12+

Объектно ориентированное программирование на Java

Тимур Машнин

Тимур Машнин

Объектно-ориентированное программирование на Java. Платформа Java SE

http://www.litres.ru/pages/biblio_book/?art=44074402
ISBN 9785005039606

Аннотация

Эта книга предназначена для тех, кто хочет научиться программировать на языке Java.С этой книгой вы обучитесь объектно-ориентированному программированию на платформе Java SE и научитесь применять принципы ООП на практике.Эта книга охватывает важные аспекты программирования на языке Java, начиная с основ и заканчивая объектно-ориентированным подходом и командной разработкой кода.

Содержание

Объектно-ориентированное программирование на Java Платформа Java SE

Тимур Машнин

ISBN 978-5-0050-3960-6
Создано в интеллектуальной издательской системе Ridero

Введение

Объектно-ориентированное программирование на Java

Введение
Лекция
Общий обзор технологии Java

На этом курсе мы будем изучать технологию Java.
Итак, что такое технология Java?
Начнем с самого понятия технологии программирования.

Технология программирования – совокупность методов и инструментов, позволяющих создавать программное обеспечение

Технология Java – это язык программирования Java и платформа Java.

Можно сказать, что *технология программирования* – это совокупность методов и инструментов, позволяющих создавать программное обеспечение.

Технологии программирования могут иметь различный уровень применения. В процессе разработки программного обеспечения могут применяться технологии, решающие как конкретные задачи, так и технологии, являющиеся платформой для создания частей приложения или всего приложения.

Поэтому, как правило, для создания программного обеспечения применяется целый набор различных технологий.

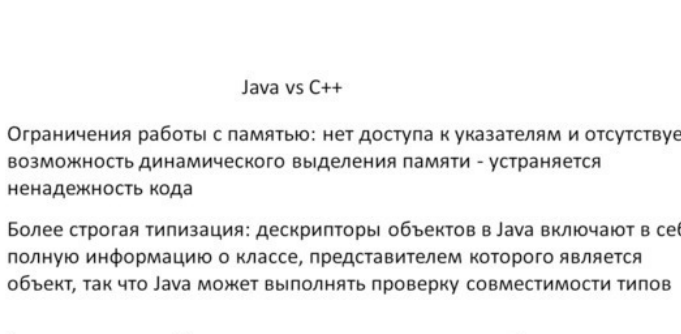
Применительно к Java, *технология Java* – это язык программирования Java и платформа Java.

Язык программирования Java представляет собой объектно-ориентированный язык программирования, имеющий

синтаксис, близкий к синтаксису языка C++.

Отличия языка Java от языка C++ обусловлены самим происхождением этих языков программирования.

Язык C++ является расширением языка C, который создавался как язык системного программирования.



Язык Java, в свою очередь, создавался для решения задач сетевого программирования и является самостоятельным языком программирования.

Главные отличия языка Java от языка C++ – это более строгая типизация, ограничения работы с памятью, автоматическая сборка мусора.

Понятно, что для создания программного обеспечения наличие одного языка программирования недостаточно.

Для компилируемых языков нужны инструменты, компилирующие исходный код в машинный, исполняемый операционной системой компьютера.

Для интерпретируемых языков программирования необходимы интерпретаторы, выполняющие исходный код в операционной системе.

В случае языка Java, реализация платформы Java как раз и обеспечивает выполнение Java-кода в операционной системе компьютера.

Таким образом, для того чтобы Java-приложение могло быть запущено, необходима реализация платформы Java.

Мы упомянули реализацию платформы Java.

Что это такое?

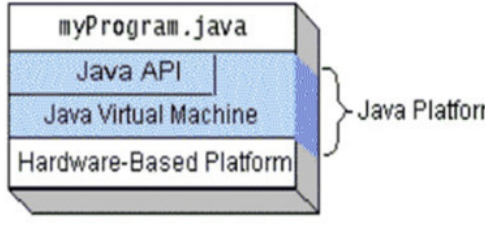
Платформа Java состоит из виртуальной машины Java Virtual Machine (JVM) и библиотек интерфейса программирования Java Application Programming Interface (API).

Реализация платформы Java – это конкретная реализация JVM для конкретной операционной системы плюс библиотеки Java API

Для всех распространенных операционных систем существуют свои виртуальные машины JVM, тем самым реализуется принцип «Write Once, Run Anywhere» – написанное однажды, работает везде.

Реализация платформы Java – это конкретная реализация JVM для конкретной операционной системы плюс библиотеки Java API.

На самом деле компанией Oracle для выполнения Java-приложений предоставляется набор сред выполнения *Java Runtime Environment (JRE),* охватывающий все распространенные операционные системы.



Виртуальная машина JVM составляет основную часть среды выполнения Java Runtime Environment (JRE).

Помимо JVM JRE содержит базовые библиотеки API, необходимые для выполнения Java-приложений, а также дополнительные инструменты, включая Java Plug-in – для запуска апплетов в браузере и Java Web Start – для развертывания Java-приложений через Интернет.

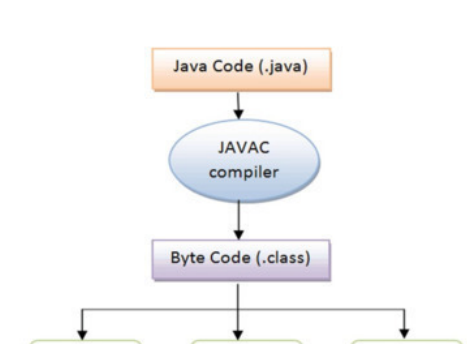
Компанией Oracle также предоставляется минимальный комплект разработки Java-приложений *Java Development Kit (JDK)*, состоящий из набора инструментов, включая компилятор в байт-код javac, документации, примеров и среды выполнения JRE.

Язык программирования Java является одновременно и интерпретируемым, и компилируемым. Причина этого

кроется в устройстве виртуальной машины JVM.

Виртуальная машина JVM – это набор специальных программ, созданных для конкретной операционной системы.

Точкой входа в виртуальную машину JVM является программа java, запускающая Java-приложение.



Приложения, написанные на языке Java, представляют собой текстовые файлы с расширением. java.

Чтобы JVM выполнила Java-приложение, приложение должно быть откомпилировано в специальный двоичный формат – *байт-код*.

Откомпилированное Java-приложение состоит из файлов с расширением. class, которые могут быть упакованы в архивный исполняемый файл с расширением. jar.

При запуске Java-приложения на вход JVM подается байт-код Java-приложения, а также байт-код используемых приложением библиотек Java API.

Виртуальная машина JVM может выполнять приложения, написанные и на других языках программирования – Scala, Groovy, Ruby, PHP, JavaScript, Python и др., при этом приложения также должны быть откомпилированы в байт-код.

В процессе обработки байт-кода виртуальная машина JVM производит его интерпретацию, т.е. выполняет команды, содержащиеся в байт-коде, или использует компилятор Just-in-time compilation (JIT), который транслирует байт-код в машинный код непосредственно во время выполнения Java-приложения, и тем самым увеличивает скорость обработки байт-кода.

Таким образом, язык Java является компилируемым, потому что необходима компиляция исходного кода в промежуточный по отношению к машинному байт-коду, и интерпретируемым, потому что байт-код не может быть исполнен самой операционной системой компьютера, а должен интерпретироваться.

Платформа Java содержит два типа JVM:

Java HotSpot Client VM (Client VM). Вызывается опцией –client инструмента java и обеспечивает быстрый запуск и потребление небольшого объема оперативной памяти.

JVM:

- Java HotSpot Client VM (Client VM): вызывается опцией – client инструмента java и обеспечивает быстрый запуск и потребление небольшого объема оперативной памяти.

- Java HotSpot Server VM (Server VM): вызывается опцией – server инструмента java и обеспечивает максимальную скорость выполнения приложения.

Java HotSpot Server VM (Server VM). Вызывается опцией — server инструмента java и обеспечивает максимальную скорость выполнения приложения.

Для обеих JVM технология Java HotSpot оптимизирует обработку байт-кода, распределение памяти, сборку мусора и управление потоками.

Технология Java – это общее понятие, на самом деле обозначающее широкий спектр Java-технологий.

Среда выполнения JRE и комплект разработки JDK являются основными продуктами платформы Java Platform, Standard Edition (Java SE).

Как уже было сказано, платформа Java содержит библиотеки интерфейса программирования Java API. Для чего они предназначены и какую роль они выполняют?

Библиотеки Java API – это готовые классы и интерфейсы, обеспечивающие для создаваемых Java-приложений общую функциональность.

Библиотеки Java API – это готовые классы и интерфейсы, обеспечивающие для создаваемых Java-приложений общую функциональность.

С библиотеками Java API программисту не нужно самому реализовывать ввод-вывод, сетевое соединение, создавать стандартные графические компоненты для интерфейса пользователя и многое-многое другое.

Все это уже предоставлено технологией Java.

Платформа Java SE является основой для всех остальных платформ технологии Java. Все вместе Java-платформы обеспечивают применение технологии Java к широкому диапазону устройств – от смарт-карт, встроенных и мобильных устройств до серверов и суперкомпьютеров.

Технология Java представлена следующими платформами:

Java Platform, Standard Edition (Java SE) – предоставляет среду выполнения и набор технологий и библиотек API для создания и запуска серверных и настольных приложений, апплетов и является основой для остальных платформ.

Java Platform, Standard Edition (Java SE) – предоставляет среду выполнения и набор технологий и библиотек API для создания и запуска серверных и настольных приложений, апплетов и является основой для остальных платформ.

Java SE Embedded – предназначена для встроенных систем, таких как интеллектуальные маршрутизаторы и коммутаторы, профессиональные принтеры и др.

Java Platform, Micro Edition (Java ME) – содержит набор сред выполнения и библиотек API, предназначенных для встроенных и мобильных устройств.

Java Card – позволяет создавать и запускать небольшие приложения (Java Card-апплеты) в смарт-картах и других устройствах.

Java Platform, Enterprise Edition (Java EE) – является расширением платформы Java SE и добавляет библиотеки, позволяющие создавать распределенные, многоуровневые серверные Java-приложения.

Кроссплатформенность обеспечивается наличием сред выполнения для различных операционных систем.

Платформа Java SE включает в себя следующие компоненты – среду выполнения Java Runtime Environment (JRE) и комплект разработчика приложений Java Development Kit (JDK).

Java SE Embedded – предназначена для встроенных си-

стем, таких как интеллектуальные маршрутизаторы и коммутаторы, профессиональные принтеры и др.

Платформа Java SE for Embedded обеспечивает ту же функциональность, что и платформа Java SE, дополнительно добавляя поддержку для платформ, специфических для встроенных систем, оптимизацию использования памяти, а также предоставляя уменьшенную среду выполнения и опцию Headless для устройств, не имеющих дисплея, мышки или клавиатуры.

Java Platform, Micro Edition (Java ME) – содержит набор сред выполнения и библиотек API, предназначенных для встроенных и мобильных устройств. В настоящее время активно применяется для Интернет вещей.

Java Card – позволяет создавать и запускать небольшие приложения (Java Card-апплеты) в смарт-картах и других устройствах с очень ограниченными ресурсами, таких как SIM-карты мобильных телефонов, банковские карточки, карты доступа цифрового телевидения и др.

Java Platform, Enterprise Edition (Java EE) – является расширением платформы Java SE и добавляет библиотеки, позволяющие создавать распределенные, многоуровневые серверные Java-приложения.

Если сравнивать язык Java с такими распространенными языками как C#, JavaScript, Python и PHP,

То сравнивая Java с C#, который работает на платформе NET, с точки зрения разработчика языки Java и C# очень

похожи.

Но у них есть некоторые синтаксические различия, и язык Java считается более простым языком.

Кроме того, C# все таки больше привязан к платформе Windows.

Так как эти два языка очень похожи, при их сравнении возникают большие дискуссии, в которые мы сейчас углубляться не будем.

Если сравнивать Java и JavaScript, язык JavaScript является только интерпретируемым и выполняется только в веб-браузерах.

Если сравнивать Java и Python, то Python также является компилируемым и интерпретируемым языком, но с полной динамической типизацией, он проще в изучении, но проигрывает в скорости Java, хотя для него есть альтернативные реализации интерпретаторов: Jython, Cython и другие.

По поводу сравнения Java и Python также ведутся жаркие дискуссии.

Если сравнивать Java и PHP, то PHP это скриптовый серверный язык для разработки веб приложений, он проще в изучении и является языком с динамической типизацией. PHP не предназначен для крупных проектов, однако, PHP хостинг более распространен, чем для Java.

Как видно у каждого языка есть свои плюсы и минусы, но «Вы должны писать на языке, который делает вас счастливее», как сказал Пэт Аллан.

Выражения

Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 1

Выражения

Теперь каким способом можно хорошо представить, что такое компьютер?

И какие есть концепции языка программирования?

Вместо того, чтобы начинать с нуля, мы начнем с устройства, которое вам очень хорошо известно, а именно, калькулятора.

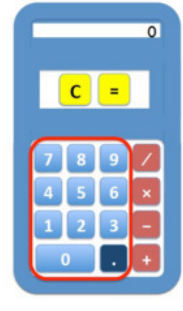
И мы постепенно преобразуем простой калькулятор в компьютер.

Сделав это, мы также перейдем от ввода последовательности клавиш калькулятора к компьютерной программе.

Таким образом, вы гораздо лучше поймете концепции, на которых основывается компьютер и языки программирования, такие как Java.

Вы использовали калькулятор много раз.

И вы знаете из чего он состоит.



Здесь есть клавиши с цифрами, которые помогут вам составить число.

Составленные числа отображаются на дисплее.

И тогда вы можете выполнять операции с этими числами,

Для которых вы используете другие клавиши, которые представляют эти операции.

Мы начнем с рассмотрения базового калькулятора.

Таким образом, эти операции могут быть сложение, вы-

читание, умножение, и деление.

И калькулятор состоит из трех основных частей – дисплея, контрольной части, где есть завершающая вычисления клавиша равно и клавиша сброса, и клавиатура с цифрами и операциями.

Теперь, используя цифры, вы можете писать выражения и запрашивать их вычисления.

Выражение содержит числа и арифметические операции.

Эти выражения называются числовыми выражениями.

Теперь давайте улучшим этот калькулятор.

Но сначала давайте поговорим о выражениях.

Как правило, мы думаем о выражениях математически.

Это выражение равно другому выражению или какому-либо значению.

И это очень хорошая абстракция в большинстве случаев.

Но на самом деле мы знаем, что вычисление выражения требует усилий и времени.

И если у нас есть более сложное выражение, в нем может быть порядок, согласно которому вычисляются разные части этого выражения.

И вычисление более сложного выражения может занять больше времени.

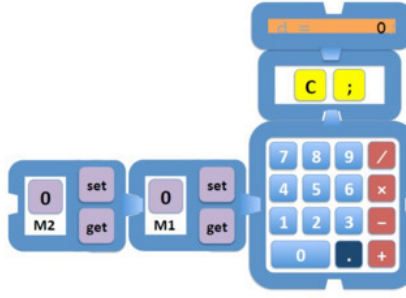
Но что более важно, представьте, что у нас есть сложное выражение, и мы вычислили его один раз.

