связан с объектом).

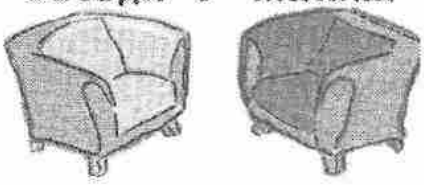
### Что такое null

Присваивая ссылке null, вы «сбрасываете» настройки пульта управления. Иными словами, вы получаете пульт, не управляющий телевизором. Нулевая ссылка содержит биты, представляющие значение null (мы не знаем, что это за биты, — за них отвечает JVM).

Имея такой пульт в реальной жизни, вы можете сколько угодно нажимать кнопки — это ни к чему не приведет. В Java нельзя нажимать кнопки (то есть использовать оператор доступа) в сочетании с нулевой ссылкой. В этом случае JVM знает (еще во время выполнения программы, а не на стадии ее компиляции), что, хотя вы и ожидаете услышать лай, у вас нет собаки!

**Если вы используете оператор «точка» в сочетании с нулевой ссылкой, то во время выполнения программы получите исключение NullPointerException.** Вся информация об исключениях приводится в главе 11.

## БЕСЕДА У КАМИНА



Сегодня в эфире: **Переменная экземпляра и локальная переменная обсуждают жизнь и смерть (демонстрируя небывалую вежливость).**

### Переменная экземпляра

Пожалуй, начнем с меня, так как я играю более важную роль для программы, чем локальная переменная. Мое предназначение — поддерживать объект (как правило, на протяжении всей его жизни). В конце концов, что такое объект без *состояния*? И что такое состояние? Это значения, хранящиеся в ***переменных экземпляра***.

Не поймите меня превратно — мне понятна роль, которую вы играете в методе, но ваша жизнь так коротка. Так мимолетна. Вот почему вас еще называют временными переменными.

Примите мои извинения. Я все понимаю.

---

Я никогда не думала об этом с такой точки зрения. Чем вы занимаетесь, когда работают другие методы и приходится ждать, пока ваш фрейм не поместят на вершину стека?

### Локальная переменная

Я ценю вашу точку зрения и, несомненно, осознаю важность состояния объекта, но мне не хотелось вводить зрителей в заблуждение. Локальные переменные *действительно* важны. Говоря вашими же словами: «В конце концов, что такое объект без *поведения*?» И что такое поведение? Алгоритмы, реализованные в методах. Можете не сомневаться, что без *локальных переменных* эти алгоритмы работать не будут.

В моей среде словосочетание «временная переменная» считается унизительным. Я предпочитаю такие синонимы, как «локальная», «стековая», «автоматическая», или выражение «с ограниченной видимостью».

Как бы то ни было, я и мои коллеги действительно долго не живем, да и жизнь эту нельзя назвать *хорошей*. Сначала нас помещают в стековый фрейм вместе с другими локальными переменными. И затем, если метод, к которому мы относимся, вызывает другой метод, то над нами появляется иной стековый фрейм. А если новый метод вызовет *еще один* метод... и т. д. Иногда приходится ждать целую вечность, прежде чем другие методы на вершине стека завершат свою работу, чтобы мы могли продолжить.

Абсолютно ничем. Это как анабиоз — спячка, в которую в фантастических фильмах впадают люди, чтобы путешествовать на большие расстояния. Мы будто на паузе, просто сидим в ожидании. Пока наш фрейм внизу стека, мы находимся в безопасности, но когда он опять продолжает работу, мы испытываем смешанные

**Переменная экземпляра**

Я когда-то видела учебный фильм на эту тему. Похоже, это довольно жестокий конец. Когда метод достигает своей закрывающей фигурной скобки, фрейм буквально *выбрасывается* из стека! Это *действительно* обидно.

Я обитаю в куче вместе с объектами. Точнее, не *с* объектами, а *в* объекте, чье состояние я храню. Стоит отметить, что в куче можно роскошно жить. Многие из нас испытывают чувство вины, особенно накануне праздников.

Гипотетически все происходит так: будучи переменной объекта Collar, отданного на съедение сборщику мусора, я, как и все переменные этого объекта, несомненно буду порезана на мелкие кусочки. Но меня уверили, что такого почти никогда не случается.

Они дадут нам *выпить*?

**Локальная переменная**

чувства. С одной стороны, мы опять становимся активными. С другой — мы неумолимо движемся к завершению своих коротких жизней. Чем дольше работает наш метод, тем ближе мы к его концу. *Все* мы знаем, что происходит потом.

И не говорите. В компьютерных науках используется термин *вытолкнул*: например, «фрейм был вытолкнут из стека». Как будто это развлечение или экстремальный спорт. Но вы сами видели, куда нас толкают. Лучше поговорим о вас. Я знаю, как выглядит мой скромный стековый фрейм, но где живете *вы*?

Но вы не *всегда* живете так же долго, как и объект, в котором объявлены, правильно? Допустим, есть объект Dog с переменной экземпляра *Collar*. Представьте, что *вы* — переменная Backle внутри объекта Collar. Вот вы удобно расположились внутри объекта *Collar*, который, в свою очередь, рад находиться внутри объекта *Dog*. Но что произойдет, если Dog захочет поменять переменную Collar или *обнулить* ее? Это сделает объект Collar доступным для сборщика мусора. Что же происходит с вами как переменной внутри объекта *Collar*, который стал недоступным?

И вы в это поверили? Они говорят так, чтобы улучшить вашу производительность и сохранить интерес к работе. Но вы кое о чем забываете. Что происходит с переменной экземпляра внутри объекта, на который ссылается только *одна локальная* переменная? Если я буду единственной ссылкой на объект, в котором вы пребываете, то ваша смерть последует сразу за моей. Нравится вам это или нет, наши судьбы могут быть связаны. Давайте же забудем обо всем и хорошенько напьемся, пока еще есть такая возможность. Как говорится, лови момент.

# Поработайте сборщиком мусора

Справа представлены строки кода, слева — класс с точкой A. Ваша задача — найти код, добавление которого в точку A приведет к тому, что один дополнительный объект станет доступным для сборщика мусора. Исходите из того, что точка A (там, где комментарий) будет выполняться достаточно долго, чтобы сборщик мусора успел сделать свою работу.

```
public class GC {
  public static GC doStuff() {
    GC newGC = new GC();
    doStuff2(newGC);
    return newGC;
  }

  public static void main(String [] args) {
    GC gc1;
    GC gc2 = new GC();
    GC gc3 = new GC();
    GC gc4 = gc3;
    gc1 = doStuff();
    A
    // Вызов других методов
  }

  public static void doStuff2(GC copyGC) {
    GC localGC
  }
}
```

1 `copyGC = null;`

2 `gc2 = null;`

3 `newGC = gc3;`

4 `gc1 = null;`

5 `newGC = null;`

6 `gc4 = null;`

7 `gc3 = gc2;`

8 `gc1 = gc4;`

9 `gc3 = null;`

Упражнение

# Популярные объекты

В этом примере создается несколько новых объектов. Ваша задача — найти «самый популярный» из них, то есть тот, с которым связано больше всего ссылок. Затем перечислите, *сколько* всего существует ссылок на этот объект и что это за ссылки! Один объект и ссылку мы берем на себя.

Удачи!

```
class Bees {
  Honey [] beeHA;
}

class Raccoon {
  Kit k;
  Honey rh;
}

class Kit {
  Honey kh;
}

class Bear {
  Honey hunny;
}

public class Honey {
  public static void main(String [] args) {
    Honey honeyPot = new Honey();
    Honey [] ha = {honeyPot, honeyPot, honeyPot, honeyPot};
    Bees b1 = new Bees();
    b1.beeHA = ha;
    Bear [] ba = new Bear[5];
    for (int x=0; x < 5; x++) {
      ba[x] = new Bear();
      ba[x].hunny = honeyPot;
    }
    Kit k = new Kit();
    k.kh = honeyPot;
    Raccoon r = new Raccoon();
    r.rh = honeyPot;
    r.k = k;
    k = null;
  }   //  Конец метода main
}
```

Вот новый объект Raccoon!

Это ссылочная переменная r.

«Мы запускали процесс симуляции четыре раза, и температура главного модуля постоянно отклонялась в сторону понижения, — раздраженно промолвила Сара.— Мы установили новые тепловые датчики на прошлой неделе. Показатели радиаторных машин, спроектированных для охлаждения жилых кварталов, похоже, находятся в пределах нормы, поэтому мы сосредоточились на анализе тепловых машин, которые помогают обогревать жилища».

Том вздохнул. Сначала казалось, что эти нанотехнологии действительно позволят им опередить график работ. Теперь же, когда осталось всего пять недель до запуска, некоторые ключевые системы поддержания жизни на орбитальном аппарате все еще не проходят испытания во время симуляции.

Пятиминутный детектив

«Какие пропорции вы моделируете?» — спросил Том.

«Если я правильно понимаю, к чему ты клонишь, то мы уже думали об этом, — ответила Сара. — Центр управления не одобрит критические системы, если мы запустим их не так, как положено. Необходимо, чтобы юниты v3 и v2, принадлежащие радиаторным машинам, работали в пропорции 2:1. В целом соотношение систем, удерживающих тепло, к радиаторным машинам должно составлять 4:3».

«Что с энергопотреблением, Сара?» — спросил Том. После небольшой паузы Сара ответила: «Это еще один интересный момент — энергии потребляется больше, чем ожидалось. Наша команда следит и за этим, но, поскольку у нас беспроводные технологии, сложно разделить энергию, потребляемую тепловыми и радиаторными машинами. В проекте предусмотрено, что общее соотношение будет равно 3:2, при этом радиаторы потребляют больше энергии из беспроводной сети».

«Хорошо, Сара, — сказал Том. — Давай взглянем на код, отвечающий за начало симуляции. Мы должны найти, в чем проблема, и сделать это быстро!»

```
import java.util.*;
class V2Radiator {
  V2Radiator(ArrayList list) {
    for(int x=0; x<5; x++) {
      list.add(new SimUnit("V2радиатор"));
    }
  }
}

class V3Radiator extends V2Radiator {
  V3Radiator(ArrayList lglist) {
    super(lglist);
    for(int g=0; g<10; g++) {
      lglist.add(new SimUnit("V3радиатор"));
    }
  }
}

class RetentionBot {
  RetentionBot(ArrayList rlist) {
    rlist.add(new SimUnit("Тепловая машина"));
  }
}
```

Продолжение пятиминутного детектива...

```
public class TestLifeSupportSim {
  public static void main(String [] args) {
    ArrayList aList = new ArrayList();
    V2Radiator v2 = new V2Radiator(aList);
    V3Radiator v3 = new V3Radiator(aList);
    for(int z=0; z<20; z++) {
      RetentionBot ret = new RetentionBot(aList);
    }
  }
}

class SimUnit {
  String botType;
  SimUnit(String type) {
    botType = type;
  }
  int powerUse() {
    if ("Тепловая машина".equals(botType)) {
      return 2;
    } else {
      return 4;
    }
  }
}
```

Том быстро просмотрел код, и по его губам скользнула странная, еле заметная улыбка. Он сказал: «Думаю, я нашел проблему, Сара. И еще я готов поспорить, что знаю, на сколько процентов отклонились показатели потребления энергии!»

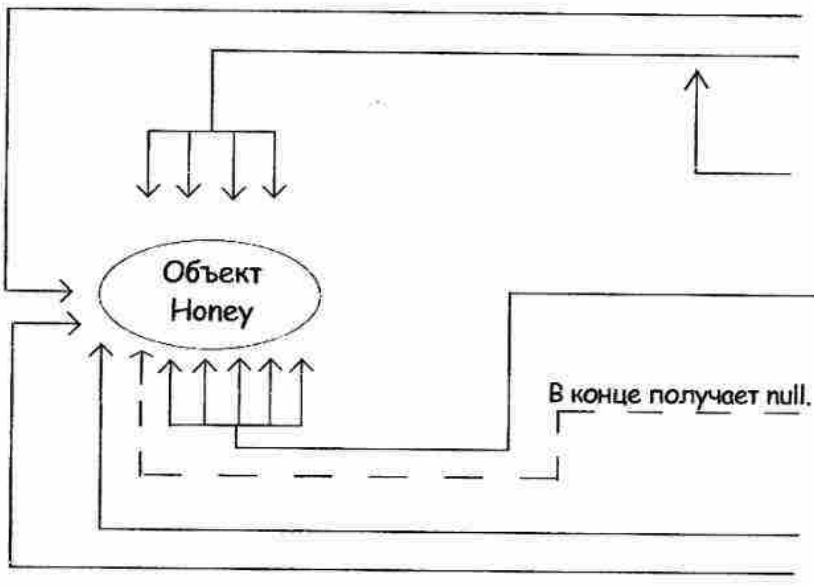
***Какие догадки появились у Тома? Как он нашел ошибку в показателях энергии и какие строки кода вы могли бы добавить, чтобы помочь починить эту программу?***

## Сборщик мусора

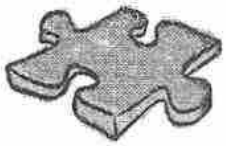
| | | |
|---|---|---|
| 1 | `copyGC = null;` | Нет — эта строка пытается получить доступ к переменной, которая находится вне текущей области видимости. |
| 2 | `gc2 = null;` | OK — gc2 была единственной переменной, ссылающейся на тот объект. |
| 3 | `newGC = gc3;` | Нет — еще одна переменная вне области видимости. |
| 4 | `gc1 = null;` | OK — gc1 была последней ссылкой, так как newGC находится вне области видимости. |
| 5 | `newGC = null;` | Нет — newGC вне области видимости. |
| 6 | `gc4 = null;` | Нет — gc3 все еще ссылается на тот объект. |
| 7 | `gc3 = gc2;` | Нет — gc4 все еще ссылается на тот объект. |
| 8 | `gc1 = gc4;` | OK — присваивание нового значения единственной ссылке на тот объект. |
| 9 | `gc3 = null;` | Нет — gc4 все еще ссылается на тот объект. |

## Популярные объекты

Вероятно, не так уж сложно понять, что объект Honey, на который сначала ссылалась переменная honeyPot, безусловно, наиболее «популярный» в классе. Немного сложнее увидеть, что все ссылки в коде, указывающие на него, ссылались на **один и тот же объект**! В самом конце метода main() их насчитывается 12 штук. Переменная *k.kh* доступна какое-то время, но в конце k получает значение null. Поскольку *r.k* по-прежнему ссылается на объект Kit, переменная *r.k.kh* (несмотря на то что она явно не объявлена) связана с объектом!




```
public class Honey {
  public static void main(String [] args
    Honey honeyPot = new Honey();
    Honey [] ha = {honeyPot, honeyPot,
                   honeyPot, honeyPot};
    Bees b1 = new Bees();
    b1.beeHA = ha;
    Bear [] ba = new Bear[5];
    for (int x=0; x < 5; x++) {
      ba[x] = new Bear();
      ba[x].hunny = honeyPot;
    }
    Kit k = new Kit();
    k.kh = honeyPot;
    Raccoon r = new Raccoon();

    r.rh = honeyPot;
    r.k = k;
    k = null;
  }  //  Конец main
}
```

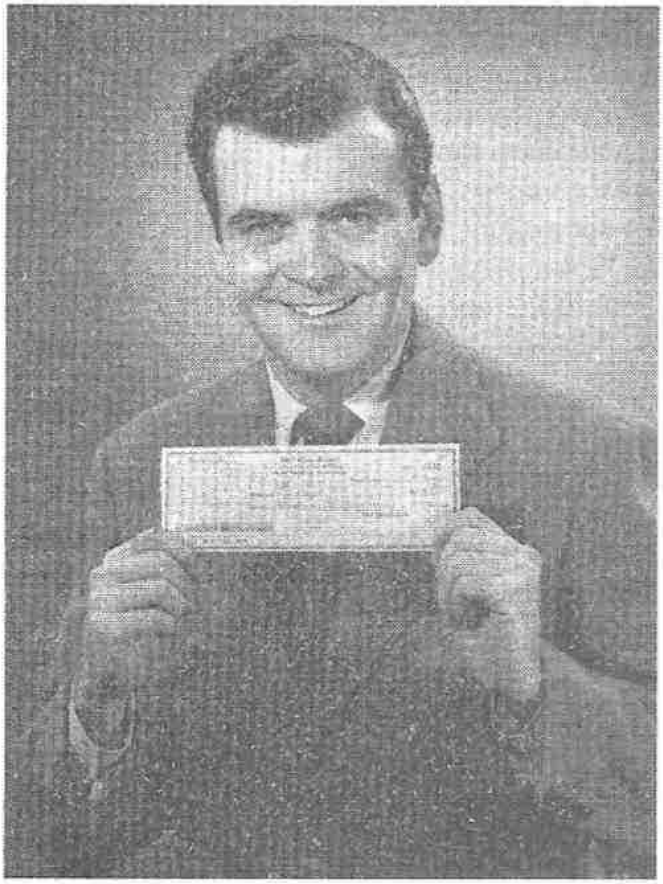


## Разгадка пятиминутного детектива

Том заметил, что конструктор класса V2Radiator принимает ArrayList. Это означает, что конструктор V3Radiator при каждом запуске передает ArrayList в конструктор V2Radiator, используя вызов super(). Таким образом, будут создаваться пять дополнительных объектов SimUnit. Если Том не ошибся, то итоговое энергопотребление будет равно 120, а не 100, как следовало из пропорций Сары.

Поскольку все машины создают объекты SimUnit, для выявления проблемы достаточно написать конструктор, который будет выводить на экран строку при создании каждого экземпляра!

# Числа имеют значение



**Поговорим о математике.** Ваша работа с цифрами будет выходить за рамки простой арифметики. Возможно, вам понадобится получить абсолютное значение числа, округлить его или найти большее из двух чисел. А если вы захотите напечатать числа с двумя знаками после запятой или разделить запятыми длинные числа, чтобы они легче воспринимались? И что насчет работы с датами? У вас может появиться желание выводить даты несколькими способами или даже *управлять* ими, чтобы иметь возможность, например, сказать: «Добавить три недели к текущей дате». И что насчет преобразования строки в число? Или превращения числа в строку? В Java API есть множество удобных и простых в использовании методов для работы с числами. Поскольку большинство из них **статические**, то сначала разберемся, какими особенностями обладают статические переменные и методы, включая константы, которые в Java считаются статическими *финализированными* переменными.

# Математические методы — наиболее близкие к глобальным

Однако в Java нет *ничего* глобального. Подумайте, что будет, если у вас есть метод, поведение которого не зависит от значения переменной экземпляра. Возьмем, к примеру, метод round() из класса Math. Он всегда делает одно и то же — округляет дробное число (которое передается в виде аргумента) до ближайшего целого. Если из 10 000 экземпляров класса Math вызвать метод round(42.2), результатом всегда будет целое число 42. Иными словами, метод зависит не от состояния переменной экземпляра, а от аргумента. Единственное значение, влияющее на работу метода round(), — это аргумент, который ему передается!

Возможно, создание экземпляров класса Math — это трата драгоценного места в куче, если его единственная цель — лишь запуск метода round()? И что насчет *других* методов из этого класса, например min(), который принимает два числовых примитива и возвращает наименьший из них? Или max(). Или abs(), возвращающий абсолютное значение числа.

***Эти методы никогда не используют значения переменных экземпляра.*** На самом деле класс Math вообще не содержит переменных, поэтому от создания его экземпляра нет никакой пользы. Но вам и не нужно этого делать. Более того, у вас это не получится.

Методы из класса Math не используют значения переменных экземпляра. И поскольку они статические, вам не нужно иметь экземпляр класса Math. Вам необходим лишь сам класс.

```
int x = Math.round(42.2);
int y = Math.min(56,12);
int z = Math.abs(-343);
```

Эти методы никогда не применяют переменные экземпляра, поэтому им ничего не нужно знать о конкретном объекте.

**Если вы попытаетесь создать экземпляр класса Math:**

```
Math mathObject = new Math();
```

**Вы получите следующую ошибку:**

```
File Edit Window Help IwasToldThereWouldBeNoMath
%javac TestMath

TestMath.java:3: Math() has private
access in java.lang.Math
    Math mathObject = new Math();
                      ^
1 error
```

Такая ошибка говорит о том, что конструктор класса Math помечен как приватный! Это означает, что вы никогда не сможете использовать оператор new в сочетании с этим классом, чтобы получить соответствующий объект.

# Разница между обычными (не статическими) и статическими методами

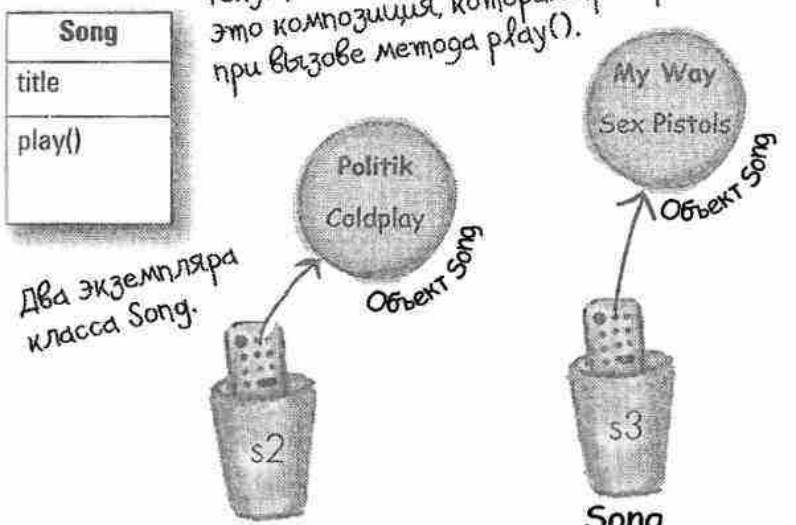
Java — объектно ориентированный язык, но иногда возникают ситуации, когда экземпляр класса не нужен (как в случае с методами из класса Math). Ключевое слово **static** позволяет методу работать ***без экземпляра класса***. Статический метод подразумевает, что его поведение не зависит от переменных экземпляра, поэтому нет необходимости в экземпляре/объекте. Нужен просто класс.

## Обычный (не статический) метод

```
public class Song {
   String title;
   public Song(String t) {
     title = t;
   }
   public void play() {
      SoundPlayer player = new SoundPlayer();
      player.playSound(title);
   }
```

Значение переменной экземпляра влияет на поведение метода play().

Текущее значение переменной title — это композиция, которая проигрывается при вызове метода play().




Два экземпляра класса Song.

```
Song
s2.play();
```

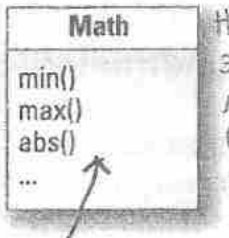
Вызов play() из этой ссылки приведет к воспроизведению песни «Politic».

```
Song
s3.play();
```

Вызов play() из этой ссылки приведет к воспроизведению песни «My Way».

## Статический метод

```
public static int min(int a, int b){
    //Возвращает наименьший
    // из аргументов a и b
}
```



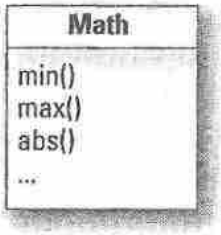

Никаких переменных экземпляра. Поведение метода не меняется вместе с состоянием переменной.

```
Math.min(42,36);
```

Используем имя класса вместо имени ссылочной переменной.

Никаких объектов! На этом рисунке нет абсолютно никаких объектов!

**Статический метод вызывается с помощью имени класса.**




```
Math.min(88,86);
```

**Обычный метод вызывается с помощью имени ссылочной переменной.**

```
Song t2 = new Song();
t2.play();
```

# Что значит иметь класс со статическими методами

Часто (хотя и не всегда) класс со статическими методами не предназначен для создания объектов. В главе 8 мы говорили об абстрактных классах и о том, как модификатор **abstract** делает невозможным использование оператора new в сочетании с типом класса. Иными словами, ***невозможно создать экземпляр абстрактного класса***.

Но вы можете запретить другому коду создавать экземпляры вашего не абстрактного класса, пометив конструктор как приватный. Помните, что приватный *метод* может вызывать только тот код, который находится с ним в одном классе. С приватным *конструктором* то же самое — его может вызывать только код из того же класса. Никто *за пределами* этого класса не сможет написать new. По такому принципу, например, работает класс Math. Его конструктор приватный — нельзя создать новый экземпляр класса Math. Компилятор знает, что ваш код не имеет доступа к этому конструктору.

Это вовсе *не* означает, что класс с одним или несколькими статическими методами не должен иметь экземпляров. На самом деле любой класс с методом main() — это класс со статическим методом!

Как правило, метод main() создается для запуска или тестирования другого класса. Часто в этом методе формируется экземпляр класса, из которого потом вызываются методы.

Можно сочетать в своем классе статические и обычные методы, хотя наличие даже одного не статического метода означает, что должен существовать способ создания экземпляра этого класса. Есть только два варианта: оператор new или десериализация (или нечто под названием Java Reflection API, останавливаться на котором мы не будем). Других путей нет. Но *кто* именно применяет оператор new — интересный вопрос, и мы его обсудим чуть позже в этой главе.

# Статические методы не могут использовать не статические переменные!

Статические методы работают, не зная о конкретных экземплярах своего класса. Как вы уже могли увидеть на предыдущей странице, у класса может даже не *быть* никаких экземпляров. Поскольку статические методы вызываются из класса (***Math***.random()), а не через *ссылку на объект* (*t2*.play()), они не могут использовать переменные класса. Статический метод не знает, в *каком* экземпляре хранится нужное ему значение переменной.

Если вы попытаетесь использовать переменную экземпляра внутри статического метода, то компилятор подумает: «Я понятия не имею, в каком объекте находиться переменная, о которой ты говоришь!» Если у вас в куче десять объектов Duck, то статический метод не будет ничего знать ни об одном из них.

## Если вы попытаетесь скомпилировать этот код:

```
public class Duck {

   private int size;

   public static void main (String[] args) {
      System.out.println("Размер утки равен" + size);
   }

   public void setSize(int s) {
      size = s;
   }
   public int getSize() {
       return size;
   }
}
```

Какая утка?

Чей размер?

Если где-нибудь в куче есть объект Duck, то мы об этом не знаем

## То получите следующую ошибку:

```
File Edit Window Help Quack
% javac Duck.java
Duck.java:6: non-static variable size cannot
be referenced from a static context
        System.out.println("Size of duck is " + size);
                                                ^
```

# Статические методы не могут использовать не статические методы!

Что делают не статические методы? ***Они обычно используют состояния переменных экземпляра, влияющих на их поведение.*** Метод getName() возвращает значение переменной name. Но чье это имя? Метод вызывается с помощью объекта.

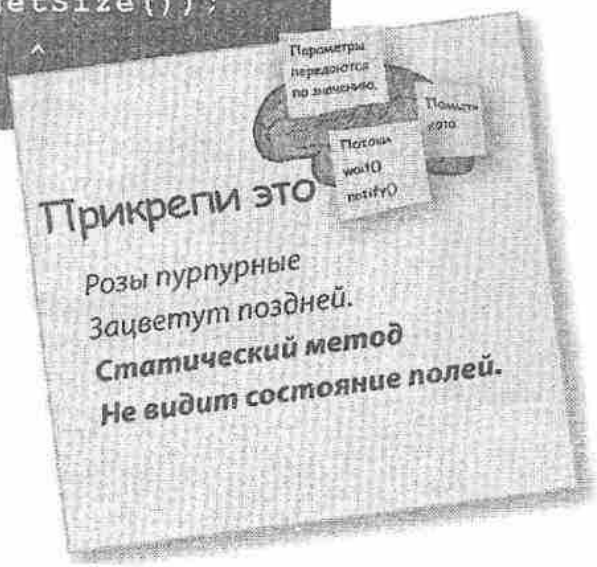
## Этот код не скомпилируется:

Вызов метода только откладывает неизбежное — getSize() использует переменную size.

```
public class Duck {

   private int size;

   public static void main (String[] args) {
      System.out.println("Размер равен" + getSize());
   }

   public void setSize(int s) {
      size = s;
   }
   public int getSize() {
       return size;
   }
}
```

Та же самая проблема... Чей размер?

```
File Edit Window Help Jack-in
% javac Duck.java
Duck.java:6: non-static method
getSize() cannot be referenced
from a static context
       System.out.println("Size
of duck is " + getSize());
                    ^
```




## Это не глупые вопросы

**В: Что будет, если из статического метода вызвать не статический, который не использует переменных экземпляра? Разрешит ли компилятор это сделать?**

О: Нет. Компилятор знает, что вы *можете* вводить переменные экземпляра в не статическом методе, даже если вы этого не делаете. Подумайте о последствиях... Вы позволите скомпилироваться такому коду, а что произойдет, если в будущем вы захотите изменить реализацию не статического метода, чтобы он *начал* использовать переменные экземпляра? И что случится, если дочерний класс *переопределит* метод и задействует в нем переменную экземпляра?

**В: Я уверен, что видел код, в котором статический метод вызывается из ссылочной переменной, а не через имя класса.**

О: Вы *можете* так делать, но, как говорится, не все хорошо, что разрешено. Хотя такой вызов и будет *работать*, он усложнит понимание кода. *Можете* написать так:

```
Duck d = new Duck();
String[] s = {};
d.main(s);
```

Такой код допустим, но компилятор просто заменит ссылку именем класса, решив, что d имеет тип Duck, а метод main() статический, поэтому можно вызвать его из класса Duck. Но использование ссылки d для вызова main() вовсе не означает, что этот метод знает хоть что-нибудь об объекте, на который ссылается d. Это просто еще один способ вызова статического метода, но сам метод остается статическим!

# Статическая переменная: одно и то же значение для всех экземпляров класса

Представьте, что вам захотелось посчитать, сколько всего объектов Duck было создано за время работы вашей программы. Как вы это сделаете? Может быть, с помощью переменной, которая будет инкрементироваться в конструкторе?

```
class Duck {
    int duckCount = 0;
     public Duck() {
         duckCount++;
    }
}
```

При каждом создании объекта Duck эта переменная будет получать значение 1.

Это не сработает, потому что duckCount — переменная экземпляра, и для каждого экземпляра Duck она будет иметь начальное значение 0. Вы можете вызывать метод из другого класса, но это еще больше запутает вас. Нужен класс с единственной общей копией переменной для всех своих экземпляров.

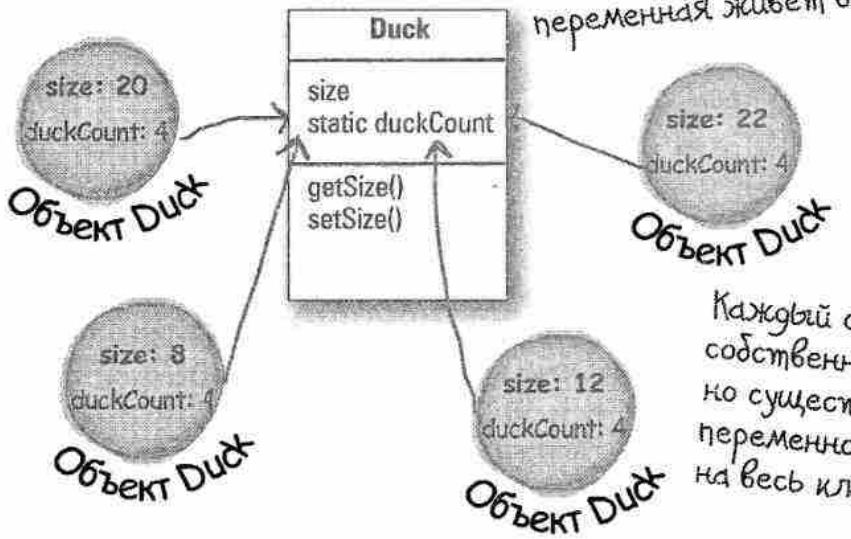
Имсннo это предоставляет вам статическая переменная: значение, общее для всех экземпляров *класса*, то есть одно значение на класс вместо одного значения на *экземпляр*.

Статическая переменная duckCount инициализируется только при первой загрузке класса, а не при каждом создании объекта.

```
public class Duck {

    private int size;
    private static int duckCount = 0;

    public Duck() {
        duckCount++;
    }

    public void setSize(int s) {
        size = s;
    }
    public int getSize() {
        return size;
    }
}
```

Теперь она будет продолжать инкрементироваться при каждом запуске конструктора Duck; duckCount — статическая переменная, которая не обнуляется.

Объект Duck не хранит свою копию переменной duckCount. Поскольку это статическая переменная, она будет доступна для всех объектов Duck в единственном экземпляре. Можете считать, что статическая переменная живет в классе, а не в объекте.




Каждый объект Duck содержит собственную переменную size, но существует лишь одна копия переменной duckCount — одна на весь класс.

***Статические переменные общедоступны.***

***Все экземпляры одного класса содержат общую копию статической переменной.***

Переменные экземпляра: по одной на **экземпляр**.

Статические переменные: по одной на **класс**.

## Штанга для мозга

Ранее в этой главе вы узнали, что если у класса есть приватный конструктор, то код за его пределами не может создавать экземпляры данного класса. Иными словами, только код внутри класса способен создать объект этого класса с помощью приватного конструктора (это похоже на проблему курицы и яйца).

Как быть, если нужно написать класс, из которого можно сделать только один объект, и все желающие использовать его экземпляр будут работать только с этим объектом?

# Инициализация статической переменной

**Все статические переменные в классе инициализируются до того, как станет возможным создать объект этого класса.**

Статические переменные инициализируются при *загрузке класса*. Он загружается в тот момент, когда JVM решит, что для этого пришло время. Как правило, JVM загружает класс, потому что кто-то впервые пытается создать его экземпляр или использовать его статические переменные/методы. Как программист, вы также можете попросить JVM загрузить класс, но вам, скорее всего, не придется этого делать. Обычно лучше положиться в этом на виртуальную машину.

У статической инициализации есть две особенности.

Статические переменные в классе инициализируются перед тем, как появится возможность создать *объект* этого класса.

Статические переменные инициализируются до того, как сможет быть запущен любой *статический метод* класса.

```
class Player {
    static int playerCount = 0;
    private String name;
    public Player(String n) {
        name = n;
        playerCount++;
    }
}

public class PlayerTestDrive {
    public static void main(String[] args) {
        System.out.println(Player.playerCount);
        Player one = new Player("Тайгер Вудс");
        System.out.println(Player.playerCount);
    }
}
```

Переменная playerCount инициализируется при загрузке класса. Мы явно присваиваем ей значение 0, но это необязательно, так как 0 — значение по умолчанию для переменных типа int. Статические переменные получают значения по умолчанию так же, как обычные.

Значения по умолчанию для объявленных, но не инициализированных статических и обычных переменных ничем не отличаются:

целые числа (long, short, etc.): 0

числа с плавающей точкой (float, double): 0.0

boolean: false

ссылки на объекты: null

Доступ к статической переменной получают так же, как и к статическому методу — через имя класса.

```
File Edit Window Help What?
% java PlayerTestDrive
0
1
```

Перед созданием любых экземпляров.

После создания объекта.

Статические переменные инициализируются при загрузке класса. Если вы не инициализируете их явно (присваивая им значения при объявлении), то они получат значения по умолчанию. Таким образом, целочисленным переменным будет присвоен 0, поэтому не нужно явно писать playerCount = 0. Объявление статических переменных без инициализации говорит о том, что они получат значения по умолчанию согласно своим типам по такому же принципу, что и обычные переменные экземпляра.

# Статические финализированные переменные представляют собой константы

Если переменная помечена модификатором final, то после инициализации она уже никогда не сможет измениться. Иначе говоря, значение статической финализированной переменной будет неизменным, пока класс загружен. Взгляните на Math.PI из API, и вы увидите:

```
public static final double PI = 3.141592653589793;
```

Переменная помечена как **public**, поэтому любой код может получить к ней доступ.

Переменная помечена как **static**, поэтому вам не нужен экземпляр класса Math (который, как вы помните, нельзя создать).

Переменная помечена как final, так как PI не меняется (по крайней мере в Java).

Нет другого способа объявлять переменные в качестве констант, но существует соглашение об именовании, которое поможет вам их распознать.

***Имена констант должны быть написаны в верхнем регистре!***

Статический инициализатор — блок кода, который запускается при загрузке класса, но перед тем, как кто-либо сможет использовать этот класс. Это отличное место для инициализации статических финализированных переменных.

```
class Foo {
    final static int x;
    static {
      x = 42;
    }
}
```

## Инициализируйте *финализированную* статическую переменную:

**1 В момент ее объявления:**

```
public class Foo {
   public static final int FOO_X = 25;
}
```

Обратите внимание на соглашение об именовании — статические финализированные переменные представляют собой константы, поэтому имя должно быть в верхнем регистре с символами подчеркивания, разделяющими слова.

**или**

**2 В статическом инициализаторе:**

```
public class Bar {
   public static final double BAR_SIGN;

   static {
      BAR_SIGN = (double) Math.random();
   }
}
```

Этот код запускается сразу после загрузки класса, перед вызовом любого статического метода и прежде, чем может быть использована любая статическая переменная.

**Если вы не присвоите значение финализированной переменной в одном из этих мест:**

```
public class Bar {
   public static final double BAR_SIGN;
}
```

Без инициализации!

**компилятор поймет это так:**

```
File Edit Window Help Jack-in
% javac Bar.java
Bar.java:1: variable BAR_SIGN
might not have been initialized
1 error
```

# Модификатор final подходит не только для статических переменных...

Ключевое слово final можно применять и для обозначения не статических переменных, включая переменные экземпляра, локальные переменные и даже параметры методов. И это всегда будет иметь один и тот же смысл: значение не может изменяться. Кроме того, разрешено использовать этот модификатор, чтобы запретить переопределение методов или создание дочерних классов.

### Не статические финализированные переменные

```
class Foof {
    final int size = 3;
    final int whuffie;

    Foof() {
       whuffie = 42;
    }

    void doStuff(final int x) {
       // Вы не можете изменить x
    }

    void doMore() {
       final int z = 7;
       // Вы не можете изменить z
    }
}
```

Теперь вы не можете менять размер.

Нельзя менять значение переменной whuffie.

### Финализированный метод

```
class Poof {
    final void calcWhuffie() {
       // Важные вещи, которые никогда
       // не должны быть переопределены
    }
}
```

### Финализированный класс

```
final class MyMostPerfectClass {
   // Не может быть наследован
}
```

***Вы не можете менять значение финализированной переменной.***

***Вы не можете переопределять финализированный метод.***

***Вы не можете расширять финализированный класс (то есть создавать дочерние классы).***

## Это не глупые вопросы

**В: Статический метод не имеет доступа к не статическим переменным. Но может ли не статический метод работать со статическими переменными?**

О: Конечно. Не статический метод всегда может вызвать статический метод из своего класса или получить доступ к статической переменной.

**В: Зачем делать класс финализированным? Не теряется ли при этом весь смысл ООП?**

О: И да и нет. Обычно класс финализируют исходя из вопросов безопасности. Например, нельзя наследовать класс String. Представьте, какими разрушительными будут последствия, если кто-нибудь создаст дочерний для String класс и с помощью полиморфизма использует его экземпляры там, где ожидается оригинальный String. Если вы хотите рассчитывать на конкретную реализацию методов в классе, сделайте его финализированным.

**В: Зачем финализировать метод, если класс уже финализирован?**

О: Если класс финализирован, то не нужно помечать методы словом final. Подумайте об этом — финализированный класс не может быть унаследован, поэтому ни один его метод нельзя переопределить.

С другой стороны, если вы *хотите* позволить другим классам наследовать ваш класс и переопределять некоторые его методы (но не все), то не помечайте класс модификатором final, а просто финализируйте каждый метод отдельно. Финализированный метод не может быть переопределен дочерним классом.

## КЛЮЧЕВЫЕ МОМЕНТЫ

- ***Статический метод*** должен вызываться с помощью имени класса, а не ссылки на объект:

```
Math.random() vs. myFoo.go()
```

- Для вызова статического метода необязательно наличие экземпляров класса в куче.
- Модификатор static хорошо подходит для служебных методов, которые не зависят от значений переменных конкретного объекта.
- Статические методы не связаны с конкретным экземпляром — только с классом, поэтому они не имеют доступа к любым не-статическим переменным своего класса. Они просто не знают, значения из *каких* экземпляров нужно использовать.
- Статический метод не имеет доступа к не статическому методу, так как последний, как правило, зависит от переменных экземпляра.
- Если ваш класс содержит исключительно статические методы и вы не хотите создавать его экземпляры, то можете сделать его конструктор приватным.
- ***Статическая переменная*** доступна для всех членов отдельного взятого класса. Существует только одна копия статической переменной — одна на весь класс, а не для каждого отдельного объекта.
- Статический метод имеет доступ к статическим переменным.
- Чтобы создать константу в Java, нужно пометить переменную сразу двумя модификаторами: static и final.
- Значение финализированной статической переменной должно быть присвоено либо во время объявления, либо при статической инициализации:

```
static {
      DOG_CODE = 420;
}
```

- Соглашение об именовании предусматривает, что имена всех констант (финализированные статические переменные) должны быть записаны в верхнем регистре.
- Значение финализированной переменной не может меняться после инициализации.
- Присваивать значение финализированной переменной экземпляра нужно либо во время ее объявления, либо при создании конструктора.
- Финализированный метод не может быть переопределен.
- Финализированный класс не может быть расширен (унаследован дочерними классами).

Наточите свой карандаш

## Что допустимо?

Учитывая все изученное о модификаторах static и final, подумайте, какой из этих фрагментов скомпилируется.

1
```
public class Foo {
   static int x;

   public void go() {
      System.out.println(x);
   }
}
```

2
```
public class Foo2 {
  int x;

  public static void go() {
      System.out.println(x);
  }
}
```

3
```
public class Foo3 {
  final int x;

  public void go() {
      System.out.println(x);
  }
}
```

4
```
public class Foo4 {
  static final int x = 12;

  public void go() {
      System.out.println(x);
  }
}
```

5
```
public class Foo5 {
  static final int x = 12;

  public void go(final int x) {
      System.out.println(x);
  }
}
```

6
```
public class Foo6 {
  int x = 12;

  public static void go(final int x) {
      System.out.println(x);
  }
}
```

# Математические методы

Теперь, зная, как работают статические методы, посмотрим на те (но не все) из них, что принадлежат классу Math. Чтобы ознакомиться с остальными методами, включая sqrt(), tan(), ceil(), floor() и asin(), загляните в API.

## Math.random()

Возвращает переменную типа double между 0.0 и 1.0 (не включительно).

```
double r1 = Math.random();
int r2 = (int) (Math.random() * 5);
```

## Math.abs()

Возвращает переменную типа double в виде абсолютного значения аргумента. Метод перегружен — если вы передадите int, то он вернет int. Передайте ему double — и он вернет double.

```
int x = Math.abs(-240);  // Возвращает 240
double d = Math.abs(240.45);  // Возвращает 240.45
```

## Math.round()

Возвращает значение типа int или long (в зависимости от типа аргумента — float или double), округленное до ближайшего целого.

```
int x = Math.round(-24.8f);  // Возвращает -25
int y = Math.round(24.45f);  // Возвращает 24
```

Помните, что тип значений с плавающей точкой по умолчанию рассматривается как double, если не добавить в конце символ «f».

## Math.min()

Возвращает наименьший из двух аргументов. Метод перегружен и может принимать значения типов int, long, float или double.

```
int x = Math.min(24,240);  // Возвращает 24
double y = Math.min(90876.5, 90876.49);  // Возвращает 90876.49
```

## Math.max()

Возвращает наибольший из двух аргументов. Метод перегружен и может принимать значения типов int, long, float или double.

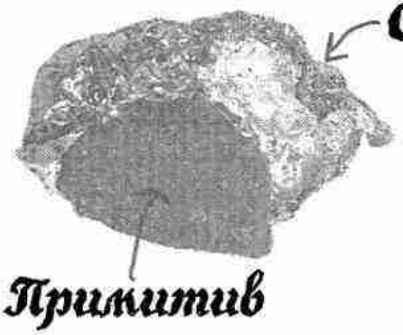
```
int x = Math.max(24,240);  // Возвращает 240
double y = Math.max(90876.5, 90876.49);  // Возвращает 90876.5
```

# Обертка для примитивов

Иногда нужно, чтобы примитив вел себя, как объект. Например, до версии 5.0 в Java нельзя было поместить переменную простого типа напрямую в такие коллекции, как ArrayList или HashMap:

```
int x = 32;
ArrayList list = new ArrayList();
list.add(x);
```

Это не будет работать в версиях Java ниже 5.0! В ArrayList нет метода add(), который принимает int! Метод add(), содержащий ArrayList, принимает ссылки на объекты, а не примитивы.




***Если вам нужно работать с примитивом, как с объектом, то применяйте обертку. Если вы пользуетесь Java версии ниже 5.0, то придется создавать обертку для добавления примитивов в такие коллекции, как ArrayList или HashMap.***

Существуют классы-обертки для каждого простого типа. Поскольку все они содержатся в пакете java.lang, вам не нужно их импортировать. Вы можете узнать их по именам, схожим с именами соответствующих типов (но первая буква соответствует соглашению об именовании классов).

Ах да, по причинам, не известным ни одному живому существу на этой планете, проектировщики API решили не связывать *напрямую* простые типы и соответствующие им классы. Вы поймете, что мы имеем в виду:

**Boolean**
**Character**
**Byte**
**Short**
**Integer**
**Long**
**Float**
**Double**

Будьте внимательны! Эти имена не имеют прямой связи с простыми типами. Имена классов записываются полностью, без сокращений.

**Упаковка значения:**

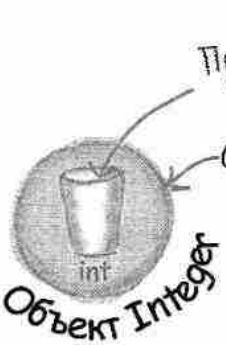
Передаем примитив в конструктор обертки.

```
int i = 288;
Integer iWrap = new Integer(i);
```

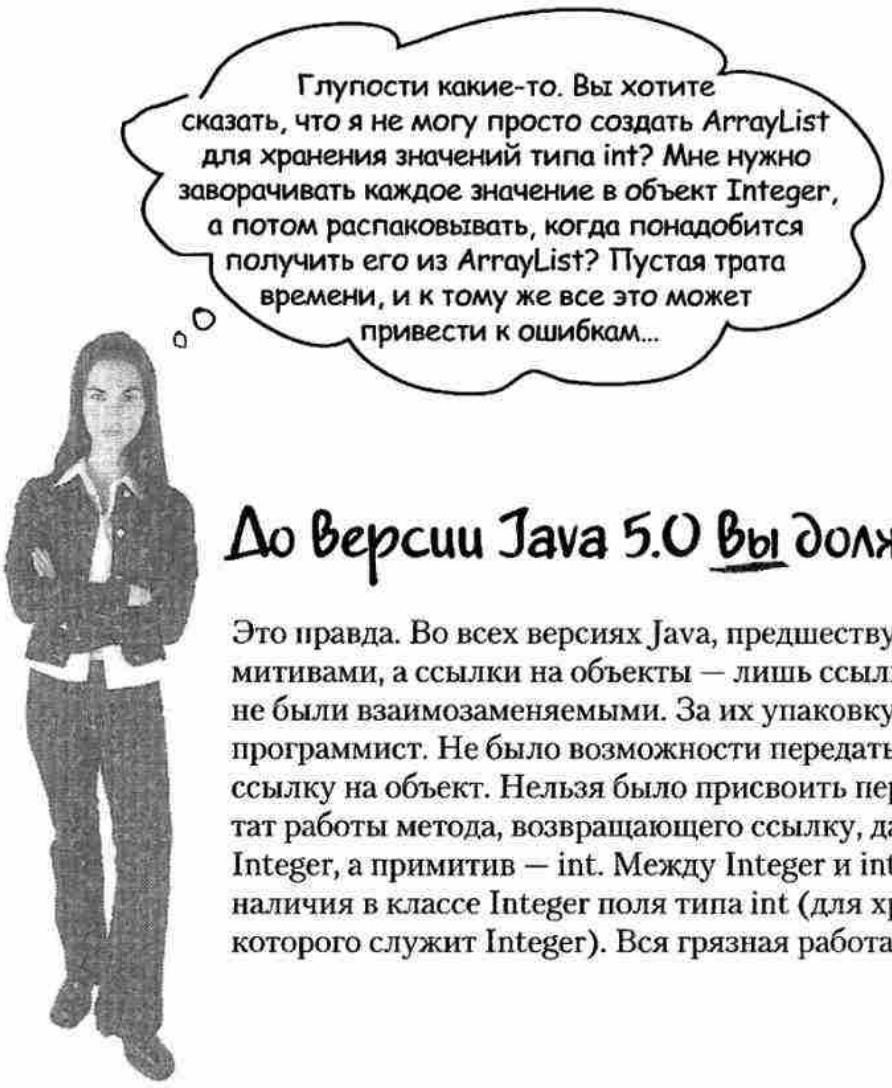
Все обертки работают по такому принципу. Boolean содержит метод booleanValue(), Character — charValue() и т. д.

***Распаковка значения:***

```
int unWrapped = iWrap.intValue();
```




Примечание: на рисунке выше показан шоколад в обертке из фольги. Некоторые люди предпочитают думать об этом, как о картошке в мундирах — и так тоже можно.




# До версии Java 5.0 вы должны были это делать

Это правда. Во всех версиях Java, предшествующих 5.0, примитивы были примитивами, а ссылки на объекты — лишь ссылками на объекты, и они никогда не были взаимозаменяемыми. За их упаковку и распаковку всегда отвечал программист. Не было возможности передать примитив в метод, ожидающий ссылку на объект. Нельзя было присвоить переменной простого типа результат работы метода, возвращающего ссылку, даже если эта ссылка имела тип Integer, а примитив — int. Между Integer и int не существовало связи, кроме наличия в классе Integer поля типа int (для хранения примитива, оберткой для которого служит Integer). Вся грязная работа доставалась разработчику.

## ArrayList для хранения значений типа int.

### Без автоматического упаковывания (версии Java до 5.0).

```
public void doNumsOldWay() {

    ArrayList listOfNumbers = new ArrayList();

    listOfNumbers.add(new Integer(3));

    Integer one = (Integer) listOfNumbers.get(0);

    int intOne = one.intValue();
}
```

Создаем ArrayList (помните, что до версии 5.0 в Java вы не могли указать тип, поэтому любой ArrayList был списком экземпляров Object).

Нельзя добавить в список простое значение 3, поэтому необходимо предварительно упаковать его в Integer.

Здесь возвращается значение типа Object, но вы можете привести его к Integer.

Наконец, вы можете получить примитив из объекта Integer.

# Автоматическая упаковка: стираем границы между примитивом и объектом

Благодаря новой упаковке, появившейся в Java 5.0, преобразование между примитивом и его оберткой происходит *автоматически*!

Посмотрим, что случится, если мы создадим ArrayList для хранения значений типа int.

## ArrayList для хранения значений типа int.

### С автоматической упаковкой (версии Java 5.0 и выше).

```
public void doNumsNewWay() {

    ArrayList<Integer> listOfNumbers = new ArrayList<Integer>();

    listOfNumbers.add(3);

    int num = listOfNumbers.get(0);
}
```

Создаем ArrayList типа Integer.

Просто добавляем!

Компилятор автоматически снимет обертку в виде объекта Integer (распакует примитив), поэтому можете присвоить значение типа int напрямую примитиву, не вызывая из Integer метод intValue().

Хотя в ArrayList нет метода add(int), компилятор сделает за вас всю работу по созданию обертки (упаковке). Иными словами, в ArrayList действительно хранится объект Integer, но вы должны сделать вид, будто в список добавляются значения типа int (в ArrayList<Integer> вы можете добавлять как Integer, так и int).

**В:** **Почему бы не объявить ArrayList<int>, если нужно хранить значения типа int?**

**О:** Потому что *так нельзя*. Вспомните правило обобщенных типов, согласно которому можно указывать только типы классов или интерфейсов, но *не примитивов*. По этому правилу ArrayList<int> не скомпилируется. Но, как понятно из примера выше, это не имеет особого значения, так как компилятор позволяет размещать значения типа int в списке ArrayList<Integer>. По сути, вам нельзя запретить помещать примитивы в ArrayList, тип которого — обертка для таких примитивов. Это работает для версий компиляторов, совместимых с Java 5.0 и предусматривающих автоматическую упаковку. Таким образом, вы можете помещать значения boolean в ArrayList<Boolean> и char в ArrayList<Character>.

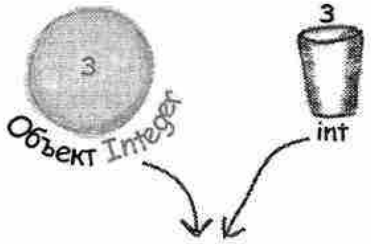
# Автоматическая упаковка работает практически везде

Автоматическая упаковка — это не просто создание обертки для примитивов и их извлечение для работы с коллекциями. Эта технология позволяет использовать примитивы везде, где подразумевается наличие их оберток (и наоборот). Подумайте об этом!

## Достоинства автоматической упаковки

### Аргументы методов

Если метод принимает тип-обертку, можно передавать ему как ссылку на объект, так и значение соответствующего простого типа. Обратное утверждение тоже верно — если метод принимает примитив, то можно передать ему либо значение совместимого с ним простого типа, либо ссылку на обертку.

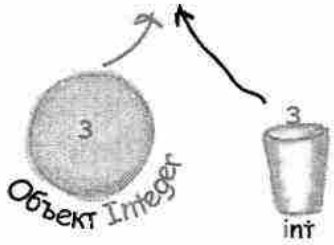



```
void takeNumber(Integer i) { }
```

### Возвращаемые значения

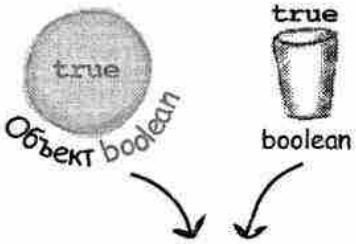
Если при объявлении метода указан простой тип возвращаемого значения, то можно возвращать как совместимый примитив, так и ссылку на обертку. И если при объявлении использовалась обертка, можно возвращать либо ссылку на нее, либо значение простого типа, которое ей соответствует.

```
int giveNumber() {
  return x;
}
```




### Булевы выражения

Везде, где ожидается булево значение, можно указывать либо выражение, которое возвращает boolean, либо булев примитив, либо ссылку на обертку Boolean.




```
if (bool) {
   System.out.println("true");
}
```

## Операции с числами

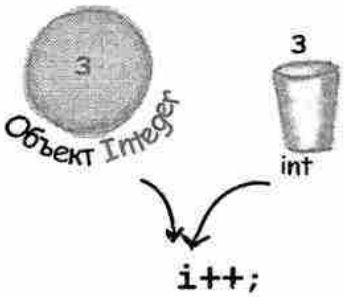
Это, вероятно, самый странный пункт — да, теперь можно использовать типы-обертки в качестве операнда там, где ожидается простой тип. Это означает, что вы можете применить, скажем, оператор инкремента к ссылке на объект Integer!

Но не волнуйтесь — это всего лишь ухищрения со стороны компилятора. Перед операцией он просто преобразует объект в значение простого типа. Хотя это, конечно, выглядит непривычно.

```
Integer i = new Integer(42);
i++;
```

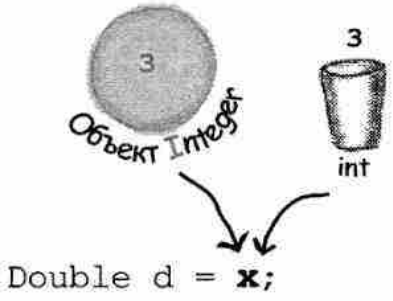
Таким образом, вы можете также делать что-то наподобие следующего:

```
Integer j = new Integer(5);
Integer k = j + 3;
```




## Операции присваивания

Можно присваивать переменной либо обертку, либо примитив, если эта переменная имеет соответствующий тип. К примеру, переменная простого типа int может быть присвоена ссылке на объект Integer; аналогично ссылка на объект Integer может быть присвоена переменной простого типа int.



### Наточите свой карандаш

Скомпилируется ли этот код? Запустится ли он? Если запустится, то что он будет делать?

Подумайте над вопросами — они касаются последствий автоматической упаковки, которые мы не затрагивали.

Чтобы найти ответы, вам придется поработать с компилятором (да, мы просим вас экспериментировать, но ради вашего же блага).

```
public class TestBox {

   Integer i;
   int j;

   public static void main (String[] args) {
      TestBox t = new TestBox();
      t.go();
   }

   public void go() {
      j=i;
      System.out.println(j);
      System.out.println(i);
   }
}
```

# Постойте! Это еще не все! Обертки также содержат статические служебные методы!

Помимо того что обертки ведут себя как обычные классы, они содержат набор полезных статических методов. Ранее мы уже использовали один из них — Integer.parseInt().

Метод принимает строку и возвращает значение простого типа.

**Преобразование объекта String в значение простого типа — это легко:**

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");

boolean b = new Boolean("true").booleanValue();
```

"2" легко превращается в 2.

Вы, наверное, думаете, что здесь должно быть Boolean.parseBoolean(). Но это не так. К счастью, есть конструктор Boolean, который принимает (и преобразовывает) строку, а при распаковке вы получаете значение простого типа.

**Если вы попытаетесь сделать следующее:**

```
String t = "два";
int y = Integer.parseInt(t);
```

Эх! Это скомпилируется без проблем, но при выполнении выдаст ошибку. Все, что не может быть разобрано в виде числа, приведет к появлению исключения NumberFormatException.

**Во время выполнения программы вы получите исключение:**

```
File Edit Window Help Clue
% java Wrappers
Exception in thread "main"
java.lang.NumberFormatException: two
at java.lang.Integer.parseInt(Integer.java:409)
at java.lang.Integer.parseInt(Integer.java:458)
at Wrappers.main(Wrappers.java:9)
```

**Любой метод или конструктор, который занимается разбором строк, может сгенерировать исключение NumberFormatException. Это происходит при выполнении программы, поэтому вы не обязаны его перехватывать или объявлять. Но все же лучше это сделать.**

Об исключениях мы поговорим в следующей главе.

# А теперь наоборот... превращение простого числового значения в строку

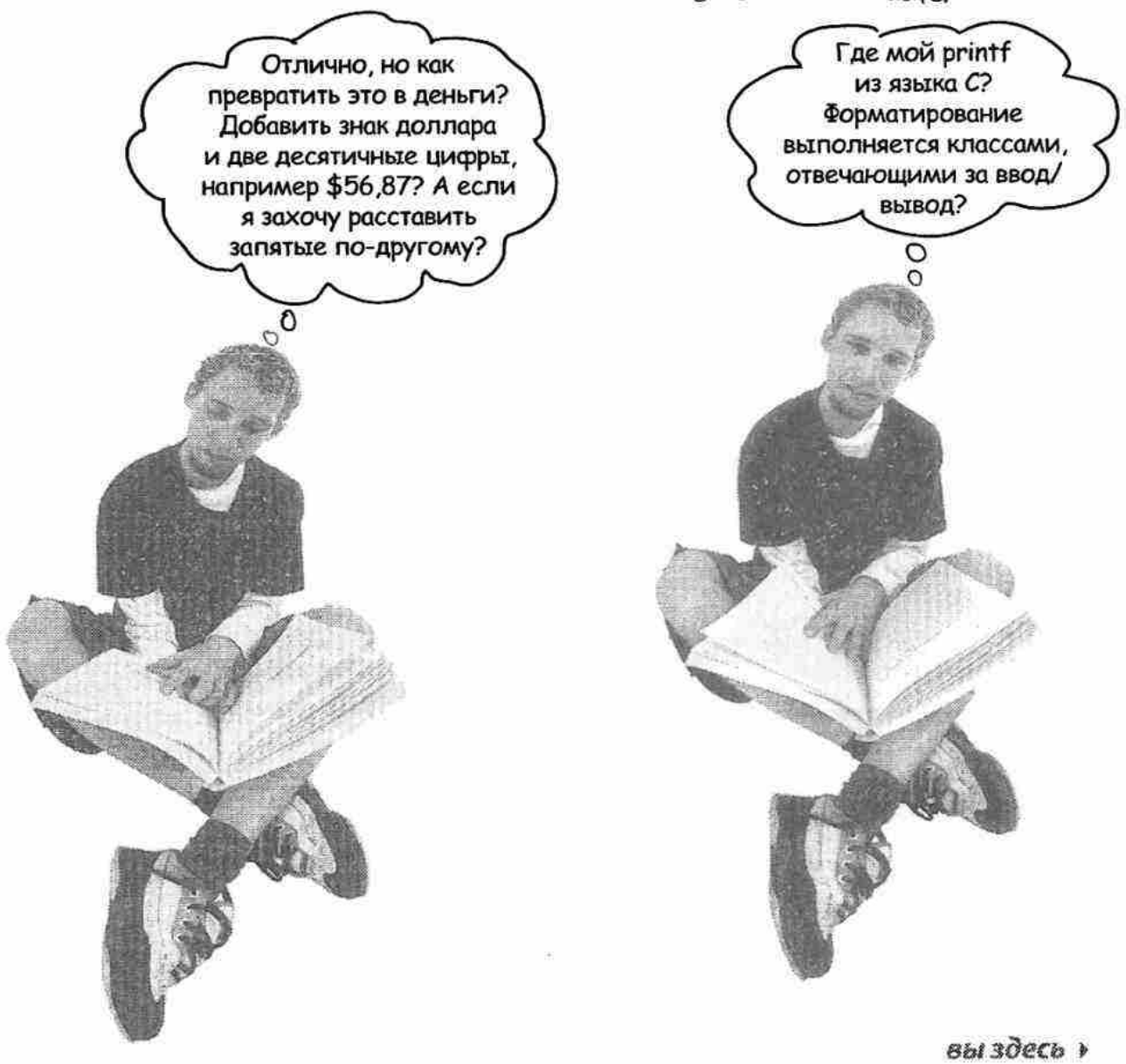
Существует несколько способов преобразовать число в объект String. Проще всего соединить его с уже существующей строкой.

```
double d = 42.5;
String doubleString = "" + d;
```

Помните, что оператор + в Java перегружен (единственный перегруженный оператор), чтобы иметь возможность соединять строки. Любое значение, добавленное к строке, тоже превращается в строку.

```
double d = 42.5;
String doubleString = Double.toString(d);
```

Еще один способ сделать это — использовать статический метод из класса Double.



# Форматирование чисел

Форматирование чисел и дат в Java не связано с вводом/выводом. Один из наиболее распространенных способов отображения чисел — вывод с помощью GUI (графического пользовательского интерфейса). Строки помещаются в прокручиваемую текстовую область или, возможно, в таблицу. Если бы форматирование было встроено только в методы вроде print, вы бы никогда не смогли располагать числа в виде изящных строк для последующего вывода в GUI. До Java 5.0 большинство операций форматирования были возложены на классы из пакета java.text, на который мы даже не взглянем в этом издании. Теперь все обстоит иначе.

В версии 5.0 команда разработчиков Java добавила более мощные и гибкие средства форматирования в виде класса Formatter из пакета java.util. Но не обязательно вручную вызывать его методы, так как в Java 5.0 такая возможность добавлена в некоторые классы ввода/вывода (включая printf()) и класс String. Достаточно вызвать статический метод String.format(), передав ему значение и инструкции по форматированию.

Естественно, вы должны знать, как записывать такие инструкции. Поначалу это может вызвать затруднения, особенно если вы не знакомы с функцией ***printf()*** из языков C/C++. К счастью, даже ничего *не* зная об этой функции, в большинстве простых случаев (которые будут рассмотрены в этой главе) вы будете лишь следовать некоторым правилам. Но если вы *захотите* смешивать и выравнивать значения *произвольным* образом, придется научиться работать с форматированием.

Начнем с простого примера и затем посмотрим на то, как он работает (мы опять вернемся к форматированию в главе о вводе/выводе).

### Форматирование числа с использованием запятых

```
public class TestFormats {

   public static void main (String[] args) {

      String s = String.format("%, d", 1000000000);
      System.out.println(s);
   }
}
```

Число, которое нужно отформатировать (мы хотим, чтобы оно содержало запятые — американский вариант представления чисел).

Инструкции, согласно которым форматируется второй аргумент (в нашем случае значение типа int). Помните, что этот метод принимает только два аргумента — первая запятая находится внутри строкового литерала, поэтому не разделяет аргументы метода format.

```
1,000,000,000
```

Теперь наше число разделено запятыми.

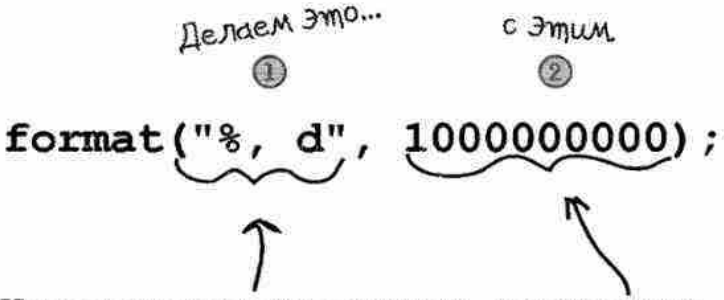
# Процесс форматирования под микроскопом...

На простейшем уровне форматирование состоит из двух главных частей (на самом деле их больше, но для пущей ясности мы начнем с этих).

1. **Инструкции форматирования.**
   Вы используете спецификаторы, которые описывают форматирование аргумента.

2. **Аргумент, который будет форматироваться.**
   Хотя аргументов может быть несколько, мы начнем с одного. Тип аргумента не *произвольный* — он должен быть совместим со спецификаторами в инструкциях форматирования. Например, если в ваших инструкциях указано *число с плавающей точкой*, нельзя передать методу format объект Dog или даже строку, которая выглядит, как дробное число.

Примечание: если вы уже знакомы с функцией printf() из C/C++, то можете пропустить следующие несколько страниц. В противном случае читайте внимательно!




**Используем эти инструкции... в сочетании с этим аргументом.**

### О чем на самом деле «говорят» такие инструкции?

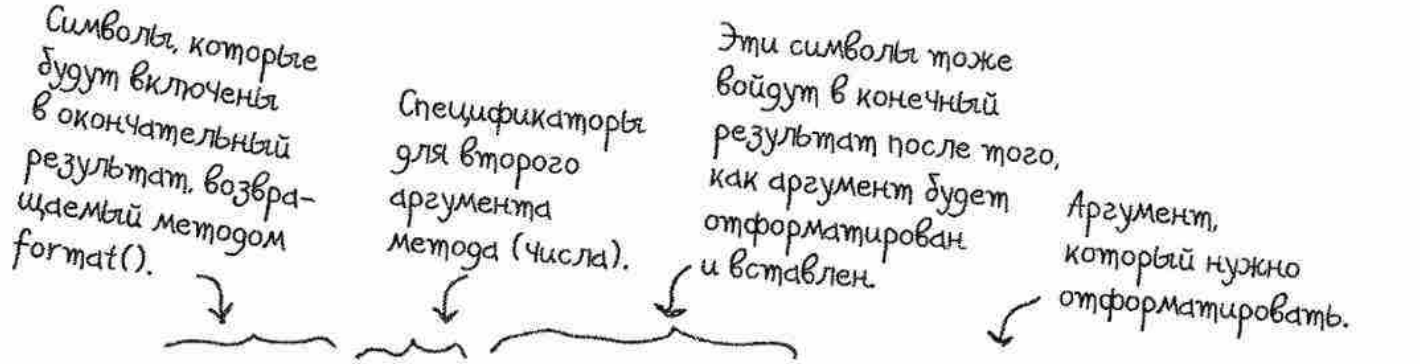
«Возьмем второй аргумент этого метода и отформатируем его в виде десятичного целого числа, разделенного **запятыми**».

### Как они это «говорят»?

В следующей главе мы более подробно рассмотрим значение синтаксиса % d. Если для вас это в новинку, представьте, что знак процента (%) в строке с инструкциями для форматирования (она всегда будет первым аргументом в методе format()) играет роль переменной, которая передается в виде второго аргумента. Символы после % описывают инструкции форматирования для аргумента.

# Знак процента (%) говорит: «Вставьте сюда аргумент и отформатируйте его по инструкции»

Первый аргумент метода format() — это формат строки. Он может содержать символы, которые вы хотите вывести без дополнительного форматирования. Думайте о знаке процента (%) как о переменной, которая представляет второй аргумент метода.




```
format("Мне нужно исправить %.2f ошибки.", 476578.09876);
```

Результат:

```
Мне нужно исправить 476578.10
```

Заметьте, что после десятичной точки потерялись некоторые цифры. Угадайте, что означает .2f?

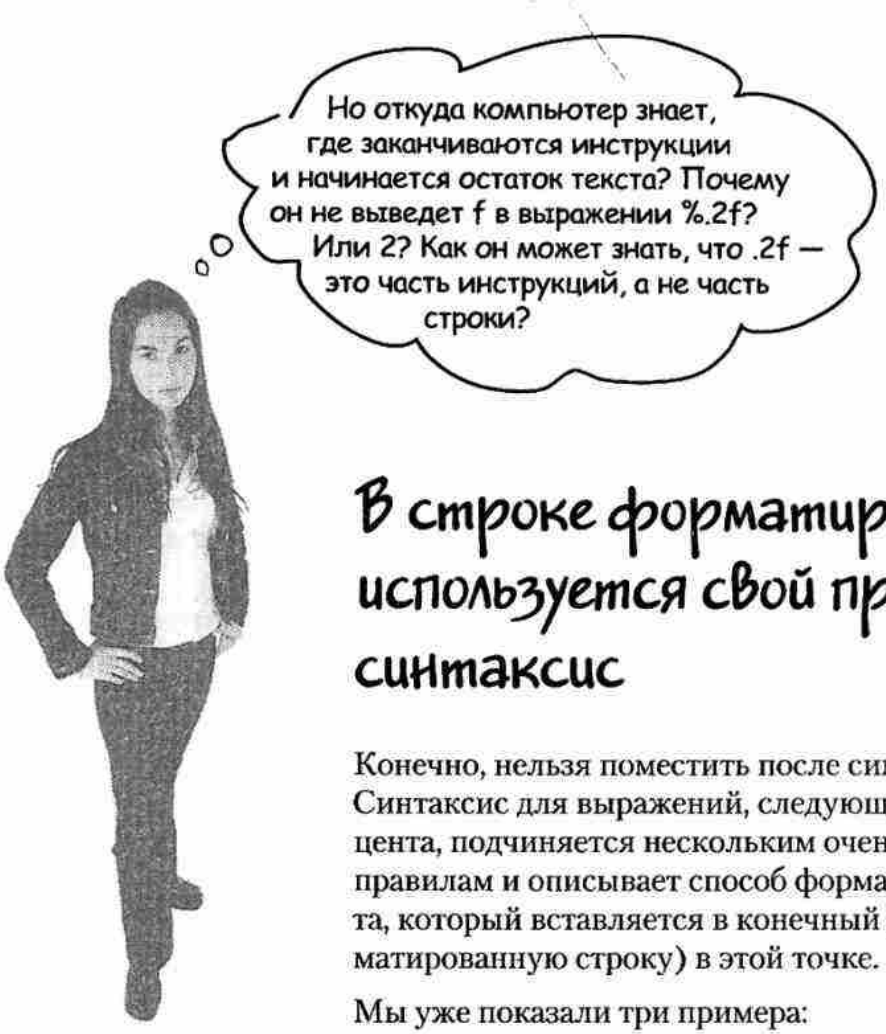
Знак % говорит методу format(), что нужно вставить второй аргумент (число) и отформатировать его, используя следующие за ним символы .2f. После этого в окончательный результат добавляется остаток строки форматирования: «ошибки».

### Добавление запятой

```
format("Мне нужно исправить %,.2f ошибки.", 476578.09876);
```

```
Мне нужно исправить 476,578.10
```

Изменив инструкции форматирования с %.2f на %,.2f, мы получили в числе запятые.




# В строке форматирования используется свой простой синтаксис

Конечно, нельзя поместить после символа % что угодно. Синтаксис для выражений, следующих за знаком процента, подчиняется нескольким очень специфическим правилам и описывает способ форматирования аргумента, который вставляется в конечный результат (в отформатированную строку) в этой точке.

Мы уже показали три примера:

**%,d** означает «вставляем запятые и форматируем число как десятичное целое».

и **%.2f** означает «форматируем число как дробное с двумя знаками после запятой».

и **%,.2f** означает «вставляем запятые и форматируем число как дробное с точностью два десятичных знака».

Как же понять, что именно размещать после символа процента, чтобы получить требуемый результат? Для этого нужно знать символы (такие как d для десятичных и f для чисел с плавающей точкой), а также порядок, в котором должны располагаться инструкции после знака процента. Например, если вы поместите запятую после d (%d, вместо %,d), это не сработает!

Или сработает? Как вы думаете, что выведет следующая строка?

```
String.format("Мне нужно исправить %,.2f, ошибки.", 476578.09876);
```

Ответ можно узнать на следующей странице.

# Спецификатор форматирования

Все, что находится между знаком процента и индикатором типа включительно (как d или f), — это часть инструкций по форматированию. Набор символов за индикатором типа рассматривается методом format() как часть выходной строки, пока не встретится очередной знак %. Хм... Разве это возможно? Разрешено ли при форматировании иметь несколько аргументов? Отложите эту мысль на потом; мы вернемся к ней через несколько минут. Сейчас посмотрим на синтаксис спецификаторов форматирования, следующих за знаком % и описывающих формат аргумента.

**Спецификатор форматирования может содержать до пяти разных частей (не включая %). Параметры в скобках [] необязательны, поэтому требуется указывать только знак процента и тип. Порядок следования тоже важен, поэтому размещайте все используемые части именно так.**

```
%[номер аргумента][флаги][ширина][.разрядность]тип
```

Мы обсудим это позже. Параметр позволяет указать конкретный аргумент, если их больше одного (пока не волнуйтесь об этом).

Параметр нужен для специальных видов форматирования, таких как вставка запятых, помещение отрицательных чисел в круглые скобки или выравнивание числа по левому краю.

Определяет минимальное количество используемых символов. Этот «минимум» не абсолютный. Если число длиннее, чем указано в спецификаторе, оно все равно будет указываться полностью, в ином случае оно будет выровнено нулями.

Об этом вы уже знаете — тут определяется разрядность. Иными словами, так задается количество знаков после запятой. Не забудьте добавить точку (.).

Тип обязателен (см. следующую страницу). Как правило, это d для десятичных или f для чисел с плавающей точкой.

%[номер аргумента][флаги][ширина][.разрядность]тип

```
format("%,6.1f", 42.000);
```

В этой строке форматирования не указан номер аргумента, но все остальные спецификаторы присутствуют.

# Единственный обязательный спецификатор предназначен для типа

Как вы поняли, обязателен для указания только тип, но не забывайте, что он всегда идет последним! Существует еще множество различных типов модификаторов (помимо даты и времени, у которых есть свой набор), но в большинстве случаев вы, вероятно, будете использовать %d (десятичные числа) или %f (с плавающей точкой). И, скорее всего, вы будете совмещать %f с индикатором разрядности, чтобы задать количество знаков после запятой, которые вы хотите вывести.

**Тип обязателен, все остальное опционально.**

**%d** ***Десятичное число***

```
format("%d", 42);
```

```
42
```

Значение 42.25 не получится! Это то же самое, что пытаться присвоить double переменной типа int.

Аргумент должен быть совместим с типом int, так что это могут быть только byte, short, int и char (или их типы-обертки).

---

**%f** ***С плавающей точкой***

```
format("%.3f", 42.000000);
```

```
42.000
```

Здесь мы совместили f с индикатором разрядности .3, поэтому в конце будет три нуля.

Аргумент должен иметь вещественный тип, то есть либо float, либо double (примитивы или обертки), либо нечто под названием BigDecimal (в этой книге не рассматривается).

---

**%x** ***Шестнадцатеричное число***

```
format("%x", 42);
```

```
2a
```

Аргумент должен принадлежать одному из следующих типов: byte, short, int, long (как примитивы, так и обертки) или BigInteger.

---

**%c** ***Символ***

```
format("%c", 42);
```

```
*
```

Число 42 превратится в *.

Аргумент должен иметь один из таких типов: byte, short, char, int (примитивы или обертки).

***Необходимо включать тип в инструкции форматирования. Сколько бы спецификаторов ни указывалось, тип всегда должен быть последним. В большинстве случаев вы, вероятно, будете использовать модификаторы d для десятичных чисел или f для чисел с плавающей точкой.***

# Что случится, если у меня будет несколько аргументов?

Допустим, вы хотите, чтобы ваша строка выглядела так:

«Уровень ***20,456,654*** из ***100,567,890.24***».

Но числа извлекаются из переменных. Что делать? Нужно просто добавить *два* аргумента после строки форматирования (первого аргумента). Это означает, что ваш вызов метода format содержит три аргумента вместо двух. И внутри первого аргумента (строки форматирования) указаны два разных спецификатора (которые начинаются с %). Первый спецификатор вставляет второй аргумент из метода, а второй спецификатор — третий аргумент. Иными словами, аргументы добавляются в строку форматирования в том порядке, в котором они переданы в метод format().

```
int one = 20456654;
double two = 100567890.248907;
String s = String.format(""Уровень %,d из %,.2f"", one, two);
```

```
Уровень 20,456,654 из 100,567,890.25
```

Мы добавили к обеим переменным запятые и сократили до двух количество знаков после точки.

Если у вас есть несколько аргументов, они добавляются в том порядке, в котором передаются в метод format().

Как вы еще сможете убедиться на примере даты, в работе часто придется использовать сразу несколько разных спецификаторов форматирования для одного аргумента. Наверное, это сложно представить, не зная, как работает форматирование *даты* (в отличие от форматирования *чисел*, которое мы рассмотрели). Через минуту вы научитесь более избирательно применять спецификаторы форматирования для каждого аргумента.

**В: Хм, здесь происходит что-то странное. Сколько же аргументов я могу передать? Иными словами, сколько перегруженных методов format() находится в классе String? Что случится, если я захочу передать, скажем, десять разных аргументов для форматирования в единственную строку?**

•

О: Да, здесь и правда *есть* кое-что странное (или по крайней мере новое и необычное), и дело не в наборе перегруженных методов format(), которые принимают разное количество аргументов. Чтобы поддерживать новый API для форматирования, язык Java нуждался в новой возможности — приеме *переменного количества аргументов*. Скорее всего, вы будете использовать эту особенность только при форматировании, поэтому мы рассмотрим ее в приложении.

# О числах поговорили. Что насчет дат?

Представьте, что вам нужна строка, которая выглядит так: «Воскресенье, Ноя 28 2004».

Ничего особенного, скажете вы? Тогда представьте, что все, от чего нужно отталкиваться, — это переменная типа Date и класс в Java, который может заменять временную метку. И вот вы хотите взять этот объект (вместо числа) и отправить его на форматирование.

Особенность форматирования даты заключается в использовании двусимвольного типа, начинающегося с *t* (в отличие от одинарных типов *f* или *d*). Следующий пример демонстрирует, как это работает:

**Полное представление даты и времени** **%tc**

```
String.format("%tc", new Date());
```

```
Вос Ноя 28 14:52:41 MST 2004
```

---

**Просто время** **%tr**

```
String.format("%tr", new Date());
```

```
03:01:47 PM
```

---

**День недели, месяц и число** **%tA %tB %td**

Не существует единственного спецификатора форматирования, который бы делал именно то, что мы хотим. Приходится комбинировать три из них, которые предназначены для дня недели (%tA), месяца (%tB) и дня месяца (%rd).

```
Date today = new Date();
String.format("%tA, %tB %td",today,today,today);
```

Запятая — это не часть форматирования, а просто символ, который мы хотим вывести после первого добавленного аргумента.

```
Воскресенье, Ноябрь 28
```

Но это означает, что мы должны передавать объект Date три раза, по одному для каждой части формата. Иными словами, %tA вернет лишь день недели, поэтому придется повторить это еще раз, чтобы получить месяц, и еще раз — для числа месяца.

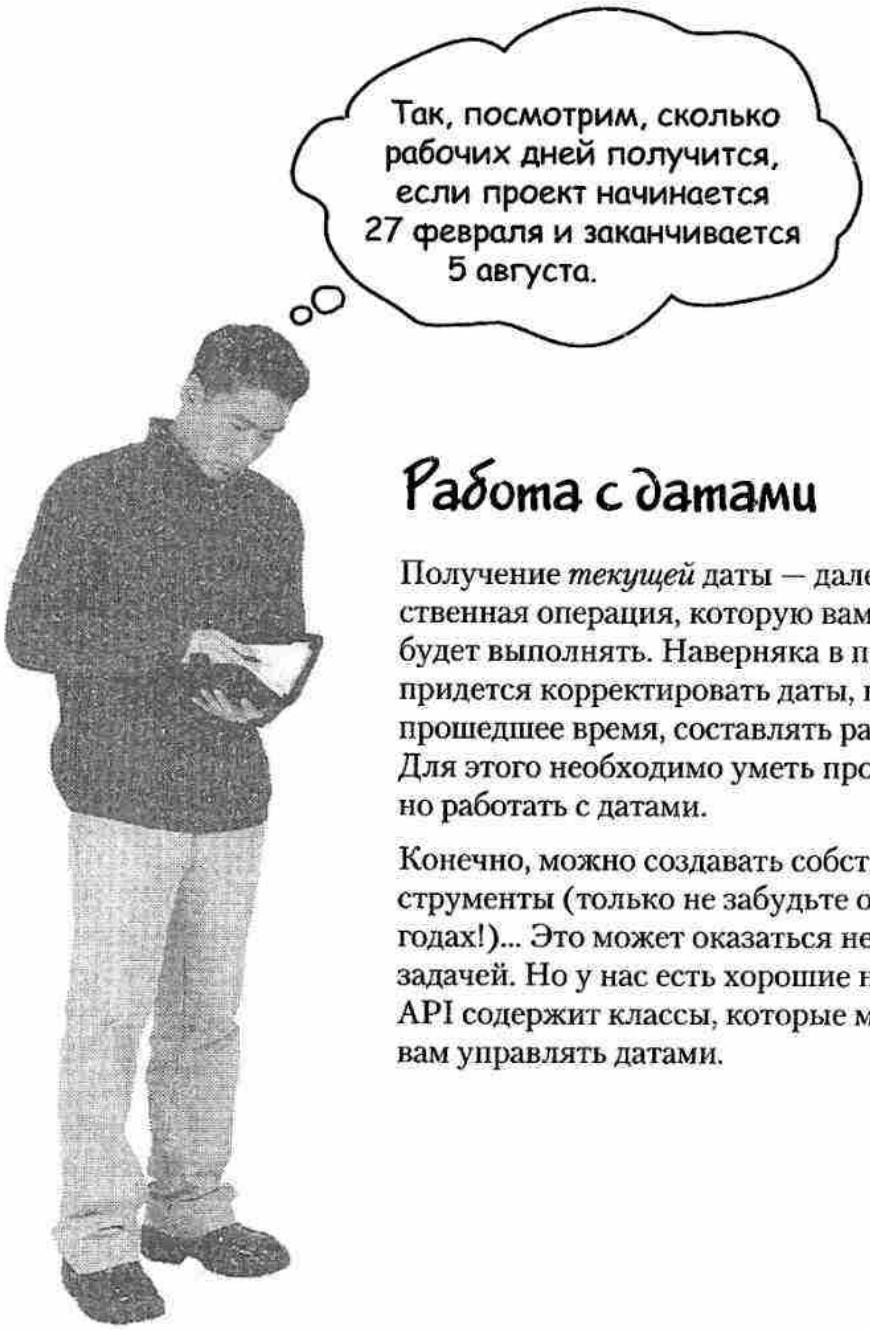
---

**Такой же код, что и выше, *но без дублирования аргументов*** **%tA %tB %td**

```
Date today = new Date();
String.format("%tA, %<tB %<td",today);
```

Можете думать об этом как о вызове трех разных геттеров из объекта Date с целью получить из него три части даты.

Угловая скобка (<) — это еще один флаг в спецификаторе, который говорит методу «используй предыдущий аргумент еще раз». Таким образом, вы избегаете повторения аргументов, форматируя единственный экземпляр тремя способами.




## Работа с датами

Получение *текущей* даты — далеко не единственная операция, которую вам нужно будет выполнять. Наверняка в программах придется корректировать даты, высчитывать прошедшее время, составлять расписание. Для этого необходимо уметь профессионально работать с датами.

Конечно, можно создавать собственные инструменты (только не забудьте о високосных годах!)... Это может оказаться непростой задачей. Но у нас есть хорошие новости: Java API содержит классы, которые могут помочь вам управлять датами.

# Перемещения во времени

Допустим, в вашей компании принята пятидневная рабочая неделя. И вам поручили составить список последних рабочих дней каждого календарного месяца в этом году...

*Для текущих временных отметок используйте класс Date. Для всего остального применяйте Calendar.*

### Похоже, что класс java.util.Date здесь не подойдет...

Ранее мы использовали java.util.Date для получения сегодняшней даты, поэтому будет логично начать с него поиски полезных инструментов для манипулирования датами. Но, взглянув на API, вы увидите, что большинство методов Date устарели!

Класс Date остается отличным средством для получения «временной отметки» — объекта, который представляет текущие дату и время. Используйте его, когда вам нужно вывести такое значение.

Теперь в API вместо Date рекомендуется применять класс **java.util.Calendar**. Посмотрим:

### Используйте java.util.Calendar для работы с датами.

Разработчики класса Calendar старались мыслить глобально. Главная идея заключается в том, что при необходимости работать с датами вы запрашиваете Calendar (с помощью статического метода, о котором пойдет речь на следующей странице), и JVM выдает его специфический дочерний класс (на самом деле Calendar — это абстрактный класс, поэтому вы всегда будете работать с его потомками).

Что еще более интересно, *тип* календаря, который вы получите, будет соответствовать вашим *региональным настройкам*. В большинстве стран используется григорианский календарь, но вы всегда можете найти библиотеки для работы с такими календарями, как буддистский, исламский или японский.

По умолчанию Java API поставляется с классом **java.util.GregorianCalendar**, поэтому мы будем использовать именно его. Хотя в большинстве случаев вам даже не придется думать о том, какой дочерний класс Calendar вы применяете. Это позволяет полностью сосредоточиться на методах класса.

# Получение объекта, расширяющего Calendar

Как на практике получить «экземпляр» абстрактного класса?

Никак, конечно же, поэтому следующий код не будет работать.

**Такое не сработает:**

```
Calendar cal = new Calendar();
```

Компилятор этого не позволит!

**Лучше используйте статический метод getInstance():**

```
Calendar cal = Calendar.getInstance();
```

Этот синтаксис должен быть вам знаком — мы вызываем статический метод.

## Вы не можете получить экземпляр Calendar, но можете получить экземпляр его конкретного дочернего класса.

Естественно, нельзя получить экземпляр класса Calendar, так как он абстрактный. Но у вас есть возможность вызывать из него *статические* методы, не имея конкретного объекта. Таким образом, вы вызываете из Calendar статический метод getInstance(), а он возвращает вам экземпляр конкретного дочернего класса, расширяющего Calendar (благодаря полиморфизму его можно присвоить ссылке типа Calendar) и способного отвечать на вызовы методов этого класса.

В большей части стран и по умолчанию в большинстве версий Java вы получите в ответ экземпляр **java.util.GregorianCalendar**.

# Работа с объектами Calendar

Есть несколько ключевых понятий, которые необходимо усвоить, чтобы начать работать с объектами Calendar.

- **Поля хранят *состояние*.** Объект Calendar включает множество полей, которые применяются для полноценного представления его состояния, даты и времени. К примеру, вы можете получать и устанавливать такие поля, как *year* и *month*.

- **Дата и время могут быть *инкрементированы*.** Класс Calendar содержит методы, которые позволяют добавлять значения к различным полям или вычитать из них. Например, можно добавить 1 к месяцу или вычесть три года.

- **Дату и время можно представить в *миллисекундах*.** Класс позволяет представлять дату в виде миллисекунд и наоборот (в частности, количество миллисекунд, которое прошло с первого января 1970 года). Это дает возможность выполнять точные вычисления разницы между двумя датами. Например, можно добавить к данному времени 63 часа 23 минуты и 12 секунд.

**Примеры работы с объектом Calendar:**

```
Calendar c = Calendar.getInstance();
c.set(2004,0,7,15,40);
long day1 = c.getTimeInMillis();
day1 += 1000 * 60 * 60;
c.setTimeInMillis(day1);
System.out.println("Новый час" + c.get(c.HOUR_OF_DAY));
c.add(c.DATE, 35);
System.out.println("Добавить 35 дней" + c.getTime());
c.roll(c.DATE, 35);
System.out.println("Прокрутить 35 дней" + c.getTime());
c.set(c.DATE, 1);
System.out.println("Установить дату в 1" + c.getTime());
```

Устанавливаем время Янв. 7, 2004 15:40 (заметьте, что месяцы начинаются с нуля).

Превращаем в большое количество миллисекунд.

Добавляем эквивалент часа в миллисекундах, затем обновляем время (обратите внимание, что += означает day1 = day1 + ...).

Добавляем к дате 35 дней, благодаря чему мы должны оказаться в феврале.

«Прокручиваем» дату на 35 дней. В результате добавляется 35 дней, но месяц не меняется!

Здесь мы не используем инкремент, а просто устанавливаем дату.

```
File Edit Window Help Time-Flies
Новый час 16
Добавить 35 дней Сред Фев 11 16:40:41 MST 2004
Прокрутить 35 дней Втор Фев 17 16:40:41 MST 2004
Установить дату в 1 Воск Фев 01 16:40:41 MST 2004
```

Этот результат подтверждает правильную работу методов.

# Основные методы класса Calendar

Мы уже упомянули некоторые поля и методы класса. Это большой API, поэтому мы рассмотрим только наиболее используемые поля и методы, которые вам могут пригодиться. В дальнейшем вы довольно легко сможете усвоить и остальные.

**Ключевые методы класса Calendar**

**add(int field, int amount)**
Добавляет или вычитает время из поля календаря.

**get(int field)**
Возвращает значение заданного поля календаря.

**getInstance()**
Возвращает экземпляр Calendar; можно задать региональные настройки.

**getTimeInMillis()**
Возвращает время в миллисекундах (тип long).

**roll(int field, boolean up)**
Добавляет или вычитает время без изменения старших полей.

**set(int field, int value)**
Устанавливает значения для заданного поля.

**set(year, month, day, hour, minute) (all ints)**
Популярная разновидность метода set для детальной установки времени.

**setTimeInMillis(long millis)**
Устанавливает время для объекта Calendar, основываясь на миллисекундах (тип long).

**// дальше...**

**Key Calendar Fields**

**DATE / DAY_OF_MONTH**
Получить/установить день месяца.

**HOUR / HOUR_OF_DAY**
Получить/установить значения часа (в 12- или 24-часовом формате).

**MILLISECOND**
Получить/установить миллисекунды.

**MINUTE**
Получить/установить минуты.

**MONTH**
Получить/установить месяц.

**YEAR**
Получить/установить год.

**ZONE_OFFSET**
Получить/установить сдвиг относительно GMT в миллисекундах.

**// дальше...**

# Больше статического! Статический импорт

Новинка, появившаяся в Java 5.0, вызывает смешанные чувства. Одним людям нравится эта идея, другие ее ненавидят. Статический импорт существует только для того, чтобы уменьшить объем набираемого разработчиком кода. Если вы не любите печатать, то оцените эту возможность. С другой стороны, при неосторожном использовании статический импорт может значительно затруднить чтение вашего кода.

Главная идея состоит в том, что при использовании статических классов, переменных или перечислений (об этом позже) можно их импортировать, не набирая часть кода.

*Будьте осторожны: статический импорт может усложнить ваш код.*

**Часть кода в старом стиле:**

```
import java.lang.Math;

class NoStatic {

  public static void main(String [] args) {

    System.out.println("sqrt " + Math.sqrt(2.0));

    System.out.println("tan " + Math.tan(60));

  }

}
```

Синтаксис, используемый при объявлении статического импорта.

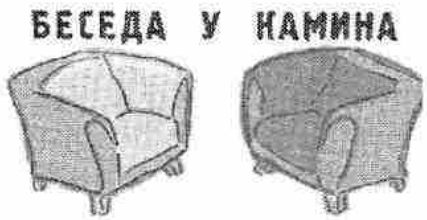
**Тот же код, но со статическим импортом:**

```
import static java.lang.System.out;

import static java.lang.Math.*;

class WithStatic {

  public static void main(String [] args) {

    out.println("sqrt " + sqrt(2.0));

    out.println("tan " + tan(60));

  }

}
```

Статический импорт в действии.

**Предостережения и подводные камни**

- Если вы намереваетесь использовать статические члены всего несколько раз, лучше воздержаться от статического импорта, чтобы не усложнять читабельность своего кода.
- Если вы собираетесь часто использовать статические члены (например, выполняя множество математических расчетов), то использование статического импорта, скорее всего, оправдано.
- Обратите внимание, что при статическом импорте вы можете использовать маски-звездочки (.*).
- Существенная проблема статического импорта заключается в том, что с его помощью легко создать конфликт имен. Например, у вас есть два класса с методами add(). Как компилятор узнает, какой из них нужно использовать?




Сегодня в эфире: **Обычная переменная делает колкие замечания в адрес своей статической коллеги.**

**Обычная переменная**

Я даже не представляю, зачем мы собрались. Всем известно, что статические переменные нужны только для констант. И сколько их всего? Думаю, весь API содержит... сколько, четыре? Да и то ими пользуются далеко не все.

Полон ими, конечно. Возможно, есть несколько штук в библиотеке Swing, но всем известно, что Swing — это особый случай.

Ладно, но помимо графического интерфейса приведите мне пример хотя бы одной статической переменной, которую используют все.

Это еще один особый случай. И, как бы то ни было, такая переменная нужна лишь для отладки.

**Статическая переменная**

Вам стоит перепроверить свои данные. Когда вы в последний раз заглядывали в API? Он просто наполнен статическими членами! Там даже есть целые классы, которые специально созданы для хранения констант. Например, есть класс SwingConstants, который полон ими.

Может это и особый случай, но он действительно важен! А что насчет класса Color? Представьте, что вам пришлось бы запоминать значения RGB для стандартных цветов. А в классе Color уже объявлены константы для синего, фиолетового, белого, красного и т. д. Это очень удобно.

Как насчет System.out? Здесь out — статическая переменная из класса System. Вы сами не создаете новый экземпляр этого класса, а просто запрашиваете у него переменную out.

А разве отладка не важна? И еще, признайтесь честно, что статические переменные более эффективны. Они существуют для целого класса, а не для отдельных экземпляров. Экономия памяти должна быть колоссальной!

**Обычная переменная**

Вам не кажется, что вы кое о чем забыли?

Статические переменные как таковые противоречат принципам ООП!! Почему бы нам заодно не сделать гигантский шаг назад и не вернуться к процедурному программированию?

Вы чем-то похожи на глобальные переменные, и любой уважающий себя программист, как правило, знает, что это плохо.

Да, вы живете внутри класса, но нет такого понятия, как *класс*-ориентированное программирование. Это просто нелепо. Вы — устаревшая модель. Мостик, с помощью которого программисты старой закалки переходят на Java.

Хорошо, если рассматривать каждый конкретный пример, то использование статических переменных имеет смысл. Но вот что я вам скажу: неправильное применение статических переменных (и методов) — признак неопытного ООП-программиста. Разработчик должен думать о состоянии *объекта*, а не *класса*.

Статические методы — худшее, что можно придумать. Они, как правило, указывают на то, что программист мыслит на процедурном уровне, вместо того чтобы думать о поведении объектов, зависящем от их уникального состояния.

А как же. Утешайте себя, сколько захотите...

**Статическая переменная**

О чем?

Что значит «*противоречат* ООП»?

Я не глобальная переменная. Я обитаю в классе! Это довольно объектно ориентированно, знаете ли. Я не парю где-то в пространстве; я естественная часть состояния объекта. Единственное отличие заключается в том, что я доступна для всех экземпляров класса. Это очень эффективно.

Здесь вам стоит остановиться. Это абсолютная неправда. Некоторые статические переменные имеют огромное значение для системы. Да и остальные весьма полезны.

Это еще почему? И что не так со статическими методами?

Конечно, я знаю, что объекты в ООП — центральное понятие. Но и для статических членов есть своя ниша. Когда они вам понадобятся, ничто не сможет их заменить.

# Поработайте компилятором

Java-код на этой странице представляет собой полноценную программу. Ваша задача — притвориться компилятором и определить, скомпилируется ли этот код. Если компиляция не сможет пройти успешно, как вы это исправите? И каков будет результат работы в ином случае?

```
class StaticSuper{

  static {
    System.out.println("Родительский статический блок");
  }

  StaticSuper{
    System.out.println(
      "Родительский конструктор");
  }
}

public class StaticTests extends StaticSuper {
  static int rand;

  static {
    rand = (int) (Math.random() * 6);
    System.out.println("Статический блок" + rand);
   }

   StaticTests() {
     System.out.println("Конструктор");
   }

  public static void main(String [] args) {
    System.out.println("Внутри main");
    StaticTests st = new StaticTests();
  }
}
```

Если код скомпилируется, что появится на экране?

**Возможный результат:**

```
File Edit Window Help Cling
%java StaticTests
Статический блок 4
Внутри main
Родительский статический блок
Родительский конструктор
Конструктор
```

**Возможный результат:**

```
File Edit Window Help Electricity
%java StaticTests
Родительский статический блок
Статический блок 3
Внутри main
Родительский конструктор
Конструктор
```

Эта глава открыла для вас чудесный статический мир Java. Ваша задача — определить, какие из следующих выражений правдивы, а какие — нет.

## Правда или ложь

1. Первое, что нужно сделать при использовании класса Math, — создать его экземпляр.
2. Вы можете пометить конструктор ключевым словом static.
3. Статические методы не имеют доступа к состоянию переменных объекта через ключевое слово 'this'.
4. Для вызова статических методов рекомендуется применять ссылочные переменные.
5. Статические переменные могут использоваться для подсчета экземпляров класса.
6. Конструкторы вызываются до инициализации статических переменных.
7. MAX_SIZE — подходящее имя для статической финализированной переменной.
8. Статический инициализатор запускается раньше конструктора класса.
9. Если класс помечен модификатором final, то все его методы должны быть финализированы.
10. Финализированный метод может быть переопределен только тогда, когда его класс унаследован.
11. Для типа boolean не существует класса-обертки.
12. Обертка применяется в тех случаях, когда примитив должен притвориться объектом.
13. Методы parseXxx всегда возвращают String.
14. Классы для форматирования (которые раньше относились к вводу/выводу) находятся в пакете java.format.

# Лунные магнитики с кодом

Это может быть полезно! Вы уже узнали, как работать с датами, а теперь потренируйтесь в этом. Итак, нам известно, что полнолуние бывает раз в 29.52 дня или около того. Полная луна была 7 января 2004 года. Необходимо переставить фрагменты кода так, чтобы получилась рабочая Java-программа, которая выведет на экран следующий текст (плюс еще больше дат, когда было полнолуние). Возможно, вам понадобятся не все магнитики. При необходимости можете добавлять фигурные скобки. Кстати, ваш результат работы программы будет отличаться от нашего, если вы живете в другом часовом поясе.

```
long day1 = c.getTimeInMillis();
c.set(2004,1,7,15,40);
import static java.lang.System.out;
static int DAY_IM = 60 * 60 * 24;
("Полнолуние было в %tc", c));
(c.format
Calendar c = new Calendar();
class FullMoons {
public static void main(String [] args) {
day1 += (DAY_IM * 29.52);
for (int x = 0; x < 60; x++) {
static int DAY_IM = 1000 * 60 * 60 * 24;
println
("full moon on %t", c));
import java.io.*;
import java.util.*;
static import java.lang.System.out;
c.set(2004,0,7,15,40);
out.println
c.setTimeInMillis(day1);
(String.format
Calendar c = Calendar.getInstance();
```

```
File Edit Window Help Howl
% java FullMoons
Полнолуние было в Пят Фев 06 04:09:35 MST 2004
Полнолуние было в Сб Мар 06 16:38:23 MST 2004
Полнолуние было в Пон Апр 05 06:07:11 MDT 2004
```

## Ответы

## Поработайте компилятором

```
StaticSuper( ) {
  System.out.println(
    "Родительский конструктор");
}
```

**StaticSuper — это конструктор. Он должен иметь в своей записи скобки (). Как видно из программного вывода, приведенного ниже, статические блоки обоих классов запускаются раньше любого из конструкторов.**

**Возможный результат работы:**

```
File Edit Window Help Cling
%java StaticTests
Родительский статический блок
Статический блок 3
Внутри main
Родительский конструктор
Конструктор
```

## Правда или ложь

1. Первое, что нужно сделать при использовании класса Math, — создать его экземпляр. **Ложь**
2. Вы можете пометить конструктор ключевым словом static. **Ложь**
3. Статические методы не имеют доступа к состоянию переменных объекта через ключевое слово 'this'. **Правда**
4. Для вызова статических методов рекомендуется применять ссылочные переменные. **Ложь**
5. Статические переменные могут использоваться для подсчета экземпляров класса. **Правда**
6. Конструкторы вызываются до инициализации статических переменных. **Ложь**
7. MAX_SIZE — подходящее имя для статической финализированной переменной. **Правда**
8. Статический инициализатор запускается раньше конструктора класса. **Правда**
9. Если класс помечен модификатором final, то все его методы должны быть финализированы. **Ложь**
10. Финализированный метод может быть переопределен только тогда, когда его класс унаследован. **Ложь**
11. Для типа boolean не существует класса-обертки. **Ложь**
12. Обертка применяется в тех случаях, когда примитив должен притвориться объектом. **Правда**
13. Методы parseXxx всегда возвращают String. **Ложь**
14. Классы для форматирования (которые раньше относились к вводу/выводу) находятся в пакете java.format. **Ложь**

# Ответы

```
import java.util.*;

import static java.lang.System.out;

class FullMoons {

  static int DAY_IM = 1000 * 60 * 60 * 24;

  public static void main(String [] args) {

    Calendar c = Calendar.getInstance();

    c.set(2004,0,7,15,40);

    long day1 = c.getTimeInMillis();

    for (int x = 0; x < 60; x++) {

       day1 += (DAY_IM * 29.52)

       c.setTimeInMillis(day1);

       out.println(String.format("full moon on %tc", c));

    }

  }

}
```

Примечания к лунным магнитикам с кодом.

Вы могли увидеть, что несколько дат, выдаваемых этой программой, смещены на день. Астрономия — довольно хитрая штука, и если мы сделаем все идеально, это слишком усложнит упражнение.

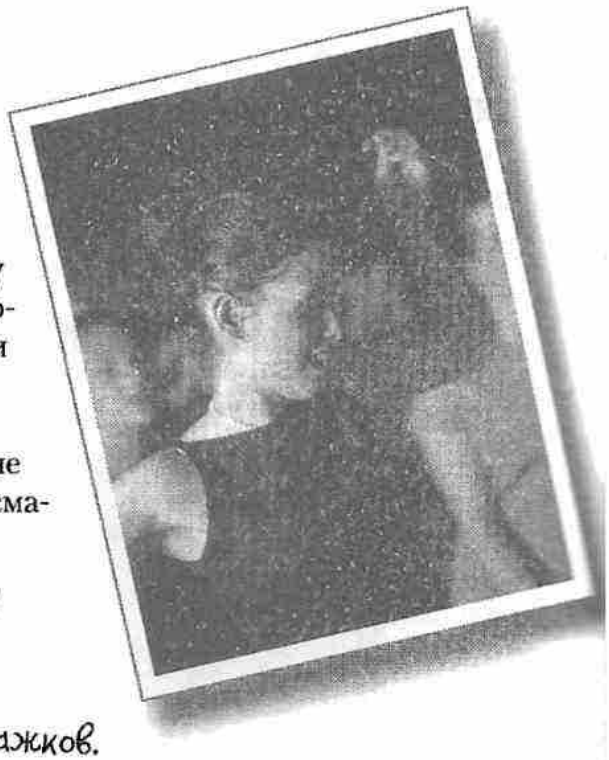
***Подсказка:*** одна проблема, которую вы можете решить, проявляется из-за разницы в часовых поясах. Сможете ли вы это уладить?

```
File Edit Window Help Howl
% java FullMoons
Полнолуние было в Пят Фев 06 04:09:35 MST 2004
Полнолуние было в Сб Мар 06 16:38:23 MST 2004
Полнолуние было в Пон Апр 05 06:07:11 MDT 2004
```

# Опасное поведение

**Всякое случается. То файл пропадает, то сервер падает.** Не важно, насколько хорошо вы программируете, ведь невозможно контролировать все. Что-то может пойти не так. *Совсем не так.* При создании опасного метода вам понадобится код, который будет обрабатывать возможные нестандартные ситуации. Но как узнать, опасен ли метод? И куда поместить код для *обработки* ***исключительной*** ситуации? До этого момента в книге мы ничем не рисковали. Конечно, случалось, что при выполнении приложения что-то работало неправильно, но проблема чаще всего заключалась в коде. Ошибки? Их мы должны исправлять еще при написании программы. Нет, обработка проблем, о которой идет речь, касается кода, чью работоспособность во время выполнения вы *не можете* гарантировать. Код, который ожидает наличия файла в правильном каталоге или надеется, что сервер работает либо поток остается приостановленным. И этим мы должны заняться прямо сейчас. Начнем создавать нечто особенное, использующее опасный JavaSound API, а именно разработаем музыкальный MIDI-проигрыватель.

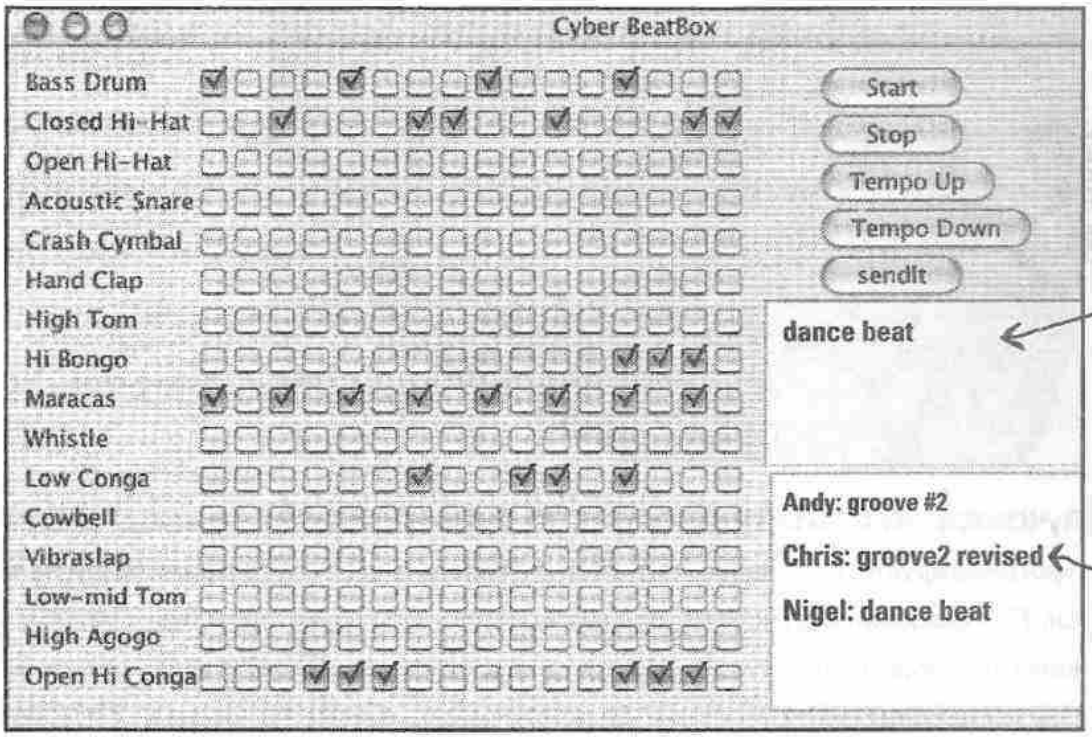
# Давайте создадим Музыкальную Машину



На протяжении следующих трех глав мы создадим несколько разных музыкальных приложений, включая драм-машину BeatBox. Фактически еще до конца книги будет готова многопользовательская версия, поэтому вы сможете посылать свои фрагменты другому музыканту — получится нечто вроде онлайн-конференции. Можете написать все с нуля или использовать заранее подготовленный код для GUI. Конечно, не каждому IT-отделу нужен новый BeatBox-сервер, но мы рассматриваем его, чтобы вы могли лучше ***изучить Java***.

Окончательная версия BeatBox выглядит примерно так:

Создан музыкальный цикл (16-тактная последовательность) с использованием флажков.




Ваше сообщение, которое вместе с текущей последовательностью отсылается другим музыкантам (при нажатии кнопки SendIt (Отправить)).

Входящие сообщения от других музыкантов. Щелкая на них, вы получаете последовательности которые идут вместе с этими сообщениями. Для воспроизведения нажмите кнопку Start (Играть).

Отметьте флажками каждый из 16 «битов» (тактов). Например, в первом такте (из 16) играют большой барабан и маракас, во втором тишина, а в третьем играют маракас и закрытый хэт. Надеюсь, вы поняли. Когда вы нажимаете кнопку Start (Играть), в цикле начинает проигрываться ваша последовательность. Она воспроизводится, пока не будет нажата кнопка Stop (Остановить). В любой момент вы можете сделать «снимок» одной из своих последовательностей, отправив ее на сервер BeatBox (позволяя тем самым любому музыканту прослушать ее). Вы также можете загрузить каждую из входящих последовательностей, щелкнув на сообщении, вместе с которым она пришла.

# Начнем с основ

Естественно, прежде чем программа будет завершена, придется выучить несколько новых понятий. Вы научитесь создавать GUI с помощью Swing, *соединяться* с другим компьютером по сети (и чтобы вы могли *отсылать* этому компьютеру данные, мы также слегка затронем тему ввода/вывода).

Чуть не забыли про JavaSound API! Именно с него мы начнем главу. Пока вы можете не вспоминать о GUI, сетях и вводе/выводе, полностью сконцентрировавшись на выжимании из своего компьютера каких-либо звуков, сгенерированных с помощью MIDI. И не переживайте, если вам ничего не известно ни о MIDI, ни о чтении или создании музыки вообще. Все необходимое вы узнаете здесь. Вы практически на расстоянии вытянутой руки от контракта со звукозаписывающей компанией.

## JavaSound API

JavaSound — это набор классов и интерфейсов, появившихся в Java в версии 1.3. Это не специальные дополнения, а часть стандартной библиотеки классов J2SE. JavaSound состоит из двух частей: MIDI и Sampled (цифровой звук). В этой книге мы рассматриваем только MIDI. Эта аббревиатура расшифровывается как Musical Instrument Digital Interface — цифровой интерфейс музыкальных инструментов. Это стандартный протокол для установления связи между различным электронным звуковым оборудованием. Но вы можете относиться к MIDI как к устройству, в которое помещаются партитуры (нотные записи), или как к высокотехнологичному «пианино». Иными словами, данные в формате MIDI на самом деле не содержат *звуков*, они представляют собой *инструкции*, которые могут быть проиграны инструментом, считывающим MIDI. Если использовать другую аналогию, то MIDI-файл — это нечто вроде HTML-документа, а инструментом для его воспроизведения служит браузер.

Данные в формате MIDI говорят, *что* делать (сыграть среднее «до» в такой-то тональности, с такой-то продолжительностью и т. д.), но они никак не описывают *звук*, который вы слышите. MIDI не знает, как издавать звуки флейты, фортепиано или гитары Джими Хендрикса. Непосредственно для звуков нужен инструмент (MIDI-устройство), способный прочитать и сыграть MIDI-файл. Но такое устройство, как правило, больше похоже *на целый оркестр*. Инструменты в нем могут быть как настоящими (как электронные синтезаторы, на которых играют рок-музыканты), так и виртуальными, представляющими собой программу в вашем компьютере.

Для нашего BeatBox мы будем использовать только программный, встроенный в Java инструмент. Он называется *синтезатором*, так как *синтезирует* звук. Звук, который вы *слышите*.

# Для начала нам нужен синтезатор

Прежде чем получится издать какой-либо звук, нужно создать объект Sequencer. Он принимает все данные в формате MIDI и отсылает их соответствующим инструментам. Это объект, который *воспроизводит* музыку. Синтезатор может выполнять много вещей, но в этой книге мы будем его использовать исключительно в качестве проигрывающего устройства. Как CD-проигрыватель в вашем музыкальном центре, но с дополнительными возможностями. Класс Sequencer находится в пакете javax.sound.midi (часть стандартной библиотеки Java, начиная с версии 1.3). Итак, сначала убедимся, что мы можем создать (или получить) объект Sequencer.

```
import javax.sound.midi.*;

public class MusicTest1 {

    public void play() {
        Sequencer sequencer = MidiSystem.getSequencer();
        System.out.println("Мы получили синтезатор");
    } // Закрываем play

    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    } // Закрываем main
} // Закрываем class
```

Импортируем пакет javax.sound.midi.

Нам нужен объект Sequencer. Это главная часть MIDI-устройства/инструмента, который мы используем. Объект Sequencer синтезирует «композицию» из информации в формате MIDI. Но мы не создаем собственный синтезатор, а запрашиваем его у класса MidiSystem.

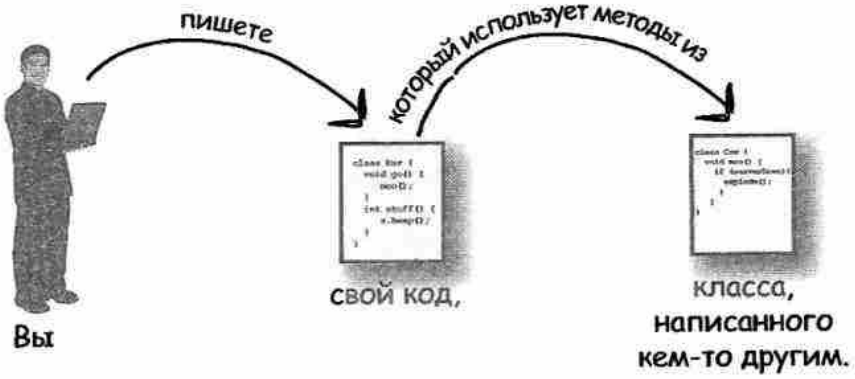
## Что-то не так!

Этот код не скомпилируется! Компилятор говорит, что у нас есть «незарегистрированное» исключение, которое должно быть отловлено или объявлено.

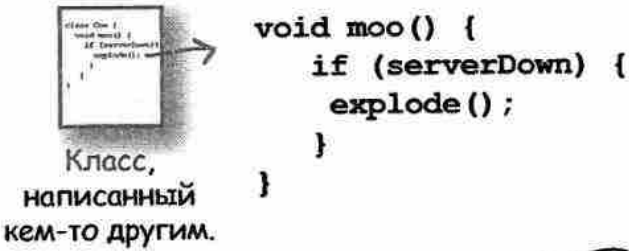
```
File Edit Window Help SayWhat?
% javac MusicTest1.java
MusicTest1.java:13: unreported exception javax.sound.midi.
MidiUnavailableException; must be caught or declared to be thrown
    Sequencer sequencer = MidiSystem.getSequencer();
                                     ^
1 errors
```

# Что происходит, если метод, который вы хотите вызвать (вероятно, из класса, написанного другим разработчиком), опасен?

1. **Представьте, что вам необходимо вызвать метод из класса, написанного кем-то другим.**




2. **Этот метод делает кое-что опасное, что может не сработать при выполнении программы.**




3. **Вам нужно *знать*, что вызываемый метод опасен.**

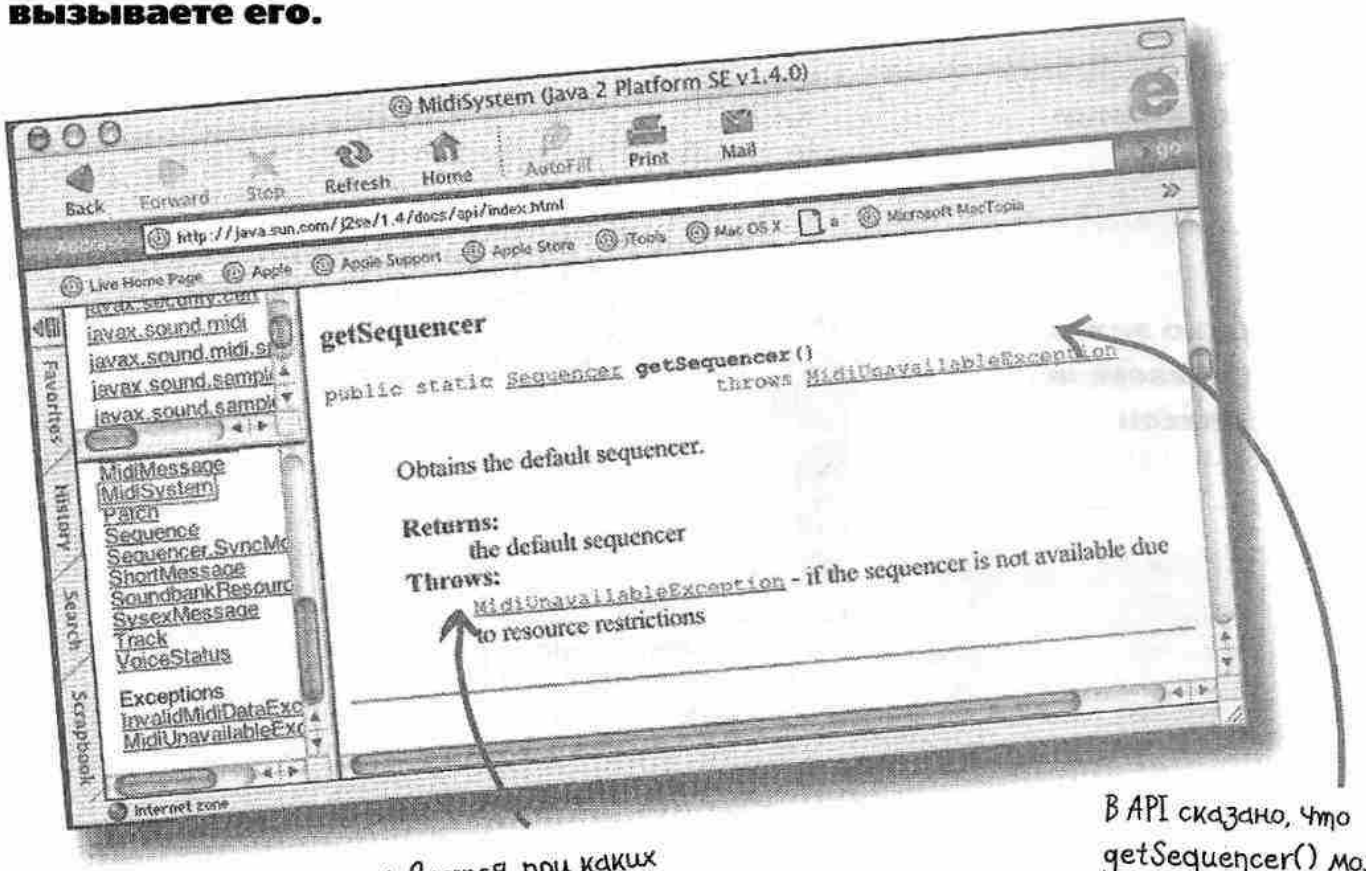
4. **Тогда вы пишете код, который способен обработать ошибку, если она *произойдет*. Вы должны быть полностью подготовлены.**

# Методы в Java используют исключения, чтобы сообщить вызывающему коду: «Случилось нечто плохое. Я потерпел неудачу».

Механизм обработки исключений в Java — это изящный и простой способ улаживать «исключительные ситуации», возникающие при выполнении программы. Он позволяет поместить весь код, занимающийся отловом ошибок, в одно место. Действие механизма основано на вашем *знании* о том, что вызываемый метод опасен (то есть *может* сгенерировать исключение). *Подозревая*, что при вызове конкретного метода можно получить исключение, вы можете *подготовиться* к этой проблеме или даже устранить ее.

Как же узнать, выдает ли метод исключение? При объявлении таких методов используется ключевое слово **`throws`**.

**Метод `getSequencer()` опасен. Его работа при выполнении программы может завершиться неудачей. Метод должен «объявить» о риске, который *вы* берете на себя, когда вызываете его.**




Здесь рассказывается, при каких обстоятельствах вы можете получить исключение — в данном случае из-за ограниченного доступа к ресурсам (иногда это означает, что синтезатор уже используется).

В API сказано, что getSequencer() может выдать исключение MidiUnavailableException. Метод должен объявить о вероятном исключении.

# Компилятор должен знать, что вы осознаете последствия вызова опасного метода.

Если вы заключите опасный код в конструкцию **try/catch**, то компилятор расслабится.

Блок try/catch говорит компилятору, что вы *знаете* об исключительной ситуации, которая может произойти в вызываемом методе, и вы готовы ее обработать. Компилятору все равно, *как* вы это сделаете; он беспокоится лишь о том, чтобы вы взяли на себя ответственность.

*Дорогой компилятор!*

*Я знаю, что иду на риск, но не кажется ли тебе, что это того стоит? Что мне делать?*

*Подпись: программист из Простоквашино.*

---

*Дорогой программист!*

*Жизнь коротка (особенно в куче). Рискуй. Пробуй. Но если что-то пойдет не так, будь готов отловить любые проблемы до того, как все выйдет из-под контроля.*

```
import javax.sound.midi.*;

public class MusicTest1 {
    public void play() {

        try {
            Sequencer sequencer = MidiSystem.getSequencer();
            System.out.println("Успешно получили синтезатор");
        } catch(MidiUnavailableException ex) {
              System.out.println("Неудача");
        }
    } // Закрываем play

    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    } // Закрываем main
} // Закрываем class
```

Помещаем опасные строки в блок try.

Создаем блок catch на тот случай, если произойдет исключительная ситуация, то есть если методом getSequencer() будет выдано исключение MidiUnavailableException.

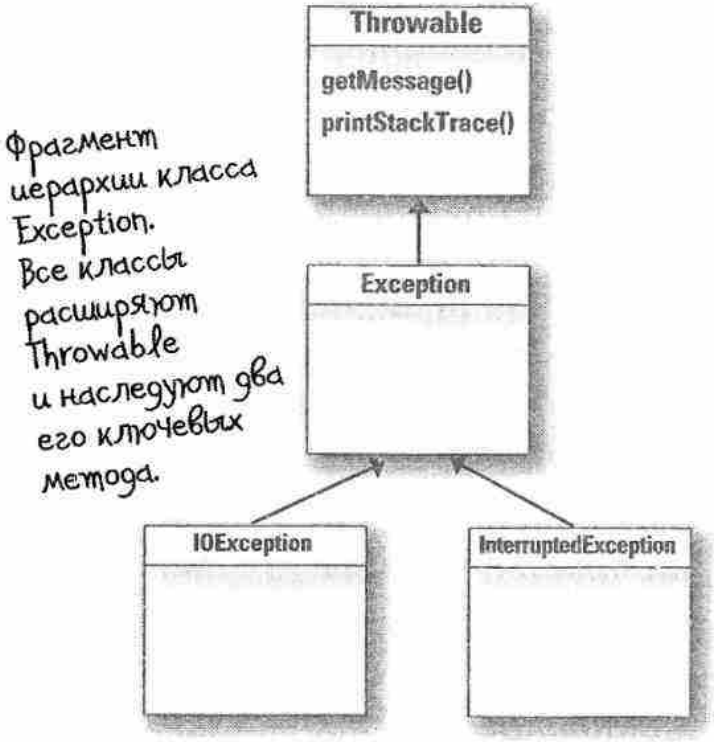
Не пытайтесь повторить это у себя дома.

## Исключение — это объект... типа Exception

И это хорошо, так как было бы намного сложнее запоминать, если бы исключения имели тип Broccoli.

Из глав о полиморфизме вы должны помнить, что объект типа Exception может быть экземпляром любого *дочернего класса* Exception.

Поскольку *Exception* — объект, то и отлавливаете вы объект. В следующем коде аргумент для **catch** объявлен с типом Exception, а параметр имеет имя *ex*.




Фрагмент иерархии класса Exception. Все классы расширяют Throwable и наследуют два его ключевых метода.

```
try {
    // Делаем опасные вещи
} catch(Exception ex) {
    // Пытаемся все исправить
}
```

То же самое, что объявление аргумента для метода.

Этот код срабатывает только тогда, когда выброшено исключение.

Содержимое блока catch зависит от исключения, которое было выброшено. Например, если сервер не отвечает, вы можете использовать блок catch для попытки связаться с другим сервером. Если файла не оказалось на месте, можно попросить пользователя помочь его найти.

# Если ваш код отлавливает исключение, то чей код его выбрасывает?

Программируя на языке Java, вы будете тратить значительно больше времени на *обработку* исключений, чем на их *создание* и *выбрасывание*. Пока имейте в виду, что опасный метод, который вы вызываете, *выбрасывает* исключение обратно *вам* — тому, кто его вызвал.

На практике вы можете оказаться по обе стороны. На самом деле не имеет значения, кто пишет код. Важно знать, какой метод *выбрасывает* исключение, а какой метод его *отлавливает*.

При написании кода, который может выбросить исключение, необходимо *объявить* это исключение.

**1 Опасный код, выбрасывающий исключение:**

Этот метод должен сообщить всем (через объявление), что он выбрасывает BadException.

```
public void takeRisk() throws BadException {
    if (abandonAllHope) {
      throw new BadException();
    }
}
```

Создаем новый объект Exception и выбрасываем его.

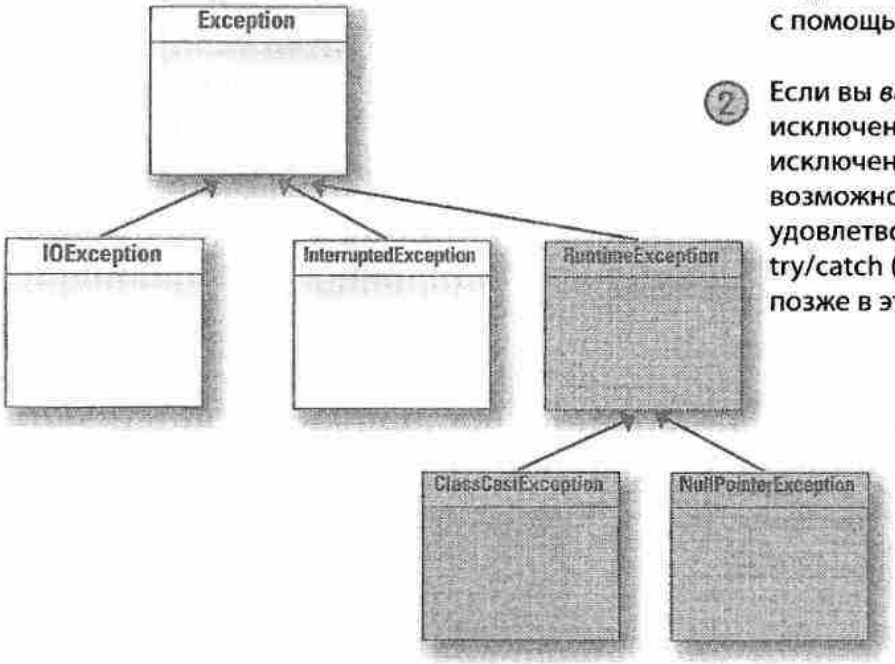
Один метод поймает то, что выбросил другой. Исключение всегда выбрасывается обратно в вызывающий код. Метод, который может выбросить исключение, должен об этом объявить.

**2 Ваш код, который вызывает опасный метод:**

```
public void crossFingers() {
    try {
      anObject.takeRisk();
    } catch (BadException ex) {
      System.out.println("Aax!");
      ex.printStackTrace();
}
```

Если вы не можете справиться с исключением, по крайней мере получите трассировку стека с помощью метода printStackTrace(), который наследуют все потомки Exception.

Исключения, не являющиеся потомками класса RuntimeException, проверяются компилятором. Это так называемые проверяемые исключения.




Потомки RuntimeException не проверяются компилятором. Они известны как непроверяемые исключения. Вы можете их выбрасывать, отлавливать и объявлять, но это необязательно.

## Компилятор проверяет все исключения, кроме RuntimeException.

## Компилятор гарантирует следующее.

1. Если вы *выбрасываете* исключения в своем коде, то должны объявить его при объявлении метода с помощью ключевого слова *throws*.

2. Если вы *вызываете* метод, который выбрасывает исключение (то есть метод, *объявляющий* о выбросе исключения), то должны *подтвердить*, что осознаете возможность этого выброса. Основной способ удовлетворить компилятор — заключить вызов в блок try/catch (есть и альтернатива, но о ней мы поговорим позже в этой главе).

### Это не глупые вопросы

**В: Погодите! Почему мы только сейчас впервые использовали блок try/catch? Что насчет исключений, с которыми я уже сталкивался — NullPointerException и DivideByZero? Я даже получил NumberFormatException из метода Integer.parseInt(). Почему не пришлось их отлавливать?**

О: Компилятора интересуют все потомки класса Exception, *кроме* особенного типа RuntimeException. Любое исключение, унаследованное от RuntimeException, получает полную свободу. Оно может быть выброшено где угодно, с объявлениями и блоками try/catch или без них. Компилятора не волнует, может ли метод выбросить RuntimeException и знает ли вызывающий код о возможности выброса такого исключения при выполнении программы.

**В: Постойте, но почему компилятору нет дела до этих исключений? Разве они не могут испортить всю программу?**

О: Большинство исключений типа RuntimeException возникают из-за логики вашего кода, а не из-за ситуаций, происходящих при выполнении программы, которые вы **не можете** предсказать или предотвратить. Вы *не можете* гарантировать, что файл будет именно там, где ожидается. Вы *не можете* гарантировать, что сервер будет работать. Но вы *можете* убедиться в том, что ваш код не использует индекс, выходящий за пределы массива (вот для чего нужен атрибут length).

Вы хотите, чтобы исключение RuntimeException случилось во время разработки и тестирования. Вы не хотите добавлять лишние блоки try/catch для отлова ситуаций, которые, скорее всего, не произойдут.

Блок try/catch предназначен для обработки исключительных ситуаций, а не дефектов кода. Используйте его, чтобы исправлять действия, успешность которых вы не можете гарантировать. В крайнем случае выведите пользователю сообщение и трассировку стека, чтобы была возможность выяснить, что произошло.

## КЛЮЧЕВЫЕ МОМЕНТЫ

- Если при выполнении программы что-то пошло не так, метод может выбросить исключение.
- Исключение — это всегда объект типа Exception (как вы помните из главы о полиморфизме, это может быть экземпляр класса, находящийся в иерархии наследования ниже Exception).
- Компилятор не обращает внимания на исключения типа **RuntimeException**. Такие исключения необязательно объявлять или заключать в блок try/catch (хотя можно сделать и то и другое).
- Все исключения, которые интересны компилятору, называются проверяемыми (в том смысле, что компилятор их проверяет). Не проверяются только исключения, имеющие тип RuntimeException. Все остальные должны быть объявлены и обработаны согласно правилам.
- Метод выбрасывает исключение с помощью ключевого слова **`throw`**, за которым следует объект Exception:

```
throw new NoCaffeineException();
```

- Метод, который *может* выбросить проверяемое исключение, ***обязан*** объявить об этом с помощью выражения **`throws Exception`**.
- Если в своем коде вы вызываете метод, который может выбросить проверяемое исключение, то должны убедить компилятор в том, что меры предосторожности были приняты.
- Если вы готовы обработать исключение, заключите вызов в блок try, поместив код по обработке/восстановлению в блок catch.
- Если вы не готовы обработать исключение, то можете «пробросить» его дальше, удовлетворив тем самым компилятор. Об этом мы поговорим чуть позже.

### Общепознавательный совет

Если вы хотите выучить что-нибудь новое, делайте это в *последнюю* очередь. Таким образом, отложив эту книгу (предполагается, что вы способны от нее оторваться), не читайте ничего серьезней упаковки с кукурузными хлопьями. Вашему мозгу нужно время для осмысления полученной информации. Это может занять несколько *часов*. Если вы попытаетесь добавить поверх знаний о Java что-то новое, то часть усвоенного материала может быстро улетучиться.

Конечно, это правило не касается приобретения физических навыков. Оттачивание ваших бальных па, скорее всего, не повлияет на изучение Java.

Для лучшего эффекта читайте эту книгу (или хотя бы просматривайте картинки) прямо перед сном.

## Наточите свой карандаш

**В какой из этих ситуаций может быть выброшено исключение, на которое компилятор обратит внимание? Нас интересуют только те действия, которые нельзя контролировать в коде. Первый пункт мы уже отметили за вас.**

Потому что он самый простой.

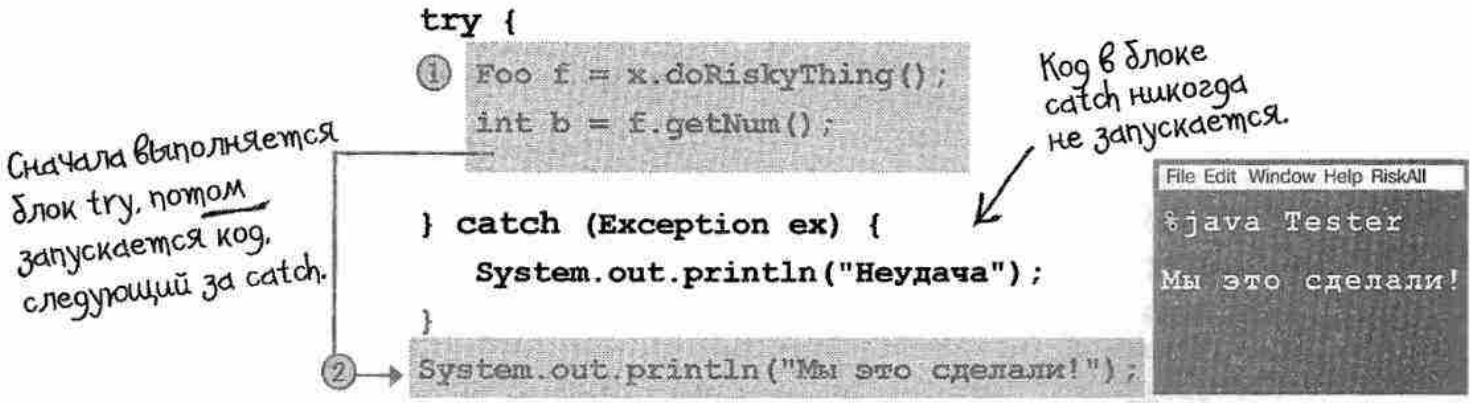
| Действия, которые вы хотите выполнить | Что может пойти не так |
|---|---|
| ✓ Подключиться к удаленному серверу. | Сервер не работает. |
| __ Выйти за пределы массива. | |
| __ Вывести на экран окно. | |
| __ Получить информацию из базы данных. | |
| __ Проверить, находится ли файл там, где вы *думаете*. | |
| __ Создать новый файл. | |
| __ Считать символ из командной строки. | |

# Управление программным потоком в блоках try/catch

Когда вы вызываете опасный метод, может случиться одно из двух: либо этот метод успешно выполнится и блок try завершится, либо он выбросит исключение вызвавшему его методу.

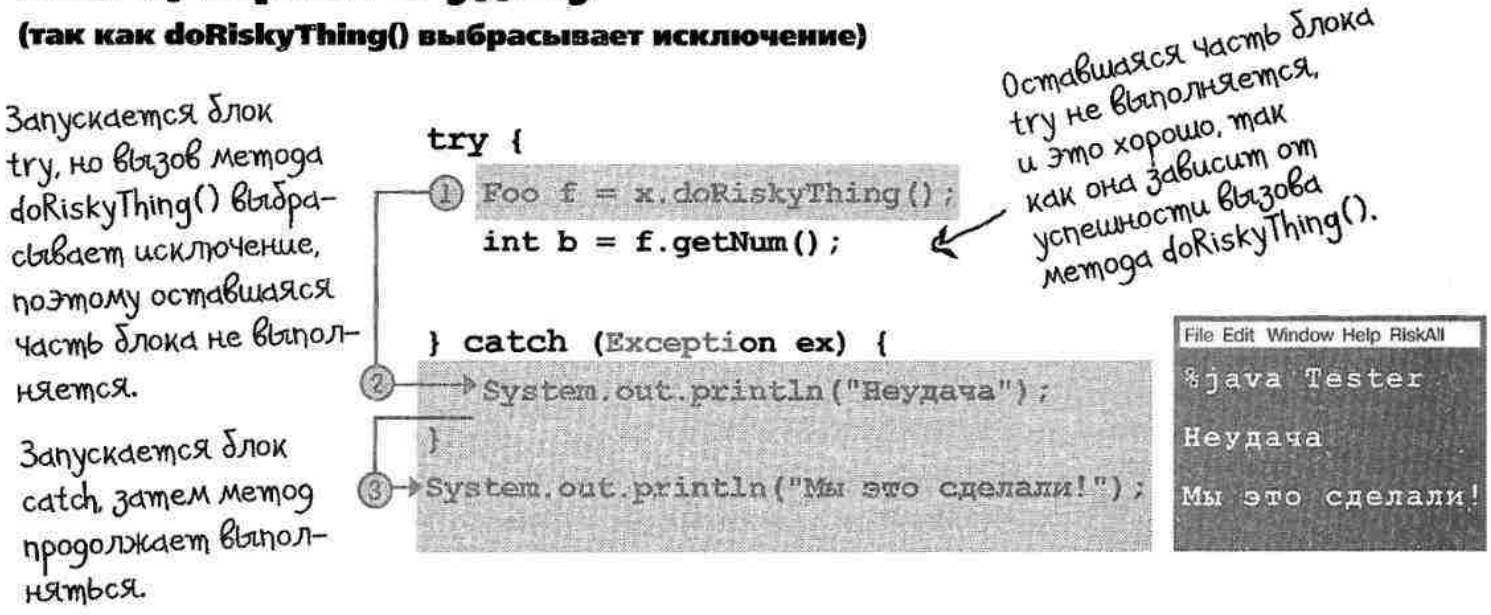
## Если блок try успешно завершается

**(doRiskyThing() не выбрасывает исключения)**




## Блок try терпит неудачу

**(так как doRiskyThing() выбрасывает исключение)**



# Finally: для действий, которые нужно выполнить несмотря ни на что

Если вы хотите что-нибудь приготовить, нужно сначала включить плиту.

Если ваше блюдо полностью **испортилось,** ***нужно выключить плиту.***

Если у вас все **получилось,** ***нужно выключить плиту.***

***Нужно выключить плиту несмотря ни на что!***

**Finally — это блок для кода, который должен выполниться *независимо* от того, было ли выброшено исключение.**

```
try {
  turnOvenOn();
  x.bake();
} catch (BakingException ex) {
    ex.printStackTrace();
} finally {
    turnOvenOff();
}
```

Без finally пришлось бы указывать метод turnOvenOff() в *обоих* блоках — try и catch, так как ***нужно выключить плиту в любом случае***. Блок finally позволяет поместить важный код для очистки в *одном* месте, избегая дублирования, пример которого приводится ниже:

```
try {
  turnOvenOn();
  x.bake();
  turnOvenOff();
} catch (BakingException ex) {
    ex.printStackTrace();
    turnOvenOff();
}
```

**Если блок try завершился неудачей (исключением),** то управление программным потоком немедленно перейдет к блоку catch. По завершении catch выполняется блок finally, после которого, в свою очередь, продолжает выполняться метод.

**Если блок try завершился успешно (без выброса исключения),** то управление программным потоком пропускает блок catch и переходит к finally, после которого, в свою очередь, продолжает выполняться метод.

**Если блоки try или catch содержат оператор return, то finally все равно будет выполняться!** Поток перейдет к finally, после чего вернется к return.

Наточите свой карандаш

# Управление программным потоком

**Взгляните на код, представленный слева. Каким, по вашему мнению, будет результат его работы? Каким он будет, если заменить третью строку на `String test = "да";`? Учитывайте, что ScaryException наследует Exception.**

```
public class TestExceptions {

  public static void main(String [] args) {

    String test = "Нет";
    try {
       System.out.println("Начало try");
       doRisky(test);
       System.out.println("Конец try");
    } catch ( ScaryException se) {
        System.out.println("Жуткое исключение");
    } finally {
        System.out.println("finally");
    }
    System.out.println("Конец main");
  }

  static void doRisky(String test) throws ScaryException {
    System.out.println("Начало опасного метода");
    if ("yes".equals(test)) {
      throw new ScaryException();
    }
    System.out.println("Конец опасного метода");
    return;
  }
}
```

**Результат при** `test = "Нет"`

**Результат при** `test = "Да"`

При test = "Нет": начало try — начало опасного метода — конец опасного метода — конец try — finally — конец main

При test = "Да": начало try — начало опасного метода — жуткое исключение — finally — конец main

# Мы уже говорили, что метод может выбрасывать несколько исключений?

Метод может выбрасывать сразу несколько исключений, если он в этом очень нуждается. Но при объявлении метода нужно указать *все* проверяемые исключения, которые он может выбросить (хотя, если два или больше из них имеют общего предка, можно ограничиться одним родительским классом).

## Перехват нескольких исключений

Компилятор проконтролирует, обработали ли вы *все* проверяемые исключения, которые выбрасываются вызываемым методом. Размещайте блоки *catch* после *try*, один за другим. Иногда порядок размещения блоков играет важную роль, но мы поговорим об этом чуть позже.

```
public class Laundry {

    public void doLaundry() throws PantsException, LingerieException {
       // Код, который может выбросить оба исключения
    }
}
```

В этом методе объявлено сразу два исключения.

---

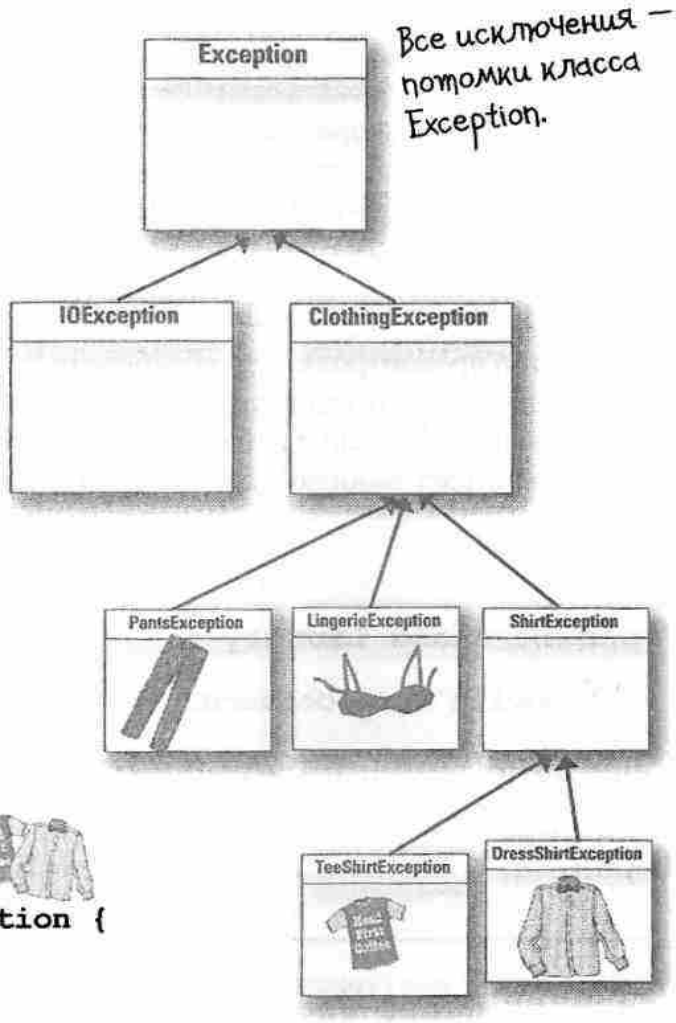
```
public class Foo {

    public void go() {
        Laundry laundry = new Laundry();
        try {
            laundry.doLaundry();
        } catch(PantsException pex) {
            // Восстановительный код

        } catch(LingerieException lex) {
            // Восстановительный код
        }
    }
}
```

Если doLaundry() выбросит исключение PantsException, то управление перейдет к блоку catch с PantsException.

Если doLaundry() выбросит исключение LingerieException, то управление перейдет к блоку catch с LingerieException.

# Исключения поддерживают полиморфизм

Не забывайте, что исключения — это объекты. От остальных объектов они отличаются лишь тем, что их можно ***выбросить***. Таким образом, как и на любой добропорядочный объект, на Exception можно ссылаться с использованием полиморфизма. *Объект* LingerieException, например, может быть присвоен *ссылке* типа ClothingException; PantsException может быть присвоен ссылке типа Exception. Основной плюс — метод не обязан объявлять все исключения, которые может выбросить; достаточно указать родительский класс. Это же касается и блоков catch — необязательно создавать их для каждого возможного исключения, если они и так их перехватывают.




**1 При объявлении вы используете родительский тип для выбрасываемых исключений.**

```
public void doLaundry() throws ClothingException {
```

Объявив ClothingException, вы можете выбрасывать любые исключения, тип которых будет дочерним по отношению к ClothingException. Это означает, что doLandry() может выбросить PantsException, LingerieException, TeeShirtException и DressShirtException, не объявляя каждое из них отдельно.

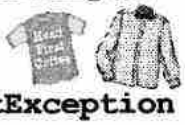
**2 Вы можете перехватывать выбрасываемые исключения с помощью их родительского типа.**

```
try {
    laundry.doLaundry();
} catch(ClothingException cex) {
    // Восстановительный код
}
```

Может поймать любой подтип ClothingException.

```
try {
    laundry.doLaundry();
} catch(ShirtException sex) {
    // Восстановительный код
}
```

Может поймать только TeeShirtException и DressShirtException.



**Как вы поняли, можно отлавливать все с помощью единственного полиморфического блока catch, но это не значит, что нужно всегда так делать.**

*Можете* использовать при обработке исключений всего *один* блок catch, указывая родительский класс Exception. Тогда вы сумеете перехватить *любое* исключение, которое может быть выброшено.

```
try {
    laundry.doLaundry();
} catch(Exception ex) {
    // Восстановительный код...
}
```

Что он восстанавливает? Этот блок catch будет перехватывать все исключения, поэтому вы не сможете автоматически узнать, что именно пошло не так.

**Создавайте отдельные блоки catch для каждого исключения, которое нужно обработать уникальным образом.**

Например, если ваш код по-разному обрабатывает исключения TeeShirtException и LingerieException, создайте для каждого из них по блоку catch. Но если с остальными потомками ClothingException вы работаете одинаково, то просто добавьте это исключение в блок catch, чтобы обработать всех его потомков.

```
try {
    laundry.doLaundry();
} catch(TeeShirtException tex) {
    // Восстановление после TeeShirtException
} catch(LingerieException lex) {
    // Восстановление после LingerieException
} catch(ClothingException cex) {
    // Восстановление после all others
}
```

Исключениям TeeShirtException и LingerieException для обработки нужен свой код, поэтому вы должны использовать разные блоки catch.

Все остальные потомки ClothingException отлавливаются здесь.

# Множественные блоки catch должны располагаться по возрастанию: от наименьшего к наибольшему

Сюда попадает исключение TeeShirtException, а остальные не помещаются.

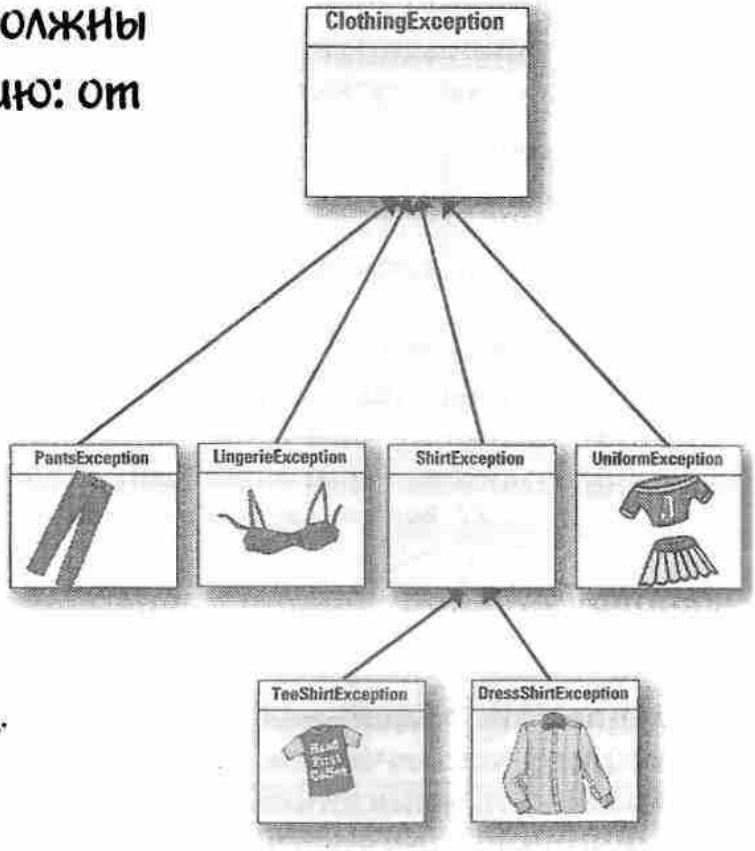
```
catch(TeeShirtException tex)
```

TeeShirtException никогда сюда не дойдет, в отличие от остальных потомков ShirtException.

```
catch(ShirtException sex)
```

Здесь отлавливаются все производные ClothingException; ShirtException никогда не зайдет так далеко.

```
catch(ClothingException cex)
```




Чем выше вы взбираетесь по дереву наследования, тем «больше» должен быть блок catch. Продвигаясь вниз по иерархии наследования, вы встречаете более специализированные классы-потомки Exception, а блоки catch становятся меньше. Это старый добрый полиморфизм.

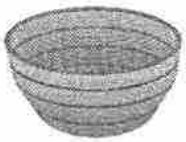
Блок catch для ShirtException достаточно вместителен и может включать в себя такие исключения, как TeeShirtException и DressShirtException (а также любой дочерний класс, который будет унаследован от ShirtException). Блок для ClothingException еще объемнее (то есть существует больше объектов, на которые можно сослаться с помощью типа ClothingException). Он может использоваться для отлова исключений типа ClothingException (еще бы) и любого его подтипа: PantsException, UniformException, LingerieException и ShirtException. Родитель всех аргументов для блока catch — тип **Exception**. С его помощью можно отловить *любые* исключения, включая непроверяемые, поэтому за пределами тестового кода лучше его не использовать.

# Нельзя размещать большие блоки catch над маленькими

Конечно, вы *можете* это сделать, но код не скомпилируется. Блоки catch не похожи на перегруженные методы, из которых выбирается самый подходящий вариант. JVM просто перемещается вниз, пока не найдет вместительный блок catch (который находится достаточно высоко в иерархии наследования), чтобы обработать исключение. Если в самом верху находится **`catch(Exception ex)`**, то компилятор поймет, что нет смысла добавлять другие блоки, так как они никогда не будут задействованы.

Не делайте так!

```
try {
  laundry.doLaundry();
} catch(ClothingException cex) {
    // Восстановление после ClothingException
} catch(LingerieException lex) {
   // Восстановление после LingerieException
} catch(ShirtException sex) {
    // Восстановление после ShirtException
}
```



***Блоки catch одного уровня могут располагаться в любом порядке, так как не способны перехватить чужое исключение.***

Вы можете разместить ShirtException над LingerieException — никто не будет возражать. Хотя ShirtException — более вместительный тип, его блок catch не может перехватить исключение LingerieException, поэтому здесь нет проблем.

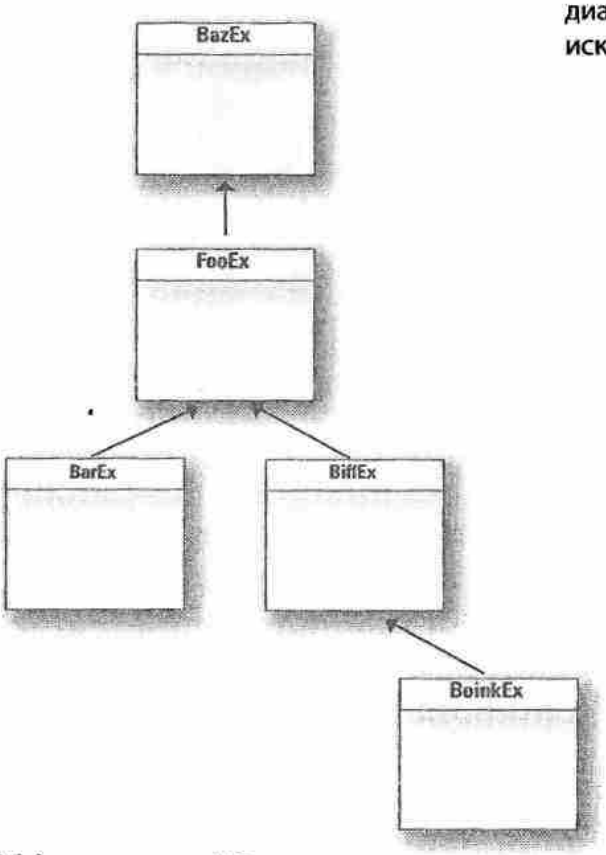
## Наточите свой карандаш

Предположим, что блоки try/catch, представленные здесь, написаны правильно. Ваша задача — нарисовать две схемы классов, которые смогли бы изобразить расположение исключений в иерархии наследования. Проще говоря, определите, при каких структурах наследования блоки try/catch окажутся корректными.

```
try {
  x.doRisky();
}  catch(AlphaEx a) {
     // Восстановление после AlphaEx
}  catch(BetaEx b) {
   // Восстановление после BetaEx
}  catch(GammaEx c) {
     // Восстановление после GammaEx
}  catch(DeltaEx d) {
     // Восстановление после DeltaEx
}
```

---

Ваша задача — создать два других *допустимых* блока try/catch (как показанные слева вверху), которые бы в точности отражали диаграмму классов, представленную слева. Исходите из того, что все исключения могут быть выброшены методом внутри блока try.



# Если вы не хотите обрабатывать исключение...

## Просто пробросьте его дальше

**Если вы не хотите обрабатывать исключение, то можете пробросить его на уровень ниже, используя объявление.**

Когда вы вызываете опасный метод, компилятор должен об этом знать. В большинстве случаев необходимо лишь заключить вызов в блок try/catch. Но есть и другой вариант — можно просто *пробросить* исключение наверх, позволяя методу, вызвавшему *ваш* код, обработать его.

Достаточно *объявить*, что *вы* выбрасываете исключение, даже если этого, строго говоря, не происходит. Вы просто позволяете исключению пройти мимо.

Но если вы пробрасываете исключение, оказывается ненужным блок try/catch. Что же случается, если опасный метод (doLaundry()) действительно выбрасывает объект Exception?

При выбросе исключения метод немедленно покидает стек, а само исключение передается следующему методу, *который* вызывал предыдущий. Но если *вызывавший метод* занимается *пробрасыванием*, у него нет блока catch, поэтому он покидает стек и передает исключение вниз по цепочке, и т. д. Когда же это закончится? Чуть позже вы сами все увидите.

```
public void foo() throws ReallyBadException {
    // Вызов опасного метода без блоков try/catch
    laundry.doLaundry();
}
```

На самом деле это не вы его выбрасываете. У вас нет блока try/catch для опасного метода, который вы вызываете, поэтому вы сами становитесь «опасным методом». Теперь все, кто вызывает ваш метод, должны разбираться с этим исключением.

# Пробрасывая (объявляя) исключение, вы только откладываете неизбежное.

## Рано или поздно кто-то должен с этим разобраться. Но если и main() пробросит исключение?

```
public class Washer {
    Laundry laundry = new Laundry();

    public void foo() throws ClothingException {
        laundry.doLaundry();
    }

    public static void main (String[] args) throws ClothingException {
        Washer a = new Washer();
        a.foo();
    }
}
```

Оба метода пробрасывают исключение (объявляя его), поэтому его некому обрабатывать! Но этот код все равно скомпилируется.

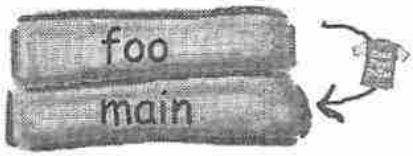
**1** doLaundry() выбрасывает ClothingException.

main() вызывает foo().

foo() вызывает doLaundry().

doLaundry() выполняется и выбрасывает ClothingException.

**2** foo() пробрасывает исключение.




doLaundry() немедленно покидает стек, и исключение переходит к foo().

Но foo() не содержит блоков try/catch, поэтому...

**3** main() пробрасывает исключение.

foo() немедленно покидает стек, и исключение переходит... куда? Кроме JVM (которая не в восторге от такого поворота событий), больше ничего не осталось.

**4** JVM завершает работу.

Чтобы визуализировать исключение ClothingException, мы использовали футболки.

# Либо обработайте, либо объявите. Это закон.

**Итак, вы уже узнали два способа удовлетворить компилятор при вызове опасного метода (выбрасывающего исключение).**

## ① Обработка

Заключаем опасный метод в блок try/catch.

```
try {
    laundry.doLaundry();
} catch(ClothingException cex) {
      // Восстановительный код
}
```

Этот блок catch должен быть достаточно большим для обработки всех исключений, которые может выбросить метод doLaundry(). В противном случае компилятор будет жаловаться, что вы не перехватываете все исключения.

## ② Объявление (проброс)

Объявляем о том, что оба метода (наш и вызываемый) выбрасывают одни и те же исключения.

```
void foo() throws ClothingException {
    laundry.doLaundry();
}
```

Метод doLandry() выбрасывает ClothingException, но благодаря объявлению метод foo() пробрасывает это исключение. Никаких try/catch.

Но это приводит к тому, что все, кто вызывает метод foo(), должны следовать закону «обработай или объяви». Если foo() пробрасывает исключение (объявляя его), то main() при вызове метода должен обработать это исключение.

```
public class Washer {
    Laundry laundry = new Laundry();

    public void foo() throws ClothingException {
        laundry.doLaundry();
    }

    public static void main (String[] args) {
        Washer a = new Washer();
        a.foo();
    }
}
```

Беда! Теперь main() не скомпилируется, и мы получим unreported exception. Компилятор обеспокоен тем, что метод foo() выбрасывает исключение.

Поскольку foo() пробрасывает ClothingException, полученное от doLaundry(), метод main() должен либо обернуть это исключение в блок try/catch, либо объявить о том, что он тоже выбрасывает ClothingException!

# Вернемся к нашему музыкальному коду...

Как вы помните, мы начали главу со знакомства с кодом из JavaSound. Мы создали объект Sequencer, но он не скомпилируется, так как метод Midi.getSequencer() объявляет проверяемое исключение MidiUnavailableException. Теперь это можно исправить, поместив вызов в блок try/catch.

```
public void play() {
        try {

            Sequencer sequencer = MidiSystem.getSequencer();
            System.out.println("Успешно получили синтезатор");

        } catch(MidiUnavailableException ex) {
            System.out.println("Неудача");
        }
    } // Закрываем play
```

С вызовом getSequencer() нет проблем, так как он заключен в блок try/catch.

Параметром для catch должно служить «правильное» исключение. Если мы напишем catch(FileNotFoundException f), то код не скомпилируется, так как с точки зрения полиморфизма MidiUnavailableException не помещается в FileNotFoundException.

Запомните: недостаточно лишь создать блок catch... Вы должны перехватить исключения, которые были выброшены!

## Правила для исключений

**1 Вы не можете использовать блоки catch или finally без try.**

```
void go() {
    Foo f = new Foo();
    f.foof();
    catch(FooException ex) { }
 }
```

Так нельзя! Где try?

**2 Вы не можете добавлять код между блоками try и catch.**

```
try {
    x.doStuff();
}
int y = 43;
} catch(Exception ex) { }
```

Так нельзя! Вы не можете поместить код между try и catch.

**3 За блоком try должны следовать catch или finally.**

```
try {
    x.doStuff();
} finally {
    // Очистка
}
```

Несмотря на отсутствие catch, это допустимо, так как есть finally. Но вы не можете указывать блок try отдельно.

**4 При использовании блока try только с finally (без catch) все равно нужно объявить исключение.**

```
void go() throws FooException {
    try {
        x.doStuff();
    } finally { }
}
```

Блок try без catch не вписывается в закон «Обработка или объявление».

# Кухня кода

***Вам не обязательно делать все самостоятельно, но так будет гораздо веселей.***

***Остальную часть этой главы можно пропустить и использовать заранее подготовленный код для всех музыкальных приложений.***

***Однако, если вы хотите узнать больше о JavaSound, просто переверните страницу.***

# Воспроизведение звука

Помните, в начале главы мы рассматривали MIDI-данные, хранящие инструкции о том, *что* (и *как*) должно быть сыграно? Мы говорили, что данные в формате MIDI на самом деле не создают звуков, *которые можно услышать*. Чтобы воспроизвести звук из динамиков, нужно отослать данные определенному типу MIDI-устройства, которое берет MIDI-инструкции и преобразует их в звук (с помощью физического инструмента или «виртуального» синтезатора). В этой книге мы будем использовать только виртуальные устройства. Посмотрите, как они работают в JavaSound.

## Вам нужны четыре составляющие

1. **Устройство, которое воспроизводит музыку.**
2. **Музыка, которая будет проигрываться, то есть композиция.**
3. **Часть синтезатора, хранящая текущую информацию.**
4. **Текущая музыкальная информация: какие ноты проигрывать, как долго и т. д.**

Синтезатор — это устройство, в котором, собственно, воспроизводится композиция. Думайте о нем как о **музыкальном CD-проигрывателе**.

Музыкальная последовательность — это композиция, которую будет воспроизводить синтезатор. В этой книге последовательность лучше представлять как музыкальный диск, на котором **содержится лишь одна песня**.

Для примера нам понадобится только один трек, поэтому представьте музыкальный диск с единственной композицией (треком). Трек содержит все данные о песне (информацию в формате MIDI).

MIDI-событие — это сообщение, которое синтезатор может понять. Если перевести название на русский, оно может означать следующее: «В данный момент времени сыграть среднее „до" с такой скоростью, в такой тональности и с такой продолжительностью».

Не исключен и иной вариант: «Заменить текущий инструмент флейтой».

*В рамках книги последовательность лучше воспринимать как CD с единственной композицией (треком). Трек содержит информацию о том, как воспроизводить песню, и представляет собой часть последовательности.*

## Вам нужно сделать пять шагов.

1. Получить **синтезатор** и открыть его.

```
Sequencer player = MidiSystem.getSequencer();
player.open();
```

2. Создать новую **последовательность**.

```
Sequence seq = new Sequence(timing,4);
```

3. Получить новый **трек** из последовательности.

```
Track t = seq.createTrack();
```

4. Заполнить трек **MIDI-событиями** и передать последовательность в синтезатор.

```
t.add(myMidiEvent1);
player.setSequence(seq);
```

# Ваше первое приложение для проигрывания звуков

Наберите этот код и запустите его. Вы услышите звук, как будто кто-то проиграл на фортепиано одну ноту (или не *кто*-то, а *что*-то)!

```
import javax.sound.midi.*;

public class MiniMiniMusicApp {

    public static void main(String[] args) {
       MiniMiniMusicApp mini = new MiniMiniMusicApp();
       mini.play();
    } // Закрываем main

   public void play() {

     try {

①      Sequencer player = MidiSystem.getSequencer();
       player.open();

②      Sequence seq = new Sequence(Sequence.PPQ, 4);

③      Track track = seq.createTrack();

④      ShortMessage a = new ShortMessage();
       a.setMessage(144, 1, 44, 100);
       MidiEvent noteOn = new MidiEvent(a, 1);
       track.add(noteOn);

       ShortMessage b = new ShortMessage();
       b.setMessage(128, 1, 44, 100);
       MidiEvent noteOff = new MidiEvent(b, 16);
       track.add(noteOff);

       player.setSequence(seq);

       player.start();

     } catch (Exception ex) {
         ex.printStackTrace();
     }
   } // Закрываем play
} // Закрываем class
```

Не забудьте импортировать пакет midi.

Получаем синтезатор и открываем его, чтобы начать использовать (изначально он не открыт).

Не обращайте внимания на аргументы для конструктора синтезатора. Нужно просто скопировать их, воспринимая как заранее подготовленные аргументы.

Запрашиваем трек у последовательности. Помните, что трек содержится внутри последовательности, а MIDI-данные — в треке.

Помещаем в трек MIDI-события. Этот код по большей части заранее подготовлен. Нужно лишь позаботиться об аргументах для метода setMessage() и конструктора MidiEvent. Мы поговорим о них на следующей странице.

Передаем последовательность синтезатору (как будто вставляем CD в проигрыватель).

Запускаем синтезатор (как будто нажимаем Play (Играть)).

# Создание MIDI-событий (данных о композиции)

MIDI-событие — это инструкция для части композиции. Набор событий — нечто вроде партитуры или ленты для механического пианино. Большинство интересующих нас событий описывают, ***что*** и в ***какой момент делать***. Часть с описанием момента времени очень важна, потому что хронометраж в музыке — самое главное. Одна нота следует за другой и т. д. Поскольку MIDI-события очень детализированы, вы должны указать, в какой момент нота *начинает* играть (событие noteOn) и в какой момент *перестает* (событие noteOff). Понятно, что если событие «закончить проигрывание ноты „до“» (сообщение noteOff) сработает *до* события «начать проигрывание ноты „до“» (noteOn), то это ни к чему не приведет.

MIDI-инструкции хранятся в объекте Message. Событие MidiEvent — это сочетание сообщения и момента времени, в который это сообщение должно «сработать». Иными словами, объект Message говорит: «Начать проигрывать среднее „до“», тогда как MidiEvent означает: «Запустить это сообщение на четвертом такте».

Всегда нужно иметь и Message, и MidiEvent.

Message говорит, *что* делать, а MidiEvent — *когда* это делать.

***MidiEvent говорит, когда и что нужно делать.***

***Любая инструкция должна включать в себя хронометраж.***

***Иначе говоря, необходимо указать, на каком такте должно произойти событие.***

1. Создаем **сообщение**.

```
ShortMessage a = new ShortMessage();
```

2. Помещаем в сообщение **инструкцию**.

```
a.setMessage(144, 1, 44, 100);
```

Сообщение гласит: «Начать проигрывать ноту № 44» (другие номера мы рассмотрим на следующей странице).

3. Используя сообщение, создаем новое **событие**.

```
MidiEvent noteOn = new MidiEvent(a, 1);
```

Инструкции хранятся в сообщении, но MidiEvent дополняет их информацией о моменте времени, в который они должны сработать. Этот экземпляр MidiEvent говорит, что сообщение a сработает на первом такте (бит 1).

4. Добавляем событие в **трек**.

```
track.add(noteOn);
```

Трек хранит все объекты MidiEvent. Они размещаются в последовательности согласно времени срабатывания, а синтезатор проигрывает их в заданном порядке. В один момент времени у вас может произойти множество событий. Например, вам понадобится проиграть одновременно две ноты или даже несколько звуков из разных инструментов.

# MIDI-сообщение: сердце MidiEvent

MIDI-сообщение хранит часть события, описывающую, *что* нужно делать. Это та самая инструкция, которую вы хотите выполнить на синтезаторе. Первым аргументом для нее всегда служит тип сообщения. От него же зависят три других аргумента. Например, сообщение типа 144 говорит: «Начать проигрывать ноту». Однако для его выполнения синтезатору нужна определенная информация. Представьте, что он вам отвечает: «Я сыграю эту ноту, но скажите, через *какой канал*». Вам нужно, чтобы ее сыграли на барабанах или на фортепиано? И *какую ноту*? Среднее до? Ре-диез? Поскольку мы об этом заговорили, определите, *какова скорость* и сила нажатия клавиши.

Чтобы получить MIDI-сообщение, необходимо создать экземпляр ShortMessage и вызвать метод setMessage(), передавая ему четыре аргумента. Сообщение описывает только то, *что* нужно делать, поэтому вы должны поместить его внутрь события, хранящего информацию о том, *когда* оно должно сработать.

## Структура сообщения

**Первый аргумент для setMessage() всегда описывает «тип» сообщения, тогда как остальные три представляют разные свойства в зависимости от указанного типа.**

> ***Сообщение*** **говорит о том, что делать.**
>
> ***MidiEvent*** **говорит о том, когда это делать.**

```
a.setMessage(144,  1,  44,  100);
```

Тип сообщения
Канал
Нота для проигрывания
Скорость и сила нажатия клавиши

Последние три аргумента зависят от типа сообщения. Это сообщение говорит о начале проигрывания ноты, поэтому остальные аргументы описывают, как именно синтезатор должен сыграть эту ноту.

### 1 Тип сообщения.

144 означает начало проигрывания ноты.

Начинаем проигрывать.

128 означает конец проигрывания ноты.

Завершаем проигрывание.

### 2 Канал.

Считайте, что канал — это музыкант в оркестре или музыкальной группе. Канал № 1 — клавишник, канал № 9 — ударник и т. д.

### 3 Нота для проигрывания.

Определяется числом от 0 до 127, начинается с низких и заканчивается высокими нотами.

### 4 Скорость и сила нажатия клавиши.

Насколько быстро и сильно вы нажимаете клавишу? 0 означает настолько слабое нажатие, что вы, наверное, ничего не услышите. 100 — хорошее стандартное нажатие.

# Как изменить сообщение

Теперь, когда вы знаете, что такое MIDI-сообщение, можно поэкспериментировать. Попробуйте изменить ноту, которая будет проигрываться, отрегулировать продолжительность ее звучания, добавить больше нот и даже поменять инструмент.

### ① Меняем ноту.

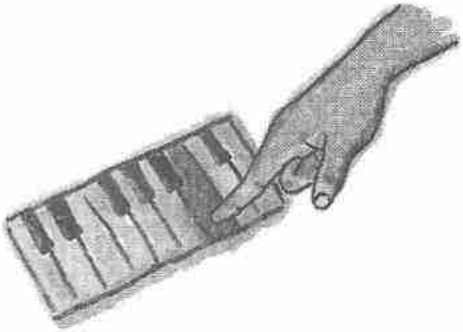
Попробуйте использовать числа в промежутке от 0 до 127 в сообщениях для включения и выключения проигрывания ноты.

```
a.setMessage(144, 1, 20, 100);
```

### ② Изменяем продолжительность звучания ноты.

Редактируйте событие (не *сообщение*) завершения воспроизведения ноты, чтобы оно сработало на более раннем или позднем такте.

```
b.setMessage(128, 1, 44, 100);
MidiEvent noteOff = new MidiEvent(b, 3);
```



### ③ Меняем инструмент.

Инструмент можно поменять до получения сообщения о начале проигрывания ноты. При этом стандартный канал № 1 (фортепиано) меняется на другой. Это сообщение имеет номер 192, а его третий аргумент служит для выбора инструмента (попробуйте числа от 0 до 127).