И мы должны снова вычислить его, если мы хотим позже получить значение выражения.

Хотя было бы неплохо иметь некий способ запомнить значение выражения для будущего использования?

Поэтому, рассмотрим такой калькулятор, где у нас есть запоминание.

Здесь у нас есть несколько клавиш для хранения или получения значений из этой памяти.



Функция запоминания позволяет нам сохранить значение для будущего использования.

Память может содержать значение, и могут быть связанные с ней операции, такие как MS, чтобы сохранить значение, и MR, чтобы восстановить его или вызвать его.

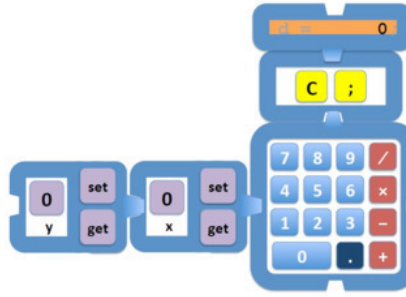
Иногда есть третья клавиша, MC для очистки памяти,

Назовем эти две клавиши для работы с памятью set и get.

Сейчас ячейки памяти названы предопределенными именами, M1, M2 и т. д.

Но мы хотели бы назвать их x и y, как мы привыкли в математике.

И мы будем присваивать этим ячейкам памяти имена переменных.



Теперь мы обсудим, что такое начальное значение переменной, которое сохраняется до того, как мы установим переменную в другое значение.

Мы можем сказать, что значение переменной неопределенно.

Поэтому, если мы попытаемся получить это значение, мы получим ошибку.

В калькуляторах, где есть числовые переменные, эта переменная обычно устанавливается равной 0, чтобы избежать ошибки.

Теперь мы хотим, чтобы дисплей показывал что-то, когда мы нажимаем кнопки Set или Get.

Давайте сначала поговорим о Set.

Предположим, что дисплей показывает число 3, и что мы нажимаем кнопку set переменной x.

Теперь значение 3 будет храниться в переменной x.

И дисплей может показать что-то вроде x равно 3 точка с запятой,

Чтобы записать то, что мы только что сделали.

Мы говорим, что мы назначили значение 3 переменной X,

и записали это как x равно 3 в инструкции присваивания.

Как только мы установили значение переменной, мы можем использовать это значение в выражениях.

Например, представьте, что у нас есть 5 на дисплее,

И мы хотим добавить значение x.

Мы нажимаем символ плюса, а затем кнопку Get x.

Таким образом, мы увидим на дисплее 5 плюс x.

Но это выражение, и до того, как мы используем оператор присваивания, что дисплей действительно отображает, выражение или законченную операцию?

Мы можем рассматривать выражения в калькуляторе как законченные операции, считая, что дисплей также может считаться переменной, переменной с прямым вводом.

Поэтому на дисплее написано d равно перед выражением.
Таким образом мы преобразуем выражение в операцию.
На слайде показаны различные выражения присваивания.

```
x = 3;
x = 3+4;
x = 3+y;
x = 3+x;
```

<Variable> = <Expression> ;

Здесь показано, что выражения могут также иметь переменные.

И для вычисления выражения, нам нужно найти сохраненное значение в соответствующих переменных.

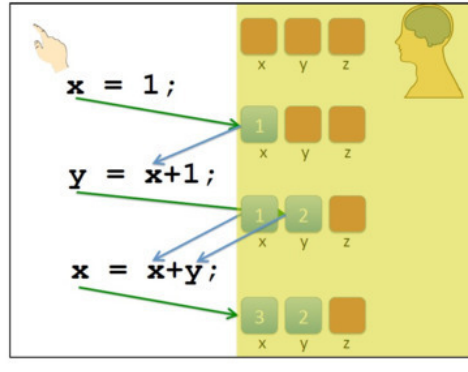
Теперь может оказаться, что одна и та же переменная появляется как слева, так и справа от присваивания.

Давайте проанализируем это более подробно.

Но сначала, давайте вспомним, что выражение присваивания состоит из переменной, за которой следует символ равенства, за которым следует выражение для вычисления, ко-

торое завершается точкой с запятой.

Представьте, что мы имеем три переменные x, y и z.



Мы не знаем их начальных значений.

У нас есть первая операция, которая присваивает 1 переменной x.

Поэтому после выполнения содержимое переменной x равно 1.

Следующая операция присваивания y равно x плюс 1.

Сначала мы должны оценить выражение справа, x плюс 1.

Для этого нам нужно получить значение, сохраненное в x.

Поэтому мы получаем 2 и 2 сохраняем в y.

Мы всегда работаем справа налево.

Сначала вычисляем выражение, а затем сохраняем ре-

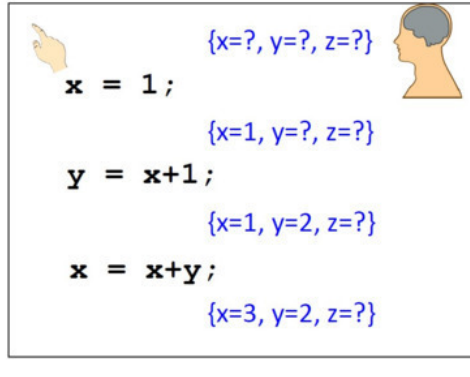
зультат в переменной.

Теперь мы сначала получаем значения x и y, складываем их вместе, получаем 3 и сохраняем 3 в x.

Переменные вместе со значениями – это то, что мы называем состоянием.

Таким образом, оператор присваивания преобразует одно состояние в другое состояние.

Здесь состояния обозначены фигурными скобками.



Коллекция значений переменных – это состояние.

Поэтому присваивание приводит к изменению состояния.

Теперь представьте, что вы сегодня делаете расчеты, и вы хотите повторить те же самые вычисления завтра.

Для этого вам нужно будет ввести все выражения снова.

Поэтому мы хотели бы иметь возможность записывать вычисления.

Точно так же, как мы хотим использовать память переменных для хранения значений, мы хотели бы теперь сохранить всю программу.

Некоторые калькуляторы печатают вычисления на бумаге.

Таким образом, у нас может быть запись наших вычислений.

Мы называем эту запись программой.

На данный момент программа является последовательностью простых вычислений.

Теперь было бы здорово, если бы мы могли повторно использовать программу, чтобы программа была не только результатом записи калькулятора, чтобы мы имели возможность подавать эту программу в калькулятор, как инструкции для повторного расчета.

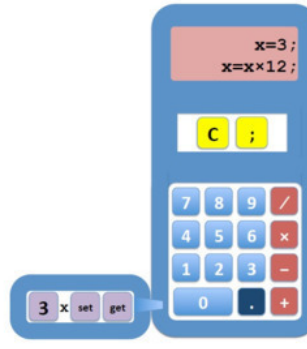
Теперь наш калькулятор становится все больше похож на компьютер.

Таким образом, последовательность инструкций является программой.

Этот набор инструкций должен быть четко определен, и каждая из инструкций должна эффективно исполняться нашим компьютером.

Теперь эти инструкции обычно представляют собой текст.

Расширенный калькулятор выглядит следующим образом.



На дисплее теперь отображается история операций.

Это то, что мы называем программой.

Помимо записи истории операций, мы также хотим возможность ввода этой истории в калькулятор.

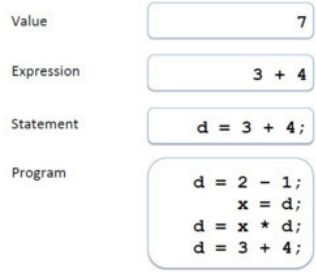
Таким образом, мы получим простой компьютер.

В общем и целом, программа является не чем иным, как записанным вычислением.

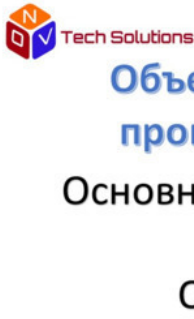
Ее можно записать на листе бумаги или сохранить другим способом, например, в памяти компьютера.

И компьютер будет интерпретировать эту программу и выполнять вычисление каждый раз, когда это потребуется.

Таким образом, мы прошли путь от значения и выражения до программы.


Value
7
Expression
3 + 4
Statement
d = 3 + 4;
Program
d = 2 - 1;
x = d;
d = x * d;
d = 3 + 4;

Основные операторы



Калькулятор, которые мы рассматривали, работал с числами.

Мы использовали числа и операции с числами для получения чисел.

Теперь, что делать, если вы хотите сравнить два числа?

Если мы хотим проверить, например, 5 меньше 6 или нет.

Ответ может быть положительным или отрицательным, – да или нет.

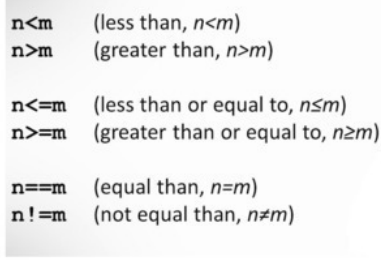
Это будет утверждение истинное или ложное.

В этом случае true и false также являются значениями,

но они не являются числовыми значениями.

Их называют булевыми значениями в честь математика Джорджа Була.

Существует шесть операций сравнения – меньше чем, больше чем, меньше или равно, больше или равно.



И наконец, мы должны проверить, являются ли два значения равными или разными.

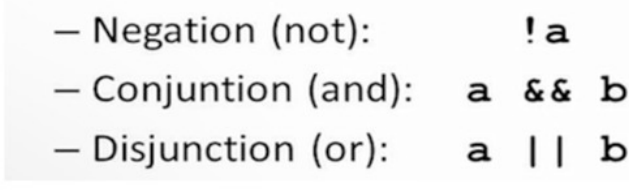
Результатом проверки будет булево значение true или false.

Булевы значения представляют собой тип данных с двумя значениями true и false.

Мы могли бы назвать их да или нет, или один и ноль, но мы будем называть их true и false, как это делает Java.

И так же, как у нас были арифметические операции, теперь мы имеем несколько булевых операций.

Давайте посмотрим на некоторые из них.



Отрицание, которое также называется «нет» и представлено восклицательным знаком.

Эта операция принимает одно логическое значение, один аргумент, и возвращает другое логическое значение.

Конъюнкция – это еще одна операция, также называемая «и», и она представлена двумя амперсандами.

Эта операция принимает два значения, два аргумента.

И еще одна операция – дизъюнкция, также называемая «или», и она представлена двумя вертикальными полосами.

Эта операция также принимает два аргумента.

Операция отрицания принимает одно логическое значение и возвращает также логическое значение, а именно другое.

Таким образом, отрицание true, это false и наоборот.

Операция «и» принимает два boolean значения в качестве аргумента и возвращает boolean значение.

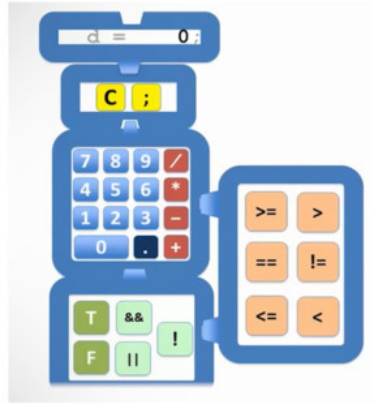
И результат true, если оба аргумента true, и false в противном случае.

Операция или также принимает два аргумента, два булевых значения и возвращает булево значение.

Теперь результат true, если какой-либо аргумент true, и false, если оба аргумента являются false.

Мы могли бы добавить все эти операции в наш калькулятор, который бы исполнял их также успешно, как и операции с числами.

Таким образом, суммируя, в Java мы имеем следующие основные операторы.



А также оператор присваивания = равно.

Арифметические операторы

A = 10, B = 20

Оператор	Описание	Пример
+	Складывает значения по обе стороны от оператора	A + B даст 30
-	Вычитает правый операнд из левого операнда	A - B даст -10
*	Умножает значения по обе стороны от оператора	A * B даст 200
/	Оператор деления делит левый операнд на правый операнд	A / B даст 2
%	Делит левый операнд на правый операнд и возвращает остаток	A % B даст 0
++	Инкремент - увеличивает значение операнда на 1	B++ даст 21
--	Декремент - уменьшает значение операнда на 1	B-- даст 19

Операторы сравнения

A = 10, B = 20

Оператор	Описание	Пример
==	Проверяет, равны или нет значения двух операндов, если да, то условие становится истинным	(A == B) — не верны
!=	Проверяет, равны или нет значения двух операндов, если значения не равны, то условие становится истинным	(A != B) — значение истинна
>	Проверяет, является ли значение левого операнда больше, чем значение правого операнда, если да, то условие становится истинным	(A > B) — не верны
<	Проверяет, является ли значение левого операнда меньше, чем значение правого операнда, если да, то условие становится истинным	(A < B) — значение истинна
>=	Проверяет, является ли значение левого операнда больше или равно значению правого операнда, если да, то условие становится истинным	(A >= B) — значение не верны
<=	Проверяет, если значение левого операнда меньше или равно значению правого операнда, если да, то условие становится истинным	(A <= B) — значение истинна

Логические операторы

A = true, B = false

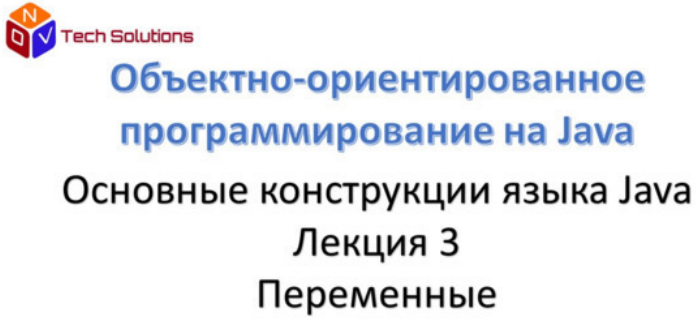
Оператор	Описание	Пример
&&	Называется логический оператор «И». Если оба операнда являются не равны нулю, то условие становится истинным	(A && B) — значение false
||	Называется логический оператор «ИЛИ». Если любой из двух операндов не равен нулю, то условие становится истинным	(A || B) — значение true
!	Называется логический оператор «НЕ». Использование меняет логическое состояние своего операнда. Если условие имеет значение true, то оператор логического «НЕ» будет делать false	!(A && B) — значение true

Переменные

Теперь, когда мы добавляли переменные в калькулятор, мы хотели называть их своими именами.

В Java, когда мы хотим использовать переменную, мы должны сначала представить ее с помощью объявления.

Объявление должно включать сначала тип данных переменной.



В Java существует несколько типов данных, предусмотренных для чисел.

На данный момент для упрощения, представьте себе, что

у нас есть один тип данных, называемый «int».

«int» включает в себя как положительные, так и отрицательные целые числа, в некоторых пределах.

```
boolean b;
int number;
int veryLongName;
```

Таким образом, объявление переменной состоит из имени типа, затем имени переменной и точки с запятой.