```
first.setMessage(192, 1, 102, 0);
```

Сообщение об изменении инструмента в канале № 1 (первый музыкант) на инструмент № 102.

# Версия № 2. Использование аргументов командной строки для экспериментов со звуками

В этой версии также проигрывается лишь одна нота, но для ее замены (или смены инструмента) вам придется использовать аргументы командной строки. Экспериментируйте, передавая два целых значения от 0 до 127. Первое число отвечает за инструмент, второе — за ноту, которая будет воспроизведена.

```
import javax.sound.midi.*;

public class MiniMusicCmdLine {    // Это первый class

   public static void main(String[] args) {
        MiniMusicCmdLine mini = new MiniMusicCmdLine();
        if (args.length < 2) {
            System.out.println("Не забудьте аргументы для инструмента и ноты");
        } else {
            int instrument = Integer.parseInt(args[0]);
            int note = Integer.parseInt(args[1]);
            mini.play(instrument, note);
        }
   } // Конец main

   public void play(int instrument, int note) {

      try {

         Sequencer player = MidiSystem.getSequencer();
         player.open();
         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track track = seq.createTrack();

         MidiEvent event = null;

         ShortMessage first = new ShortMessage();
         first.setMessage(192, 1, instrument, 0);
         MidiEvent changeInstrument = new MidiEvent(first, 1);
         track.add(changeInstrument);

         ShortMessage a = new ShortMessage();
         a.setMessage(144, 1, note, 100);
         MidiEvent noteOn = new MidiEvent(a, 1);
         track.add(noteOn);

         ShortMessage b = new ShortMessage();
         b.setMessage(128, 1, note, 100);
         MidiEvent noteOff = new MidiEvent(b, 16);
         track.add(noteOff);
         player.setSequence(seq);
         player.start();

      } catch (Exception ex) {ex.printStackTrace();}
   } // Конец play
} // Конец class
```

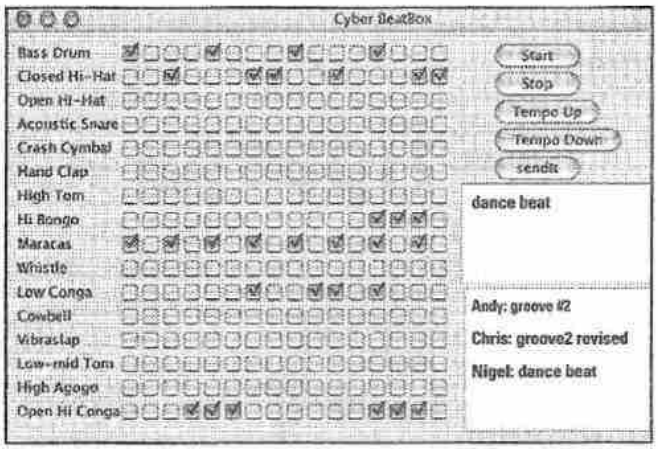
Запустите эту программу с двумя целочисленными аргументами от 0 до 127. Для начала попробуйте такие:

```
File Edit Window Help Attenuate
%java MiniMusicCmdLine 102 30
%java MiniMusicCmdLine 80 20
%java MiniMusicCmdLine 40 70
```

# Что вас ждет на Кухне кода в остальных главах

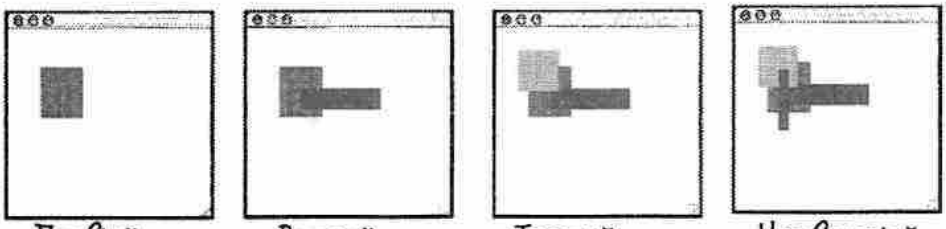
## Глава 15: цель

В конце вы получите работающее приложение BeatBox, которое к тому же будет клиентом для чата и обмена партитурами. Вам предстоит изучить GUI (включая обработку событий), ввод/вывод, работу с сетью и потоки. Этим будем заниматься на протяжении следующих трех глав (12-14).



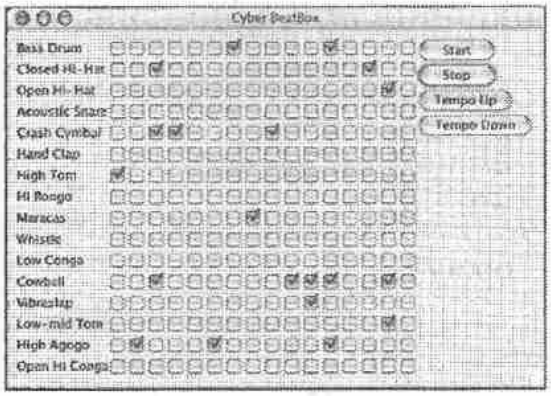

## Глава 12: MIDI-события

В Кухне кода этой главы вы создадите небольшое «музыкальное видео», в котором для каждого такта MIDI-музыки будут отображаться случайные прямоугольники. Вы научитесь создавать и запускать множество различных MIDI-событий (не считая тех, что мы уже рассмотрели в этой главе).



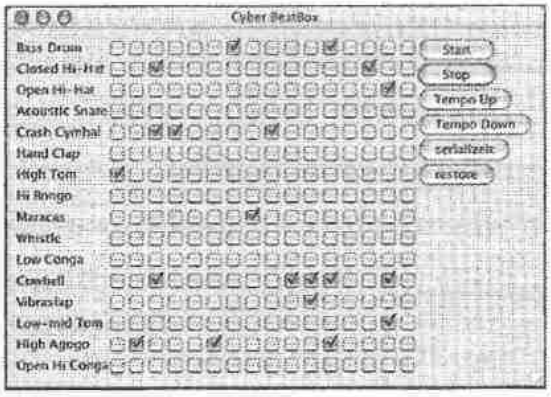

## Глава 13: автономное приложение BeatBox

Здесь вы займетесь непосредственным созданием настоящей программы BeatBox, пользовательским интерфейсом для нее и т. д. Но будут некоторые ограничения — изменив последовательность тактов, вы потеряете предыдущую версию. В ней не будет функций по сохранению и восстановлению, и она не сможет взаимодействовать с сетью. Несмотря на это, вы сможете использовать ее для улучшения своих навыков по созданию музыкальных шаблонов.




## Глава 14: сохранение и восстановление

Создав прекрасный шаблон, вы сможете сохранить его в файл, а затем восстановить, если захочется вновь его проиграть. В этой главе вы вплотную приблизитесь к окончательной версии (глава 15), где научитесь передавать шаблоны по сети чат-серверу вместо того, чтобы записывать их в файл.



Эта глава открыла перед вами удивительный мир исключений. Ваша задача — определить, какие из следующих утверждений правдивы, а какие — нет.

# Правда или ложь

1. За блоком try должны следовать блоки catch и finally.
2. Если вы пишете метод, который может сгенерировать проверяемое компилятором исключение, то вы обязаны поместить этот опасный код в блок try/catch.
3. Блоки catch могут быть полиморфичными.
4. Перехватываться могут только исключения, проверяемые компилятором.
5. При наличии блоков try/catch необязательно использовать finally.
6. Объявляя блок try, вы можете добавить к нему соответствующие блоки catch или finally (или оба сразу).
7. При написании метода, в объявлении которого указана возможность выброса исключения, также нужно заключить опасный код в блок try/catch.
8. Метод main() в программе должен перехватывать все необработанные исключения, которые были ему выброшены.
9. У одного блока try может быть много разных блоков catch.
10. Метод может выбрасывать только один тип исключений.
11. Блок finally выполняется независимо от того, какое исключение было выброшено.
12. Блок finally может существовать без блока try.
13. Блок try может существовать сам по себе, без блоков catch или finally.
14. Обработка исключения — это нечто вроде «проброса».
15. Порядок следования блоков catch не имеет значения.
16. Для метода, содержащего блоки try и finally, при необходимости можно объявить исключение.
17. Исключения, возникающие во время выполнения программы, должны быть *обработаны* или *объявлены*.

# Магнитики с кодом

Части рабочего Java-приложения разбросаны по всему холодильнику. Можете ли вы сгруппировать фрагменты кода так, чтобы итоговая программа выводила текст, приведенный ниже? Некоторые фигурные скобки упали на пол. Они настолько маленькие, что их нельзя поднять. Можете сами добавлять столько скобок, сколько понадобится.

```
System.out.print("r");
```

```
try {
```

```
System.out.print("t");
```

```
doRisky(test);
```

```
System.out.println("s");
```

```
} finally {
```

```
System.out.print("o");
```

```
class MyEx extends Exception { }

public class ExTestDrive {
```

```
System.out.print("w");
```

```
if ("yes".equals(t)) {
```

```
System.out.print("a");
```

```
throw new MyEx();
```

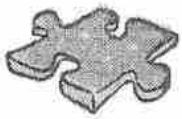
```
} catch (MyEx e) {
```

```
static void doRisky(String t) throws MyEx {
  System.out.print("h");
```

```
public static void main(String [] args) {
  String test = args[0];
```

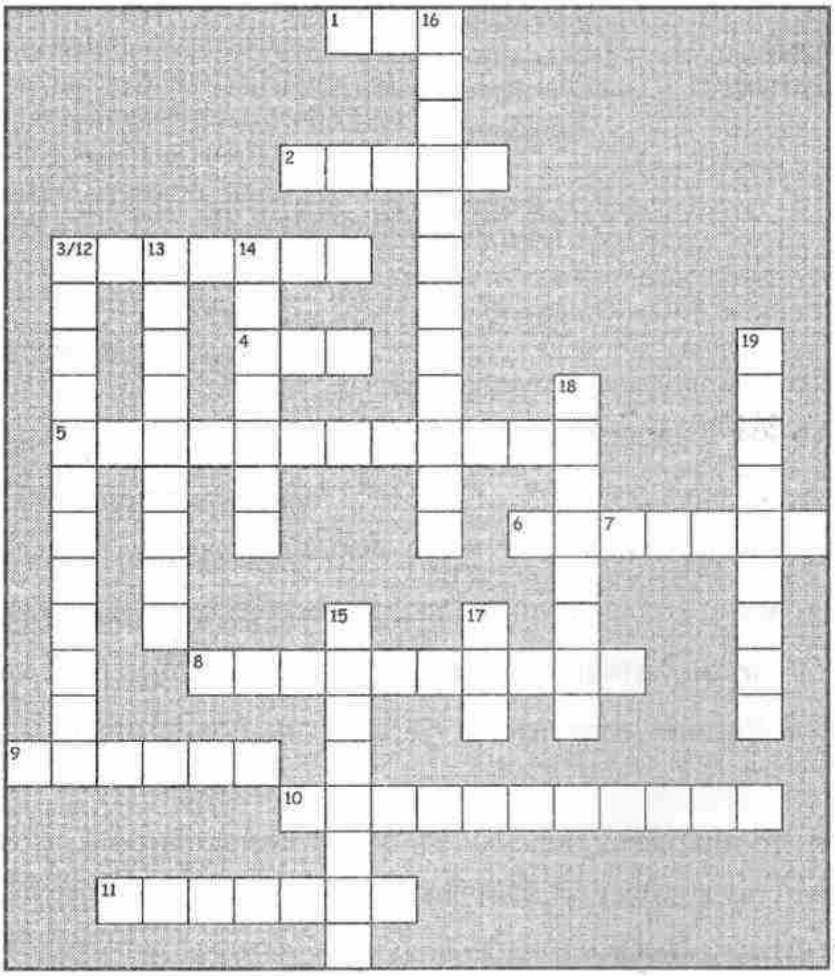
File Edit Window Help ThrowUp

```
% java ExTestDrive yes
thaws

% java ExTestDrive no
throws
```



# JavaCross 7.0

Вы знаете, что делать!




## По горизонтали

1. Рискованная попытка.
2. Шаблон.
3. Слишком горячее, чтобы обработать.
4. Обычный простой тип.
5. Несдержанный поступок метода.
6. Класс из API с преимущественно статическими методами.
7. Источник плохих новостей.
8. Проблемные объекты.
9. Метод, который меняет значение поля.
10. Java следит за такими исключениями.
11. Начать.

## По вертикали

12. Давать значение.
13. Не проброс.
14. Не показывайте его детям.
15. Рецепт для кода.
16. Приобрести автоматически.
17. Начинает цепочку событий.
18. Фамильное древо.
19. Не о поведении.

## Больше подсказок:

**По горизонтали**

2. По умолчанию он публичный.
5. С этого начинаются проблемы.
6. Числа...
9. Не геттер.

**По вертикали**

13. Вместо объявления.
16. _____ семейные богатства.

# Ответы

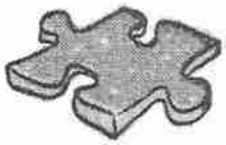
## Правда или ложь

1. Ложь. Один из двух.
2. Ложь. Вы можете объявить исключение.
3. Правда.
4. Ложь. Любые исключения могут быть перехвачены.
5. Правда.
6. Правда. Оба блока допустимы.
7. Ложь. Объявления будет достаточно.
8. Ложь. Но в противном случае JVM должна завершить свою работу.
9. Правда.
10. Ложь.
11. Правда. Этот блок часто используется для приведения в порядок незавершенных задач.
12. Ложь.
13. Ложь.
14. Ложь. «Проброс» — синоним объявления.
15. Ложь. Наиболее вместительный блок catch должен быть объявлен в последнюю очередь.
16. Ложь. Если у вас нет блока catch, вы *должны* объявить исключение.
17. Ложь.

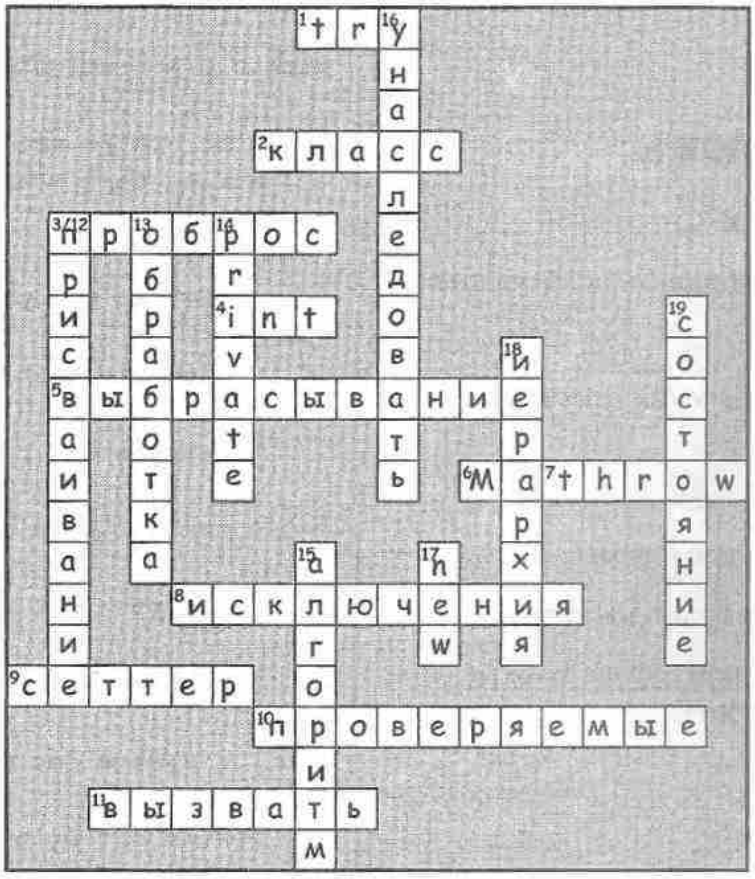
## Магнитики с кодом

```
class MyEx extends Exception { }

public class ExTestDrive {

  public static void main(String [] args) {
    String test = args[0];
    try {

       System.out.print("t");

       doRisky(test);

       System.out.print("o");

    } catch ( MyEx e) {

        System.out.print("a");

    } finally {

        System.out.print("w");
    }
    System.out.println("s");
  }

  static void doRisky(String t) throws MyEx {
    System.out.print("h");

    if ("yes".equals(t)) {

      throw new MyEx();
    }

    System.out.print("r");
```

```
Window Help Chill
% java ExTestDrive yes
thaws

% java ExTestDrive no
throws
```



# Ответы на кроссворд



# Очень графическая история

**Соберитесь, вам предстоит придумать GUI (graphical user interface — графический пользовательский интерфейс).** Если вы создаете приложение, которое начнут использовать другие люди, вам *понадобится* графический интерфейс. Даже если вы уверены, что всю оставшуюся жизнь будете писать код для серверных программ, где работающий на клиентской стороне пользовательский интерфейс представляет собой веб-страницу, рано или поздно вам придется создавать инструменты и вы захотите применить графический интерфейс. Естественно, приложения с командной строкой — это прошлый век. Они слабые, негибкие и недружелюбные. Работе над GUI посвящены две главы, из которых вы узнаете ключевые особенности языка Java, в том числе такие, как **обработка событий** и **внутренние классы**. В этой главе мы расскажем, как поместить на экран кнопку и заставить ее реагировать на нажатие. Кроме того, вы научитесь рисовать на экране, добавлять изображение в формате JPEG и даже создавать анимацию.

## Все начинается с окна

JFrame — это объект, который представляет собой окно на экране. В это окно вы будете помещать все элементы интерфейса: кнопки, флажки, поля ввода и т. п. Оно может иметь старую добрую строку меню с пунктами. Кроме того, в окне вы найдете все значки, которые выполняют функции сворачивания, разворачивания и закрытия окна; при этом не важно, какая у вас платформа.

JFrame по-разному выглядит в зависимости от платформы. Здесь представлен JFrame на Mac OS X.

JFrame со строкой меню и двумя виджетами: кнопкой и переключателем

**«Если я еще раз увижу хоть одно приложение с командной строкой, считайте, что вы уволены».**

## Помещаем виджеты в окно

При наличии JFrame вы можете поместить в окно элементы управления, добавив их в JFrame. Существует множество компонентов Swing — ищите их в пакете javax.swing. Наиболее распространенные — JButton, JRadioButton, JCheckBox, JLabel, JList, JScrollPane, JSlider, JTextArea, JTextField и JTable. Большинство из них просты в использовании, но некоторые (например, JTable) могут заставить потрудиться.

**Создавать GUI — это просто.**

1. Создаем фрейм (JFrame):
```
JFrame frame = new JFrame();
```

2. Создаем виджет (кнопку, поле ввода и т. д.):
```
JButton button = new JButton("click me");
```

3. Добавляем виджет во фрейм:
```
frame.getContentPane().add(button);
```

Вы не добавляете элементы непосредственно во фрейм. Думайте о фрейме как об оконной раме и представьте, что добавляете элементы на стекло.

4. Выводим фрейм на экран, присваиваем ему размер и делаем видимым:
```
frame.setSize(300,300);
frame.setVisible(true);
```

# Ваш первый графический интерфейс: кнопка во фрейме

```
import javax.swing.*;

public class SimpleGui1 {
    public static void main (String[] args) {

        JFrame frame = new JFrame();
        JButton button = new JButton("click me");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.getContentPane().add(button);

        frame.setSize(300,300);

        frame.setVisible(true);
    }
}
```

Импортируем пакет Swing.

Создаем фрейм и кнопку.

Передаем конструктору кнопки текст, который будет на ней отображаться.

Эта строка завершает работу программы при закрытии окна (если вы не добавите ее, то приложение будет «висеть» на экране вечно).

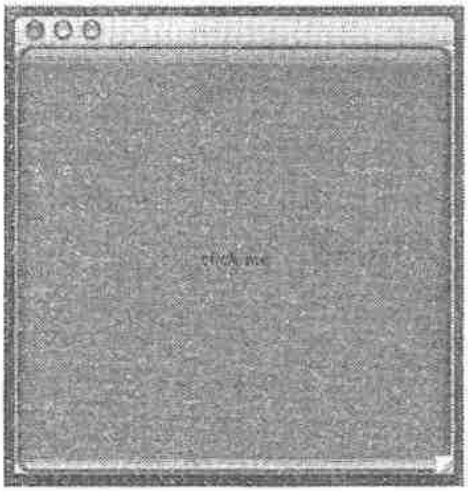
Добавляем кнопку на панель фрейма.

Присваиваем фрейму размер (в пикселах).

И наконец, делаем фрейм видимым (если вы пропустите этот шаг, то, выполнив данный код, ничего не увидите).

**Посмотрим, что произойдет при запуске:**

```
%java SimpleGui1
```



*Ого! Это действительно большая кнопка.*

Она заполняет все доступное пространство фрейма. Позже вы узнаете, как управлять положением (и размером) кнопки во фрейме.

## Но когда я нажимаю ее, ничего не происходит...

Это не совсем верно. Когда вы нажимаете кнопку, она меняет свой вид — как бы вдавливается (вид изменяется в зависимости от платформы, но кнопка всегда делает хоть *что-то*, чтобы показать, что она нажата).

На самом деле вопрос звучит так: «Как мне заставить кнопку делать что-либо в тот момент, когда пользователь нажимает ее?»

**Понадобятся две вещи.**

1. ***Метод***, вызываемый при нажатии (действие, которое должно произойти в результате нажатия кнопки).

2. *Способ* ***узнавать***, когда запускать данный метод. Другими словами, это способ распознавать, когда пользователь нажимает кнопку.

Мы хотим знать, когда пользователь нажимает кнопку.

Нас интересует событие, происходящее в момент воздействия на кнопку.

### Это не глупые вопросы

**В:** **Если я запущу программу в Windows, будет ли кнопка выглядеть так, как другие кнопки в этой системе?**

**О:** Да. Можете выбирать среди нескольких классов «внешнего вида» в основной библиотеке, которая контролирует, каким будет интерфейс. В большинстве случаев допускается выбор по крайней мере между двумя видами: стандартным оформлением Java, также известным как ***Metal***, и родным оформлением для вашей платформы. Изображения на экране Mac OS X в данной книге используют оформление OS X ***Aqua*** или ***Metal***.

**В:** **Могу ли я сделать так, чтобы программа всегда имела вид Aqua? Даже если она запускается в Windows?**

**О:** Нет. Не все варианты оформления доступны на каждой платформе. Если вы не хотите рисковать, то можете конкретно установить внешний вид Metal, чтобы точно знать, чего ожидать, несмотря на платформу, где запускается приложение. Кроме того, вы можете не определять внешний вид и принять вариант, предлагаемый системой.

**В:** **Я слышал, что Swing работает медленно и его никто не использует.**

**О:** Так было раньше, но сейчас все изменилось. На слабых машинах вы еще можете испытать трудности со Swing. Однако на современных ПК и в Java версии 1.3 и выше вы, скорее всего, даже не почувствуете разницу между Swing GUI и стандартным графическим интерфейсом. Сегодня Swing широко используется во всех видах приложений.

# Получаем пользовательское событие

Допустим, вы хотите, чтобы текст на кнопке изменялся с *click me* (Нажми меня) на *I've been clicked* (Меня нажали) при нажатии пользователем этой кнопки. Сначала можно написать метод, который будет менять текст (быстро просмотрев API, вы найдете этот метод):

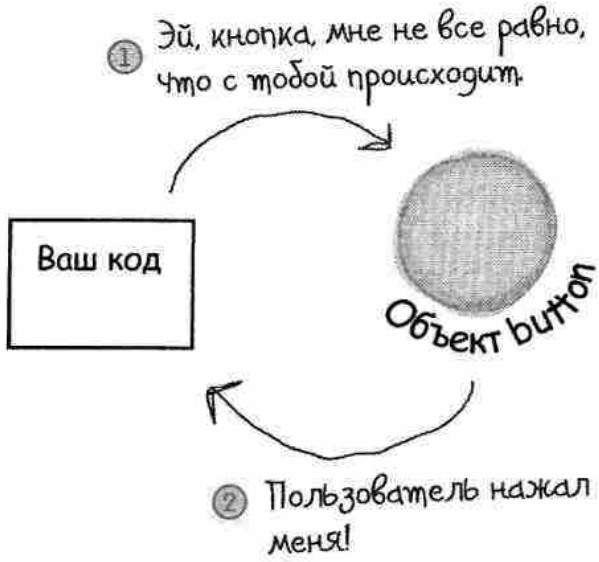
```
public void changeIt() {
     button.setText("I've been clicked!");
}
```

Но *что* теперь? Как мы узнаем, когда запускать этот метод? ***Как мы узнаем, что кнопка нажата?***

В языке Java процесс получения пользовательского события и управление им называется *обработкой событий*. Существует множество различных типов событий, несмотря на то что большинство из них касаются пользовательских действий в GUI. Когда пользователь нажимает кнопку, происходит событие, которое говорит: «Пользователь хочет, чтобы выполнилось действие, закрепленное за этой кнопкой». Если это кнопка Slow Tempo (Замедлить темп), то пользователь хочет, чтобы темп замедлился. Если это кнопка Send (Отправить) в клиенте чата, то пользователь хочет, чтобы его сообщение отправилось. Таким образом, наиболее простое событие — когда пользователь нажимает кнопку, сигнализируя о необходимости выполнить определенное действие.

При работе с кнопками обычно не важны переходные события, например, когда кнопку только начинают нажимать или уже отпустили. Мы лишь хотим сказать кнопке: «Меня не волнует, как пользователь станет играть с тобой, как долго будет держать над тобой указатель мыши, сколько раз поменяет свое решение и произнесет его вслух, перед тем как отпустит тебя, и т. д. ***Просто дай знать, когда он перейдет к делу!*** Иначе говоря, не зови меня, пока пользователь не нажмет тебя, то есть пока не захочет, чтобы ты сделала то, что обещаешь сделать!»

**Во-первых, кнопке нужно знать о том, что нас интересует.**




**Во-вторых, кнопке нужен способ вызвать нас, когда произойдет событие — ее нажатие.**

1. Как сообщить объекту button, что вас интересуют его события? Как сказать, что вы — заинтересованный слушатель?

2. Каким образом кнопка будет выполнять обратный вызов? Предположим, что нет способа сообщить ей имя уникального метода (changeIt()). Как еще убедить кнопку, что у нас есть особенный метод, который она может вызвать при наступлении события?

**Если вас интересуют события кнопки, реализуйте интерфейс, который скажет: «Я отслеживаю твои события».**

**Такой интерфейс станет мостом между отслеживающей стороной (вами) и источником события (кнопкой).**

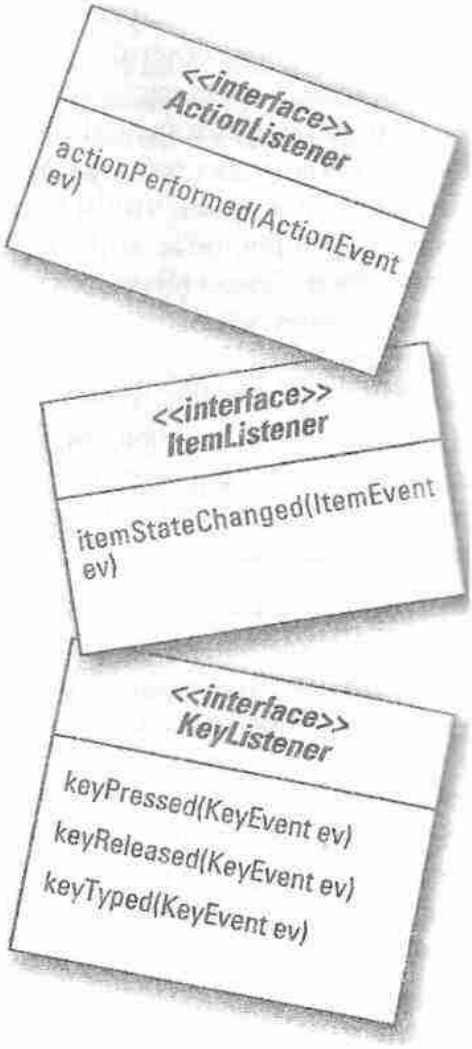
Графические компоненты в Swing — это источники событий. Выражаясь терминами языка Java, источник события представляет собой объект, который может преобразовывать действия пользователя (щелчок кнопкой мыши, нажатие клавиши, закрытие окна) в события. Как и фактически все остальное в Java, событие представляется в виде объекта, в данном случае принадлежащего какому-либо классу событий. Если вы просмотрите пакет java.awt.event в API, то увидите набор событийных классов (их легко найти — в названии каждого есть слово ***Event***). Там вы найдете MouseEvent, KeyEvent, WindowEvent, ActionEvent и несколько других.

***Источник*** события (например, кнопка) создает ***объект события***, когда пользователь делает что-нибудь, имеющее значение (например, *нажимает* кнопку). Большая часть кода, которую вы напишете (а также весь код в этой книге) будет *получать* события, а не *создавать* их. Другими словами, вы будете проводить основную часть своего времени в качестве *слушателя* событий, а не как их *источник*.

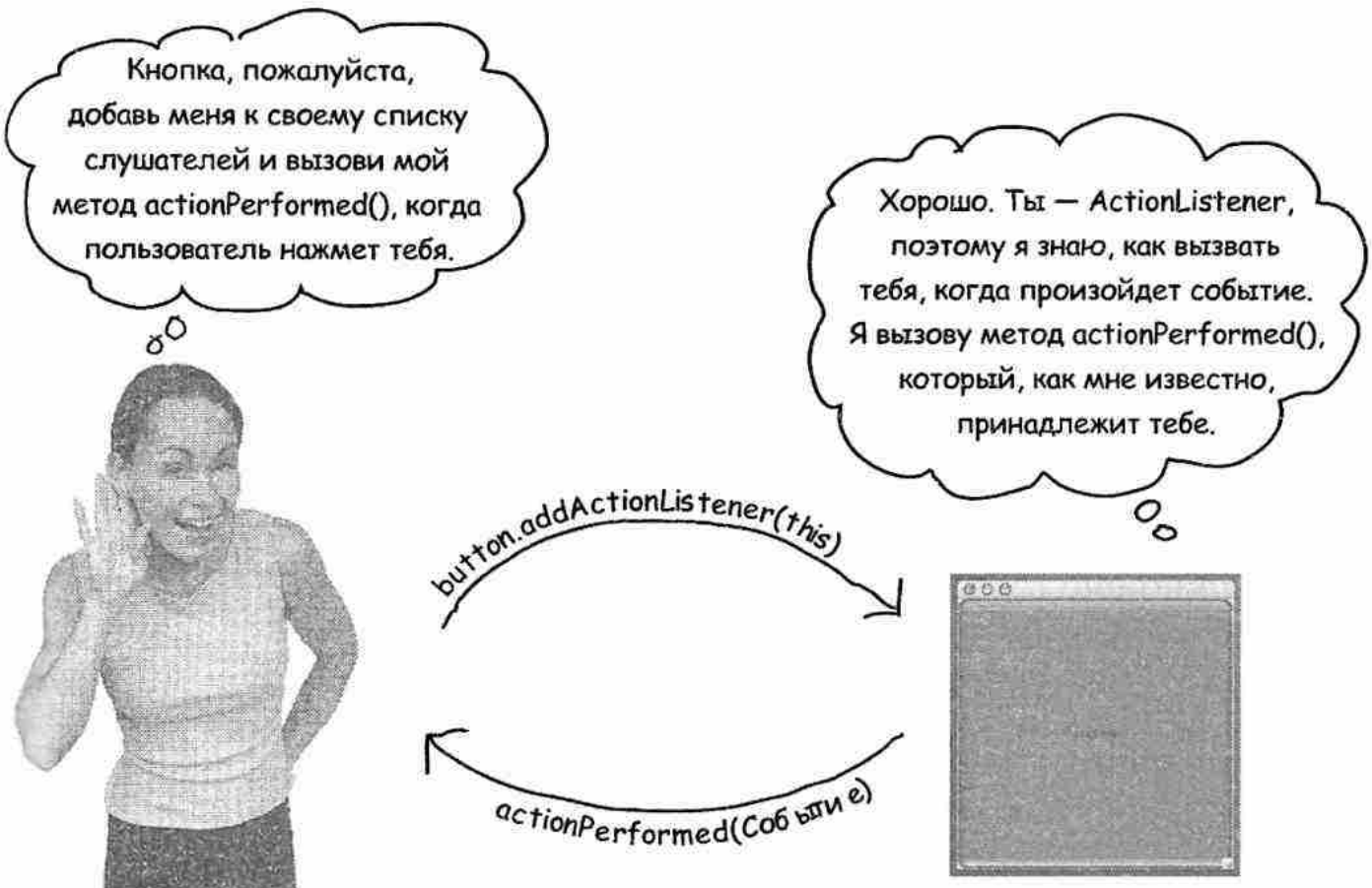
Каждый тип события имеет соответствующий интерфейс для отслеживания. Если вам нужно событие MouseEvent, реализуйте интерфейс MouseListener. Хотите WindowEvent? Реализуйте WindowListener. Идея понятна. И помните правило интерфейсов — при реализации интерфейса вы объявляете об этом. Это означает, что вы должны написать способы реализации для каждого метода в интерфейсе.

Некоторые интерфейсы содержат несколько методов, потому что само событие бывает разных типов. Например, если вы реализуете MouseListener, то можете получить событие для mousePressed (кнопка мыши нажата), mouseReleased (кнопка мыши отпущена), mouseMoved (мышь двигается) и т. д. Каждое из событий мыши имеет отдельный метод в интерфейсе, несмотря на то что они все принимают MouseEvent. Если вы реализуете интерфейс MouseListener, то метод mousePressed() будет вызываться при нажатии пользователем кнопки мыши. Когда пользователь отпустит кнопку, сработает метод mouseReleased(). Таким образом, для событий мыши существует только один объект *события* — MouseEvent, но при этом есть несколько различных *методов*, представляющих различные *типы* событий мыши.

Реализуя интерфейс для отслеживания, вы предоставляете кнопке возможность обратного вызова. Интерфейс — это место, где объявляется метод обратного вызова.



## Как слушатель и источник обмениваются информацией




### Слушатель

Если ваш класс хочет знать о событиях ActionEvent, принадлежащих кнопке, реализуйте интерфейс ActionListener. Кнопке нужно знать, что вы заинтересованы в ее событиях, поэтому зарегистрируйте ее с помощью метода addActionListener(this) и передайте ей ссылку на ActionListener (в данном случае вы — ActionListener и передаете this). Кнопке нужен способ вас вызвать, когда произойдет событие, поэтому она вызывает метод из интерфейса слушателя. Как ActionListener, вы должны реализовать единственный метод интерфейса — actionPerformed(). Компилятор обеспечит это.

### Источник события

Кнопка — это источник событий ActionEvent, поэтому она должна знать, какие объекты являются заинтересованными слушателями. Существует метод addActionListener(), который обеспечивает заинтересованным объектам (слушателям) способ *сообщить* кнопке об этом интересе.

При запуске метода кнопки addActionListener() (поскольку потенциальный слушатель вызвал его) кнопка принимает параметр (ссылку на объект-слушатель) и сохраняет его в список. Когда пользователь нажимает кнопку, она запускает событие, вызывая метод actionPerformed(), принадлежащий каждому слушателю в списке.

# Получение событий ActionEvent кнопки

1. Реализуем интерфейс ActionListener.
2. Регистрируем кнопку (сообщаем ей, что хотим отслеживать ее события).
3. Определяем метод обработки событий (реализуем метод actionPerformed() из интерфейса ActionListener).

```
import javax.swing.*;
import java.awt.event.*;

public class SimpleGui1B implements ActionListener {
    JButton button;

    public static void main (String[] args) {
       SimpleGui1B gui = new SimpleGui1B();
       gui.go();
     }

    public void go() {
       JFrame frame = new JFrame();
       button = new JButton("click me");

       button.addActionListener(this);

       frame.getContentPane().add(button);
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       frame.setSize(300,300);
       frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
       button.setText("I've been clicked!");
    }
}
```

Новый оператор импорта для пакета, в котором хранятся ActionListener и ActionEvent.

① Реализуем интерфейс. Код говорит: «Экземпляр SimpleGui1B реализует интерфейс ActionListener».

Кнопка будет передавать события только тем, кто реализует ActionListener.

② Регистрируем нашу заинтересованность в кнопке. Код говорит кнопке: «Добавь меня к своему списку слушателей». Передаваемый аргумент ДОЛЖЕН быть объектом класса, реализуемого ActionListener!

③ Реализуем метод actionPerformed() интерфейса ActionListener. Это фактический метод обработки событий!

Кнопка вызывает этот метод, чтобы известить о наступлении события. Она отправляет объект ActionEvent как аргумент, но он нам не нужен. Достаточно знать, что событие произошло.

# Слушатели, источники и события

Большую часть вашей звездной карьеры в качестве программиста на языке Java вы не будете выступать в роли *источника* событий. Не важно, как сильно вы представляете себя центром своей социальной вселенной. Привыкайте к этому. ***Ваша работа — быть хорошим слушателем.*** И если вы будете выполнять ее добросовестно, это улучшит вашу общественную жизнь.

Моя работа в качестве слушателя заключается в **реализации** интерфейса, **регистрации** кнопки и **обеспечении** обработки событий.

**Слушатель получает событие.**

Моя работа в качестве источника событий заключается в **приеме** запросов на регистрацию (от слушателей), **получении** событий от пользователя и **вызове** метода обработки событий слушателя (когда пользователь нажимает меня).

**Источник отправляет событие.**

Эй, а как же я? Я тоже участвую, и вы это знаете! Я представляю собой **аргумент** метода обратного вызова события (из интерфейса), и моя работа заключается в **передаче данных** о событии обратно к слушателю.

**Объект события содержит данные о событии.**

**В:** **Почему я не могу быть источником событий?**

**О:** Можете. Мы просто сказали, что большую часть времени вы будете получателем, а не создателем события (по крайней мере в начале своей блестящей карьеры Java-программиста). Большинство событий, которые могут вас заинтересовать, запускаются классами в API Java, и вы должны быть только их слушателем. Однако можно разработать программу, где понадобится пользовательское событие, скажем, StockMarketEvent. Оно будет вызываться, например, когда ваше приложение по наблюдению за фондовой биржей найдет что-нибудь, по ее мнению, важное. В этом случае вы создаете объект StockWatcher — источник события и делаете то же самое, что делает кнопка (или любой другой источник). Иными словами, создаете интерфейс слушателя для своего пользовательского события, регистрируете метод (addStockListener()), а когда кто-нибудь вызывает его, добавляете вызвавшую сторону в список слушателей. Затем, когда происходит событие, создаете экземпляр класса StockEvent (еще один класс, который вы напишете) и отправляете его слушателям в своем списке, вызывая их метод stockChanged(StockEvent). И не забудьте, что для каждого *типа события* должен быть *свой интерфейс слушателя* (поэтому вы создаете интерфейс StockListener с методом stockChanged()).

## Это не глупые вопросы

**В:** **Я не понимаю важности объекта события, который передается методом обратного вызова. Если кто-нибудь вызывает мой метод mousePressed, какая информация мне может понадобиться?**

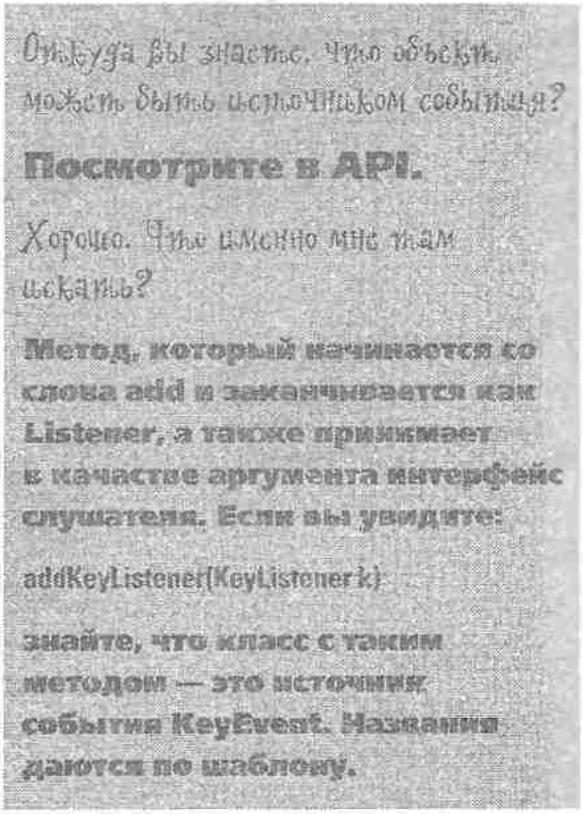
**О:** В большинстве разработок вам не понадобится объект события. Это всего лишь маленький носитель данных, который не предназначен для отправки больших объемов информации о событии. Но иногда вам потребуются определенные подробности о событии. Например, если вызывается ваш метод mousePressed то вы знаете, что кнопка мыши нажата. А если вы хотите точно знать, где был указатель мыши в это время? Другими словами, что, если вы хотите знать координаты X и Y экрана, где была нажата кнопка мыши?

Или, допустим, вы захотите зарегистрировать *одного* слушателя для *множества* объектов. Например, экранный калькулятор имеет 10 числовых кнопок. Поскольку они делают одно и то же, вы, возможно, не захотите создавать отдельных слушателей для каждой кнопки. Вместо этого вы могли бы зарегистрировать одного слушателя для всех 10 кнопок и при получении события (потому что был вызван метод обратного вызова) вызвать метод объекта события, чтобы выяснить, кто был его источником. Другими словами, узнать, *какая кнопка отправила это событие*.

### Наточите свой карандаш

Каждый из этих виджетов (элементов пользовательского интерфейса) представляет собой источник одного или нескольких событий. Соединяйте виджеты с теми событиями, которые они могут вызвать. Некоторые виджеты могут быть источником нескольких событий, а отдельные события могут быть вызваны несколькими виджетами.

| Виджеты | Методы событий |
|---|---|
| флажок | windowClosing() |
| поле ввода | actionPerformed() |
| прокручиваемый список | ItemStateChanged() |
| кнопка | mousePressed() |
| диалоговое окно | keyTyped() |
| переключатель | mouseExited() |
| пункт меню | focusGained() |



# Вернемся к графике...

Теперь, когда вы немного знаете, как работают события (позже вы узнаете еще больше), вернемся к размещению объектов на экране. Потратим несколько минут на интересные способы вывода графики, перед тем как вновь перейти к обработке событий.

## Рассмотрим три способа поместить объекты в GUI.

**1 Размещаем виджеты во фрейме.**
Добавляем кнопки, меню, переключатели и т. д.

```
frame.getContentPane().add(myButton);
```

Пакет javax.swing содержит больше дюжины типов виджетов.

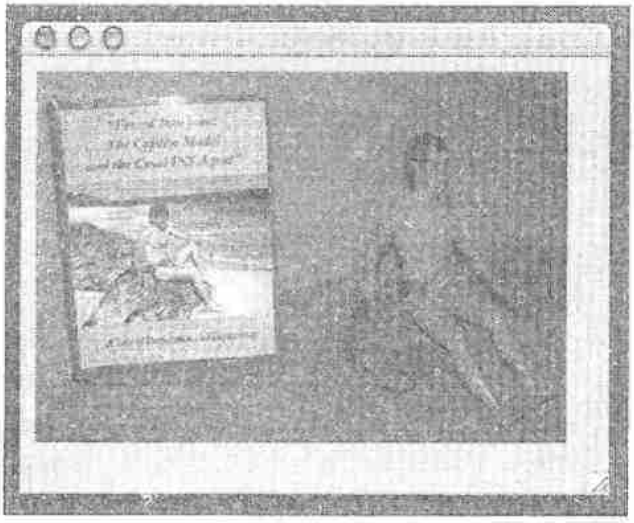
**2 Рисуем на виджете двумерную графику.**
Используем графические объекты для рисования фигур.

```
graphics.fillOval(70,70,100,100);
```

Вы можете рисовать не только квадраты и круги: Java2D API полон интересных и изысканных графических методов.

Искусство, игры, моделирование и т. д.

Диаграммы, деловые графики и т. д.



**3 Помещаем JPEG на виджет**
Можете вставить на виджет собственное изображение.

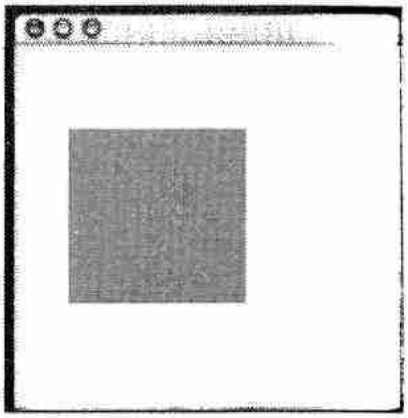
```
graphics.drawImage(myPic,10,10,this);
```

# Создайте личный виджет для рисования

Если вы хотите добавить собственную графику, лучше всего создать личный виджет для рисования. Его нужно поместить во фрейм, как кнопку или любой другой виджет, и, когда вы выведете его на экран, увидите свои изображения. Вы можете даже заставить их двигаться с помощью анимации или менять цвета на экране каждый раз, когда нажимаете кнопку.

Это проще простого.

### Создайте подкласс JPanel и замените один метод под названием paintComponent().



Весь ваш графический код располагается внутри метода paintComponent(). Считайте, что этот метод вызывается системой, чтобы сказать: «Эй, виджет, пришло время нарисовать себя». Если вы хотите нарисовать круг, то метод paintComponent() будет содержать код для рисования круга. Когда фрейм, содержащий вашу панель для отрисовки, будет показан на экране, будет вызван метод paintComponent() и появится круг. Если пользователь свернет окно, то JVM будет знать, что фрейму нужно «восстановиться» при разворачивании, и снова вызовет метод paintComponent(). В любое время, когда JVM посчитает, что экран нужно обновить, будет вызван ваш метод paintComponent().

Помните еще кое-что: ***вы никогда не будете вызывать этот метод сами!*** Аргумент для него (объект Graphics) — это фактически холст для рисования, который наложили на *настоящий* экран. Вы не сможете добиться этого самостоятельно: он должен быть передан вам системой. Однако позже вы увидите, что можете запрашивать систему на обновление дисплея (repaint()), что в итоге приведет к вызову метода paintComponent().

```
import java.awt.*;
import javax.swing.*;

class MyDrawPanel extends JPanel {

    public void paintComponent(Graphics g) {

        g.setColor(Color.orange);

        g.fillRect(20,50,100,100);
    }
}
```

Вам понадобятся оба этих пакета.

Создаем наследника JPanel — виджет, который вы сможете добавить во фрейм как любой другой. Только это будет ваш личный пользовательский виджет.

Это большой важный графический метод. Вы никогда не будете вызывать его сами. Система вызывает его и говорит: «Вот прекрасная новая поверхность для рисования, принадлежащая типу Graphics, и ты можешь на ней рисовать».

Представьте, что g — это машина для рисования. Вы сообщаете ей, каким цветом и какую фигуру нарисовать (с координатами местоположения и размерами).

# Что интересного можно сделать в paintComponent()

Рассмотрим несколько вещей, которые можно сделать в paintComponent(). Хотя интереснее будет, если вы сами начнете экспериментировать. Поиграйте с числами и поищите в API класс Graphics (позже вы увидите, что ваши возможности далеко не ограничиваются этим классом).

## Изображаем JPEG

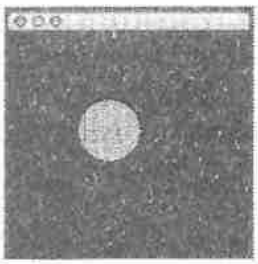
```
public void paintComponent(Graphics g) {

    Image image = new ImageIcon("catzilla.jpg").getImage();

    g.drawImage(image,3,4,this);

}
```

Имя вашего файла указывается здесь.

Координаты x и y для верхнего левого угла картинки. Код говорит: «Отступить 3 пиксела от левого края панели и 4 пиксела от верхнего». Эти числа всегда относятся к виджету (в данном случае к вашему наследнику JPanel), а не ко всему фрейму.

---

## Рисуем на черном фоне круг произвольного цвета



```
public void paintComponent(Graphics g) {

    g.fillRect(0,0,this.getWidth(), this.getHeight());

    int red = (int) (Math.random() * 255);
    int green = (int) (Math.random() * 255);
    int blue = (int) (Math.random() * 255);

    Color randomColor = new Color(red, green, blue);
    g.setColor(randomColor);
    g.fillOval(70,70,100,100);
}
```

Закрасим всю панель черным (цвет по умолчанию).

Два первых аргумента определяют координаты верхнего левого угла по отношению к панели, где начнется рисование. Здесь 0,0 означает: «Начни с 0 пикселов от левого края и 0 пикселов от верхнего». Два других аргумента говорят: «Сделай ширину прямоугольника, как у панели (this.width()), и высоту такую же, как у панели (this.height())».

Вы можете задать цвет тремя целыми числами, представляющими RGB-значения.

Начинаем рисование с 10 пикселов слева и 10 пикселов сверху, а также задаем ширину и высоту по 100 пикселов.

# За каждой хорошей графической ссылкой стоит объект Graphics2D

Аргумент для метода paintComponent() имеет тип Graphics (java.awt.Graphics).

```
public void paintComponent(Graphics g) { }
```

Здесь параметр g — экземпляр класса Graphics. Это означает, что он *может* быть *наследником* класса Graphics (из-за полиморфизма). И в действительности он им является.

***Объект, на который ссылается параметр g, фактически является экземпляром класса Graphics2D.***

Почему это важно? Потому что есть вещи, которые вы можете делать со ссылкой типа Graphics2D и не могли бы делать со ссылкой Graphics. Объект Graphics2D может делать больше, чем объект Graphics, и в действительности объект Graphics2D притаился за ссылкой Graphics.

Вспомните о полиморфизме. Какой метод вы можете вызвать, компилятор решает, основываясь на типе ссылки, а не на типе объекта. Если у вас есть объект Dog, на который указывает ссылка типа Animal:

```
Animal a = new Dog();
```

то вы НЕ сможете сказать:

```
a.bark();
```

Компилятор смотрит на a, видит, что его тип — Animal, и выясняет, что в этом классе нет удаленной кнопки управления для bark(). Но вы все еще можете привести объект к типу Dog, чем он и является, написав так:

```
Dog d = (Dog) a;
d.bark();
```

Суть объекта Graphics заключается в следующем:

**Если вам нужно использовать метод из класса Graphics2D, то нельзя *задействовать* параметр (g) метода paintComponent() напрямую. Вы можете *указать* его с новой переменной и определить тип Graphics2D.**

```
Graphics2D g2d = (Graphics2D) g;
```

**Методы, которые можно вызвать с помощью ссылки типа Graphics:**

- drawImage()
- drawLine()
- drawPolygon
- drawRect()
- drawOval()
- fillRect()
- fillRoundRect()
- setColor()

**Чтобы привести объект Graphics2D к ссылке типа Graphics2D, напишите следующее:**

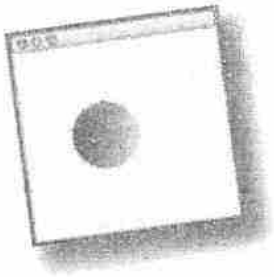
```
Graphics2D g2d = (Graphics2D) g;
```

**Методы, которые можно вызвать с помощью ссылки типа Graphics2D:**

- fill3DRect()
- draw3DRect()
- rotate()
- scale()
- shear()
- transform()
- setRenderingHints()

Это не полный перечень методов. За более подробной информацией обратитесь к API.

# Жизнь слишком коротка, чтобы тратить ее на рисование одноцветных кругов, когда есть плавный градиент



```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;
    GradientPaint gradient = new GradientPaint(70,70,Color.blue, 150,150, Color.orange);
    g2d.setPaint(gradient);
    g2d.fillOval(70,70,100,100);
}
```

Это настоящий объект Graphics2D, замаскированный под простой объект Graphics.

Указываем его, чтобы иметь возможность вызвать нечто такое, что есть у Graphics2D, но отсутствует у Graphics.

Начальная точка. Начальный цвет. Конечная точка. Конечный цвет.

Здесь мы назначаем для виртуальной кисти градиент вместо сплошного цвета.

Метод fillOval() в данном случае позволяет закрасить овал тем, что находится на кисти — градиентом.

---

```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;

    int red = (int) (Math.random() * 255);
    int green = (int) (Math.random() * 255);
    int blue = (int) (Math.random() * 255);
    Color startColor = new Color(red, green, blue);

    red = (int) (Math.random() * 255);
    green = (int) (Math.random() * 255);
    blue = (int) (Math.random() * 255);
    Color endColor = new Color(red, green, blue);

    GradientPaint gradient = new GradientPaint(70,70,startColor, 150,150, endColor);
    g2d.setPaint(gradient);
    g2d.fillOval(70,70,100,100);
}
```

Это такой же код, что и сверху, только в нем начальные и конечные цвета для градиента выбираются произвольно. Попробуйте!

## КЛЮЧЕВЫЕ МОМЕНТЫ

### События

- Чтобы создать GUI, начните с окна. Как правило, это JFrame:

```
JFrame frame = new JFrame();
```

- Вы можете добавлять виджеты (кнопки, поля ввода и т. д.) в JFrame, используя запись вида:

```
frame.getContentPane().add(button);
```

- В отличие от большинства других компонентов, JFrame не позволяет добавлять в него объекты напрямую, поэтому приходится делать это через панель содержимого, принадлежащую JFrame.
- Чтобы вывести на экран окно (JFrame), нужно сделать его видимым и указать размер:

```
frame.setSize(300,300);
frame.setVisible(true);
```

- Для отслеживания событий нужно обозначить свой интерес к источнику события. Источник события — это объект (кнопка, переключатель и т. д.), который «запускает» событие, основываясь на действиях пользователя.
- Интерфейс слушателя предоставляет источнику события возможность обратного вызова, так как интерфейс определяет метод, вызываемый источником события при совершении события.
- Чтобы связать события с источником, вызовите регистрационный метод источника. Методы для регистрации всегда принимают следующий вид: **add<ТипСобытия>Listener**. Например, для регистрации события ActionEvent, генерируемого кнопкой, вызовите:

```
button.addActionListener(this);
```

- Создавайте интерфейс слушателя, реализуя все методы для обработки событий интерфейса. Поместите свой код обработки событий в метод обратного вызова слушателя. Для ActionEvent это такой метод:

```
public void actionPerformed(ActionEvent
                                    event) {
    button.setText("you clicked!");
}
```

### Графика

- Объект события передается в метод обработки события и несет с собой информацию о событии, в том числе его источник.
- Вы можете рисовать двумерную графику прямо на виджете.
- Вы можете помещать GIF- или JPEG-изображения прямо на виджет.
- Чтобы рисовать собственные изображения (включая GIF или JPEG), создайте подкласс JPanel и замените метод paintComponent().
- Метод paintComponent() вызывается графической системой. **Вы никогда не вызывайте его сами.** Аргументом для paintComponent() является объект Graphics, который предоставляет поверхность для рисования, размещаемую на экране. Вы не можете сами создать этот объект.
- Из объекта Graphics (параметра paintComponent) вызываются следующие типичные методы:

```
graphics.setColor(Color.blue);
g.fillRect(20,50,100,120);
```

- Чтобы создать JPG-изображение, добавьте объект Image:

```
Image image = new
ImageIcon("catzilla.jpg").
getImage();
```

и нарисуйте изображение:

```
g.drawImage(image,3,4,this);
```

- Объект, ссылка на который имеет тип Graphics и передается в качестве параметра для метода paintComponent(), фактически является экземпляром класса Graphics2D. Этот класс содержит множество методов, включая такие: fill3DRect(), draw3DRect(), rotate(), scale(), shear(), transform()
- Для вызова методов, предоставляемых объектом Graphics2D, нужно перевести тип параметра из Graphics в Graphics2D:

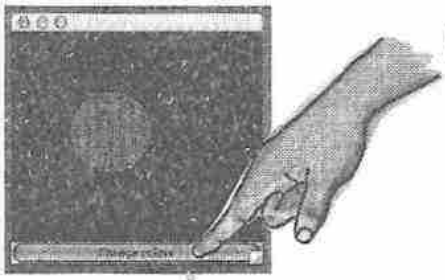
```
Graphics2D g2d = (Graphics2D) g;
```

# Мы можем получить событие.
# Мы можем рисовать графику.
# Но можем ли мы рисовать графику при получении события?

Соединим событие с определенным действием на нашей панели рисования. Заставим круг менять цвет при каждом нажатии кнопки. Рассмотрим, как будет работать программа.

Запускаем приложение.

1. Создается фрейм с двумя виджетами (панелью рисования и кнопкой). Создается слушатель, который связывается с кнопкой. Затем фрейм выводится на экран и просто ждет, когда пользователь нажмет кнопку.



2. Пользователь нажимает кнопку, она создает объект события и вызывает обработчик события, принадлежащий слушателю.

3. Обработчик события запускает метод repaint(), принадлежащий фрейму. Система вызывает метод paintComponent() в контексте панели для отрисовки.

4. Вуаля! Появляется новый цвет, так как метод paintComponent() вновь запустился и закрасил круг произвольным цветом.

# Компоновка графических элементов: помещаем несколько виджетов во фрейм

Мы подробно опишем компоновку графических элементов в *следующей* главе, а сейчас лишь бегло рассмотрим их, чтобы вы могли двигаться дальше. По умолчанию фрейм имеет пять областей, на которые можно добавлять объекты. Разрешается добавить только *один* объект на область, но только без паники! Этим объектом может быть панель, которая будет содержать три других объекта, включая панель, содержащую еще два объекта, и т. д. В действительности мы схитрили, когда добавляли кнопку во фрейм таким образом:

```
frame.getContentPane().add(button);
```

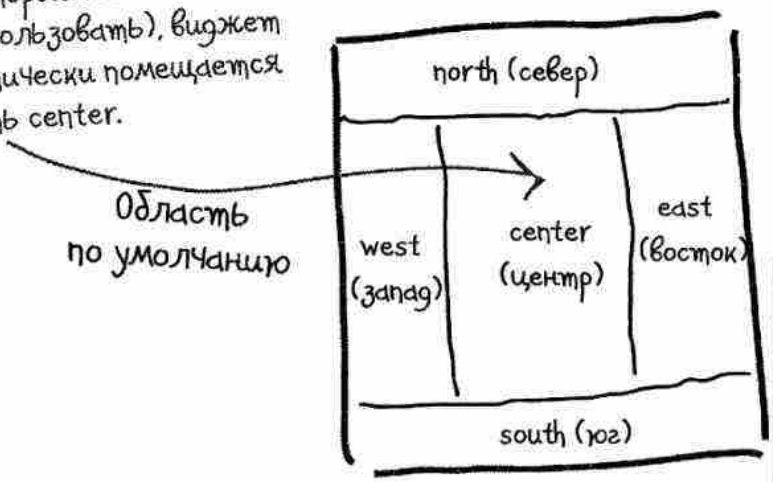
Это не тот способ, который нужно использовать (метод добавления с одним аргументом).

Такой способ лучше подходит (и обычно является обязательным) для добавления на панель содержимого фрейма, установленную по умолчанию. Всегда отмечайте, куда (в какую область) вы хотите поместить виджет.

```
frame.getContentPane().add(BorderLayout.CENTER, button);
```

Мы вызываем метод add с двумя аргументами. Он принимает область (используя константу) и виджет, который будет добавлен в эту область.

Когда вы вызываете метод добавления с одним аргументом (который нам не следует использовать), виджет автоматически помещается в область center.




## Наточите свой карандаш

Согласно изображениям на странице 399 напишите код, который будет добавлять во фрейм кнопку и панель.

## Круг меняет цвет при каждом нажатии кнопки

Пользовательская панель для рисования (экземпляр MyDrawPanel) находится в области CENTER фрейма.

Кнопка в области SOUTH фрейма.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class SimpleGui3C implements ActionListener {

    JFrame frame;

    public static void main (String[] args) {
       SimpleGui3C gui = new SimpleGui3C();
       gui.go();
    }

    public void go() {
       frame = new JFrame();
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       JButton button = new JButton("Change colors");
       button.addActionListener(this);

       MyDrawPanel drawPanel = new MyDrawPanel();

       frame.getContentPane().add(BorderLayout.SOUTH, button);
       frame.getContentPane().add(BorderLayout.CENTER, drawPanel);
       frame.setSize(300,300);
       frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
        frame.repaint();
    }
}
```

Добавляем слушателя (this) к кнопке.

Добавляем два виджета (кнопку и панель для рисования) в две области фрейма.

Когда пользователь нажимает кнопку, вызываем для фрейма метод repaint(). Это значит, что метод paintComponent() вызывается для каждого виджета во фрейме!

---

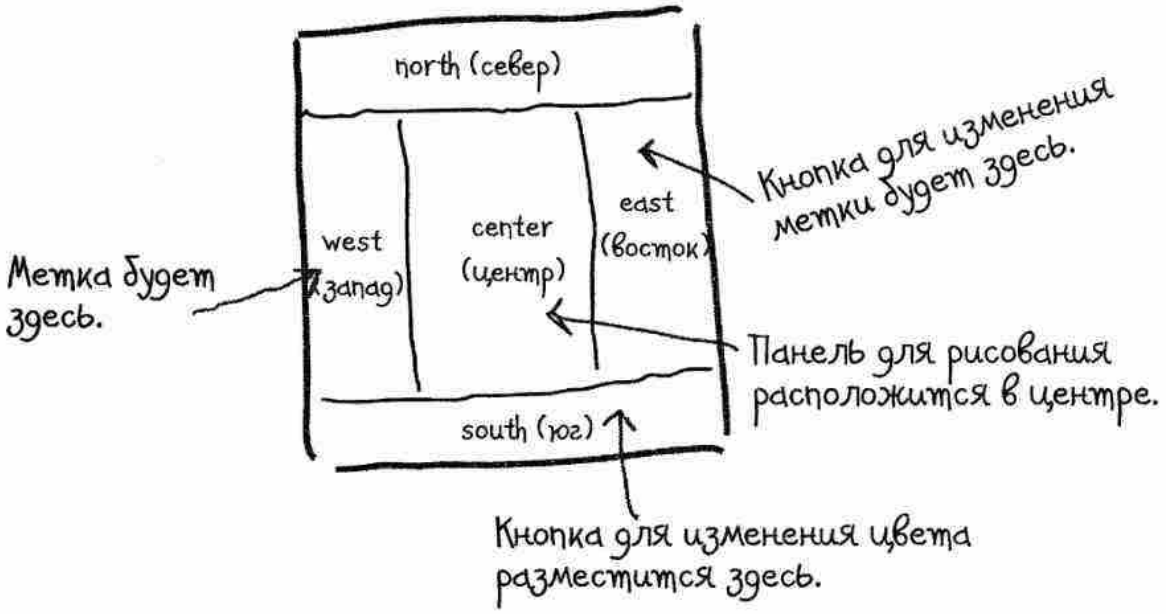
```
class MyDrawPanel extends JPanel {

   public void paintComponent(Graphics g) {
     // Код для заполнения овала произвольным цветом
     // Смотрите его на странице 395
   }

}
```

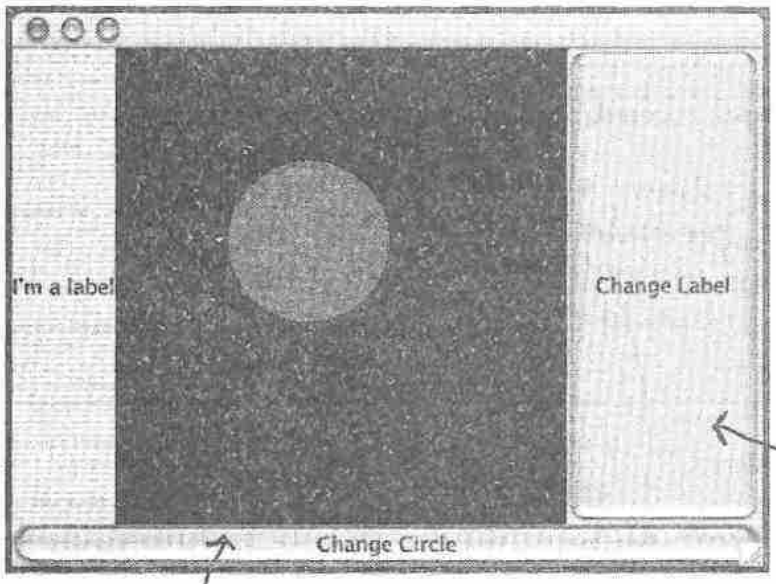
Метод панели для рисования paintComponent() вызывается при каждом нажатии кнопки.

# Попробуем сделать это с двумя кнопками

Кнопка в области south будет делать то же, что и сейчас, — вызывать перерисовку фрейма. Вторая кнопка (которую мы поместим в область east) будет менять содержимое метки, которая представляет собой текст на экране.

# Теперь нам понадобятся четыре виджета




# И нам нужно получить два события

Ой! Разве так можно? Как вы получите *два* события, когда у вас есть только *один* метод actionPerformed()?

ак вы получите события для двух разных кнопок, когда аждая кнопка делает что-то свое?

## ① Способ первый. Реализуем два метода actionPerformed().

```
class MyGui implements ActionListener {
    // Много кода здесь, а затем:

    public void actionPerformed(ActionEvent event) {
        frame.repaint();
    }

    public void actionPerformed(ActionEvent event) {
        label.setText("Больно!");
    }
}
```

Но это невозможно!

**Недостаток: вы не можете этого сделать!** Нельзя реализовать один и тот же метод дважды в одном классе в Java. Такой код не скомпилируется. И даже если бы у вас была такая *возможность*, как бы источник события узнал, *какой* из двух методов ему вызывать?

## ② Способ второй. Регистрируем одного слушателя для обеих кнопок.

```
class MyGui implements ActionListener {
    // Здесь мы объявляем множество полей

    public void go() {
        // Создаем gui
        colorButton = new JButton();
        labelButton = new JButton();
        colorButton.addActionListener(this);
        labelButton.addActionListener(this);
        // Еще больше графического кода здесь...
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == colorButton) {
            frame.repaint();
        } else {
            label.setText("Больно!");
        }
    }
}
```

Регистрируем одного слушателя для обеих кнопок.

Запрашиваем объект события, чтобы выяснить, какая именно кнопка инициировала его появление, и используем эту информацию для принятия решения о дальнейших действиях.

**Недостаток: это будет работать, но в большинстве случаев это не совсем объектно ориентированное программирование.** Один обработчик событий, выполняющий множество разных действий, означает, что у вас лишь один метод для всех этих действий. Если вам понадобится внести изменения в способ обработки события *одного* источника, то придется разбираться и с *остальными* обработчиками событий. Иногда это хорошее решение, но обычно оно ухудшает сопровождаемость и расширяемость.

ак вы получите события для двух разных кнопок, когда аждая кнопка делает что-то свое?

---

### ③ Способ третий. Создаем два отдельных класса ActionListener.

```
class MyGui  {
   JFrame frame;
   JLabel label;
   void gui() {
      // Код для создания экземпляров двух слушателей и связывание одного
      // из них с кнопкой для цвета, а другого — с кнопкой для метки
   }
} // Закрываем класс
```

---

```
class ColorButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        frame.repaint();
    }
}
```

↖ Не сработает! Этот класс не содержит ссылку на переменную frame класса MyGui.

---

```
class LabelButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        label.setText("Больно!");
    }
}
```

↖ Проблема! Этот класс не содержит ссылку на переменную label.

**Недостаток: эти классы не будут иметь доступ к переменным, с которыми им нужно взаимодействовать — frame и label.** Вы можете это исправить, но придется предоставлять каждому классу слушателя ссылку на основной класс GUI. В этом случае внутри методов actionPerformed() слушатель сможет использовать ссылку класса GUI для доступа к его переменным. Но такие действия нарушают инкапсуляцию, поэтому нам, возможно, понадобится создать для виджетов специальные методы — геттеры (getFrame(), getLabel() и т. д.). Кроме того, придется добавить в класс слушателя конструктор для передачи ссылки на объект GUI слушателю во время создания его экземпляра. Все становится еще запутаннее и сложнее.

***Должен быть более удобный способ!***

---

Разве не замечательно было бы иметь два разных класса слушателя, но чтобы они при этом получали доступ к полям главного класса GUI так, будто классы слушателя *принадлежат* к другому классу. Тогда вы имели бы оба преимущества. Да, было бы здорово. Но это лишь мечты...

# Внутренний класс спешит на помощь!

Вы можете иметь один класс, вложенный в другой. Это легко. Просто убедитесь, что определение для внутреннего класса находится *внутри* фигурных скобок внешнего класса.

**Простой внутренний класс:**

```
class MyOuterClass  {

   class MyInnerClass {
      void go() {
      }
   }

}
```

Внутренний класс полностью заключен во внешний класс.

Внутренний класс получает специальное разрешение на применение элементов внешнего класса, *даже если они приватные*. Он может использовать эти переменные и методы внешнего класса, как будто они определяются во внутреннем классе. Это основное достоинство внутренних классов — они имеют большинство преимуществ обычного класса, но со специальными правами доступа.

*Внутренний класс может использовать все методы и переменные внешнего класса, даже если те являются приватными.*

*Внутренний класс использует эти переменные и методы, как будто они объявлены во внутреннем классе.*

**Внутренний класс использует переменную внешнего класса:**

```
class MyOuterClass  {

   private int x;

   class MyInnerClass {
      void go() {
         x = 42;
      }
   } // Закрываем внутренний класс

} // Закрываем внешний класс
```

Используем x, как будто это переменная внутреннего класса!

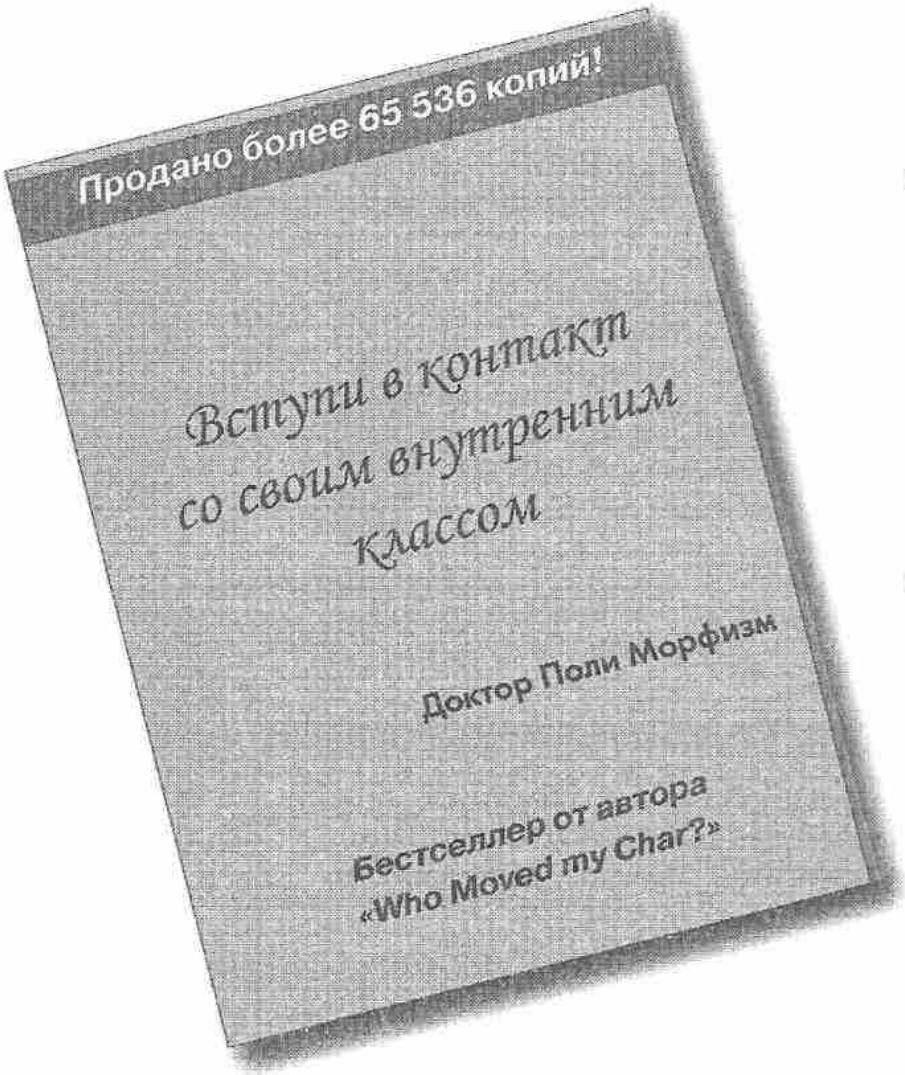
# Экземпляр внутреннего класса должен быть привязан к экземпляру внешнего класса*

**Внутренний объект испытывает особую привязанность к внешнему объекту.**

Вспомните, когда мы говорили о внутреннем классе, получающем доступ к чему-нибудь во внешнем классе, мы на самом деле имели в виду *экземпляр* внутреннего класса, получающий доступ к чему-нибудь, что находится в *экземпляре* внешнего класса. Но *какой именно* экземпляр?

Может ли *какой-нибудь* произвольный экземпляр внутреннего класса иметь доступ к методам и переменным *любого* экземпляра внешнего класса? **Нет!**

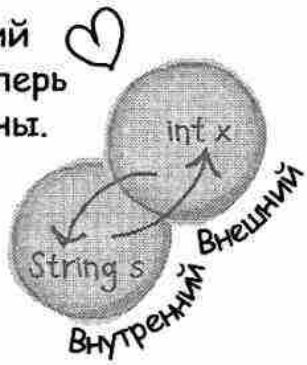
***Внутренний** объект должен быть привязан к определенному **внешнему** объекту в куче.*




1. Создаем экземпляр внешнего класса.

2. Создаем экземпляр внутреннего класса, используя экземпляр внешнего класса.

3. Внешний и внутренний объекты теперь тесно связаны.




Эти два объекта в куче обладают особой связью. Внутренний объект может использовать переменные внешнего (и наоборот).

* Существует исключение для тех особых случаев, когда внутренний класс определяется внутри статического метода. Но мы не будем касаться этого — вы можете всю жизнь проработать с Java и даже не повстречать подобного.

# Как создать экземпляр внутреннего класса

Если вы создаете экземпляр внутреннего класса с помощью кода *внутри* внешнего класса, то этот экземпляр будет «связан» с внутренним объектом. Например, если код внутри метода создает экземпляр внутреннего класса, то внутренний объект будет связан с экземпляром того класса, чей метод запущен.

Код внешнего класса может создать экземпляр одного из своих внутренних классов точно таким же образом, как он создает экземпляр любого другого класса: **new MyInner()**

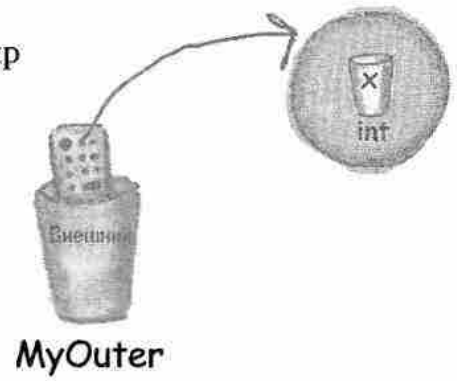
```
class MyOuter  {

   private int x;

   MyInner inner = new MyInner();

   public void doStuff() {
      inner.go();
   }

   class MyInner {
      void go() {
         x = 42;
      }
   } // Закрываем внутренний класс

} // Закрываем внешний класс
```

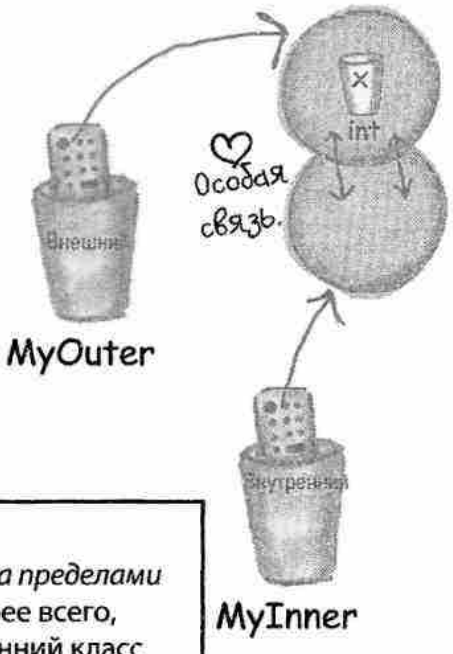
Внешний класс содержит приватную переменную экземпляра x.

Создаем экземпляр внутреннего класса.

Вызываем метод внутреннего класса.

Метод внутреннего класса использует переменную экземпляра x внешнего класса, как будто она принадлежит внутреннему классу.



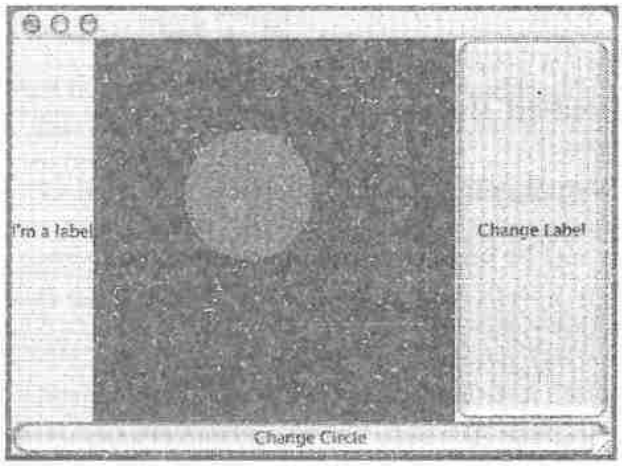




## Дополнительно

Вы *можете* создать экземпляр внутреннего класса из кода, запущенного *за пределами* внешнего класса, но придется использовать специальный синтаксис. Скорее всего, проработав всю свою жизнь на Java, вы так никогда и не создадите внутренний класс извне, но если вам интересно...

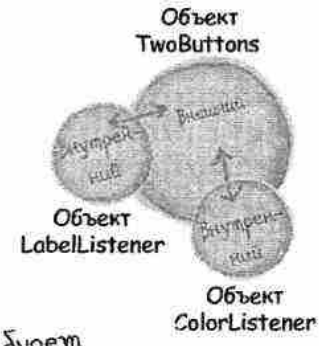
```
class Foo {
   public static void main (String[] args) {
      MyOuter outerObj = new MyOuter();
      MyOuter.MyInner innerObj = outerObj.new MyInner();
   }
}
```

## Теперь мы можем получить рабочий код с двумя кнопками




```
public class TwoButtons {                    // В данный момент главный класс GUI не реализует интерфейс ActionListener.

    JFrame frame;
    JLabel label;

    public static void main (String[] args) {
        TwoButtons gui = new TwoButtons ();
        gui.go();
    }

    public void go() {
        frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton labelButton = new JButton("Change Label");
        labelButton.addActionListener(new LabelListener());

        JButton colorButton = new JButton("Change Circle");
        colorButton.addActionListener(new ColorListener());

        label = new JLabel("I'm a label");
        MyDrawPanel drawPanel = new MyDrawPanel();

        frame.getContentPane().add(BorderLayout.SOUTH, colorButton);
        frame.getContentPane().add(BorderLayout.CENTER, drawPanel);
        frame.getContentPane().add(BorderLayout.EAST, labelButton);
        frame.getContentPane().add(BorderLayout.WEST, label);

        frame.setSize(300,300);
        frame.setVisible(true);
    }

    class LabelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            label.setText("Ouch!");
        }
    } // Закрываем внутренний класс

    class ColorListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            frame.repaint();
        }
    } // Закрываем внешний класс

}
```

Вместо того чтобы передавать (this) методу регистрации слушателя кнопки, мы передаем ему экземпляр соответствующего класса слушателя.




Теперь у нас будет два ActionListener в одном классе!

Внутренний класс знает о label.

Внутренний класс использует переменную экземпляра frame без явной ссылки на объект внешнего класса.

# РАЗОБЛАЧЕНИЕ ЯЗЫКА JAVA

## Интервью этой недели: Экземпляр внутреннего класса.

**HeadFirst:** Что делает внутренние классы такими важными?

**Внутренний объект:** Мы даем вам шанс несколько раз реализовать один и тот же интерфейс в классе. Помните, вы не можете повторно реализовать метод в обычном классе в Java. Но если используются *внутренние* классы, каждый из них может реализовать *один и тот же* интерфейс, что позволяет иметь *различные* реализации для одних и тех же методов интерфейса.

**HeadFirst:** Для чего вообще нужно реализовывать один и тот же метод дважды?

**Внутренний объект:** Вернемся к обработчикам событий пользовательского интерфейса. Если вы хотите, чтобы три кнопки имели различную реакцию на события, то используйте для этого три внутренних класса, которые будут реализовывать ActionListener. Это означает, что каждый класс будет реализовывать свой собственный метод actionPerformed.

**HeadFirst:** Таким образом, применение обработчиков событий — единственная причина для использования внутренних классов?

**Внутренний объект:** Конечно, нет. Обработчики событий — это всего лишь яркий пример. Если вам нужен отдельный класс, но при этом вы хотите, чтобы он вел себя так, будто является частью *другого* класса, то лучшим, а иногда и *единственным* способом будет применение внутреннего класса.

**HeadFirst:** Я все еще не могу понять. Если вы хотите, чтобы внутренний класс *вел себя* так, как будто он принадлежит внешнему классу, для чего вообще нужно иметь отдельный класс? Почему бы изначально не поместить код внутреннего класса во внешний класс?

**Внутренний объект:** Я только что *описал* вам один вариант развития событий, где нужно было иметь несколько реализаций интерфейса. Но даже если вы не будете использовать интерфейсы, вам могут понадобиться два разных *класса*, потому что они будут представлять два разных *«предмета»*. Это хороший пример объектно ориентированного программирования.

**HeadFirst:** Подождите. Я думал, что основная часть объектно ориентированного программирования заключается в повторном использовании и сопровождении. Иначе говоря, идея в том, что если есть два отдельных класса, то каждый из них можно модифицировать и использовать независимо, вместо того чтобы добавлять все в один класс. Но с *внутренним* классом получается, что вы работаете все еще с одним *фактическим* классом, верно? Класс, включающий его в себя, будет единственным повторно используемым и независящим от остального. Нельзя сказать, что внутренний класс тоже повторно используемый. Между прочим, я слышал, их называют повторно бесполезными.

**Внутренний объект:** Да, это так, внутренний класс не является таким же повторно используемым, как обычный класс. По сути, его иногда совсем невозможно вновь задействовать, потому что он тесно связан со свойствами и методами внешнего класса. Но он...

**HeadFirst:** ...это только доказывает мою мысль! Если нельзя повторно использовать, зачем вообще связываться с отдельным классом? Я имею в виду другую проблему, не относящуюся к интерфейсам, что выглядит для меня как обходной прием.

**Внутренний объект:** Как я говорил, вам нужно подумать о категоризации и полиморфизме.

**HeadFirst:** Хорошо. И я думаю о них, потому что...

**Внутренний объект:** Потому что внешним и внутренним классам, возможно, придется проходить *разные* проверки на соответствие. Начнем с примера полиморфического слушателя GUI. Какой тип аргумента объявляется для метода, отвечающего за регистрацию слушателя кнопки? Другими словами, если вы посмотрите и сверитесь с API, то *что* (тип класса или интерфейса) нужно будет передать методу addActionListener()?

**HeadFirst:** Вы должны передать слушателя. Что-нибудь, что реализует определенный интерфейс слушателя, в данном случае ActionListener. Да, мы все это знаем. К чему вы клоните?

**Внутренний объект:** Я имею в виду, что с точки зрения полиморфизма у вас есть метод, который принимает только один определенный *тип*. Что-то, что проходит проверку на соответствие интерфейсу ActionListener. Здесь начинается самое важное: что, если вашему классу нужно быть экземпляром чего-то, чьим типом является *класс*, а не интерфейс?

**HeadFirst:** А нельзя ли для этого просто наследовать класс, частью которого вы хотите быть? Разве не в этом суть дочерних классов? Если B — подкласс A, то везде, где ожидается A, можно использовать B. Вот такая вещь: передай собаку туда, где объявлен тип Животное.

**Внутренний объект:** Да! Точно! А теперь скажите, что будет, если вам придется пройти проверку на соответствие двум разным классам? Классам, что не находятся в одной иерархии наследования?

**HeadFirst**: Что ж... Думаю, я начинаю понимать. Вы всегда можете *реализовать* несколько интерфейсов, но расширить сможете только один класс. Вы можете быть экземпляром только одного вида, если речь идет о *классах*.

**Внутренний объект**: Правильно! Да, вы не можете одновременно быть и Собакой, и Кнопкой. Но если вы — Собака и иногда вам нужно быть Кнопкой (чтобы передать себя методам, принимающим Кнопку), то класс Собака (который наследует класс Животное и поэтому не может быть наследником класса Кнопка) сможет иметь *внутренний* класс, действующий от имени Собаки как Кнопка, тем самым наследуя класс Кнопка. И таким образом везде, где требуется Кнопка, Собака сможет передать свою внутреннюю Кнопку вместо себя. Другими словами, вместо того чтобы вызывать x.takeButton(this), объект Собаки вызывает x.takeButton(new MyInnerButton()).

**HeadFirst**: Могу ли я получить более понятный пример?

**Внутренний объект**: Помните панель рисования, которую мы использовали, когда создавали собственный подкласс JPanel? На данный момент он является отдельным, не внутренним классом. Так и должно быть, потому что он не нуждается в специальном доступе к полям главного GUI. Но что, если бы они были ему нужны? Допустим, мы создаем на такой панели анимацию, и класс получает ее координаты из главного приложения (например, основываясь на действиях пользователя в каком-нибудь другом месте GUI). В этом случае, если мы сделаем панель для рисования внутренним классом, она станет подклассом JPanel, в то время как внешний класс все еще может наследовать что-то другое.