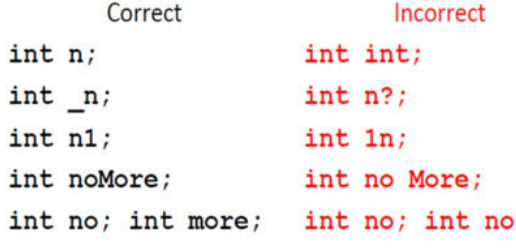
Имя переменной можно выбирать с некоторыми ограничениями.

В некоторых случаях мы также называем имя переменной идентификатором переменной.

Теперь, как мы можем создавать имена для переменных?

По сути, имена – это слова, которые должны следовать некоторым правилам.

И вот некоторые правила.



Имена должны начинаться с буквы или символа подчеркивания.

И они могут содержать буквы – маленькие или заглавные буквы, цифры, и символ подчеркивания.

Другие специальные символы не допускаются.

Исключением является знак доллара, который используется в начале для автоматически генерируемых переменных.

Итак, «n» и «_n» являются правильными именами, тогда как «n?» не может использоваться.

И вы не можете использовать цифру в начале имени.

«n1» является правильным именем, а «1n» – нет.

Кроме того, есть некоторые слова, которые запрещены.

Такие как зарезервированные ключевые слова, например,

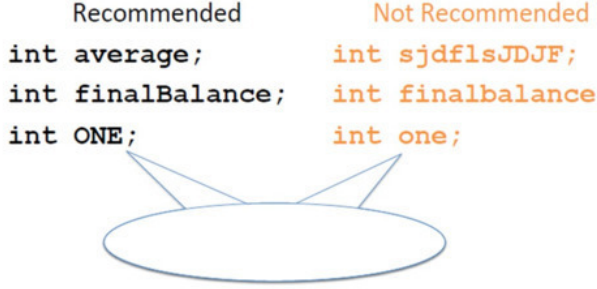
«int» или «boolean», или литералы, такие как «true» и «false».

Таким образом, вы не можете иметь «int» или «true» как имя переменной.

Кроме того, в имени не должно быть пробелов.

И, наконец, будет ошибкой объявление одного и того же имени в одной и той же области видимости.

Теперь есть рекомендации по выбору имен переменных.



Во-первых, имена должны иметь смысл.

Это поможет вам и другим людям понять, как использовать переменные.

Теперь, если вы хотите объединить несколько слов в одно имя, хорошей практикой является начинать каждое следующее слово с большой буквы.

И, наконец, если у нас будет переменная, значение которой не должно изменяться в программе, хорошей практикой будет написать его заглавными буквами.

И мы поставим также что-то перед «int», чтобы сигнализировать о постоянстве переменной.

После того, как мы объявили переменные, мы готовы использовать их и назначить им значения.

```
boolean b;
int n;
b=true;
n=1;
boolean b=true;
int n=1;
int n=true;
```

Также мы можем объявить и присвоить значения одновременно.

Строки и печать

Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 4

Строки и печать

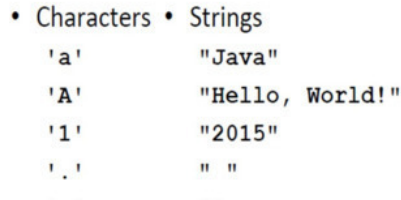
Мы заинтересованы не только в работе с числами.

Нам также нужно работать с текстом.

Поэтому мы будем расширять теперь наш калькулятор значениями и операциями для текста.

Текст состоит из последовательности символов.

Один символ – это символ, который вы можете найти на клавиатуре.



Строка представляет собой последовательность символов.

Строка может состоять из нескольких символов, но она может также иметь только один символ, как в этом примере строки с пробелом.

Строка также может не содержать никаких символов.

В этом случае мы говорим о пустой строке.

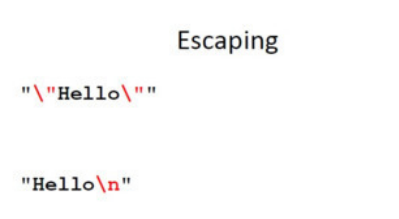
Обратите внимание, что мы помещаем одиночные символы в одинарные кавычки и строки в двойные кавычки.

Это позволяет нам чётко различать литералы строк и символов. Если бы и строки, и символы можно было задавать с помощью одного и того же типа кавычек, то пришлось бы при операциях проверять, символ ли это, или строка.

Теперь, что, если мы хотим иметь двойную кавычку

в строке?

Метод, который мы используем, заключается в том, чтобы поставить escape-символ, обратную косую черту.



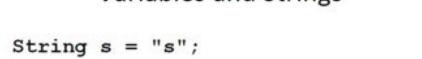
Здесь внешние двойные кавычки не являются частью строки.

Они просто указывают, что у нас есть строка.

Но теперь, если обратная косая черта является символом со специальным свойством, что делать, если мы хотим иметь обратную косую черту в строке?

Тогда мы тоже ставим перед ней обратную косую черту.

Теперь это объявление переменной для строки с именем s, которой мы присваиваем строку, состоящую из просто символа s.



Так что не путайте имя переменной со строкой.

Вот почему мы используем двойные кавычки.

Теперь, какие есть основные операции для строк?

Очень важной операцией является конкатенация или соединение строк.

Обратите внимание, что символ для операции конкатенации – тот же самый, что и для сложения.

Concatenation

```
String s = "s";
String t = "top";
String p = s + t;       "stop"
String q = t + s;       "tops"
```

Это знак плюса.

Вы должны быть осторожны, чтобы не путать число один со строкой «1» в кавычках.

В этом примере n является целым числом и s строкой.

```
int n = 1;
String s = "1";

int m = n + n;          2
String p = s + s;       "11"
String q = s + n;       "11"
```

Поэтому, если говорить n плюс n, мы складываем числа и в результате получим целое число 2.

Если, смотреть на s плюс s, мы объединяем две строки и получаем строку 11.

Интересно отметить, что разрешено писать s плюс n – строка плюс число.

Если один из операндов является строкой, другой также преобразуется в строку.

Поэтому в последнем примере целое число 1 преобразуется в строку «1»

И в результате получим строку 11.

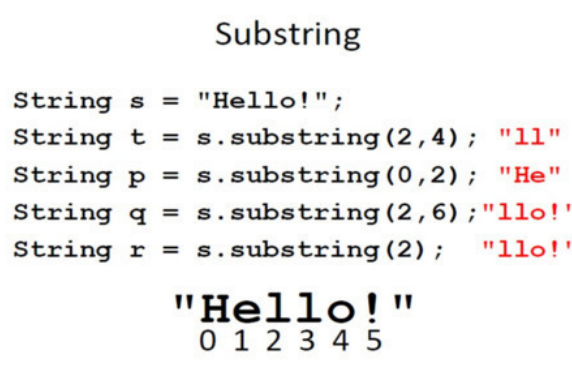
length – это операция, которая применяется к строке и возвращает число, соответствующее количеству символов в строке.

Length

```
String s = "s";
String t = "top";
String p = "";
String q = " ";
int n = s.length();      1
int m = t.length();      3
int j = p.length();      0
int k = q.length();      1
```

Интересно отметить, что длина конкатенации двух строк – это сумма их длин.

С операцией substring мы можем извлечь часть данной строки.



Предположим, что у нас есть строка с этими 6 символами, Hello восклицательный знак.

Первый символ, H находится в нулевой позиции.

Второй E в позиции 1 и так далее, до позиции 5.

Таким образом, substring (2,4) означает, что мы извлекаем подстроку, которая начинается в позиции 2, L, и заканчивается в позиции до 4.

Таким образом, позиция 4 не включена.

Мы включаем символы в позициях 2 и 3, два L.

substring (0,2) выбирает два первых символа, а substring (2,6) остальные.

Также возможно написание одного аргумента в substring.

Это означает, что подстрока выбрана до конца строки.

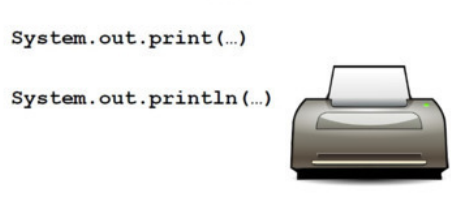
Теперь есть много других операций для строк, таких как

indexof, compareto и т. д.

Которые мы увидим позже.

Если вы хотите напечатать строку в Java, вы можете использовать оператор System.out.print.

И этот оператор принимает аргумент, который нужно напечатать.



Это может быть строка или другой тип.

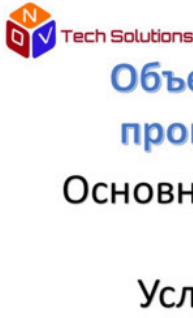
System.out.println, в отличие от System.out.print, переводит печать на новую строку после печати.

Теперь надо отметить, что фактически, String не является примитивным типом данных как boolean или «int».

Вот почему вы пишете String с заглавной буквы S.

Но мы поговорим об этом в позже.

Условия if и else



Теперь поговорим об условии if.

Мы принимаем все время решения.

Если мы думаем, что пойдет дождь, мы берем зонт, прежде чем уйти из дома.

Но если мы думаем, что погода прояснится, мы оставим зонтик дома.

Мы видели, как мы можем составлять выражения последовательно, чтобы сделать программу.

Выражения выполнялись по порядку одно за другим.

Представьте себе, что мы хотим выполнить одну после-

довательность выражений, если выполнено какое-либо условие, и некоторую другую последовательность, если это условие не выполнено.

Давайте посмотрим пример.

Предположим, что мы хотим вычислить квадратный корень из числа.

```
…
…
if (n<0) n=-n;
…
…
```

И мы знаем, что число должно быть положительным, чтобы квадратный корень был реальным числом.

Поэтому, если нам дано отрицательное число, нам нужно сделать его положительным.

Если число положительное, нам не нужно ничего делать.

Как мы сделаем это на Java?

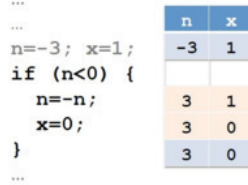
Ключевое слово if вводит условное выражение.

В этом примере выражение присваивания n равно минус n, выполняется только в том случае, если выполняется условие n меньше 0.

Если это условие ложно, ничего не делается.

Теперь, что, если мы хотим выполнить более одного выражения в зависимости от условия.

Мы просто помещаем выражения между фигурными скобками, делая их блоком.



Если условие ложно, ни одно из выражений этого блока не выполняется.

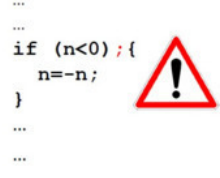
В общем, рекомендуется писать фигурные скобки, даже если при этом условии должно быть только одно выражение.

Логическое выражение для условия должно всегда нахо-

диться между круглыми скобками.

И следите, чтобы не поставить точку с запятой после логического выражения.

Выражение при этом условии – это пустое выражение, которое представлено точкой с запятой, и следующее выражение в фигурных скобках всегда будет выполняться независимо от значения логического выражения.



Таким образом, условное выражение позволяет нам выполнить выражение или блок выражений, в зависимости от значения логического выражения.

Это одна из структур, контролирующих поток выполнения программы.

Иногда мы сталкиваемся с альтернативой на своем пути.

В зависимости от некоторых условий мы идем так или иначе.

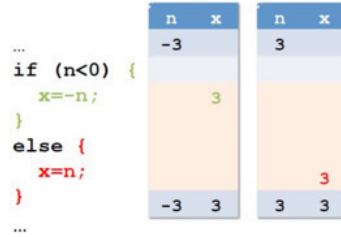
Как мы это выразим в Java?

Сейчас мы знаем, как выполнить выражение в зависимости от одного условия.

Если условие не выполняется, ничего не делается.

Теперь мы хотим выполнить альтернативное выражение в этом случае.

Здесь мы видим простой пример.



x присваивается минус n, если n отрицательно.

Если это не так, x присваивается n.

Таким образом, существует два альтернативных блока выражений.

Тот, который выполняется, если условие истинно.

И тот, который выполняется, если условие ложно.

Этот блок записывается после ключевого слова else.

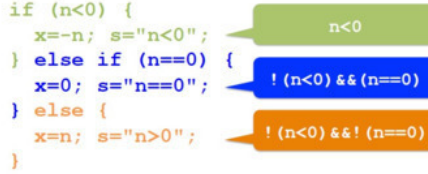
Конечно, в каждой из двух альтернатив, у нас может быть блок выражений вместо одного выражения.

Что теперь, если мы хотим разделить не только два случая, но и больше, например, три случая.

Поскольку условное утверждение является выражением, мы можем поместить его в любую из ветвей.

Например, давайте напишем условное выражение внутри другой ветви.

Новое условие проверяет, равно ли n 0.

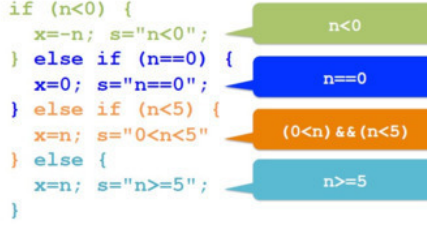


Если это так, мы что-то делаем.

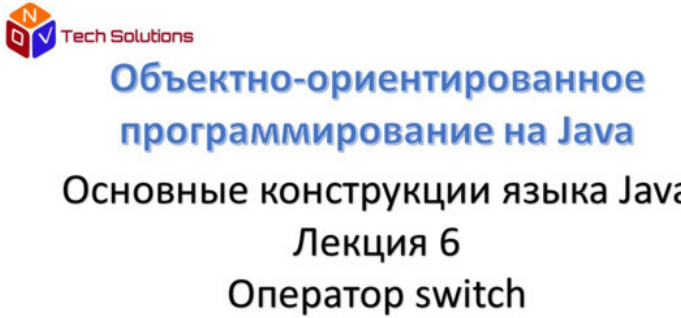
Иначе мы делаем что-то еще.

В целом, теперь у нас есть три случая, из которых только один выполняется.

Здесь показан пример с 4 случаями.

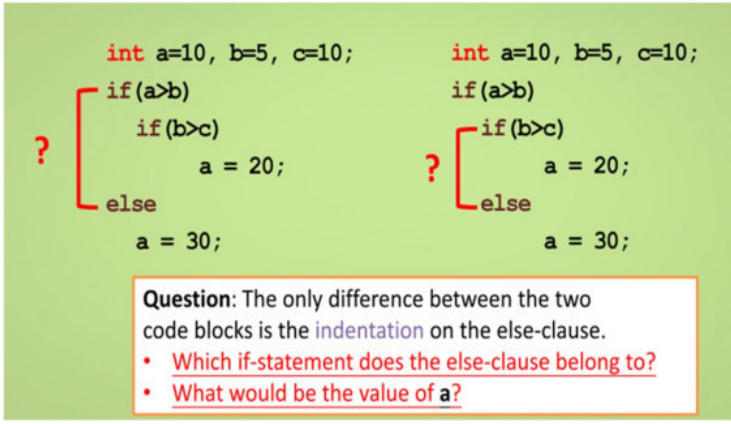


Выражение switch



Для исследования проблемы else, давайте взглянем на эти два блока кода.

Единственное различие между двумя блоками является идентификация принадлежности else.



И здесь могут быть два вопроса.

Первый, к какому выражению if выражение else принадлежит?

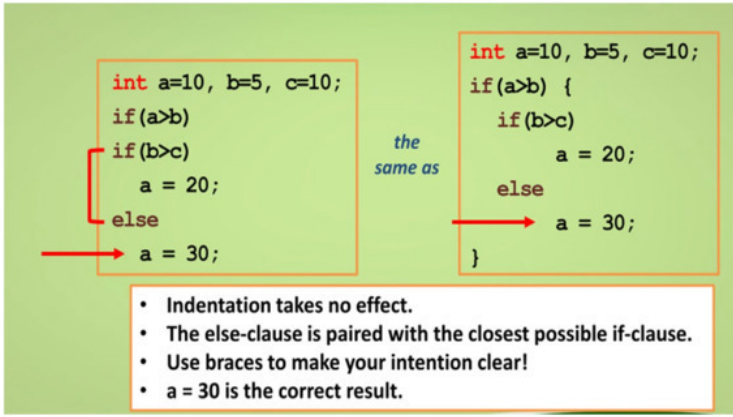
Второй вопрос, это то, каким будет значение после оценки if выражения?

Идентификация фактически не влияет на то, как компилятор будет интерпретировать блоки кодов.

В Java, else выражение соотносится с ближайшим возможным if выражением.

В этом случае, это проверка значения b.

Таким образом, здесь блок кода слева такой же, как код блока справа, с парой вставленных фигурных скобок.



Результат оценки блока кода приведет к установке значения a = 30 в конце выполнения.

Мы можем также использовать комбинацию if-else if.

Пример здесь показывает, как эта комбинация может быть использована для определения уровня знаний в зависимости от оценки.

```
if (score >=90)
grade = 'A';
else if (score >=80)
grade = 'B';
else if (score >=70)
grade = 'C';
else if (score >=50)
grade = 'D';
else
grade = 'F';
```

Обратите внимание, что это будет иметь большое значение, если ключевое слово else остается перед if.

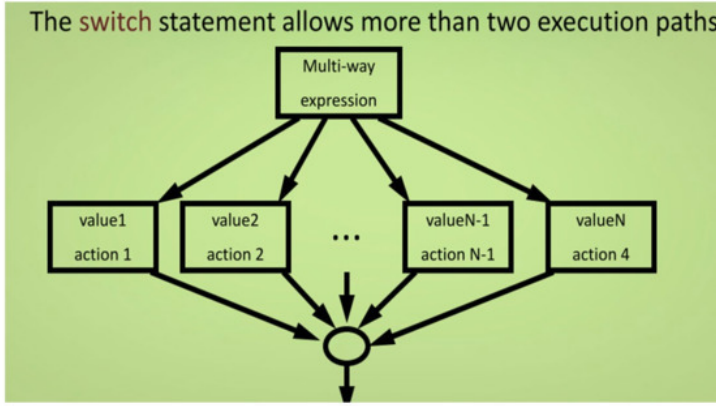
Сравните со случаем, когда else убрано.

```
if (score >=90)
grade = 'A';
if (score >=80)
grade = 'B';
if (score >=70)
grade = 'C';
if (score >=50)
grade = 'D';
else
grade = 'F';
```

В этом случае поток выполнения прерываться не будет.

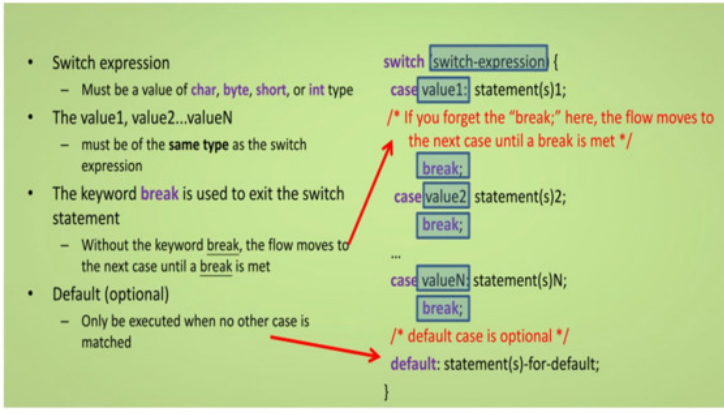
В то время как if выражение позволяет выбрать из двух возможных путей, switch выражение позволяет более двух путей выполнения.

Вот диаграмма для иллюстрации потока управления switch выражения.



И вот синтаксис switch выражения.

Синтаксис switch выражения начинается с ключевого слова switch.



Выражение switch может иметь тип char, byte, short или int, и String.

Значения case value1, value2 и т.д., должны быть того же типа, что и выражение switch.

Ключевое слово break используется для выполнения switch выражения.

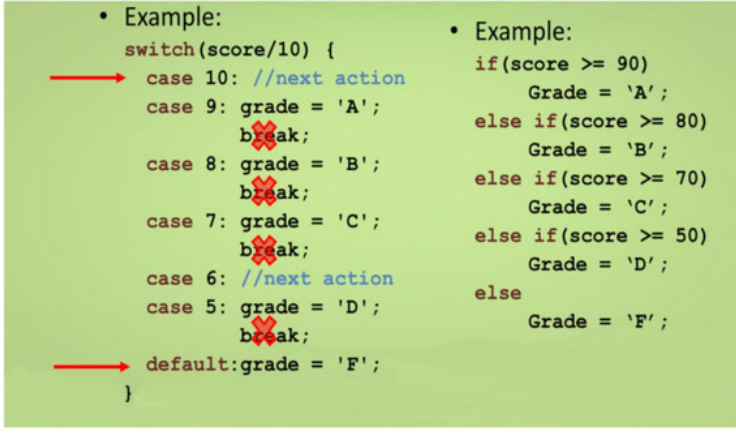
Важно помнить, что без break, поток будет продолжать двигаться к следующему case, пока break не будет найден.

Наконец, есть опция по умолчанию.

С ключевым словом default, эта часть кода будет выполняться только, когда никакие другие случаи не соответствуют.

Теперь посмотрим пример с использованием switch выражения.

Угадайте, что произойдет, если убрать все ключевые слова break?



Это будет то же самое, как если в примере if-else if убрать ключевое слово else.

На самом деле, все, что может быть сделано с помощью switch выражения, также может быть сделано с помощью if-else выражения.

Таким образом, в отличие от операторов if и else оператор switch может иметь несколько возможных путей выполнения.

И switch работает с примитивными типами данных char, byte, short или int и строками.

Решение о том, следует ли использовать операторы if

и else или оператор switch, зависит от выражения, которое тестирует оператор.

Операторы if и else могут тестировать выражения на основе диапазонов значений или условий, тогда как оператор switch проверяет выражения, основанные только на одном перечисляемом значении.

Тернарный оператор

Объектно-ориентированное программирование на Java

Основные конструкции языка Java

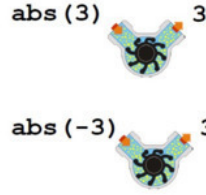
Лекция 7

Тернарный оператор

Представьте, что мы хотим вычислить абсолютное значение числа.

Это число без знака.

Предположим, что abs, является функцией, которая вычисляет абсолютное значение.



Таким образом, abs 3 равна 3, а abs -3 также равно 3.

Давайте определим проблему более формально.

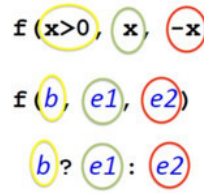
Если условие x больше 0 вычисляется как true, тогда вычисление abs x совпадает с вычислением x.

Если условие x больше 0 вычисляется как false, тогда вычисление abs x – это то же самое, что и вычисление значения минус x.

Теперь мы хотели бы написать выражение, которое вычисляет абсолютное значение.

Мы бы решили проблему, если бы у нас была функция f с тремя аргументами.

Первый аргумент – это условие.



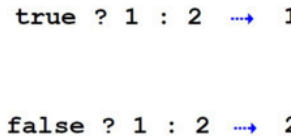
Второй аргумент – это выражение для вычисления в случае true.

И третий аргумент – это выражение для вычисления в случае false.

В Java эта функция существует, называется она тернарный оператор, и имеет определенный синтаксис.

Здесь используется знак вопроса между условием и выражением для случая true и двоеточие между выражением для случая true и выражением для случая false.

В этом примере, если условие истинно, оператор выдает 1.



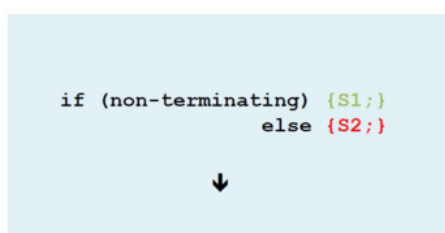
Если условие ложно, оператор выдает 2.

Основным типом данных в условных выражениях является тип boolean, который имеет два значения: true и false.

Но существуют ли в наших условных выражениях if else только два возможных случая?

Представьте, что вы плохо запрограммировали логическое выражение, тогда это приведет к вычислению, которое не может завершиться.

В этом случае, если вычисление логического выражения не завершается, вся программа не будет завершена.



Поэтому, на самом деле, у нас есть три случая, это true, false и undefined.

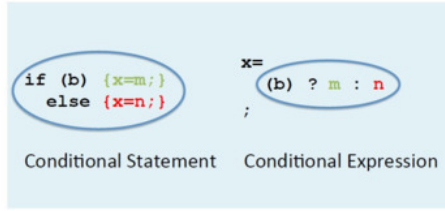
В дальнейшем, анализируя сегменты кода, мы также должны учитывать это неопределенное значение.

Для логических выражений это означает, что у нас есть три возможных случая – true, false и undefined.

И это отличается от традиционной математики, где мы обычно имеем только истину и ложь.

Теперь, давайте немного вспомним о возможностях, которые мы видели.

Здесь, слева, у нас есть условное утверждение, где, в зависимости от значения булевой переменной b, мы присваиваем m или n переменной x.



С другой стороны, у нас есть тройной оператор, который позволяет писать логические выражения.

Оба сегмента кода эквивалентны.

Теперь рассмотрим этот пример.

Представьте, что у нас есть булево значение b и что выражение сравнивает b с true.

```
if (b==true) {S1;} else {S2;}

if (b) {S1;} else {S2;}
```

Это может быть явно упрощено до b, так как если b истинно, b == true, вычисляется как true.

И если b является ложным, b == true, вычисляется как false.

И если b не определено, выражение b == true также не определено.

Так почему бы не написать более простую версию, просто b как условие?

Аналогично вы можете поступить, если мы имеем выражение b == false.

```
if (b==false) {S1;} else {S2;}

if (!b) {S1;} else {S2;}

if (b) {S2;} else {S1;}
```

Вы можете выбрать более простую версию, не b.

И еще вы можете написать b как условие, и поменять операторы S1 и S2.

Здесь у нас есть другое выражение.

```
b ? true : false
```

Давайте проанализируем его.

Здесь, если b не определено, результат не определен.

Если b истинно, результат будет истинным.

И если b является ложным, результат будет ложным.

Мы рассмотрели все возможные значения b и всего выражения

И мы видим, что они имеют одинаковые значения, что они эквивалентны.

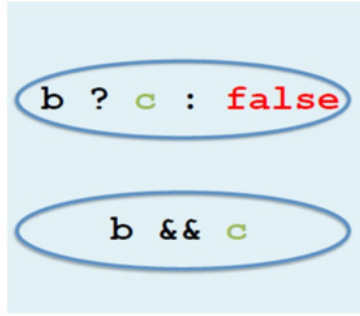
Поэтому вместо всего этого выражения мы можем написать только b.

Та же самая ситуация будет с выражением не b.

Теперь, давайте посмотрим выражение b? c: false.

```
b ? false : true

!b
```

Если b не определено, все выражение не определено.



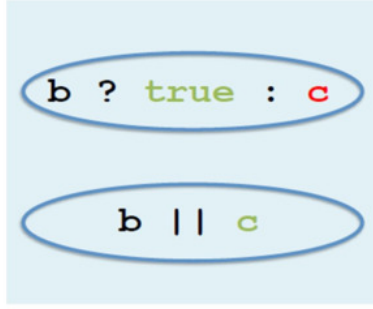
Если b истинно, результат равен c.

Однако, если b является ложным, результат будет ложным.

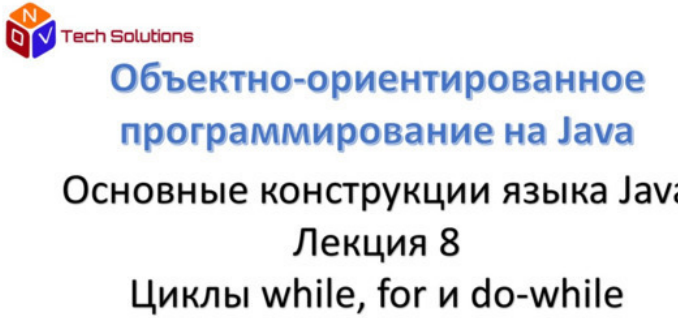
Результат будет истина, только если b и c истина, во всех других случаях результат будет ложным.

Это эквивалентно логическому оператору и.

И наоборот, выражение b? true: c эквивалентно логическому оператору или.

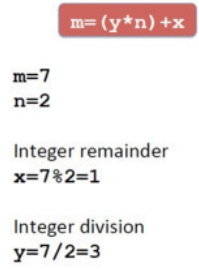


Циклы while и for



Давайте представим, что мы хотим разделить целое число m на другое целое число n.

И мы хотим получить результат целочисленного деления, то есть самое большое количество раз, которое n вписывается в m.



Например, целочисленное деление 7 на 2, равно 3, потому что 2 по 3 раза, это 6.

Остаток равен 1.

И представьте себе, что у нас нет встроенной операции, которая выполняет эту операцию для нас.

Поэтому нам нужно сделать повторяемые вычитания.

И если нам удастся вычесть 2 из 7 три раза, это означает, что целочисленное деление равно 3.

Целочисленное деление y и целочисленный остаток x соответствуют формуле, m равно y умножить на n плюс x.

Предположим, что нам даны целые числа m и n.

А в x сохраняется оставшееся значение после вычитаний.

```
int x=m; int y=0;
if (x>=n){x=x-n; y=y+1;}
if (x>=n){x=x-n; y=y+1;}
if (x>=n){x=x-n; y=y+1;}
if (x>=n){x=x-n; y=y+1;}
if (x>=n){x=x-n; y=y+1;}
```

x	y
7	0
5	1
3	2
1	3
1	3
1	3

Итак, давайте начнем с x равно m.

y содержит результат целочисленного деления.

Мы инициализируем y 0 и приращиваем y на 1 каждый раз, когда мы вычитаем n из x.

И мы продолжаем вычитать n из x, пока x не меньше n.

Если x больше или равно n, мы вычитаем n из x и увеличим y на 1.

Таким образом, эта программа делает то, что мы хотим, но тут есть проблема.

Мы не знаем, сколько операторов if мы должны добавить.