**HeadFirst**: Да, теперь я вижу! В любом случае панель для рисования недостаточно подходит для повторного использования, чтобы стать отдельным классом, ведь созданные на ней рисунки фактически предназначены лишь для данного графического приложения.

**Внутренний объект**: Да! Вы все поняли!

**HeadFirst**: Хорошо. Теперь мы можем перейти к вопросу *отношений* между вами и внешним экземпляром.

**Внутренний объект**: Что это с вами? Не хватает грязных сплетен в такой серьезной теме, как полиморфизм?

**HeadFirst**: Эй, вы понятия не имеете, сколько готова заплатить публика за небольшое количество старой доброй грязи. Значит, кто-то вас создает и становится постоянно связанным с внешним объектом, правильно?

**Внутренний объект**: Да, все верно. И да, некоторые сравнивают это с браком по расчету. Мы не выбираем объекты, с которыми нас связывают.

**HeadFirst**: Хорошо, буду говорить по аналогии с браком. Вы можете получить *развод* и вступить в повторный брак с *чем-то еще*?

**Внутренний объект**: Нет, это на всю жизнь.

**HeadFirst**: Чью жизнь? Вашу? Внешнего объекта? Обоих?

**Внутренний объект**: Мою. Я не могу быть привязан ни к какому другому внешнему объекту. Мой единственный выход — сборщик мусора.

**HeadFirst**: А что можно сказать по поводу внешнего объекта? Он может быть связан с какими-либо другими внутренними объектами?

**Внутренний объект**: Вот мы и приехали. Вот чего вы *действительно* хотели. Да, у моего так называемого напарника может быть столько внутренних объектов, сколько он захочет.

**HeadFirst**: Это что-то вроде «многосерийного брака»? Или все они могут быть в одно время?

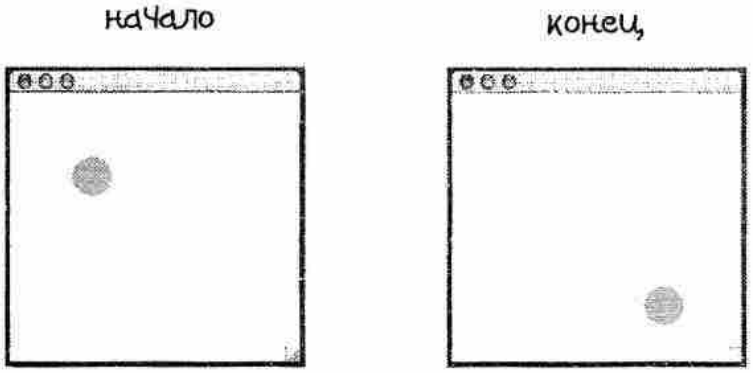
**Внутренний объект**: Все в одно время. Удовлетворены?

**HeadFirst**: Это действительно имеет смысл. И не будем забывать, что это *вы* расхваливали достоинства многократной реализации одного и того же интерфейса. Таким образом, понятно, что, если бы у внешнего класса было три кнопки, ему потребовалось бы три различных внутренних класса (и три различных объекта внутренних классов) для обработки событий. Спасибо за все. Вот это материал!

# Используем внутренний класс для анимации

Мы рассказали, почему внутренние классы так удобны для слушателей событий, так как вам придется реализовывать один и тот же метод обработки событий несколько раз. Но теперь мы рассмотрим, насколько полезен внутренний класс при использовании в качестве подкласса, не наследуемого внешним классом. Другими словами, рассмотрим случай, когда внешний и внутренний классы находятся в различных иерархиях наследования.

Наша цель состоит в том, чтобы сделать простую анимацию, где круг преодолевает экран от верхнего левого угла к нижнему правому.




## Как работает простая анимация

1. Рисуем объект по определенным координатам x и y:

```
g.fillOval(20,50,100,100);
```

20 пикселов от левого края,
50 пикселов от верхнего края.

2. Рисуем объект по другим координатам x и y:

```
g.fillOval(25,55,100,100);
```

25 пикселов от левого края,
55 пикселов от верхнего края (объект переместился немного вниз и вправо).

3. Повторяем предыдущий шаг с изменением значений x и y столько раз, сколько должна длиться анимация.

## Это не глупые вопросы

**В: Зачем мы здесь изучаем анимацию? Вряд ли я буду создавать игры.**

О: Может, вы и не станете создавать игры, но будете заниматься моделированием, где объекты со временем изменяются для отображения результатов процесса. Или будете создавать инструмент визуализации, который обновляет графику и показывает, сколько памяти использует программа или сколько трафика проходит через ваш распределяющий нагрузку сервер. Иными словами, все, что можно перевести из набора постоянно изменяющихся чисел во что-то более практичное для восприятия из них информации.

Разве это не деловой стиль? Конечно, это всего лишь формальное оправдание. Настоящая причина, по которой мы рассматриваем анимацию, состоит в том, что она предоставляет простой способ продемонстрировать другое применение внутренних классов.

## То, чего мы действительно добиваемся, представляет собой что-то вроде...

```
class MyDrawPanel extends JPanel {
      public void paintComponent(Graphics g) {
         g.setColor(Color.orange);
         g.fillOval(x,y,100,100);
      }
}
```

Каждый раз, когда вызывается метод paintComponent(), овал рисуется в другом месте.

Наточите свой карандаш

**Но где мы получим новые координаты x и y?**

**И кто вызывает метод repaint()?**

Посмотрим, сможете ли вы **придумать простое решение** и создать анимацию, в которой шар из верхнего левого угла панели для рисования перемещается в нижний правый угол. Ответ вы найдете на следующей странице, но не переворачивайте эту, пока не закончите!

Очень большая подсказка: сделайте панель для рисования внутренним классом.

Другая подсказка: не помещайте никакие повторяющиеся циклы в метод paintComponent().

**Здесь напишите свои идеи (или код):**

## Полный код для вывода простой анимации

```
import javax.swing.*;
import java.awt.*;


public class SimpleAnimation {

    int x = 70;
    int y = 70;

    public static void main (String[] args) {
       SimpleAnimation gui = new SimpleAnimation ();
       gui.go();
    }

    public void go() {
       JFrame frame = new JFrame();
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       MyDrawPanel drawPanel = new MyDrawPanel();

       frame.getContentPane().add(drawPanel);
       frame.setSize(300,300);
       frame.setVisible(true);

       for (int i = 0; i < 130; i++) {

          x++;
          y++;

          drawPanel.repaint();

          try {
            Thread.sleep(50);
          } catch(Exception ex) { }
       }

    }// Закрываем метод go()

    class MyDrawPanel extends JPanel {

       public void paintComponent(Graphics g) {
          g.setColor(Color.green);
          g.fillOval(x,y,40,40);


       }
    } // Закрываем внутренний класс
} // Закрываем внешний класс
```

Создаем два поля в главном классе GUI для координат x и y круга.

Здесь ничего нового. Создаем виджеты и помещаем их во фрейм.

Вот где происходит все действие!

Повторяем это 130 раз.

Увеличиваем координаты x и y.

Говорим панели, чтобы она себя перерисовала (и мы смогли увидеть круг на новом месте).

Немного замедляем процесс (иначе он будет выполняться так быстро, что вы не увидите никакого движения). Не волнуйтесь, вы еще не должны были знать этого. Мы дойдем до потоков в главе 15.

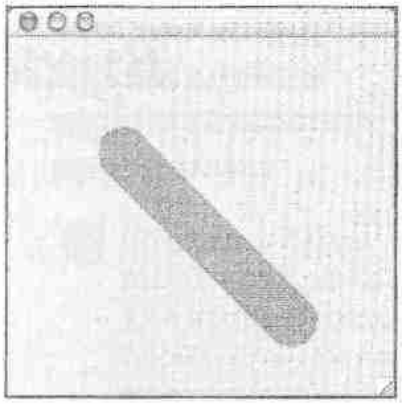
Теперь он стал внутренним классом.

Используем постоянно обновляющиеся координаты x и y внешнего класса.

## Ой. Он не двигается... а *размазывается*.

Что мы сделали не так?

В методе paintComponent() есть один маленький недостаток.



Выглядит не совсем так, как мы хотели.

### Мы забыли стереть то, что уже появилось, поэтому получаем такой след.

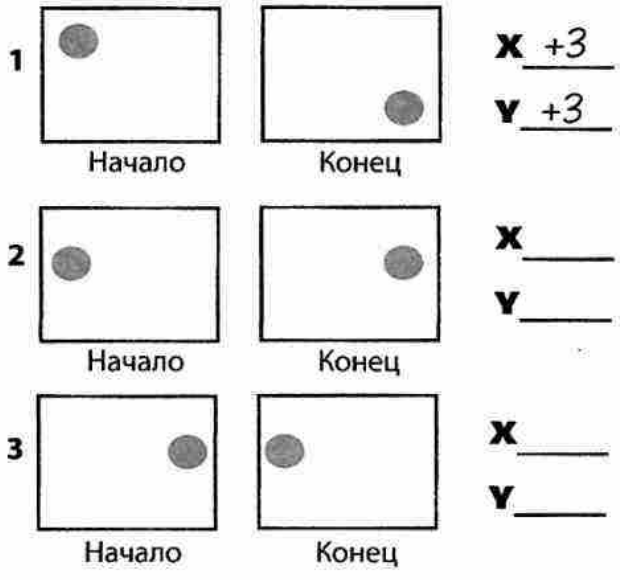
Чтобы исправить это, мы должны перед каждой перерисовкой круга всего лишь закрасить всю панель фоновым цветом. Приведенный далее код добавляет две строки в начало метода: одна из них устанавливает белый цвет (фоновый цвет панели для рисования), а другая заполняет весь прямоугольник панели этим цветом. Иными словами, код говорит: «Закрась прямоугольник, начиная с координат x и y, равных 0 (0 пикселов от левого края и 0 пикселов от верхнего края), и сделай его таким же широким и высоким, как эта панель».

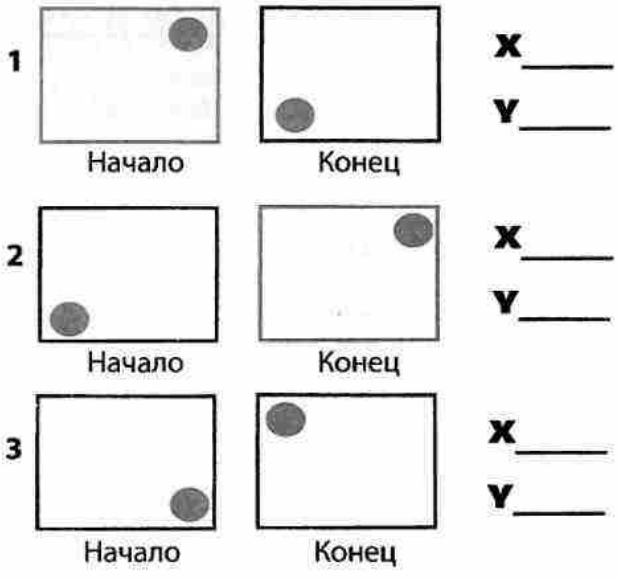
```
public void paintComponent(Graphics g) {
   g.setColor(Color.white);
   g.fillRect(0,0,this.getWidth(), this.getHeight());

   g.setColor(Color.green);
   g.fillOval(x,y,40,40);

}
```

Методы getWidth() и getHeight() унаследованы от JPanel.

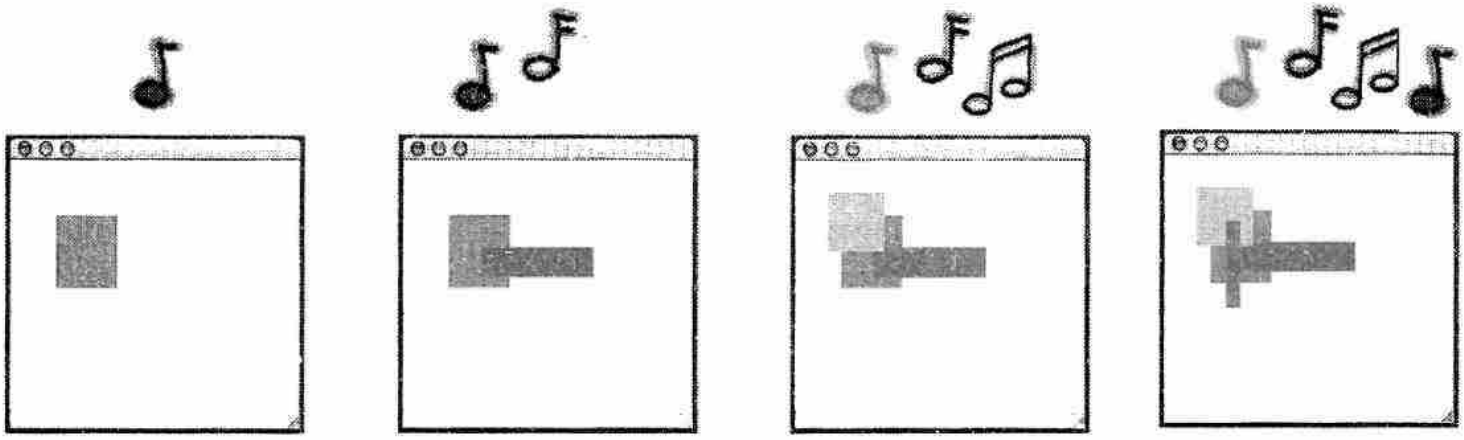
### Наточите свой карандаш (необязательно, просто ради интереса)

Какие изменения нужно внести в координаты x и y, чтобы получить показанную ниже анимацию? Допустим, что в первом примере при перемещении круга его координаты увеличились на 3 пиксела.



# Кухня кода




***Создадим музыкальный клип. Мы будем использовать графику, произвольно генерируемую с помощью языка Java. Она будет рисоваться под ритм музыки.***

***Вместе с этим мы будем регистрировать (и слушать) новые, не связанные с GUI события, запускаемые самой музыкой.***

Помните, что эта часть необязательна и выполняется по желанию. Но мы думаем, что она пойдет на пользу и понравится вам. В дальнейшем вы сможете использовать ее, чтобы произвести на кого-либо впечатление.

Конечно, это подействует только на очень впечатлительных людей, но все же...

# Отслеживаем событие, не связанное с GUI

Хорошо, может, мы сделаем не совсем музыкальный клип, а программу, которая будет рисовать на экране произвольную графику в такт музыке. Иными словами, программа будет слушать ритм музыки и рисовать произвольные графические прямоугольники на каждый такт.

Это создает новые проблемы. Пока мы отслеживали только графические события, но теперь нам нужно следить за MIDI-событиями определенного типа. Оказывается, что отслеживание событий, не связанных с GUI, происходит точно так же, как и в случае событий GUI: вы реализуете интерфейс слушателя, связываете слушателя с источником события, а затем ждете, когда тот вызовет ваш метод обработки событий (метод, определенный в интерфейсе слушателя).

Простейший способ следить за музыкальным ритмом — регистрация и прослушивание действующих MIDI-событий, поэтому, когда синтезатор будет получать событие, наш код тоже его получит и сможет нарисовать графику. Но здесь есть проблема. Ошибка, которая фактически не позволит нам отслеживать создаваемые MIDI-события (связанные с включением воспроизведения назначенных нот).

Подойдем к проблеме с другой стороны. Существует иной тип MIDI-события, который мы сможем отслеживать. Он называется ControllerEvent. Решение заключается в том, что мы будем регистрировать ControllerEvent, а затем проверять, что для каждого события, связанного с включением воспроизведения назначенных нот, имеется подходящее событие ControllerEvent, запущенное в тот же самый «такт». Как мы проверим, что ControllerEvent запущено в то же самое время? Мы добавим его в дорожку так же, как и остальные события! Другими словами, наша музыкальная последовательность будет выглядеть таким образом:

ТАКТ 1 — включение воспроизведения назначенных нот, ControllerEvent;

ТАКТ 2 — отключение воспроизведения назначенных нот;

ТАКТ 3 — включение воспроизведения назначенных нот, ControllerEvent;

ТАКТ 4 — отключение воспроизведения назначенных нот и т. д.

Однако перед тем как перейти непосредственно к программе, сделаем процесс добавления и создания MIDI-сообщений/событий немного легче, так как в *этой* программе у нас их будет предостаточно.

## Что должна делать музыкальная художественная программа

1. Создавать серии MIDI-сообщений/событий для воспроизведения произвольных нот на фортепиано (или на *любом* другом инструменте, который вы выберете).
2. Регистрировать слушателя для событий.
3. Запускать синтезатор.
4. При каждом вызове метода обработки событий слушателя отображать произвольный прямоугольник на панели для рисования и вызывать перерисовку.

## Мы создадим все это в три шага

1. Версия первая: пишем код, упрощающий создание и добавление MIDI-событий, так как их у нас будет много.
2. Версия вторая: регистрируем и слушаем события, но без графики. Выводим сообщение в командную строку с каждым тактом.
4. Версия третья: настоящее действие. Добавляем графику во вторую версию.

# Легкий способ создания сообщений/событий

На данный момент создание и добавление сообщений и событий в дорожку — утомительное занятие. Для каждого сообщения мы должны создавать отдельный экземпляр (в данном случае ShortMessage), вызывать setMessage(), создавать MidiEvent для сообщения и добавлять событие в дорожку. В предыдущей главе мы проходили каждый шаг для отдельного сообщения. Это восемь строк кода только для того, чтобы проиграть, а затем остановить ноту! Четыре строки на добавление события включения воспроизведения ноты и столько же на добавление события отключения воспроизведения ноты.

```
ShortMessage a = new ShortMessage();
a.setMessage(144, 1, note, 100);
MidiEvent noteOn = new MidiEvent(a, 1);
track.add(noteOn);

ShortMessage b = new ShortMessage();
b.setMessage(128, 1, note, 100);
MidiEvent noteOff = new MidiEvent(b, 16);
track.add(noteOff);
```

## Рассмотрим, что происходит при каждом событии.

1. Создаем экземпляр сообщения:

```
ShortMessage first = new ShortMessage();
```

2. Вызываем метод setMessage() с аргументами:

```
first.setMessage(192, 1, instrument, 0)
```

3. Создаем экземпляр MidiEvent для сообщения:

```
MidiEvent noteOn = new MidiEvent(first, 1)
```

4. Добавляем в дорожку событие:

```
track.add(noteOn);
```

## Напишем вспомогательный статический метод, который будет создавать сообщения и возвращать MidiEvent.

Четыре параметра для сообщения.

Событие tick происходит в моменте появления данного сообщения.

```
public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
        ShortMessage a = new ShortMessage();
        a.setMessage(comd, chan, one, two);
        event = new MidiEvent(a, tick);

    }catch(Exception e) { }
    return event;
}
```

Ого! Метод с пятью параметрами.

Создаем сообщение и событие, используя параметры метода.

Возвращаем событие (MidiEvent полностью загружено сообщением).

# Пример: используем новый статический метод makeEvent()

Здесь нет обработки события или графики, а есть лишь последовательность из 15 восходящих нот. Цель данного кода — изучение работы нашего нового метода makeEvent(). Код для следующих двух версий будет намного меньше и проще благодаря этому методу.

```
import javax.sound.midi.*;

public class MiniMusicPlayer1 {

   public static void main(String[] args) {

      try {

         Sequencer sequencer = MidiSystem.getSequencer();
         sequencer.open();

         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track track = seq.createTrack();

      for (int i = 5; i < 61; i+= 4) {

         track.add(makeEvent(144,1,i,100,i));
         track.add(makeEvent(128,1,i,100,i + 2));

       } // Конец цикла

         sequencer.setSequence(seq);
         sequencer.setTempoInBPM(220);
         sequencer.start();
      } catch (Exception ex) {ex.printStackTrace();}
  } // Закрываем main

   public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
          MidiEvent event = null;
          try {
            ShortMessage a = new ShortMessage();
            a.setMessage(comd, chan, one, two);
           event = new MidiEvent(a, tick);

          }catch(Exception e) { }
         return event;
       }
} // Закрываем класс
```

Не забываем про импортирование.

Создаем (и открываем) синтезатор.

Создаем последовательность и дорожку.

Создаем группу событий, чтобы ноты продолжали подниматься (от ноты фортепиано 5 до ноты 61).

Вызываем новый метод makeEvent(), чтобы создать сообщение и событие, а затем добавляем результат (MidiEvent, полученное из makeEvent()) в дорожку. Они представляют собой пару включения (144) и отключения воспроизведения ноты (128).

Запускаем его.

# Версия вторая: регистрируем и получаем ControllerEvent

Нам нужно отслеживать события ControllerEvent, поэтому мы реализуем интерфейс слушателя.

```
import javax.sound.midi.*;
public class MiniMusicPlayer2 implements ControllerEventListener {

    public static void main(String[] args) {
        MiniMusicPlayer2 mini = new MiniMusicPlayer2();
        mini.go();
    }
  public void go() {

     try {
        Sequencer sequencer = MidiSystem.getSequencer();
        sequencer.open();

        int[] eventsIWant = {127};
        sequencer.addControllerEventListener(this, eventsIWant);

        Sequence seq = new Sequence(Sequence.PPQ, 4);
        Track track = seq.createTrack();

        for (int i = 5; i < 60; i+= 4) {
           track.add(makeEvent(144,1,i,100,i));

           track.add(makeEvent(176,1,127,0,i));

           track.add(makeEvent(128,1,i,100,i + 2));
        } // Конец цикла

        sequencer.setSequence(seq);
        sequencer.setTempoInBPM(220);
        sequencer.start();
     } catch (Exception ex) {ex.printStackTrace();}
  } // Закрываем

  public void controlChange(ShortMessage event) {
        System.out.println("ля");
  }

  public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
        MidiEvent event = null;
        try {
          ShortMessage a = new ShortMessage();
          a.setMessage(comd, chan, one, two);
          event = new MidiEvent(a, tick);

        }catch(Exception e) { }
        return event;
      }
} // Закрываем класс
```

Регистрируем события синтезатором. Метод, отвечающий за регистрацию, принимает объект слушателя и целочисленный массив, представляющий собой список событий ControllerEvent, которые нам нужны. Нас интересует только одно событие — #127.

Вот так мы ловим ритм — добавляем наше собственное событие ControllerEvent (176 означает, что тип события — ControllerEvent) с аргументом для события номер #127. Оно ничего не будет делать! Мы вставляем его лишь для того, чтобы иметь возможность реагировать на воспроизведение каждой ноты. Другими словами, его единственная цель — запуск чего-нибудь, что можно отслеживать (мы не можем следить за событиями включения/выключения воспроизведения нот). Заметьте, что мы запускаем это событие в тот же самый момент, когда включается воспроизведение ноты. Поэтому когда произойдет событие включения воспроизведения ноты, мы сразу узнаем об этом, так как наше событие запустится в то же самое время.

Метод обработки события (из интерфейса слушателя события ControllerEvent). При каждом получении события мы пишем в командной строке слово «ля».

* Код, который отличается от предыдущей версии, помещен на серую плашку (и в этот раз мы не запускаем весь код внутри main()).

# Версия третья: рисуем графику в такт музыке

Последняя версия создается на основе второй с добавлением графической части. Мы создаем фрейм, добавляем в него панель для рисования и при каждом получении события рисуем новый прямоугольник, перерисовывая экран. Еще одно отличие от второй версии состоит в том, что ноты проигрываются в случайном порядке, а не просто вверх по гамме.

Наиболее важное изменение в коде (не считая создания простого GUI) заключается в том, что мы заставляем панель для рисования реализовывать интерфейс ControllerEventListener, вместо того чтобы доверять это самой программе. Поэтому, когда панель для рисования (внутренний класс) получает событие, она знает, как поступить, и рисует прямоугольник.

Полный код для этой версии находится на следующей странице.

## Внутренний класс панели рисования:

Панель для рисования — это слушатель.

```
class MyDrawPanel extends JPanel implements ControllerEventListener {

    boolean msg = false;

    public void controlChange(ShortMessage event) {
       msg = true;
       repaint();
    }

    public void paintComponent(Graphics g) {
     if (msg) {

       Graphics2D g2 = (Graphics2D) g;

       int r = (int) (Math.random() * 250);
       int gr = (int) (Math.random() * 250);
       int b = (int) (Math.random() * 250);

       g.setColor(new Color(r,gr,b));

       int ht = (int) ((Math.random() * 120) + 10);
       int width = (int) ((Math.random() * 120) + 10);
       int x = (int) ((Math.random() * 40) + 10);
       int y = (int) ((Math.random() * 40) + 10);
       g.fillRect(x,y,ht, width);
       msg = false;

     } // Закрываем if
    } // Закрываем метод
  }  // Закрываем внутренний класс
```

Присваиваем флагу значение false и будем устанавливать true, когда получим событие.

Мы получили событие, поэтому присваиваем флагу значение true и вызываем repaint().

Мы должны использовать флаг, потому что другие объекты могут запустить repaint(), а мы хотим рисовать только тогда, когда произойдет событие ControllerEvent.

Оставшийся код нужен для генерации случайного цвета и рисования полупроизвольного прямоугольника.

**Код** MiniMusicPlayer3

Здесь приведен полный код для третьей версии. Он создается непосредственно на основе второй версии. Попытайтесь самостоятельно добавить для него комментарии, не заглядывая на предыдущую страницу.

```
import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;

public class MiniMusicPlayer3 {

     static JFrame f = new JFrame("Мой первый музыкальный клип");
     static MyDrawPanel ml;

     public static void main(String[] args) {
           MiniMusicPlayer3 mini = new MiniMusicPlayer3();
           mini.go();
     } // Закрываем метод


     public  void setUpGui() {
        ml = new MyDrawPanel();
        f.setContentPane(ml);
        f.setBounds(30,30, 300,300);
        f.setVisible(true);
    } // Закрываем метод

    public void go() {
       setUpGui();

       try {

          Sequencer sequencer = MidiSystem.getSequencer();
          sequencer.open();
          sequencer.addControllerEventListener(ml, new int[] {127});
          Sequence seq = new Sequence(Sequence.PPQ, 4);
          Track track = seq.createTrack();

          int r = 0;
          for (int i = 0; i < 60; i+= 4) {

             r = (int) ((Math.random() * 50) + 1);
             track.add(makeEvent(144,1,r,100,i));
             track.add(makeEvent(176,1,127,0,i));
             track.add(makeEvent(128,1,r,100,i + 2));
          } // Конец цикла

          sequencer.setSequence(seq);
          sequencer.start();
          sequencer.setTempoInBPM(120);
       } catch (Exception ex) {ex.printStackTrace();}
    } // Закрываем метод
```

```
  public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
      MidiEvent event = null;
      try {
         ShortMessage a = new ShortMessage();
         a.setMessage(comd, chan, one, two);
          event = new MidiEvent(a, tick);

       }catch(Exception e) { }
       return event;
   } // Закрываем метод


class MyDrawPanel extends JPanel implements ControllerEventListener {
     boolean msg = false;

     public void controlChange(ShortMessage event) {
        msg = true;
        repaint();
     }

     public void paintComponent(Graphics g) {
      if (msg) {

        Graphics2D g2 = (Graphics2D) g;

        int r = (int) (Math.random() * 250);
        int gr = (int) (Math.random() * 250);
        int b = (int) (Math.random() * 250);

        g.setColor(new Color(r,gr,b));

        int ht = (int) ((Math.random() * 120) + 10);
        int width = (int) ((Math.random() * 120) + 10);

        int x = (int) ((Math.random() * 40) + 10);
        int y = (int) ((Math.random() * 40) + 10);

        g.fillRect(x,y,ht, width);
        msg = false;

      } // Закрываем if
    } // Закрываем метод
  }  // Закрываем внутренний класс

 // Закрываем класс
```

## Упражнение

# Кто я такой?

Команда звезд из мира Java хочет сыграть с вами в игру «Кто я такой?». Они дают вам подсказку, а вы пытаетесь угадать их имена. Считайте, что они всегда рассказывают о себе правду. Если они говорят что-нибудь подходящее сразу для нескольких участников, то записывайте всех, к кому применимо данное утверждение. Заполните пустые строки именами одного или нескольких участников рядом с утверждениями.

**Сегодня участвуют:**

**Здесь могут появиться любые харизматичные личности из этой главы!**

В моих руках весь GUI. _____

Каждый тип события обладает одним из них. _____

Ключевой метод слушателя. _____

Этот метод присваивает объекту JFrame его размер. _____

Вы добавляете код к этому методу, но никогда не вызываете его. _____

Когда пользователь действительно что-нибудь делает, это является. _____

Большинство из них — источники событий. _____

Я передаю данные назад к слушателю. _____

Метод addXxxListener() говорит, что объект является. _____

Так записывается слушатель. _____

Метод, где помещается весь код, отвечающий за графику. _____

Я обычно привязан к экземпляру. _____

g в (Graphics g) на самом деле принадлежит этому классу. _____

Метод, запускающий paintComponent(). _____

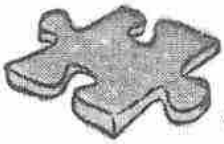
Пакет, где обитает большинство объектов Swing. _____

## Упражнение

# Поработайте компилятором

На этой странице представлен полноценный исходный код на языке Java. Ваша задача — сыграть роль компилятора и определить, компилируется ли этот код. Если он не компилируется, то как бы вы исправили его, а если компилируется, то что делает?

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class InnerButton {

  JFrame frame;
  JButton b;

  public static void main(String [] args) {
    InnerButton gui = new InnerButton();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                  JFrame.EXIT_ON_CLOSE);

    b = new JButton("A");
    b.addActionListener();

    frame.getContentPane().add(
                  BorderLayout.SOUTH, b);
    frame.setSize(200,100);
    frame.setVisible(true);
  }

  class BListener extends ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```



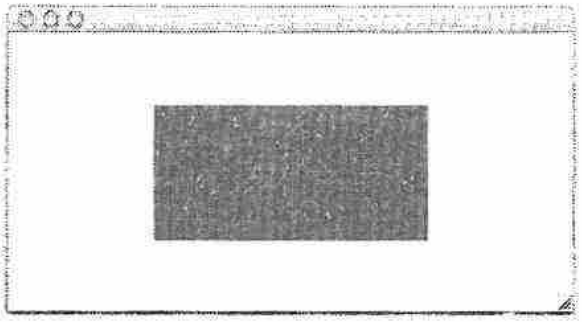
# Головоломка у бассейна

Ваша ***задача*** — взять фрагменты кода со дна бассейна и поместить их в пустые секции. Вы ***можете*** использовать один фрагмент несколько раз, и необязательно задействовать все фрагменты. Ваша ***цель*** — сделать класс, который скомпилируется, запустится и создаст приведенный ниже результат.

**Результат**

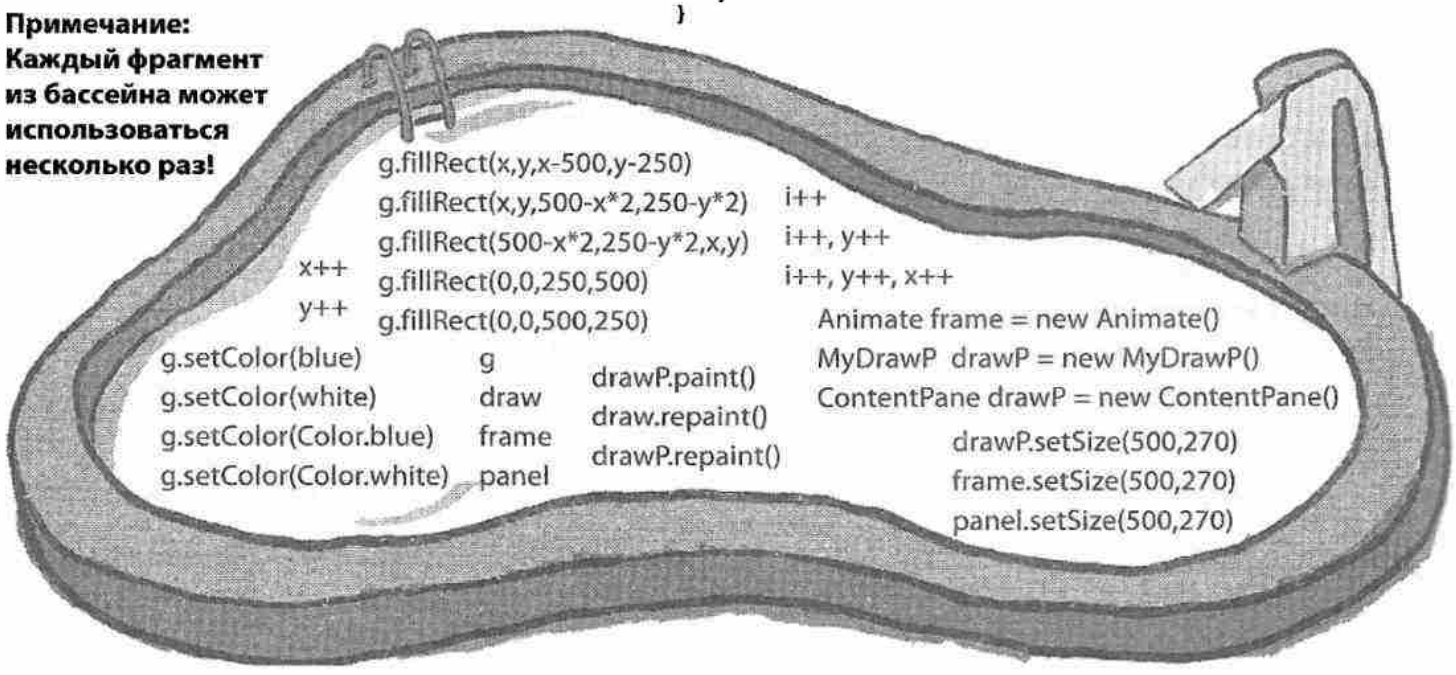
Удивительный сжимающийся синий прямоугольник.

Программа создаст синий прямоугольник, который будет сжиматься, пока не исчезнет на белом полотне.



```
import javax.swing.*;
import java.awt.*;
public class Animate {
    int x = 1;
    int y = 1;
    public static void main (String[] args) {
       Animate gui = new Animate ();
       gui.go();
    }
    public void go() {
       JFrame _________ = new JFrame();
       frame.setDefaultCloseOperation(
                        JFrame.EXIT_ON_CLOSE);
       ____________________________________;
       _________.getContentPane().add(drawP);
       ____________________________;
       _________.setVisible(true);
       for (int i=0; i<124; _______________) {
       ____________________;
       ____________________;
       try {
          Thread.sleep(50);
       } catch(Exception ex) { }
       }
    }
    class MyDrawP extends JPanel {
            public void paintComponent (Graphics
                                        _________) {
          ____________________________________;
          ____________________________________;
          ____________________________________;
          ____________________________________;
       }
    }
}
```

**Примечание: Каждый фрагмент из бассейна может использоваться несколько раз!**



# Ответы

## Кто я такой?

| | |
|---|---|
| В моих руках весь GUI. | JFrame. |
| Каждый тип события обладает одним из них. | Интерфейс слушателя. |
| Ключевой метод слушателя. | actionPerformed(). |
| Этот метод присваивает объекту JFrame его размер. | setSize(). |
| Вы добавляете код к этому методу, но никогда не вызываете его. | paintComponent(). |
| Когда пользователь действительно что-нибудь делает, это является _____ | событием. |
| Большинство из них — источники событий. | Компоненты Swing. |
| Я передаю данные назад к слушателю. | Объект события. |
| Метод addXxxListener() говорит, что объект является _____ | источником события. |
| Так записывается слушатель. | addActionListener(). |
| Метод, где помещается весь код, отвечающий за графику. | paintComponent(). |
| Я обычно привязан к экземпляру. | Внутренний класс. |
| g в (Graphics g) на самом деле принадлежит этому классу. | Graphics2d. |
| Метод, запускающий paintComponent(). | repaint( ). |
| Пакет, где обитает большинство компонентов Swing. | javax.swing. |

## Поработайте компилятором

Если исправить этот код, он создаст пользовательский интерфейс с кнопкой, которая при нажатии будет переключаться между буквами A и B.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class InnerButton {

  JFrame frame;
  JButton b;

  public static void main(String [] args) {
    InnerButton gui = new InnerButton();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);

    b = new JButton("A");
    b.addActionListener( new BListener() );

    frame.getContentPane().add(
                    BorderLayout.SOUTH, b);
    frame.setSize(200,100);
    frame.setVisible(true);
  }

  class BListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```

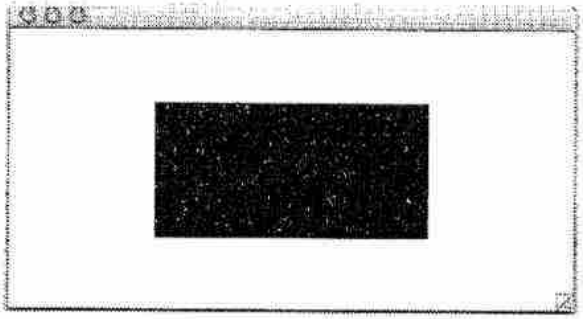
Метод addActionListener() принимает класс, который реализует интерфейс ActionListener.

ActionListener — это интерфейс. Интерфейсы реализуются, а не наследуются.

## Головоломка у бассейна

```
import javax.swing.*;
import java.awt.*;
public class Animate {
    int x = 1;
    int y = 1;
    public static void main (String[] args) {
       Animate gui = new Animate ();
       gui.go();
   }
   public void go() {
       JFrame frame = new JFrame();
       frame.setDefaultCloseOperation(
                        JFrame.EXIT_ON_CLOSE);
       MyDrawP drawP = new MyDrawP();
       frame.getContentPane().add(drawP);
       frame.setSize(500,270);
       frame.setVisible(true);
       for (int i = 0; i < 124; i++,x++,y++ ) {
          x++;
          drawP.repaint();
          try {
            Thread.sleep(50);
          } catch(Exception ex) { }
       }
   }
   class MyDrawP extends JPanel {
       public void paintComponent(Graphics g ) {
              g.setColor(Color.white);
             g.fillRect(0,0,500,250);
              g.setColor(Color.blue);
              g.fillRect(x,y,500-x*2,250-y*2);
       }
   }
}
```

Удивительный сжимающийся синий прямоугольник.



# Улучшай свои навыки

**Swing — это просто.** Если, конечно, вас *не волнует*, в каком месте в конце концов все окажется на экране. Код для работы со Swing *выглядит* просто, но, скомпилировав его, запустив и посмотрев на экран, вы подумаете: «Эй, *этот* объект должен быть *в другом месте*». Инструмент, который упрощает создание кода и одновременно усложняет управление им — это **диспетчер компоновки**. Его объекты отвечают за размер и положение виджетов в GUI для программ на языке Java. Они делают за вас большой объем работы, но результат не всегда удовлетворительный. Вы хотите, чтобы две кнопки были одинакового размера, но они оказываются разными? Вы хотите, чтобы поле ввода было длиной три дюйма, а оно получается девятидюймовым и находится *под* меткой, вместо того чтобы быть *рядом* с ней? Приложив небольшие усилия, вы можете подчинить себе диспетчер компоновки. В этой главе мы будем работать над Swing и ближе познакомимся с виджетами. Мы будем создавать их, выводить (где захотим) и использовать в программе. Сьюзи бы это не понравилось...

# Компоненты Swing

***Компонент*** — это правильное название для *виджета*. Объекты, которые вы вставляете в GUI и *которые пользователь видит и использует*: поля ввода, кнопки, прокручиваемые списки, переключатели и т. д., — все это компоненты. По сути, они все наследуют класс javax.swing.JComponent.

***Виджет — это компонент Swing. Почти любой объект, вставляемый в GUI, наследуется от javax.swing.JComponent.***

## Компоненты могут быть вложенными

В Swing практически *все* компоненты могут включать в себя другие компоненты. Иными словами, можно *вставить почти что угодно во что-то еще*. Но большую часть времени вы будете вставлять *интерактивные* компоненты, такие как кнопки и списки, в фоновые компоненты — фреймы и панели. Однако, хотя у вас есть возможность вложить, скажем, панель в кнопку, это будет выглядеть несуразно и не принесет вам пользы.

За исключением JFrame, разница между *интерактивными* и *фоновыми* компонентами довольно условна. JPanel, например, обычно используется как фон для группирования остальных компонентов, но даже он может быть интерактивным. Как и с другими компонентами, вы можете привязывать к JPanel события, в том числе щелчки кнопкой мыши или нажатие клавиш.

### Четыре шага для создания GUI (обзор)

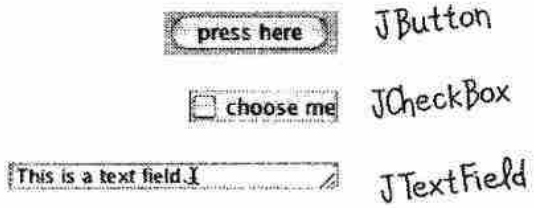
1. Создаем окно (JFrame):
```
JFrame frame = new JFrame();
```

2. Создаем компонент (кнопка, поле ввода и т. д.):
```
JButton button = new JButton("click me");
```

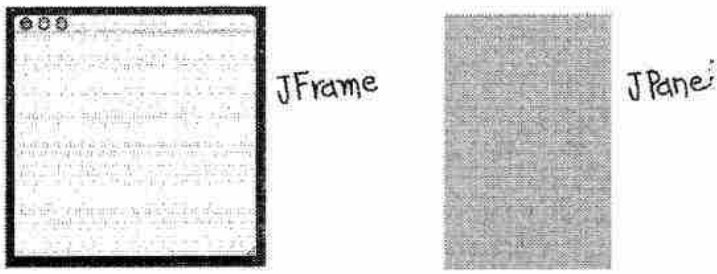
3. Добавляем компонент внутрь фрейма:
```
frame.getContentPane().add(BorderLayout.EAST, button);
```

4. Выводим его на экран (задаем ему размер и делаем видимым):
```
frame.setSize(300,300);
frame.setVisible(true);
```

### Вставляем интерактивные компоненты:

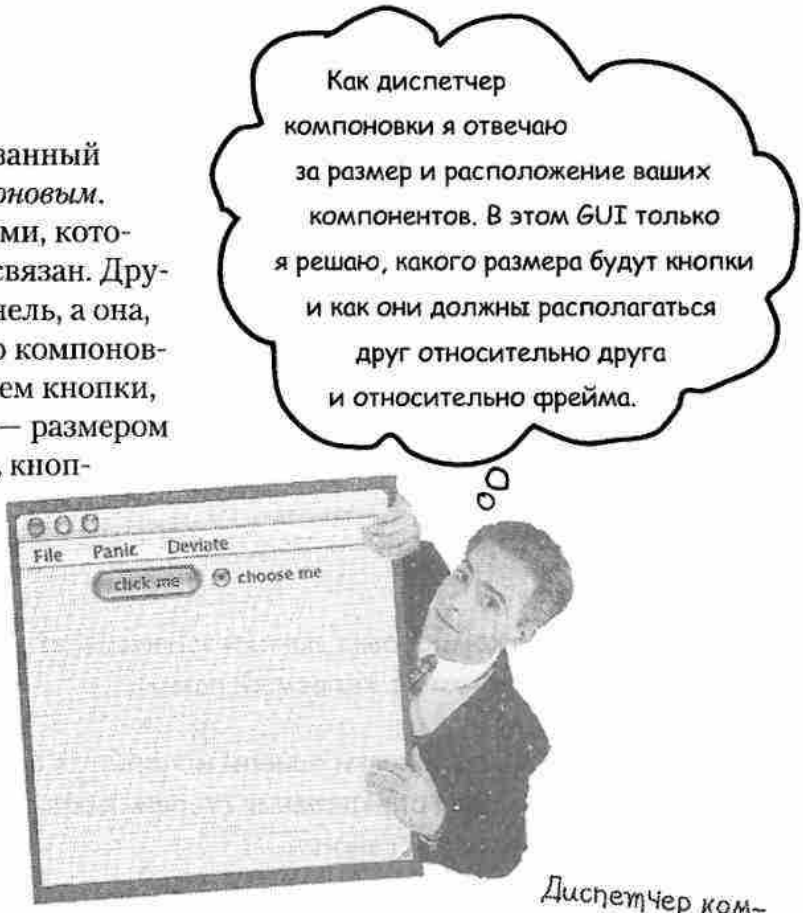



### В фоновые компоненты:



# Диспетчеры компоновки

Диспетчер компоновки — это Java-объект, связанный с определенным компонентом, почти всегда *фоновым*. Диспетчер компоновки управляет компонентами, которые содержатся *внутри* него и с которыми он связан. Другими словами, если фрейм включает в себя панель, а она, в свою очередь, содержит кнопку, то диспетчер компоновки панели управляет размером и расположением кнопки, в то время как диспетчер компоновки фрейма — размером и расположением панели. С другой стороны, кнопка не нуждается в диспетчере компоновки, потому что не содержит иных компонентов.




Допустим, панель включает в себя пять объектов. Даже если каждый из них обладает собственным диспетчером компоновки, размер и расположение пяти объектов на панели контролируются ее диспетчером компоновки. Если эти пять объектов, в свою очередь, включают *другие* объекты, то те располагаются в соответствии с диспетчером компоновки содержащего их объекта.

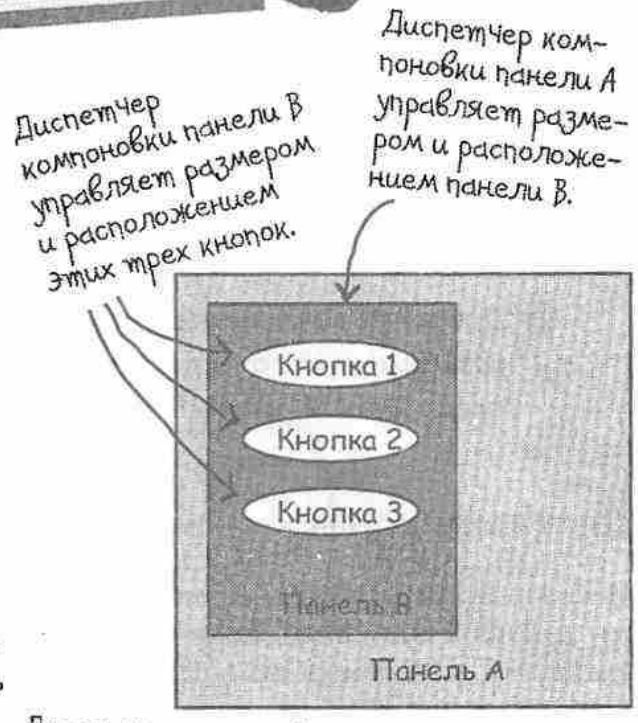
Когда мы говорим *«содержит»*, то в действительности имеем в виду *«добавлена»*. Панель *содержит* кнопку, потому что кнопка *добавлена* с помощью определенного кода, например:

```
myPanel.add(button);
```

Диспетчеры компоновки бывают нескольких типов, и любой фоновый компонент может иметь собственный диспетчер компоновки. Кроме того, они должны следовать определенным правилам при построении схем размещения.

Например, один диспетчер компоновки может требовать, чтобы все компоненты на панели были одинакового размера и упорядочены по сетке, в то время как другой предоставляет каждому компоненту возможность выбрать собственный размер, но располагает их вертикально. Здесь представлены примеры вложенных схем размещения:

```
JPanel panelA = new JPanel();
JPanel panelB = new JPanel();
panelB.add(new JButton("Кнопка 1"));
panelB.add(new JButton("Кнопка 2"));
panelB.add(new JButton("Кнопка 3"));
panelA.add(panelB);
```




Диспетчер компоновки панели A ничего не знает об этих трех кнопках. Иерархия контроля содержит только один уровень — диспетчер компоновки панели A управляет лишь теми объектами, которые добавлены непосредственно на панель A, и не управляет вложенными в них объектами.

# Как диспетчер компоновки принимает решения

Различные диспетчеры компоновки по-разному организуют компоненты (присваивают одинаковый размер, выстраивают по сетке или по вертикали и т. д.). Однако размещаемые компоненты также играют небольшую роль в данном вопросе. В основном, процесс размещения фонового компонента выглядит следующим образом.

**Сценарий компоновки**

1. Создаем панель и добавляем на нее три кнопки.
2. Диспетчер компоновки панели запрашивает у каждой кнопки ее желаемый размер.
3. Диспетчер компоновки панели использует свои правила размещения для определения степени соблюдения предпочтений этих кнопок.
4. Добавляем панель внутрь фрейма.
5. Диспетчер компоновки фрейма запрашивает у панели ее желаемый размер.
6. Диспетчер компоновки фрейма использует свои правила размещения для определения степени соблюдения предпочтений панели.

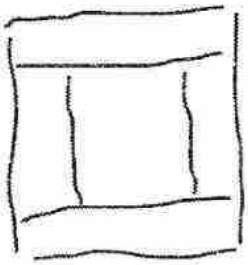
## Различные диспетчеры компоновки работают по разным правилам

Некоторые диспетчеры компоновки учитывают, какого размера хочет быть тот или иной компонент. Если кнопка желает иметь размер 30×50 пикселов, то диспетчер компоновки так и сделает. Другие диспетчеры компоновки учитывают только часть предпочтений компонента. Если кнопка желает иметь размер 30×50 пикселов, ее высота составит 30 пикселов, а ширина будет зависеть от размеров *панели*. Кроме того, другие диспетчеры компоновки учитывают предпочтения только *самых больших* компонентов, а все остальные устанавливаются одного размера. В некоторых случаях работа диспетчера компоновки может стать очень запутанной, но в основном, как только вы узнаете его правила, вы научитесь понимать его действия.

# Три главных диспетчера компоновки: border, flow и box

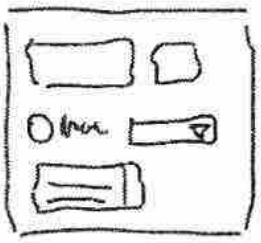
### BorderLayout

Диспетчер BorderLayout делит фоновый компонент на пять областей. В каждую область, управляемую BorderLayout, вы можете добавить только один компонент. Компоненты, размещенные этим диспетчером, обычно не имеют предпочтений по размерам. **BorderLayout — по умолчанию диспетчер компоновки для фрейма!**



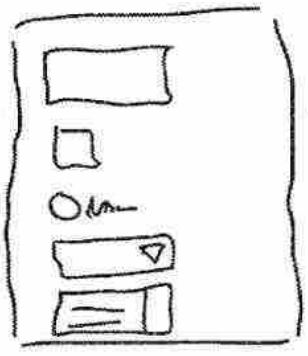

### FlowLayout

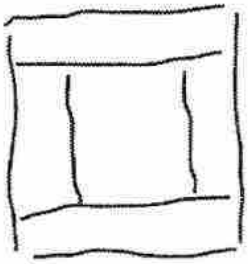
Диспетчер FlowLayout работает с компонентами наподобие текстового процессора. Каждый компонент имеет желаемый размер, и все они размещаются слева направо, в порядке добавления, с возможностью переноса на новую строку. Когда компонент не помещается по горизонтали, он переносится на следующую строку в компоновке. **FlowLayout — по умолчанию диспетчер компоновки для панели!**




### BoxLayout

Диспетчер BoxLayout похож на FlowLayout тем, что все его компоненты получают собственный размер и располагаются в порядке добавления. Однако, в отличие от FlowLayout, BoxLayout позволяет расположить их вертикально (или горизонтально, но обычно нас интересует вертикальное расположение). Все происходит, как во FlowLayout, но вместо автоматического переноса компонентов вы можете добавить что-то вроде разрыва строки и **заставить** компонент перенестись на новую строку.





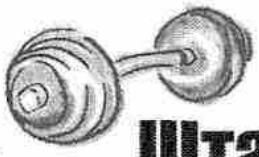
**BorderLayout управляет пятью областями: east, west, north, south и center.**

**Добавим кнопку в область east:**

```
import javax.swing.*;
import java.awt.*;

public class Button1 {

    public static void main (String[] args) {
        Button1 gui = new Button1();
        gui.go();
    }

    public void go() {
        JFrame frame = new JFrame();
        JButton button = new JButton("click me");
        frame.getContentPane().add(BorderLayout.EAST, button);
        frame.setSize(200,200);
        frame.setVisible(true);
    }
}
```

BorderLayout находится в пакете java.awt.

Задаем область.

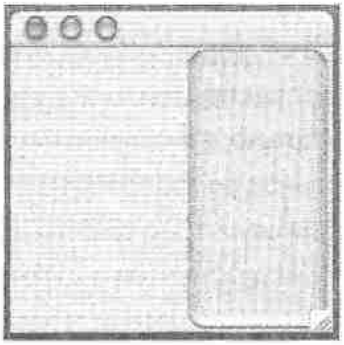


## Штанга для мозга

Как диспетчер BorderLayout поступил с размером кнопки?

Какие факторы должен учитывать диспетчер компоновки?

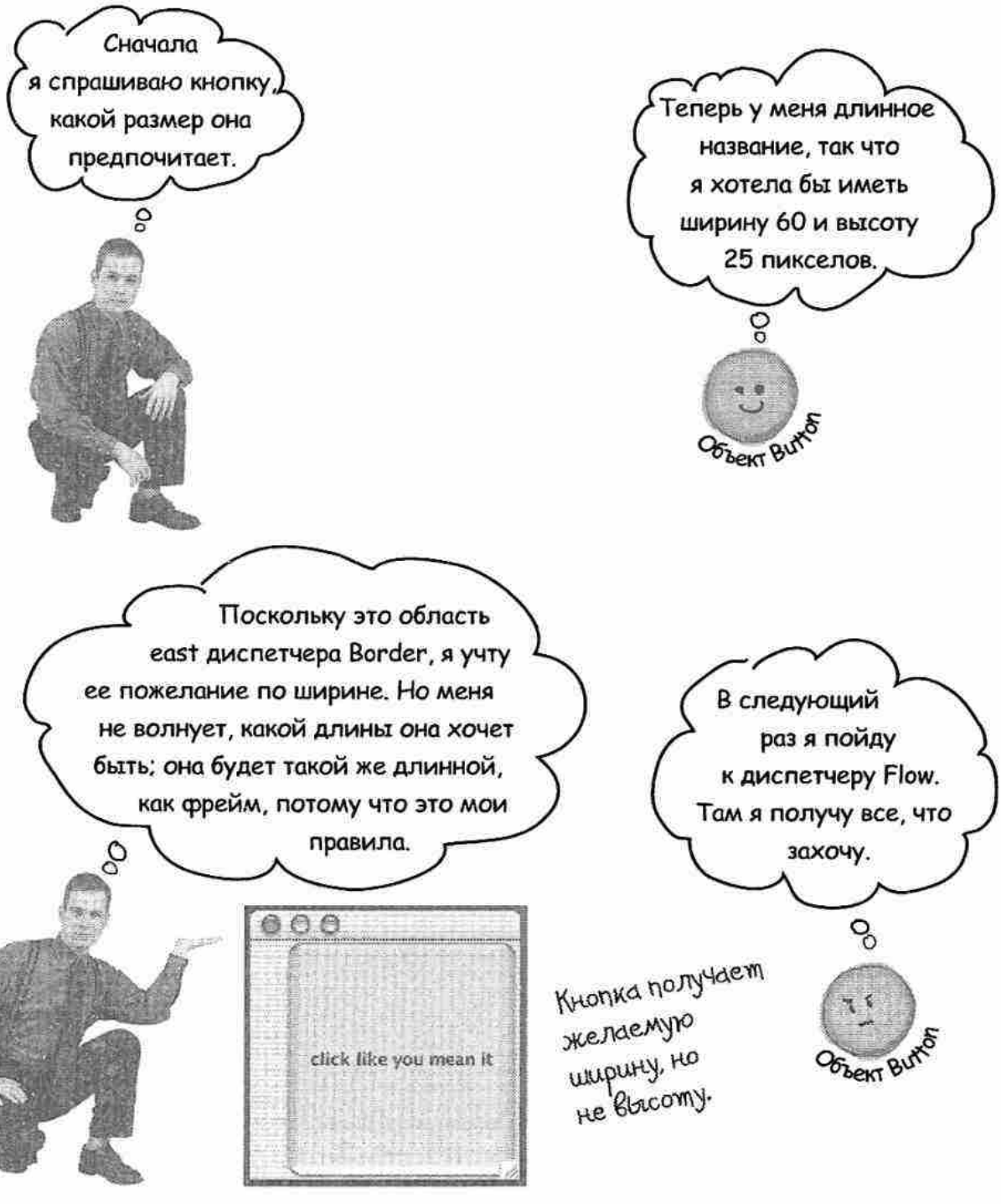
Почему кнопка не шире или выше?



## Смотрите, что происходит, когда мы даем кнопке длинное название...

Мы изменили только текст на кнопке.

```
public void go() {
    JFrame frame = new JFrame();
    JButton button = new JButton("click like you mean it");
    frame.getContentPane().add(BorderLayout.EAST, button);
    frame.setSize(200,200);
    frame.setVisible(true);
}
```



## Испытаем кнопку в области north

```
public void go() {
    JFrame frame = new JFrame();
    JButton button = new JButton("There is no spoon...");
    frame.getContentPane().add(BorderLayout.NORTH, button);
    frame.setSize(200,200);
    frame.setVisible(true);
}
```

Кнопка получила желаемую высоту, но ширина осталась, как у фрейма.

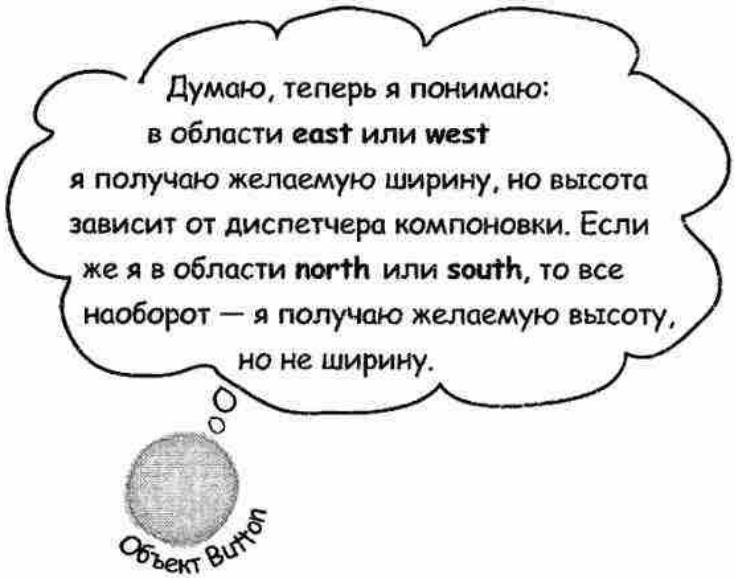
## Теперь сделаем так, чтобы кнопка попросила стать выше

Как мы это сделаем? Ширина кнопки и так максимальная — как у фрейма. Попытаемся сделать ее выше, добавив шрифт большего размера.

```
public void go() {
    JFrame frame = new JFrame();
    JButton button = new JButton("Click This!");
    Font bigFont = new Font("serif", Font.BOLD, 28);
    button.setFont(bigFont);
    frame.getContentPane().add(BorderLayout.NORTH, button);
    frame.setSize(200,200);
    frame.setVisible(true);
}
```

Увеличение размера шрифта заставит фрейм назначить больше места для высоты кнопки.

Ширина остается прежней, но теперь кнопка стала выше. Область north растянулась, чтобы вместить в себя кнопку с новой высотой.




# Но что происходит в области center?

## Область center получает все, что осталось!

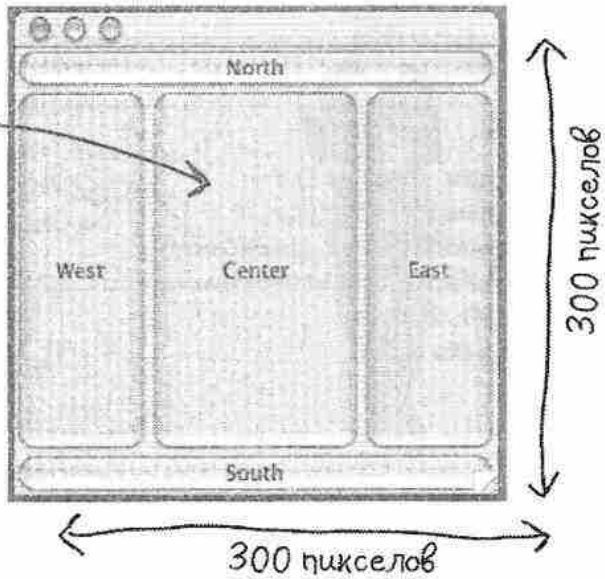
**Кроме одного особого случая, который мы рассмотрим позже.**

```
public void go() {
    JFrame frame = new JFrame();

    JButton east = new JButton("East");
    JButton west = new JButton("West");
    JButton north = new JButton("North");
    JButton south = new JButton("South");
    JButton center = new JButton("Center");

    frame.getContentPane().add(BorderLayout.EAST, east);
    frame.getContentPane().add(BorderLayout.WEST, west);
    frame.getContentPane().add(BorderLayout.NORTH, north);
    frame.getContentPane().add(BorderLayout.SOUTH, south);
    frame.getContentPane().add(BorderLayout.CENTER, center);

    frame.setSize(300,300);
    frame.setVisible(true);
}
```

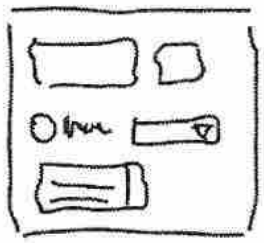
Компоненты в области center получают оставшееся место, основываясь на размерах фрейма (300x300 в этом коде).

Компоненты в областях east и west получают желаемую ширину.

Компоненты в областях north и south получают желаемую высоту.




Когда вы размещаете компонент в областях north или south, он растягивается по всей ширине фрейма, поэтому объекты, расположенные в областях east и west, не будут такими высокими, какими были бы при пустых north и south.



**FlowLayout управляет потоком компонентов: слева направо, сверху вниз, в порядке добавления.**

## Добавим панель в область east

Диспетчер компоновки объекта JPanel — это по умолчанию FlowLayout. Когда мы добавляем панель во фрейм, ее размер и расположение все еще находятся под контролем диспетчера компоновки BorderLayout. Но все, что размещено *внутри* панели (другими словами, компоненты, добавленные вызовом метода `panel.add(Компонент)`), будет находиться под контролем диспетчера FlowLayout. Начнем с того, что поместим пустую панель внутрь фрейма в области east, а затем добавим на нее объекты.

Панель пуста, поэтому в области east она имеет небольшую ширину.

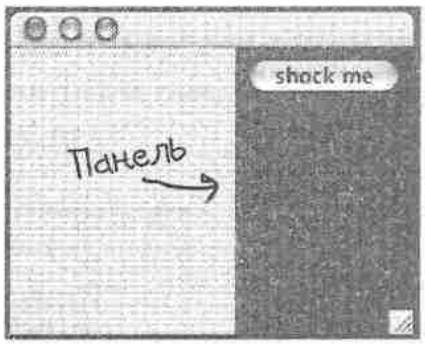
```
import javax.swing.*;
import java.awt.*;

public class Panel1 {

    public static void main (String[] args) {
        Panel1 gui = new Panel1();
        gui.go();
    }

    public void go() {
        JFrame frame = new JFrame();
        JPanel panel = new JPanel();
        panel.setBackground(Color.darkGray);
        frame.getContentPane().add(BorderLayout.EAST, panel);
        frame.setSize(200,200);
        frame.setVisible(true);
    }
}
```

Сделаем панель серой, чтобы различать ее на фоне фрейма.

## Добавим на панель кнопку

```
public void go() {
      JFrame frame = new JFrame();
      JPanel panel = new JPanel();
      panel.setBackground(Color.darkGray);

      JButton button = new JButton("shock me");

      panel.add(button);
      frame.getContentPane().add(BorderLayout.EAST, panel);

      frame.setSize(250,200);
      frame.setVisible(true);
}
```

Добавим кнопку на панель, а панель — внутрь фрейма. Диспетчер компоновки панели (Flow) контролирует кнопку, а диспетчер компоновки фрейма (Border) управляет панелью.




Панель расширилась, и кнопка получила желаемый размер как по ширине, так и по высоте, потому что панелью управляет диспетчер Flow, а кнопка — это часть панели (а не фрейма).

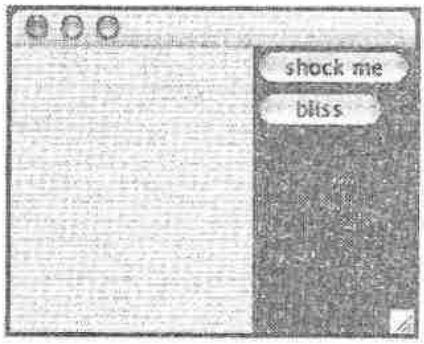
## Что произойдет, если мы добавим на панель две кнопки?

```
public void go() {
    JFrame frame = new JFrame();
    JPanel panel = new JPanel();
    panel.setBackground(Color.darkGray);

    JButton button = new JButton("shock me");
    JButton buttonTwo = new JButton("bliss");

    panel.add(button);
    panel.add(buttonTwo);

    frame.getContentPane().add(BorderLayout.EAST, panel);
    frame.setSize(250,200);
    frame.setVisible(true);
}
```

Создаем две кнопки.

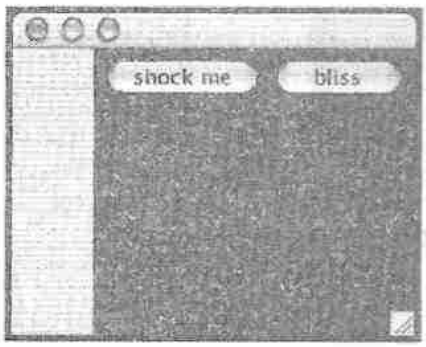
Добавляем обе кнопки на панель.

**Что мы *хотели*:**




Мы хотели расположить кнопки друг над другом

**Что мы *получили*:**




Панель расширилась, чтобы вместить обе кнопки, расположенные рядом.

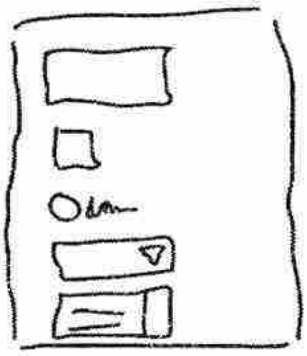
Заметьте, что кнопка bliss (Счастье) меньше кнопки shock me (Порази меня) — так работает диспетчер Flow. Кнопка получает такой размер, какой ей нужен.

### Наточите свой карандаш

Если приведенный выше код изменить следующим образом, то как будет выглядеть GUI?

```
JButton button = new JButton("shock me");
JButton buttonTwo = new JButton("bliss");
JButton buttonThree = new JButton("huh?");
panel.add(button);
panel.add(buttonTwo);
panel.add(buttonThree);
```

Нарисуйте, как, по вашему мнению, будет выглядеть GUI, если вы запустите этот код. А затем испытайте его!



# BoxLayout спешит на помощь!

## Он накладывает компоненты друг на друга, даже если есть место, позволяющее расположить их рядом.

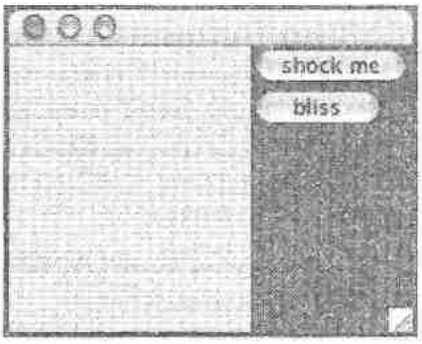
**В отличие от FlowLayout, BoxLayout может ввести новую строку, чтобы перенести на нее компоненты, даже если есть место, позволяющее вместить их по горизонтали.**

Теперь нужно изменить диспетчер компоновки панели с FlowLayout, установленного по умолчанию, на BoxLayout.

```
public void go() {
    JFrame frame = new JFrame();
    JPanel panel = new JPanel();
    panel.setBackground(Color.darkGray);

    panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

    JButton button = new JButton("shock me");
    JButton buttonTwo = new JButton("bliss");
    panel.add(button);
    panel.add(buttonTwo);
    frame.getContentPane().add(BorderLayout.EAST, panel);
    frame.setSize(250,200);
    frame.setVisible(true);
}
```

Изменяем диспетчер компоновки на новый экземпляр BoxLayout.

Конструктору диспетчера BoxLayout нужно знать, какие компоненты он размещает (в данном случае панель) и какую ось нужно использовать (мы указываем Y_AXIS для вертикального расположения).




Заметьте, как снова сузилась панель, — ей больше не нужно вмещать по горизонтали обе кнопки. Она сообщает фрейму, что ей хватит столько места, сколько нужно только для самой большой кнопки с названием «Порази меня».

## Это не глупые вопросы

**В: Как получается, что вы не можете добавлять объекты напрямую внутрь фрейма так же, как добавляете их на панель?**

О: Объект JFrame особенный, потому что указывается в участке кода, где начинается процесс вывода изображения на экран. Ваши Swing-компоненты полностью написаны на языке Java, в то время как JFrame должен подключаться непосредственно к ОС, чтобы получить доступ к дисплею. Думайте о панели с содержимым, как об уровне абстракции, на 100 % состоящем из Java и работающем *поверх* объекта JFrame. Или представьте, что JFrame — это оконная рама, а панель с содержимым — стекло. Вы даже можете поменять панель содержимого на собственную JPanel, чтобы назначить ее для фрейма, используя следующий код:

```
myFrame.setContentPane(myPanel);
```

**В: Могу ли я поменять диспетчер компоновки фрейма? Что нужно сделать, чтобы фрейм использовал Flow вместо Border?**

О: Самый легкий способ сделать это — создать панель, построить в ней GUI желаемым способом, а затем, используя код из предыдущего ответа, сделать ее панелью содержимого для фрейма (вместо панели, предоставляемой по умолчанию).

**В: Как задать другой размер? Существует ли метод setSize() для компонентов?**

О: Такой метод есть, но диспетчер компоновки не будет его учитывать. Существует различие между *предпочитаемым* размером компонента и размером, который *вы* хотите для него назначить. Первый — это размер, который действительно *нужен* компоненту (компонент принимает это решение самостоятельно). Диспетчер компоновки вызывает метод getPreferredSide(), и его не волнует, вызывали ли вы до этого метод setSize() для компонента.

**В: Могу ли я размещать объекты там, где захочу? Могу ли я отключить диспетчеры компоновки?**

О: Да. Вы можете вызвать метод `setLayout(null)`, а затем задать точное положение компонента на экране и его размеры. Хотя, надо отметить, почти всегда легче использовать диспетчеры компоновки.

## КЛЮЧЕВЫЕ МОМЕНТЫ

- Диспетчеры компоновки управляют размером и положением компонентов, вложенных внутрь других компонентов.
- При добавлении компонента внутрь другого компонента (который иногда называется фоновым) управление переходит к диспетчеру компоновки фонового компонента.
- Перед принятием решения о размещении диспетчер компоновки спрашивает у компонента его предпочитаемый размер. В зависимости от своих правил диспетчер компоновки может учитывать все, некоторые или вовсе не учитывать желания компонента.
- Диспетчер BorderLayout позволяет добавлять компонент в одну из пяти областей. Область при добавлении компонента указывают, используя следующий синтаксис:

  ```
  add(BorderLayout.EAST, panel);
  ```

- При работе с BorderLayout компоненты, расположенные в областях north и south, получают желаемую высоту, но не ширину. Компонентам, размещенным в области center, достается то пространство, которое осталось (если вы не используете метод `pack()`).
- Метод pack() — это что-то вроде термоусадочной упаковки для компонентов. Он использует полный предпочитаемый размер центрального компонента, затем определяет размер фрейма, взяв центр за отправную точку, и в итоге выстраивает все, что осталось, основываясь на том, что находится в других областях.
- FlowLayout располагает компоненты слева направо, сверху вниз, перенося их на новую строку, если они не помещаются по горизонтали.
- FlowLayout предоставляет компонентам их предпочитаемый размер в обоих направлениях как по ширине, так и по высоте.
- BoxLayout позволяет размещать компоненты по вертикали, даже если они могут поместиться рядом. Как и FlowLayout, диспетчер BoxLayout предоставляет желаемый размер компонента в обоих направлениях.
- BoxLayout — по умолчанию диспетчер компоновки для фрейма; FlowLayout — для панели.
- Если вы хотите, чтобы панель использовала вместо диспетчера Flow что-то другое, вызовите для нее метод `setLayout()`.

# Играем со Swing-компонентами

Мы уже рассмотрели основные особенности диспетчеров компоновки, поэтому попробуем поработать с наиболее распространенными компонентами: полем ввода, прокручиваемой текстовой областью, переключателем и списком. Мы не будем приводить весь API для каждого компонента, а остановимся на нескольких важных моментах.

## JTextField

### Конструкторы

```
JTextField field = new JTextField(20);
```

20 — это 20 столбцов, а не 20 пикселов. Это значение определяет предпочитаемую ширину поля ввода.

```
JTextField field = new JTextField("Ваше имя");
```

### Как использовать поле ввода

1. Извлекаем из него текст:

```
System.out.println(field.getText());
```

2. Помещаем в поле текст:

```
field.setText("Что угодно");
field.setText("");
```

Это очищает поле.

3. Получаем событие ActionEvent при нажатии пользователем клавиши Enter или Return:

Вы также можете регистрировать нажатия клавиш, если действительно хотите знать о каждом из них.

```
field.addActionListener(myActionListener);
```

4. Выделяем текст в поле:

```
field.selectAll();
```

5. Помещаем курсор назад в поле (чтобы пользователь сразу мог печатать):

```
field.requestFocus();
```

# JTextArea

В отличие от JTextField, компонент JTextArea может включать в себя несколько строк текста. Однако изначально он не содержит полос прокрутки и возможности переноса строк. Чтобы сделать JTextArea прокручиваемым, нужно вставить его внутрь объекта ScrollPane, который заботится о прокрутке текстовой области.

## Конструктор

```
JTextArea text = new JTextArea(10,20);
```

10 — это значит 10 строк (устанавливает предпочитаемую высоту).

20 — это 20 столбцов (определяет предпочитаемую ширину).

## Как использовать текстовую область

1. Делаем так, чтобы она имела только вертикальную полосу прокрутки:

```
JScrollPane scroller = new JScrollPane(text);
text.setLineWrap(true);

scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

Создаем объект JScrollPane и присваиваем ему текстовую область, которую он будет прокручивать.

Включаем перенос текста.

Указываем панели прокрутки использовать только вертикальную полосу.

Важно! Вы присваиваете объекту JScrollPane текстовую область (с помощью конструктора JScrollPane), а затем добавляете область прокрутки на панель. Вы не добавляете текстовую область непосредственно на панель!

2. Заменяем находящийся в области текст:

```
text.setText("Не все потерявшиеся — бродяги");
```

3. Вносим изменения в существующий текст:

```
text.append("Кнопка нажата");
```

4. Выделяем текст в поле:

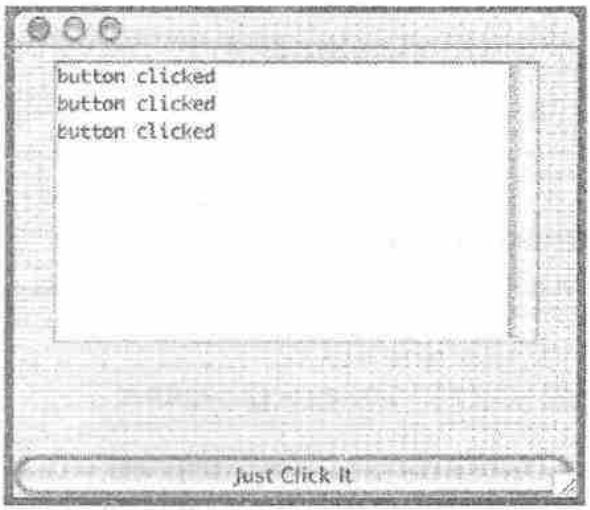
```
text.selectAll();
```

5. Помещаем курсор назад в поле (чтобы пользователь сразу мог печатать):

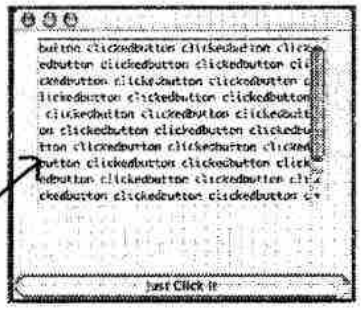
```
text.requestFocus();
```

## Пример JTextArea

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TextArea1 implements ActionListener {

    JTextArea text;

    public static void main (String[] args) {
       TextArea1 gui = new TextArea1();
       gui.go();
    }

    public void go() {
       JFrame frame = new JFrame();
       JPanel panel = new JPanel();
       JButton button = new JButton("Just Click It");
       button.addActionListener(this);
       text = new JTextArea(10,20);
       text.setLineWrap(true);

       JScrollPane scroller = new JScrollPane(text);
       scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
       scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

       panel.add(scroller);

       frame.getContentPane().add(BorderLayout.CENTER, panel);
       frame.getContentPane().add(BorderLayout.SOUTH, button);

       frame.setSize(350,300);
       frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent ev) {
       text.append("button clicked \n ");
    }
}
```




Вставьте перенос строки, чтобы слова переходили на новую строку при каждом нажатии кнопки, иначе они будут идти друг за другом.



## JCheckBox

### Конструктор

```
JCheckBox check = new JCheckBox("Goes to 11");
```

### Как использовать флажок

1. Отслеживаем событие флажка (установлен он или снят):

```
check.addItemListener(this);
```

2. Обрабатываем событие и определяем, установлен флажок или нет:

```
public void itemStateChanged(ItemEvent ev) {
   String onOrOff = "off";
   if (check.isSelected()) onOrOff = "on";
   System.out.println("Check box is " + onOrOff);
}
```

3. Устанавливаем или снимаем флажок:

```
check.setSelected(true);
check.setSelected(false);
```

## Это не глупые вопросы

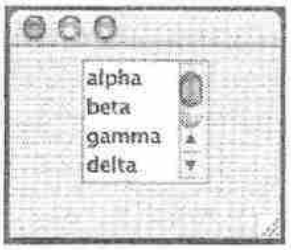
**В: Вам не кажется, что от диспетчера компоновки больше проблем, чем пользы? Если необходимо преодолеть все эти трудности, то с таким же успехом можно просто внести в код размеры и координаты всех элементов.**

О: Получить от диспетчера компоновки нужный результат может быть нелегко. Но подумайте о том, что он для вас делает. Даже такая, на первый взгляд, простая задача, как вычисление положения объектов на экране, может оказаться достаточно сложной. Например, диспетчер компоновки следит за тем, чтобы ваши компоненты не перекрывали друг друга. Иначе говоря, он знает, как распределить пространство между компонентами (и между краями фрейма). Конечно, вы можете сделать это самостоятельно, но как быть, если нужно, чтобы компоненты размещались плотно друг к другу? Если вы расположите их правильно вручную, то это будет работать только для вашей JVM!

Почему? Потому что компоненты могут немного различаться в зависимости от платформы, особенно если используют системный внешний вид. Такие тонкие оформительские элементы, как скос кнопок, могут настолько различаться, что ровно выстроенные компоненты на одной платформе будут кривыми на другой.

Теперь представьте, что произойдет, когда пользователь будет изменять размер окна! Или, допустим, у вас динамический GUI, где компоненты появляются и исчезают. Если вам придется каждый раз размещать все компоненты при изменениях размера или содержимого фонового компонента... Тогда вам не позавидуешь!

## JList




Конструктор объекта JList (списка) принимает массив объектов любого типа. Им необязательно иметь строковый тип, но отображаться в списке они будут как String.

### Конструктор

```
String [] listEntries = {"alpha", "beta", "gamma", "delta",
                         "epsilon", "zeta", "eta", "theta "};

list = new JList(listEntries);
```

Здесь все происходит так же, как и с JTextArea, — вы создаете панель JScrollPane (и присваиваете ей список), а затем добавляете на нее область прокрутки (не список).

### Как использовать список

① Создаем для него вертикальную полосу прокрутки:

```
JScrollPane scroller = new JScrollPane(list);
scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

② Устанавливаем количество строк, изображаемых до прокрутки:

```
list.setVisibleRowCount(4);
```

③ Ограничиваем выбор до одной строки:

```
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

④ Регистрируем события выбора в списке:

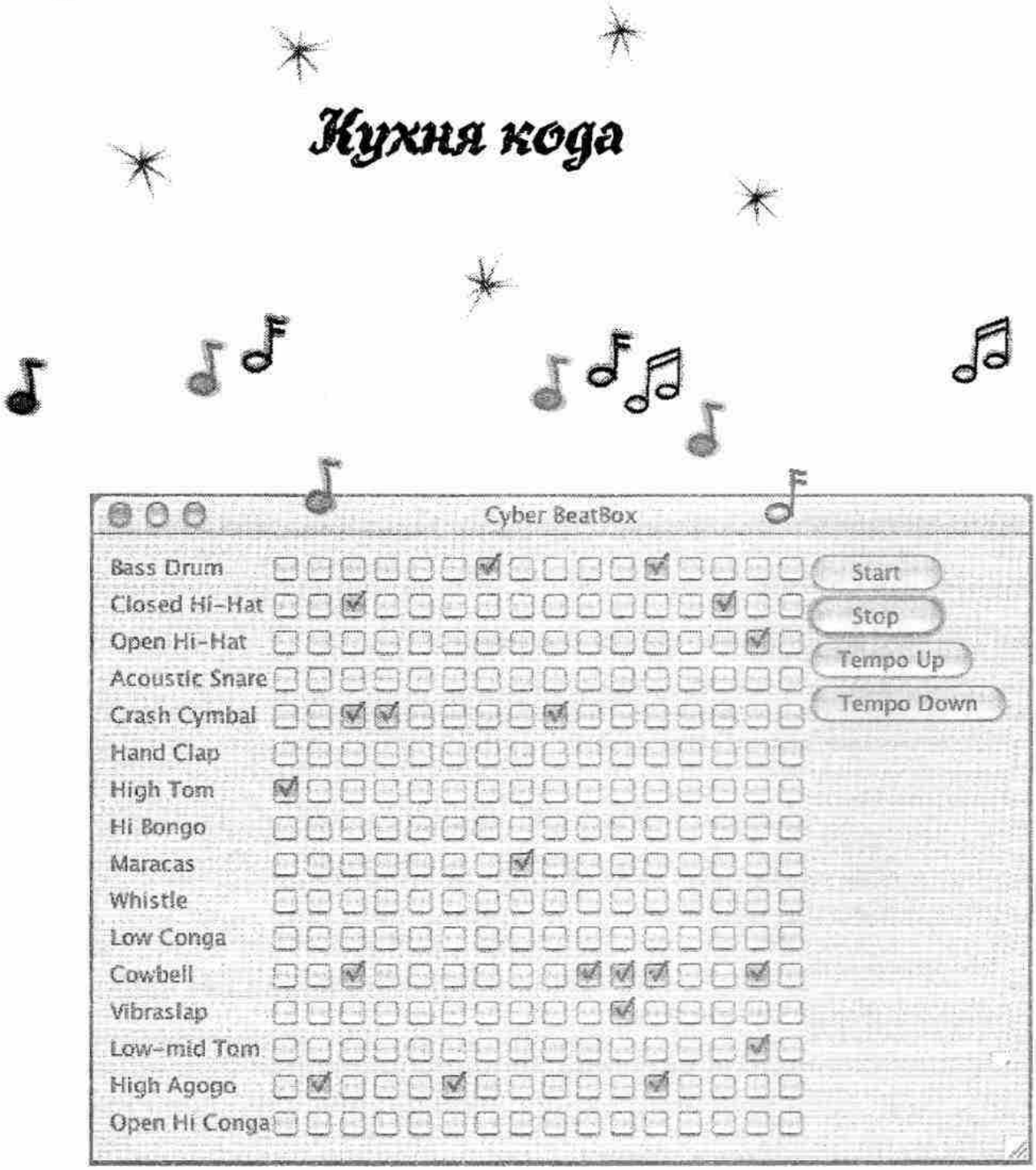
```
list.addListSelectionListener(this);
```

⑤ Обрабатываем события (выясняем, какая строка в списке была выбрана):

```
public void valueChanged(ListSelectionEvent lse) {
       if( !lse.getValueIsAdjusting()) {
         String selection = (String) list.getSelectedValue();
         System.out.println(selection);
       }
  }
```

Вы получите событие дважды, если не вставите в код такую проверку условия.

Метод getSelectedValue() фактически возвращает тип Object. Список не ограничен только объектами String.

# Кухня кода

***Задание необязательно, но желательно выполнить. Здесь мы создадим полноценный битбокс (BeatBox) с GUI. В главе 14 мы изучим, как сохранять и восстанавливать барабанные схемы. А в главе 15, посвященной работе с сетью, мы превратим BeatBox в рабочий клиент для чата.***

# Создаем BeatBox

Здесь приведен полный код для создания BeatBox с кнопками для запуска, остановки и изменения темпа. Представленный листинг полный и снабжен подробными комментариями. Однако мы пока рассмотрим его основные части.

1. Проектируем GUI, который будет иметь 256 снятых флажков (JCheckBox), 16 меток (JLabel) для названий инструментов и четыре кнопки.

2. Связываем ActionListener с каждой из четырех кнопок. Слушатели для флажков нам не нужны, потому что мы не будем динамически изменять схему звучания (то есть когда пользователь устанавливает флажки). Вместо этого мы ждем, пока пользователь нажмет кнопку Старт, а затем пробегаем через все 256 флажков, чтобы получить их состояния, и, основываясь на этом, создаем MIDI-дорожку.