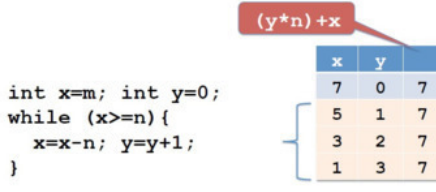
Потому что это зависит от фактических значений m и n.

Например, с 7 и 2, это будет три выражения if.

При других входных данных это должно быть другое число if выражений.

В Java эту проблему решает оператор while.

Теперь эта программа делает то же самое, что и прежде, повторяет выражение, пока выполняется условие.



Но теперь у нас есть одно большое преимущество.

Выражения повторяются столько раз, сколько это необходимо, автоматически.

Но теперь вы должны быть очень осторожны при написании условия while.

Потому что есть опасность войти в бесконечный цикл, если условие while никогда не прекратится.

Преимущество цикла while заключается в том, что нам не нужно заранее знать, сколько раз мы должны что-либо повторять.

```
while (Boolean Exp.){
   Statements;
}
```

Мы повторяем, пока не будет достигнута цель, выраженная логическим условием.

Иногда, однако, мы знаем, сколько раз нам нужно что-либо повторить.

Это легко реализовать подсчетом.

Хитрость заключается в том, чтобы ввести счетчик.

Это целочисленная переменная, которую мы обновляем на каждой итерации.

```
int i=0;
while (i<4){
  i=i+1;
}
```

i	i<4
0	
0	true
1	true
2	true
3	true
4	false

Здесь существует три важных элемента: величина, с которой мы хотим начать, значение в конце и шаг между значениями.

Здесь мы начинаем с 0 и заканчиваем 3. И шаг 1.

Поэтому мы выполняем четыре итерации для i равного 0, 1, 2 и 3.

Теперь, помимо подсчета, мы можем захотеть что-то сделать в теле цикла.

В этом случае предположим, что у нас есть другая переменная, n, которую мы хотим умножать на 2 при каждой итерации.

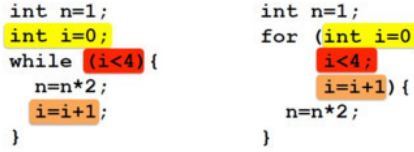
```
int n=1;
int i=0;
while (i<4){
  n=n*2;
  i=i+1;
}
```

i	i<4	n
0		1
0	true	2
1	true	4
2	true	8
3	true	16
4	false	

Так как такого рода подсчет используется часто, в Java для этого есть специальная конструкция.

А именно, цикл for.

Этот цикл объединяет три важных момента для переменной счетчика:



Ее инициализацию, условие остановки, и выражение обновления.

Имейте в виду, что обновление выполняется после того, как выполняется тело цикла, а не раньше.

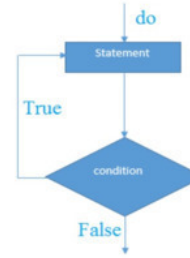
Для цикла for скобки являются обязательными, а также две точки с запятой внутри.

Фигурные скобки необходимы только в том случае, если есть более одного выражения в теле цикла.

Но это хорошая практика, чтобы всегда писать их.

Как вы видели, если в начальный момент условное выражение, управляющее циклом while, ложно, тело цикла вообще не будет выполняться.

```
do {
// тело цикла
} while (условие);

int n = 5;
do {
System.out.println("do-while " + n);
  }  while (--n>0);
```



Однако иногда желательно выполнить тело цикла хотя бы один раз, даже если в начальный момент условное выражение ложно.

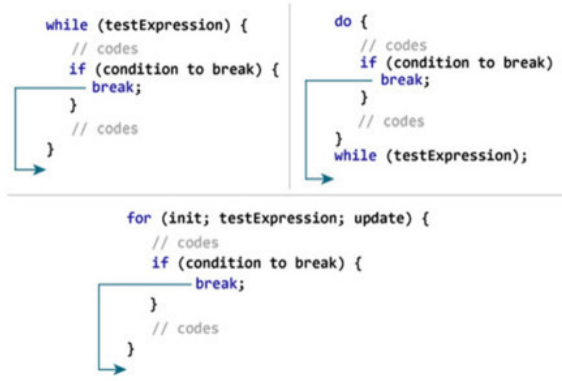
Иначе говоря, существуют ситуации, когда проверку условия прерывания цикла желательно выполнять в конце цикла, а не в его начале.

И в Java есть именно такой цикл: do-while.

Этот цикл всегда выполняет тело цикла хотя бы один раз, так как его условное выражение проверяется в конце цикла.

В приведенном примере тело цикла выполняется до первой проверки условия завершения.

Мы уже видели оператор break в выражении switch.



Но оператор break также может прерывать любой цикл.

Предположим, у вас есть цикл.

И иногда желательно немедленно завершить цикл, не проверяя условие.

В таких случаях используется оператор break.

Оператор break немедленно завершает цикл, и управление программой переходит к следующему выражению, следующему за циклом.

Оператор break почти всегда используется вместе с выражением if else.

Также иногда желательно не прервать цикл, а пропустить код тела цикла и перейти к следующей итерации.

Оператор continue пропускает текущую итерацию цикла и когда выполняется оператор continue, управление программой переходит к концу цикла.

Затем проверяется условие, которое управляет циклом.

Оператор continue также почти всегда используется вместе с выражением if else.

Массивы

Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 9

Массивы

На почте мы могли арендовать ячейку, чтобы получать письма.

Есть также много других мест, где мы можем арендовать ячейку: в банке, для хранения ценностей, на вокзале, чтобы оставить багаж.

Ячейки обычно обозначаются последовательными номерами.

Ячейки или шкафчики могут быть разного размера.

В программировании мы видели переменные, которые позволяют хранить значения.

Здесь размеры также могут отличаться в зависимости от того, хотим ли мы сохранить логическое значение или число с плавающей запятой.

Однако в некоторых случаях нам может понадобиться упорядоченный набор значений одного и того же типа.

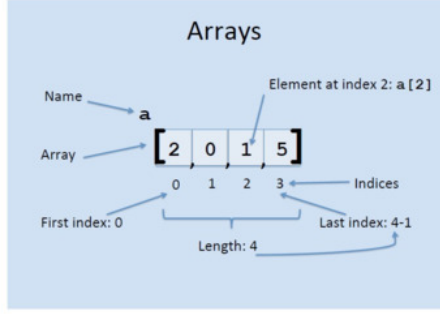
Например, когда мы хотим хранить оценки учеников

класса, или температуры каждого дня месяца.

Точно так же, как мы могли бы арендовать ряд ячеек, нам может потребоваться зарезервировать набор или массив переменных одного и того же типа.

Как нам к этим переменным обращаться?

Мы привыкли свободно выбирать имена переменных.



И таким же образом мы можем дать имя массиву переменных.

Для обозначения местоположения одной переменной используется индекс.

Так, например, мы могли бы назвать массив a.

Предположим, что у него четыре элемента в четырех позициях.

Мы будем ссылаться на каждую позицию, добавляя индекс в квадратные скобки.

Обратите внимание, что мы начинаем с индекса 0 и увеличиваем его на единицу.

Здесь мы видим примеры массивов.

Массивы могут содержать разные типы значений, но в каждом массиве, тип является одинаковым для всех значений.

Массив может иметь любую длину, но после определения длины при создании массива его длина остается фиксированной.

Надо помнить, что есть два шага при работе с массивами. Объявление массива и его создание.

Creation

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

Declaration and Creation

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

Элементы в массиве можно получить с помощью индекса.

Мы не должны путать значение элемента с его индексом.

Еще одна вещь, которую следует помнить, это то, что первым элементом массива является элемент с индексом 0.

Таким образом, индексы начинаются с 0 и до длины массива минус 1.

Мы объявляем массив, указывая тип элементов, затем открываем и закрываем квадратные скобки, и затем указываем имя, которое мы выбрали для нашего массива.

После объявления, создавая массив с помощью ключевого слова new, мы физически резервируем для него место в памяти, как в почтовом отделении.

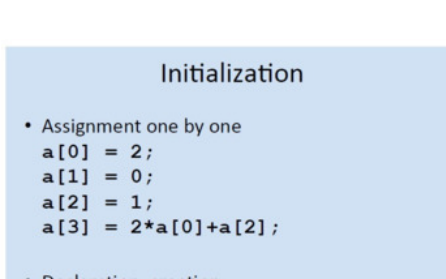
Мы также можем сделать это вместе: объявить и создать

массив в одной строке.

Теперь мы можем хранить значения в разных позициях.

Как мы сохраняем значения?

Мы используем оператор присваивания, как раньше мы использовали его для переменных.



Имя массива с индексом используется, как мы раньше использовали идентификаторы переменных.

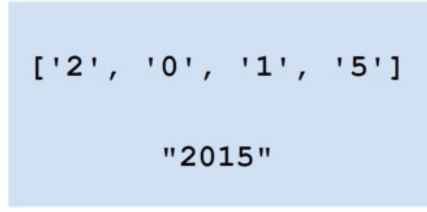
Мы также можем объявить, создать и инициализировать массив сразу, как мы видим здесь, в последней строке, используя фигурные скобки.

Обратите внимание, что в этом случае нам не нужно писать ключевое слово «new».

Теперь, если строки – это упорядоченные последователь-

ности символов, вопрос, является ли строка и массив символов одним и тем же.

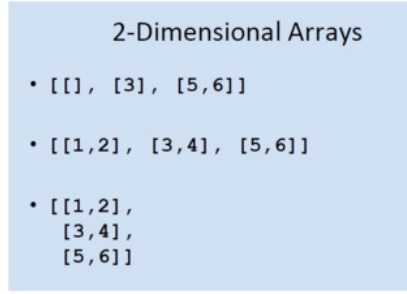
Это не так, хотя можно конвертировать одно в другое.



Другой вопрос заключается в том, может ли элемент массива быть массивом.

Здесь ответ да.

Таким образом, мы получаем то, что мы называем двумерными массивами.



Но возможны и многомерные массивы.

Таким образом, массивы – это упорядоченные последовательности элементов одно и того же типа.

И длина фиксируется при создании массива.

И элементы массива могут быть массивами.

Массивы и циклы for имеют нечто общее.

Массив состоит из последовательности данных, а цикл for выполняет выражения последовательно несколько раз подряд.

Здесь мы видим массив с четырьмя целыми числами от 0 до 3.

```
[0, 1, 2, 3]



for (int i=0;
     i<4;
     i=i+1)
{…}
```

И ниже приведена структура цикла for, которая повторяет выражения четыре раза.

Теперь, если мы хотим сделать одно и то же преобразование для всех значений в массиве, цикл for является хорошим для этого способом.

Например, если применить операцию возведения в степень 2 к целому числу 3, получим 9.

```
y = square(x);


for (int i=0; i<4; i=i+1){
  y[i] = square(x[i]);
}
```

Теперь представьте, что мы хотим применить эту операцию ко всем целым числам в массиве.

Цикл for поможет нам последовательно брать все значения в массиве и возводить их в степень 2, начиная с индекса 0 до индекса 3.

Другой пример – сложить все числа в массиве.

```
[0, 1, 2, 3] ➔ 6


int z = 0;
for (int i=0;
     i<4;
     i=i+1){
  z += x[i];
}
```

Если вы хотите сделать это для любой длины массива, используйте x. length вместо 4.

```
[0, 1, 2, 3] ➔ 6


int z = 0;
for (int i=0;
     i<x.length;
     i=i+1){
  z += x[i];
}
```

Перебор элементов массива в цикле for, начиная с индекса 0 до длины массива, настолько распространен, что для этого существует специальный цикл for.

```
[0, 1, 2, 3] → 6


int z = 0;
for (int elem: x){
  z += elem;
}
```

В этом цикле for мы можем проинструктировать переменную elem последовательно использовать все элементы массива.

Представление данных и типы данных

Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 10

Представление данных и типы данных

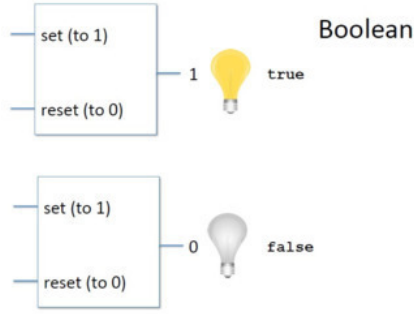
Давайте посмотрим под капот калькулятора или компьютера, и посмотрим, как мы можем представлять данные.

И начнем с простого.

Давайте посмотрим на логические значения, потому что там есть только два значения, true и false.

Цифровые компьютеры состоят из электроники, которая может находиться только в одном из двух состояний.

Триггер – это базовая единица, которая может оставаться либо в одном положении, либо в другом.



Выход триггера остается в одном из этих двух состояний и будет оставаться там до тех пор, пока не появится сигнал для его изменения.

В действительности 1 может иметь нулевое напряжение, а другое состояние – пять вольт.

Но мы можем произвольно интерпретировать их как 0 и 1.

Поэтому мы можем сказать, что триггер может хранить один бит информации.

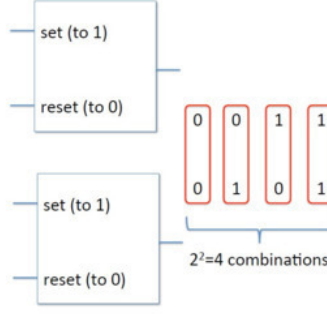
Теперь это именно то, что нам нужно, чтобы сохранить логическое значение, потому что логических значений также два, ложь и истина.

И мы, опять же, можем произвольно присвоить 0 false и 1 true.

Итак, мы говорим, что нам нужен бит, чтобы сохранить логическое значение.

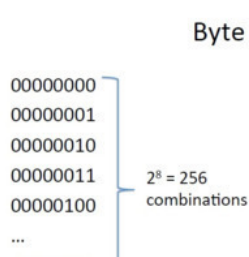
Теперь, если у вас есть два триггера, мы можем сохранить два бита.

Если мы соберем их вместе, у нас будет четыре возможных комбинации: 0—0, 0—1, 1—0 и 1—1, поскольку каждый из них может иметь состояние 0 или 1 независимо друг от друга.



И если мы возьмем восемь триггеров, чтобы сохранить восемь бит, у нас будет 2 в степени 8 различных комбинаций.

То есть 256 комбинаций в целом.



Что мы можем с ними делать?

Восемь бит называется байт.

Итак, что мы можем сделать с байтом?

Мы можем представить 256 различных чисел.

Например, натуральные числа от 0 до 255.

Мы также можем отображать 256 уровней красного, от черного до ярко-красного.

И мы можем получить любой цвет, составляя уровни красного, зеленого и синего.

Для каждого из этих компонентов мы используем один байт.

Таким образом, это всего три байта или 24 бита, что означает 2 в степени 24, что почти 17 миллионов цветовых комбинаций.

Звуки, фильмы, все представлено битами 0 и 1.

Это позволяет нам иметь богатую информацию, но в тоже время иметь единый способ обработки этой информации.

Наконец, мы можем также представлять отдельные символы, как те, которые есть у вас на клавиатуре, а также некоторые другие специальные символы.

Для этого существует множество кодировок.