3. Устанавливаем систему MIDI (мы делали это раньше), получая доступ к синтезатору, создаем объект Sequencer и дорожку для него. Мы используем новый метод интерфейса Sequencer setLoopCount(), появившийся в Java 5.0. Он позволяет определять желаемое количество циклов последовательности. Мы также будем использовать коэффициент темпа последовательности для настройки уровня темпа и сохранять новый темп от одной итерации цикла к другой.

4. При нажатии пользователем кнопки Старт начинается настоящее действие. Обработчик событий кнопки запускает метод buildTrackAndStart(). В нем мы пробегаем через все 256 флажков (по одному ряду за один раз, один инструмент на все 16 тактов), чтобы получить их состояния, а затем используем эту информацию для создания MIDI-дорожки (с помощью удобного метода makeEvent(), который мы применяли в предыдущей главе). Как только дорожка построена, мы запускаем секвенсор, который будет играть (потому что мы его зацикливаем), пока пользователь не нажмет кнопку Стоп.

## Код BeatBox

```
import java.awt.*;
import javax.swing.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;


public class BeatBox {

    JPanel mainPanel;
    ArrayList<JCheckBox> checkboxList;
    Sequencer sequencer;
    Sequence sequence;
    Track track;
    JFrame theFrame;

    String[] instrumentNames = {"Bass Drum", "Closed Hi-Hat",
       "Open Hi-Hat","Acoustic Snare", "Crash Cymbal", "Hand Clap",
       "High Tom", "Hi Bongo", "Maracas", "Whistle", "Low Conga",
       "Cowbell", "Vibraslap", "Low-mid Tom", "High Agogo",
       "Open Hi Conga"};
    int[] instruments = {35,42,46,38,49,39,50,60,70,72,64,56,58,47,67,63};


    public static void main (String[] args) {
        new BeatBox2().buildGUI();
    }

    public void buildGUI() {
        theFrame = new JFrame("Cyber BeatBox");
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        BorderLayout layout = new BorderLayout();
        JPanel background = new JPanel(layout);
        background.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

        checkboxList = new ArrayList<JCheckBox>();
        Box buttonBox = new Box(BoxLayout.Y_AXIS);

        JButton start = new JButton("Start");
        start.addActionListener(new MyStartListener());
        buttonBox.add(start);

        JButton stop = new JButton("Stop");
        stop.addActionListener(new MyStopListener());
        buttonBox.add(stop);

        JButton upTempo = new JButton("Tempo Up");
        upTempo.addActionListener(new MyUpTempoListener());
        buttonBox.add(upTempo);

        JButton downTempo = new JButton("Tempo Down");
```

Мы храним флажки в массиве ArrayList.

Это названия инструментов в виде строкового массива, предназначенные для создания меток в пользовательском интерфейсе (на каждый ряд).

Эти числа представляют собой фактические барабанные клавиши. Канал барабана — это что-то вроде фортепиано, только каждая клавиша на нем — отдельный барабан. Номер 35 — это клавиша для Bass drum, а 42 — Closed Hi-Hat и т. д.

Пустая граница позволяет создать поля между краями панели и местом размещения компонентов.

Здесь нет ничего нового — с большей частью кода вы уже знакомы.

```
    downTempo.addActionListener(new MyDownTempoListener());
    buttonBox.add(downTempo);

    Box nameBox = new Box(BoxLayout.Y_AXIS);
    for (int i = 0; i < 16; i++) {
       nameBox.add(new Label(instrumentNames[i]));
    }

    background.add(BorderLayout.EAST, buttonBox);
    background.add(BorderLayout.WEST, nameBox);

    theFrame.getContentPane().add(background);

    GridLayout grid = new GridLayout(16,16);
    grid.setVgap(1);
    grid.setHgap(2);
    mainPanel = new JPanel(grid);
    background.add(BorderLayout.CENTER, mainPanel);

    for (int i = 0; i < 256; i++) {
        JCheckBox c = new JCheckBox();
        c.setSelected(false);
        checkboxList.add(c);
        mainPanel.add(c);
    } // Конец цикла

    setUpMidi();

    theFrame.setBounds(50,50,300,300);
    theFrame.pack();
    theFrame.setVisible(true);
} // Закрываем метод


public void setUpMidi() {
  try {
    sequencer = MidiSystem.getSequencer();
    sequencer.open();
    sequence = new Sequence(Sequence.PPQ,4);
    track = sequence.createTrack();
    sequencer.setTempoInBPM(120);

  } catch(Exception e) {e.printStackTrace();}
} // Закрываем метод
```

Еще код для GUI. Ничего особенного.

Создаем флажки, присваиваем им значения false (чтобы они не были установлены), а затем добавляем их в массив ArrayList и на панель.

Обычный MIDI-код для получения синтезатора, секвенсора и дорожки. По-прежнему ничего особенного.

Вот здесь все и происходит! Мы преобразуем состояния флажков в MIDI-события и добавляем их на дорожку.

```
public void buildTrackAndStart() {
  int[] trackList = null;

  sequence.deleteTrack(track);
  track = sequence.createTrack();

   for (int i = 0; i < 16; i++) {
     trackList = new int[16];

     int key = instruments[i];

     for (int j = 0; j < 16; j++ ) {

        JCheckBox jc = (JCheckBox) checkboxList.get(j + (16*i));
        if ( jc.isSelected()) {
           trackList[j] = key;
        } else {
           trackList[j] = 0;
        }
     } // Закрываем внутренний цикл

     makeTracks(trackList);
     track.add(makeEvent(176,1,127,0,16));
  } // Закрываем внешний

  track.add(makeEvent(192,9,1,0,15));
  try {

     sequencer.setSequence(sequence);
      sequencer.setLoopCount(sequencer.LOOP_CONTINUOUSLY);
      sequencer.start();
      sequencer.setTempoInBPM(120);
  } catch(Exception e) {e.printStackTrace();}
} // Закрываем метод buildTrackAndStart

public class MyStartListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        buildTrackAndStart();
    }
} // Закрываем внутренний класс
```

Создаем массив из 16 элементов, чтобы хранить значения для каждого инструмента, на все 16 тактов.

Избавляемся от старой дорожки и создаем новую.

Делаем это для каждого из 16 рядов (то есть для Bass, Congo и т. д.).

Задаем клавишу, которая представляет инструмент (Bass, Hi-Hat и т. д.). Массив содержит MIDI-числа для каждого инструмента.

Делаем это для каждого такта текущего ряда.

Установлен ли флажок на этом такте? Если да, то помещаем значение клавиши в текущую ячейку массива (ячейку, которая представляет такт). Если нет, то инструмент не должен играть в этом такте, поэтому присвоим ему 0.

Для этого инструмента и для всех 16 тактов создаем события и добавляем их на дорожку.

Мы всегда должны быть уверены, что событие на такте 16 существует (они идут от 0 до 15). Иначе BeatBox может не пройти все 16 тактов, перед тем как заново начнет последовательность.

Позволяет задать количество повторений цикла или, как в этом случае, непрерывный цикл.

Теперь мы проигрываем мелодию!

Первый из внутренних классов — слушателей для кнопок.

```
public class MyStopListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        sequencer.stop();
    }
} // Закрываем внутренний класс

public class MyUpTempoListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
       float tempoFactor = sequencer.getTempoFactor();
        sequencer.setTempoFactor((float)(tempoFactor * 1.03));
    }
 } // Закрываем внутренний класс

 public class MyDownTempoListener implements ActionListener {
     public void actionPerformed(ActionEvent a) {
       float tempoFactor = sequencer.getTempoFactor();
        sequencer.setTempoFactor((float)(tempoFactor * .97));
    }
} // Закрываем внутренний класс
```

Другие внутренние классы — слушатели для кнопок.

Коэффициент темпа определяет темп синтезатора. По умолчанию он равен 1.0, поэтому щелчком кнопкой мыши можно изменить его на +/- 3 %.

Метод создает события для одного инструмента за каждый проход цикла для всех 16 тактов. Можно получить int[] для Bass drum, и каждый элемент массива будет содержать либо клавишу этого инструмента, либо ноль. Если это ноль, то инструмент не должен играть на текущем такте. Иначе нужно создать событие и добавить его в дорожку.

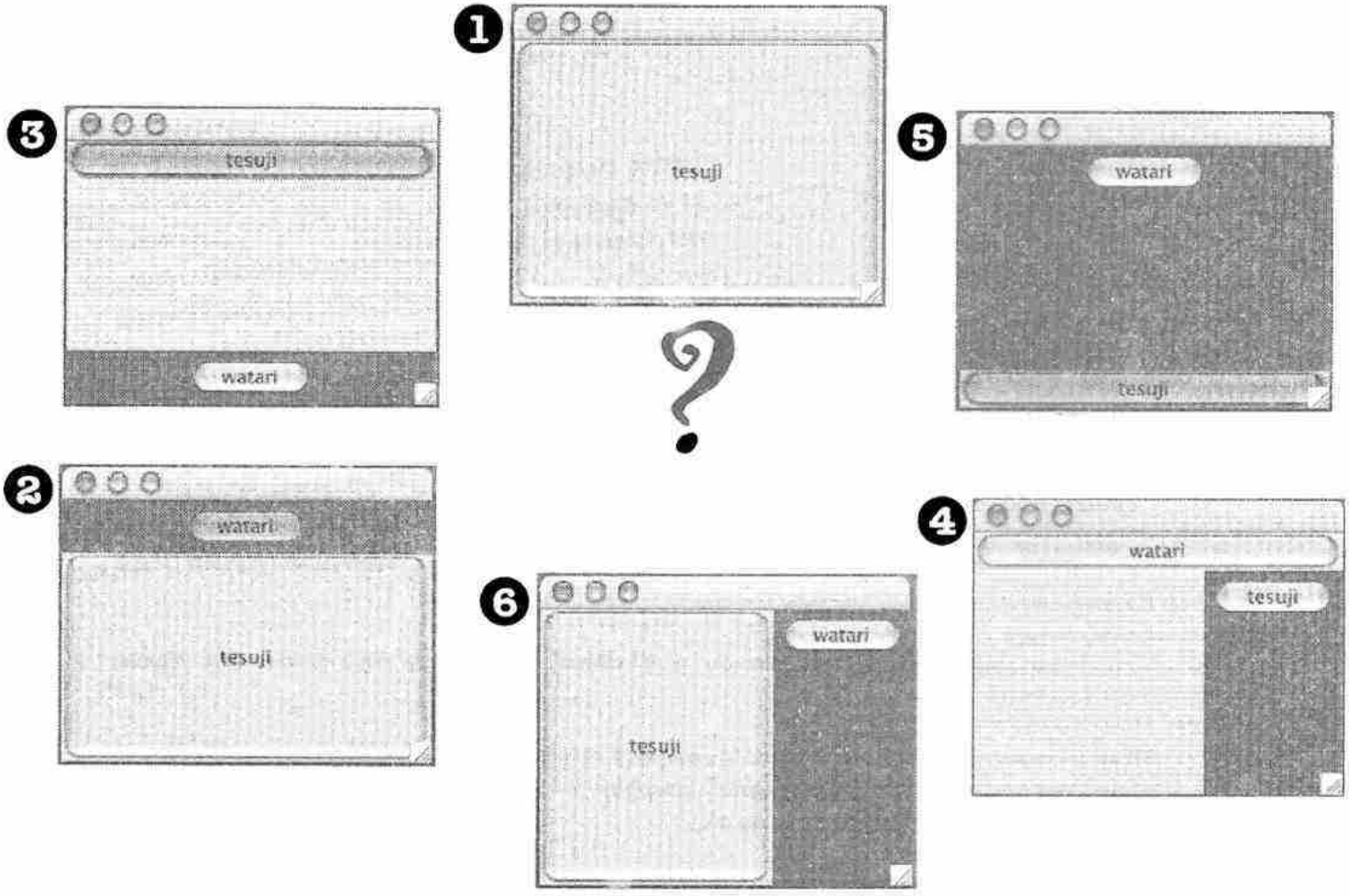
```
public void makeTracks(int[] list) {

   for (int i = 0; i < 16; i++) {
      int key = list[i];

      if (key != 0) {
         track.add(makeEvent(144,9,key, 100, i));
         track.add(makeEvent(128,9,key, 100, i+1));
      }
   }
}

public  MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
        ShortMessage a = new ShortMessage();
        a.setMessage(comd, chan, one, two);
        event = new MidiEvent(a, tick);

    } catch(Exception e) {e.printStackTrace(); }
    return event;
}

} // Закрываем класс
```

Создаем события включения и выключения и добавляем их в дорожку.

Это полезный метод из предыдущей главы Кухни кода.

## Какие фрагменты кода создают следующие схемы размещения?

Пять из шести представленных ниже окон были получены в результате работы фрагментов кода, находящихся на следующей странице. Свяжите каждый из пяти фрагментов с одним рисунком, который, по вашему мнению, соответствует выбранному коду.



# Фрагменты кода

**D**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

---

**B**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

---

**C**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

---

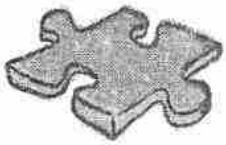
**A**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH,buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

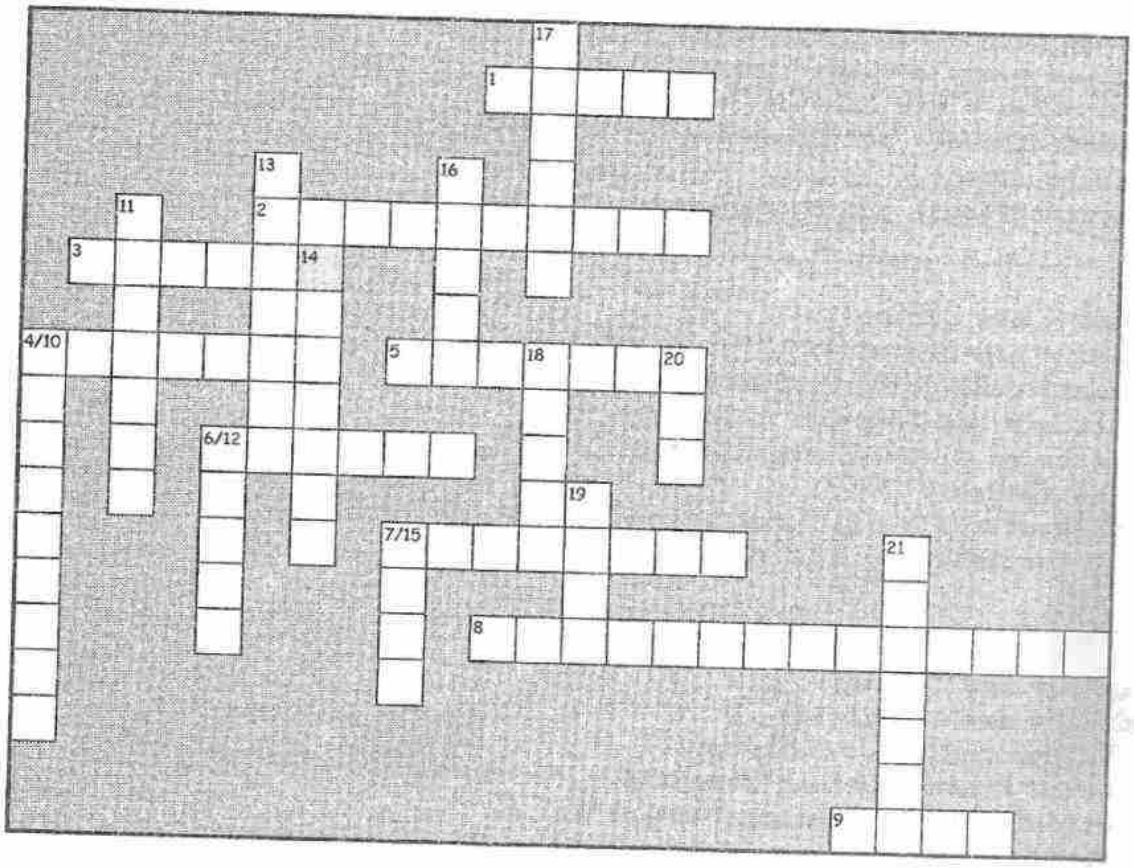
---

**E**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.NORTH,button);
```



# GUI-Cross 7.0



Вы можете это сделать.

## По горизонтали

1. Внешний вид Java.
2. Схема _____.
3. Приятель кнопки.
4. Происшествие.
5. Веселее, чем текст.
6. Виджет заднего плана.
7. То, что движется.
8. Дом для ActionPerformed.
9. Местоположение помощи.

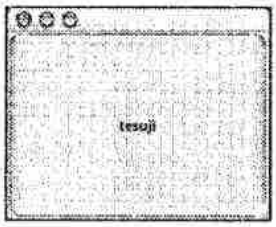
## По вертикали

10. Следит за событиями.
11. Их можно выбирать много.
12. Диспетчер компоновки, по умолчанию служащий диспетчером для объекта Jpanel.
13. Инструкция диспетчера.
14. Друг внутреннего.
15. Внешний вид для Mac.
16. Верхняя область диспетчера Border.
17. Область диспетчера Border, которая распоряжается тем, что осталось.
18. Объект, для которого диспетчер компоновки Border является диспетчером по умолчанию.
19. Правая область диспетчера Border.
20. Папа для Swing.
21. Границы фрейма.

# Ответы

## Фрагменты кода

**1**




**C**

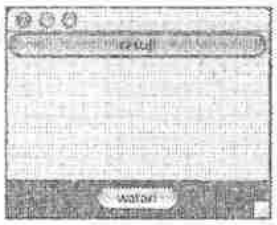
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

**2**

**D**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

**3**




**E**

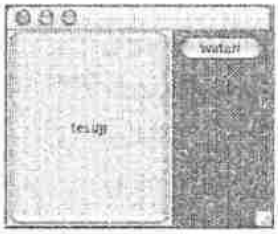
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.NORTH,button);
```

**4**

**A**

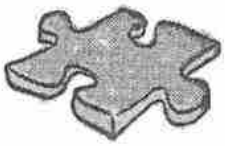
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH,buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

**6**



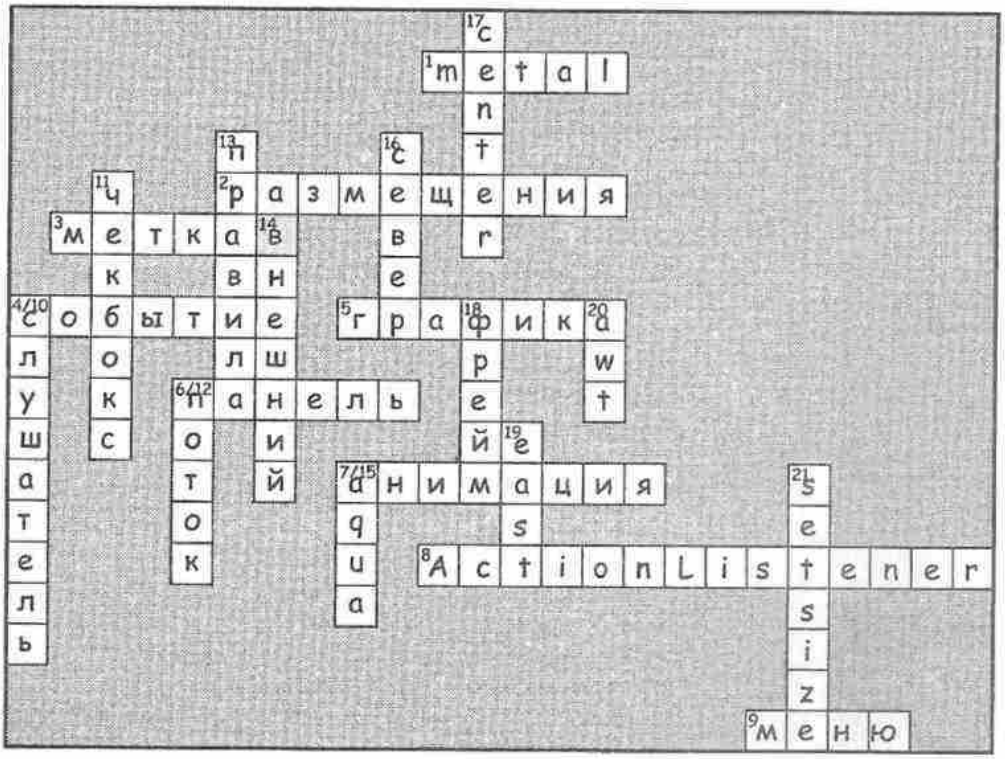

**B**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
frame.getContentPane().add(BorderLayout.EAST, panel);
```



## Ответы

# GUI-Cross 7.0



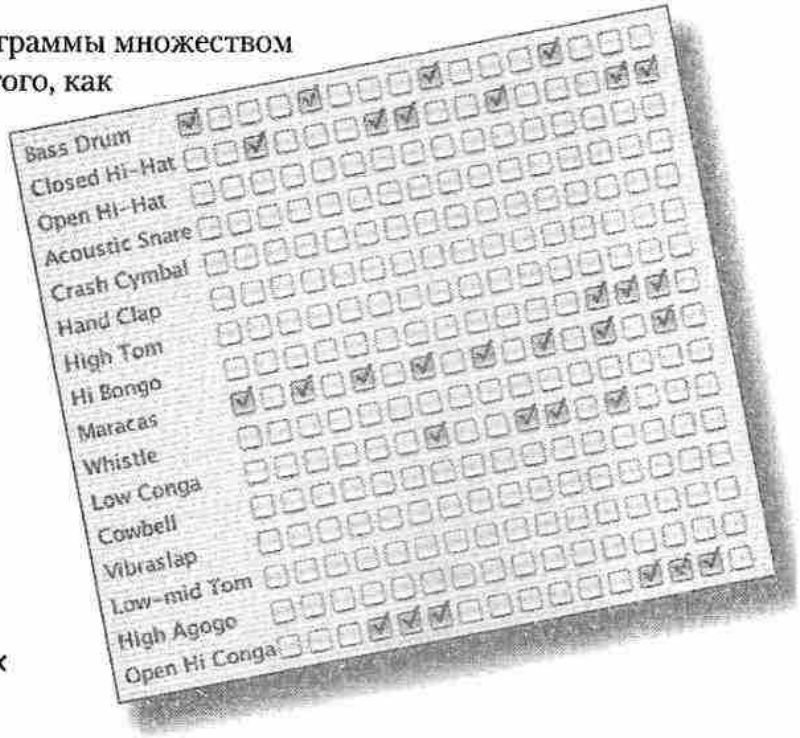
# Сохранение объектов

**Объекты могут быть сплющенными и восстановленными.** Они характеризуются состоянием и *поведением*. Поведение содержится в *классе*, а *состояние* определяется каждым *объектом* в отдельности. Что же происходит при сохранении состояния объекта? Если вы создаете игру, вам понадобится функция сохранения/восстановления игрового процесса. Если вы пишете приложение, которое рисует графики, вам также понадобится функция сохранения/восстановления. Если вашей программе нужно сохранить состояние, *можете сделать это сложным способом*, запрашивая каждый объект, а затем тщательно записывая значения всех переменных экземпляра в файл в созданном вами формате. Вы **также можете сделать это легким объектно ориентированным способом** — нужно просто сублимировать/сплющить/сохранить/опустошить сам объект, а затем реконструировать/надуть/восстановить/наполнить его, чтобы получить снова. Но *иногда* вам придется делать это и сложным способом, особенно когда файл, сохраняемый вашим приложением, должен читаться другими программами, написанными не на Java, поэтому мы рассмотрим оба метода.

# Лови ритм

Вы *создали* превосходную структуру и хотите ее сохранить. Можно записать ее на лист бумаги, но вместо этого лучше нажать кнопку ***Save*** (Сохранить) (или выбрать пункт Save (Сохранить) в меню File (Файл)). Затем нужно присвоить ей имя, выбрать каталог и наконец выдохнуть, зная, что шедевр не исчезнет вместе с синим экраном смерти.

Вы можете сохранить состояние своей Java-программы множеством способов, и выбор, вероятно, будет зависеть от того, как вы планируете *использовать* сохраненное состояние. Вот варианты, которые мы рассмотрим в данной главе.




**Если ваши данные будут задействованы только в создавшей их Java-программе:**

**① Используйте сериализацию.**

Создайте файл, который будет хранить преобразованные (сериализованные) объекты. Затем заставьте программу прочитать их из файла и вернуть им состояние живых объектов, населенных множеством элементов.

**Если ваши данные будут использоваться другими программами:**

**② Создайте простой текстовый файл.**

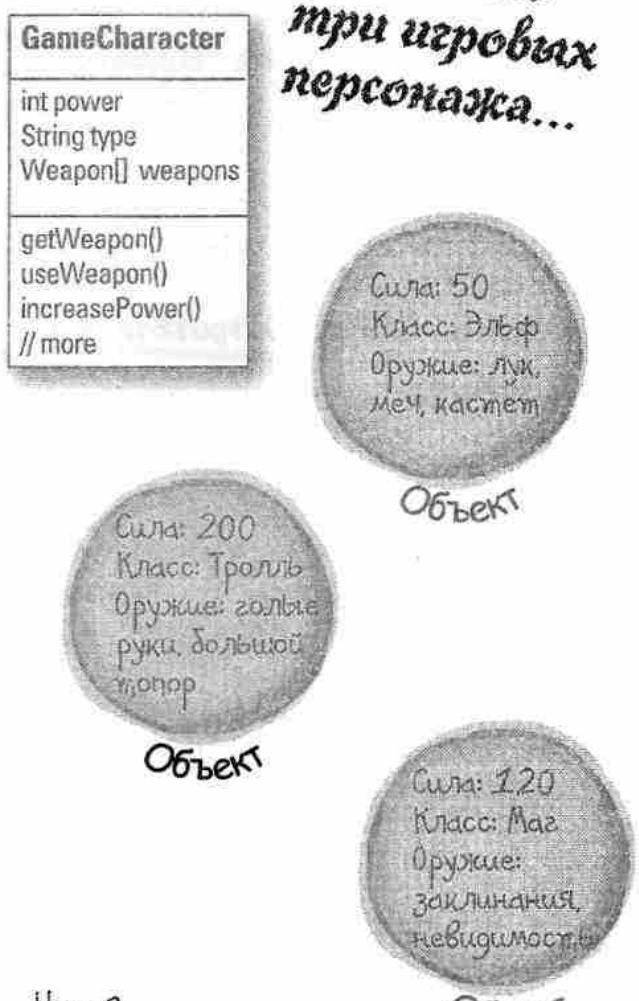
Создайте файл с разделителями, которые смогут распознать другие программы. Например, это может быть файл с разделителями на основе табуляции, который будет доступен для использования электронной таблице или приложению с базой данных.

Конечно, это не единственные варианты. Вы можете сохранить информацию в любом выбранном формате. Например, вместо того чтобы записывать символы, попробуйте сохранить данные в виде байтов. Или же записывайте любые примитивы из языка Java *как* Java-примитивы — существуют методы, позволяющие вводить значения типов int, long, boolean и т. д. Независимо от используемого способа фундаментальные методы ввода/вывода в значительной степени представляют собой то же самое: нужно записать данные *куда-нибудь*, и обычно это место — файл на диске либо поток из сетевого соединения. Чтение данных — такой же процесс, но в обратном порядке: происходит считывание информации из файла на диске или из сетевого соединения. И помните: все, о чем мы тут рассказываем, актуально до тех пор, пока вы не начнете работать с настоящими базами данных.

# Сохранение состояния

Представьте, что у вас есть программа, например приключенческая игра в жанре фэнтези. За один раз ее нельзя пройти — нужен не один сеанс. При прохождении игры герои становятся сильнее, слабее, умнее и т. д., а также собирают и используют (или теряют) оружие. Наверняка вам не хочется начинать игру сначала каждый раз при запуске — потребуется вечность, чтобы подготовить героев к захватывающей битве. Таким образом, нужен способ сохранения состояния персонажей и возможность их восстановления при возобновлении игры. Как разработчик этой игры вы хотите, чтобы процесс сохранения/восстановления был максимально легким (и защищенным от неправильного использования).

*Представьте, что вам нужно сохранить три игровых персонажа...*




Человеку сложно прочитать сериализованный файл, но вашей программе будет намного легче (и безопаснее) восстановить три объекта с помощью сериализации, чем считывать значения переменных объектов, сохраненных в текстовом файле. Например, представьте, что вы случайно считали значения в неправильном порядке. Класс персонажа может стать кастетом вместо эльфа, а эльф окажется оружием.

### ① Способ первый

**Запишите три сериализованных объекта персонажей в файл.**

Создайте файл и запишите три сериализованных объекта персонажей. Файл будет представлять собой абракадабру, если вы попытаетесь прочитать его, как текст:

```
¨ÌsrGameCharacter
¨%gê8MÛIpowerLjava/lang/
String;[weaponst[Ljava/lang/
String;xp2tlfur[Ljava.lang.String;≠"VÁÈ{Gxptb
owtswordtdustsq~»tTrolluq~tbare handstbig ax
sq~xtMagicianuq~tspellstinvisibility
```

### ② Способ второй

**Создайте простой текстовый файл.**

Создайте файл и запишите три текстовых строки — для каждого персонажа, разделяя части состояний запятыми:

```
50,Эльф,лук,меч,кастет
200,Тролль,голые руки,большой топор
120,Маг,заклинания,невидимость
```

# Запись сериализованного объекта в файл

Здесь перечислены шаги по сериализации (сохранению) объекта. Не пытайтесь все запомнить; мы еще рассмотрим их подробнее в этой главе.

Если файла MyGame.ser не существует, то он будет создан автоматически.

**1 Создаем объект FileOutputStream.**

```
FileOutputStream fileStream = new FileOutputStream("MyGame.ser");
```

Создаем объект FileOutputStream. Этот объект знает, как подключиться к файлу (и как создать его).

**2 Создаем ObjectOutputStream.**

```
ObjectOutputStream os = new ObjectOutputStream(fileStream);
```

ObjectOutputStream позволит записывать объекты, но не сможет напрямую подключаться к файлу. Ему потребуется «помощник». Это фактически называется «связыванием» одного потока с другим.

**3 Записываем объект.**

```
os.writeObject(characterOne);
os.writeObject(characterTwo);
os.writeObject(characterThree);
```

Сериализируем объекты, на которые указывают ссылки characterOne, characterTwo, characterThree, и записываем их в файл MyGame.ser.

**4 Закрываем ObjectOutputStream.**

```
os.close();
```

Закрывая поток верхнего уровня, мы закрываем и исходный поток, так что FileOutputStream и файл закроются автоматически.

# Данные в потоках перемещаются с одного места на другое

Потоки для соединения представляют собой подключение к источнику или к пункту назначения (файлу, сокету и т. д.), тогда как цепные потоки не могут соединяться и должны связываться с потоком для соединения.

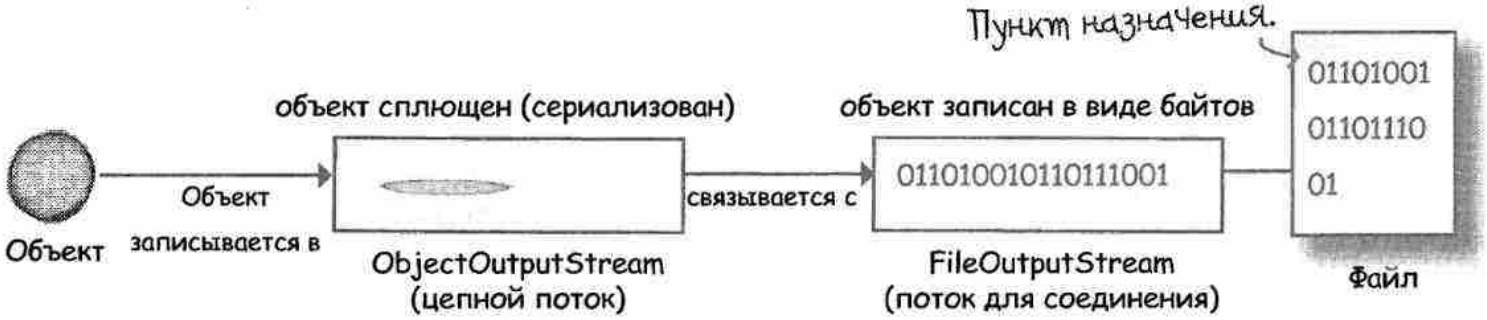
API ввода/вывода в Java содержит потоки для ***соединений***, которые представляют собой подключения к пунктам назначения и источникам, например файлам или сетевым сокетам. Этот API соединяет потоки, работающие только в связке с другими потоками.

Чтобы сделать что-нибудь полезное, часто приходится связывать по меньшей мере два потока — *один* из них играет роль соединения, а *другой* вызывает методы. Почему два? Потому что потоки для *соединений* обычно низкоуровневые. FileOutputStream (поток для соединения), к примеру, имеет методы для записи *байтов*. Но нужно записывать не *байты*, а *объекты*, так что понадобится высокоуровневый *цепной* поток.

Хорошо, тогда почему бы не использовать одиночный поток, который *точно* сделает то, что нужно? Поток, который позволит записывать объекты, но при этом будет преобразовывать их в байты. Нужно мыслить категориями объектно ориентированного программирования. Каждый класс делает хорошо только *одну* вещь: FileOutputStream записывает байты в файл; ObjectOutputStream преобразует объекты в данные, которые могут быть записаны в поток. Мы создаем FileOutputStream, который позволяет делать записи в файл, и прикрепляем к нему ObjectOutputStream (цепной поток).

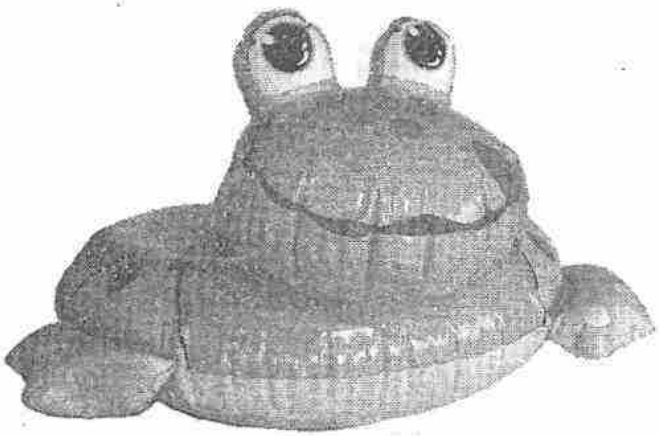
Когда мы вызываем метод writeObject() из объекта ObjectOutputStream, объект подключается к потоку и переходит к FileOutputStream, где в результате записывается в файл в виде байтов.

Способность смешивать и подбирать различные комбинации цепных потоков и потоков для соединения дает вам огромную гибкость! Если бы пришлось использовать *единственный* потоковый класс, вы оказались бы во власти разработчиков API, надеясь, что они предусмотрели *все*, что вы хотели бы сделать. А со связыванием вы можете собирать свои собственные *пользовательские* цепочки.



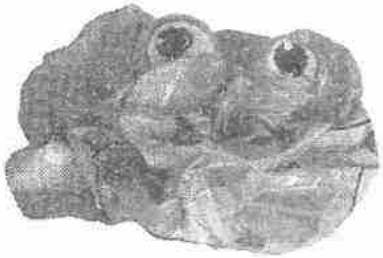
# Что на самом деле происходит с объектом при сериализации

## ❶ Объект в куче.

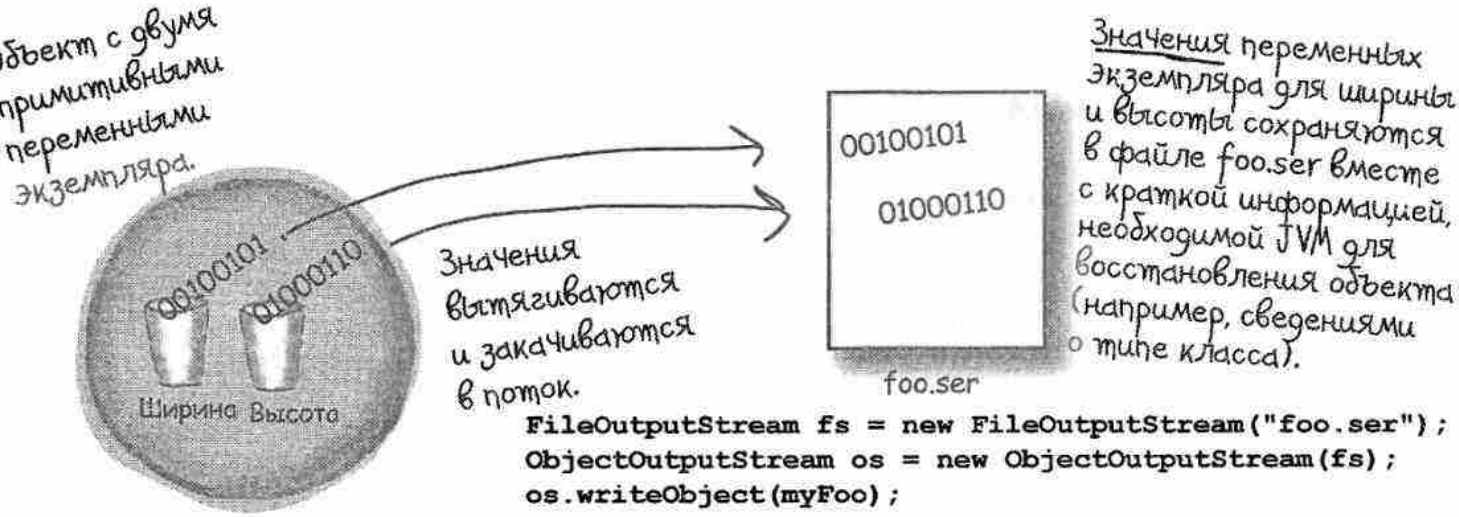


Объекты в куче обладают состоянием — значением переменных экземпляра. Эти значения делают один экземпляр класса отличным от других экземпляров того же класса.

## ❷ Сериализованный объект.



Сериализованные объекты **сохраняют значения переменных**, чтобы идентичный экземпляр (объект) можно было вернуть обратно в кучу.




```
Foo myFoo = new Foo();
myFoo.setWidth(37);
myFoo.setHeight(70);
```

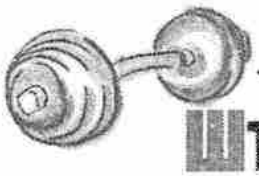
```
FileOutputStream fs = new FileOutputStream("foo.ser");
ObjectOutputStream os = new ObjectOutputStream(fs);
os.writeObject(myFoo);
```

Создает экземпляр FileOutputStream, который соединится с файлом, а затем прикрепится к ObjectOutputStream, чтобы записать объект.

# Но что конкретно представляет собой состояние объекта? Что нужно сохранять?

Вот теперь становится интереснее. Легко сохранять *примитивные* значения 37 и 70. Но что делать, если объект содержит переменную в виде *ссылки* на объект? А что насчет объекта, который включает пять переменных в виде ссылок на объекты? Как быть, если эти переменные хранят другие переменные?

Задумайтесь, какая часть объекта потенциально уникальна. Что необходимо восстановить, чтобы получить объект, идентичный сохраненному? Это, конечно, будет другая ячейка памяти, но не важно. Все, что важно, находится в куче; мы получим объект с таким же состоянием, с каким он был сохранен.



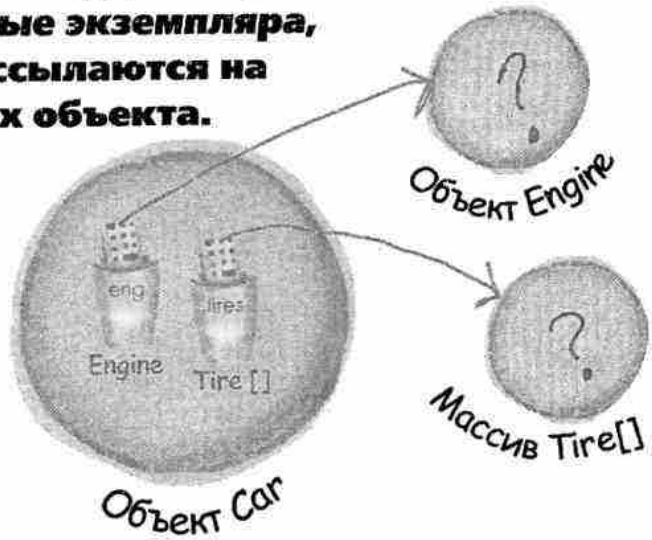
## Штанга для мозга

Что должно произойти с объектом Car при его сохранении таким образом, чтобы его можно было восстановить к оригинальному состоянию?

Подумайте, что и как вам нужно сохранить в объекте Car.

Что случится, если объект Engine содержит ссылку на Carburator?? И что находится внутри массива Tire[]?

**Объект Car содержит две *переменные экземпляра*, которые ссылаются на два других объекта.**

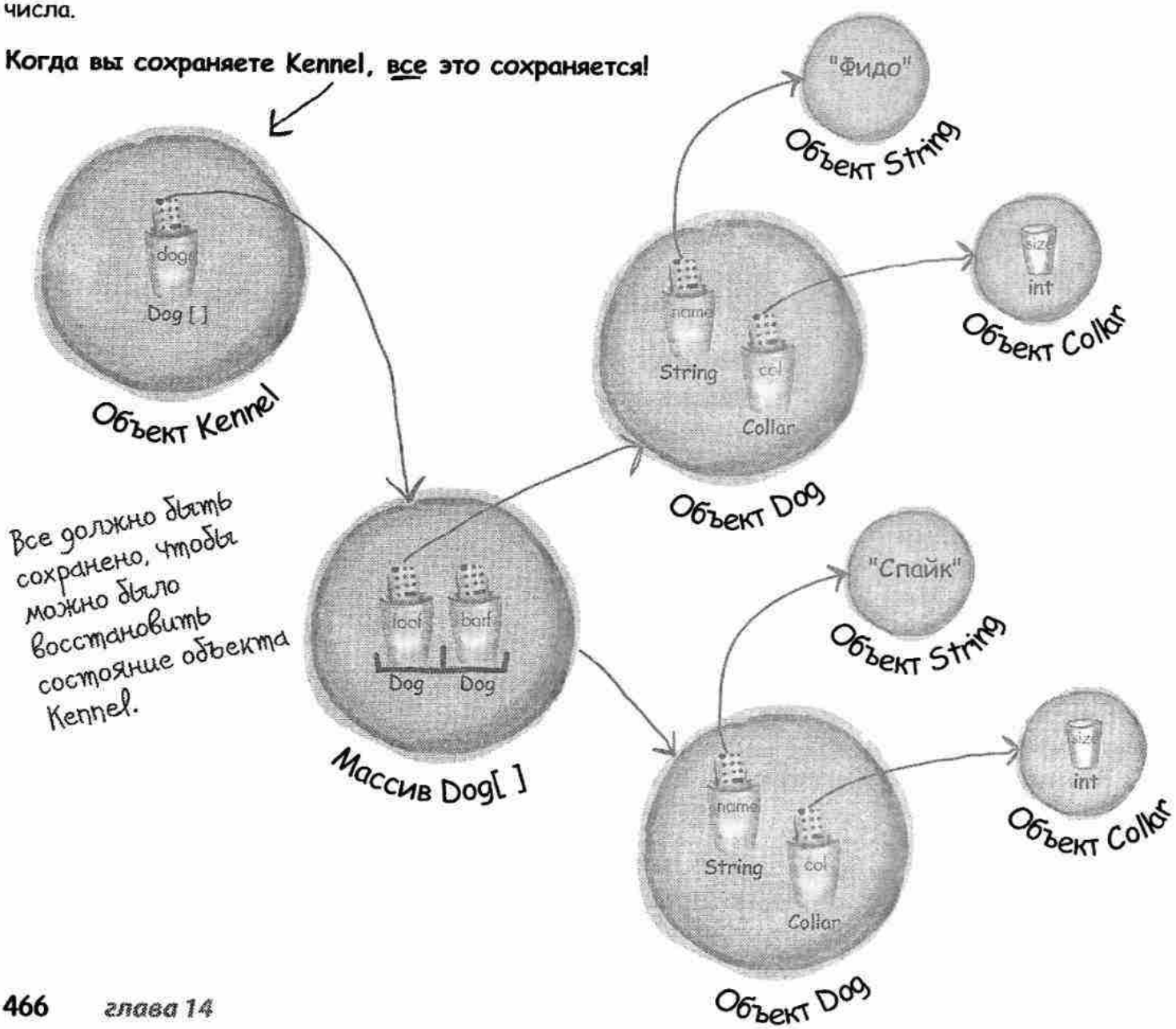



**Что потребуется для сохранения объекта Car?**

**При сериализации объекта также сериализуются все объекты, на которые он ссылается с помощью своих переменных экземпляра. И все объекты, на которые ссылаются те объекты, тоже сериализуются. И все объекты, на которые ссылаются те объекты, тоже сериализуются... Хорошо, что это происходит автоматически!**

*Сериализация сохраняет полный граф объекта. Переменные экземпляра ссылаются на все объекты, начиная с того, который сериализуется.*

Объект Kennel содержит ссылку на массив Dog[]. Массив содержит ссылки на два объекта Dog. Каждый объект Dog содержит ссылки на объекты String и Collar. Объекты String включают в себя набор символов, а объекты Collar — целые числа.



# Если вы хотите, чтобы ваш класс был сериализуемым, добавьте интерфейс Serializable

Интерфейс Serializable известен как *разметочный*, или *теговый*, потому что не содержит методов. Его единственная цель — объявить, что класс *сериализуемый (serializable)*. Другими словами, объекты данного типа сохраняются с применением механизма сериализации. Если какой-либо предок класса будет сериализуемым, то его потомки также автоматически станут таковыми, даже если явно не объявляют *реализацию Serializable*. Интерфейсы всегда так *работают*.

```
objectOutputStream.writeObject(myBox);
```

Что бы здесь ни находилось, оно должно реализовывать интерфейс Serializable, иначе при выполнении программы произойдет сбой.

---

```
import java.io.*;
public class Box implements Serializable {

    private int width;
    private int height;

    public void setWidth(int w) {
       width = w;
    }

    public void setHeight(int h) {
       height = h;
    }

public static void main (String[] args) {

      Box myBox = new Box();
      myBox.setWidth(50);
      myBox.setHeight(20);

      try {
         FileOutputStream fs = new FileOutputStream("foo.ser");
         ObjectOutputStream os = new ObjectOutputStream(fs);
         os.writeObject(myBox);
         os.close();
      } catch(Exception ex) {
          ex.printStackTrace();
      }
   }
}
```

Интерфейс Serializable находится в пакете java.io, поэтому его нужно импортировать.

Нет методов, которые необходимо реализовать, однако, когда вы пишете implements Serializable, вы говорите JVM: «Все в порядке, объекты этого типа можно сериализовывать».

Эти два значения будут сохранены.

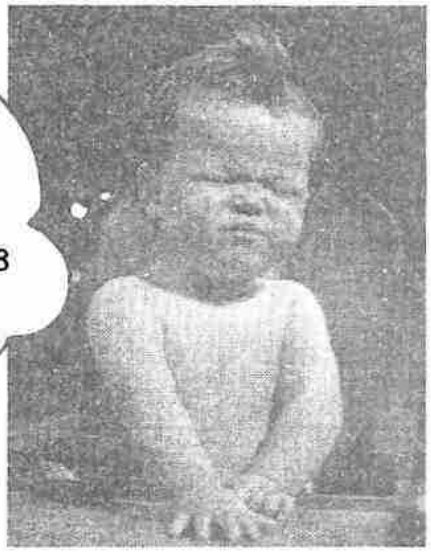
Операции ввода/вывода могут вызывать исключения.

Соединяемся с файлом с именем foo.ser, если он существует. В ином случае будет создан новый файл с именем foo.ser.

Связываем ObjectOutputStream с потоком соединения. Приказываем ему записать объект.

# Сериализация — это все или ничего.

**Можете ли вы представить, что случится, если какое-нибудь состояние объекта некорректно сохранится?**



Уууууу! Мне страшно уже при одной мысли об этом! А если собака вернется без веса? Или без ушей. Или ошейник вернется с размером 3 вместо 30. Это недопустимо!

**Либо полный граф объекта будет корректно сериализован, либо вся операция провалится.**

**Вы не можете сериализовать объект Pond, если переменная экземпляра Duck отказывается сериализовываться (из-за того, что не добавлен интерфейс Serializable).**

```
import java.io.*;

public class Pond implements Serializable {

    private Duck duck = new Duck();

    public static void main (String[] args) {
       Pond myPond = new Pond();
       try {
         FileOutputStream fs = new FileOutputStream("Pond.ser");
ObjectOutputStream os = new ObjectOutputStream(fs);
         os.writeObject(myPond);
         os.close();

       } catch(Exception ex) {
          ex.printStackTrace();
       }
    }
}
```

Объекты класса Pond могут сериализоваться.

Класс Pond содержит одну переменную экземпляра — Duck.

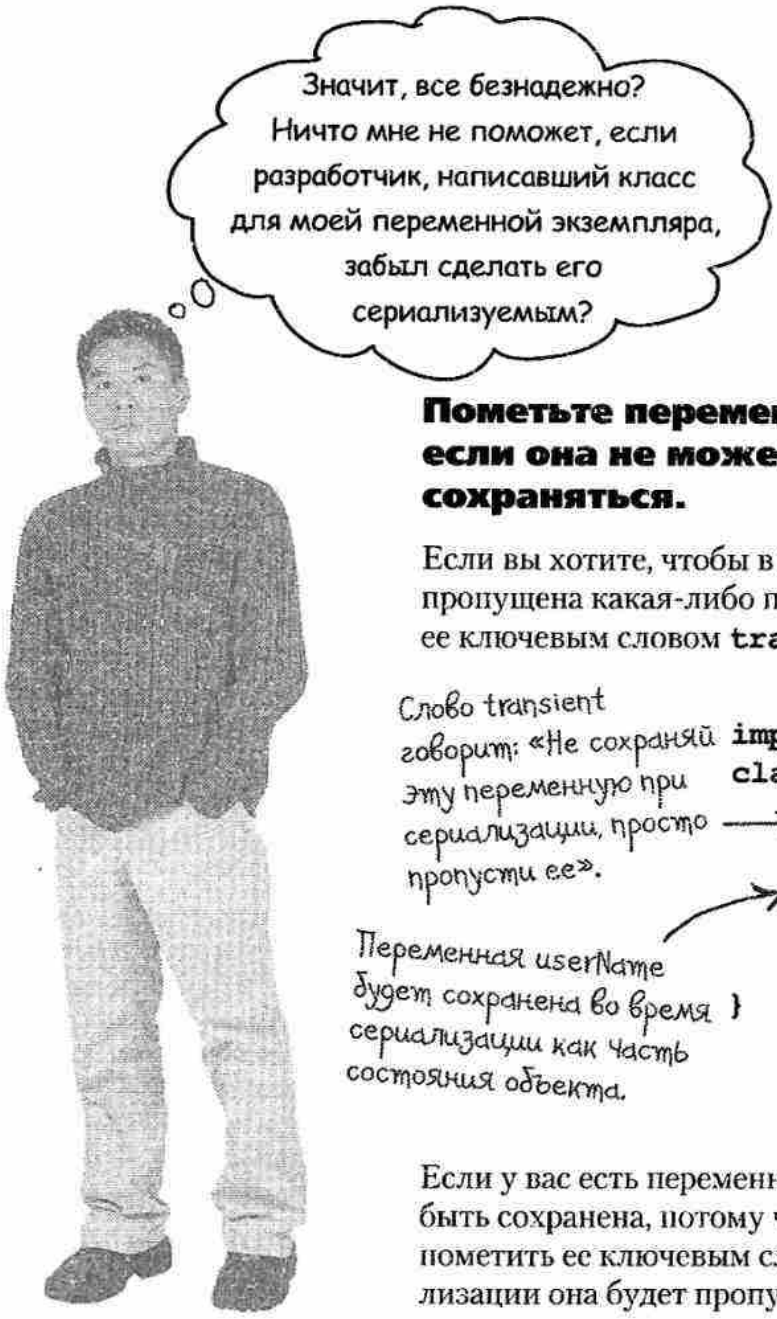
При сериализации myPond (объекта Pond) автоматически сериализуется его переменная Duck.

```
public class Duck {
   // Код утки размещается здесь
}
```

Ой! Утка не подлежит сериализации! Она не реализует интерфейс Serializable, поэтому, когда вы пытаетесь сериализовать объект Pond, происходит сбой, так как переменная экземпляра Duck объекта Pond не может сохраниться.

При попытке запустить метод main из класса Пруд получается следующее:

```
File Edit Window Help Regret
% java Pond
java.io.NotSerializableException: Duck
        at Pond.main(Pond.java:13)
```




## Пометьте переменную как transient, если она не может (или не должна) сохраняться.

Если вы хотите, чтобы в процессе сериализации была пропущена какая-либо переменная экземпляра, пометьте ее ключевым словом **transient** (переходный).

Слово transient говорит: «Не сохраняй эту переменную при сериализации, просто пропусти ее».

```
import java.net.*;
class Chat implements Serializable {
  transient String currentID;
  String userName;
  // Еще код
}
```

Переменная userName будет сохранена во время сериализации как часть состояния объекта.

Если у вас есть переменная экземпляра, которая не может быть сохранена, потому что не сериализуется, то можете пометить ее ключевым словом transient. В процессе сериализации она будет пропущена.

Но почему переменная будет несериализуема? Возможно, разработчик просто *забыл* реализовать в классе интерфейс Serializable. Или объект зависит от информации, которая доступна при выполнении программы и не может быть сохранена. Несмотря на то что большинство элементов в библиотеках классов Java сериализуемы, вы не можете сохранять их как сетевые соединения, потоки или файловые объекты. Они зависят от процесса выполнения программы и могут меняться при каждом запуске. Другими словами, их экземпляры уникальны для каждого запуска вашей программы, для платформы, на которой она выполняется, для определенной версии JVM. Как только программа закрывается, в понятном виде эти элементы уже не вернуть — каждый раз их нужно создавать с нуля.

## Это не глупые вопросы

**В: Если сериализация так важна, почему она не доступна по умолчанию для всех классов? Почему бы не реализовать интерфейс Serializable в классе Object, чтобы все его дочерние классы автоматически стали Serializable?**

О: Большинство классов реализуют Serializable, но у вас всегда есть выбор. Прежде чем «включить» сериализацию с помощью интерфейса Serializable, нужно принять сознательное решение для каждого разрабатываемого класса. Подумайте, как бы вы отключали сериализацию, если бы она была установлена по умолчанию. Интерфейсы указывают на функциональность, а не на ее *нехватку*, поэтому модель полиморфизма будет работать некорректно, если придется писать implements NonSerializable, чтобы сообщить о невозможности сохранения.

**В: Зачем создавать несериализуемый класс?**

О: Существует не так много причин. Например, у вас могут возникнуть проблемы с безопасностью, когда вы не захотите хранить объект пароля. Или у вас может быть объект, сохранять который не имеет смысла, потому что его ключевые переменные экземпляра несериализуемы, и нет никакой пользы от сериализации *вашего* класса.

**В: Если класс, который я использую, несериализуем, но для этого нет серьезной причины (за исключением того, что разработчик просто забыл), могу ли я наследовать «дефектный» класс и сделать его потомка сериализуемым?**

О: Да! Если сам класс может быть наследован, то вы можете сделать сериализуемый подкласс и просто подставить его в своем коде на место родительского класса. Полиморфизм позволяет так делать. Однако это вызывает другой интересный вопрос: что на самом деле *означает* ситуация, когда родительский класс несериализуем?

**В: Иначе говоря, какой смысл иметь сериализуемый подкласс несериализуемого родительского класса?**

О: Сначала нужно посмотреть, что происходит при десериализации класса (мы поговорим об этом через несколько страниц). При десериализации объекта, чей тип наследован от *несериализуемого класса*, конструктор этого родительского класса будет запускаться так, как если бы создавался новый объект такого типа. Хорошее решение — сделать дочерний класс сериализуемым, если нет причин для противоположной ситуации.

**В: Я только что понял нечто важное... Если я помечаю переменную как transient, то во время сериализации ее значение пропускается. Но что с ней тогда происходит? Я решаю проблему наличия несериализуемой переменной экземпляра с помощью ключевого слова transient, но разве она не нужна мне при восстановлении объекта? Получается, основная цель сериализации — сохранить состояние объекта, не так ли?**

О: Да, это проблема, но, к счастью, у нее есть решение. Если вы сериализуете объект, то ссылка на переменную, помеченную как transient, будет возвращена со значением *null* независимо от того, на что она указывала при сохранении. В результате весь граф объектов, соединенный с этой переменной, не сохранится. Очевидно, это вас не устроит, потому что вам, возможно, необходимо исходное значение данной переменной.

У вас есть два пути.

1. При восстановлении объекта инициализируйте переменную каким-нибудь значением по умолчанию. Это работает, если десериализованный объект не зависит от определенного значения такой переходной переменной. Рассмотрим на примере: может быть, у класса Dog есть объект Collar, но, возможно, все объекты Collar одинаковы, и не имеет значения, что вы даете возрожденному объекту Dog совершенно новый объект Collar; все равно никто не различит.

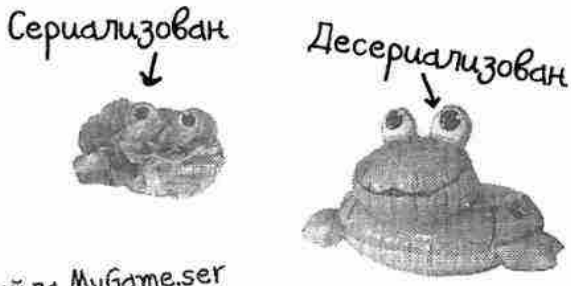
2. Если значение переходной переменной *действительно* важно (скажем, цвет и дизайн переходного Collar уникальны для каждого объекта Dog), то нужно сохранять ключевые значения Collar и использовать их при восстановлении Dog, чтобы сделать совершенно новый Collar идентичным оригиналу.

**В: Что происходит, если два объекта в графе оказываются одним и тем же объектом? Например, есть два разных объекта Cat в классе Kennel, но оба имеют ссылку на один и тот же объект Owner. Сохраняется ли в этом случае Owner дважды? Я надеюсь, что нет.**

О: Превосходный вопрос! Сериализация достаточно разумна и знает, когда два объекта в графе представляют собой одно и то же. В этом случае сохраняется только один объект и при десериализации восстанавливаются любые ссылки на него.

# Десериализация: восстановление объекта

Смысл сериализации объекта заключается в возможности его восстановления обратно к оригинальному состоянию, уже при другом «запуске» JVM (которая может быть даже не той же самой JVM, запущенной при сериализации объекта). Десериализация во многом похожа на сериализацию в обратном порядке.




**1 Создаем объект FileInputStream.**

Если файла MyGame.ser не существует, то будет выброшено исключение.

```
FileInputStream fileStream = new FileInputStream("MyGame.ser");
```

Создаем объект FileInputStream. Он знает, как соединиться с существующим файлом.

**2 Создаем ObjectInputStream.**

```
ObjectInputStream os = new ObjectInputStream(fileStream);
```

ObjectInputStream позволяет прочитать объекты, но не может напрямую соединиться с файлом. Ему нужно подключиться к потоку соединения, в данном случае к FileInputStream.

**3 Читаем объекты.**

```
Object one = os.readObject();
Object two = os.readObject();
Object three = os.readObject();
```

Каждый раз, когда мы вызываем метод readObject(), мы получаем следующий объект в потоке. В итоге мы будем прочитывать их в том же порядке, в котором они были записаны. Мы получим большое исключение, если попытаемся прочитать больше объектов, чем было записано.

**4 Приводим объекты.**

```
GameCharacter elf = (GameCharacter) one;
GameCharacter troll = (GameCharacter) two;
GameCharacter magician = (GameCharacter) three;
```

Возвращаемое значение метода readObject() имеет тип Object (как и в случае с ArrayList), так что нужно привести его обратно к типу, которому оно действительно принадлежит.

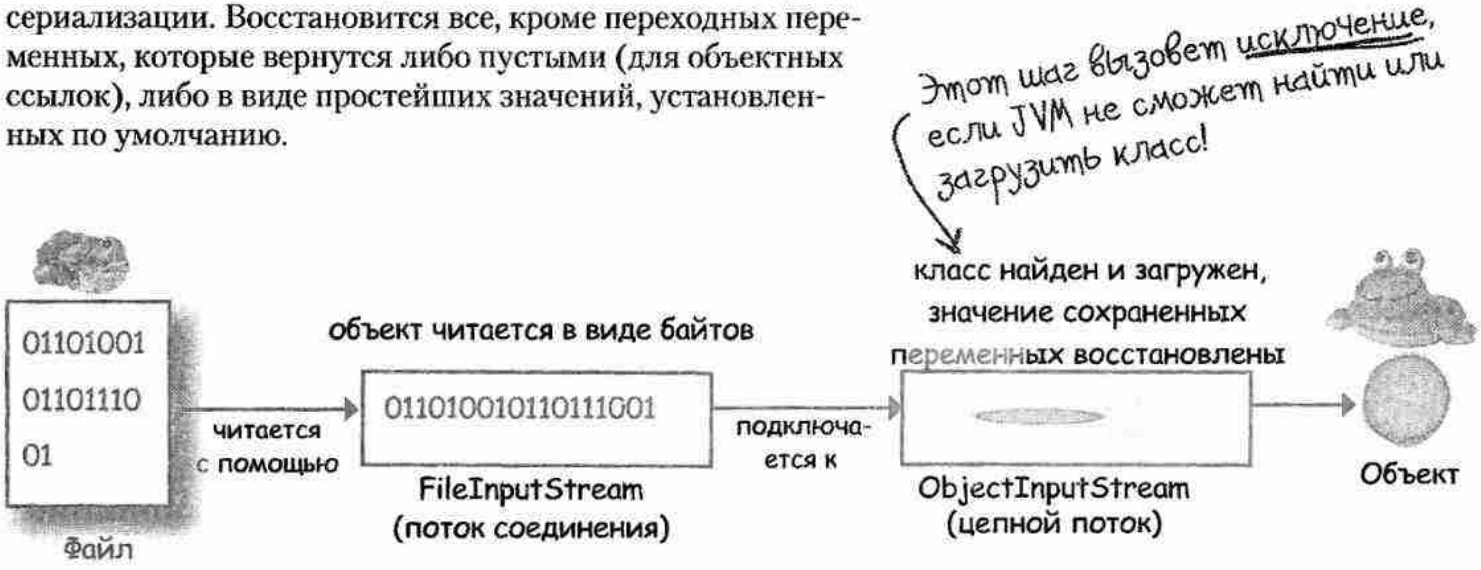
**5 Закрываем ObjectInputStream.**

```
os.close();
```

Закрывая верхний поток, мы закрываем и находящиеся ниже, так что FileInputStream (и файл) закроются автоматически.

# Что происходит при десериализации

При десериализации JVM пытается восстановить объект, создавая его новый экземпляр в куче. Он будет иметь такое же состояние, как и сериализованный объект во время сериализации. Восстановится все, кроме переходных переменных, которые вернутся либо пустыми (для объектных ссылок), либо в виде простейших значений, установленных по умолчанию.




1. Объект **читается** из потока.

2. JVM определяет (по информации, сохраненной вместе с сериализованным объектом) **тип класса** объекта.

3. JVM пытается **найти и загрузить** класс объекта. Если она не может найти и/или загрузить класс, то вызывает исключение и десериализация завершается неудачей.

4. Для нового объекта выделяется место в куче, но **конструктор сериализованного объекта не запускается**! Очевидно, если бы конструктор запускался, то это восстановило бы состояние объекта к его оригинальному «новому» состоянию, но это нам не нужно. Требуется, чтобы объект восстановился к состоянию, которое имел *при сериализации*, а не при первом создании.

5. Если у объекта есть несериализуемый класс где-нибудь на вершине его иерархии наследования, то **конструктор этого класса будет запущен** вместе с любыми конструкторами выше него (даже если они будут сериализуемыми). Как только начнется соединение конструкторов, вы уже не сможете остановить его. Это значит, что все родительские классы, начиная с первого несериализуемого, будут заново инициализироваться.

6. **Переменным** объекта **присваивают значения из сериализованного состояния**. Переходным переменным присваивают пустое значение (для ссылок на объекты) или значение по умолчанию (0, false и т. д.), если речь идет о простых типах.

## Это не глупые вопросы

**В: Почему класс не сохраняется как часть объекта? В этом случае не было бы проблем с поиском класса.**

О: Конечно, можно реализовать сериализацию именно так. Но это чревато излишними накладными расходами. Такой подход не вызывает трудностей при записи объектов в файл на локальном диске, но сериализация используется и для отсылки объектов через сетевые соединения. Если класс связан с каждым сериализованным (отправляемым) объектом, то пропускная способность становится гораздо большей проблемой.

Однако для объектов, сериализуемых для отправки по сети, *разработан* механизм, при котором сериализованный объект может «маркироваться» адресом URL, чтобы можно было найти его класс. Такой механизм применяется в программном интерфейсе вызова удаленных методов Java (Remote Method Invocation, RMI). Вы можете отсылать сериализованные объекты как часть, скажем, аргумента метода. Если JVM получает вызов, который не содержит класса, она может использовать URL для получения и загрузки класса по сети, и все это происходит автоматически. Мы поговорим об RMI в главе 17.

**В: А что вы скажете о статических переменных? Они сериализуются?**

О: Нет. Помните, что «статичный» означает «один на класс», а не «один на объект». Статические переменные не сохраняются, и, когда объект десериализуется, у него остаются все статические переменные, которые *в это время* содержит его класс. Мораль: не делайте сериализуемые объекты зависимыми от динамически изменяющейся статической переменной! При возвращении объекта она может стать другой.

# Сохранение и восстановление игровых персонажей

```
import java.io.*;

public class GameSaverTest {
   public static void main(String[] args) {
     GameCharacter one = new GameCharacter(50, "Эльф", new String[] {"лук", "меч", "кастет"});
     GameCharacter two = new GameCharacter(200, "Тролль", new String[] {"голые руки", "большой топор"});
     GameCharacter three = new GameCharacter(120, "Маг", new String[] {"заклинания", "невидимость"});

     // Представьте себе код, который может изменить значения состояния персонажей

     try {
        ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("Game.ser"));
        os.writeObject(one);
        os.writeObject(two);
        os.writeObject(three);
        os.close();
     } catch(IOException ex) {
        ex.printStackTrace();
     }
     one = null;
     two = null;
     three = null;

     try {
       ObjectInputStream is = new ObjectInputStream(new FileInputStream("Game.ser"));
        GameCharacter oneRestore = (GameCharacter) is.readObject();
        GameCharacter twoRestore = (GameCharacter) is.readObject();
        GameCharacter threeRestore = (GameCharacter) is.readObject();

        System.out.println("Тип первого:" + oneRestore.getType());
        System.out.println("Тип второго:" + twoRestore.getType());
        System.out.println("Тип третьего:" + threeRestore.getType());
     } catch(Exception ex) {
        ex.printStackTrace();
     }
   }
}
```

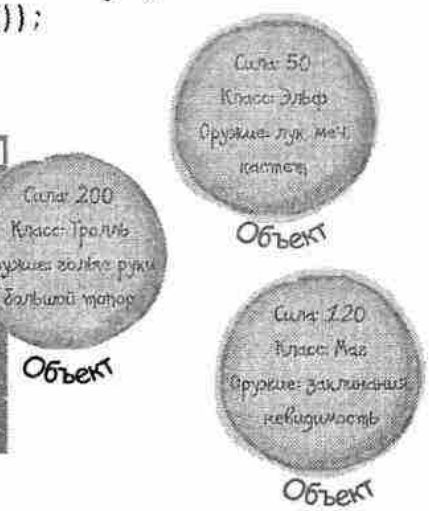
Создаем несколько персонажей...

Присвоим им значения null, чтобы мы не смогли обратиться к объектам в «куче».

Теперь прочитаем их из файла...

Сделаем проверку, чтобы убедиться, что это сработало.

```
File Edit Window Help Resuscitate
% java GameSaver
Elf
Troll
Magician
```



## Класс GameCharacter

```
import java.io.*;

public class GameCharacter implements Serializable {
   int power;
   String type;
   String[] weapons;

   public GameCharacter(int p, String t, String[] w) {
       power = p;
       type = t;
       weapons = w;
   }

   public int getPower() {
     return power;
   }

   public String getType() {
      return type;
   }

   public String getWeapons() {
       String weaponList = "";

       for (int i = 0; i < weapons.length; i++) {
          weaponList += weapons[i] + " ";
       }
       return weaponList;
   }
}
```

Это стандартный класс, предназначенный для тестирования сериализации. У нас вообще-то нет такой игры, но мы оставим вам класс для экспериментирования.

# Сериализация объектов

## КЛЮЧЕВЫЕ МОМЕНТЫ

- Вы можете сохранить состояние объекта, сериализовав его.
- Чтобы сериализовать объект, вам понадобится поток ObjectOutputStream (из пакета java.io).
- Потоки делятся на потоки для соединения и цепные потоки.
- Потоки для соединения могут представлять собой подключение к источнику или пункту назначения. Обычно это файл, соединение через сетевой сокет или консоль.
- Цепные потоки не могут подключиться к источнику или пункту назначения и должны связываться с потоком для соединения (или с другим потоком).
- Чтобы сериализовать объект в файл, создайте поток FileOutputStream и соедините его с ObjectOutputStream.
- Чтобы сериализовать объект, вызовите метод *writeObject(theObject)* из потока ObjectOutputStream. Не нужно вызывать методы из FileOutputStream.
- Чтобы стать сериализованным, объект должен поддерживать интерфейс Serializable. Если один из предков класса реализует Serializable, то подкласс автоматически будет сериализуемым, даже если конкретно не объявит *реализацию Serializable*.
- При сериализации объекта сериализуется весь его граф. Это означает, что любые объекты, на которые ссылаются его переменные экземпляра, тоже сериализуются. Кроме того, сериализуются объекты, на которые ссылаются те объекты, и т. д.
- Если объект в графе будет несериализуемым, это вызовет исключение при выполнении программы, когда переменная, ссылающаяся на данный объект, не будет пропущена.
- Отметьте переменную экземпляра ключевым словом transient, если хотите, чтобы при сериализации она была пропущена. Переменная будет восстановлена как пустая (для ссылок на объекты) или со значением по умолчанию (для простых типов).
- Во время десериализации класс всех объектов в графе должен быть доступным для JVM.
- Вы считываете объекты (используя метод readObject ()) в том порядке, в котором они первоначально были записаны.
- Значение, возвращаемое методом readObject(), имеет тип Object, поэтому десериализованные объекты должны быть приведены к их реальным типам.
- Статические переменные не сериализуются! Нет смысла сохранять значение статической переменной как часть состояния объекта, так как все объекты данного типа используют только значение, находящееся в классе.

# Запись строки в текстовый файл

Сохранение объектов с помощью сериализации — самый легкий способ сохранения и восстановления данных Java-программы между запусками. Но иногда нужно сохранять информацию в простой текстовый файл. Представьте, что ваша программа должна записать данные в простой текстовый файл, который нужно прочитать другой программе (возможно, написанной не на Java). Например, у вас есть сервлет — исполняемая вашим веб-сервером программа на языке Java. Сервлет принимает из формы данные, введенные пользователем в браузере, и записывает их в текстовый файл, который кто-то еще загружает в электронную таблицу для анализа.

Запись текстовых данных (фактически строки) похожа на запись объекта, если не считать того, что вы записываете строку вместо объекта и используете FileWriter вместо FileOutputStream (и не привязываете его к ObjectOutputStream).

Так будут выглядеть данные игровых персонажей, если вы запишете их в виде читаемого текстового файла.

```
50,Эльф,лук,меч,кастет
200,Тролль,голые руки,
большой топор
120,Маг,заклинания,
невидимость
```

### Для записи сериализованного объекта:

```
objectOutputStream.writeObject(someObject);
```

### Для записи строки:

```
fileWriter.write("Моя первая строка для сохранения");
```

```
import java.io.*;

class WriteAFile {
    public static void main (String[] args) {

        try {
            FileWriter writer = new FileWriter("Foo.txt");

            writer.write("Привет, Фу!");

            writer.close();
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Для FileWriter понадобится пакет java.io!

Если файла Foo.txt не существует, FileWriter создаст его.

Методы для ввода/вывода нужно использовать внутри операторов try/catch. Что угодно может вызвать исключение ввода/вывода!!

Метод write() принимает строку!

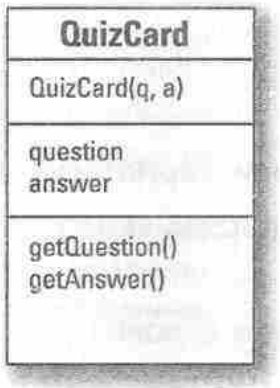
Нужно закрыть его по завершении записи!

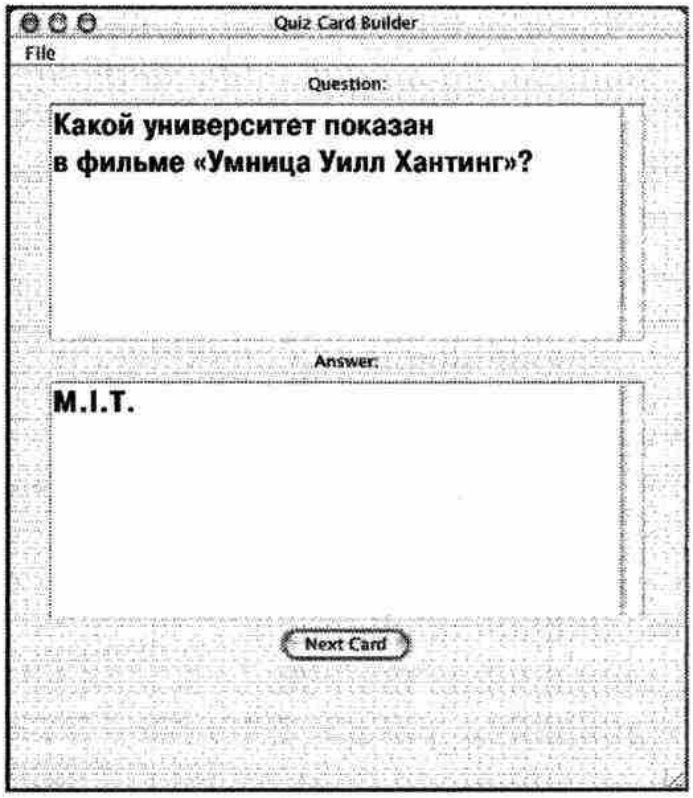
# Пример текстового файла: электронные флеш-карты

Помните карточки с текстом, которыми вы пользовались в школе? Там вопрос был написан на одной стороне, а ответ — на другой. Не стоит ждать от них помощи, когда нужно что-то понять, но ничто не превзойдет их при зубрежке и механическом запоминании. Кроме того, они прекрасно подходят для мелких игр.

**Сделаем электронную версию, в которой будет три класса.**

1) ***QuizCardBuilder*** — простой инструмент для создания и сохранения электронных флеш-карт.
2) ***QuizCardPlayer*** — игровой движок, который будет загружать набор флеш-карт и разыгрывать их для пользователя.
3) ***QuizCard*** — простой класс, разделяющий данные по карточкам. Мы пройдемся по коду для классов Builder и Player, а класс QuizCard вам придется сделать самостоятельно, используя это:




**QuizCardBuilder**

Содержит меню File (Файл) с командой Save (Сохранить) для сохранения текущего набора карточек в текстовый файл.

**QuizCardPlayer**

Включает меню File (Файл) с командой Load (Загрузить) для загрузки набора карточек из текстового файла.

# Quiz Card Builder: структура кода

```
public class QuizCardBuilder {

  public void go() {
    // Формируем и выводим на экран GUI
  }

                    Внутренний класс
  private class NextCardListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // Добавляем текущую карточку в список и очищаем текстовые области
    }
  }

                    Внутренний класс
  private class SaveMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // Вызываем диалоговое окно,
      // позволяющее пользователю называть и сохранять набор
    }
  }

                    Внутренний класс
  private class NewMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // Очищаем список карточек и текстовые области
    }
  }


  private void saveFile(File file) {
    // Проходим по списку карточек и записываем
    // каждый элемент в текстовый файл, который потом
    // можно будет прочитать
    // (то есть с чистыми разделителями между частями)

  }
}
```

Формируем и выводим на экран GUI, включая создание и регистрацию слушателей для событий.

Срабатывает при нажатии пользователем кнопки Next Card (Следующая карточка).

Запускается при выборе команды Save (Сохранить) из меню File (Файл). Означает, что пользователь хочет сохранить все карточки в текущем списке в виде набора (например, набор карточек о квантовой механике, небольшие факты про Голливуд, правила языка Java и т. д.).

Запускается при выборе команды New (Создать) из меню File (Файл). Означает, что пользователь хочет создать новый набор (очистив список карточек и текстовые области).

Вызывается классом SaveMenuListener. Непосредственно записывает данные в файл.

## Код Quiz Card Builder

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class QuizCardBuilder {

    private JTextArea question;
    private JTextArea answer;
    private ArrayList<QuizCard> cardList;
    private JFrame frame;


    public static void main (String[] args) {
       QuizCardBuilder builder = new QuizCardBuilder();
       builder.go();
    }


    public void go() {
        // Формируем GUI

        frame = new JFrame("Quiz Card Builder");
        JPanel mainPanel = new JPanel();
        Font bigFont = new Font("sanserif", Font.BOLD, 24);
        question = new JTextArea(6,20);
        question.setLineWrap(true);
        question.setWrapStyleWord(true);
        question.setFont(bigFont);

        JScrollPane qScroller = new JScrollPane(question);
        qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        answer = new JTextArea(6,20);
        answer.setLineWrap(true);
        answer.setWrapStyleWord(true);
        answer.setFont(bigFont);

        JScrollPane aScroller = new JScrollPane(answer);
        aScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        aScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);


        JButton nextButton = new JButton("Next Card");

        cardList = new ArrayList<QuizCard>();

        JLabel qLabel = new JLabel("Question:");
        JLabel aLabel = new JLabel("Answer:");

        mainPanel.add(qLabel);
        mainPanel.add(qScroller);
        mainPanel.add(aLabel);
        mainPanel.add(aScroller);
        mainPanel.add(nextButton);
        nextButton.addActionListener(new NextCardListener());
        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");
        JMenuItem newMenuItem = new JMenuItem("New");
```

Весь этот код нужен для построения GUI. В коде нет ничего особенного, хотя, возможно, вы захотите взглянуть на фрагменты в классах MenuBar, Menu и MenuItems.

```
        JMenuItem saveMenuItem = new JMenuItem("Save");
        newMenuItem.addActionListener(new NewMenuListener());

        saveMenuItem.addActionListener(new SaveMenuListener());
        fileMenu.add(newMenuItem);
        fileMenu.add(saveMenuItem);
        menuBar.add(fileMenu);
        frame.setJMenuBar(menuBar);
        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        frame.setSize(500,600);
        frame.setVisible(true);
    }

    public class NextCardListener implements ActionListener {
        public void actionPerformed(ActionEvent ev) {

            QuizCard card = new QuizCard(question.getText(), answer.getText());
            cardList.add(card);
            clearCard();
        }
    }

    public class SaveMenuListener implements ActionListener {
        public void actionPerformed(ActionEvent ev) {
            QuizCard card = new QuizCard(question.getText(), answer.getText());
            cardList.add(card);

            JFileChooser fileSave = new JFileChooser();
            fileSave.showSaveDialog(frame);
            saveFile(fileSave.getSelectedFile());
        }
    }

    public class NewMenuListener implements ActionListener {
        public void actionPerformed(ActionEvent ev) {
            cardList.clear();
            clearCard();
        }
    }

    private void clearCard() {
        question.setText("");
        answer.setText("");
        question.requestFocus();
    }

    private void saveFile(File file) {
      try {
        BufferedWriter writer = new BufferedWriter(new FileWriter(file));

        for(QuizCard card:cardList) {
            writer.write(card.getQuestion() + "/");
            writer.write(card.getAnswer() + "\n");
        }
       writer.close();

      } catch(IOException ex) {
        System.out.println("couldn't write the cardList out");
        ex.printStackTrace();
      }
    }
}
```

Мы создаем объект JMenuBar и добавляем в него меню File (Файл) с пунктами New (Создать) и Save (Сохранить). Затем говорим главной панели, что она должна использовать JMenuBar. Пункты меню могут запускать ActionEvent.

Вызывается диалоговое окно, и программа останавливается на этой строке, пока пользователь не выберет меню Save (Сохранить). Всю навигацию, выбор файла и т. д. за вас выполняет класс JFileChooser! Это действительно просто.

Метод, который непосредственно записывает файл (вызывается обработчиком событий класса SaveMenuListener). Аргумент — это объект File, который сохраняется пользователем. Мы рассмотрим класс File на следующей странице.

Мы соединяем BufferedWriter с новым FileWriter для более эффективной записи (поговорим об этом через несколько страниц).

Пробегаем через ArrayList с карточками и записываем их по одной на строку, разделяя вопрос и ответ символом «/» и в конце добавляя символ новой строки «\n».

# Класс java.io.File

Класс java.io.File *представляет* файл на диске, но фактически не представляет *содержимое* этого файла. Думайте об объекте File как о *пути* к файлу (или даже *каталоге*), а не как о самом файле. У класса File, например, нет методов для чтения и записи. Одна очень полезная особенность объекта File состоит в том, что он предлагает более безопасный способ представить файл, чем простое указание имени файла в виде строки. Например, большинство классов, принимающих строковое имя файла в свой конструктор (например, FileWriter или FileInputStream), могут вместо этого взять объект File. Вы можете создать его, проверить, что путь работает, и затем предоставить объект File классам FileWriter или FileInputStream.

**Объект File хранит имя или путь файла или каталога на диске, например: /Users/Kathy/Data/GameFile.txt**

**Но он не предоставляет доступ к данным в файле!**

Адрес дома — это не сам дом! Объект File представляет собой что-то вроде уличного адреса. Он содержит имя и местоположение определенного файла, но не является самим файлом.

## Что можно сделать с объектом File

① **Создать объект File, представляющий существующий файл:**

```
File f = new File("MyCode.txt");
```

② **Создать новый каталог:**

```
File dir = new File("Chapter7");
dir.mkdir();
```

③ **Вывести содержимое каталога:**

```
if (dir.isDirectory()) {
      String[] dirContents = dir.list();
      for (int i = 0; i < dirContents.length; i++) {
          System.out.println(dirContents[i]);
      }
}
```

④ **Получить абсолютный путь файла или каталога:**

```
System.out.println(dir.getAbsolutePath());
```

⑤ **Удалить файл или каталог (при выполнении возвращает значение true):**

```
boolean isDeleted = f.delete();
```

Объект File представляет имя файла GameFile.txt

**GameFile.txt**

50,Эльф,лук,меч,кастет
200,Тролль,голые руки, большой топор
120,Маг,заклинания, невидимость

Объект File не содержит данные, которые расположены внутри файла, и не предоставляет к ним доступ!

# В чем прелесть буфера

**Не пользоваться буфером — все равно что покупать товары без тележки. Вам пришлось бы тащить каждую вещь к своей машине по очереди: всего одну банку с супом или один рулон туалетной бумаги за один раз.**

Буфер дает временное хранилище для объектов, пока держатель (как тележка) не заполнится. Благодаря буферу вам придется делать намного меньше рейсов.

```
BufferedWriter writer = new BufferedWriter(new FileWriter(aFile));
```

Заметьте, что вам даже не нужно хранить ссылку на объект FileWriter. Единственный важный объект — BufferedWriter, методы которого мы вызываем. При закрытии он позаботится обо всей цепочке.

*Гораздо* эффективнее работать с буфером, чем без него. Вы можете заполнить файл с помощью метода write(someString), используя только FileWriter, но объекту придется записывать отдельно каждый элемент, который вы передаете в файл. Это ненужные издержки, так как каждое обращение к диску — важное событие по сравнению с манипуляцией данными в памяти. После присоединения BufferedWriter к FileWriter первый будет хранить все записанные вами элементы, пока не заполнится. *FileWriter начнет запись в файл на диск, когда буфер заполнится.*

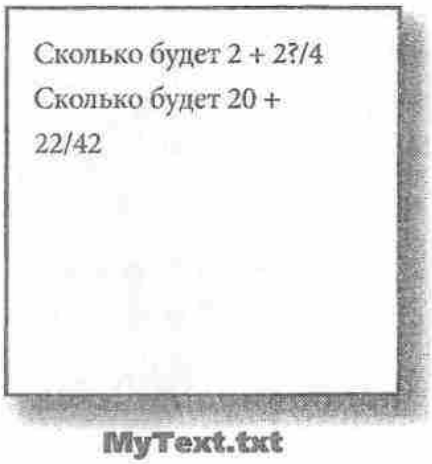
Если же вы хотите отослать данные до заполнения буфера, то можете это сделать. ***Нужно просто сбросить его.*** Вызов метода writer.flush() говорит: «***Сейчас же*** отошли все, что находится в буфере!»

# Чтение из текстового файла

В чтении текста из файла нет ничего сложного, но на этот раз мы будем использовать объект File для представления файла, FileReader — для фактического чтения и BufferedReader — для большей эффективности всего процесса.

При чтении строки считываются в цикле *while*. Цикл заканчивается, если метод readLine() возвращает null. Это наиболее распространенный способ чтения данных, не являющихся сериализованными объектами.

Файл с двумя строками текста.




Не забудьте импортировать пакет

```
import java.io.*;

class ReadAFile {
    public static void main (String[] args) {

        try {
           File myFile = new File("MyText.txt");
           FileReader fileReader = new FileReader(myFile);

           BufferedReader reader = new BufferedReader(fileReader);

           String line = null;

           while ((line = reader.readLine()) != null) {
              System.out.println(line);
           }
           reader.close();

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

FileReader — поток соединения для символов, который подключается к текстовому файлу.

Для более эффективного чтения соединим FileReader с BufferedReader. Тогда FileReader будет обращаться к файлу только в том случае, если буфер будет пуст (потому что программа прочла все, что там находилось).

Создаем строковую переменную для временного хранения каждой строки в процессе чтения.

Этот код говорит: «Прочитай строку текста и присвой ее строковой переменной "line". Пока эта переменная не пуста (так как там было что прочитать), выводи на экран только что прочтенную строку».

Существует еще один способ выразить это: «Пока здесь есть строки для чтения, считывай их и выводи их на экран».

# Quiz Card Player: структура кода

```
public class QuizCardPlayer {

  public void go() {
    // Сформируем и выведем на экран GUI
  }

  class NextCardListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // Если это вопрос, показываем ответ, иначе показываем следующий вопрос
      // Установим флаг для того, что мы видим, — вопрос это или ответ
    }
  }

  class OpenMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // Вызываем диалоговое окно, позволяющее пользователю выбирать,
      // какой набор карточек открыть
    }
  }

  private void loadFile(File file) {
      // Нужно создать ArrayList с карточками, считывая их из текстового файла,
      // вызванного из обработчика событий класса OpenMenuListener,
      // прочитать файл по одной строке за один раз
      // и вызвать метод makeCard() для создания новой карточки из строки
      // (одна строка в файле содержит вопрос и ответ, разделенные символом /)
  }


   private void makeCard(String lineToParse) {
     // Вызывается методом loadFile, берет строку из текстового файла,
     // делит ее на две части — вопрос и ответ — и создает новый объект QuizCard,
     // а затем добавляет его в ArrayList с помощью CardList

   }
}
```

## Код Quiz Card Builder

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class QuizCardPlayer {

    private JTextArea display;
    private JTextArea answer;
    private ArrayList<QuizCard> cardList;
    private QuizCard currentCard;
    private int currentCardIndex;
    private JFrame frame;
    private JButton nextButton;
    private boolean isShowAnswer;

    public static void main (String[] args) {
       QuizCardPlayer reader = new QuizCardPlayer();
       reader.go();
    }

    public void go() {

        // Формируем gui

        frame = new JFrame("Quiz Card Player");
        JPanel mainPanel = new JPanel();
        Font bigFont = new Font("sanserif", Font.BOLD, 24);

        display = new JTextArea(10,20);
        display.setFont(bigFont);

        display.setLineWrap(true);
        display.setEditable(false);

        JScrollPane qScroller = new JScrollPane(display);
        qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
        nextButton = new JButton("Show Question");
        mainPanel.add(qScroller);
        mainPanel.add(nextButton);
        nextButton.addActionListener(new NextCardListener());

        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");
        JMenuItem loadMenuItem = new JMenuItem("Load card set");
        loadMenuItem.addActionListener(new OpenMenuListener());
        fileMenu.add(loadMenuItem);
        menuBar.add(fileMenu);
        frame.setJMenuBar(menuBar);
        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        frame.setSize(640,500);
        frame.setVisible(true);

    } // Закрыть метод go
```

На этой странице размещен код для создания GUI; ничего особенного.

```
public class NextCardListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
       if (isShowAnswer) {
          // Показываем ответ, так как вопрос уже был увиден
          display.setText(currentCard.getAnswer());
          nextButton.setText("Next Card");
          isShowAnswer = false;
       } else {
           // Показываем следующий вопрос
          if (currentCardIndex < cardList.size()) {

             showNextCard();

          } else {
             // Больше карточек нет!
             display.setText("That was last card");
             nextButton.setEnabled(false);
          }
       }
    }
 }

public class OpenMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
        JFileChooser fileOpen = new JFileChooser();
        fileOpen.showOpenDialog(frame);
        loadFile(fileOpen.getSelectedFile());
    }
 }

private void loadFile(File file) {

   cardList = new ArrayList<QuizCard>();
   try {
       BufferedReader reader = new BufferedReader(new FileReader(file));
       String line = null;
       while ((line = reader.readLine()) != null) {
          makeCard(line);
       }
       reader.close();

   } catch(Exception ex) {
       System.out.println("couldn't read the card file");
       ex.printStackTrace();
   }

  // Пришло время показать первую карточку
  showNextCard();
}

private void makeCard(String lineToParse) {
   String[] result = lineToParse.split("/");
   QuizCard card = new QuizCard(result[0], result[1]);
   cardList.add(card);
   System.out.println("made a card");
}

private void showNextCard() {
     currentCard = cardList.get(currentCardIndex);
     currentCardIndex++;
     display.setText(currentCard.getQuestion());
     nextButton.setText("Show Answer");
     isShowAnswer = true;
}
} // Закрываем class
```

Проверяем значение флага isShowAnswer, чтобы узнать, что сейчас отображается — вопрос или ответ, и в зависимости от результата выполняем соответствующие действия.

Вызываем файловое окно, позволяющее выбирать файл для открытия.

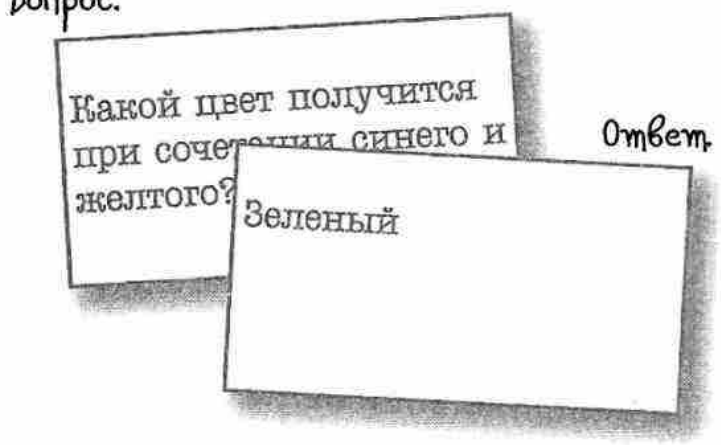
Создаем BufferedReader, связанный с новым FileReader. Предоставляем объекту FileReader объект File, выбранный пользователем в окне открытия файла.

Читаем по одной строке за один раз, передавая результат в метод makeCard(), который разделяет его и преобразует в настоящий объект QuizCard, а затем добавляет в ArrayList.

Каждая строка текста соответствует одной флешкарте, но нам нужно разделить вопрос и ответ. Для этого мы используем метод split() из класса String, который разбивает строку на две лексемы (одна для вопроса и одна для ответа). Мы рассмотрим метод split() на следующей странице.

# Разбор текста с помощью метода split() из класса String

**Представьте, что у вас есть такая карточка:**




**Она сохранена в файле таким образом:**

```
Какой цвет получится
при сочетании синего
и желтого?/Зеленый
Какой цвет получится
при сочетании красного
и синего?/Пурпурный
```

**Как вы разделите вопрос и ответ?**

При чтении файла вопрос и ответ соединены в одну строку и отделены друг от друга символом / (потому что так мы записывали файл в коде класса QuizCardBuilder).

**Метод split() позволяет разбивать строку на части.**

Метод split() говорит: «Дайте мне разделитель, и я разобью для вас строку на части, а затем внесу результат в строковый массив».

Лексема 1 Разделитель Лексема 2

Так в приложении QuizCardPlayer выглядит одиночная строка при ее считывании из файла.

```
String toTest = "Какой цвет получится при сочетании синего и желтого?/Зеленый";
String[] result = toTest.split("/");
for (String token:result) {
   System.out.println(token);
}
```

Метод split() берет символ / и использует его для разбиения строки (в данном случае на две части). Заметьте: split() — мощное средство, и здесь мы задействуем его не в полную силу. Он может выполнять очень сложный разбор строк, используя фильтры, универсальные символы и т. д.

Проходим в цикле по массиву и выводим на экран каждую лексему. В данном примере их только две: «Какой цвет получится при сочетании синего и желтого?» и «Зеленый».

## Это не глупые вопросы

**В: Я заглянул в API и нашел там около пяти миллионов классов в пакете java.io. Откуда вы знаете, какой из них применять?**

О: API ввода/вывода использует модульную связующую концепцию, поэтому разрешается связывать вместе потоки соединения и цепные потоки (их еще называют фильтрующими) во множестве комбинаций, чтобы получить все, что может понадобиться.

Цепочки не ограничиваются двумя уровнями; вы можете привязать множество цепных потоков друг к другу, чтобы получить нужное количество связующих звеньев.

Однако чаще всего вы будете использовать одну и ту же небольшую группу классов. Если вы записываете текстовые файлы, то, возможно, вам понадобятся лишь BufferedReader и BufferedWriter (связанные с FileReader и FileWriter). Если вы записываете сериализованные объекты, то можете использовать ObjectOutputStream и ObjectInputStream (связанные с FileInputStream и FileOutputStream).

Другими словами, 90 % действий по вводу/выводу на языке Java будут основаны на уже рассмотренных объектах.

**В: А что вы скажете о новых классах ввода/вывода (nio), добавленных в версии 1.4?**

О: Классы пакета java.nio значительно увеличивают производительность работы и используют для этого мощность машины, на которой запускается программа. Одна из ключевых новых возможностей nio состоит в том, что непосредственно вы контролируете буфер. Другая новая особенность — неблокирующий режим ввода/вывода, то есть ваш код ввода/вывода не будет ждать появления данных для чтения или записи. Некоторые существующие классы (включая FileInputStream и FileOutputStream) скрыто используют новые возможности. Классы nio более сложные в использовании, и, если они вам *действительно* не нужны, можете пользоваться более простыми версиями, которые уже рассматривались. К тому же, если вы не будете соблюдать осторожность, работа с классами nio может привести к потере производительности. Ввод/вывод без nio, вероятно, подойдет для 90 % обычных процедур, особенно если вы только начали изучать язык Java.

*Можно* облегчить свой путь к nio-классам, используя FileInputStream и получая доступ к его каналу через метод getChannel() (добавленный в FileInputStream в версии 1.4).

### КЛЮЧЕВЫЕ МОМЕНТЫ

- Чтобы записать данные в текстовый файл, начните с создания потока для соединения FileWriter.
- Для эффективности подключите FileWriter к BufferedWriter.
- Объект File указывает на файл в виде определенного пути, но при этом не представляет фактического содержимого файла.
- С помощью объекта File можно создавать, прослеживать и удалять каталоги.
- Большинство потоков, использующих строковое имя файла, могут также использовать объект File, и это может оказаться безопаснее.
- Чтобы прочитать файл, начните с создания потока для соединения FileReader.
- Для эффективности подключите FileReader к BufferedReader.
- Для разбора текстового файла нужно убедиться, что он записан способом, позволяющим различать отдельные элементы. Чаще всего для разделения отдельных частей используют какой-нибудь символ.
- Применяйте метод split() из класса String для разделения строки на отдельные лексемы. Строка с одним разделителем будет содержать две лексемы — по одной с каждой стороны разделителя. *Разделитель не считается лексемой.*

# Идентификатор версии: большой подвох сериализации

Теперь вы узнали, насколько прост ввод/вывод в языке Java, особенно если использовать наиболее распространенные комбинации соединений. Но есть одна проблема, которая *действительно* может заставить вас поволноваться.

## Крайне важно контролировать версии!

Если вы сериализуете объект, то должны иметь класс для его десериализации и дальнейшего использования. Это очевидно. Но что произойдет, если вы за это время ***измените класс***? Представьте, что вы пытаетесь возвратить объект Dog, а одна из его переменных (непереходная) поменяла свой тип с double на String. Это в значительной степени нарушает безопасность типов в языке Java. Но это не единственное изменение, которое может навредить совместимости. Подумайте о следующем.

**Изменения класса, которые могут навредить десериализации.**

Удаление переменной экземпляра.

Изменение объявленного типа переменной.

Замена непереходной переменной на переходную.

Перемещение класса вверх или вниз по иерархии наследования.

Изменение класса (где угодно в графе объекта) из Serializable в не-Serializable (удаляя implements Serializable из объявления класса).

Превращение обычной переменной в статическую.

**Допустимые изменения класса.**

Добавление новых переменных в класс (существующие объекты будут десериализованы со значениями по умолчанию для переменных, которых они не имели до сериализации).

Добавление классов к иерархии наследования.

Удаление классов из иерархии наследования.

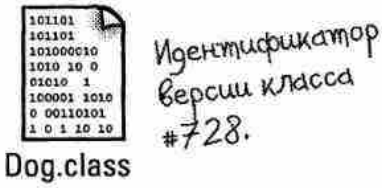
Изменение уровня доступа переменной не влияет на способность десериализации присваивать ей значение.

Замена непереходной переменной на переходную (ранее сериализованные объекты имели значение по умолчанию для переменных, которые до этого были переходными).

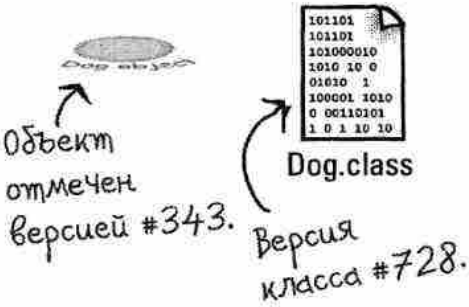
1. Вы создаете класс Dog.

2. Вы сериализуете объект Dog, используя этот класс.

3. Вы изменяете класс Dog.




4. Вы десериализуете объект Dog, используя измененный класс.




5. Сериализация провалилась!

JVM сообщает: «Вы не можете научить старый объект Dog новому коду».

# Использование serialVersionUID

Каждый раз, когда объект сериализуется, он вместе со всеми объектами в графе «маркируется» идентификационным номером версии своего класса. Этот номер (ID) называют serialVersionUID. Он вычисляется на основании информации о структуре класса. Если класс был изменен после сериализации, то при десериализации объекта он может иметь уже другой serialVersionUID и операция провалится. Но вы можете контролировать этот процесс.

**Если есть хоть какая-то возможность, что ваш класс мог *поменяться*, добавьте в него серийный идентификационный номер версии.**

Когда Java пробует десериализовать объект, сравнивается serialVersionUID сериализованного объекта с serialVersionUID класса, используемого JVM. Например, экземпляр класса Dog был сериализован с ID 23 (в действительности serialVersionUID намного длиннее). Когда JVM десериализует объект Dog, то сначала сравнивает serialVersionUID объекта Dog с serialVersionUID класса Dog. Если числа не совпадают, то JVM предполагает, что класс не совместим с ранее сериализованным объектом и во время десериализации вы получаете исключение.

Итак, решение состоит в том, чтобы поместить serialVersionUID в ваш класс, и тогда, даже если ваш класс будет развиваться, его serialVersionUID останется прежним, а JVM скажет: «Хорошо, круто, класс совместим с этим сериализованным объектом», даже если класс фактически изменился.

Это работает, если *только* вы будете осторожно изменять класс! Другими словами, *вы* берете на себя всю ответственность за любые проблемы, которые могут возникнуть, когда более старый объект будет восстанавливаться более новым классом.

Чтобы получить serialVersionUID для класса, используйте инструмент serialver, который поставляется вместе с пакетом программ для разработки приложений на Java.

```
File Edit Window Help serialKiller
% serialver Dog
Dog: static final long
serialVersionUID =
-5849794470654667210L;
```

**Если вы думаете, что ваш класс мог измениться после того, как кто-то сериализовал его объекты...**

1. Используйте инструмент командной строки serialver для получения идентификатора версии класса.

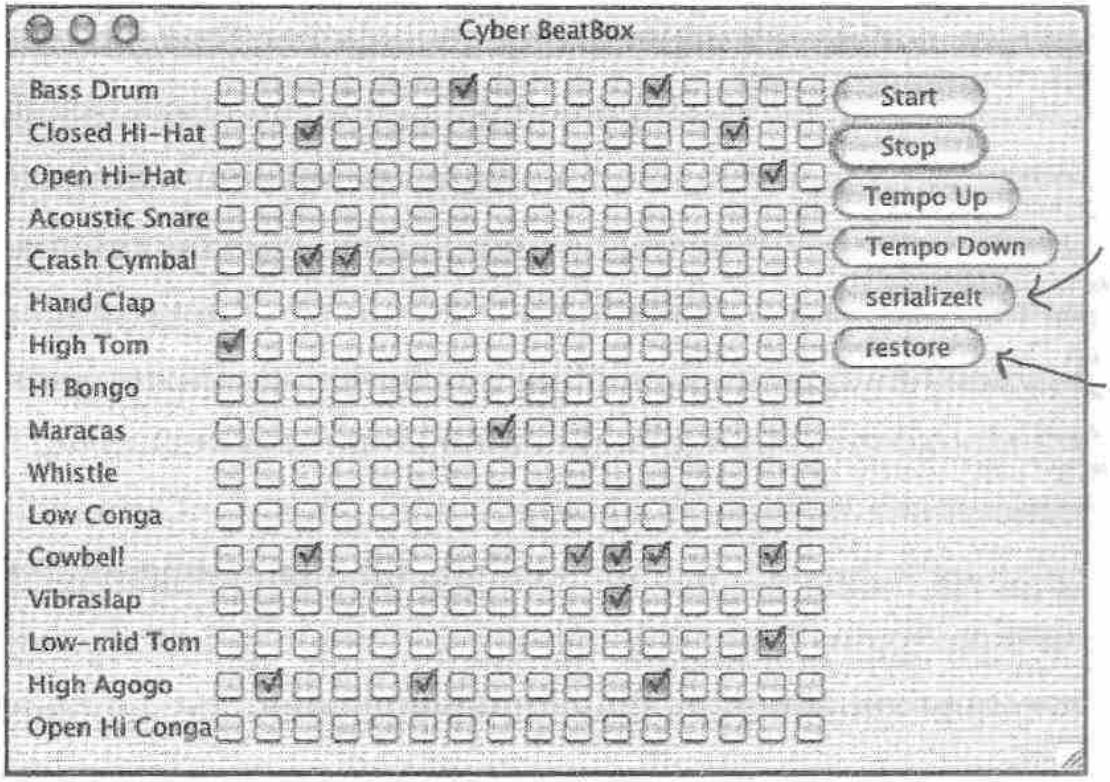
```
File Edit Window Help serialKiller
% serialver Dog
Dog: static final long
serialVersionUID =
-5849794470654667210L;
```

2. Вставьте полученный результат в свой класс.

```
public class Dog {

    static final long serialVersionUID =
                    -6849794470754667710L;

    private String name;
    private int size;

    // Здесь находится код метода
}
```

3. Внося изменения в класс, убедитесь, что берете на себя ответственность за их последствия! Например, убедитесь, что ваш новый класс Dog может иметь дело со старым, десериализованным со значениями по умолчанию для переменных, добавленных в класс *после того*, как Dog был сериализован.

# Кухня кода




При нажатии кнопки serializeIt (Сериализуй это) текущая схема будет сохранена.

Кнопка restore (Восстановить) возвращает сохраненную схему.

***Сделаем так, чтобы BeatBox сохранял и восстанавливал вашу любимую схему***

# Сохранение схемы BeatBox

В BeatBox схема барабанов — это набор флажков. При воспроизведении последовательности код пробегает по флажкам, чтобы узнать, какие барабаны играют в каждом из 16 ударов. Для сохранения схемы необходимо сохранить состояния этих флажков.

Можно сделать простой булев массив, содержащий состояния каждого из 256 флажков. Объект массива считается сериализуемым, если все его элементы также сериализуемы, так что не должно возникнуть проблем при сохранении массива с булевыми элементами.

Чтобы загрузить схему, программа считывает единственный объект — булев массив (десериализуем его) — и восстанавливает состояния флажков. Большую часть вы уже видели в Кухне кода, когда мы создавали GUI для BeatBox, поэтому сейчас рассмотрим только код сохранения и восстановления.

Эта Кухня кода подготовит вас к материалу следующей главы, где вы научитесь отсылать схему по *сети* на сервер, вместо того чтобы записывать ее в *файл*. Кроме того, не придется загружать схемы из файла — можно будет получать их от *сервера* каждый раз при отсылке участником.

**Сериализация схемы**

Это внутренний класс внутри кода BeatBox.

```
public class MySendListener implements ActionListener {

    public void actionPerformed(ActionEvent a) {

        boolean[] checkboxState = new boolean[256];

        for (int i = 0; i < 256; i++) {

            JCheckBox check = (JCheckBox) checkboxList.get(i);
            if (check.isSelected()) {
               checkboxState[i] = true;
            }
        }

        try {
           FileOutputStream fileStream = new FileOutputStream(new File("Checkbox.ser"));
           ObjectOutputStream os = new ObjectOutputStream(fileStream);
           os.writeObject(checkboxState);
        } catch(Exception ex) {
            ex.printStackTrace();
        }

    } // Закрываем метод
} // Закрываем внутренний класс
```

Все это происходит при нажатии кнопки, после чего срабатывает ActionEvent.

Создаем булев массив для хранения состояния каждого флажка.

Пробегаем через checkboxList (ArrayList, содержащий состояния флажков), считываем состояния и добавляем полученные значения в булев массив.

Это самая простая часть. Просто запишите/сериализуйте булев массив!

# Восстановление схемы BeatBox

Этот процесс в значительной степени представляет собой сохранение в обратном порядке: считываем булев массив и используем его для восстановления состояний флажков. Все происходит при нажатии кнопки restore (Восстановить).

## Восстановление схемы

Это другой класс внутри класса Beatbox.

```
public class MyReadInListener implements ActionListener {

    public void actionPerformed(ActionEvent a) {
       boolean[] checkboxState = null;
       try {
           FileInputStream fileIn = new FileInputStream(new File("Checkbox.ser"));
           ObjectInputStream is = new ObjectInputStream(fileIn);
           checkboxState = (boolean[]) is.readObject();

       } catch(Exception ex) {ex.printStackTrace();}


       for (int i = 0; i < 256; i++) {
          JCheckBox check = (JCheckBox) checkboxList.get(i);
          if (checkboxState[i]) {
             check.setSelected(true);
          } else {
             check.setSelected(false);
           }
       }

       sequencer.stop();
       buildTrackAndStart();

    } // Закрываем метод
} // Закрываем внутренний класс
```

Считываем объект из файла и определяем его как булев массив (помните, readObject() возвращает ссылку на тип Object).

Теперь восстанавливаем состояние каждого флажка в ArrayList, содержащий объекты JCheckBox (checkboxList).

Здесь мы останавливаем проигрывание мелодии и восстанавливаем последовательность, используя новые состояния флажков в ArrayList.

### Наточите свой карандаш

У этой версии есть огромный недостаток! При нажатии кнопки serializeIt (Сериализуй это) сериализация происходит автоматически в файл Checkbox.ser (который при необходимости создается). Но при каждом сохранении вы переписываете ранее сохраненный файл.

Усовершенствуйте функции сохранения и восстановления, добавив JFileChooser, чтобы можно было именовать и сохранять столько различных схем, сколько понадобится, и загружать/восстанавливать их из *любых* ранее сохраненных файлов.

Наточите свой карандаш

## Можно ли их сохранить?

Как вы думаете, что из этого можно сериализовать, а что — нет? Если нельзя, то почему? Не имеет смысла? Рискуем безопасностью? Работает только для текущей сессии JVM? Постарайтесь ответить, не подглядывая в API.

| Тип объекта | Сериализуемо? | Если нет, то почему? |
|---|---|---|
| Object | Да/Нет | _____ |
| String | Да/Нет | _____ |
| File | Да/Нет | _____ |
| Date | Да/Нет | _____ |
| OutputStream | Да/Нет | _____ |
| JFrame | Да/Нет | _____ |
| Integer | Да/Нет | _____ |
| System | Да/Нет | _____ |

## Что допустимо?

Обведите фрагменты кода, которые скомпилируются (при условии, что находятся в допустимом классе).

```
FileReader fileReader = new FileReader();
BufferedReader reader = new BufferedReader(fileReader);
```

---

```
FileOutputStream f = new FileOutputStream(new File("Foo.ser"));
ObjectOutputStream os = new ObjectOutputStream(f);
```

---

```
BufferedReader reader = new BufferedReader(new FileReader(file));
String line = null;
while ((line = reader.readLine()) != null) {
    makeCard(line);
}
```

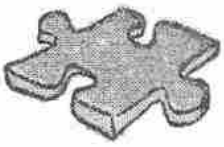
---

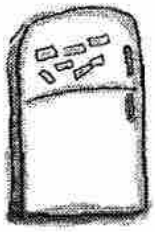
```
ObjectInputStream is = new ObjectInputStream(new FileOutputStream("Game.ser"));
GameCharacter oneAgain = (GameCharacter) is.readObject();
```

Эта глава открыла вам удивительный мир ввода/вывода в языке Java. Определите, истинны или ложны следующие утверждения, касающиеся ввода/вывода.

## Правда или ложь

1. Сериализация подходит для сохранения только тех данных, которые используются программами не на Java.
2. Состояние объекта можно сохранить лишь с помощью сериализации.
3. ObjectOutputStream используется для сериализации объектов.
4. Цепные потоки могут использоваться сами по себе или с потоками для соединения.
5. Одиночный вызов метода writeObject() может сохранить множество объектов.
6. Все классы сериализуемы по умолчанию.
7. Модификатор transient позволяет делать сериализуемыми переменные экземпляра.
8. Если родительский класс несериализуем, то его потомок не может быть сериализуемым.
9. При десериализации объекты считываются в обратном порядке (начиная с последнего в файле).
10. При десериализации объекта его конструктор не запускается.
11. Оба процесса — и сериализация, и сохранение в текстовый файл — могут вызвать исключения.
12. BufferedWriter можно связать с FileWriter.
13. Объекты File представляют файлы, но не каталоги.
14. Вы не можете заставить буфер отослать данные, пока он не наполнился.
15. Оба процесса — и чтение, и запись файла — могут выполняться с помощью буфера.
16. Метод split() из класса String включает разделители в результирующий массив.
17. Любое изменение класса разрушает ранее сериализованные объекты этого класса.





# Магнитики с кодом

Это задание немного мудреное, поэтому мы повысили его статус с упражнения до головоломки. Восстановите фрагменты кода, чтобы получилась рабочая программа на языке Java, которая выдаст результат, приведенный ниже. Возможно, вам не понадобятся все магнитики. Кроме того, один магнитик можно использовать несколько раз.

```
class DungeonGame implements Serializable {

try {

FileOutputStream fos = new
  FileOutputStream("dg.ser");

short getZ() {
  return z;

e.printStackTrace();

oos.close();

ObjectInputStream ois = new
  ObjectInputStream(fis);

int getX() {
  return x;

System.out.println(d.getX()+d.getY()+d.getZ());

FileInputStream fis = new
  FileInputStream("dg.ser");

public int x = 3;
transient long y = 4;
private short z = 5;

long getY() {
  return y;

class DungeonTest {

ois.close();

import java.io.*;

fos.writeObject(d);

} catch (Exception e) {

d = (DungeonGame) ois.readObject();

ObjectOutputStream oos = new
  ObjectOutputStream(fos);

oos.writeObject(d);

public static void main(String [] args) {
  DungeonGame d = new DungeonGame();
```

```
File Edit Window Help Torture
% java DungeonTest
12
8
```

Ответы

## Правда или ложь

1. Сериализация подходит для сохранения только тех данных, которые используются программами не на Java. **Ложь**
2. Состояние объекта можно сохранить лишь с помощью сериализации. **Ложь**
3. ObjectOutputStream используется для сериализации объектов. **Правда**
4. Цепные потоки могут использоваться сами по себе или с потоками для соединения. **Ложь**
5. Одиночный вызов метода writeObject() может сохранить множество объектов. **Правда**
6. Все классы сериализуемы по умолчанию. **Ложь**
7. Модификатор transient позволяет делать сериализуемыми переменные экземпляра. **Ложь**
8. Если родительский класс несериализуем, то его потомок не может быть сериализуемым. **Ложь**
9. При десериализации объекты считываются в обратном порядке (начиная с последнего в файле). **Ложь**
10. При десериализации объекта его конструктор не запускается. **Правда**
11. Оба процесса — и сериализация, и сохранение в текстовый файл — могут вызвать исключения. **Правда**
12. BufferedWriter можно связать с FileWriter. **Правда**
13. Объекты File представляют файлы, но не каталоги. **Ложь**
14. Вы не можете заставить буфер отослать данные, пока он не наполнился. **Ложь**
15. Оба процесса — и чтение, и запись файла — могут выполняться с помощью буфера. **Правда**
16. Метод split() из класса String включает разделители в результирующий массив. **Ложь**
17. Любое изменение класса разрушает ранее сериализованные объекты этого класса. **Ложь**

## Магнитики с кодом

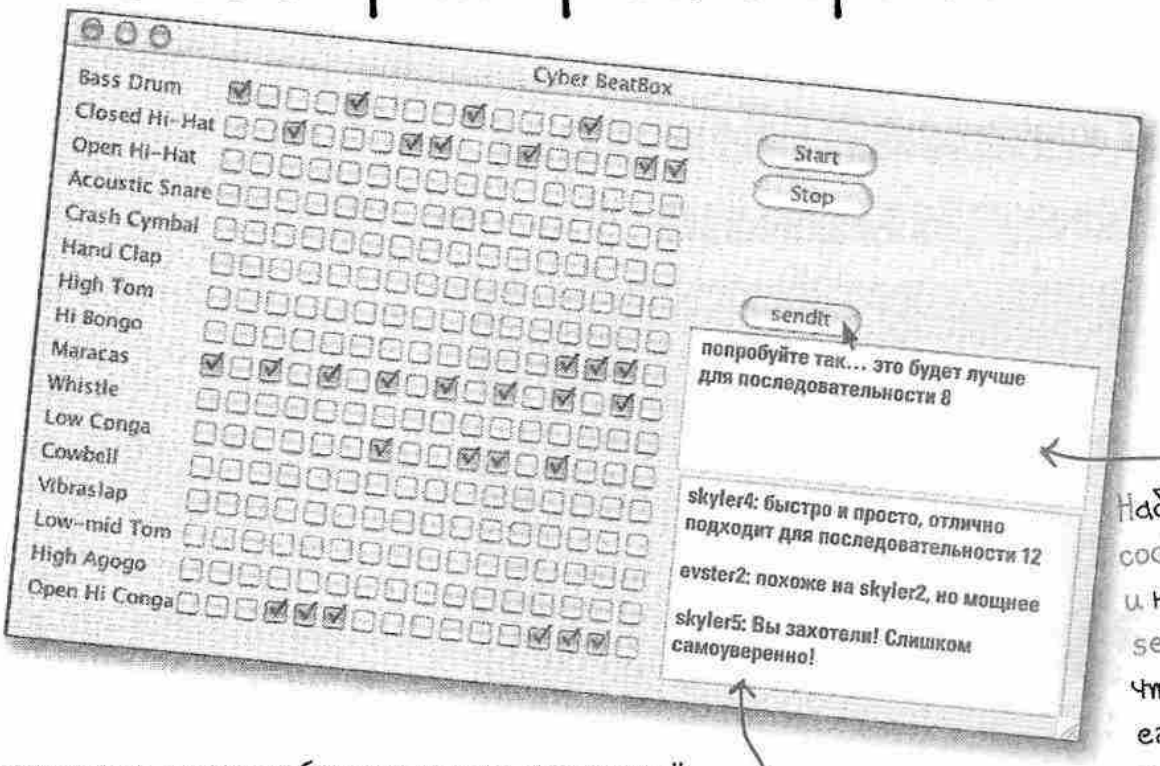
```
import java.io.*;

class DungeonGame implements Serializable {
  public int x = 3;
  transient long y = 4;
  private short z = 5;
  int getX() {
    return x;
  }
  long getY() {
    return y;
  }
  short getZ() {
    return z;
  }
}

class DungeonTest {
  public static void main(String [] args) {
    DungeonGame d = new DungeonGame();
    System.out.println(d.getX() + d.getY() + d.getZ());
    try {
      FileOutputStream fos = new FileOutputStream("dg.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(d);
      oos.close();
      FileInputStream fis = new FileInputStream("dg.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      d = (DungeonGame) ois.readObject();
      ois.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
    System.out.println(d.getX() + d.getY() + d.getZ());
  }
}
```

```
File Edit Window Help Escape
% java DungeonTest
12
8
```

# Устанавливаем соединение

**Свяжитесь с внешним миром.** Ваша Java-программа может соединяться и взаимодействовать с программами на других компьютерах. Это просто. Обо всех низкоуровневых сетевых компонентах заботятся классы из библиотеки java.net. В Java отправка и получение данных по Сети — это обычный ввод/вывод, разве что со слегка измененным соединительным потоком в конце цепочки. Получив класс BufferedReader, вы можете *считывать* данные. Ему все равно, откуда они приходят — из файла или по сетевому кабелю. В этой главе вы установите связь с внешним миром с помощью сокетов. Вы создадите *клиентские* и *серверные* сокеты. В итоге у вас будут *клиенты* и *серверы*. И вы сделаете так, чтобы они смогли общаться между собой. К концу этой главы вы получите полноценный многопоточный клиент для чатов. Ой, мы только что сказали *«многопоточный»*? Именно! Сейчас вы узнаете, как, не прерывая беседу с Бобом, слушать, о чем говорит Сьюзи.

# Чат в BeatBox в режиме реального времени

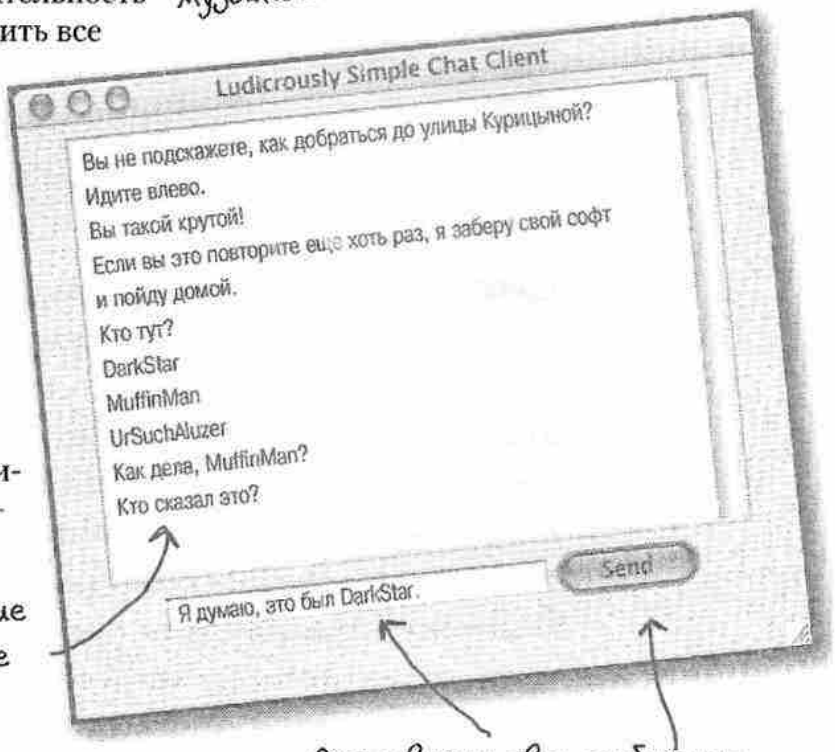



Набираем сообщение и нажимаем кнопку sendIt (Отправить), чтобы отослать его вместе с музыкальным шаблоном.

При щелчке на полученном сообщении также загружается музыкальный шаблон.

Представьте, что вы работаете над компьютерной игрой. Вместе со своей командой вы создаете звуковое оформление для каждой игровой составляющей. Используя версию BeatBox с чатом, ваша команда может работать совместно — отправлять последовательность тактов с сообщением, которые смогут получить все участники чата. У вас появляется возможность не только *читать* сообщения своих коллег, но и загружать и *проигрывать* музыкальные последовательности, щелкая на сообщениях в соответствующей области.

В этой главе вы узнаете, что нужно сделать для создания такого чат-клиента. Мы даже немного затронем тему создания чат-*сервера*. Полную версию BeatBox с поддержкой чата мы оставим для Кухни кода, а пока разработаем два чат-клиента, которые принимают и отправляют текстовые сообщения, — «невероятно простой» и «очень простой».



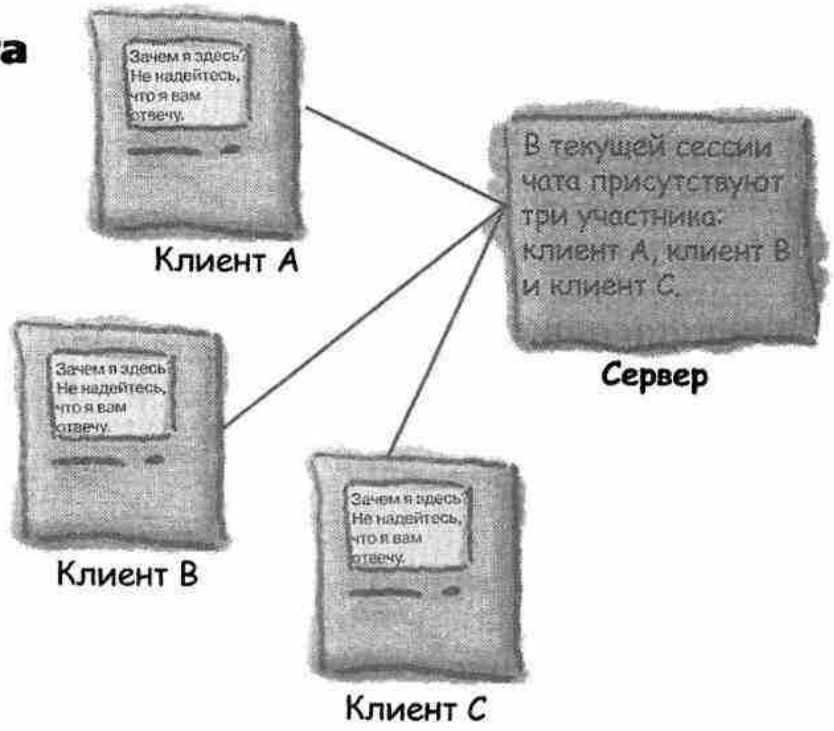

Можно проводить в чате настоящие интеллектуальные беседы. Каждое сообщение отправляется всем участникам.

Отправляем свое сообщение на сервер.

## Обзор программы для чата

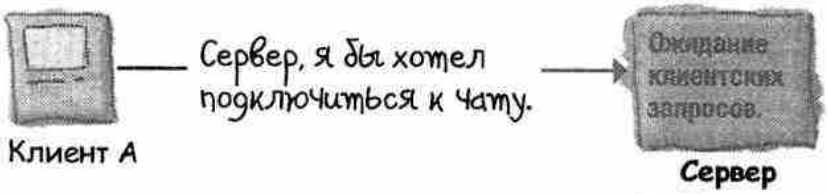
Клиент должен знать о сервере.

Сервер должен знать обо всех клиентах.



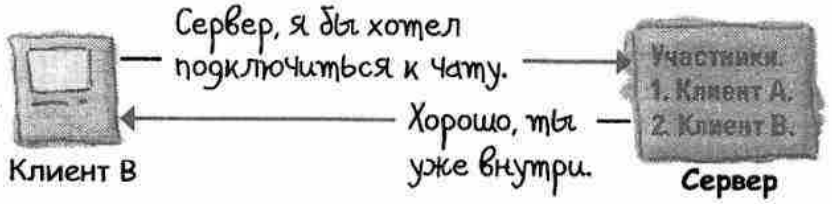

## Как это работает.

1. Клиент соединяется с сервером.

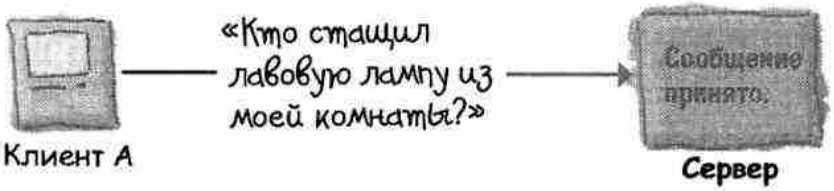



2. Сервер устанавливает соединение и добавляет клиент в список участников.

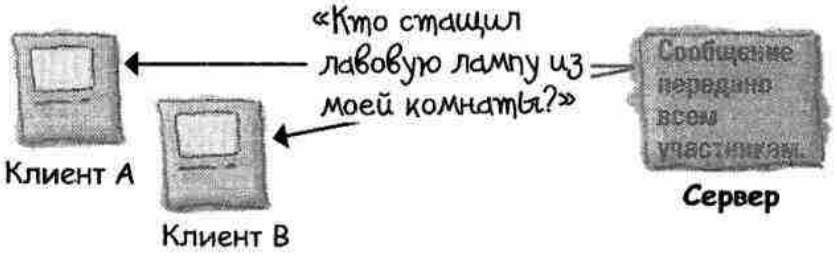
3. Подключается еще один клиент.




4. Клиент A отправляет сообщение в чат.




5. Сервер передает сообщение всем участникам (включая оригинального отправителя).



# Подключение, отправка и прием

Чтобы заставить клиент работать, нужно научиться:

- инициировать **соединение** между клиентом и сервером;
- **отправлять** сообщения *на* сервер;
- **принимать** сообщения *от* сервера.

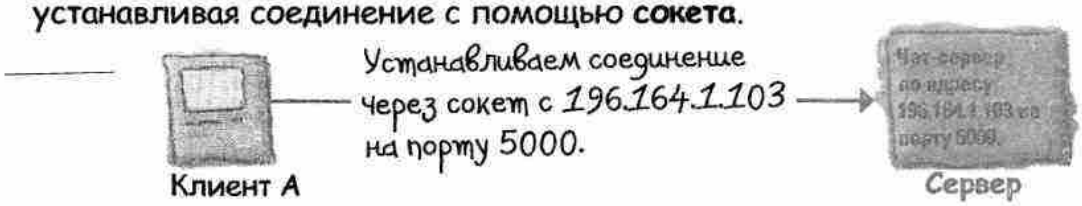
Во время этого на низком уровне происходит много разных вещей. Но нам повезло, так как пакет для работы с сетью из Java API (java.net) значительно упрощает жизнь программистам. Вы намного чаще будете иметь дело с кодом для GUI, нежели с сетью и вводом/выводом.

Но это еще не все.

В простом клиенте для чата затаилась проблема, с которой мы до сих пор не сталкивались: выполнение двух действий одновременно. Установление соединения — это единовременная операция (соединение либо устанавливается, либо нет). Но после этого участник чата хочет *отправлять* и **в то же время** *принимать сообщения* от других клиентов (через сервер). Вот над этим нужно поразмыслить, что мы и сделаем уже через несколько страниц.

### ❶ Соединение.

Клиент подключается к серверу, устанавливая соединение с помощью **сокета**.




### ❷ Отправка.

Клиент **отправляет** сообщение на сервер.

### ❸ Прием.

Клиент **получает** сообщение от сервера.

# Устанавливаем сетевое соединение с помощью сокета

Чтобы подключиться к другому компьютеру, необходимо установить соединение через сокет. Сокет (класс java.net.Socket) — это объект, представляющий сетевое соединение между двумя компьютерами. Что за соединение? Это *отношения* между **двумя узлами, программное обеспечение которых знает о существовании друг друга**. Что еще более важно, это ПО знает, как между собой *общаться*, то есть обмениваться *битами*.

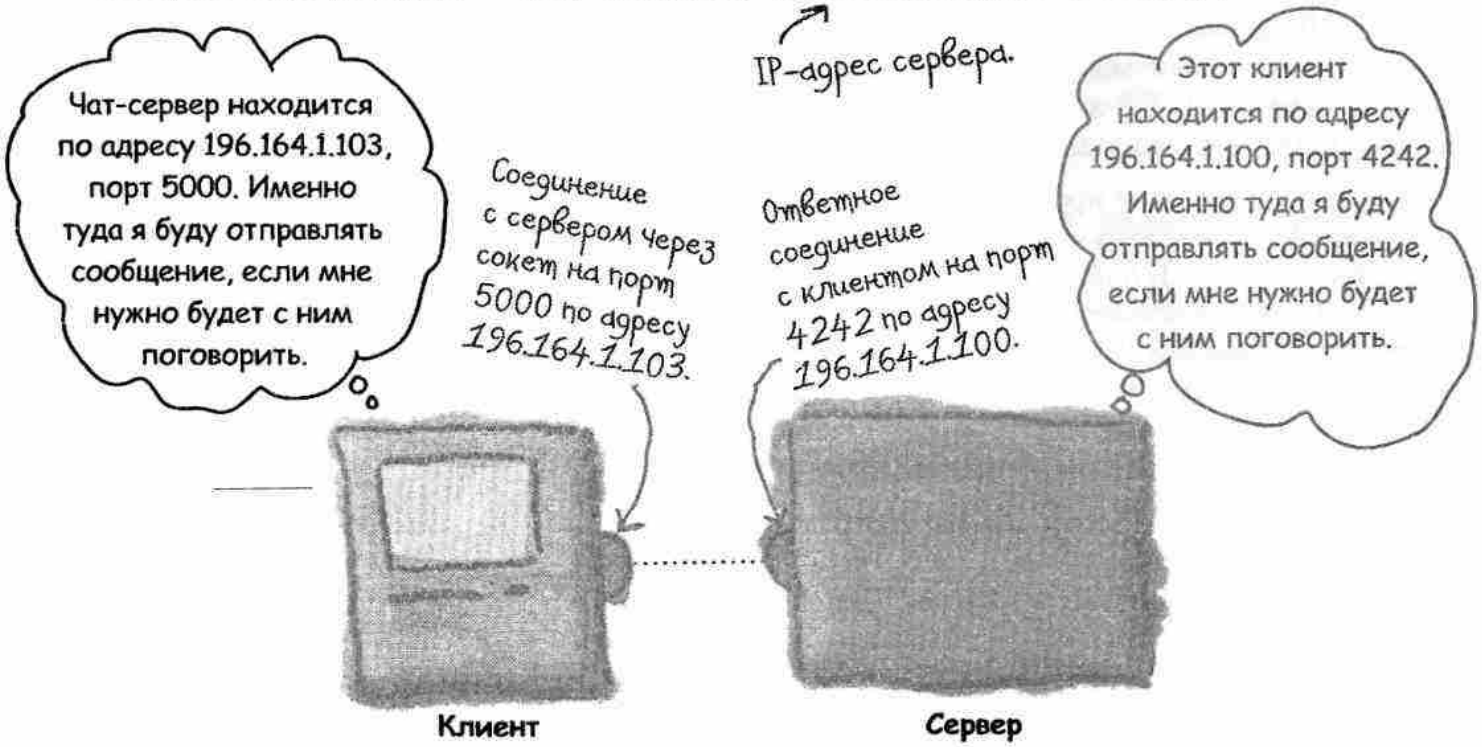
К счастью, нас не интересуют подробности, так как они скрыты на более низком уровне сетевого стека. Не переживайте, если вам не известно значение данного термина. Это просто способ представления слоев, через которые должна проходить информация (биты), чтобы из программы, работающей под управлением JVM на какой-то операционной системе, добраться до физического устройства (например, сетевого кабеля), откуда она отправится на другой компьютер. *Кто-то* должен позаботиться обо всех мелочах. Но не вы. Этот «кто-то» — сочетание специфического для системы программного обеспечения и сетевого API, предоставляемого Java. Вам же достается самая простая работа, касающаяся *очень* высокого программного уровня. Готовы?

Чтобы создать соединение через сокет, нужно знать о сервере две вещи: где он находится и на каком порту работает. Иными словами, нужно знать IP-адрес и номер порта.

Номер TCP-порта.

```
Socket chatSocket = new Socket("196.164.1.103", 5000);
```

IP-адрес сервера.




Соединение с помощью сокета подразумевает, что у двух компьютеров есть информация друг о друге, включая сетевое местоположение (IP-адрес) и порт TCP.

# Порт TCP — это просто 16-битное число, с помощью которого распознается конкретная программа на сервере

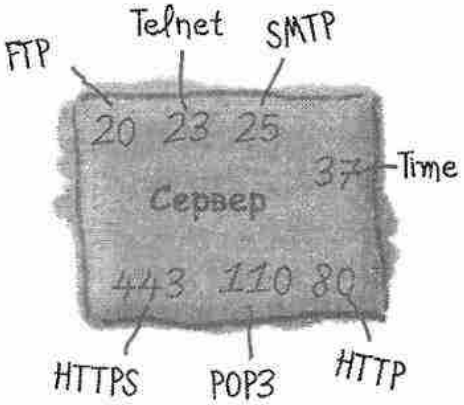
Веб-серверы (HTTP) в Интернете работают на порту 80. Это стандарт. Если у вас есть Telnet-сервер, он работает на порту 23. FTP? 20. Почтовый POP3-сервер? 110. SMTP? 25. Сервер точного времени разместился на порту 37. Воспринимайте эти числа как уникальные идентификаторы. Они представляют логическую связь с конкретной частью программного обеспечения, работающего на сервере. И больше ничего. Повертев в руках системный блок, вы не сможете найти в нем TCP-порт. Но на сервере таких портов 65 536 (0–65 536). Естественно, это не те порты, к которым можно подключать физические устройства. Это просто числа, представляющие приложения.

Без номера порта сервер не сможет узнать, к какому приложению хочет подключиться клиент. Подумайте, какими проблемами это чревато, ведь у каждой программы может быть свой протокол. Представьте, что ваш браузер вместо HTTP-сервера попытается подключиться к POP3. Почтовый сервер не знает, как обрабатывать HTTP-запрос! Но даже если бы знал, он все равно не умеет обслуживать запросы по этому протоколу.

При создании серверной программы необходимо указывать порт, на котором она должна работать (далее вы увидите, как это делается в Java). В приложении для чата, которое мы здесь напишем, будет задан порт 5000 просто потому, что нам так захотелось. Кроме того, он удовлетворяет требованию, по которому номер порта должен быть числом между 1024 и 65 535. Почему 1024? Потому что порты от 0 до 1023 зарезервированы для известных сервисов (некоторые мы перечислили выше).

Если вы пишете сервисы (серверные программы) для корпоративной сети, то обратитесь к системным администраторам и узнайте, какие порты уже заняты. Например, вам могут сообщить, что порты ниже 3000 использовать нельзя. В любом случае, если не хотите неприятностей, лучше не используйте недоступные номера портов. Другое дело, когда вы находитесь в своей *домашней* сети. В таком случае нужно лишь посоветоваться со своими *детьми*.

Общеизвестные номера TCP-портов для популярных серверных приложений.




На сервере может быть запущено вплоть до 65 536 различных серверных приложений: по одному на каждый порт.

Номера TCP-портов с 0 по 1023 зарезервированы для популярных сервисов. Не используйте их для своих серверных приложений!*

Чат-сервер, который мы создаем, использует порт 5000. Мы просто выбрали номер между 1024 и 65 535.

* Один из этих портов вы, наверное, *сможете* использовать, но системный администратор, скорее всего, убьет вас за это.

## Это не глупые вопросы

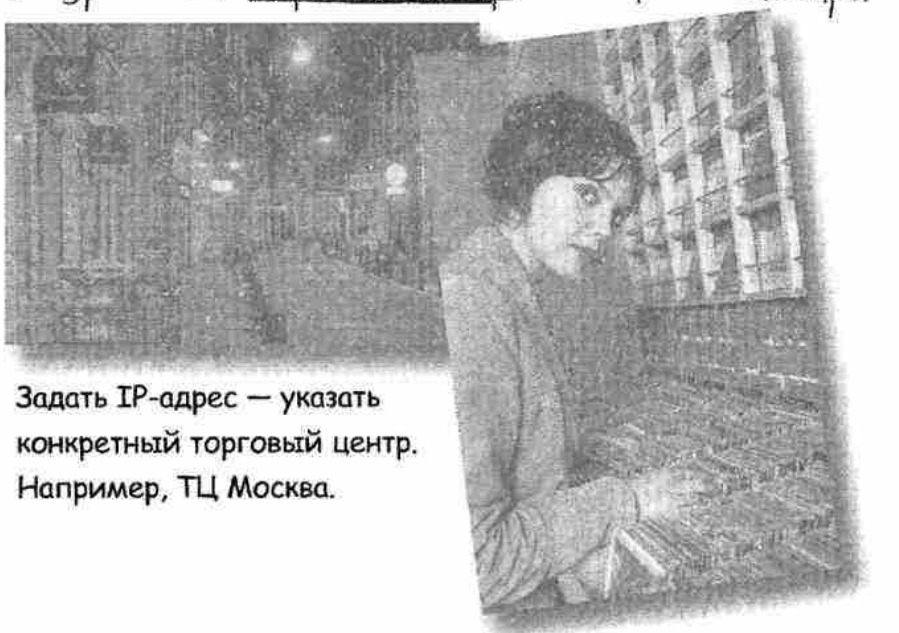
**В: Как вы узнаете номер порта серверной программы, с которой хотите взаимодействовать?**

О: Это зависит от того, относится ли программа к популярным сервисам. Если вы подключаетесь к одному из сервисов, перечисленных на предыдущей странице (HTTP, SMTP, FTP и т. д.), то можете узнать его порт в Интернете (поищите в Google «известные порты TCP»). Вы также можете спросить у своего друга системного администратора.

В противном случае придется обратиться к настройщику сервиса. Спросите его. Или ее. Как правило, когда человек пишет сетевой сервис и хочет, чтобы другие люди создавали для него свои клиенты, он публикует соответствующий IP-адрес, порт и протокол. Например, если вам захотелось написать клиент для сервера игры в го, вы можете посетить сайт этого сервера и найти нужную информацию.

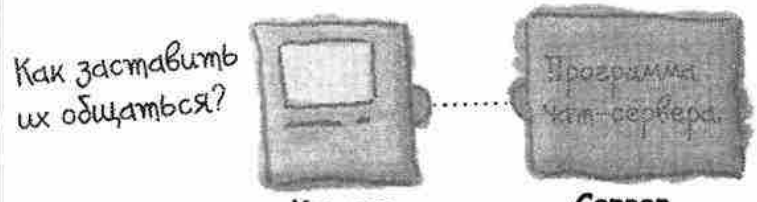
**В: Может ли на одном порту работать несколько программ? Иначе говоря, могут ли два приложения на одном сервере использовать один и тот же номер порта?**

О: Нет! Если вы попытаетесь *привязать* серверную программу к определенному порту, который уже занят, то получите исключение BindException. Вы больше узнаете об этом, когда мы дойдем до раздела, посвященного серверам.



## Штанга для мозга

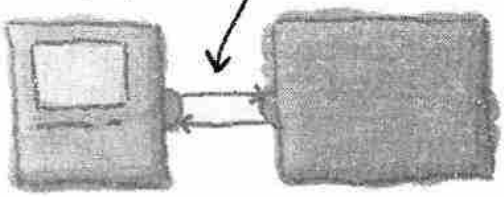
Хорошо, вы наладили соединение с помощью сокета. Клиент и сервер знают IP-адреса и номера портов друг друга. Что дальше? Как общаться через это соединение? Иными словами, как перегонять биты от одной программы к другой? Представьте, какие сообщения потребуется принимать и отправлять вашему чат-клиенту.



# Чтобы считать данные из сокета, используйте BufferedReader

Для передачи информации через сокет применяются потоки. Старые добрые потоки ввода/вывода, которые мы задействовали в предыдущей главе. Одна из самых замечательных особенностей Java заключается в том, что в подавляющем большинстве случаев при работе с вводом/выводом не надо вникать, к чему именно подключен высокоуровневый поток. Проще говоря, вы можете использовать BufferedReader точно так же, как вы это делали при записи в файл; разница только в том, что теперь поток будет подключаться к *сокету* (Socket), а не к *файлу* (File)!

Входящий и исходящий потоки, предоставляемые сокетом.



**1 Создаем сокет и связываемся с сервером.**

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

Номер порта чат-сервера, который вы уже знаете, ведь мы сказали вам, что это 5000.

127.0.0.1 — локальный IP-адрес, то есть адрес компьютера, на котором выполняется данный код. Вы можете использовать его при тестировании своих клиентов и серверов на одной отдельной машине.

**2 Создаем InputStreamReader и связываем его с низкоуровневым потоком (соединением) через сокет.**

```
InputStreamReader stream = new InputStreamReader(chatSocket.getInputStream());
```

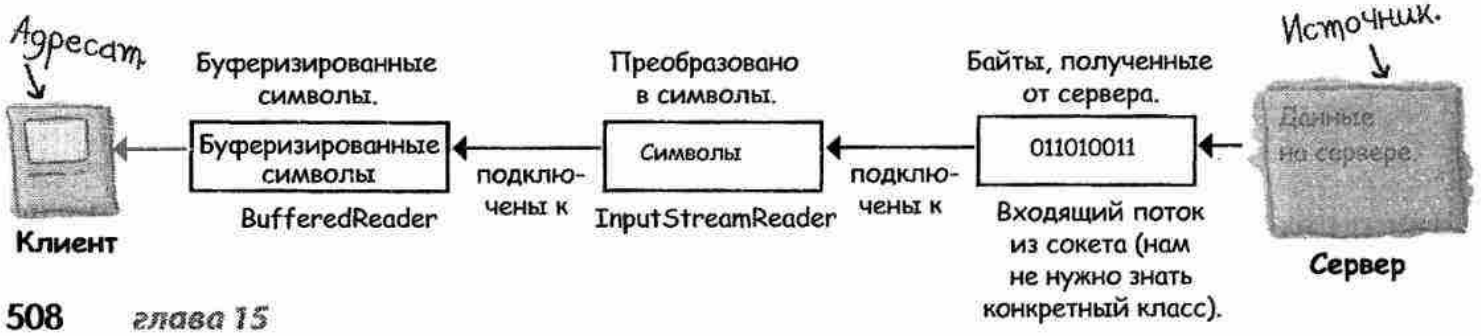
InputStreamReader — это «мост», соединяющий низкоуровневый поток байтов (скажем, получаемый из сокета) и высокоуровневый символьный поток (например, предоставляемый объектом BufferedReader, который находится на вершине цепочки).

Нужно лишь запросить у сокета входящий поток! Он низкоуровневый, но мы просто подсоединим его к объекту, который лучше умеет работать с текстом.

**3 Создаем BufferedReader и считываем данные!**

```
BufferedReader reader = new BufferedReader(stream);
String message = reader.readLine();
```

Подключим BufferedReader к InputStreamReader (который, в свою очередь, подключен к низкоуровневому потоку из сокета).



# Чтобы записать данные в сокет, используйте PrintWriter

В предыдущей главе вместо PrintWriter применялся BufferedWriter. Здесь вам предоставляется выбор, но стандартным классом для записи строк (одной за раз) считается именно PrintWriter. Он содержит два знакомых вам метода — print() и println()! Как в старом добром System.out.

**1 Создаем сокет и связываемся с сервером.**

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

Здесь все то же самое, что и на предыдущей странице, — чтобы записать данные на сервер, необходимо к нему подключиться.

**2 Создаем PrintWriter и связываем его с низкоуровневым исходящим потоком, полученным из сокета.**

```
PrintWriter writer = new PrintWriter(chatSocket.getOutputStream());
```

PrintWriter ведет себя как мост между символьными данными и байтами, которые получает из низкоуровневого исходящего потока, предоставляемого сокетом. Подключив PrintWriter к исходящему потоку, мы можем записывать строки в сокет.

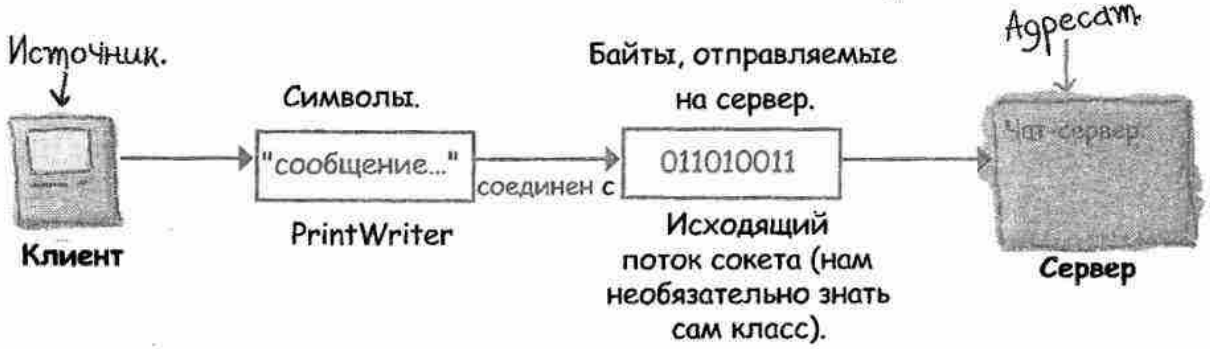
Сокет дает нам низкоуровневый поток. Мы передаем его в конструктор PrintWriter, создавая цепочку.

**3 Записываем (выводим) что-нибудь.**

```
writer.println("Сообщение для отправки");
writer.print("Другое сообщение");
```

println() добавляет символ переноса строки в конце данных, которые шлет

print() не добавляет символ переноса строки.



# DailyAdviceClient

Прежде чем приступить к созданию чата, начнем с чего-нибудь более скромного. Советчик — это серверная программа, дающая практичные, вдохновляющие советы, которые помогают скрасить трудовые будни.

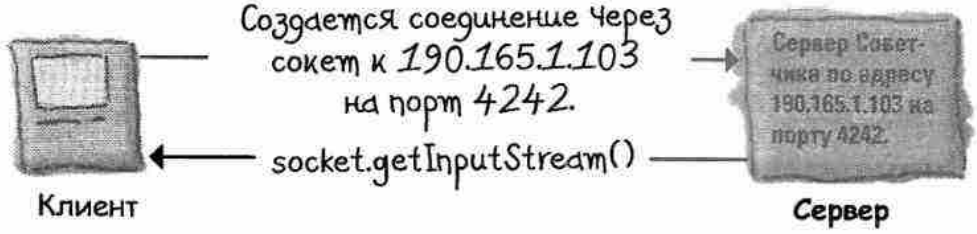
Мы создадим для этого приложения клиента. При каждом подключении он будет принимать сообщение от сервера.

Чего же вы ждете? Кто *знает*, какие возможности вы теряете без этой программы?

Советчик

## ❶ Подключаемся.

Клиент подключается к серверу и принимает от него входящий поток.




## ❷ Считываем.

Клиент считывает сообщение, полученное от сервера.

# Код программы DailyAdviceClient

**Программа создает сокет и объект BufferedReader с помощью других потоков, а затем считывает одну строку, которую ей передает серверное приложение, работающее на порту 4242.**

```
import java.io.*;
import java.net.*;

public class DailyAdviceClient {

    public void go() {
        try {
            Socket s = new Socket("127.0.0.1", 4242);

            InputStreamReader streamReader = new InputStreamReader(s.getInputStream());
            BufferedReader reader = new BufferedReader(streamReader);

            String advice = reader.readLine();
            System.out.println("Сегодня ты должен:" + advice);

            reader.close();
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        DailyAdviceClient client = new DailyAdviceClient();
        client.go();
    }
}
```

Класс Socket из java.net.

Здесь может случиться что угодно.

Создаем соединение через сокет к приложению, работающему на порту 4242, на том же компьютере, где выполняется данный код (localhost).

Подключаем BufferedReader к InputStreamReader (который уже соединен с исходящим потоком сокета).

Здесь закрываются все потоки.

Метод readLine() работает точно так же, как если бы BufferedReader был подключен к файлу. Иными словами, работая с методом из BufferedReader, вы не знаете (или вам все равно), откуда пришли символы.

## Наточите свой карандаш

Проверьте свои знания о потоках/классах для чтения и записи данных в сокет. Постарайтесь не смотреть на соседнюю страницу!

Чтобы ***прочитать*** текст из сокета:

**Клиент**

Нарисуйте и подпишите цепочку потоков, которая используется клиентом для чтения данных из сервера.

Источник.

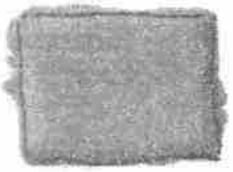
**Сервер**

Чтобы ***отправить*** текст в сокет:

**Клиент**

Нарисуйте и подпишите цепочку потоков, которая используется клиентом для отправки данных на сервер.

Адресат.



**Сервер**

## Наточите свой карандаш

**Заполните пустые поля:**

Какие две части информации нужны клиенту, чтобы установить соединение с сервером через сокет? _____ _____

Какие TCP-порты зарезервированы для широко известных сервисов HTTP и FTP? _____

Правда или Ложь: диапазон допустимых TCP-портов может быть представлен переменной типа short? _____

# Создание простого сервера

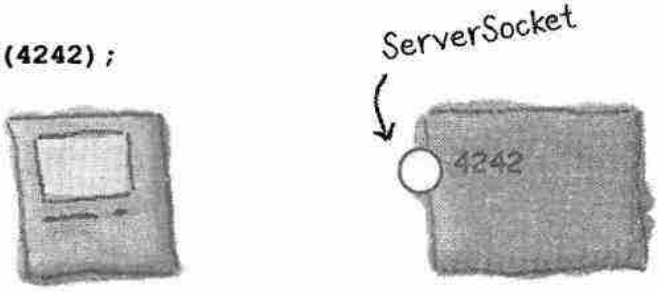
Что же понадобится для написания серверного приложения? Всего пара сокетов. Да, пара, то есть *два*. ServerSocket, который ожидает запросов со стороны клиента (когда клиент выполняет new Socket()), и обычный Socket для общения с клиентом.

## Как это работает.

1. Серверное приложение создает объект ServerSocket, указывая конкретный порт:

```
ServerSocket serverSock = new ServerSocket(4242);
```

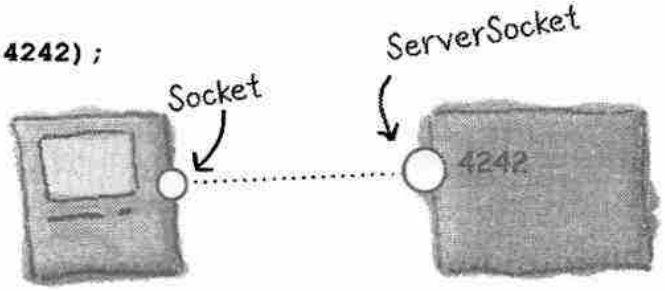
Таким образом, сервер начинает отслеживать входящие клиентские запросы на порту 4242.




2. Клиент создает сокет и связывается с серверным приложением:

```
Socket sock = new Socket("190.165.1.103", 4242);
```

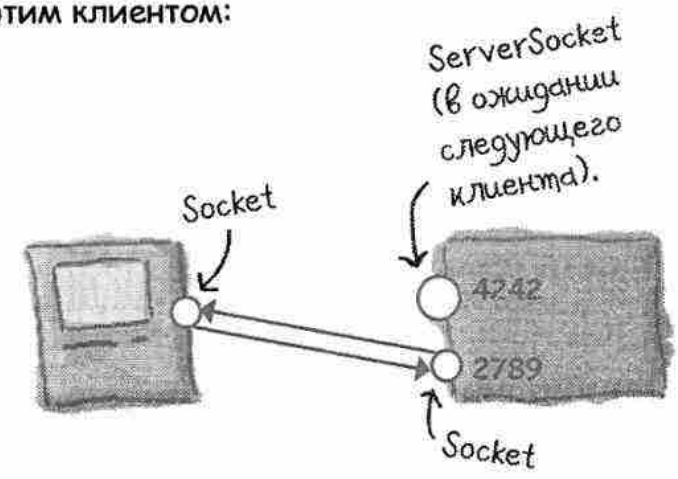
Клиент знает IP-адрес и номер порта (опубликованные или переданные ему настройщиком сервера).




3. Сервер создает новый сокет для общения с этим клиентом:

```
Socket sock = serverSock.accept();
```

Метод accept() блокирует программу, пока ожидает подключения клиентского сокета. Когда клиент наконец пытается подключиться, метод возвращает обычный объект Socket (на *другом* порту), который знает, как взаимодействовать с клиентом (то есть знает его IP-адрес и номер порта). Порт, на котором работает Socket, отличается от порта ServerSocket; поэтому ServerSocket может дальше отслеживать новые подключения.



# Код приложения DailyAdviceServer

Эта программа создает ServerSocket и ожидает клиентские запросы. При получении такого запроса (когда клиент выполнил для этого приложения new Socket()) сервер создает объект Socket и устанавливает соединение с этим клиентом. Сервер создает экземпляр PrintWriter (с помощью исходящего потока из сокета) и отправляет клиенту сообщение.

```
import java.io.*;
import java.net.*;

public class DailyAdviceServer {

   String[] adviceList = {"Ешьте меньшими порциями", "Купите облегающие джинсы. Нет,
они не делают вас полнее.", "Два слова: не годится", "Будьте честны хотя бы сегодня.
Скажите своему начальнику все, что вы *на самом деле* о нем думаете.", "Возможно, вам
стоит подобрать другую прическу."};

   public void go() {

      try {
         ServerSocket serverSock = new ServerSocket(4242);

         while(true) {

            Socket sock = serverSock.accept();

            PrintWriter writer = new PrintWriter(sock.getOutputStream());
            String advice = getAdvice();
            writer.println(advice);
            writer.close();
            System.out.println(advice);
         }

      } catch(IOException ex) {
         ex.printStackTrace();
      }
   } // Закрываем go

   private String getAdvice() {
      int random = (int) (Math.random() * adviceList.length);
      return adviceList[random];
   }

   public static void main(String[] args) {
      DailyAdviceServer server = new DailyAdviceServer();
      server.go();
   }
}
```

Не забудьте об операторах импорта.

Ежедневные советы берутся из этого массива.

Помните, что здесь есть переносы строк, созданные редактором. Никогда не нажимайте Enter посередине строки!

Благодаря ServerSocket приложение отслеживает клиентские запросы на порту 4242 на том же компьютере, где выполняется данный код.

Сервер входит в постоянный цикл, ожидая клиентских подключений (и обслуживая их).

Метод accept() блокирует приложение до тех пор, пока не поступит запрос, после чего возвращает сокет (на анонимном порту) для взаимодействия с клиентом.

Теперь мы используем соединение объекта Socket с клиентом для создания экземпляра PrintWriter, после чего отправляем с его помощью (println()) строку с советом. Затем мы закрываем сокет, так как работа с клиентом закончена.

# Штанга для мозга

***Откуда сервер знает, как связываться с клиентом?***

Клиенту известен IP-адрес и порт сервера, но как сервер соединяется с клиентом (создавая входящие и исходящие потоки)?

Подумайте о том, как, когда и откуда сервер получает информацию о клиенте.

## Это не глупые вопросы

**В:** **Наш сервер-советчик, код которого мы привели на предыдущей странице, имеет очень серьезное ограничение — он может одновременно работать лишь с одним клиентом!**

**О:** Верно. Он не может принимать входящие запросы, пока не закончит работу с текущим клиентом и снова не войдет в бесконечный цикл. В этом цикле сервер вызывает accept() в ожидании запросов, после обработки которых опять создаст сокет, но уже для нового клиента, и все повторится заново.

**В:** **Попробую спросить по-другому: как я могу создать сервер, который способен работать с несколькими клиентами одновременно? Например, текущая реализация абсолютно не годится для чат-сервера.**

**О:** Это не сложно. Используйте отдельные потоки для каждого нового клиентского сокета. Вы вот-вот узнаете, как это делается!

## КЛЮЧЕВЫЕ МОМЕНТЫ

- Клиентские и серверные приложения взаимодействуют с помощью сокетов.
- Сокет — это соединение между двумя приложениями, которые могут работать на разных компьютерах (а могут и на одном).
- Клиент должен знать IP-адрес (или доменное имя) и TCP-порт серверного приложения.
- TCP-порт — это 16-битное положительное число, которое присваивается конкретному серверному приложению. Благодаря номерам TCP-портов разные клиенты могут подключаться к одному компьютеру, но взаимодействовать на нем с разными приложениями.
- Номера портов от 0 до 1023 зарезервированы для таких «общеизвестных сервисов», как HTTP, FTP, SMTP и т. д.
- Клиент подключается к серверу, создавая сокет:

```
Socket s = new Socket("127.0.0.1", 4200);
```

- Подключившись, клиент может работать с входящими и исходящими потоками, предоставляемыми сокетом. Это так называемые низкоуровневые потоки:

```
sock.getInputStream();
```

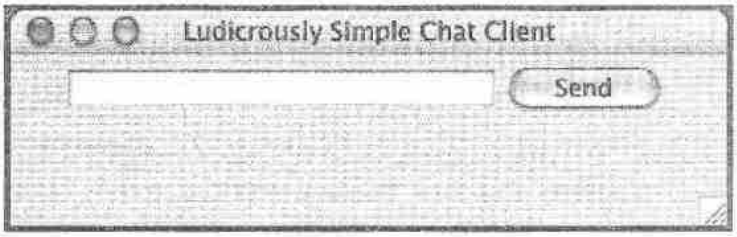
- Чтобы прочитать данные, полученные от сервера, нужно создать объект BufferedReader и подключить его к InputStreamReader, который, в свою очередь, связан с входящим потоком из сокета.
- Объект InputStreamReader — это «промежуточный» поток, который принимает байты и конвертирует их в текстовые (символьные) данные. Как правило, он используется в качестве связующего звена между высокоуровневым BufferedReader и низкоуровневым входящим потоком из сокета.
- Чтобы записать текстовые данные на сервер, нужно создать объект PrintWriter, напрямую подключенный к исходящему потоку из сокета. Чтобы отправить строку на сервер, достаточно воспользоваться методами print() или println().
- На сервере используется ServerSocket, который ожидает поступления клиентских запросов на определенном порту.
- Когда ServerSocket получает запрос, он его «принимает», создавая соединение с клиентом через объект Socket.

# Создание чат-клиента

Мы напишем клиентское приложение для чата в два этапа. Сначала мы создадим версию, которая умеет отправлять сообщения на сервер, но не способна принимать их от других участников (удивительное и загадочное перекручивание всей концепции чат-конференций).

Затем мы доработаем чат так, чтобы он мог как отправлять сообщения, так и принимать их.

## Версия № 1: отправка




Наберите сообщение, затем нажмите кнопку Send (Отправить), чтобы передать его на сервер. В этой версии мы не будем получать сообщений от сервера, поэтому здесь нет прокручиваемой текстовой области.

## Общая структура кода

```
public class SimpleChatClientA {

   JTextField outgoing;
   PrintWriter writer;
   Socket sock;

   public void go() {
        // Создаем GUI и подключаем слушатель для событий к кнопке отправки
        // Вызываем метод setUpNetworking()
   }

   private void setUpNetworking() {
        // Создаем сокет и PrintWriter
        // Присваиваем PrintWriter переменной writer
   }

   public class SendButtonListener implements ActionListener {
      public void actionPerformed(ActionEvent ev) {
         // Получаем текст из текстового поля и отправляем
         // его на сервер с помощью переменной writer (PrintWriter)
      }
   }  // Закрываем вложенный класс SendButtonListener

} // Закрываем внешний класс
```

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleChatClientA {

    JTextField outgoing;
    PrintWriter writer;
    Socket sock;

    public void go() {
        JFrame frame = new JFrame("Ludicrously Simple Chat Client");
        JPanel mainPanel = new JPanel();
        outgoing = new JTextField(20);
        JButton sendButton = new JButton("Send");
        sendButton.addActionListener(new SendButtonListener());
        mainPanel.add(outgoing);
        mainPanel.add(sendButton);
        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        setUpNetworking();
        frame.setSize(400,500);
        frame.setVisible(true);
    } // Закрываем go

    private void setUpNetworking() {
        try {
            sock = new Socket("127.0.0.1", 5000);
            writer = new PrintWriter(sock.getOutputStream());
            System.out.println("networking established");
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    } // Закрываем setUpNetworking

  public class SendButtonListener implements ActionListener {
     public void actionPerformed(ActionEvent ev) {
         try {
            writer.println(outgoing.getText());
            writer.flush();

         } catch(Exception ex) {
            ex.printStackTrace();
         }
         outgoing.setText("");
         outgoing.requestFocus();
     }
    }  // Закрываем вложенный класс SendButtonListener

    public static void main(String[] args) {
       new SimpleChatClientA().go();
    }
} // Закрываем внешний класс
```

Импорт пакетов для потоков (java.io), сокетов (java.net) и GUI.

Создание GUI. Здесь нет ничего нового и связанного с сетью или вводом/выводом.

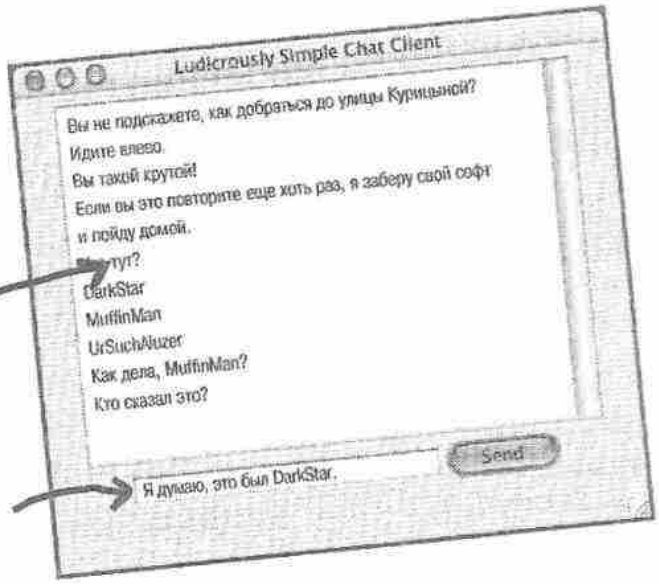
Мы используем localhost, поэтому можно тестировать клиент и сервер на одном компьютере.

Здесь мы создаем объекты Socket и PrintWriter (этот метод вызывается прямо перед выводом графического интерфейса на экран).

Именно здесь начинается непосредственная запись. Помните, что переменная writer подключена к входящему потоку из сокета, поэтому при каждом вызове println() данные передаются по сети на сервер.

**Если вы хотите проверить программу прямо сейчас, наберите заранее подготовленный код для сервера, размещенный на последних страницах этой главы. Сначала запустите сервер в командной строке, затем используйте еще один терминал, чтобы запустить клиент.**

## Версия № 2: отправка и прием




Входящие сообщения.

Отправляемое сообщение.

Приняв сообщение, сервер шлет его всем клиентам участников конференции. Пока сообщение не будет отправлено всем, оно не появится на панели входящих сообщений создавшего его клиента.

### Важный вопрос: *как* вы будете получать сообщения от сервера?

Это легко. При настройке соединения создайте также входящий поток с помощью BufferedReader. Затем прочитайте сообщение, используя readLine().

### Еще более важный вопрос: *когда* вы будете получать сообщения от сервера?

Подумайте об этом. Какие варианты возможны?

1. **Первый вариант: опрашивать сервер каждые 20 секунд.**

   **Плюсы**: Это выполнимо.

   **Минусы**: Как сервер узнает, что вы уже видели, а что — еще нет? Ему придется хранить сообщения, вместо того чтобы просто переправлять их. И почему 20 секунд? Такая задержка повлияет на практичность приложения, но, уменьшив ее, вы рискуете загрузить сервер бесполезными запросами. Неэффективно.

2. **Второй вариант: читать данные с сервера каждый раз, когда пользователь отправляет сообщение.**

   **Плюсы**: Выполнимо, очень просто.

   **Минусы**: Глупость. Почему для проверки сообщений выбирается случайный момент времени? А если пользователь предпочитает читать и ничего не отправляет?

3. **Третий вариант: читать сообщение в тот момент, когда оно приходит с сервера.**

   **Плюсы**: Наиболее эффективно и практично.

   **Минусы**: Как выполнять два действия одновременно? Где вы разместите код? Вам нужен цикл в таком месте, где можно всегда быть готовым прочесть ответ сервера. Но где найти подобное место? После загрузки GUI в программе больше ничего не происходит, пока графический компонент не инициирует событие.

С Java вы действительно можете прогуливаться и жевать резинку одновременно.

## Как вы догадались, мы выбрали третий вариант.

Мы хотим, чтобы проверка сообщений на сервере происходила непрерывно и *не отбирала у пользователя возможность взаимодействовать с GUI*! Итак, пока пользователь успешно вводит новые сообщения или прокручивает список, что-то *невидимое* должно считывать новые входящие данные с сервера.

Это означает, что нам наконец-то понадобился новый поток (не путать с сетевым потоком). Новый, отдельный стек.

Мы хотим, чтобы все работало так, как в первой версии, но при этом параллельно выполнялся новый *процесс*, считывающий информацию с сервера и отображающий ее в текстовом поле.

Или не совсем так. Каждый новый поток в Java не считается отдельным процессом, выполняющимся операционной системой, если только на компьютере не установлено несколько процессоров. Но *выглядит* это именно так.

## Многопоточность в Java

Поддержка многопоточности встроена непосредственно в язык Java. Создать новый исполняемый поток не составляет труда:

```
Thread t = new Thread();
t.start();
```

Вот и все. Создавая новый *объект* Thread, мы запускаем отдельный *исполняемый поток*, содержащий свой стек вызовов.

***Только есть одна проблема.***

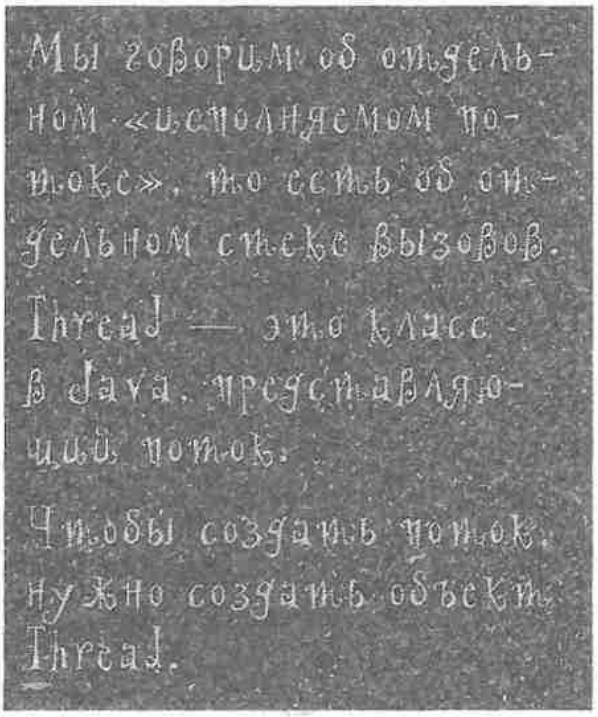
Этот поток ничего не *делает*, поэтому он «умрет» сразу после своего появления. Со смертью потока исчезает и его новый стек. Конец истории.

Итак, мы упустили один ключевой компонент — *задачу*, выполняемую потоком. Иными словами, нужен код, который мы хотим выполнять в отдельном потоке.

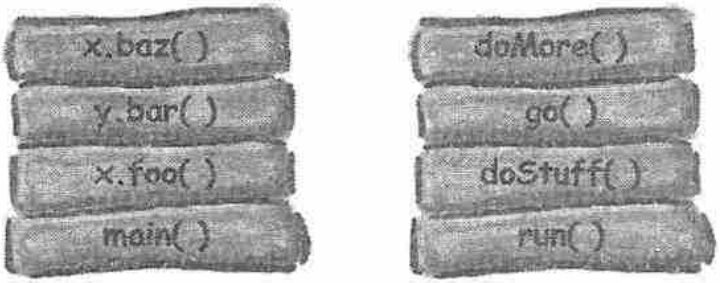
Многопоточность в Java подразумевает наличие как самого *потока*, так и выполняемой им задачи. Вы также должны познакомиться с *классом* Thread из пакета java.lang (помните, что этот пакет импортируется сам и содержит классы, наиболе фундаментальные для языка, включая String и System).

# Несмотря на поддержку нескольких потоков, в Java есть лишь один класс Thread

Мы можем говорить как о самих *потоках* (threads), так и о классе **Thread** (Потоке с большой буквы). Когда вы видите слово *«поток»*, значит, мы говорим об отдельном исполняемом потоке. Когда вы видите слово **«Поток»** (или Thread), вспомните соглашение об именовании, принятое в Java. Что должно начинаться с большой буквы? Названия классов и интерфейсов. В данном случае **Thread** — класс из пакета java.lang. Объект **Thread** представляет собой *поток выполнения*. Каждый раз, когда вам нужно начать выполнение нового потока, вы создаете экземпляр класса **Thread**.




**Поток**

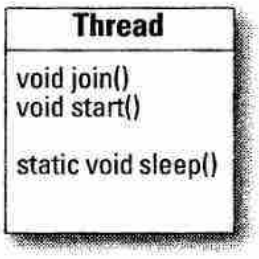



**Главный поток**

**Еще один поток, созданный внутри кода**

Поток выполняется отдельно. Это означает, что у него есть свой стек вызовов. Любая Java-программа создает главный поток, который на дне своего стека содержит метод main(). За запуск главного потока отвечает JVM (она контролирует и другие потоки, например поток для сборки мусора). Как программист вы можете написать код для создания собственных потоков.

**Thread**




**Класс java.lang.Thread**

Thread — это класс, который представляет собой исполняемый поток. Он содержит методы для запуска и приостановления потока, для присоединения одного потока к другому (на самом деле методов больше; мы перечислили те, без которых не сможем обойтись).

# Что значит иметь несколько стеков вызовов

При наличии нескольких стеков вызовов создается *видимость* того, что разные действия выполняются одновременно. На практике только настоящая мультипроцессорная система способна выполнять несколько операций в один момент времени. В случае с потоками в Java управление может переходить от одного стека к другому настолько быстро, что создается впечатление, будто все стеки выполняются одновременно. Помните, Java — это всего лишь процесс, работающий под управлением исходной ОС. Итак, Java *сам* представляет собой «текущий процесс», выполняющийся в операционной системе. Но *что* именно при этом *запускается* внутри JVM? Какой байт-код выполняется? Код, который находится на вершине текущего стека! И на протяжении каких-то 100 мс система может переключиться на *другой* метод в *другом* стеке.