Java использует кодировку юникода, использующую 16 бит.

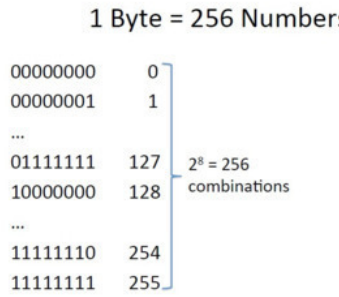
Другие кодировки используют только восемь бит.

Таким образом, все в компьютере представлено битами.

Все сводится к нулям и единицам.

Давайте сосредоточимся на том, как мы представляем числа в двоичной форме битами.

С 1 байтом – 8 бит – мы можем сформировать 256 различных комбинаций, или 2 в степени 8.



Поэтому мы можем представить 256 различных чисел.

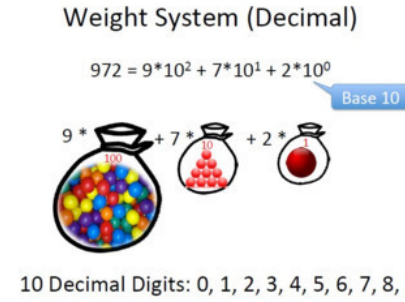
Это могут быть, например, натуральные числа от 0 до 255.

Но какая комбинация байт соответствует какому числу?

Давайте проанализируем систему, которую мы используем для представления чисел в нашей десятичной системе, которая использует 10 цифр, от 0 до 9.

Используем систему, основанную на весах.

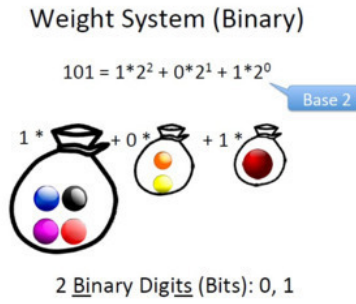
Чем больше мы двигаемся влево, тем выше вес.



Когда мы пишем 972, мы имеем в виду 9 умножить на 100 плюс 7 умножить на 10 плюс 2.

Так как здесь основание 10, система исчисления называется десятичной.

Для двоичной системы исчисления тот же принцип, только основанием будет 2.



Соответственно, перевести число из двоичной системы в десятичную очень просто, нужно сложить получившийся ряд.

10110110 =

Base 2

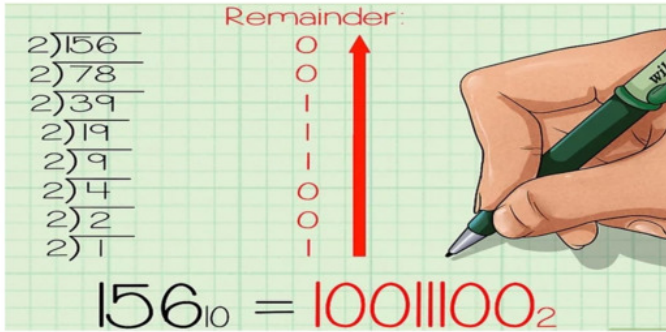
$1*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0 =$

$128 + 32 + 16 + 4 + 2 =$

182

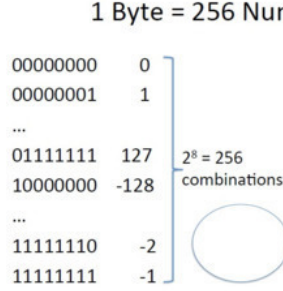
Base 10

Перевести число из десятичной системы в двоичную тоже просто, нужно делить на 2 и записывать остаток.



Но как насчет отрицательных чисел? Нам тоже нужно работать с ними.

Неотрицательные числа, т. е. 0 и положительные числа – закодированы по-прежнему, где самый левый бит установлен в 0.



И у нас осталось семь бит.

Таким образом, мы можем иметь 2 в степени 7 различных неотрицательных чисел, а именно от 0 до 127.

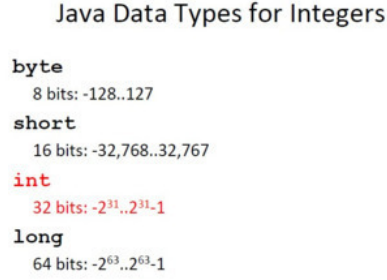
Для отрицательных чисел они кодируются таким образом, что сумма отрицательного числа и его положительного аналога равна 2 в степени числа бит, т. е. восемь, или 256, или 1, а затем восемь 0.

Таким образом, с этим кодированием мы можем представлять, как положительные, так и отрицательные числа.

Теперь давайте сосредоточимся на Java.

Какие типы данных мы используем для целых чисел?

На самом деле это не один тип данных, а доступно несколько типов данных.



У нас есть тип данных, называемый «байт», который использует точно восемь бит – это и есть, один байт.

Мы можем представить цифры от -128 до 127, как мы только что видели.

Есть тип данных «short», который использует 16 бит и находится в диапазоне от -32 000 до плюс

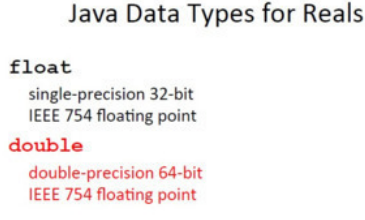
32000.

Но основным типом данных, которым мы будем пользоваться, будет «int».

Здесь максимальное положительное число составляет более 2 миллиардов.

Если вам потребуются большие цифры, можно использовать «long» с 64 битами.

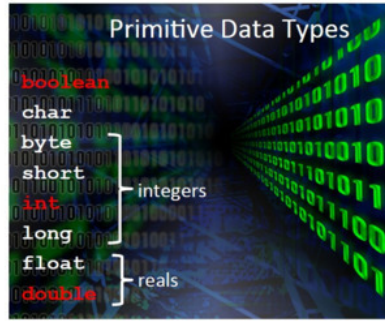
Для чисел с плавающей запятой есть два типа данных в Java: «float», который использует 32 бита, и «double», который использует 64 бита.



Рекомендуется использовать double, когда нужны числа с плавающей запятой.

Подводя итог, существует восемь примитивных типов данных в Java.

Два для представления нечисловых данных: boolean для булевых значений, true и false, и char – для представления одного символа.



И числовые типы данных.

int – это основной тип данных, который нужно запомнить для представления целых чисел.

И остальные байт, short, и long.

И double – это основной тип данных для чисел с плавающей запятой.

Другой тип – float.

Таким образом мы не можем работать с бесконечно большими числами или числами с бесконечной точностью.

Методы

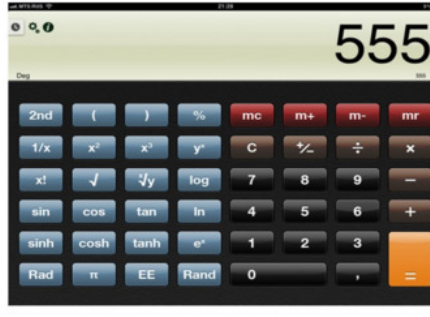
Представьте себе, что вам приходится многократно вычислять квадрат числа.

Для каждого числа, вы вводите одно число, затем оператор умножения, а затем снова число.

И то же самое для другого числа и т. д.

Поэтому в калькуляторе было бы неплохо иметь программируемую кнопку, которая выполняет любую операцию, которую мы определим.

Это может быть квадрат или квадратный корень, или любые вычисления, которые нам понадобятся.



В Java также возможно определять пользовательские операции.

Но вместо того, чтобы называть их операциями, мы называем их методами.

Это является терминологией Java.

В других языках программирования они называются функциями или процедурами.

Метод – это вычисление, которому мы даем имя, так что мы можем вызывать его в любое время, когда нам нужно выполнить это вычисление.

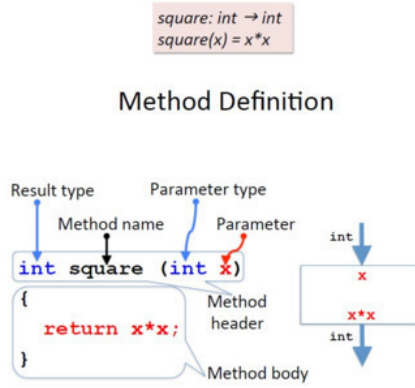
Метод может зависеть от одного или любого числа параметров.

И метод может привести к какому-то результату или ка-

кому-то эффекту.

Рассмотрим метод вычисления квадрата числа.

Можно представить этот метод как черный ящик, который получает целое число в качестве входных данных и выводит другое целое число.



В математических терминах мы можем определить его как функцию следующим образом.

Мы дадим функции имя, например, square.

И эта функция принимает целое число как параметр и возвращает целое число.

Функция определяется следующим образом.

Если мы назовем аргумент или параметр как x, результат получается умножением x на x.

Теперь, как мы определим это в Java?

Сначала мы напишем что-то похожее на первую строку в математическом определении.

Но порядок немного другой.

Во-первых, мы пишем тип результата, затем имя метода, а затем в круглых скобках тип параметра и далее идентификатор параметра.

При этом у нас может быть несколько параметров.

Все это называется заголовком метода.

Затем мы напишем в фигурных скобках то, что мы должны сделать, чтобы вычислить результат.

И мы указываем, что это результат возврата, поместив ключевое слово return перед выражением.

Затем в фигурных скобках мы пишем вычисление, которое хотим выполнить.

И мы называем это телом метода.

Имя метода может быть любым допустимым идентификатором.

Но мы будем следовать соглашению, и напишем его с маленькой буквы.

И обычно это глагол.

Если нам нужно больше одного слова, мы будем писать каждое следующее слово с заглавной буквы.

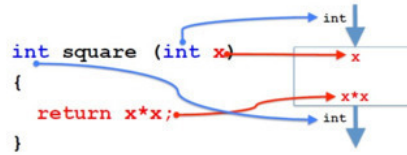
isEmpty

Как мы видим здесь в isEmpty.

И рекомендуется, чтобы имя метода имело значение, чтобы другие могли легко понять, что здесь вычисляется.

Имена параметров мы также можем свободно выбирать.

Нам нужно дать имя параметру, потому что нам нужно обращаться к параметру в теле метода.



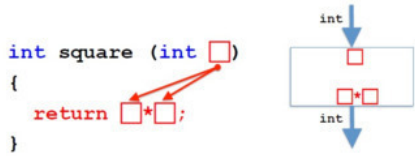
Но этот идентификатор является внутренним.

Если мы заменим его на другой идентификатор, мы не изменим метод.

Вместо x мы можем указать y в качестве идентификатора параметра.

Так как, по существу, x или y являются просто заполнителями для фактического параметра, который мы указываем при вызове метода.

Сколько входных параметров может иметь метод?

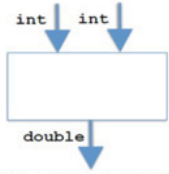


И что насчет результата?

Мы видели, как определить метод с одним параметром и одним результатом.

Можем ли мы также иметь больше параметров?

У нас может быть несколько параметров.




```
double average(int x, int y)
{
  return (x+y)/2;
}
```

Здесь мы видим метод с двумя параметрами.

Обратите внимание, что они разделяются запятыми.

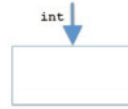
И у нас может быть еще больше параметров, разделенных запятыми.

Также у нас может не быть никаких параметров.

Теперь круглые скобки пустые.

В этом случае этот метод всегда возвращает одно и то же значение.

Или у нас может не быть никакого возвращаемого результата.




```
void printInverse(int x)
{
  System.out.println(1/x);
}
```

В этом случае мы пишем void как тип результата.

Это имеет смысл, например, если мы хотим что-то напечатать.

В других языках программирования говорят о процедурах, если нет возвращаемого значения.

И о функциях, если возвращается результат.

Но в Java мы просто говорим о методах.

Наконец, мы можем иметь метод без параметров и без результатов.

```
void printHello()
{
  System.out.println("Hello!");
}
```

Теперь мы рассмотрели все возможные случаи.

Область видимости переменных

Объектно-ориентированное программирование на Java

Основные конструкции языка Java

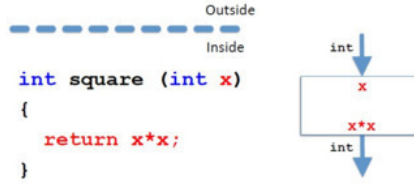
Лекция 12

Область видимости переменных

В предыдущей лекции мы узнали, как определить метод.

И мы хотим знать, что происходит, когда мы его вызываем.

Возьмем снова метод, вычисляющий квадрат числа.



Он называется square и принимает одно значение и возвращает другое значение – квадрат числа.

Важно отметить, что определение метода идентифицирует два контекста – внутри и снаружи.

Внутри мы можем использовать параметры x или y или что угодно.

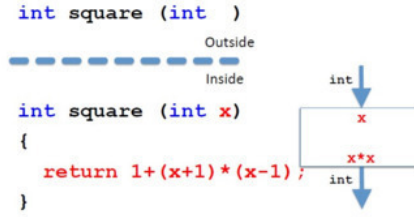
Но не снаружи.

Извне мы просто знаем название метода, параметры, и тип результата.

Как вычисляется метод, это вопрос внутреннего контекста.

В какой-то момент мы могли бы изменить тело метода.

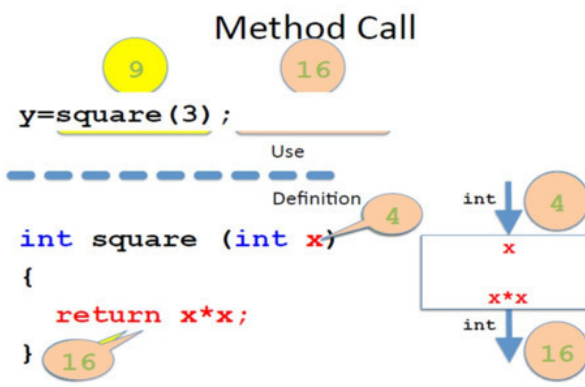
Здесь мы видим альтернативный способ вычисления квадрата числа.



Но мы не знали бы этого извне, из контекста вызова.

Теперь давайте посмотрим, что происходит, когда мы вызываем метод с заданным значением.

Мы могли бы проанализировать, что происходит, когда мы вызываем square (3).



Но давайте сделаем немного интереснее.

Попробуем оценить выражение square (3) + square (4).

Чтобы получить результат суммы, сначала мы должны вычислить первый операнд, square (3).

И для этого мы перейдем к определению метода, где x теперь равно 3.

Это означает, что мы должны заменить все x на 3.

Таким образом, мы вычисляем 3 умножить на 3.

Результат будет – 9, и это то, что возвращает вызов метода.

9 теперь является значением первого операнда суммы.

Затем нам нужно вычислить значение для square (4).

Перейдем к определению метода, но теперь x равно 4.

3 больше не существует.

Поэтому мы заменяем все x на 4, и поэтому умножаем

4 на 4.

Этот вызов метода возвращает 16 вызывающему выражению.

Теперь у нас есть оба операнда, и мы можем сложить 9 и 16.

Во всех этих вычислениях важно отметить, что два вызова одного и того же метода полностью независимы.