Одна из обязательных функций потока — отслеживание выражения (находящегося в методе), которое в данный момент выполняется в его стеке.

Выглядит это примерно так.

**1 JVM вызывает метод main().**

```
public static void main(String[] args) {
...
}
```

---

**2 Метод main() запускает новый поток. Главный поток временно приостановлен, пока стартует новый поток.**

```
Runnable r = new MyThreadJob();
Thread t = new Thread(r);
t.start();
Dog d = new Dog();
```

Очень скоро вы узнаете, что это означает...

---

**3 JVM переключается между новым потоком (пользовательским потоком A) и исходным главным потоком, пока оба не завершатся.**

# Как запустить новый поток

## ❶ Создаем объект Runnable (задачу для потока).

```
Runnable threadJob = new MyRunnable();
```

Runnable — это интерфейс, который мы рассмотрим на следующей странице. Далее мы создадим класс, реализующий Runnable. В этом классе будет описываться работа, которую должен выполнить поток (то есть метод, который запустится из нового стека вызовов, принадлежащего потоку).

## ❷ Создаем объект Thread (исполнитель) и передаем ему Runnable.

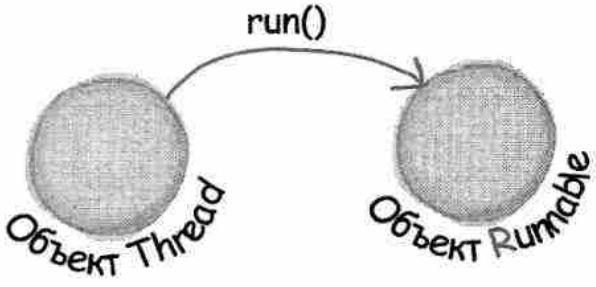
```
Thread myThread = new Thread(threadJob);
```

Передаем объект Runnable конструктору класса Thread. Благодаря этому свежесозданный поток узнает, какой метод нужно поместить на самое дно нового стека — метод run() из Runnable.

## ❸ Запускаем поток.

```
myThread.start();
```

Пока вы не запустите метод start(), ничего не случится. В этот момент экземпляр класса Thread превращается в новый исполняемый поток. При запуске поток берет метод run() из объекта Runnable и помещает его на дно нового стека.



# Любому потоку нужна задача для выполнения — метод, расположенный в нижней части нового стека

Runnable для Потока — то же самое, что задание для работника. Runnable — это задача, которую поток должен выполнять.

Runnable содержит метод run(), который отправляется на дно нового стека.

Объекту Thread нужна задача, которую новый поток начнет выполнять при запуске. По сути, это метод, который помещается в стек нового потока и всегда выглядит следующим образом:

```
public void run() {
  // Код, который будет выполняться в новом потоке
}
```

Интерфейс Runnable описывает всего один метод — public void run(). Помните — это интерфейс, значит, метод будет публичным независимо от того, как вы его пометили.

Откуда поток узнает, какой метод нужно помещать на дно стека? Ответ прост — Runnable описывает контракт, потому что Runnable — интерфейс. Задачу для выполнения в потоке можно указать в любом классе, реализующем этот интерфейс. Конструктор Thread заботится только о том, чтобы ему передали объект такого класса.

Передавая Runnable в конструктор класса Thread, вы на самом деле позволяете потоку получить метод run(). Вы даете ему работу, которую он должен выполнить.

# Чтобы создать задачу для своего потока, реализуйте интерфейс Runnable

Runnable находится в пакете java.lang, поэтому его не нужно импортировать.

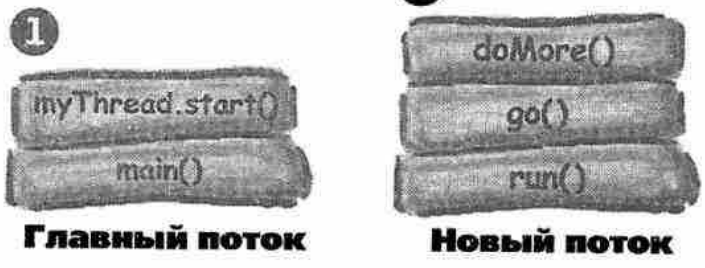
```
public class MyRunnable implements Runnable {

    public void run() {
        go();
    }

    public void go() {
        doMore();
    }

    public void doMore() {
        System.out.println("Вершина стека");
    }
}
```

У Runnable есть всего один метод, который необходимо реализовать, — public void run() (без аргументов). Именно сюда помещается задача, которую поток должен выполнить. Это метод, который появляется на дне нового стека.

Передаем экземпляр Runnable в конструктор Thread. Благодаря этому поток знает, какой метод нужно поместить на дно нового стека. Иными словами, это первый метод, который будет запущен.

```
class ThreadTester {

    public static void main (String[] args) {

        Runnable threadJob = new MyRunnable();
        Thread myThread = new Thread(threadJob);

        myThread .start();

        System.out.println("Возвращаемся в метод main");
    }
}
```

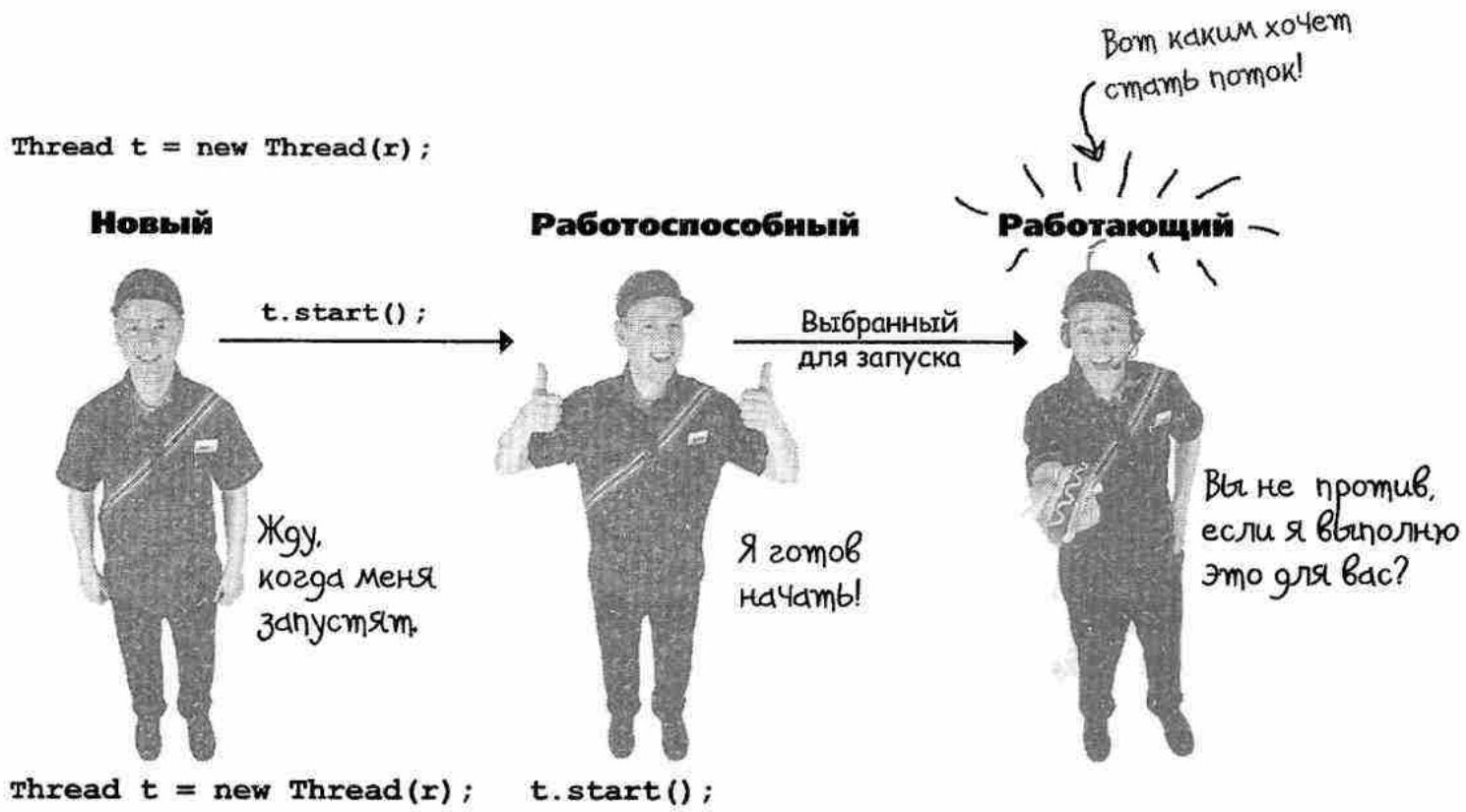
Вы не получите новый исполняемый поток, пока не вызовете метод start() из экземпляра Thread. Поток появляется только после своего запуска. До этого у вас есть обычный объект Thread, но настоящей «поточности» он в себе не несет.



## Штанга для мозга

Как вы думаете, каким будет программный вывод, если запустить класс ThreadTester? Ответ вы найдете через несколько страниц.

# Три состояния нового потока




Экземпляр класса Thread создан, но не запущен. Иначе говоря, есть объект Thread, но нет *исполняемого потока*.

Когда вы запускаете поток, он становится работоспособным. Это означает, что он готов к выполнению и просто ждет своей очереди. На этом этапе у потока уже есть свой стек вызовов.

Это состояние, которое страстно желает обрести любой поток! Стать избранным. Потоком, выполняющимся в данный момент. Только планировщик потоков из JVM может принять такое решение. Иногда можно на него *повлиять*, но нельзя заставить поток работать. В этом состоянии поток (именно данный поток) содержит активный стек вызовов, а метод на его вершине выполняется.

**Но это еще не все. Став работоспособным, поток может менять свое состояние с работающего и обратно, а также становиться *временно неработоспособным* (или *заблокированным*).**

## Типичный цикл состояний работоспособный/работающий

Как правило, поток постоянно становится то работоспособным, то работающим, так как планировщик в JVM позволяет поработать разным потокам.

## Поток может стать временно неработоспособным

Планировщик может сделать работающий поток заблокированным. Для этого есть несколько причин. Например, в потоке выполняется код для чтения входящих данных из сокета, но данные там не обнаруживаются. Планировщик может блокировать поток до тех пор, пока что-то не станет доступно. Или исполняемый код может сказать потоку, чтобы тот приостановил свою работу (sleep()). Или поток может ждать ответа от метода, чей объект заблокирован. В таком случае потоку нельзя продолжать свою работу, пока другой поток не освободит этот объект.

Все эти (и другие) условия приводят к тому, что поток становится временно неработоспособным.

# Планировщик потоков

Планировщик принимает все решения о том, какой поток из работоспособного превращается в работающий и в какой момент (и при каких обстоятельствах) он должен изменить свое состояние обратно. Планировщик решает, кто и как долго должен работать, а также что случается с потоками, когда они перестают выполняться.

Вы не можете управлять планировщиком. Для этого не предусмотрено соответствующих методов. Что еще более важно, процесс планирования потоков не дает вам гарантий (выполнение некоторых принципов *почти* гарантируется, но это очень неопределенно)!

Главный вывод, который из этого следует: ***не допускайте, чтобы корректная работа вашего приложения зависела от конкретной модели поведения планировщика***! Для разных версий JVM существуют различные реализации планировщиков, и даже одна программа на одном и том же компьютере может выдавать разные результаты. Одна из грубейших ошибок начинающих Java-программистов заключается в том, что они тестируют многопоточное приложение на единственной системе. При этом они надеются, что планировщик потоков будет работать одинаково в любой системе.

Что это означает с точки зрения принципа — «пиши один код — запускай его везде»? Все просто: чтобы можно было писать код, совместимый с различными системами, ваша многопоточная программа должна работать независимо от поведения планировщика потоков. Например, вы не можете рассчитывать на то, что планировщик обеспечит идеально паритетное выполнение потоков. Хотя в нынешних реалиях трудно себе представить, что планировщик скажет вашему приложению нечто вроде «Хорошо, поток № 5, ты в деле, и пока я за тебя отвечаю, можешь оставаться здесь до самого завершения метода run()».

Секрет кроется в *приостановке*. Приостанавливая поток даже на несколько миллисекунд, вы заставляете его изменить свое состояние и уступить место другому потоку. Метод sleep() гарантирует выполнение *одного* правила: приостановленный поток не вернется к работе, пока не истечет срок его приостановки. Например, если вы прикажете своему потоку остановиться на две секунды (2000 мс), он ни за что не продолжит свою работу *до* истечения этого времени (но это не значит, что он возобновится сразу через две секунды).

Планировщик потоков принимает все решения относительно того, кто должен работать, а кто — нет. Как правило, он неплохо справляется с чередованием потоков. Но здесь нет никаких гарантий. Планировщик может позволить одному потоку работать сколько его душе угодно, в то время как второй будет «голодать».

# Пример того, каким непредсказуемым может быть планировщик...

## Выполняем этот код на одном и том же компьютере:

```
public class MyRunnable implements Runnable {

   public void run() {
      go();
   }

   public void go() {
      doMore();
   }

   public void doMore() {
      System.out.println("Вершина стека");
   }
}



class ThreadTestDrive {

   public static void main (String[] args) {

      Runnable threadJob = new MyRunnable();
      Thread myThread = new Thread(threadJob);

      myThread.start();

      System.out.println("Возвращаемся в метод main");
   }
}
```

## Программный вывод

```
File Edit Window Help PickMe
% java ThreadTestDrive
Возвращаемся в метод main
Вершина стека
% java ThreadTestDrive
Вершина стека
Возвращаемся в метод main
% java ThreadTestDrive
Вершина стека
Возвращаемся в метод main
% java ThreadTestDrive
Вершина стека
Возвращаемся в метод main
% java ThreadTestDrive
Вершина стека
Возвращаемся в метод main
% java ThreadTestDrive
Вершина стека
Возвращаемся в метод main
% java ThreadTestDrive
Возвращаемся в метод main
Вершина стека
```

Обратите внимание, как случайно меняется порядок. Иногда новый поток завершается первым, иногда — вторым.

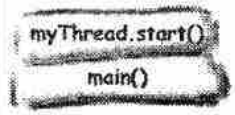
# Откуда появились разные результаты?

## Иногда это работает так:

main() запускает новый поток.

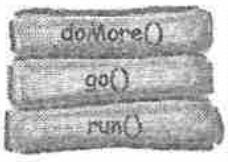
**Главный поток**

Планировщик прерывает работу главного метода и делает его работоспособным, чтобы мог выполниться новый поток.




**Новый поток**

Планировщик позволяет новому потоку полностью отработать и вывести слова «Вершина стека».




**Новый поток**

Новый поток исчезает, так как его метод run() завершился. Главный поток вновь начинает выполняться и выводит «Возвращаемся в метод main».

**Главный поток**

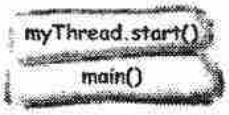
Время

## А иногда — вот так:

main() запускает новый поток.

**Главный поток**

Планировщик прерывает работу главного метода и делает его работоспособным, чтобы мог выполниться новый поток.

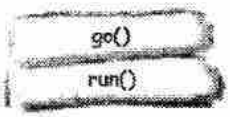



**Главный поток**

Планировщик позволяет новому потоку поработать какое-то время, но этого недостаточно, чтобы завершился метод run().

**Новый поток**

Планировщик делает новый поток работоспособным.




**Новый поток**

Планировщик снова дает главному потоку возможность выполняться. Тот выводит «Возвращаемся в метод main».

**Главный поток**

Новый поток возвращается в состояние работающего и выводит «Вершина стека».

**Новый поток**

Время

## Это не глупые вопросы

**В: Я видел примеры, где вместо отдельной реализации Runnable использовался дочерний от Thread класс с переопределенным методом run(). Таким образом, при создании нового потока вызывается конструктор без аргументов:**

```
Thread t = new Thread(); // без Runnable
```

О: Да, *существует* другой способ создавать потоки, но подумайте об этом с точки зрения ООП. В чем суть наследования? Вспомните, что мы говорили о двух разных сущностях — *потоке* и *задании*. В контексте ООП их функции кардинально различаются, поэтому они описаны в отдельных классах. Единственной причиной для наследования класса Thread может стать необходимость создания нового, более специфического потока. Иными словами, если вы думаете о потоке как об исполнителе, не расширяйте класс Thread (если только вам не нужен *исполнитель* с нестандартным поведением). Но если вам нужно лишь выполнить новую *задачу* с помощью потока/исполнителя, то просто реализуйте Runnable в отдельном классе, ориентированном на задачу, а не на исполнителя.

Эта проблема касается проектирования и не связана с производительностью или языком. Наследовать класс Thread и переопределять его метод run() вполне допустимо, но это не очень хорошая идея.

**В: Можно ли повторно использовать объект Thread? Можно ли дать ему новое задание и опять вызвать метод start()?**

О: Нет. Если метод run() завершил свою работу, то поток уже нельзя запустить. Фактически, в этот момент поток входит в состояние, о котором мы еще не упоминали, — состояние ***смерти***. Несмотря на то что объект Thread навсегда потерял свою «поточность», он по-прежнему может находиться в куче, и у вас есть возможность вызывать другие его методы (если ситуация позволяет). Иными словами, у Thread больше нет отдельного стека вызовов, это уже не *поток*. Это просто такой же объект, как и все остальные.

Существует шаблон проектирования для создания пула потоков, которым вы можете воспользоваться для выполнения разных заданий. Но он не перезапускает мертвые потоки.

### КЛЮЧЕВЫЕ МОМЕНТЫ

- Слово «поток» с маленькой буквы «п» определяет отдельный исполняемый поток в Java.
- Любой поток в Java имеет собственный стек вызовов.
- Thread — это класс java.lang.Thread. Объект Thread представляет собой исполняемый поток.
- Потоку нужно задание, которое он будет выполнять. Это экземпляр класса, который реализует интерфейс Runnable.
- У интерфейса Runnable есть всего один метод — run(). Именно он помещается на дно нового стека вызовов. Проще говоря, это первый метод, который будет запущен в новом потоке.
- Чтобы запустить новый поток, вам нужно передать Runnable в конструктор Thread.
- Поток считается новым, если вы создали экземпляр класса Thread, но еще не вызвали метод start().
- Когда вы запускаете поток (вызывая из объекта Thread метод start()), создается новый стек, на дне которого находится метод run() из Runnable. В этом состоянии поток считается работоспособным, он ожидает, когда его выберут для выполнения.
- Поток считается работающим, когда планировщик потоков в JVM выбирает его для выполнения в текущий момент. На однопроцессорном компьютере может быть лишь один такой поток.
- Иногда поток меняет свое состояние с работающего на заблокированное (становится временно неработоспособным). Поток может быть заблокирован при ожидании данных из сети. Кроме того, его могут приостановить или он ждет ответа от заблокированного объекта.
- Нет гарантий, что процесс планирования будет проходить определенным образом, поэтому вы не можете быть уверены в предсказуемости чередования своих потоков. Периодически приостанавливая их, вы можете повлиять на этот процесс.

# Приостановление потока

Один из лучших способов чередовать потоки — периодически их приостанавливать (усыплять). Требуется лишь вызвать метод sleep() и передать ему продолжительность такого «сна» в миллисекундах.

Например:

```
Thread.sleep(2000);
```

выведет поток из работающего состояния, а через две секунды сделает его работоспособным. Поток *не сможет* продолжить свою работу до истечения этого времени.

К сожалению, метод sleep() может выбросить проверяемое исключение InterruptedException, поэтому все его вызовы должны быть заключены в блок try/catch (или можно объявить исключение). Вызов sleep() на практике выглядит так:

```
try {
    Thread.sleep(2000);
} catch(InterruptedException ex) {
    ex.printStackTrace();
}
```

Усыпите свой поток, чтобы быть уверенным в том, что другие потоки имеют шанс выполнить свою работу.

При пробуждении поток всегда становится работоспособным и ждет момента, когда планировщик позволит ему работать.

Скорее всего, ваш поток *никогда* не будет прерван в приостановленном состоянии. Описанное исключение — часть API для поддержки межпоточного взаимодействия, которое в реальном мире почти никто не использует. Тем не менее необходимо подчиняться правилу «обработай или объяви», поэтому вызов sleep() должен быть заключен в блок try/catch.

Вы уже знаете, что ваш поток не проснется до истечения указанного промежутка времени, но может ли он проснуться после? И да и нет. На самом деле это не имеет значения, так как при просыпании поток ***всегда становится работоспособным***! Он не возобновляет свою работу автоматически в назначенное время. Просыпаясь, поток снова отдает себя на милость планировщику. Для приложений, которые не требуют идеального хронометража и содержат всего несколько объектов Thread, все будет выглядеть так, словно поток возобновляет свою работу точно по графику (скажем, через 2000 мс). Но не рассчитывайте на это при создании своих приложений.

# Используем метод sleep(), чтобы сделать программу более предсказуемой

Помните пример, который мы недавно рассматривали, где одна программа каждый раз выдавала разные результаты? Вернемся к нему и изучим код вместе с вариантом программного вывода. Иногда методу main() приходится ждать, пока не завершится новый поток (и не выведет фразу «На вершине стека»), а иногда новый поток, не закончив начатое, становится работоспособным, давая возможность главному потоку напечатать «Возвращаемся в метод main». Как это можно исправить? Остановитесь на секунду и попробуйте ответить на вопрос: «Куда можно поместить вызов метода sleep(), чтобы фраза из main() всегда выводилась перед фразой из главного потока?»

Мы подождем, пока вы ищете ответ (а правильных ответов сразу несколько).

Ну как, догадались?

Это то, чего мы добиваемся, — последовательный порядок операторов print.

```
File Edit Window Help SnoozeButton
% java ThreadTestDrive
Возвращаемся в метод main
На вершине стека
% java ThreadTestDrive
Возвращаемся в метод main
На вершине стека
% java ThreadTestDrive
Возвращаемся в метод main
На вершине стека
% java ThreadTestDrive
Возвращаемся в метод main
На вершине стека
% java ThreadTestDrive
Возвращаемся в метод main
На вершине стека
```

```
public class MyRunnable implements Runnable {

   public void run() {
      go();
   }

   public void go() {

      try {
        Thread.sleep(2000);
      } catch(InterruptedException ex) {
        ex.printStackTrace();
      }

      doMore();
   }

   public void doMore() {
      System.out.println("top o' the stack");
   }
}

class ThreadTestDrive {
   public static void main (String[] args) {
      Runnable theJob = new MyRunnable();
      Thread t = new Thread(theJob);
      t.start();
      System.out.println("Возвращаемся в метод main");
   }
}
```

Этот вызов sleep() заставляет новый поток поменять свое работающее состояние на другое!

Главный поток опять начинает выполняться и выводит строку «Возвращаемся в метод main». После этого следует пауза (около двух секунд), которая отсрочивает выполнение этой строки (вызов метода doMore() и вывод фразы «На вершине стека»).

# Создание и запуск двух потоков

У потоков есть имена. Вы можете выбрать имя по своему вкусу или согласиться со стандартным названием. Но самое интересное в именах то, что вы можете их использовать для вывода информации о потоке, работающем в данный момент. В следующем примере запускаются два потока. Оба имеют одно задание: войти в цикл и выводить при каждой итерации имя потока, который сейчас работает.

```
public class RunThreads implements Runnable {

    public static void main(String[] args) {
        RunThreads runner = new RunThreads();
        Thread alpha = new Thread(runner);
        Thread beta = new Thread(runner);
        alpha.setName("поток альфа");
        beta.setName("поток бета");
        alpha.start();
        beta.start();
    }

    public void run() {
        for (int i = 0; i < 25; i++) {
            String threadName = Thread.currentThread().getName();
            System.out.println("Сейчас работает " + threadName);
        }
    }
}
```

Создаем один экземпляр Runnable.

Создаем два потока с одним заданием (об одном объекте Runnable сразу для двух потоков мы поговорим через несколько страниц).

Имена потоков.

Запускаем потоки.

Каждый поток пройдет через этот цикл, выводя на экран свое имя при каждой итерации.

## Что же произойдет?

Часть программного вывода, когда цикл повторяется 25 раз.

```
File Edit Window Help Centauri
Сейчас работает поток альфа
Сейчас работает поток альфа
Сейчас работает поток альфа
Сейчас работает поток бета
Сейчас работает поток альфа
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток бета
Сейчас работает поток альфа
```

Будут ли потоки чередоваться? Увидите ли вы их имена сменяющимися поочередно? С какой частотой они будут меняться? При каждой итерации? Один раз в пять итераций?

Ответ таков: *мы не знаем!* Это все на совести планировщика. На вашей ОС с конкретной версией JVM и вашим же процессором вы можете получить совсем другие результаты.

Под управлением OS X 10.2 (Jaguar) с максимум пятью итерациями поток альфа выполняется до конца, затем завершает свою работу поток бета. Без гарантий, но очень последовательно.

При увеличении количества итераций до 25 и выше начинаются колебания. Поток альфа может не успеть завершить все 25 проходов, прежде чем планировщик сделает его работоспособным и даст шанс потоку бета.

## Существует и обратная сторона медали. Использование потоков может сопровождаться «проблемами» параллелизма.

Проблемы параллелизма вызывают «состояние гонки» (одна из классических ошибок проектирования). Такое состояние приводит к повреждению данных. Повреждение данных вызывает страх... Вы знаете, что происходит дальше.

Все это ведет к одному потенциально смертельному сценарию: два или больше потоков имеют доступ к *данным* одного объекта. Иными словами, методы, которые выполняются в двух разных стеках, вызывают, например, геттеры или сеттеры из одного и того же объекта в куче.

Вот уж действительно «левая рука не знает, что делает правая». Два потока, отрешенных от всего мира, вызывают в одиночку свои методы, и каждый думает, что именно он, единственный и неповторимый, имеет значение. В конце концов когда поток становится работоспособным (или заблокированным), он как будто теряет сознание. И когда работа возобновляется, поток не знает, что его останавливали.

# Брак разваливается. Можно ли спасти эту пару?

***Специальный выпуск Шоу доктора Стива***

[Стенограмма 42-го эпизода]

Добро пожаловать на Шоу доктора Стива!

Наша сегодняшняя история затрагивает две главные причины, по которым распадаются пары, — деньги и постель.

Проблемная пара, о которой мы будем говорить, — Райан и Моника, делят одну кровать и банковский счет. Но этому скоро придет конец, если мы не найдем решение. А проблема классическая — «человека два, а счет в банке один».

Райан и Моника: жертвы проблемы, когда «человека два, а счет в банке один».

Вот как Моника описывает сложившуюся ситуацию:

«Мы с Райаном договорились, что ни один из нас не будет превышать кредитный лимит. Поэтому, перед тем как снимать деньги со счета, мы *должны* проверять баланс. На первый взгляд, проще не бывает. Но однажды наши чеки внезапно перестали принимать, и мы узнали, что превысили кредит!

Я думала, что это невозможно и наша процедура была безопасной. Но потом кое-что случилось.

Райан задремал после того, как проверил баланс, но перед тем, как снял деньги. Проснувшись, он провел транзакцию без повторной проверки счета.

Райану нужно было $50, поэтому он проверил баланс на счету и увидел, что там находилось $100. Нет проблем. Он решил снять деньги. **Но перед этим он задремал!**

Пока он дремал, я пришла и тоже захотела снять деньги — $100. Я проверила баланс, увидела нужную сумму (ведь Райан еще не проснулся и не снял деньги) и подумала, что никаких проблем не будет. Я спокойно сняла деньги. Но потом Райан проснулся, снял *свои* $50, и мы внезапно оказались в минусе! Он даже не заметил, как задремал, поэтому и не проверил баланс перед тем, как выполнить свою транзакцию. Вы должны нам помочь, доктор Стив!»

Можно ли решить эту проблему? Или они обречены? Нельзя запретить Райану спать, но можно ли быть уверенным, что, пока он не проснется, Моника будет держаться подальше от банковского счета?

Воспользуйтесь моментом и подумайте над этим, а мы пока уйдем на рекламную паузу.

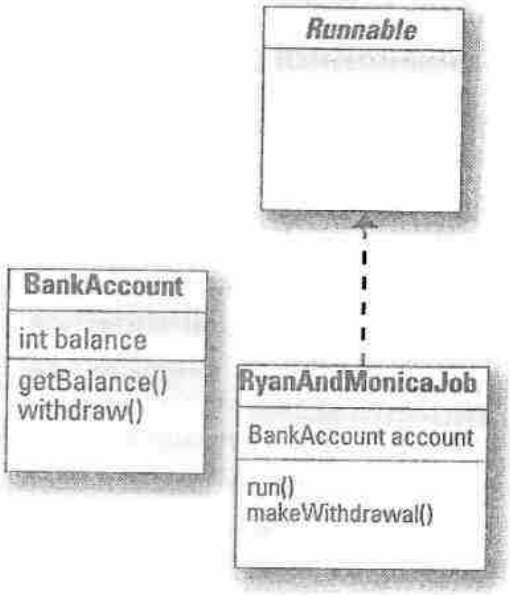
# Проблема Райана и Моники, представленная в виде кода

В следующем примере показано, что может случиться, когда *два* потока (Райан и Моника) делят *один* объект (банковский счет).

Код состоит из двух классов — BankAccount и MonicaAndRyanJob. Второй класс реализует интерфейс Runnable и описывает поведение, присущее и Райану, и Монике — проверка баланса и снятие денег со счета. Но, естественно, каждый поток будет засыпать *между* проверкой баланса и самой транзакцией.

Класс MonicaAndRyanJob имеет переменную экземпляра типа BankAccount, которая представляет их общий счет.

Код работает следующим образом.




**1 Создаем экземпляр класса MonicaAndRyanJob.**

Класс MonicaAndRyanJob реализует Runnable (задачу, которую нужно выполнить), а так как Райан и Моника делают одно и то же (проверяют баланс и снимают деньги), то и экземпляр нам нужен всего один.

```
RyanAndMonicaJob  theJob = new RyanAndMonicaJob();
```

**2 Создаем два потока с одним и тем же Runnable** (экземпляром MonicaAndRyanJob).

```
Thread one = new Thread(theJob);
Thread two = new Thread(theJob);
```

**3 Даем имена потокам и запускаем их.**

```
one.setName("Райан");
two.setName("Моника");
one.start();
two.start();
```

**4 Наблюдаем за тем, как оба потока выполняют метод run()** (проверяют баланс и снимают наличные).

Один поток представляет Райана, второй — Монику. Оба непрерывно проверяют баланс и затем снимают деньги, но только в том случае, если это безопасно!

```
if (account.getBalance() >= amount) {
   try {
      Thread.sleep(500);
   } catch(InterruptedException ex) {ex.printStackTrace(); }
}
```

**В методе run() мы делаем в точности то, что делали бы Райан и Моника — проверяем баланс и, если денег достаточно, снимаем нужную сумму.**

**Это должно предотвратить превышение кредитного лимита.**

**Разве что... Райан и Моника всегда будут дремать после проверки баланса, но перед выполнением транзакции.**

# Пример с Райаном и Моникой

```
class BankAccount {
   private int balance = 100;

   public int getBalance() {
      return balance;
   }
   public void withdraw(int amount) {
      balance = balance - amount;
   }
}

 public class RyanAndMonicaJob implements Runnable {

   private BankAccount account = new BankAccount();

   public static void main (String [] args) {
      RyanAndMonicaJob  theJob = new RyanAndMonicaJob();
      Thread one = new Thread(theJob);
      Thread two = new Thread(theJob);
      one.setName("Райан");
      two.setName("Моника");
      one.start();
      two.start();
    }

  public void run() {
   for (int x = 0; x < 10; x++) {
      makeWithdrawl(10);
      if (account.getBalance() < 0) {
        System.out.println("Превышение лимита!");
      }
    }
 }

  private void makeWithdrawal(int amount) {
     if (account.getBalance() >= amount) {
     System.out.println(Thread.currentThread().getName() + "собирается снять деньги");
      try {
        System.out.println(Thread.currentThread().getName() + "идет подремать");
        Thread.sleep(500);
      } catch(InterruptedException ex) {ex.printStackTrace(); }
      System.out.println(Thread.currentThread().getName() + "просыпается");
      account.withdraw(amount);
      System.out.println(Thread.currentThread().getName() + "заканчивает транзакцию");
      }
    else {
      System.out.println("Извините, для клиента" + Thread.currentThread().getName()) + " недостаточно денег";
    }
  }
}
```

Изначально на счету $100.

У нас будет только один экземпляр MonicaAndRyanJob. Это означает, что оба потока будут получать доступ к одному банковскому счету.

Экземпляр Runnable (задача).

Создаем два потока с одной задачей Runnable. Это значит, оба потока будут работать с одним экземпляром счета, который находится в классе Runnable.

В методе run() поток зацикливается и при каждой итерации пытается снять деньги со счета. После снятия он еще раз проверяет баланс, чтобы убедиться, что лимит не превышен.

Проверяем баланс и, если на счету недостаточно денег, просто выводим сообщение. Если же средств хватает, приостанавливаем поток, затем возобновляем его, чтобы завершить транзакцию (как это делал Райан).

Мы поместили сюда набор методов println и можем наблюдать, что происходит во время выполнения программы.

File Edit Window Help Visa

```
Райан собирается снять деньги
Райан идет подремать
Моника просыпается
Моника заканчивает транзакцию
Моника собирается снять деньги
Моника идет подремать
Райан просыпается
Райан заканчивает транзакцию
Райан собирается снять деньги
Райан идет подремать
Моника просыпается
Моника заканчивает транзакцию
Моника собирается снять деньги
Моника идет подремать
Райан просыпается
Райан заканчивает транзакцию
Райан собирается снять деньги
Райан идет подремать
Моника просыпается
Моника заканчивает транзакцию
Извините, для клиента Моника
недостаточно денег
Извините, для клиента Моника
недостаточно денег
Извините, для клиента Моника
недостаточно денег
Извините, для клиента Моника
недостаточно денег
Извините, для клиента Моника
недостаточно денег
Райан просыпается
Райан заканчивает транзакцию
```

Как это случилось?

**Метод makeWithdrawal() всегда проверяет баланс, прежде чем снять деньги, но мы все равно превысили кредитный лимит.**

**Вот пример такого сценария:**

Райан проверяет баланс и видит, что денег достаточно, а затем идет спать.

Тем временем Моника пришла проверить баланс. Она тоже видит, что баланс положительный. Моника не догадывается, что Райан проснется и завершит свою транзакцию.

Моника засыпает.

Райан просыпается и снимает деньги.

Моника просыпается и завершает свою транзакцию. Вот и проблема! В промежутке между тем, как она проверяла баланс и завершала транзакцию, Райан проснулся и забрал деньги со счета.

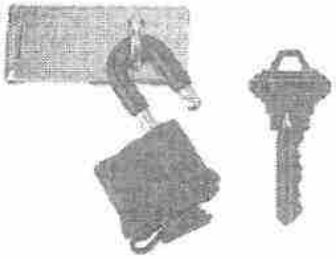
**Проверка баланса Моникой потеряла свою актуальность, так как Райан уже пребывал на пути от проверки к снятию.**

Монике следовало воздержаться от операций со счетом и подождать, пока Райан не проснется и не завершит свою транзакцию. И наоборот.

# Им нужна возможность блокировать счет!

## Блокировка работает следующим образом.

1. Существует замок, связанный с транзакциями на банковском счету (проверка баланса и вывод денег). Есть только один ключ, и он остается в замке, пока кто-нибудь не захочет получить доступ к счету.



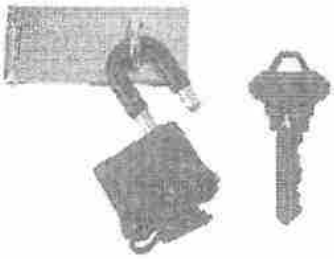
**Пока банковский счет никто не использует, доступ к его транзакциям открыт.**

2. Когда Райан хочет получить доступ к банковскому счету (чтобы проверить баланс и снять деньги), он запирает замок и кладет ключ в карман. Теперь никто другой не может получить доступ к счету, так как ключ один.

**Когда Райан хочет получить доступ к счету, он закрывает его на замок и забирает ключ с собой.**

3. **Райан хранит ключ в кармане, пока не закончит транзакцию.** Он владеет единственным ключом, поэтому Моника не может воспользоваться счетом (или чековой книжкой), пока Райан не откроет замок и не вернет ключ на место.

   Теперь, даже если Райан пойдет подремать, он может не сомневаться, что и после пробуждения баланс останется прежним, так как ключ все это время будет находиться у него!



**Закончив свои дела, Райан открывает замок и возвращает ключ на место. Теперь Моника (или опять Райан) может взять ключ и провести операции со счетом.**

# Нам нужно, чтобы метод makeWithdrawal() выполнялся как одна атомарная операция.

Мы должны быть уверены, что, запустив метод makeWithdrawal(), наш поток будет *иметь возможность завершить* его, прежде чем другой поток сможет начать работу со счетом.

Иными словами, нужны гарантии, что поток, который начал проверять баланс, сможет возобновить свою работу и закончить транзакцию *до того, как другой поток получит возможность проверить деньги на счету*!

Чтобы метод мог выполняться одновременно только в одном потоке, используйте ключевое слово **synchronized**.

Так вы сберегаете деньги. Вы не закрываете на замок сам банковский счет, а блокируете метод, который выполняет банковскую транзакцию. Поток может полностью завершить транзакцию, даже если будет приостановлен во время этого процесса!

Но если вы не блокируете банковский счет, что же блокируется *на самом деле*? Объект Runnable? Целый поток?

Мы рассмотрим это на следующей странице. Хотя с точки зрения кода все выглядит просто — достаточно добавить в объявление метода модификатор synchronized.

**Ключевое слово synchronized означает, что для работы с отмеченным кодом требуется ключ. Чтобы защитить свои данные (например, банковский счет), синхронизируйте методы, которые работают с этими данными.**

```
private synchronized void makeWithdrawal(int amount) {

    if (account.getBalance() >= amount) {
       System.out.println(Thread.currentThread().getName() + "собирается снять деньги");
       try {
          System.out.println(Thread.currentThread().getName() + "идет подремать");
          Thread.sleep(500);
       } catch(InterruptedException ex) {ex.printStackTrace(); }
       System.out.println(Thread.currentThread().getName() + "просыпается");
       account.withdraw(amount);
       System.out.println(Thread.currentThread().getName() + "заканчивает транзакцию");
    } else {
       System.out.println("Извините, для клиента" + Thread.currentThread().getName() + " недостаточно денег");
    }
}
```

Примечание для читателей, подкованных в физике: используя слово «атомарный», мы не учитываем наличие любых субатомных частиц. Когда вы слышите это слово в контексте потоков или транзакций, думайте о Ньютоне, а не об Эйнштейне. Это ведь не мы придумали. Будь наша воля, мы бы использовали принцип неопределенности Гейзенберга для всего, что связано с потоками.

# Использование блокировки объектов

У каждого объекта есть «замок» (блокировка). Большую часть времени он открыт, и вы можете представить себе виртуальный ключ, который вставлен в этот замок. Он запирается только при выполнении синхронизированных методов. Если у объекта есть один или несколько синхронизированных методов, ***поток может выполнять их лишь при наличии ключа к объекту***!

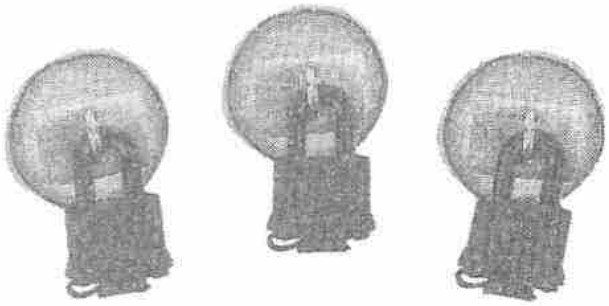
Такие замки распространяются не на каждый *метод* отдельно, а на весь *объект*. Если у объекта есть два синхронизированных метода, значит, два потока не смогут выполнять *любой* из них, а не только какой-то один.

Подумайте об этом. Если вы создали несколько методов, которые потенциально могут влиять на переменные объекта, они все должны быть защищены ключевым словом synchronized.

Цель синхронизации — защита критически важных данных. Но помните, что вы не блокируете данные, а синхронизируете методы, которые с ними *работают*.

Что же происходит, когда поток проходит по своему стеку вызовов (начиная с метода run()) и неожиданно натыкается на синхронизированный метод? Поток понимает, что, прежде чем запустить метод, нужно получить ключ от объекта. Он ищет ключ (все это происходит в недрах JVM; в Java нет API для доступа к блокировке объектов) и, если находит, забирает его, а затем выполняет метод.

Начиная с этого момента поток хранит ключ как зеницу ока. Он не отдаст его, пока не закончит выполнять синхронизированный метод. Таким образом, пока ключ у него, другой поток не сможет начать выполнять *ни один* синхронизированный метод объекта.



У каждого объекта в Java есть замок. У каждого замка есть только один ключ.

Большую часть времени замок открыт и никому нет до него дела.

Но если у объекта есть синхронизированные методы, поток может выполнять их, когда доступен ключ для данного объекта. Иначе говоря, пока другой поток не завладел единственным ключом.

# Ужасная проблема «последнего изменения»

Вот еще одна классическая проблема параллелизма, пришедшая из мира баз данных. Она тесно переплетается с историей Райана и Моники. Проблему «последнего изменения» можно понять из следующего процесса.

Шаг 1. Получаем баланс на счету:

```
int i = balance;
```

Шаг 2. Добавляем к балансу единицу:

```
balance = i + 1;
```

Хитрость в том, что мы заставили компьютер изменить баланс в два шага. На практике это делается одним выражением:

```
balance++;
```

*Два* шага нужны для того, чтобы акцентировать внимание на проблеме, присущей неатомарным процессам. Представьте, что вместо обычного получения баланса и его инкрементирования у нас есть несколько более сложных шагов, которые нельзя заменить одним выражением.

Проблема «последнего изменения» заключается в том, что у нас есть два потока, каждый из которых пытается инкрементировать баланс.

```
class TestSync implements Runnable {

    private int balance;

    public void run() {
       for(int i = 0; i < 50; i++) {
        increment();
        System.out.println("Баланс равен" + balance);
      }
    }

    public void increment() {
       int i = balance;
       balance = i + 1;
    }
}

public class TestSyncTest {
   public static void main (String[] args) {
     TestSync job = new TestSync();
     Thread a = new Thread(job);
     Thread b = new Thread(job);
     a.start();
     b.start();
   }
}
```

Каждый поток инкрементирует баланс 50 раз.

Вот где ключевой момент! Мы инкрементируем баланс, добавляя единицу к значению, которое было на момент чтения, а не увеличивая текущее значение, каким бы оно ни было.

# Запустим этот код...

① Некоторое время работает поток A.

Помещаем значение баланса в переменную *i*.
Баланс равен нулю, поэтому *i* тоже хранит 0.
Делаем значение баланса равным результату выражения *i* + 1.
Сейчас баланс равен 1.
Помещаем значение баланса в переменную *i*.
Баланс равен 1, поэтому переменная *i* сейчас равна 1.
Делаем значение баланса равным результату выражения *i* + 1.
Сейчас баланс равен 2.

② Некоторое время работает поток B.

Помещаем значение баланса в переменную *i*.
Баланс равен 2, поэтому *i* сейчас хранит 2.
Делаем значение баланса равным результату выражения *i* + 1.
Сейчас баланс равен 3.
Помещаем значение баланса в переменную *i*.
Баланс равен 3, поэтому переменная *i* сейчас равна 3.
*[Теперь поток B возвращается в работоспособное состояние, не успев установить значение баланса, равное 4]*

③ Поток A возвращается к работе с того места, на котором его прервали.

Помещаем значение баланса в переменную *i*.
Баланс равен 3, поэтому *i* сейчас хранит 3.
Делаем значение баланса равным результату выражения *i* + 1.
Сейчас баланс равен 4.
Помещаем значение баланса в переменную *i*.
Баланс равен 4, поэтому *i* сейчас равно 4.
Делаем значение баланса равным результату выражения *i* + 1.
Сейчас баланс равен 5.

④ Поток B возвращается к работе с того места, на котором его прервали.

Делаем значение баланса равным результату выражения *i* + 1.
**Сейчас баланс равен 4.** ← Черт побери!!!

Поток A обновил баланс до 5, но B вернулся и нивелировал изменение, как будто его и не было.

**Мы потеряли последние изменения, которые сделал поток A! Ранее поток B «прочитал» значение баланса, и, проснувшись, продолжил выполняться так, будто его работа никогда не прерывалась.**

# Сделайте метод increment() атомарным. Синхронизируйте его!

Синхронизация метода increment() решает проблему «последнего изменения», так как при этом два отдельных шага превращаются в единое целое.

```
public synchronized void increment() {
    int i = balance;
    balance = i + 1;
}
```

**Мы должны быть уверены, что, начав выполнять метод, поток доведет дело до конца (в виде единого атомарного процесса), прежде чем другой поток сможет работать с этим методом.**

---

Это не

## глупые вопросы

**В: Похоже, имеет смысл синхронизировать все подряд, чтобы код был безопасен с точки зрения потоков.**

**О:** Нет, это плохая идея. Синхронизация имеет свои побочные эффекты. Во-первых, синхронизированный метод обладает определенной избыточностью. Иными словами, когда выполнение кода доходит до такого метода, снижается производительность (хотя, скорее всего, вы этого даже не замечаете), так как дается ответ на вопрос. Доступен ли ключ?

Во-вторых, синхронизированный метод может замедлить вашу программу, потому что синхронизация ограничивает распараллеливание. Метод вынуждает другие потоки выстраиваться в ряд и ждать своей очереди. Эта проблема может быть неактуальна для вашего кода, но ее нужно учитывать.

В-третьих, синхронизированные методы могут привести к взаимной блокировке, а это самое страшное (см. страницу 546)!

Хорошее практическое решение — синхронизация минимально необходимого количества методов, которые не могут без этого обойтись. Фактически вы можете синхронизировать не только методы, но и более мелкие участки кода. В книге мы этого не рассматриваем, но ключевое слово synchronized можно применять даже к одному или нескольким выражениям.

Метод doStuff() не нуждается в синхронизации, поэтому мы не указываем ключевое слово synchronized при объявлении.

```
public void go() {
    doStuff();

    synchronized(this) {
        criticalStuff();
        moreCriticalStuff();
    }
}
```

Теперь только два этих вызова сгруппированы в одну атомарную сущность. Используя ключевое слово synchronized внутри метода, а не при его объявлении, вы должны предоставить аргумент в виде объекта, чей ключ необходимо получить потоку.

Хотя существуют и другие варианты, вы почти всегда будете выполнять синхронизацию с текущим объектом (this). Именно его вы бы заблокировали при синхронизации целого метода.

① Некоторое время работает поток A.

Пытаемся войти в метод increment().

Метод синхронизирован, поэтому **берем ключ** для этого объекта.

Помещаем значение баланса в переменную *i*.

Баланс равен нулю, поэтому i тоже хранит 0.

Делаем значение баланса равным результату выражения *i* + 1.

Сейчас баланс равен 1.

**Возвращаем ключ** (поток закончил выполнение метода increment()).

Опять входим в метод increment() и **получаем ключ**.

Помещаем значение баланса в переменную *i*.

Баланс равен 1, поэтому *i* тоже хранит 1.

*[Теперь поток A возвращается в работоспособное состояние, но он не успел завершить синхронизированный метод, поэтому ключ остается у него]*

② Для выполнения выбран поток B.

Пытаемся войти в метод increment(). Метод синхронизирован, поэтому нужно получить ключ.

**Ключ недоступен.**

*[Теперь поток B ждет, пока освободится ключ для этого объекта]*

③ Поток A возвращается к работе с того места, на котором его прервали (помните, ключ по-прежнему у него).

Делаем значение баланса равным результату выражения *i* + 1.

Сейчас баланс равен 2.

**Возвращаем ключ.**

*[Теперь поток A возвращается в работоспособное состояние, но он завершил выполнение метода increment(), поэтому ключ ему больше не нужен]*

④ Для выполнения выбран поток B.

Пытаемся войти в метод increment(). Метод синхронизирован, поэтому необходимо получить ключ.

На этот раз ключ доступен. Берем его.

Делаем значение баланса равным результату выражения *i* + 1.

*[продолжаем выполнение...]*

# Темная сторона синхронизации

Будьте осторожны при синхронизации кода, так как для программы нет ничего опаснее, чем взаимное блокирование. Взаимное блокирование двух потоков происходит тогда, когда каждый из них содержит ключ, который нужен другому потоку. Нет никакой возможности выйти из этой ситуации, поэтому потоки просто остановятся и будут ждать.

Если вы знакомы с базами данных или другими серверными приложениями, то наверняка уловили суть проблемы — базы данных часто обладают механизмом блокирования наподобие синхронизации. Настоящая система по управлению транзакциями способна справиться с отдельными случаями взаимного блокирования. Она может распознать их, например, когда две транзакции выполняются слишком долго. Но в отличие от Java такие серверные приложения умеют делать «откат», когда состояние данных возвращается к моменту до выполнения транзакции (атомарность в действии).

У Java нет механизмов для противодействия взаимному блокированию. JVM даже не будет *знать*, что это случилось. Будьте осторожны при проектировании, так как вся ответственность лежит на вас. Если вам придется часто писать многопоточный код, то ознакомьтесь с книгой «Java Threads» Скотта Оакса (Scott Oaks) и Генри Вонга (Henry Wong). В ней содержатся советы, которые помогут избежать взаимного блокирования. Одна из основных рекомендаций — уделять внимание порядку, в котором запускаются потоки.

**Для взаимного блокирования достаточно двух объектов и двух потоков.**

## Пример ситуации с взаимным блокированием.

## КЛЮЧЕВЫЕ МОМЕНТЫ

- Статический метод Thread.sleep() вынуждает поток приостановить свое выполнение как минимум на тот промежуток времени, что указан в качестве аргумента. Thread.sleep(200) «усыпит» поток на 200 мс.
- Метод sleep выбрасывает проверяемое исключение (InterruptedException), поэтому все его вызовы должны быть заключены в блоки try/catch либо объявлены.
- С помощью sleep() вы можете увеличить шансы каждого потока на выполнение, хотя нет гарантии, что при пробуждении поток сразу продолжит свою работу. В большинстве случаев вызовы sleep() с правильно подобранным хронометражем полностью решают проблему переключения потоков.
- Вы можете дать имя потоку с помощью метода setName(). У всех потоков есть имена по умолчанию, но благодаря явно заданному имени вам будет легче их отслеживать, особенно если вы выполняете отладку с использованием методов print.
- У вас могут быть серьезные проблемы, если несколько потоков получают доступ к одному объекту в куче.
- Доступ к одному объекту из разных потоков может привести к повреждению данных. Например, один поток может приостановить свою работу прямо во время манипуляций с критически важным состоянием объекта.
- Чтобы сделать объекты потокобезопасными, вам следует определиться, какие выражения должны выполняться в виде единого атомарного процесса. Иными словами, необходимо решить, какие методы должны быть выполнены до конца, прежде чем другой поток сможет запускать их из того же объекта.
- Используйте ключевое слово synchronized при объявлении методов, если не хотите, чтобы они выполнялись в нескольких потоках одновременно.
- У каждого объекта есть один замок (блокировка) и один ключ к нему. Большую часть времени этот замок не важен для вас; он имеет значение только тогда, когда у его объекта есть синхронизированные методы.
- При попытке запустить синхронизированный метод поток должен получить ключ для объекта, которому этот метод принадлежит. Если ключ недоступен (потому что другой поток его уже забрал), поток останавливается и ждет, пока не сможет получить ключ.
- Даже если у объекта несколько синхронизированных методов, он все равно владеет только одним ключом. Если поток начал выполнять какой-либо синхронизированный метод объекта, то остальные методы становятся недоступными для любого другого потока. Благодаря этому ограничению вы можете защитить данные, с которыми работают синхронизированные методы.

# Новая улучшенная версия SimpleChatClient

Вспомните, с чего начиналась эта глава: мы создали приложение SimpleChatClient, которое может *отправлять* исходящие сообщения на сервер, но не может ничего принимать. Не забыли? Именно из-за этого мы начали разговор о потоках, так как нужен был способ выполнять два действия одновременно: отправлять сообщение *на* сервер (взаимодействуя с GUI) и в то же время считывать входящие сообщения, полученные *от* сервера, отображая их в прокручиваемой текстовой области.

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleChatClient {

    JTextArea incoming;
    JTextField outgoing;
    BufferedReader reader;
    PrintWriter writer;
    Socket sock;

    public static void main(String[] args) {
        SimpleChatClient client = new SimpleChatClient();
        client.go();
    }

    public void go() {

        JFrame frame = new JFrame("Ludicrously Simple Chat Client");
        JPanel mainPanel = new JPanel();
        incoming = new JTextArea(15,50);
        incoming.setLineWrap(true);
        incoming.setWrapStyleWord(true);
        incoming.setEditable(false);
        JScrollPane qScroller = new JScrollPane(incoming);
        qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
        outgoing = new JTextField(20);
        JButton sendButton = new JButton("Send");
        sendButton.addActionListener(new SendButtonListener());
        mainPanel.add(qScroller);
        mainPanel.add(outgoing);
        mainPanel.add(sendButton);
        setUpNetworking();

        Thread readerThread = new Thread(new IncomingReader());
        readerThread.start();

        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        frame.setSize(400,500);
        frame.setVisible(true);

    } // Закрываем go
```

Эта глава когда-нибудь закончится. Но не сейчас...

В основном это код, который вы видели ранее. Ничего особенного, если не считать выделенного фрагмента, в котором мы запускаем новый поток для «считывания».

Мы запускаем новый поток, используя вложенный класс в качестве Runnable (задачи). Работа потока заключается в чтении данных с сервера через сокет и выводе любых входящих сообщений в прокручиваемую текстовую область.

```
private void setUpNetworking() {

    try {
       sock = new Socket("127.0.0.1", 5000);
       InputStreamReader streamReader = new InputStreamReader(sock.getInputStream());
       reader = new BufferedReader(streamReader);
       writer = new PrintWriter(sock.getOutputStream());
       System.out.println("networking established");
    } catch(IOException ex) {
       ex.printStackTrace();
    }
  } // Закрываем setUpNetworking
```

Мы используем сокет для получения входящего и исходящего потоков. Исходящий поток уже задействован для отправки данных, но теперь к нему добавился входящий поток, поэтому наш объект Thread может получать сообщения от сервера.

```
public class SendButtonListener implements ActionListener {
   public void actionPerformed(ActionEvent ev) {
       try {
          writer.println(outgoing.getText());
          writer.flush();

       } catch(Exception ex) {
          ex.printStackTrace();
       }
       outgoing.setText("");
       outgoing.requestFocus();
   }
 }  // Закрываем вложенный класс
```

Здесь ничего нового. Когда пользователь нажимает кнопку Send (Отправить), содержимое текстового поля отправляется на сервер.

```
public class IncomingReader implements Runnable {
   public void run() {
       String message;
       try {

          while ((message = reader.readLine()) != null) {
             System.out.println("read " + message);
             incoming.append(message + "\n");

          } // Закрываем цикл while
       } catch(Exception ex) {ex.printStackTrace();}
    } // Закрываем run
 } // Закрываем вложенный класс

} // Закрываем внешний класс
```

Это работа, которую выполняет поток! В методе run() поток входит в цикл (пока ответ сервера будет равняться null), считывает за раз одну строку и добавляет ее в прокручиваемую текстовую область (используя в конце символ переноса строки).

**Код, готовый к употреблению**

# Очень-очень простой чат-сервер

Вы можете использовать этот код для обеих версий чат-клиента. Чтобы обратить ваше внимание на ключевые моменты в коде, мы избавились от многих фрагментов, которые обязательно понадобятся настоящему серверу. Иными словами, код работает, но сломать его можно сотней различных способов. Если после прочтения этой книги вам захочется решить по-настоящему серьезную задачу, то можете вернуться сюда и сделать этот код более надежным.

Еще одно упражнение, которое вы можете выполнить прямо сейчас: прокомментируйте код. Сами выясните, как он работает, вместо того чтобы читать наши объяснения — это поможет вам лучше во всем разобраться. Но, поскольку код приготовлен заранее, не будет ничего страшного, если кое-что в нем вы не поймете. Он нужен только для того, чтобы поддерживать обе версии чат-клиента.

**Чтобы запустить клиент для чата, необходимо иметь две командные строки. Сначала нужно запустить сервер в одном терминале, а затем загрузить клиент в другом.**

```
import java.io.*;
import java.net.*;
import java.util.*;

public class VerySimpleChatServer {

    ArrayList clientOutputStreams;

    public class ClientHandler implements Runnable {
         BufferedReader reader;
         Socket sock;

         public ClientHandler(Socket clientSocket) {
           try {
             sock = clientSocket;
             InputStreamReader isReader = new InputStreamReader(sock.getInputStream());
             reader = new BufferedReader(isReader);

           } catch(Exception ex) {ex.printStackTrace();}
          } // Закрываем конструктор

        public void run() {
           String message;
           try {
             while ((message = reader.readLine()) != null) {
                System.out.println("read " + message);
                tellEveryone(message);

                } // Закрываем while
           } catch(Exception ex) {ex.printStackTrace();}
       } // Закрываем run
   } // Закрываем вложенный класс
```

```
    public static void main (String[] args) {
         new VerySimpleChatServer().go();
    }

    public void go() {
       clientOutputStreams = new ArrayList();
       try {
         ServerSocket serverSock = new ServerSocket(5000);

         while(true) {
            Socket clientSocket = serverSock.accept();
            PrintWriter writer = new PrintWriter(clientSocket.getOutputStream());
            clientOutputStreams.add(writer);

            Thread t = new Thread(new ClientHandler(clientSocket));
            t.start();
            System.out.println("got a connection");
         }

       } catch(Exception ex) {
         ex.printStackTrace();
       }
    } // Закрываем go

    public void tellEveryone(String message) {

       Iterator it = clientOutputStreams.iterator();
       while(it.hasNext()) {
          try {
             PrintWriter writer = (PrintWriter) it.next();
             writer.println(message);
             writer.flush();
          } catch(Exception ex) {
              ex.printStackTrace();
          }

       } // Конец цикла while

    } // Закрываем tellEveryone
} // Закрываем класс
```

**В: Что насчет защиты состояния статических переменных? Можно ли использовать синхронизацию в сочетании со статическими методами, которые меняют значения статических переменных?**

О: Да! Не забывайте, что статические методы выполняются в контексте класса, а не для каждого его отдельного экземпляра. Вас, должно быть, интересует, замок от какого объекта будет использоваться в статическом методе. В конце концов у такого класса может не быть ни одного экземпляра. К счастью, аналогично *объектам*, загружаемые классы тоже имеют замки (блокировки). Это означает, что если у вас есть три объекта Dog в куче, то будет четыре связанных с ними замка. Три из них принадлежат экземплярам Dog и один — непосредственно классу Dog. Когда вы синхронизируете статический метод, Java использует замок, принадлежащий классу. Чтобы запустить *один из* статических синхронизированных методов, потоку нужен ключ от класса, которому эти методы принадлежат.

**В: Какие приоритеты имеют потоки? Я слышал, что есть способ управлять их раписанием.**

О: Приоритеты потоков *могут* помочь вам повлиять на планировщика, но они тоже не дают никаких гарантий. Приоритеты потоков — это числовые значения, которые говорят планировщику (если они ему интересны), насколько важен для вас конкретный поток. Как правило, планировщик приостанавливает выполнение потока, если есть другой работоспособный поток с более высоким приоритетом. Но еще раз повторим: нет никаких гарантий. Мы рекомендуем вам использовать приоритеты только в том случае, если нужно повлиять на *производительность*. Никогда не допускайте, чтобы от них зависела правильность работы вашей программы..

## Это не глупые вопросы

**В: Почему бы просто не синхронизировать все геттеры и сеттеры класса с данными, которые мы хотим защитить? Например, вместо синхронизации метода makeWithdrawal() мы могли бы синхронизировать методы checkBalance() и withdraw() из класса BankAccount.**

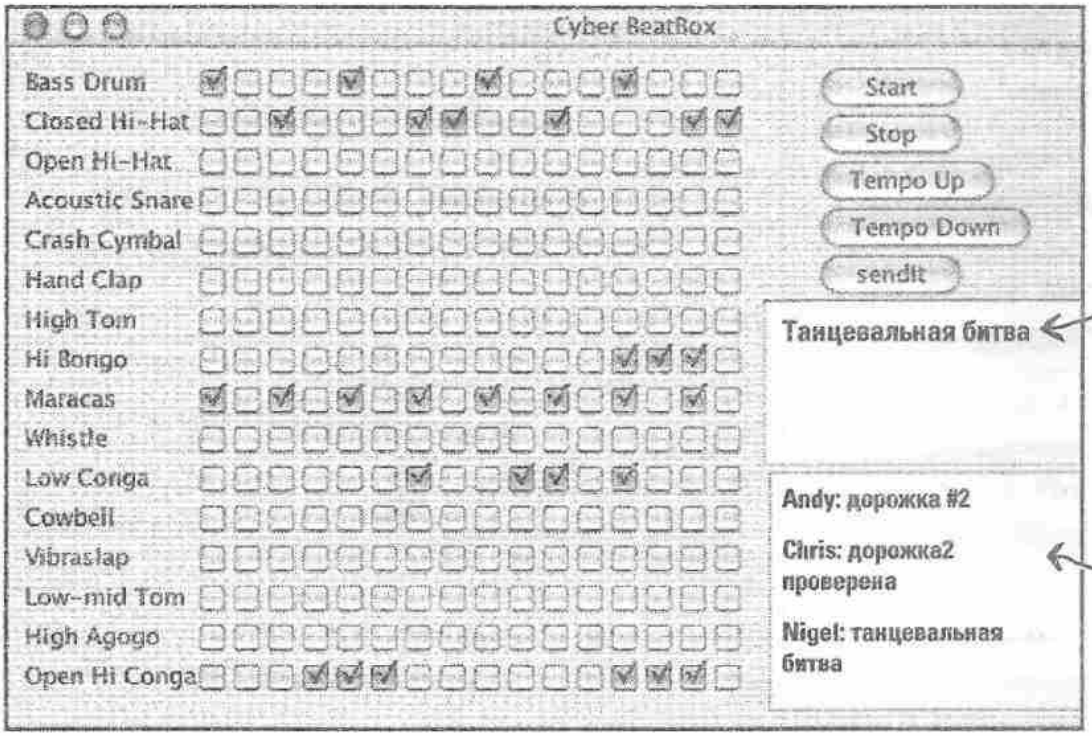
О: Вообще-то *следовало бы* синхронизировать эти методы, чтобы другие потоки не смогли получить к ним доступ окольными путями. Но мы себя этим не утруждали, так как в нашем примере нет другого кода для доступа к счету.

Тем не менее только синхронизации геттеров и сеттеров (в данном случае это checkBalance() и withdraw()) недостаточно. Помните, цель синхронизации — заставить участки кода работать атомарно. Иными словами, нас беспокоят не отдельные методы, а методы, которые ***выполняются в несколько шагов***! Подумайте об этом. Если мы не синхронизируем метод makeWithdrawal(), то Райану придется сразу завершать его и возвращать ключ, как только он проверит баланс (вызывая метод checkBalance())!

Конечно, он опять получит ключ, когда проснется, и сможет вызвать метод withdraw(), но в таком случае возникает проблема, с которой мы уже сталкивались до синхронизации! Райан может проверить баланс и пойти спать, но прежде чем он проснется и завершит свою транзакцию, Моника тоже может проверить баланс.

В такой ситуации хорошее решение — синхронизация всех методов для доступа к данным, так как это не позволит вмешиваться в процесс другим потокам. Но вам по-прежнему необходимо синхронизировать методы с выражениями, которые должны выполняться как единое целое.

# Кухня кода



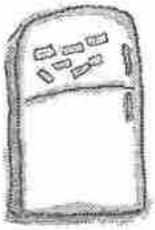

При нажатии кнопки Send (Отправить) ваши сообщения наряду с текущим музыкальным шаблоном отправляются другим участникам.

Входящие сообщения от участников чата. Щелкните на сообщении, чтобы загрузить шаблон, который идет вместе с ним. Чтобы проиграть его, нажмите кнопку Start (Играть).

***Это последняя версия программы BeatBox!***

***Программа подключается к простому музыкальному серверу, который позволяет отправлять и принимать музыкальные шаблоны от других клиентов.***

***Код довольно объемный, поэтому в полном виде он содержится только в Приложении А.***



# Магнитики с кодом

Части рабочего Java-приложения разбросаны по всему холодильнику. Можете ли вы восстановить из фрагментов работоспособную программу, которая выведет на экран текст, приведенный ниже? Некоторые фигурные скобки упали на пол; они настолько маленькие, что их нельзя поднять. Можете добавлять столько скобок, сколько понадобится!

```
public class TestThreads {
```

```
class Accum {
```

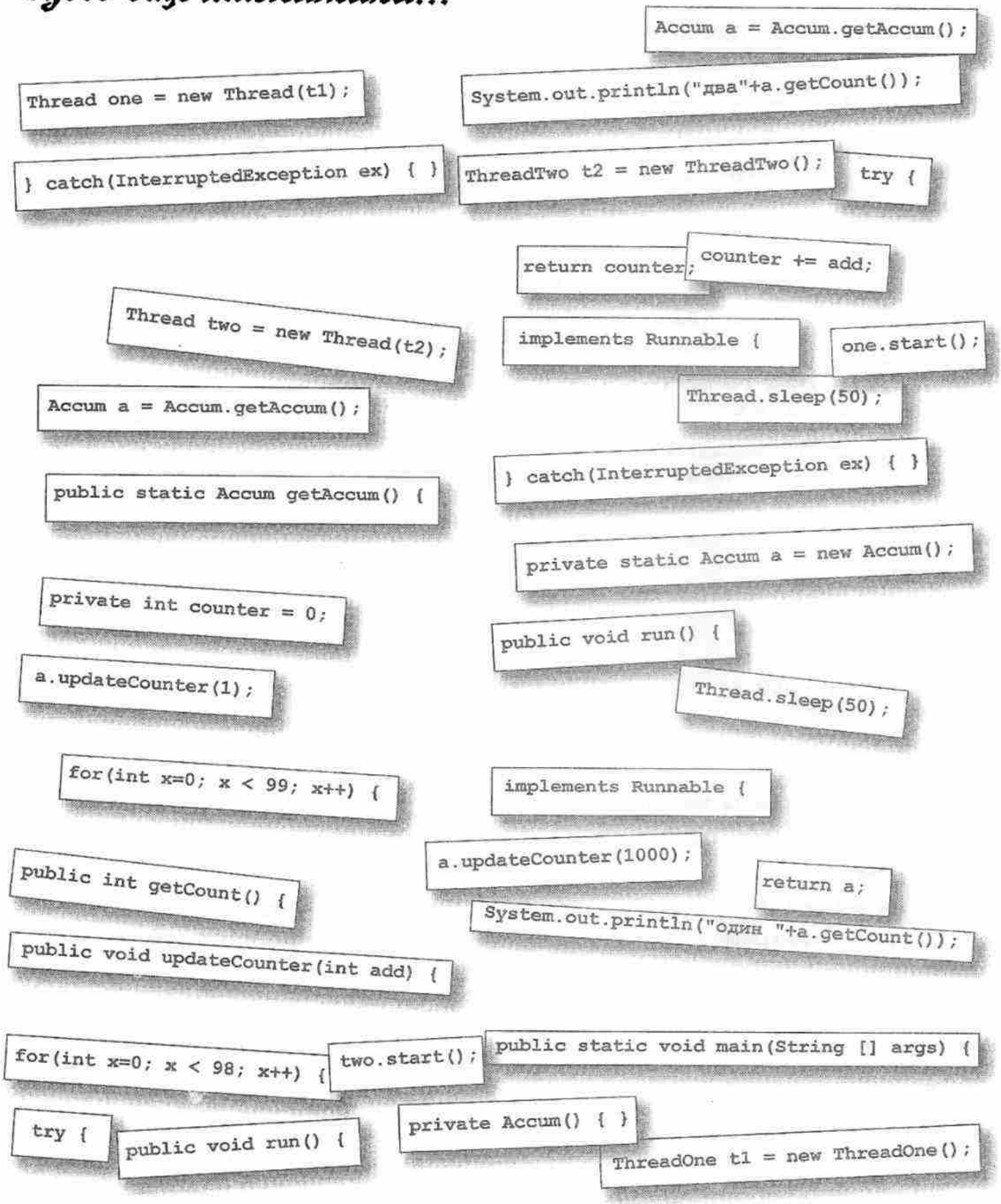
```
class ThreadOne
```

```
class ThreadTwo
```

```
File Edit Window Help Sewing
% java TestThreads
Один 98098
Два 98099
```

**Призовой вопрос:** Как вы думаете, почему мы использовали именно такие модификаторы для класса Accum?

# Здесь еще магнитики...



```
public class TestThreads {
  public static void main(String [] args) {
    ThreadOne t1 = new ThreadOne();
    ThreadTwo t2 = new ThreadTwo();
    Thread one = new Thread(t1);
    Thread two = new Thread(t2);
    one.start();
    two.start();
  }
}

class Accum {
  private static Accum a = new Accum();
  private int counter = 0;

  private Accum() { }

  public static Accum getAccum() {
    return a;
  }

  public void updateCounter(int add) {
    counter += add;
  }

  public int getCount() {
    return counter;
  }
}

class ThreadOne implements Runnable {
  Accum a = Accum.getAccum();
  public void run() {
    for(int x=0; x < 98; x++) {
      a.updateCounter(1000);
      try {
        Thread.sleep(50);
      } catch(InterruptedException ex) { }
    }
    System.out.println("one "+a.getCount());
  }
}
```

Создаем статический экземпляр класса Accum.

Приватный конструктор.

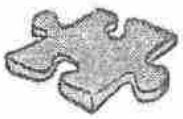
# Ответы

Потоки из двух разных классов обновляют один и тот же объект — единственный экземпляр Accum. В строке

private static Accum a = new Accum();

создается статический экземпляр этого класса (помните, статический значит один на весь класс), причем у Accum есть приватный конструктор, поэтому никто другой не может создать объект такого класса. Эти приемы (приватный конструктор и статический геттер), используемые вместе, позволяют создать нечто под названием «Синглтон» — шаблон проектирования в ООП. Его цель — ограничить количество экземпляров объекта, которые могут присутствовать в приложении. Как правило, Синглтон подразумевает наличие лишь одного экземпляра (отсюда и название), но вы вольны решать, как ограничивать создание объектов.

```
class ThreadTwo implements Runnable {
  Accum a = Accum.getAccum();
  public void run() {
    for(int x=0; x < 99; x++) {
      a.updateCounter(1);
      try {
        Thread.sleep(50);
      } catch(InterruptedException ex) { }
    }
    System.out.println("two "+a.getCount());
  }
}
```



## Происшествие внутри шлюза

Присоединившись к совещанию команды на борту корабля, Сара бросила взгляд на портал, освещенный лучами утреннего солнца, восходящего над Индийским океаном. Несмотря на то что корабельный конференц-зал был невероятно тесным, вид увеличивающегося бело-синего полумесяца, коим представлялась планета с такого расстояния, вызвал у Сары трепет и восхищение.

Сегодняшнее совещание было посвящено системам управления орбитальными шлюзами. Поскольку строительство подходило к концу, количество запланированных полетов разительно выросло, как и пассажиропоток на шлюзах корабля в обоих направлениях. «Доброе утро, Сара, — сказал Том. — Ты как раз вовремя: мы только что начали подробно рассматривать проект».

Пятиминутный детектив

«Как вам всем известно, — продолжил Том, — каждый шлюз на входе и выходе оснащен графическими терминалами, приспособленными для космических нужд. Когда астронавты прибывают на корабль или покидают его, они используют эти терминалы, чтобы открыть или закрыть шлюз». Сара зевнула: «Том, ты не мог бы нам напомнить, какие последовательности методов применяются при входе и выходе?» Том покраснел и, оттолкнувшись, «поплыл» к доске. «Для начала вот псевдокод последовательности методов, которая отвечает за выход», — сказал он и начал быстро писать на доске.

```
orbiterAirlockExitSequence()
  verifyPortalStatus();
  pressurizeAirlock();
  openInnerHatch();
  confirmAirlockOccupied();
  closeInnerHatch();
  decompressAirlock();
  openOuterHatch();
  confirmAirlockVacated();
  closeOuterHatch();
```

«Чтобы быть уверенными, что последовательность не будет прервана, мы должны синхронизировать все методы, вызываемые из orbiterAirlockExitSequence(), — объяснил Том. — Нам бы не хотелось, чтобы возвращающийся астронавт случайно столкнулся с давним приятелем, который не успел натянуть свои космические штаны!»

Пока Том вытирал доску, его коллеги тихо посмеивались. Но кое-что в этом объяснении показалось Саре неправильным, и, как только Том начал расписывать псевдокод входящей последовательности, она, наконец, поняла, что именно было не так. «Погоди минутку, Том!, — выкрикнула Сара. — Мне кажется, я нашла большой изъян в проекте исходящей последовательности, давай вернемся и пересмотрим ее заново, это может быть очень важно!»

***Почему Сара прервала собрание? Какие подозрения у нее появились?***

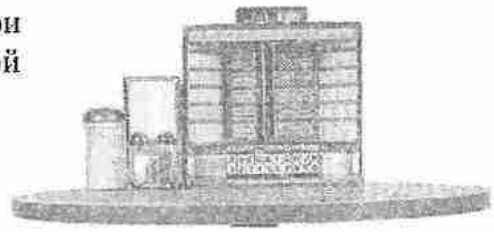
## Что было на уме у Сары?

Сара поняла, что исходящая последовательность сможет работать без прерываний только в том случае, если весь метод **`orbiterAirlockExitSequence()`** будет синхронизирован. При имеющемся же проекте возвращающийся астронавт может прервать процесс выхода! Исходящий поток нельзя остановить посреди работы любого из низкоуровневых методов, но в его работу можно вмешаться между их вызовами. Сара понимала, что вся последовательность должна выполняться как единое целое, и если метод **`orbiterAirlockExitSequence()`** будет синхронизирован, его невозможно будет прервать.

# Структуры данных

**Сортировка в Java — проще простого.** У вас уже есть все необходимые инструменты для сбора данных и управления ими, поэтому не нужно писать собственные алгоритмы для сортировки (если только вы не читаете эту книгу, сидя на лекции по компьютерной дисциплине, — в таком случае, поверьте нам, вы еще вдоволь насладитесь написанием сортировочного кода, в то время как мы будем вызывать готовые методы из Java API). Фреймворк для работы с коллекциями в Java (Java collections framework) содержит структуры данных, которые должны подойти практически для всего, что вам может понадобиться. Хотите получить список, в который можно добавлять элементы? Хотите найти что-нибудь по имени? Хотите создать список, который автоматически убирает все дубликаты? Желаете отсортировать своих сослуживцев по количеству ударов, которые они нанесли вам в спину, или домашних любимцев по количеству трюков, которые они выучили? Читайте главу...

# Отслеживание популярности песен в музыкальном автомате



Поздравляем, у вас новая работа — обслуживать музыкальный автомат в закусочной. В нем нет языка Java, но каждый раз при проигрывании песни информация о ней добавляется в простой текстовый файл.

Ваша задача — с помощью этих данных отслеживать популярность песен, генерировать отчеты и управлять списками воспроизведения. Вам не нужно создавать приложение с нуля (этим занимаются другие разработчики/официанты), но вы ответственны за управление и сортировку информации внутри Java-программы. И поскольку владелец закусочной не хочет применять базы данных, вам придется работать только с коллекциями, которые хранятся в памяти. Все, что вы получаете, — это файл, куда музыкальный автомат записывает информацию. Вы должны его извлечь.

Вы уже знаете, как читать и анализировать файлы. К тому же вы сортировали данные внутри ArrayList.

**SongList.txt**

```
Pink Moon/Nick Drake
Somersault/Zero 7
Shiva Moon/Prem Joshua
Circles/BT
Deep Channel/Afro Celts
Passenger/Headmix
Listen/Tahiti 80
```

*Это файл, куда музыкальный автомат записывает информацию. Ваш код должен сначала прочитать файл, а затем обработать данные о композициях.*

## Задача 1. Сортировать песни в алфавитном порядке.

У вас есть файл со списком песен (по одной на каждую строку). Название песни и имя исполнителя разделены слешем. Таким образом, не должно быть проблем с разбором файла и записью всех композиций в ArrayList.

Сначала список будет содержать только названия, так как вашего начальника больше ничего не интересует.

Но, как видите, данные размещены совсем не в алфавитном порядке... Как это исправить?

Вы знаете, что в ArrayList элементы хранятся в том порядке, в котором их добавляли, поэтому никакой «упорядоченности» здесь не наблюдается, разве что... Может, в классе ArrayList есть метод sort()?

# Вот что мы имеем на текущий момент без сортировки:

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

    ArrayList<String> songList = new ArrayList<String>();

    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        getSongs();
        System.out.println(songList);
    }

    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line =  null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        songList.add(tokens[0]);
    }

}
```

Мы будем хранить названия песен в списке ArrayList, состоящем из строковых данных.

Метод, который загружает файл и выводит содержимое songList (ArrayList).

Здесь нет ничего особенного: считываем файл и вызываем для каждой его строки метод addSong().

Метод addSong работает по тому же принципу, что и класс QuizCard из главы 14. — вы разбиваете строку (содержащую как название, так и имя исполнителя) на две части (лексемы) с помощью метода split().

Нам нужно только название песни, поэтому добавляем в songList (ArrayList) лишь первую лексему.

```
File Edit Window Help Dance
%java Jukebox1
[Pink Moon, Somersault, Shiva Moon, Circles, Deep Channel, Passenger, Listen]
```

Список songList выводит песни в том порядке, в котором они были добавлены (в том же порядке они размещались и в оригинальном текстовом файле).

Это точно не алфавитный порядок!

# Но у класса ArrayList нет метода sort()!

Посмотрев на описание ArrayList, вы не найдете ничего похожего на метод для сортировки. Не поможет и продвижение вверх по иерархии наследования — совершенно ясно, ***что нельзя вызвать метод sort() из ArrayList***!

# ArrayList — не единственная коллекция

Чаще всего вы будете использовать именно ArrayList, но существуют и другие списки со специальными возможностями. Ниже представлены некоторые ключевые классы коллекций:

Не пытайтесь запомнить все прямо сейчас. Чуть позже мы рассмотрим их более подробно.

- **TreeSet**
  Хранит элементы отсортированными и предотвращает дублирование.

- **HashMap**
  Позволяет хранить элементы и получать к ним доступ с помощью пар ключ/значение.

- **LinkedList**
  Разработан, чтобы обеспечить лучшую производительность в тех случаях, когда элементы вставляются и удаляются посередине коллекции (на практике вы, скорее всего, предпочтете ArrayList).

- **HashSet**
  Предотвращает дублирование и позволяет быстро найти заданный элемент в коллекции.

- **LinkedHashMap**
  Представляет собой то же самое, что и обычный HashMap, но умеет запоминать порядок, в котором вставлены элементы (пары ключ/значение). Кроме того, его можно настроить так, чтобы помнить, к какому элементу в последний раз предоставлялся доступ.

# Вы можете использовать TreeSet... Или метод Collections.sort()

Если вы поместите все строки (названия песен) в **TreeSet** вместо ArrayList, то они автоматически распределятся в алфавитном порядке. Когда бы вы ни вывели содержимое списка, его элементы будут отсортированы по алфавиту.

Этот класс отлично подходит для тех случаев, когда вам нужно *множество* (мы поговорим о них совсем скоро) или когда вы знаете, что список *всегда* должен оставаться отсортированным в алфавитном порядке.

С другой стороны, если вам не нужно, чтобы список оставался отсортированным, TreeSet может оказаться избыточным — ***каждый раз, когда вы добавляете в него элемент, класс тратит время на поиск места для этого элемента***. Подобная операция в ArrayList выполняется значительно быстрее, так как новый элемент добавляется в конец списка.

**java.util.Collections**

```
public static void copy(List destination, List source)
public static List emptyList()
public static void fill(List listToFill, Object objToFillItWith)
public static int frequency(Collection c, Object o)
public static void reverse(List list)
public static void rotate(List list, int distance)
public static void shuffle(List list)
public static void sort(List list)
public static boolean replaceAll(List list, Object oldVal, Object newVal)
// Многие другие методы
```

Хммммм... В классе Collections и правда есть метод sort(). Он принимает список List. Но, поскольку ArrayList реализует интерфейс List, они соответствуют друг другу. Благодаря полиморфизму можно передавать ArrayList в метод, который рассчитан на работу с List.

**В: Но вы можете добавить в ArrayList элемент по конкретному индексу, а не просто в конец списка — есть перегруженный метод add(), который наряду с элементом принимает целочисленное значение. Не будет ли это медленнее, чем обычное добавление в конец списка?**

О: Да, добавление элемента в ArrayList происходит медленнее, если при этом задается его конкретная позиция, *отличная* от конца списка. Поэтому метод add(index, element) работает не так быстро, как add(element). Но в большинстве случаев вам и не придется использовать при добавлении конкретные индексы.

**В: Я вижу, что в API есть класс LinkedList. Может быть, он лучше подойдет для добавления элементов в середину списка? По крайней мере так нам говорили на курсе по структурам данных в колледже, если я ничего не путаю...**

О: Хорошее замечание. LinkedList *может* оказаться быстрее при добавлении элементов по произвольному индексу, но для большинства приложений это не имеет значения, если только вы не работаете с *огромным* количеством записей. Через несколько минут мы более подробно поговорим о LinkedList.

Примечание: это не настоящий класс Collection. Мы упростили его, выбросив информацию об обобщенных типах (с которыми вы познакомитесь через несколько страниц).

# Добавляем Collections.sort() в код приложения Jukebox

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

    ArrayList<String> songList = new ArrayList<String>();

    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }

    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line =  null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        songList.add(tokens[0]);
    }
}
```

Метод Collections.sort() сортирует список строк в алфавитном порядке.

Вызываем статический метод sort() из класса Collections, а затем вновь выводим содержимое списка. Во втором случае элементы следуют в алфавитном порядке!

```
File Edit Window Help Chill
%java Jukebox1

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```

До вызова sort().

После вызова sort().

# Теперь нужно сортировать объекты Song, а не обычные строки

Вашему начальнику захотелось, чтобы в списке хранились экземпляры класса Song (он предоставляет много данных), а не просто строки. Новый музыкальный автомат записывает больше информации, поэтому каждая строка файла будет состоять из *четырех* частей (лексем), а не из двух.

Класс Song действительно прост, но обладает одной интересной особенностью — в нем есть переопределенный метод toString(). Помните, что этот метод объявлен в классе Object, поэтому его наследует любой класс в Java. И поскольку он вызывается в тот момент, когда объект нужно вывести на экран (System.out.println(anObject)), вы должны переопределить его, чтобы получить нечто более информативное, чем уникальный идентификатор. Метод toString() будет вызываться при выводе каждого объекта.

**SongListMore.txt**

```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
```

В новом файле у каждой песни есть четыре атрибута вместо двух. И мы хотим их все добавить в список, поэтому необходимо создать класс Song с переменными экземпляра для каждого атрибута.

```
class Song {
    String title;
    String artist;
    String rating;
    String bpm;

    Song(String t, String a, String r, String b) {
        title = t;
        artist = a;
        rating = r;
        bpm = b;
    }

    public String getTitle() {
        return title;
    }

    public String getArtist() {
        return artist;
    }

    public String getRating() {
        return rating;
    }

    public String getBpm() {
        return bpm;
    }

    public String toString() {
        return title;
    }
}
```

Четыре переменные экземпляра для четырех атрибутов песни, получаемых из файла.

Переменные, которые передаются в конструктор при создании нового объекта Song.

Геттеры для четырех атрибутов.

Мы переопределяем toString(), потому что при вызове System.out.println(aSongObject) хотим увидеть название песни. Этот вызов происходит при выводе каждого элемента списка.

# Адаптируем код Jukebox для использования объектов Song вместо строк

Код поменяется совсем незначительно — работа с вводом/выводом и разбор файла (String.split()) останутся прежними, изменится лишь количество лексем (теперь их *четыре*). Кроме того, все лексемы будут использоваться для создания объекта Song. И, естественно, тип ArrayList изменится со <String> на <Song>.

```
import java.util.*;
import java.io.*;

public class Jukebox3 {

    ArrayList<Song> songList = new ArrayList<Song>();
    public static void main(String[] args) {
        new Jukebox3().go();
    }
    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }
    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line =  null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");

        Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
        songList.add(nextSong);
    }
}
```

Меняем тип для ArrayList со String на Song.

Создаем новый объект Song с помощью четырех лексем (четыре составляющие каждой строки файла, содержащие информацию о песне), затем добавляем этот объект в список.

# Оно не скомпилируется!

Что-то здесь не так... В описании класса Collections совершенно точно есть метод sort(), который принимает List.

ArrayList соответствует интерфейсу List, так как реализует его, значит, все *должно* работать!

***Но оно не работает!***

Компилятор говорит, что не может найти метод, принимающий ArrayList<Song>, — наверное, ему просто не нравятся списки ArrayList с объектами Song? Он не возражал против ArrayList<String>, тогда какая разница между Song и String? Где причина ошибки компиляции?

```
File Edit Window Help Bummer
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
                Collections.sort(songList);
                           ^
1 error
```

И, наверное, вы уже спросили себя: «По *чему* будет выполняться сортировка?» Откуда метод sort() *знает*, чем одна песня лучше (больше) другой? Конечно, если вы хотите, чтобы сортировка происходила по *названию* песни, нужно как-то сообщить сортировочному методу о том, что он должен смотреть именно на название, а не на количество тактов в минуту, например.