Мы использовали x с двумя независимыми значениями.

Сначала 3, а затем 4.

И когда мы использовали 4, 3 уже не существовало.

Каждый раз, когда мы делаем новый вызов, параметры создаются со значениями вызова.

Значения, которые мы имели от предыдущих вызовов, просто забываются.

Мы использовали идентификаторы или имена в разных целях: для переменных, для методов, для параметров метода и т. д.

Теперь возникает вопрос: если у нас есть переменная с именем «x», а затем у нас есть метод с параметром.

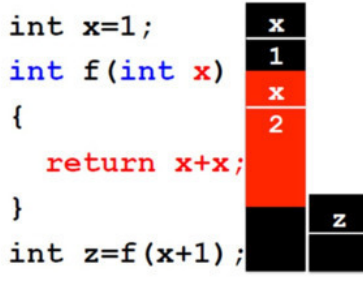
Можно ли назвать этот параметр как «x»?

Или будет какая-то несовместимость?

Можем ли мы использовать одно и то же имя в разных контекстах?

Давайте рассмотрим пример.

Представьте, что у нас есть программа, где есть целочисленная переменная с именем x,



Которую мы инициализируем в значение 1.

И у нас также есть метод «f», который имеет целочисленный параметр.

И мы просто решили назвать его «x».

Вопрос, можем ли мы это сделать?

И если да, то что этот метод вернет в качестве результата?

Ответ на этот вопрос при написании кода на Java – да, мы можем это сделать.

Каким образом, мы управляем двумя x?

Каждый x действителен в определенном контексте, при выполнении определенного сегмента кода.

У нас есть черный x, который действителен, и который существует, и для которого мы сохраняем пространство в па-

мяти, когда объявляем переменную.

Мы также зарезервировали пространство в памяти для z.

И когда мы вызываем f с x плюс 1, значение x равно 1.

1 плюс 1 равно 2, и мы вызываем f с 2.

Далее мы переходим к определению метода.

Вызываем f с 2.

Таким образом, красный x равен 2.

Итак, мы выполняем x плюс x со значением 2.

2 плюс 2 равно 4.

И это то, что этот метод возвращает и что хранится в z.

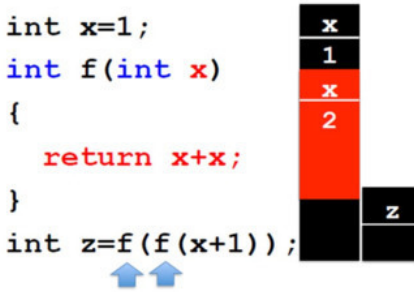
Теперь помните, что параметр x метода f является просто заполнителем.

Поэтому, если f вызывается с переменной x, а значение x равно 2, f с x возвращает 4.

И с этим нет никаких проблем.

Мы говорим, что первое x является глобальной переменной, тогда как параметр x является локальным для метода.

В этом примере мы видим, что эта локальная переменная – этот параметр – создается дважды: во-первых, для внутреннего вызова f с x плюс 1, со значением 2, – и второй раз для внешнего вызова со значением 4.



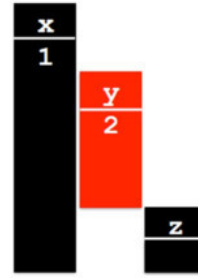
После выхода из каждого определения метода, созданная переменная будет уничтожена.

И выделенное пространство в памяти компьютера будет освобождено.

Поэтому, если вы хотите использовать внешнюю переменную в теле метода, вы должны выбрать другое имя.

Здесь мы видим, как использовать глобальную переменную x и параметр y в теле метода.

```
int x=1;
int f(int y)
{
  return x+y;
}
int z=f(x+1);
```

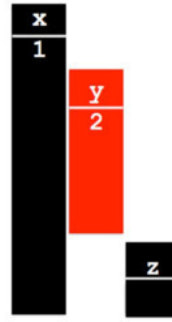


В этом случае у нас есть переменная x, которая видна во всем теле метода и за его пределами, и у нас есть переменная y, которая существует и видна только в теле метода.

Вне этого метода y не существует.

В этом примере, все, что мы только что сказали для параметров метода, применяется к переменным, объявленным внутри тела метода.

```
int x=1;
int f()
{
  int y=2;
  return x+y;
}
int z=f();
```



Здесь переменная y является локальной переменной в методе f.

В этом методе мы используем глобальную переменную x и локальную переменную y.

Этот пример аналогичен предыдущему.

```
int x=1;
int f()
{
  int x=2;
  return x+x;
}
int z=f();
```

Но в этом случае мы решили назвать локальную переменную внутри метода x, так же, как и глобальную переменную.

Таким образом, в этом случае у нас нет доступа к глобальной переменной.

Когда мы вызываем f для вычисления z, мы вызываем f, где внутри определяется x со значением 2.

Таким образом, мы возвращаем 2 плюс 2, равно 4.

Метод f всегда возвращает 4.

И это то, что мы сохраним в переменной z.

Таким образом, мы видели, что у нас есть глобальные и локальные переменные.

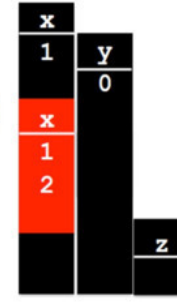
Глобальные переменные существуют, начиная с объявления и для остальной части программы.

Но они могут временно затеняться другими локальными

переменными с тем же именем.

В этом примере показан цикл.

```
int x=1;
int y=0;
for (int x=1; x<3; x++)
{
  y=y+x;
}
int z=y;
```



Для циклов также объявляются локальные переменные.

Здесь переменная x цикла for не позволяет нам видеть глобальную переменную при выполнении цикла.

Здесь у нас есть глобальная переменная x и глобальная переменная y.

Они инициализируются 1 и 0 соответственно.

Затем у нас есть глобальная переменная z, которая сохраняет значение y, но после выполнения этого цикла for.

Этот цикл for выполняется дважды.

Один раз для x равного 1 и один раз для x равного 2.

В каждом цикле for, y накапливает значение x.

Таким образом, при первом запуске y получает значение 1, а во втором y получает значение 1 плюс 2, равно 3.

Когда мы выходим из цикла for, локальная переменная x исчезает, остается только глобальная.

y имеет значение 3, и это значение, которое мы сохраняем в z.

Таким образом, мы видим точно такое же поведение для этих переменных в цикле for, как мы видели с локальными переменными в методах и с параметрами в методах.

В этом примере у нас есть глобальная переменная x.

```
int x=1;
int f(int x){
  int y=0;
  for (int x=1;x<3;x++)
      {y=y+x;}
  return y+x;
}
int z=f(x+2)+x;
```

И у нас есть метод с параметром x.

И внутри этого метода у нас есть цикл for с другой переменной x.

Таким образом, в этом случае у нас есть 3 переменных x.

Поэтому, когда мы вызываем f c x плюс 2, в последней строке, где x равно 1, мы вызываем f c 3, чтобы вычислить z.

В методе, параметр x равен 3.

Внутри метода мы объявляем переменную y, инициализированную 0, и затем мы определяем цикл for.

Этот цикл for выполняется два раза, как в предыдущем примере.

Здесь, мы объявляем другую переменную x, которая делает невидимыми предыдущие две переменные x, пока мы не выполним цикл for.

Здесь мы увеличиваем значение y.

y в конце получает 3 и возвращает y плюс x.

Но что это за x?

Это не та переменная x в цикле for, потому что мы вышли из цикла for.

Эта x равна 3 и это параметр метода.

Поэтому возвращается 3 плюс 3.

Это то, что мы возвращаем z, и что добавляется к x, но в этом случае это глобальная переменная x, поэтому мы получаем 7 и присваиваем 7 в z.

Этот пример легко проанализировать.

```
int x=1;
int f()
{
  return x;
}
int z=f();
```

Метод f определяется в контексте, где x равно 1.

Таким образом, этот метод всегда возвращает 1 независимо откуда он был вызван.

x равно 1 и z также присваивается 1.

Важно отметить, что f получает свое определение в том месте, где он определен.

Если он определен в том месте, где x равно 1, метод f определяется, чтобы вернуть 1.

И это видно в этом примере.

В этом примере у нас есть два метода: f и g.

```
int x=1;
int f()
{
  return x;
}
int g()
{
  int x=0;
  return f();
}
int z=g();
```

g вызывает f, и он вызывает его в контексте, где x равно 0.

И здесь нужно учитывать, что метод f был определен в контексте, где x равно 1.

И мы уже сказали, что метод f всегда возвращает 1 независимо от того, где он вызывается.

Так как здесь x равно 1.

Это называется лексической областью действия или статической областью действия в отличие от динамической области действия.

Большинство языков программирования имеют статическую область действия, в том числе и Java.

Поэтому, как только метод определен, его значение и его поведение, зафиксированы.

Теперь, если мы удалим самое верхнее объявление x, пе-

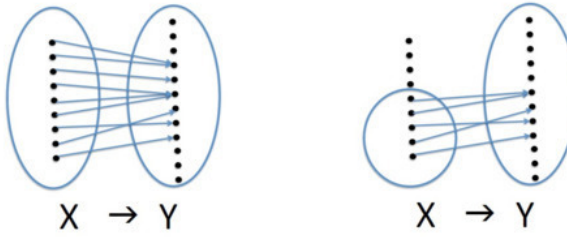
ременная x не определяется при объявлении f.

```
int f()
{
  return x;
}
int g()
{
  int x=0;
  return f();
}
int z=g();
```

Следовательно, этот сегмент кода выдаст ошибку во время компиляции.

Далее мы проанализируем взаимосвязь между частично определенными функциями в математике, и методами в Java, которые не определены для некоторых входных значений.

В математике мы изучаем функции, т. е. отображения между множествами значений, где значения области определения X сопоставляются значениям множества Y.



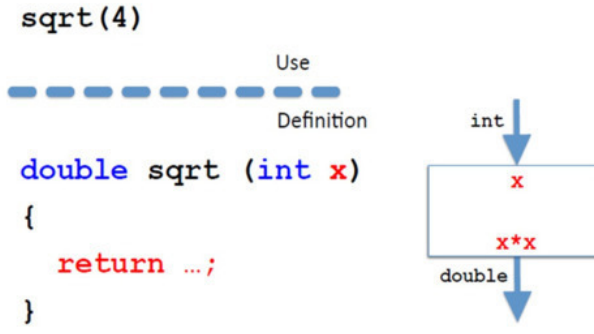
Обычно для всех значений из множества X существуют значения во множестве Y.

Однако может быть случай, когда для некоторых значений X нет отображения, определенного в Y.

В этом случае мы говорим о частично определенной функции.

Если вы хотите избежать частично определенных функций и всегда работать с полными функциями, вы можете выделить в X меньшее множество, где все значения имеют отображение.

Теперь вернемся к Java.



Предположим, мы хотим вычислить квадратный корень из 4.

Здесь есть два результата, плюс 2 и минус 2.

Предположим, что наш метод просто возвращает положительное значение, плюс 2.

Мы всегда можем получить другое решение, добавив знак минус.

Теперь, что произойдет, если мы вызовем метод square с аргументом минус 4?

Мы знаем, что решением в этом случае являются не действительные числа, а мнимые числа.

Таким образом, не существует реального числа, которое может быть предложено в качестве результата метода.

Метод не определен для отрицательных чисел.

В математике мы можем определить функции более подробно.

Мы можем настроить область определения в соответствии с тем, что нам нужно.

Например, мы могли бы сказать, что область определения этой функции не множество целых чисел, а множество натуральных чисел, то есть 0 и положительные целые числа.

Таким образом, функция будет определена для всех значений в этой области определения натуральных чисел.

Но в программировании мы имеем дело с существующими типами.

Теперь, как мы определяем в Java частично определенные функции или частично определенные методы?

Что мы можем сделать в случае метода, который не определен для всех возможных входных значений.

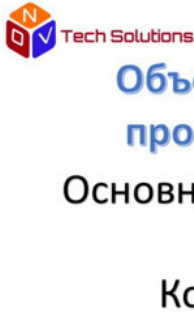
Во-первых, так как возникает ошибка при вызове метода square с отрицательным числом, мы будем ожидать ошибку, и программа должна завершиться с ошибкой.

Это, конечно, не самый удобный способ для решения этой проблемы.

Во-вторых, мы можем проверять значения параметров метода в самом методе или при вызове метода.

Или мы можем перехватить и обработать возникшую ошибку в самом методе или после его вызова, и об этом мы поговорим, когда будем обсуждать исключения Java.

Комментарии. Javadoc



В прошлой лекции мы говорили о том, что мы можем сделать в случае метода, который не определен для всех возможных входных значений.

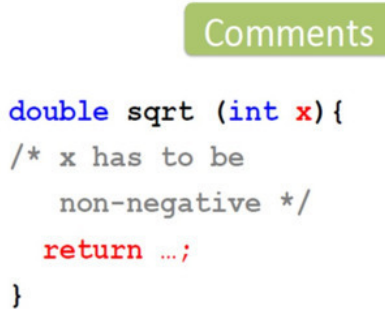
Программы могут содержать сотни тысяч строк кода.

И очень сложно отслеживать все возможности.

Поэтому мы также можем использовать языковые конструкции, чтобы избежать ошибки, как для себя, так и для других программистов, которые могут использовать ваш код.

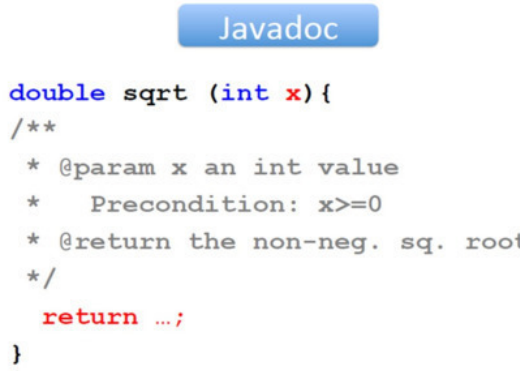
Для этого можно использовать комментарии.

Комментарии представляют собой текст, чередующийся с кодом, и этот текст не должен выполняться компьютером, а должен читаться людьми.



Еще одна возможность – это изготовить сопроводительную документацию к программе.

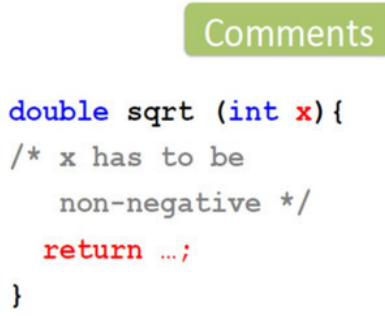
Javadoc – это инструмент, который является генератором документации на основе специальных комментариев.



Если вы используете эти специальные комментарии, вы можете автоматически создать хорошую документацию.

Таким образом, комментарий представляет собой текст в программе, бесполезный с точки зрения компьютера, но который может быть полезен для программиста.