Все это мы обсудим через несколько страниц, но сначала выясним, почему компилятор не позволяет передать в метод sort() список объектов Song.

## Объявление метода sort()

Collections (Java 2 Platform SE 5.0)

file:///Users/kathy/Public/docs/api/index.html

Google

**Method Detail**

**sort**

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

Судя по документации (переходим к классу java.util.Collections и прокручиваем вниз до метода sort()), метод sort() объявлен... *необычно*. Или, по крайней мере, не так, как те методы, что мы видели до сих пор.

А все потому, что метод sort() (наряду с другими элементами фреймворка для работы с коллекциями в Java) полноценно использует *обобщения (generics)*. Угловые скобки в исходных текстах или документации по Java — это обобщения (возможность, появившаяся в Java 5.0). Похоже, прежде чем мы сможем выяснить, как в ArrayList сортировать объекты Song вместо String, придется разобраться, что именно написано в документации.

# Обобщения обеспечивают большую безопасность типов

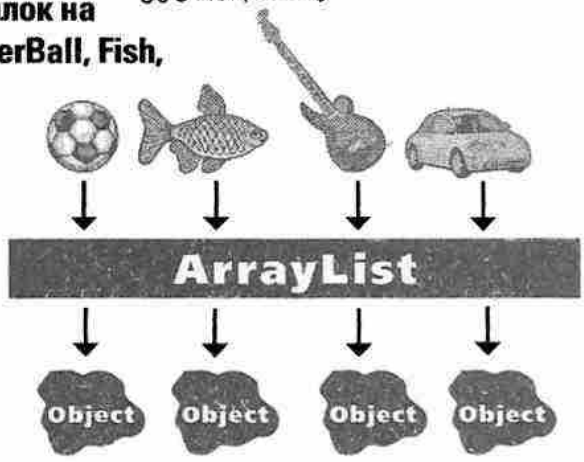
Сразу признаемся: *практически любой написанный вами код, в котором используются обобщения, будет связан с коллекциями.* Хотя для обобщений есть и другие области применения, их главный смысл заключается в возможности создавать безопасные с точки зрения типов коллекции. Иными словами, обобщения не позволят вам передать объект Dog туда, где ожидается Duck.

До появления обобщений (то есть до Java 5.0) компилятору было абсолютно все равно, что вы поместите в коллекцию, так как все реализации коллекций предназначались для хранения объектов типа Object. Вы могли поместить *что угодно* в любой ArrayList, как будто все экземпляры ArrayList объявлялись в виде ArrayList<Object>.

До обобщений не существовало способа объявить тип ArrayList, поэтому его метод add() принимал объекты Object.

**Без обобщений**

**Объекты попадают внутрь в виде ссылок на экземпляры SoccerBall, Fish, Guitar и Car.**




**И выходят как ссылки типа Object.**

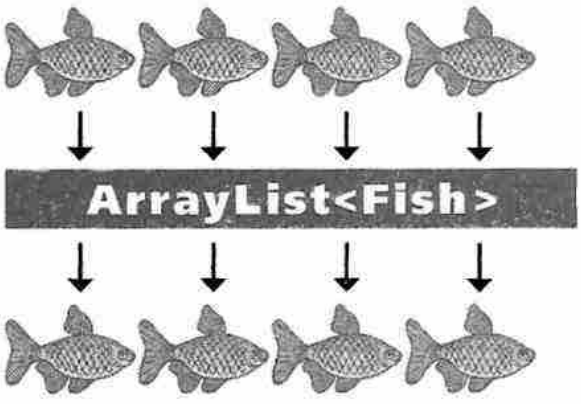
С помощью обобщений можно создавать коллекции, которые более строго следят за типами, вследствие чего большинство проблем отлавливается на этапе компиляции.

Без обобщений компилятор с радостью позволил бы вам засунуть объект Ритрки в ArrayList, предназначенный для хранения объектов типа Cat.

---

**С обобщениями**

**Внутрь попадают ссылки типа Fish (и только они).**




**Наружу выходят те самые ссылки типа Fish.**

Теперь с помощью обобщений можно помещать в ArrayList<Fish> только объекты типа Fish, поэтому на выходе вы получите то же, что помещали внутрь. Можете не волноваться, что кто-то добавит туда Volkswagen или что в результате получится объект, который нельзя привести к Fish.

# Изучаем обобщения

Из всего множества информации, которую вы могли бы узнать об обобщениях, реальную ценность для большинства программистов составляют лишь несколько вещей:

① **Создание экземпляров обобщенных *классов* (как ArrayList).**

Создавая объект ArrayList, вы должны указать тип элементов, допустимый для этого списка, — точно так же, как делали это со старыми добрыми массивами.

```
new ArrayList<Song>()
```

② **Объявление и инициализация *переменных*, имеющих обобщенный тип.**

Как же на самом деле работает полиморфизм для обобщенных типов? Если у вас есть ссылочная переменная ArrayList<Animal>, можете ли вы присвоить ей ArrayList<Dog>? А что насчет ссылки List<Animal>? Можете ли вы присвоить ей ArrayList<Animal>? Скоро вы об этом узнаете...

```
List<Song> songList =
      new ArrayList<Song>()
```

③ **Объявление (и вызов) *методов*, которые принимают значения обобщенных типов.**

Если у вас есть метод, принимающий в качестве параметра, скажем, ArrayList<Animal>, то о чем это говорит? Можно ли ему также передать ArrayList<Dog>? Мы рассмотрим хитрые и неочевидные проблемы, связанные с полиморфизмом и совершенно не характерные для методов, принимающих обычный массив.

В принципе, в последних двух пунктах речь идет об одном и том же, но это только подчеркивает, насколько серьезно мы относимся к данному вопросу.

```
void foo(List<Song> list)

x.foo(songList)
```

**В: Может быть, мне стоит научиться создавать собственные обобщенные классы? Как быть, если я захочу написать класс, который при создании своих экземпляров позволяет указывать тип возможных сущностей?**

**О:** Скорее всего, вам не придется этим заниматься. Судите сами — разработчики API создали целую библиотеку классов-коллекций, которые покрывают собой большинство нужных вам структур данных. Коллекции — это практически единственный тип классов, которые действительно нуждаются в обобщении. Иными словами, эти классы спроектированы для хранения других элементов, чей тип программисты должны указывать при объявлении и создании экземпляров.

Конечно, когда-нибудь вам может захотеться *создать* обобщенный класс, но это исключительно редкая ситуация и мы не будем ее рассматривать (но по материалам данной книги вы и сами сможете догадаться, как это делается).

# Использование обобщенных классов

Поскольку ArrayList — наиболее популярный обобщенный тип, мы начнем с изучения его документации. Есть два ключевых момента, на которые стоит обратить внимание при рассмотрении обобщенного класса.

1. Объявление *класса*.
2. Объявление *методов*, с помощью которых можно добавлять элементы.

Думайте о символе «E» как о типе элементов, которые вы хотели бы хранить в своей коллекции и возвращать обратно (E от слова Element).

## Рассмотрение документации по ArrayList, или Каков истинный смысл «E»?

«E» — это заполнитель для реального типа, который вы будете использовать при объявлении и создании ArrayList.

ArrayList — потомок класса AbstractList. Какой бы тип вы ни указали для ArrayList, он автоматически будет применяться и для AbstractList.

```
public class ArrayList<E> extends AbstractList<E> implements List<E>  ... {

    public boolean add(E o)

    // Еще код
}
```

Вот это важная часть! Тип, которым вы замените «E», определяет, какие элементы можно будет добавлять в ArrayList.

Тип (значение <E>), который также становится типом интерфейса List.

«E» представляет тип, который используется для создания экземпляра ArrayList. Натыкаясь на обозначение «E» в документации, вы можете мысленно подставить вместо него тип, с помощью которого вы хотите создавать объект ArrayList.

Таким образом, new ArrayList<Song> означает, что «E» превращается в «Song» в объявлениях всех методов или переменных, использующих «E».

# Использование типовых параметров на примере ArrayList

**Этот код:**

```
ArrayList<String> thisList = new ArrayList<String>
```

**означает, что ArrayList**

```
public class ArrayList<E> extends AbstractList<E> ... {

    public boolean add(E o)
    // Еще код
}
```

**будет воспринят компилятором как:**

```
public class ArrayList<String> extends AbstractList<String>... {

    public boolean add(String o)
    // Еще код
}
```

Проще говоря, «E» заменяется *настоящим* типом (известным как *типовой параметр*), который используется при создании ArrayList. Именно поэтому метод add() из ArrayList не позволяет вам добавлять в список ничего, кроме объектов ссылочного типа, совместимых с типом «E». Если вы создадите ArrayList<**String**>, то метод add() внезапно превратится в **add(String o)**. Если вы создадите список ArrayList для объектов **Dog**, то метод add() будет иметь вид **add(Dog o)**.

**В: Туда можно добавлять только «E»? В документации для метода sort() используется «T»...**

О: Можно применять все, что в Java считается допустимым идентификатором. Все, что может быть использовано в качестве имени метода или переменной, годится и для типового параметра. Но в соглашении по именованию рекомендуется применять в этих целях один символ (и именно это вам следует делать). Более того, «E» используется для создания коллекции и представляет «тип элемента, который она будет хранить». Для остальных случаев рекомендуется применять «T».

# Использование обобщенных методов

*Класс* считается обобщенным, если его объявление содержит типовой параметр. Аналогично обобщенные *методы* содержат в своей сигнатуре типовые параметры.

Можно по-разному применять типовые параметры в сочетании с методами.

① **Использование типового параметра, описанного при объявлении класса.**

```
public class ArrayList<E> extends AbstractList<E> ... {
    public boolean add(E o)
```

Вы можете использовать здесь символ «E» только потому, что он уже формально считается частью класса.

При объявлении типового параметра для класса вы можете использовать этот тип везде, где уместен *настоящий* класс или интерфейс. Тип, описанный в аргументе метода, по сути, замещается реальным типом, который применяется при создании экземпляра класса.

② **Использование типового параметра, который не был описан при объявлении класса.**

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

Здесь мы можем применять обозначение <T>, так как «T» был описан при объявлении метода.

Даже если класс обходится без типовых параметров, вы можете задавать их для отдельных методов (используя необычное, но вполне допустимое место — *до типа возвращаемого значения*). В этом методе говорится, что «T» может быть «типом Animal или любым его потомком».

# Вот с этого момента начинаются странности...

***Это:***

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

***не то же самое, что:***

```
public void takeThing(ArrayList<Animal> list)
```

Оба выражения корректны, но они *разные*!

Прежде всего **<T extends Animal>** — часть объявления метода. Это означает, что для ArrayList допустим как Animal, так и любой его дочерний тип (например, Dog или Cat). Метод, указанный первым, нужно вызывать с помощью ArrayList<Dog>, ArrayList<Cat> или ArrayList<Animal>.

Но... во втором примере, где список ArrayList<Animal> представляет собой аргумент метода, допустимо использовать *только* ArrayList<Animal>. Иными словами, первая версия принимает ArrayList любого типа, который соответствует Animal (Animal, Dog, Cat и т. д.), а для второй подходит *только* Animal. Ни ArrayList<Dog>, ни ArrayList<Cat>, только ArrayList<Animal>.

Да, на первый взгляд, это противоречит сути полиморфизма. Но все прояснится, когда мы вернемся к подобным примерам в конце главы. Пока просто не забывайте, что к этой теме мы пришли, пытаясь понять, как выполнить сортировку в списке SongList (это заставило нас изучить API для метода sort(), из-за чего мы и взялись за объявления с использованием обобщенных типов).

***На данном этапе нужно понять следующее: синтаксис первого примера вполне допустим, и потому можно передавать объект ArrayList с типовым параметром Animal или любым его дочерним типом.***

А теперь вернемся к нашему методу sort()...

**Вспомните, как это было...**

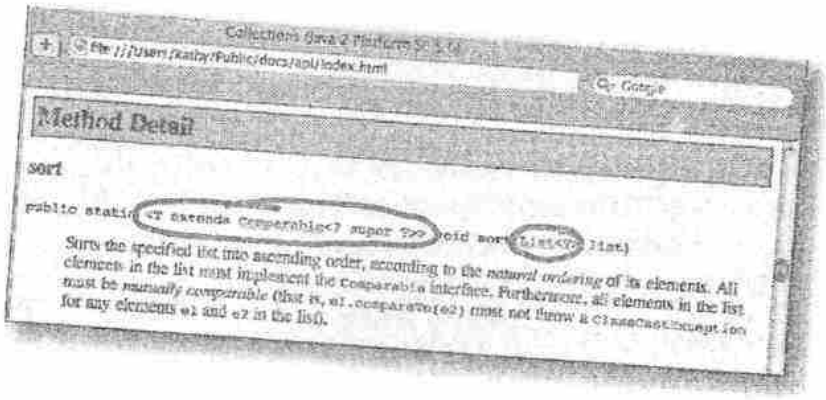
```
File Edit Window Help Bummer
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
                Collections.sort(songList);
                           ^
1 error
```

```
import java.util.*;
import java.io.*;

public class Jukebox3 {
    ArrayList<Song> songList = new ArrayList<Song>();
    public static void main(String[] args) {
        new Jukebox3().go();
    }
    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }
    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line = null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
        songList.add(nextSong);
    }
}
```

Вот где происходит ошибка! При передаче ArrayList<String> программа работает нормально, но как только мы пытаемся отсортировать ArrayList<Song>, компиляция завершается неудачей.

# Возвращаемся к методу sort()

Итак, мы вернулись и пытаемся прочитать документацию по методу sort(), чтобы понять, почему сортировать список строк можно, а список объектов Song — нет. Похоже, все дело в...




**Метод sort() может принимать только списки объектов Comparable.**

**Song — это не потомок Comparable, поэтому нельзя сортировать список объектов этого типа.**

***По крайней мере не сейчас...***

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Это означает, что любой «T» должен соответствовать типу Comparable.

Пока пропустите эту часть. Но если вам интересно, эта запись говорит о том, что типовой параметр для Comparable должен иметь тип «T» (или это может быть один из его подтипов).

Сюда вы можете передать только List (или такой его подтип, как ArrayList), который использует параметризованный (обобщенный) тип, расширяющий Comparable.

Хм... Я только что проверила документацию по классу String и выяснила, что он не наследует Comparable, а реализует его. **Comparable — это интерфейс**, поэтому выражение <T **extends** Comparable> — абсурд.

```
public final class String extends Object implements Serializable,
                                    Comparable<String>, CharSequence
```

## В контексте обобщений extends может означать как «расширяет», так и «реализует»

В контексте обобщений ключевое слово extends подразумевает отношение IS-A и работает как для классов, так и для интерфейсов.

Создатели Java должны были предоставить вам способ вводить ограничения для параметризованных типов, чтобы вы могли, к примеру, разрешить использование только дочерних классов Animal. Но такое ограничение может коснуться классов, реализующих определенный интерфейс. Нужен синтаксис, который будет работать в обеих ситуациях — при наследовании *(extends)* и реализации *(implements)*.

И победило ключевое слово... *extends*. На самом деле оно подразумевает отношение IS-A и работает вне зависимости от того, что указано справа — интерфейс или класс.

Comparable — интерфейс, поэтому данная часть выражения читается как «T должен быть типом, реализующим интерфейс Comparable».

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Не важно, что находится справа — класс или интерфейс... Вы все равно используете ключевое слово extends.

**В: Почему разработчики просто не ввели новое ключевое слово is?**

**О:** Добавление в язык нового ключевого слова — очень серьезный шаг, так как это может привести к поломке кода, написанного для предыдущих версий. Подумайте, ведь вы могли использовать переменную с именем is (что мы и делали в этой книге для представления входящих потоков). А поскольку нельзя использовать ключевые слова в качестве идентификаторов, то ранее написанный код, в котором это делалось (еще *до* появления нового ключевого слова), перестанет работать. И если у инженеров компании Sun есть возможность задействовать уже существующее ключевое слово, как в случае с extends, они ее, как правило, не упускают. Но иногда у них просто нет выбора...

За все время в Java было добавлено всего несколько ключевых слов, например ***assert*** в Java 1.4 и ***enum*** в Java 5.0 (последнее будет рассмотрено в Приложении Б). И эти нововведения ломали чужой код. Но иногда можно так скомпилировать и запустить *новую* версию Java, чтобы она вела себя, как *старая*. Это делается с помощью специального флага командной строки для компилятора или JVM. Флаг говорит нечто вроде: «Да, конечно, я знаю, что ты Java 1.4, но прошу, притворись версией 1.3, потому что я использую в своем коде переменную с именем assert, и когда-то вы говорили, что так можно делать!»

Чтобы проверить, доступен ли этот флаг, наберите в командной строке javac (для компилятора) или java (для JVM), не добавляя ничего в конце — на экране должен будет появиться список возможных параметров. Больше об этих флагах вы узнаете в главе 17.

# Наконец-то мы поняли, в чем ошибка... Класс Song должен реализовывать интерфейс Comparable

Мы можем передать ArrayList<Song> в метод sort() только в том случае, если класс Song реализует Comparable, потому что именно так этот метод был объявлен. Взглянув на документацию по API, становится понятно, что интерфейс Comparable довольно прост и требует реализации всего одного метода:

**java.lang.Comparable**

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

В описании метода compareTo() сказано:

**Возвращает:**
Отрицательное целое число, ноль или положительное число, если этот объект меньше, равен или больше указанного соответственно.

Похоже, метод compareTo() будет вызываться из одного объекта Song и принимать ссылку на другой объект Song. Экземпляр Song, из которого выполняется вызов, должен выяснить, где относительно него после сортировки разместится другой объект — выше, ниже или на том же уровне.

Теперь ваша главная задача — решить, что делает одну песню больше другой, и воплотить этот принцип в методе compareTo(). Любое отрицательное число означает, что объект Song, переданный в качестве аргумента, больше объекта Song, из которого вызывался метод. Возвращая положительное число, вы подразумеваете, что экземпляр Song, из которого вызывался compareTo(), больше переданного ему аргумента. Ноль означает, что объекты Song равны (по крайней мере с точки зрения сортировки). Например, у вас может быть две песни с одинаковым названием.

И здесь появляется множество различных проблем, о которых мы поговорим позже...

**Важный вопрос: что делает одну песню меньше (больше или равной) *другой*?**

**Вы не сможете реализовать интерфейс Comparable, пока не найдете ответ.**

Наточите свой карандаш

Подумайте, как можно реализовать метод compareTo(), чтобы объекты Song сортировались по названию. Напишите здесь псевдокод (а лучше настоящий код).

Подсказка: если вы на правильном пути, это должно занять не более трех строк кода!

# Новый, улучшенный класс Song, объекты которого можно сравнивать

Мы решили, что хотим сортировать объекты по названию, поэтому реализовали метод compareTo() так, чтобы он сравнивал переменные title двух объектов. Из одного объекта переменная была вызвана, а второй был передан ей в качестве аргумента. Проще говоря, песня, чей метод выполняется, должна сама определить, как ее название соотносится с названием переданного ей параметра.

Нам известно, что класс String должен знать об алфавитном порядке, так как метод sort() работает со списками строк. Нам также известно, что String содержит метод compareTo() — так почему бы просто его не вызвать? Тогда мы сможем сравнивать названия на уровне строк и не придется писать алгоритм для сравнения/упорядочивания!

Мы указываем тип, с которым может сравниваться класс, реализующий Comparable. Как правило, они совпадают. Получается, что при сортировке объекты Song могут сравниваться с другими объектами Song.

```
class Song implements Comparable<Song> {
    String title;
    String artist;
    String rating;
    String bpm;

    public int compareTo(Song s) {
        return title.compareTo(s.getTitle());
    }

    Song(String t, String a, String r, String b) {
        title = t;
        artist = a;
        rating = r;
        bpm = b;
    }

    public String getTitle() {
        return title;
    }

    public String getArtist() {
        return artist;
    }

    public String getRating() {
        return rating;
    }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
     return title;
  }
}
```

Метод sort() передает объект Song в compareTo(), чтобы увидеть, как тот соотносится с экземпляром Song, из которого вызван метод.

Здесь все просто! Мы лишь перекладываем всю работу на объекты String, представляющие собой переменные title, так как знаем, что у String есть метод compareTo().

На этот раз программа работает. Она выводит список и сортирует объекты по названию в алфавитном порядке.

```
File Edit Window Help Ambient
%java Jukebox3

[Pink Moon, Somersault, Shiva Moon, Circles,
Deep Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger,
Pink Moon, Shiva Moon, Somersault]
```

# Мы можем отсортировать список, но...

Твоя программа недостаточно хороша. Иногда мне нужно сортировать песни не по названию, а по имени исполнителя.

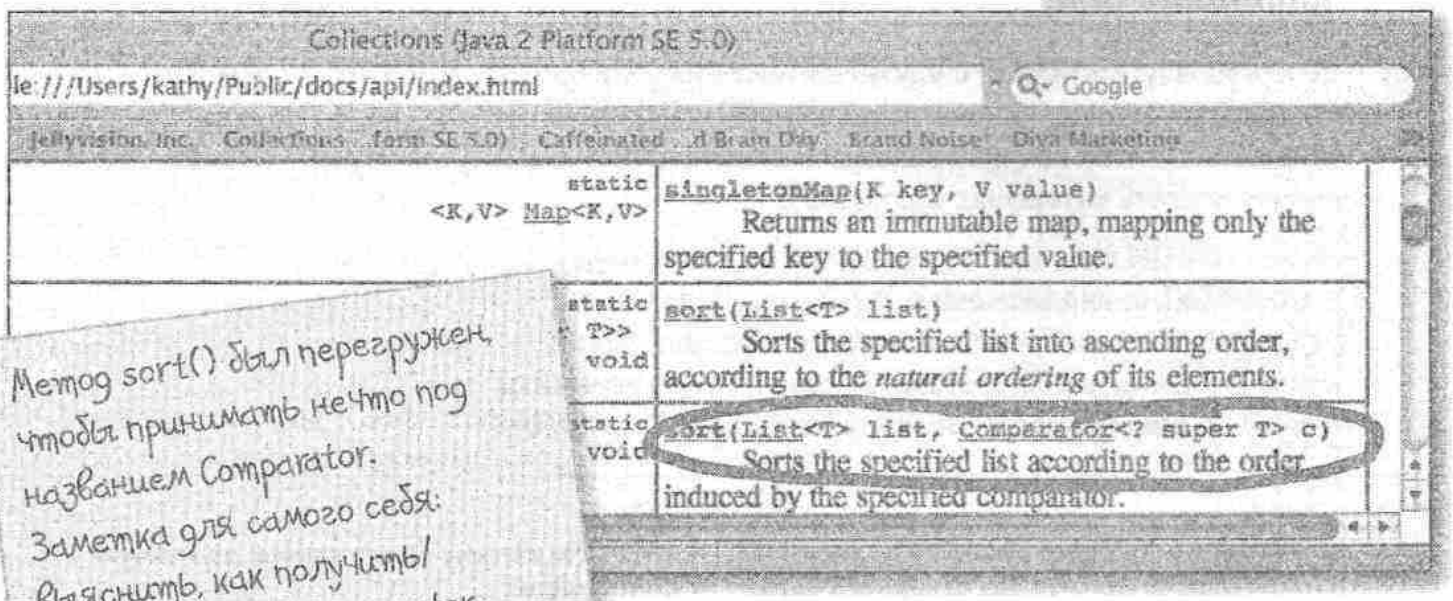
Есть одна проблема: владельцу закусочной нужны два варианта сортировки списка песен — по названию и по имени исполнителя!

Реализуя интерфейс Comparable, вы получаете возможность реализовать лишь один метод compareTo(). Что же делать?

Худшее решение — указать в классе Song флаг, исходя из значения которого метод compareTo() будет возвращать разные результаты, используя для сравнения либо название, либо исполнителя.

Но существует кое-что намного лучше. Нечто, встроенное в API именно для этих целей, когда нужно отсортировать одну и ту же сущность несколькими способами.

**Взгляните на документацию по классу Collections. Существует второй метод sort() — и он принимает Comparator.**




Метод sort() был перегружен, чтобы принимать нечто под названием Comparator.
Заметка для самого себя: выяснить, как получить/создать объект Comparator, который смог бы сравнивать и упорядочивать песни по имени исполнителя, а не по названию...

# Использование собственной реализации интерфейса Comparator

**java.util.Comparator**

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

Элементы списка, имеющие один и тот же тип, могут сравниваться единственным способом — с помощью метода compareTo(). Но реализация Comparator считается внешней по отношению к типу сравниваемого элемента — это отдельный класс. Таким образом, можете создать их столько, сколько вам нужно! Хотите сравнить песни по имени исполнителя? Создайте ArtistComparator. Хотите сделать сортировку по названию? Создайте TitleComparator.

Все, что вам остается сделать, — вызвать перегруженный метод sort(), который наряду с List принимает Comparator (он и будет указывать, в каком порядке должны располагаться элементы).

Для сортировки будет использоваться Comparator, передаваемый в sort(), а не метод compareTo() из каждого элемента. Иначе говоря, если sort() примет Comparator, он даже не будет *вызывать* метод compareTo() из элементов списка. Вместо этого будет использоваться метод **compare()**, принадлежащий реализации Comparator.

Здесь действуют следующие правила.

- **Вызов метода sort(List o) с одним аргументом означает, что порядок определяют методы compareTo() каждого элемента. По этой причине элементы списка должны реализовывать интерфейс Comparable.**
- **Вызов sort(List o, Comparator c) означает, что методы compareTo(), принадлежащие элементам списка, не будут вызываться. Вместо этого будет задействован метод compare() из Comparator. В таком случае элементы списка не должны реализовывать интерфейс Comparable.**

Если вы передадите Comparator методу sort(), то именно он будет определять порядок сортировки, а не методы compareTo(), принадлежащие отдельным элементам.

---

**В: Если у меня есть класс, который не реализует Comparable и поставляется без исходного кода, я все равно могу выполнять сортировку с помощью Comparator?**

О: Да. Еще один вариант — наследовать класс элемента (если это возможно) и сделать так, чтобы он реализовывал Comparable.

**В: Но почему не все классы реализуют Comparable?**

О: Неужели вы действительно верите, что *все на свете* можно упорядочить? Если тип ваших элементов не поддается ни одному естественному виду сортировки, то, реализовав Comparable, вы введете в заблуждение других программистов. Игнорируя этот интерфейс, вы ничем не рискуете, так как разработчик сам может сделать сравнение, используя собственный Comparator.

# Обновляем Jukebox с использованием Comparator

Мы внесли в код три новых изменения.

1. Создали вложенный класс, реализующий Comparator (а заодно и метод ***compare()***, работу которого ранее выполнял ***compareTo()***).
2. Создали экземпляр вложенной реализации Comparator.
3. Вызвали перегруженный метод sort(), передавая ему список песен и экземпляр вложенного класса Comparator.

Примечание: мы также обновили метод toString() из класса Song, чтобы он вместе с названием песни выводил имя исполнителя (он выводит строку ***«название: исполнитель»*** вне зависимости от вида сортировки).

```
import java.util.*;
import java.io.*;

public class Jukebox5 {
   ArrayList<Song> songList = new ArrayList<Song>();
   public static void main(String[] args) {
      new Jukebox5().go();
   }

   class ArtistCompare implements Comparator<Song> {
     public int compare(Song one, Song two) {
         return one.getArtist().compareTo(two.getArtist());
     }
   }
   public void go() {
     getSongs();
     System.out.println(songList);
     Collections.sort(songList);
     System.out.println(songList);

     ArtistCompare  artistCompare = new ArtistCompare();
     Collections.sort(songList, artistCompare);

     System.out.println(songList);
   }

   void getSongs() {
       // Код для ввода/вывода
   }

   void addSong(String lineToParse) {
       // Делаем разбор строки
       // и добавляем песню в список
   }
}
```

Создаем новый вложенный класс, реализующий Comparator (обратите внимание, что тип параметра совпадает с типом, который мы собираемся сравнивать: в данном случае это Song).

Это становится строкой (именем исполнителя).

Переклыдываем всю работу по сравнению на строковые объекты, так как они уже умеют сортироваться в алфавитном порядке.

Создаем экземпляр вложенного класса Comparator.

Вызываем метод sort(), передаем ему список и ссылку на новый объект Comparator.

Примечание: поскольку мы оставили нетронутым метод compareTo () в классе Song, то по умолчанию при сортировке будет использоваться название. Но то же самое можно сделать, создав вложенные реализации Comparator как для названия, так и для имени исполнителя, совсем отказавшись от реализации Comparable для Song. Таким образом, мы бы всегда использовали версию Collections.sort() с двумя аргументами.

# Обновляем Jukebox с использованием Comparator

Мы внесли в код три новых изменения.

1. Создали вложенный класс, реализующий Comparator (а заодно и метод ***compare()***, работу которого ранее выполнял ***compareTo()***).
2. Создали экземпляр вложенной реализации Comparator.
3. Вызвали перегруженный метод sort(), передавая ему список песен и экземпляр вложенного класса Comparator.

Примечание: мы также обновили метод toString() из класса Song, чтобы он вместе с названием песни выводил имя исполнителя (он выводит строку ***«название: исполнитель»*** вне зависимости от вида сортировки).

```
import java.util.*;
import java.io.*;

public class Jukebox5 {
   ArrayList<Song> songList = new ArrayList<Song>();
   public static void main(String[] args) {
      new Jukebox5().go();
   }

   class ArtistCompare implements Comparator<Song> {
     public int compare(Song one, Song two) {
         return one.getArtist().compareTo(two.getArtist());
     }
   }
   public void go() {
     getSongs();
     System.out.println(songList);
     Collections.sort(songList);
     System.out.println(songList);

     ArtistCompare  artistCompare = new ArtistCompare();
     Collections.sort(songList, artistCompare);

     System.out.println(songList);
   }

   void getSongs() {
       // Код для ввода/вывода
   }

   void addSong(String lineToParse) {
       // Делаем разбор строки
       // и добавляем песню в список
   }
}
```

Создаем новый вложенный класс, реализующий Comparator (обратите внимание, что тип параметра совпадает с типом, который мы собираемся сравнивать: в данном случае это Song).

Это становится строкой (именем исполнителя).

Перекладываем всю работу по сравнению на строковые объекты, так как они уже умеют сортироваться в алфавитном порядке.

Создаем экземпляр вложенного класса Comparator.

Вызываем метод sort(), передаем ему список и ссылку на новый объект Comparator.

Примечание: поскольку мы оставили нетронутым метод compareTo () в классе Song, то по умолчанию при сортировке будет использоваться название. Но то же самое можно сделать, создав вложенные реализации Comparator как для названия, так и для имени исполнителя, совсем отказавшись от реализации Comparable для Song. Таким образом, мы бы всегда использовали версию Collections.sort() с двумя аргументами.

```
import _____________________;

public class SortMountains {

  LinkedList_______________ mtn = new LinkedList_____________();

  class NameCompare ____________________________________ {
    public int compare(Mountain one, Mountain two) {
      return _____________________________;
    }
  }
  class HeightCompare ____________________________________ {
    public int compare(Mountain one, Mountain two) {
      return (_____________________________);
    }
  }
  public static void main(String [] args) {
    new SortMountain().go();
  }
  public void go() {
    mtn.add(new Mountain("Лонг-Рейндж", 14255));
    mtn.add(new Mountain("Эльберт", 14433));
    mtn.add(new Mountain("Марун", 14156));
    mtn.add(new Mountain("Касл", 14265));
    System.out.println("В порядке добавления:\n" + mtn);
    NameCompare nc = new NameCompare();
    ____________________________________;
    System.out.println("По названию:\n" + mtn);
    HeightCompare hc = new HeightCompare();
    ____________________________________;
    System.out.println("По высоте:\n" + mtn);
  }
}

class Mountain {

  ____________________;
  ____________________;

  _____________________ {
    ____________________;
    ____________________;
  }
  _____________________ {
    ____________________________________;
  }
}
```

## Программирование задом наперед

Представьте, что этот код расположен в одном файле. Ваша задача — заполнить пропущенные фрагменты так, чтобы программа вывела текст, приведенный ниже.

Примечание: ответы вы найдете в конце главы.

**Результат:**

```
File Edit Window Help ThisOne'sForBob
В порядке добавления:
[Лонг-Рейндж 14255, Эльберт 14433, Марун 14156, Касл 14265]
По названию:
[Касл 14265, Лонг-Рейндж 14255, Марун 14156, Эльберт 14433]
По высоте:
[Эльберт 14433, Касл 14265, Лонг-Рейндж 14255, Марун 14156]
```

Наточите свой карандаш

## Заполните пустые поля

В пустых полях каждого из вопросов, приведенных ниже, подставьте по одному слову из списка возможных ответов. Правильные ответы находятся в конце главы.

**Возможные ответы:**

**Comparator,**

**Comparable,**

**compareTo( ),**

**compare( ),**

**да,**

**нет**

Дано следующее компилируемое выражение:

```
Collections.sort(myArrayList);
```

1. Что должен реализовывать класс, объекты которого хранятся в `myArrayList`? _____
2. Какой метод должен реализовывать класс, объекты которого хранятся в `myArrayList`? _____
3. Может ли класс, объекты которого хранятся в `myArrayList`, одновременно реализовывать `Comparator` и `Comparable`? _____

Дано следующее компилируемое выражение:

```
Collections.sort(myArrayList, myCompare);
```

4. Может ли класс объектов, хранящихся в `myArrayList`, реализовывать интерфейс `Comparable`? _____
5. Может ли класс объектов, хранящихся в `myArrayList`, реализовывать интерфейс `Comparable`? _____
6. Должен ли класс объектов, хранящихся в `myArrayList`, реализовывать интерфейс `Comparable`? _____
7. Должен ли класс объектов, хранящихся в `myArrayList`, реализовывать интерфейс `Comparable`? _____
8. Что должен реализовывать класс объекта `myCompare`? _____
9. Какой метод должен реализовывать класс объекта `myCompare`? _____

# Ох, сортировка-то работает, но теперь у нас появились дубликаты...

Сортировка работает отлично. Теперь мы знаем, как упорядочивать записи по *названию* (используя метод compareTo() из объекта Song) и по *имени исполнителя* (с помощью метода compare() из Comparator). Но существует другая проблема, которую мы не заметили в тестовом примере файла из автомата, — ***отсортированный список содержит дубликаты***.

Это происходит потому, что музыкальный автомат в закусочной продолжает записывать данные в файл независимо от того, проигрывалась ли ранее текущая песня (в этом случае она уже есть в текстовом файле). Файл SongListMore.txt — это полный список песен, воспроизведенных автоматом, и его записи могут повторяться несколько раз.

```
File Edit Window Help TooManyNotes
%java Jukebox4

[Pink Moon: Nick Drake, Somersault: Zero 7, Shiva Moon:
Prem Joshua, Circles: BT, Deep Channel: Afro Celts,
Passenger: Headmix, Listen: Tahiti 80, Listen: Tahiti
80, Listen: Tahiti 80, Circles: BT]

[Circles: BT, Circles: BT, Deep Channel: Afro Celts,
Listen: Tahiti 80, Listen: Tahiti 80, Listen: Tahiti
80, Passenger: Headmix, Pink Moon: Nick Drake, Shiva
Moon: Prem Joshua, Somersault: Zero 7]

[Deep Channel: Afro Celts, Circles: BT, Circles: BT,
Passenger: Headmix, Pink Moon: Nick Drake, Shiva Moon:
Prem Joshua, Listen: Tahiti 80, Listen: Tahiti 80,
Listen: Tahiti 80, Somersault: Zero 7]
```

До сортировки.

После сортировки с помощью метода compareTo() из Song (сортировка по названию).

После сортировки с помощью ArtistCompare — реализации Comparator (сортировка по имени исполнителя).

**SongListMore.txt**

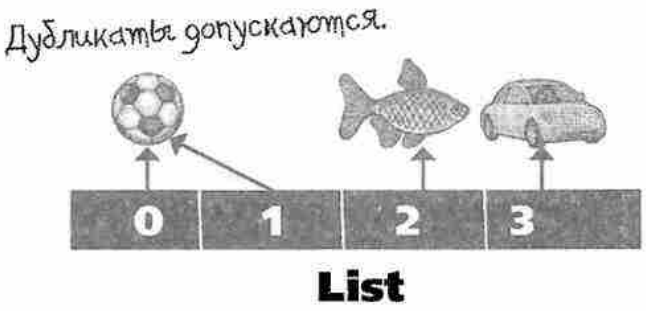
```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Circles/BT/5/110
```

Теперь текстовый файл SonListMore содержит дубликаты, так как музыкальный автомат записывает каждую проигрываемую песню. Кому-то захотелось три раза подряд прослушать композицию «Listen», после чего была выбрана песня «Circles», воспроизводимая ранее.

Мы не можем изменить порядок записи в файл, потому что когда-нибудь нам может понадобиться вся эта информация. Нам нужно изменить код.

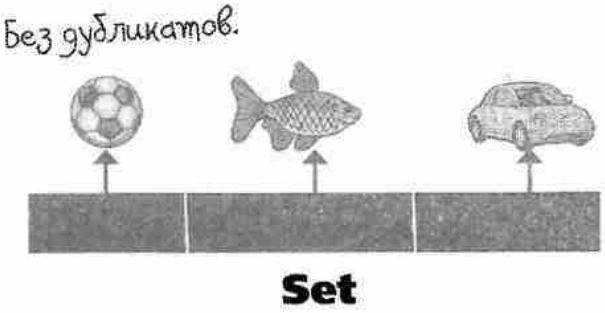
# Вместо списка нужно множество

Судя по API для работы с коллекциями, существует три главных интерфейса — **List** (список), **Set** (множество) и **Map** (отображение, или ассоциативный массив). ArrayList — список, но нам, похоже, нужно именно множество.




- **Список применяется, когда имеет значение порядок следования.**

  Коллекции, которые учитывают ***индекс (позицию)***. Списки знают, где хранятся их элементы. Сразу несколько элементов могут ссылаться на один объект.




- **Множество используется, когда имеет значение уникальность.**

  Коллекции, которые ***не допускают дублирования***.

  Множества знают, содержат ли они объект. У вас никогда не может быть несколько элементов, ссылающихся на один и тот же объект (или на два идентичных объекта — вопрос идентичности мы рассмотрим в ближайшее время).

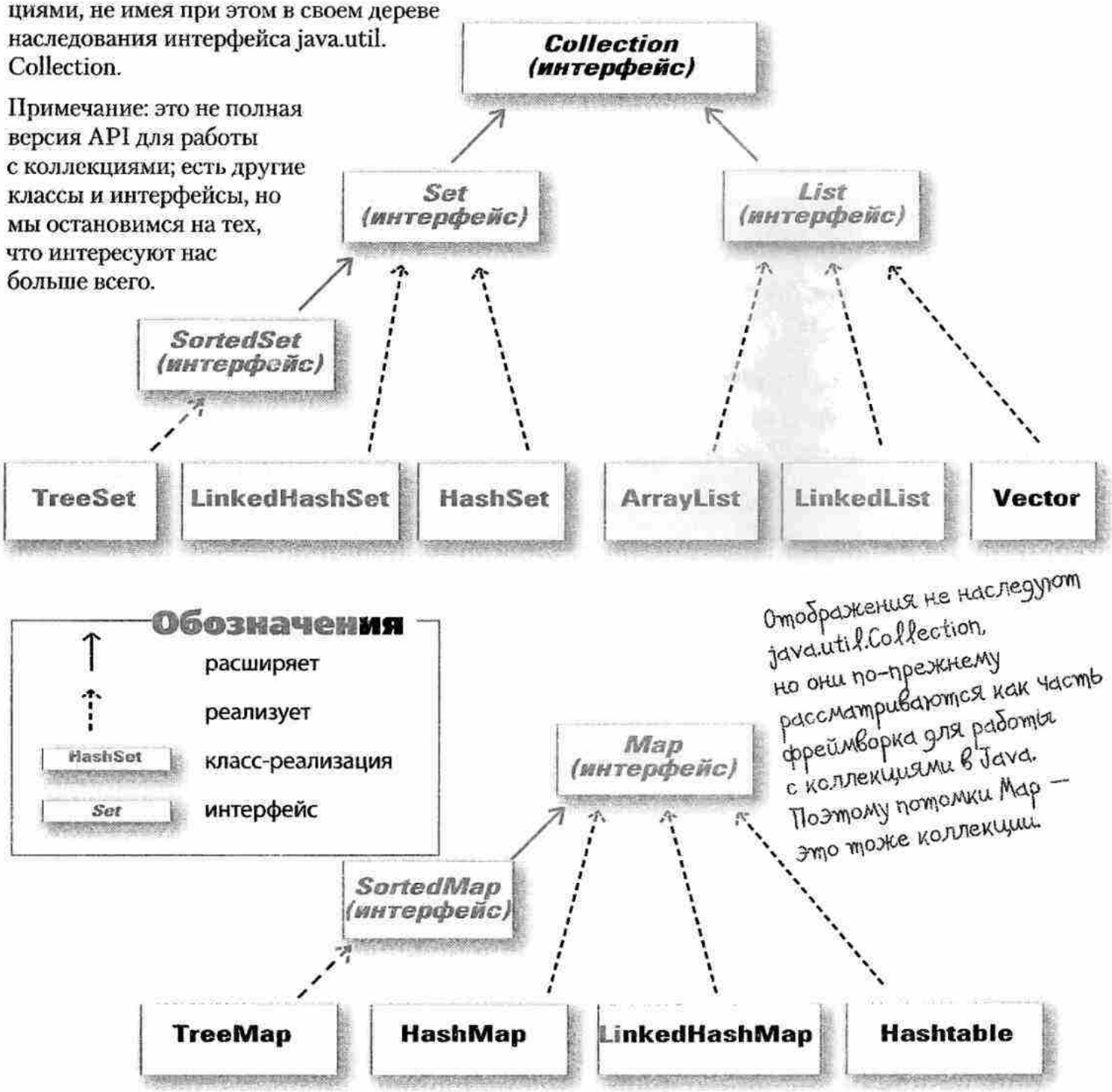
- **Отображение применяется, когда нужно найти что-нибудь по ключу.**

  Коллекции, в которых используются пары ***«ключ — значение»***.
  Отображению известно значение, которое связано с данным ключом. У вас может быть два ключа, ссылающихся на одно и то же значение, но они не могут дублироваться. В качестве ключей, как правило, используются строковые имена (например, так создаются списки свойств имя/значение), но ключом также может выступать любой объект.

# API для работы с коллекциями (его часть)

Хотя интерфейс Map на самом деле не наследует Collection, он все равно рассматривается как часть Collection Framework (также известного как API для работы с коллекциями). Таким образом, отображения считаются коллекциями, не имея при этом в своем дереве наследования интерфейса java.util.Collection.

Примечание: это не полная версия API для работы с коллекциями; есть другие классы и интерфейсы, но мы остановимся на тех, что интересуют нас больше всего.




Отображения не наследуют java.util.Collection, но они по-прежнему рассматриваются как часть фреймворка для работы с коллекциями в Java. Поэтому потомки Map — это тоже коллекции.

# Использование HashSet вместо ArrayList

Мы исправили Jukebox так, чтобы он сохранял композиции в множество HashSet.

Примечание: мы опустили некоторые части кода Jukebox, но вы можете скопировать их из предыдущих версий. Чтобы вам было легче читать программный вывод, мы вернулись к первоначальному варианту метода toString() из класса Song, поэтому теперь он выдает только название песни, *без* имени исполнителя.

```
import java.util.*;
import java.io.*;

public class Jukebox6 {
   ArrayList<Song> songList = new ArrayList<Song>();
   // Метод method и т. д.

   public void go() {
      getSongs();
      System.out.println(songList);
      Collections.sort(songList);
      System.out.println(songList);
      HashSet<Song> songSet = new HashSet<Song>();
      songSet.addAll(songList);
      System.out.println(songSet);
   }
   // Методы getSongs() и addSong()
}
```

Мы не меняли getSongs(), поэтому он все еще добавляет песни в ArrayList.

Здесь мы создаем множество HashSet, предназначенное для хранения объектов Song.

addAll() — простой метод, который принимает другую коллекцию и использует ее, чтобы заполнить HashSet. Результат такой же, как при добавлении каждой песни отдельно (только намного проще).

```
File Edit Window Help GetBetterMusic
%java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen, Listen, Listen, Circles]

[Circles, Circles, Deep Channel, Listen, Listen, Listen,
Passenger, Pink Moon, Shiva Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles, Listen, Deep
Channel, Passenger, Circles, Listen, Somersault]
```

До сортировки ArrayList.

После сортировки ArrayList (по названию).

После размещения песен в HashSet и вывода его содержимого (мы не вызывали sort() повторно).

Множество не помогло! У нас по-прежнему остались все дубликаты!

И при передаче списка в HashSet нарушился порядок сортировки, но этим мы займемся позже...

# Что делает два объекта равными

Прежде всего мы должны спросить себя — из-за чего дублируются ссылки на объекты Song? Они должны рассматриваться как ***равные*** (эквивалентные). Они ссылаются на один и тот же объект или объектов два, но оба имеют одинаковую переменную *title*?

В этом кроется проблема ключей: равенство *ссылок* противопоставляется равенству *объектов*.

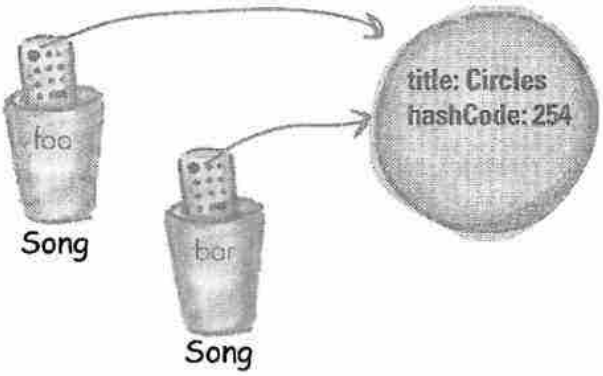
**Если объекты foo и bar равны, то вызов foo.equals(bar) должен возвращать true, а результаты выполнения методов hashCode() для обоих объектов будут совпадать. Чтобы множество восприняло объекты как равные, необходимо переопределить методы hashCode() и equals(), наследуемые от класса Object. Таким образом, два разных объекта смогут быть восприняты как дубликаты.**

## Равенство ссылок

**Две ссылки, один объект в куче.**

Две ссылки, связанные с одним объектом в куче, равны. Точка. Вызвав метод **hashCode()** для двух таких ссылок, вы получите один результат. Если вы не переопределите метод hashCode() (помните, что он унаследован от класса Object), то по умолчанию каждый объект получит уникальный номер. Большинство версий Java присваивают идентификаторы исходя из адресов памяти, по которым объекты находятся в куче, поэтому два объекта не могут иметь один идентификатор.

Если хотите знать, указывают ли две *ссылки* на один и тот же объект, используйте оператор ==, который, как вы помните, сравнивает содержащиеся в переменных биты. Если обе ссылки связаны с одним объектом, то их биты будут идентичны.



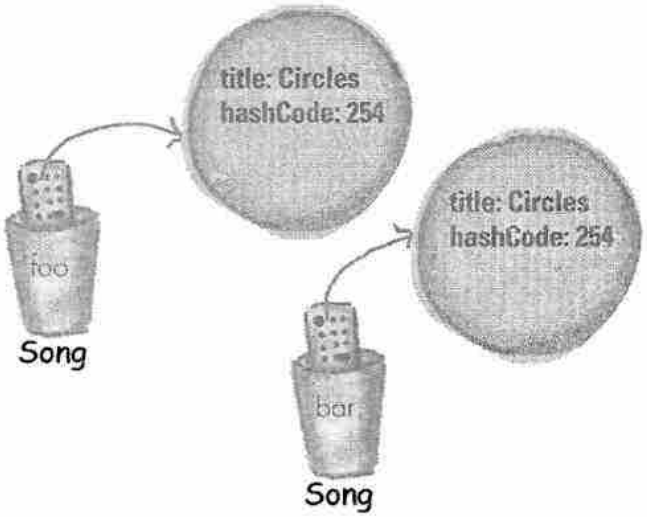

```
if (foo == bar) {
   // Обе ссылки указывают на один
   // и тот же объект в куче
}
```

## Равенство объектов

**Две ссылки, два объекта в куче, но объекты рассматриваются как *равные*.**

Если вы хотите, чтобы два разных объекта Song были восприняты как одинаковые (например, вы решите, что для равенства двух песен достаточно, чтобы совпадали их *названия*), то переопределите *оба* метода — **hashCode()** и **equals()**, унаследованные от класса Object.

Как уже было сказано, если вы *не* переопределите метод hashCode(), то по умолчанию (как в Object) каждому объекту будет выдаваться уникальный идентификатор. Если же нужно, чтобы два эквивалентных объекта возвращали один код, переопределите этот метод. Кроме того, нужно переопределить метод equals(), чтобы, вызывая его из одного объекта и передавая ему другой, всегда получать **true**.




```
if (foo.equals(bar) && foo.hashCode() =
                              = bar.hashCode()) {
  // Обе ссылки указывают либо на один и тот же
  // объект, либо на два равных объекта
}
```

# Как HashSet распознает дубликаты: hashCode() и equals()

Когда вы помещаете объект в множество HashSet, оно использует значение идентификатора объекта, чтобы определить, где именно тот будет размещен. При этом идентификатор сравнивается с идентификаторами всех элементов внутри HashSet. Если совпадений нет, то подразумевается, что новый объект не дублирует никакой другой.

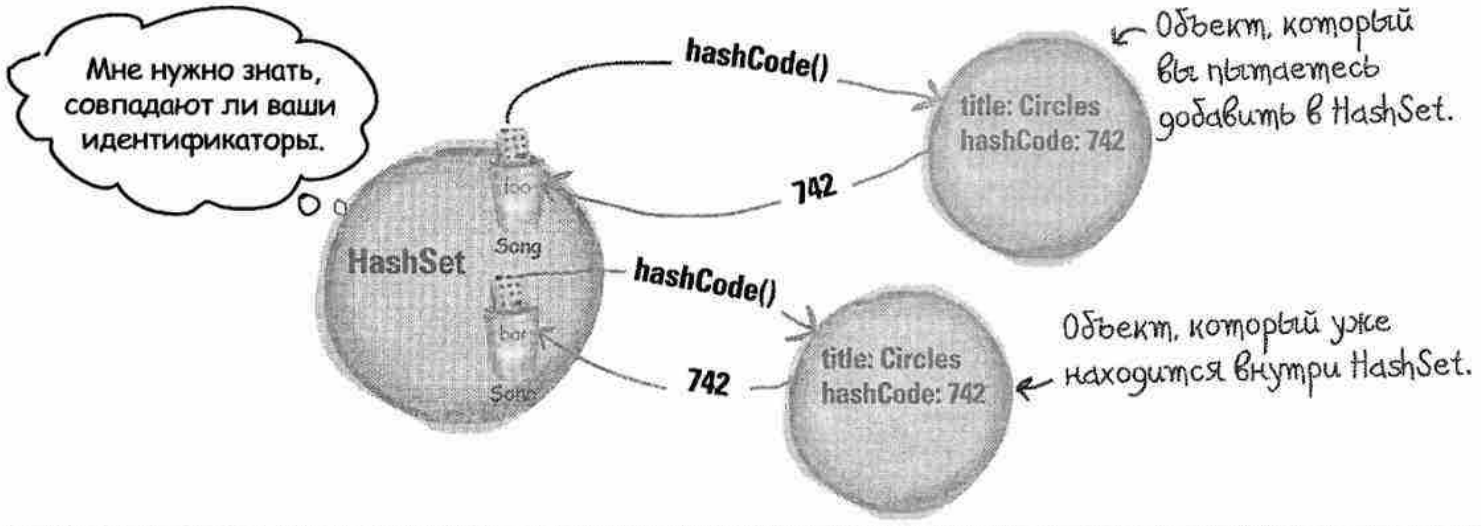
Иными словами, если идентификаторы различаются, то HashSet уверено, что объекты не могут быть равны!

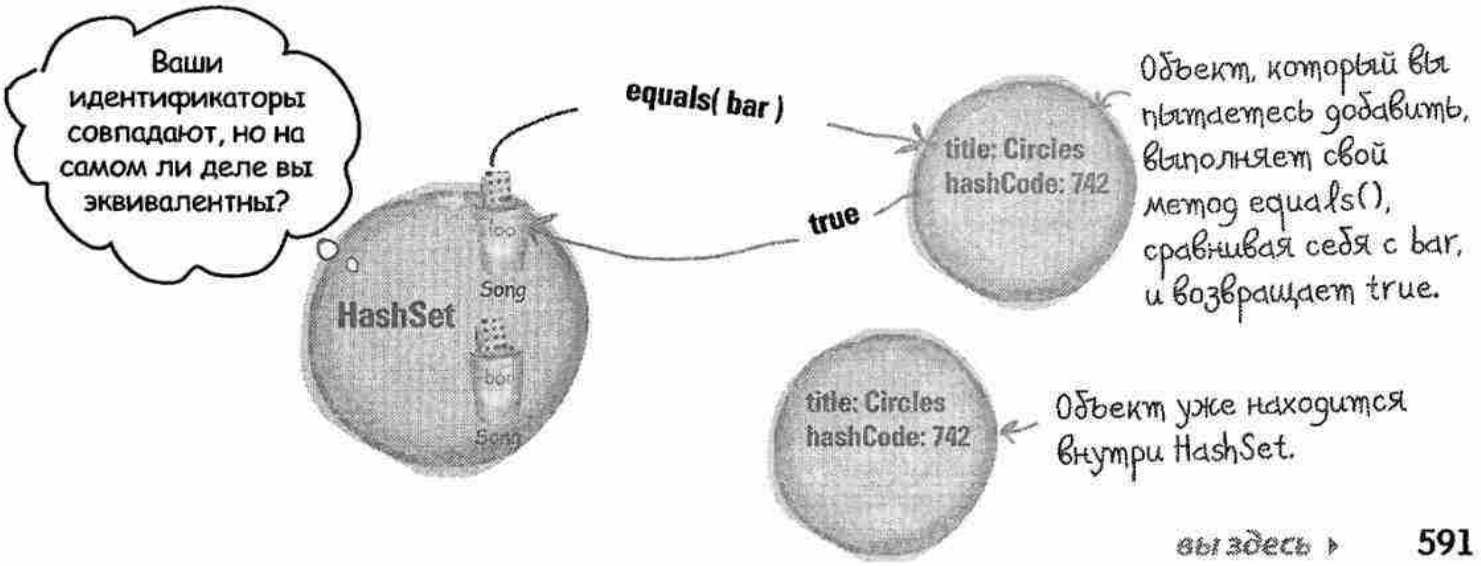
Для того чтобы они все же совпали, необходимо переопределить метод hashCode().

Но даже если два объекта возвращают одинаковое значение hashCode(), они все еще могут быть *не* равны (более подробно об этом на следующей странице). Видя два одинаковых идентификатора у объектов, один из которых добавляется в множество, а другой уже там находится, HashSet вызывает метод equals(), принадлежащий одному из объектов, чтобы убедиться, что они *действительно* эквивалентны.

Удостоверившись в этом, HashSet будет знать, что при добавлении такого объекта произойдет дублирование, значит, он не попадет в множество.

Вы не получите исключение, но метод add(), принадлежащий HashSet, возвращает булево значение, чтобы сообщить, был ли добавлен новый объект (если вас это интересует). Когда add() возвратит *false*, вы будете знать, что новый объект дублирует один из элементов множества.



# Класс Song с переопределенными методами hashCode() и equals()

```
class Song implements Comparable<Song>{
   String title;
   String artist;
   String rating;
   String bpm;

  public boolean equals(Object aSong) {
     Song s = (Song) aSong;
     return getTitle().equals(s.getTitle());
  }

  public int hashCode() {
      return title.hashCode();
  }

   public int compareTo(Song s) {
      return title.compareTo(s.getTitle());
   }

   Song(String t, String a, String r, String b) {
      title = t;
      artist = a;
      rating = r;
      bpm = b;
   }

   public String getTitle() {
       return title;
   }

   public String getArtist() {
      return artist;
   }

   public String getRating() {
      return rating;
   }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
     return title;
  }
}
```

HashSet (или кто-нибудь еще, кто вызывает этот метод) передает сюда другой объект Song.

Хорошая новость заключается в том, что title — это строка, а у строки есть переопределенный метод equals(). Нужно лишь спросить у переменной title, совпадает ли ее значение с названием переданной песни.

Здесь то же самое... Класс String содержит переопределенный метод hashCode(), поэтому вы просто можете вызвать его из переменной title и вернуть результат. Обратите внимание, что hashCode() и equals() используют одну переменную.

Теперь программа работает! При выводе содержимого HashSet нет дубликатов. Но мы не вызвали sort() повторно, поэтому при добавлении ArrayList в HashSet порядок сортировки был нарушен.

File Edit Window Help RebootWindows

```
%java Jukebox6

[Pink Moon, Somersault, Shiva Moon,
Circles, Deep Channel, Passenger, Listen,
Listen, Listen, Circles]

[Circles, Circles, Deep Channel, Listen,
Listen, Listen, Passenger, Pink Moon, Shiva
Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles,
Deep Channel, Passenger, Somersault]
```

## Правила для объектов в Java, касающиеся методов hashCode() и equals()

**В документации по классу Object содержатся правила, которым вы обязаны следовать.**

- **Если два объекта равны, то они должны иметь одинаковые идентификаторы.**
- **Если два объекта равны, то вызов equals() для каждого из них должен возвращать true. Иными словами, если (a.equals(b)), то (b.equals(a)).**
- **Если два объекта содержат одинаковые идентификаторы, то они не обязательно должны быть равны. Но если они равны, то их идентификаторы должны совпадать.**
- **Если вы переопределяете equals(), то также должны переопределить hashCode().**
- **По умолчанию hashCode() генерирует уникальное целое число для каждого объекта в куче, поэтому, если вы не переопределите данный метод для определенного класса, никакие два объекта этого класса никогда не смогут рассматриваться как равные.**
- **По умолчанию equals() выполняет операцию сравнения ==, то есть проверяет, указывают ли две ссылки на один объект в куче. Таким образом, если вы не переопределите метод equals() в заданном классе, никакие два объекта этого класса никогда не будут рассматриваться как равные, потому что ссылки на два разных объекта всегда будут содержать разные наборы битов.**
- ***a.equals(b) должно также означать, что***
- ***a.hashCode() == b.hashCode()***
- **Но если a.hashCode() == b.hashCode(),**
- **это вовсе не значит, что a.equals(b)**

## Это не глупые вопросы

**В: Как же у разных объектов могут совпадать идентификаторы?**

О: Множество HashSet использует идентификаторы для хранения элементов так, чтобы значительно ускорить доступ к ним. Если вы пытаетесь найти в ArrayList объект, передавая его копию (вместо индекса), то придется начинать поиски с самого начала, проверяя на совпадение каждый элемент списка. Однако HashSet может найти объект намного быстрее, так как применяет идентификаторы в качестве меток для участков памяти, где хранится элемент. Например, вы скажете: «Я хочу найти внутри множества объект, который выглядит в точности, как этот...» Тогда HashSet возьмет идентификатор из копии объекта Song, который вы ему передали (скажем, 742), и ответит: «Я точно знаю, где хранится объект с идентификатором #742...» После этого он перейдет прямо к участку памяти #742.

Конечно, из различных компьютерных дисциплин вы смогли бы узнать намного больше, но этого будет достаточно, чтобы эффективно использовать HashSet. На практике создание хорошего алгоритма для работы с идентификаторами может послужить темой для докторской диссертации, и это явно выходит за рамки нашей книги.

Суть в том, что идентификаторы могут совпадать, хотя объекты при этом не обязательно должны быть равными. Утверждение основывается на том, что алгоритм хеширования (создания идентификатора), используемый в методе hashCode(), может возвращать одинаковые значения для разных экземпляров. Это также означает, что все подобные объекты будут размещаться в одной области внутри HashSet (так как каждая область представлена единственным идентификатором), но в этом нет ничего страшного. Все это немного понизит эффективность HashSet (значительное понижение может произойти только при очень большом количестве элементов). Но если HashSet найдет несколько объектов в области, обозначенной идентификатором, оно просто применит метод equals(), чтобы проверить, есть ли точное совпадение. Другими словами, значения идентификаторов иногда применяются для ускорения поиска, но, чтобы найти точное совпадение, HashSet по-прежнему должно взять все элементы конкретной области (где находятся объекты с одинаковыми идентификаторами) и выполнить для каждого метод equals(), проверяя наличие дубликатов для области.

# Для случаев, когда множество должно оставаться отсортированным, существует TreeSet

Множество TreeSet, как и HashSet, предотвращает дублирование элементов. Кроме того, оно *сохраняет* их отсортированными. Это работает аналогично методу sort() – при создании экземпляра TreeSet с помощью конструктора, не принимающего аргументы, для сортировки будет применяться метод compareTo() каждого элемента. Но вместо этого вы можете передать Comparator в конструктор TreeSet. Недостаток множества заключается в том, что, если вам не *нужна* сортировка, вы все равно расплачиваетесь за нее снижением производительности. Но для большинства приложений этот эффект будет практически незаметен.

```
import java.util.*;
import java.io.*;
public class Jukebox8 {
   ArrayList<Song> songList = new ArrayList<Song>();
   int val;

   public static void main(String[] args) {
      new Jukebox8().go();
   }

   public void go() {
      getSongs();
      System.out.println(songList);
      Collections.sort(songList);
      System.out.println(songList);
      TreeSet<Song> songSet = new TreeSet<Song>();
      songSet.addAll(songList);
      System.out.println(songSet);
   }

   void getSongs() {
      try {
         File file = new File("SongListMore.txt");
         BufferedReader reader = new BufferedReader(new FileReader(file));
         String line =  null;
         while ((line= reader.readLine()) != null) {
             addSong(line);
         }

      } catch(Exception ex) {
         ex.printStackTrace();
      }
   }

   void addSong(String lineToParse) {
      String[] tokens = lineToParse.split("/");
      Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
      songList.add(nextSong);
   }
}
```

Создаем экземпляр TreeSet вместо HashSet. Вызов конструктора без аргументов означает, что для сортировки будут использоваться методы compareTo() из каждого элемента.

Но мы можем передать Comparator.

Можно добавить все песни из HashSet с помощью метода addAll(). Допустимо также добавлять каждую песню отдельно, используя вызов songSet.add(), как мы это делали с ArrayList.

# Что вы должны знать о TreeSet...

TreeSet выглядит простым, но убедитесь, что вы действительно понимаете, какие именно действия нужно выполнить для его использования. Этот вопрос показался нам настолько важным, что мы специально придумали упражнение. Не переворачивайте страницу, пока не выполните его. *Мы говорим серьезно.*

Наточите свой карандаш

Взгляните на этот код. Внимательно прочтите его, а затем ответьте на вопросы, размещенные ниже.

Примечание: в коде нет синтаксических ошибок.

```
import java.util.*;

public class TestTree {
   public static void main (String[] args) {
      new TestTree().go();
   }

   public void go() {
     Book b1 = new Book("Как устроены кошки");
     Book b2 = new Book("Постройте заново свое тело");
     Book b3 = new Book("В поисках Эмо");

     TreeSet<Book> tree = new TreeSet<Book>();
     tree.add(b1);
     tree.add(b2);
     tree.add(b3);
     System.out.println(tree);
   }
}

class Book {
   String title;
   public Book(String t) {
     title = t;
   }
}
```

1. Что выведет этот код, если его скомпилировать?

2. Каков будет результат выполнения класса TestTree, если код скомпилируется?

3. Как исправить проблемы в коде (на этапе компиляции или выполнения), если таковые возникнут?

# Элементы TreeSet должны реализовывать интерфейс Comparable

TreeSet не может прочитать мысли программиста, чтобы выяснить, как именно должны сортироваться объекты. Скажите об этом сами.

## Для использования TreeSet должно выполняться одно из этих условий.

### Элементы в списке обязаны иметь тип, который реализует интерфейс *Comparable*.

Класс Book на предыдущей странице не реализует Comparable, поэтому не будет работать. Только подумайте, единственная цель жизни TreeSet — поддержание ваших элементов в определенном порядке, но он понятия не имеет, как сортировать объекты Book! Он проходит компиляцию, так как метод add(), принадлежащий TreeSet, не принимает значения типа Comparable, а использует тип, который вы указали при создании TreeSet. Иначе говоря, если вы написали TreeSet<Book>(), то метод add() просто примет вид add(Book). И никто не требует, чтобы класс Book реализовывал Comparable! Но если вы попытаетесь добавить в множество второй элемент, программа выдаст ошибку. В этот момент множество впервые попытается вызвать метод compareTo() из первого элемента и... не сможет.

```
class Book implements Comparable {
   String title;
   public Book(String t) {
     title = t;
   }
   public int compareTo(Object b) {
     Book book = (Book) b;
     return (title.compareTo(book.title));
   }
}
```

**ИЛИ**

### При создании TreeSet вы используете перегруженный конструктор, который принимает *Comparator*.

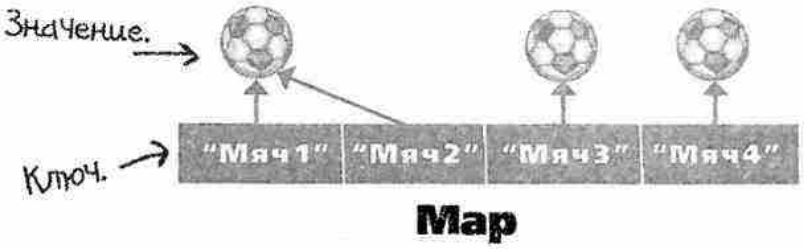
Принципы работы TreeSet и метода sort() во многом схожи. У вас есть выбор: использовать метод compareTo() из каждого элемента, реализующего интерфейс Comparable, или передать свою реализацию Comparator, где известно, как сортировать элементы внутри множества. Чтобы воспользоваться вторым вариантом, вызовите конструктор TreeSet, принимающий Comparator.

```
public class BookCompare implements Comparator<Book> {
   public int compare(Book one, Book two) {
      return (one.title.compareTo(two.title));
   }
}

class Test {
     public void go() {
     Book b1 = new Book("Как устроены кошки");
     Book b2 = new Book("Постройте заново свое тело");
     Book b3 = new Book("В поисках Эмо");
     BookCompare bCompare = new BookCompare();
     TreeSet<Book> tree = new TreeSet<Book>(bCompare);
     tree.add(new Book("Как устроены кошки");
     tree.add(new Book("В поисках Эмо");
     tree.add(new Book("Постройте заново свое тело");
     System.out.println(tree);
   }
}
```

# Мы уже имели дело со списками и множествами, пришло время поработать с отображениями

Списки и множества — это хорошо, но в некоторых случаях в качестве коллекции лучше использовать отображения (хотя они и считаются частью API для работы с коллекциями, они не реализуют интерфейс Collections).

Представьте, что вам нужна коллекция, которая ведет себя как список свойств: вы даете ей ключ, а она в ответ возвращает связанное с ним значение. Хотя в качестве ключей часто используются строки, Java позволяет сделать ключом любой объект (или даже упакованный примитив).




**Map**

**Каждый элемент в отображении на самом деле состоит из двух объектов: *ключа* и *значения*. У вас могут быть совпадающие значения, но дублирование ключей не допускается.**

## Пример отображения

```
import java.util.*;

public class TestMap {

   public static void main(String[] args) {

     HashMap<String, Integer> scores = new HashMap<String, Integer>();

     scores.put("Кэти", 42);
     scores.put("Берт", 343);
     scores.put("Скайлер", 420);

      System.out.println(scores);
      System.out.println(scores.get("Берт"));
   }
}
```

HashMap нуждается в двух типовых параметрах: для ключа и для значения.

Вместо add() используется put(), который по понятным причинам принимает два аргумента (ключ, значение).

Метод get() принимает ключ и возвращает значение (в данном случае Integer).

```
File Edit Window Help WhereAmI
%java TestMap

{Скайлер=420, Берт=343, Кэти=42}
343
```

При выводе на экран отображение представляется в виде пар ключ=значение, заключенных в фигурные скобки {} вместо угловых (которые можно наблюдать при выводе списков и множеств).

# Наконец-то возвращаемся к обобщениям

Помните, как раньше в этой главе мы говорили, что методы, принимающие параметры обобщенных типов, могут быть... *странными*? Странными с точки зрения полиморфизма. Если вы почувствуете, что перестаете улавливать суть, все равно продолжайте читать дальше — эта тема полностью раскрывается на следующих нескольких страницах.

Начнем с того, что вспомним, как работают (в полиморфическом смысле) *аргументы-массивы*, а затем перейдем к обобщенным спискам. Приведенный ниже код компилируется и запускается без ошибок.

**Если аргумент метода — это массив объектов Animal, то метод сможет принимать массивы любых потомков Animal.**

**Проще говоря, если метод объявлен как:**

***void foo(Animal[] a) {}***

**то вы можете делать следующие вызовы (подразумевается, что Dog унаследован от Animal):**

***foo(anAnimalArray);***

***foo(aDogArray);***

## Вот как он будет работать с обычными *массивами*:

```
import java.util.*;

public class TestGenerics1 {
   public static void main(String[] args) {
      new TestGenerics1().go();
   }

   public void go() {
      Animal[] animals = {new Dog(), new Cat(), new Dog()};
      Dog[] dogs = {new Dog(), new Dog(), new Dog()};
      takeAnimals(animals);
      takeAnimals(dogs);
   }

  public void takeAnimals(Animal[] animals) {
      for(Animal a: animals) {
        a.eat();
      }
   }
}
```

Объявляем и создаем массив типа Animal, который содержит объекты Cat и Dog.

Объявляем и создаем массив типа Dog, который содержит только элементы Dog (компилятор не позволит поместить в него объекты типа Cat).

Вызываем takeAnimals(), используя в качестве аргументов массивы обоих типов...

Это ключевой момент. Метод takeAnimals() может принимать как Animal[], так и Dog[], так как Dog является Animal. Полиморфизм в действии.

Не забывайте, что можно вызывать только те методы, которые были объявлены в классе Animal, так как именно он содержит массив, переданный в метод (приведение типов мы не делали, потому что массив может хранить и Dog, и Cat).

---

```
abstract class Animal {
   void eat() {
     System.out.println("животное ест");
   }
}
class Dog extends Animal {
   void bark() { }
}
class Cat extends Animal {
   void meow() { }
}
```

Упрощенная иерархия наследования класса Animal.

# Использование полиморфических аргументов и обобщений

Итак, мы узнали, как все это работает с массивом, но что будет, если мы заменим его объектом ArrayList? Справедливый вопрос, не так ли?

Для начала попробуем проверить это на примере единственного ArrayList с параметризованным типом Animal. Мы внесли несколько изменений в метод go().

## Передаем только ArrayList<Animal>

Просто меняем Animal[] на ArrayList<Animal>.

```
public void go() {
    ArrayList<Animal> animals = new ArrayList<Animal>();
    animals.add(new Dog());
    animals.add(new Cat());
    animals.add(new Dog());

    takeAnimals(animals);
}
```

Мы должны передавать по одному элементу. Здесь нет сокращенного синтаксиса, как при создании массивов.

Этот код практически не изменен, только переменная animal указывает на ArrayList вместо массива.

```
public void takeAnimals(ArrayList<Animal> animals) {
    for(Animal a: animals) {
    a.eat();
    }
}
```

Теперь метод принимает ArrayList вместо массива, а в остальном ничего не изменилось. Помните, что синтаксис цикла for применим как для массивов, так и для коллекций.

## Компилируется и работает без проблем

```
File Edit Window Help CatFoodIsBetter
%java TestGenerics2

животное ест
животное ест
животное ест
животное ест
животное ест
животное ест
```

# Но сработает ли это для ArrayList<Dog>?

Учитывая полиморфизм, компилятор позволяет нам передавать массив типа Dog туда, куда можно передать массив типа Animal. Нет проблем. И ArrayList<Animal> разрешено передать в метод с таким же аргументом. Вопрос в том, можно ли на место аргумента ArrayList<Animal> подставить объект ArrayList<Dog>? Если это сработало с массивами, почему это не может произойти и с коллекциями?

### Передаем только ArrayList<Dog>

```
public void go() {
   ArrayList<Animal> animals = new ArrayList<Animal>();
   animals.add(new Dog());
   animals.add(new Cat());
   animals.add(new Dog());
   takeAnimals(animals);
   ArrayList<Dog> dogs = new ArrayList<Dog>();
   dogs.add(new Dog());
   dogs.add(new Dog());
   takeAnimals(dogs);
}

public void takeAnimals(ArrayList<Animal> animals) {
     for(Animal a: animals) {
     a.eat();
    }
 }
```

Мы знаем, что эта строка будет работать.

Создаем ArrayList и добавляем в него несколько объектов Dog.

Будет ли это работать теперь, когда мы изменили массив на ArrayList?

### При компиляции мы получим:

```
File Edit Window Help CatsAreSmarter
%java TestGenerics3

TestGenerics3.java:21: takeAnimals(java.util.
ArrayList<Animal>) in TestGenerics3 cannot be
applied to (java.util.ArrayList<Dog>)
      takeAnimals(dogs);
      ^
1 error
```

Это выглядело очень хорошо, но что-то пошло не так...

## Что бы случилось, если бы так можно было делать...

Представьте, что компилятор снял для вас это ограничение. Он позволил передавать ArrayList<Dog> в метод, объявленный как:

```
public void takeAnimals(ArrayList<Animal> animals) {
   for(Animal a: animals) {
      a.eat();
   }
}
```

На первый взгляд, в методе нет ничего коварного, не так ли? В конце концов суть полиморфизма заключается в том, что Dog может делать все то же самое, что и Animal (в частности, мы рассматриваем метод eat()). Что же плохого в том, чтобы вызывать метод eat() из каждой ссылки Dog?

*Ничего*. Совершенно ничего.

С *этим* кодом все в порядке. Но представьте, что вместо него мы имеем:

```
public void takeAnimals(ArrayList<Animal> animals) {
   animals.add(new Cat());
}
```

← Ой! Мы только что добавили объект Cat в ArrayList, который, как предполагается, должен хранить только объекты Dog!

В этом и проблема. Никто не запрещает добавлять объекты Cat в ArrayList<Animal>. Более того, в этом вся суть использования для ArrayList родительского типа наподобие Animal — вы можете размещать всех его потомков в едином списке.

Но если вы передадите ArrayList типа Dog (пригодный для хранения только объектов Dog) в метод, который принимает ArrayList типа Animal, то рискуете столкнуться с экземпляром класса Cat в списке объектов Dog. Если компилятор позволит вам передать в такой метод ArrayList<Dog>, то во время выполнения программы кто угодно сможет добавить в список объект типа Cat. И чтобы этого не случилось, компилятор просто не позволит вам рисковать.

***Если вы объявите метод, который принимает ArrayList<Animal>, то он не сможет принимать ничего другого: ни ArrayList<Dog>, ни ArrayList<Cat>.***

Погодите, если единственная причина, по которой ArrayList типа Dog нельзя передавать в метод, принимающий ArrayList типа Animal, заключается в том, чтобы не позволить поместить объект Cat в список объектов Dog, то почему это работает с *массивами*? Разве с ними нет таких проблем? Ведь мы можем добавить объект Cat в Dog[].

## Типы элементов массива повторно проверяются при выполнении программы, тогда как тип коллекции — только на этапе компиляции.

Представьте, что вы добавили объект Cat в массив Dog[] (а сам массив передали в метод, который принимает Animal[], что для массивов абсолютно допустимо).

```
public void go() {
   Dog[] dogs = {new Dog(), new Dog(), new Dog()};
   takeAnimals(dogs);
}

public void takeAnimals(Animal[] animals) {
   animals[0] = new Cat();
}
```

Мы добавили новый экземпляр Cat в массив типа Dog. Компилятор позволил это сделать, так как знает, что вы можете передавать в метод массивы типа Cat или Animal. С его точки зрения проблем может не быть.

**Код компилируется, но если мы его запустим:**

Ого! По крайней мере JVM это остановила.

```
File Edit Window Help CatsAreSmarter
%java TestGenerics1
Exception in thread "main" java.lang.ArrayStoreException: Cat
        at TestGenerics1.takeAnimals(TestGenerics1.java:16)
        at TestGenerics1.go(TestGenerics1.java:12)
        at TestGenerics1.main(TestGenerics1.java:5)
```

Ах, если бы можно было использовать полиморфические типы коллекций в качестве аргументов для методов, чтобы ветеринарная программа смогла принимать как списки объектов Dog, так и списки объектов Cat. Я бы смогла пройтись по элементам этих списков, вызывая метод immunize() и не рискуя случайно добавить экземпляр Cat в список для Dog. Ах, мечты, мечты...

# На помощь приходят заполнители

Может, это звучит необычно, но *есть* способ создать аргумент для метода, способный принимать ArrayList любого дочернего от Animal типа. Проще всего сделать это с помощью *заполнителя*, который добавлен в язык Java именно для таких целей.

```
public void takeAnimals(ArrayList<? extends Animal> animals) {
   for(Animal a: animals) {
      a.eat();
   }
}
```

Помните, ключевое слово extends в данном случае означает либо наследование, либо реализацию в зависимости от типа. Таким образом, если вы хотите использовать ArrayList, чей тип реализует интерфейс Pet, то можете выразить это так:

ArrayList<? extends Pet>

Сейчас вы, наверное, думаете: «В чем же *разница*? Разве прежде мы не сталкивались с подобной проблемой? Метод, приведенный выше, не делает ничего опасного (у любого подтипа Animal он гарантированно есть), но разве нельзя это изменить, добавив экземпляр Cat в список *Animal*, даже если на самом деле это ArrayList<Dog>? И что нового дает нам заполнитель, если код не проверяется при выполнении программы?»

И правильно делаете, что так думаете. Если в объявлении указан заполнитель <?>, компилятор не позволит вам выполнить добавление!

**Когда вы используете заполнитель как аргумент для своего метода, компилятор не дает вам сделать ничего, что может повредить список, на который ссылается параметр этого метода.**

**Вы по-прежнему можете вызывать методы из элементов списка, но нельзя ничего добавлять в него.**

**Иными словами, разрешено манипулировать элементами списка, но запрещено добавлять в него новые элементы. Вы защищены во время выполнения программы, так как компилятор не позволяет сделать ничего, что могло бы нарушить работу приложения.**

**Таким образом, внутри takeAnimals() можно написать:**

```
for(Animal a: animals) {
   a.eat();
}
```

**Но это уже не скомпилируется:**

```
animals.add(new Cat());
```

# Альтернативный синтаксис для выполнения тех же задач

Вы, вероятно, помните, что метод sort(), который мы рассматривали, использовал обобщенный тип, но в необычном формате. Тогда типовой параметр был объявлен перед типом возвращаемого значения. Это лишь еще один способ описания типового параметра, но результат тот же:

**Это:**

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

**делает то же самое, что и это:**

```
public void takeThing(ArrayList<? extends Animal> list)
```

## Это не глупые вопросы

**В: Если обе синтаксические конструкции делают одно и то же, то какую из них выбрать?**

О: Все зависит от того, хотите ли вы использовать «T» где-нибудь еще. Например, вам нужно получить метод с двумя аргументами в виде списков, и типы этих списков наследуют Animal. В таком случае эффективнее будет объявить типовой параметр только один раз:

```
public <T extends Animal> void takeThing(ArrayList<T> one, ArrayList<T> two)
```

вместо того чтобы писать:

```
public void takeThing(ArrayList<? extends Animal> one,
                      ArrayList<? extends Animal> two)
```

# Поработайте компилятором (повышенная сложность)

Ваша задача — притвориться компилятором и определить, какие из этих выражений скомпилируются. Однако некоторые фрагменты представленного кода в этой главе не рассматривались, поэтому вам нужно найти ответы, исходя из имеющейся информации и применяя «правила» в новых для вас условиях. Иногда придется угадывать, но смысл упражнения — найти подходящий ответ, основываясь на уже полученных знаниях.

Примечание: подразумевается, что код находится внутри корректно написанных классов и методов.

**Скомпилируется?**

- ☐ `ArrayList<Dog> dogs1 = new ArrayList<Animal>();`
- ☐ `ArrayList<Animal> animals1 = new ArrayList<Dog>();`
- ☐ `List<Animal> list = new ArrayList<Animal>();`
- ☐ `ArrayList<Dog> dogs = new ArrayList<Dog>();`
- ☐ `ArrayList<Animal> animals = dogs;`
- ☐ `List<Dog> dogList = dogs;`
- ☐ `ArrayList<Object> objects = new ArrayList<Object>();`
- ☐ `List<Object> objList = objects;`
- ☐ `ArrayList<Object> objs = new ArrayList<Dog>();`

Решение усложненной задачи «Программирование задом наперед»

```
import java.util.*;

public class SortMountains {

  LinkedList<Mountain> mtn = new LinkedList<Mountain>();

  class NameCompare implements Comparator <Mountain> {
    public int compare(Mountain one, Mountain two) {
      return one.name.compareTo(two.name);
    }
  }
  class HeightCompare implements Comparator <Mountain> {
    public int compare(Mountain one, Mountain two) {
      return (two.height - one.height);
    }
  }
  public static void main(String [] args) {
    new SortMountain().go();
  }
  public void go() {
    mtn.add(new Mountain("Лонг-Рейндж", 14255));
    mtn.add(new Mountain("Эльберт", 14433));
    mtn.add(new Mountain("Марун", 14156));
    mtn.add(new Mountain("Касл", 14265));
    System.out.println("В порядке добавления:\n" + mtn);
    NameCompare nc = new NameCompare();
    Collections.sort(mtn, nc);
    System.out.println("По названию:\n" + mtn);
    HeightCompare hc = new HeightCompare();
    Collections.sort(mtn, hc);
    System.out.println("По высоте:\n" + mtn);
  }
}

class Mountain {
  String name;
  int height;

  Mountain(String n, int h) {
    name = n;
    height = h;
  }
  public String toString( ) {
    return name + " " + height;
  }
```

Вы заметили, что список высот — это нисходящая последовательность? :)

**Результат:**

```
File Edit Window Help ThisOne'sForBob
%java SortMountains
В порядке добавления:
[Лонг-Рейндж 14255, Эльберт 14433, Марун 14156, Касл 14265]
По названию:
[Касл 14265, Лонг-Рейндж 14255, Марун 14156, Эльберт 14433]
По высоте:
[Эльберт 14433, Касл 14265, Лонг-Рейндж 14255, Марун 14156]
```

## Решение упражнения

**Возможные ответы:**

**Comparator,**

**Comparable,**

**compareTo( ),**

**compare( ),**

**да,**

**нет**

Дано следующее компилируемое выражение:

```
Collections.sort(myArrayList);
```

1. Что должен реализовывать класс, объекты которого хранятся в `myArrayList`? **Comparable**
2. Какой метод должен реализовывать класс, объекты которого хранятся в `myArrayList`? **compareTo( )**
3. Может ли класс, объекты которого хранятся в `myArrayList`, одновременно реализовывать `Comparator` и `Comparable`? **да**

Дано следующее компилируемое выражение:

```
Collections.sort(myArrayList, myCompare);
```

4. Может ли класс объектов, хранящихся в `myArrayList`, реализовывать интерфейс `Comparable`? **да**
5. Может ли класс объектов, хранящихся в `myArrayList`, реализовывать интерфейс `Comparable`? **да**
6. Должен ли класс объектов, хранящихся в `myArrayList`, реализовывать интерфейс `Comparable`? **нет**
7. Должен ли класс объектов, хранящихся в `myArrayList`, реализовывать интерфейс `Comparable`? **нет**
8. Что должен реализовывать класс объекта `myCompare`? **Comparator**
9. Какой метод должен реализовывать класс объекта `myCompare`? **compare( )**

# Ответы

## Поработайте компилятором

**Скомпилируется?**

- [ ] `ArrayList<Dog> dogs1 = new ArrayList<Animal>();`
- [ ] `ArrayList<Animal> animals1 = new ArrayList<Dog>();`
- [x] `List<Animal> list = new ArrayList<Animal>();`
- [ ] `ArrayList<Dog> dogs = new ArrayList<Dog>();`
- [ ] `ArrayList<Animal> animals = dogs;`
- [x] `List<Dog> dogList = dogs;`
- [x] `ArrayList<Object> objects = new ArrayList<Object>();`
- [x] `List<Object> objList = objects;`
- [ ] `ArrayList<Object> objs = new ArrayList<Dog>();`

# Выпусти свой код

**Пришло время его отпустить.** Вы написали код. Вы его протестировали и откорректировали. Вы рассказали всем знакомым, что больше не желаете видеть ни единой строки из него. По большому счету вы создали произведение искусства. Это то, что действительно работает! Но что дальше? *Как* вы доставите его пользователям? *Что* именно они получат? Вы даже не знаете, к кому попадет ваше приложение! В последних двух главах мы расскажем, как организовывать, упаковывать и развертывать (внедрять или доставлять) код на языке Java. Мы рассмотрим локальный, полулокальный и удаленный варианты развертывания, включая исполняемые Java-архивы (JAR), Java Web Start, RMI и сервлеты. Большую часть этой главы мы посвятим организации и упаковыванию вашего кода — это вещи, которые нужно знать вне зависимости от выбранного варианта доставки. Последнюю главу мы завершим рассмотрением одной из самых впечатляющих возможностей Java. Расслабьтесь. Выпуская свой код, вы не прощаетесь с ним. Ведь его еще нужно поддерживать...