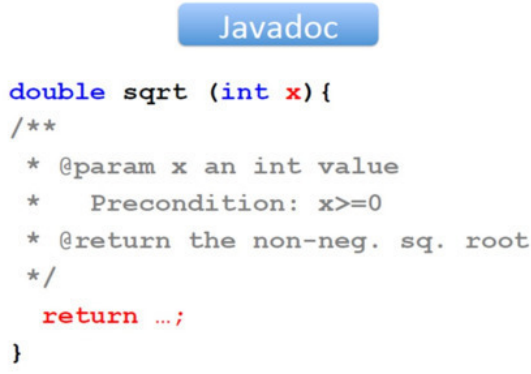
Здесь мы видим один из возможных способов написания комментария.



Комментарий начинается с косой черты и звездочки и заканчивается звездочкой и косой чертой.

Комментарий может включать в себя несколько строк.

Здесь у нас есть еще один комментарий.



Это комментарий, так как он начинается с косой черты и звездочкой и заканчивается несколькими строками позже звездочкой и косой чертой.

Но на разных строках есть еще несколько звездочек.

И это указание для специальной программы под названием Javadoc.

Javadoc принимает в качестве входа Java-код с этими специальными комментариями и выдает документацию для ее использования программистами.

Специальные команды, такие как @param и @return, имеют смысл, который Javadoc понимает при подготовке итоговой документации.

Операционная система компьютера, веб-браузер, приложения мобильного телефона, все они – состоят из очень

сложных частей программного обеспечения.

Например, смартфон с операционной системой Android имеет более 12 миллионов строк кода.

Из них более 2 миллионов написано на языке Java.

Представьте себе, что вы кодируете все эти строки самостоятельно.

Вам понадобится много времени.

Как программистам, нам нужно работать с другими программистами для достижения цели.

Нам также необходимо расширять или изменять предыдущие программы, написанные другими людьми, которых мы не знаем.

Также и другие программисты вполне вероятно будут работать с нашим кодом.

Попытка понять все строки кода, которые нам нужно использовать, требует огромных усилий.

Поэтому очень полезно писать заметки в наших программах, чтобы помочь другим и нам самим понять код, используя человеческий язык в этих заметках, но при этом не подвергая опасности выполнение нашей программы.

Эти примечания в программе – это то, что мы называем комментариями.

Это дополнительный текст, который мы добавляем в наш код, чтобы улучшить его читаемость и повторное использование.

Эти комментарии прозрачны для компьютера, поскольку

они служат только для людей, но не имеют вычислительного смысла.

Как и во всем, что есть в жизни, существуют разные подходы в том, как мы можем писать эти комментарии.

Комментарии полезны для разных целей.

Например, описание кода, то есть резюмирование целей сегмента кода.

Описание алгоритма, который вы создаете.

Комментирование сегмента кода, который не работает должным образом.

Или автоматическое создание документации.

В Java существуют разные способы написания комментариев.

Сначала, мы сосредоточимся на тех типах комментариев, которые направленны на предоставление сведений о вашем коде вам и другим программистам.

Если для нашего комментария нужна только одна строка, мы будем писать две косые черты перед текстом комментария.

```
// This is a single line comment in Java
```

```
/* This is a multiple
   line comment in Java */
```

И комментарий будет идти до конца строки.

Если мы хотим включить комментарий из нескольких строк, мы будем писать косую черту, за которой следует звездочка.

И мы закончим комментарий звездочкой, а затем косой чертой.

При этом начало и конец комментария могут быть в одной строке или в разных строках.

Будьте осторожны и избегайте вложения друг в друга этих типов комментариев.

Существуют рекомендации по написанию кода на языке Java.

Советуют использовать комментарии с несколькими строками только при комментировании блока кода.

И использовать однострочные комментарии для всего остального.

Вы можете задаться вопросом, сколько комментариев вы можете вставить в свой код.

Для этого нет однозначного ответа.

Убедитесь, что ваши комментарии соответствуют вашему коду.

Не забывайте обновлять свои комментарии при изменении кода.

Хороший программист создает не только хороший код, но также предоставляет другим возможность использовать свой код.

То есть, дает хорошие комментарии.

Есть еще один полезный и почти обязательный тип комментариев, который предназначен для создания подробной документации о нашем коде.

Существует программа под названием Javadoc, которая генерирует документацию из кода Java в HTML-файлы, чтобы мы могли легко их прочитать в нашем браузере.

Документация в Java-коде должна начинаться с косой черты, а затем идут две звездочки, и заканчивается одной звездочкой, а затем косой чертой.

```
/**
* Description: Separate paragraphs with <p>
* Blank comment line
* @ tags
*/
```

```
/**
 * Returns the number of pixels of an image
 *
 * @param x the width of the image, measured in pixels
 * @param y the length of the image, measured in pixels
 * @return the number of pixels of the image
 */
 int numberPixels(int x, int y);
```

Javadoc просматривает вашу программу, ища строки, начинающиеся с косой черты и двух звездочек, и создает HTML-документацию.

Но почему мы должны использовать этот комментарий?

Вместо поиска комментариев в миллионах строк кода, вы можете открыть веб-страницу и найти всю важную информацию о программе.

Когда мы говорим в Java об автоматической генерации документации, мы используем термин Javadoc.

Какую информацию мы должны включить в Javadoc?

На сайте Oracle вы можете найти руководство по эффективной практике написания комментариев для инструмента Javadoc.

Мы попытаемся обобщить наиболее важные из них, ис-

пользуя пример.

Мы начнем с определения Javadoc-комментария.

Комментарий Javadoc написан в формате HTML и должен предшествовать коду.

Он состоит из двух частей: описания и блока тегов.

Рассмотрим теги, которые вы должны использовать и как их использовать.

Давайте посмотрим на метод, который здесь указан, и вид информации, которая должна быть предоставлена для него в Javadoc.

Вы должны начать свой комментарий Javadoc с краткого и полного описания того, что делает этот метод.

Если в вашем Javadoc-комментарии есть несколько абзацев, разделите их тэгом p.

Затем вставьте пустую строку комментария, между описанием и блоком тегов.

Обратите внимание, что каждый комментарий Javadoc имеет только одно описание.

И как только инструмент Javadoc найдет пустую строку, он решит, что описание закончено.

Затем вы используете теги для добавления информации о вашем методе.

Наконец, вы должны поместить в конце строку со звездочкой и косой чертой, чтобы отметить конец комментария Javadoc.

Какая информация должна быть включена в блок тегов?

Для описания метода нам понадобятся, в основном, два типа тегов – @param и @return.

@param описывает аргумент метода.

И его необходимо указать для всех аргументов метода.

За тегом всегда следует имя аргумента.

Это имя всегда указывается в нижнем регистре.

Затем идет описание аргумента.

Далее вы должны всегда указывать тип данных аргумента.

Единственным исключением является тип данных, int, который вы можете опустить.

Чтобы разделить имя, описание и тип данных аргумента, вы можете добавить один или несколько пробелов.

Теги @param должны быть перечислены в порядке объявления аргумента.

Что касается описания, если это фраза без глагола, начните его с маленькой буквы.

Если это предложение с глаголом, начните его с заглавной буквы.

Таким образом, Javadoc – это полезный инструмент, который позволяет программистам автоматически генерировать HTML-страницы с документацией из их кода.

Исключения

Объектно-ориентированное программирование на Java

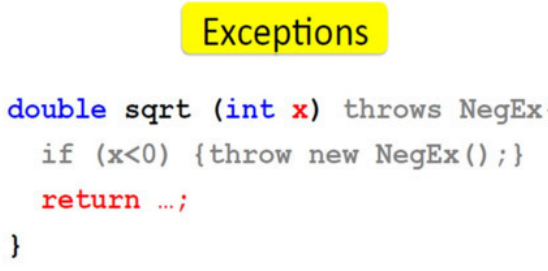
Основные конструкции языка Java

Лекция 14

Исключения

Когда мы говорили о том, что мы можем сделать в случае метода, который не определен для всех возможных входных значений, мы сказали, что мы можем запрограммировать, что нужно сделать в исключительной ситуации.

То есть мы можем программировать, что делать в обычных случаях, и что делать в исключительных случаях.



Это сделает нашу программу более надежной.

Таким образом, у нас есть исключения.

В этом случае мы используем не комментарии, а используем конструкции программирования языка Java.

Мы программируем, что делать для значений, которые не желательны.

Часто бывает, что наши программы хорошо написаны, их синтаксис и последовательность инструкций верны.

И они прекрасно компилируются.

Но когда мы их запускаем, возникают ошибки.

Java обрабатывает ошибки, возникающие в наших программах, во время выполнения с использованием исключений.

Oracle определяет исключения как события, которые про-

исходят во время выполнения программы, и которые нарушают нормальный поток выполнения инструкций.

Однако важно учитывать, что совсем не плохо иметь программы, которые выбрасывают исключения.

Исключения позволяют отделить основную логику программы от действий, которые нужно предпринять, когда происходит что-то необычное.

Кроме того, исключения позволяют классифицировать и дифференцировать типы ошибок систематическим образом.

Таким образом, исключение – это ошибка, возникающая во время выполнения программы. Исключения могут возникать во многих случаях, например:

Пользователь ввел некорректные данные.

Или файл, к которому обращается программа, не найден.

Или сетевое соединение с сервером было утеряно во время передачи данных.

Рассмотрим некоторые из наиболее распространенных исключений в программах Java, а также механизм их обработки, чтобы обеспечить нормальный поток программы, даже если во время выполнения происходят ошибки.

Первое исключение, которое мы здесь видим, это ArithmeticException.

ArithmeticException

```
int a = 1;
int b = 0;
System.out.println(a/b);


int a = 1;
int b = 0;
System.out.println(a%b);
```

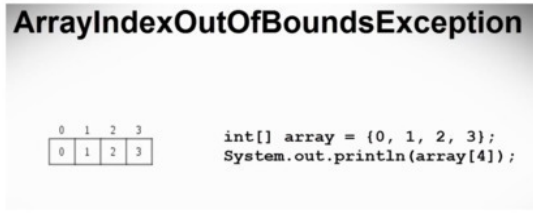
```
void printDivision (int a, int b){
    if (b != 0){
        System.out.println(a/b);
    } else {
        System.out.println("NaN");
    }
}
```

Это исключение выбрасывается при возникновении арифметических ошибок, например, при делении целого на ноль.

Эти сегменты кода вызывают исключение ArithmeticException.

Если вы программируете метод, в котором вы используете математическое деление, вы всегда должны проверять, что делитель отличается от нуля, так как вы не знаете, какие значения будут иметь аргументы, которые вы принимаете в этом методе.

Другим распространенным исключением является ArrayIndexOutOfBoundsException.



Это исключение вызывается, когда пытаются получить доступ к элементу массива с недействительным индексом.

Если мы определим, например, четырехэлементный массив, то это исключение будет выбрасываться при попытке доступа к элементу массива с индексом четыре или выше, а также при попытке доступа к элементу массиву с отрицательным индексом.

Поэтому, когда метод должен получить доступ к элементу в массиве, важно сначала проверить, что нужная позиция находится в пределах массива.

ArrayIndexOutOfBoundsException

```
void printElementInArray (int[] array, int index){
    if (index >= 0 && index < array.length){
        System.out.println(array[index]);
    } else {
        System.out.println("The index is not correct");
    }
}
```

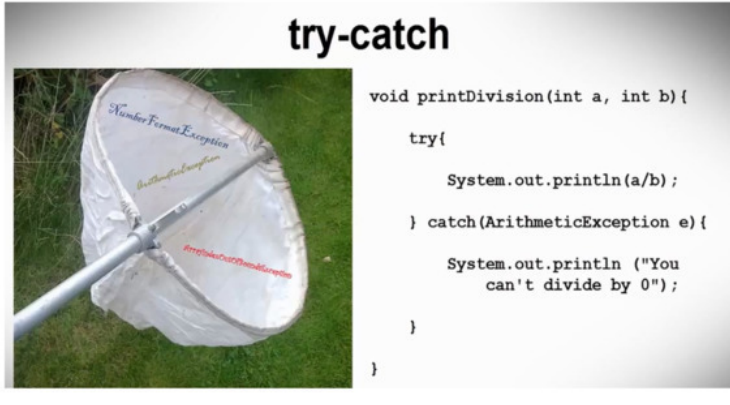
Еще одно исключение, это NumberFormatException.

NumberFormatException

```
String op1 = "1";
String op2 = "s";
System.out.println(op1 + op2); // 1s
System.out.println(Integer.parseInt(op1) +
    Integer.parseInt(op2));
    // NumberFormatException
```

Это исключение вызывается, когда пытаются преобразовать строку в числовой тип, например, double или integer, но при этом строка не содержит числа.

Стандартный способ управления этими инструкциями, которые могут выбросить исключение, это заключить их в оператор try-catch.



Здесь вы видите, что код метода printDivision заключен в выражение try-catch.

В этом случае нет необходимости проверять значение b, делителя.

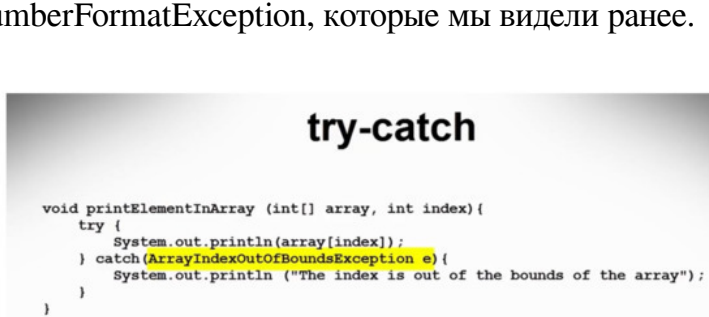
Если b отличен от нуля, будет выполнен метод System.out.printIn (a / b);

В противном случае Java выбросит исключение

ArithmeticException с сообщением, что вы не можете делить на ноль.

Поток программы будет продолжен, как обычно, после обнаружения этого исключения.

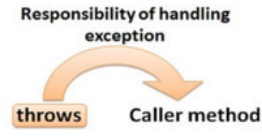
Выражение try-catch также может быть применено к примерам ArrayIndexOutOfBoundsException и NumberFormatException, которые мы видели ранее.



В обоих случаях, и благодаря выражению try-catch, мы можем гарантировать, что, если возникает исключение, пользователь получит соответствующую информацию, и поток выполнения программы продолжится далее нормально.

Вы также можете не обрабатывать исключения блоком try-catch в методе, а передать это право вызывающему этот ме-

тод.



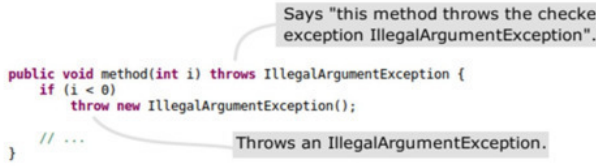

```
тип имя_метода(список_параметров) throws
список_исключений {
  // код внутри метода
}
```

```
public void method(int i) throws IllegalArgumentException {


}
```

В этом случае вы используете ключевое слово throws, которое прописывается в сигнатуре метода, и обозначающее что этот метод потенциально может выбросить исключение с указанным типом.

И уже в вызывающем коде обработать вызов этого метода блоком try-catch.

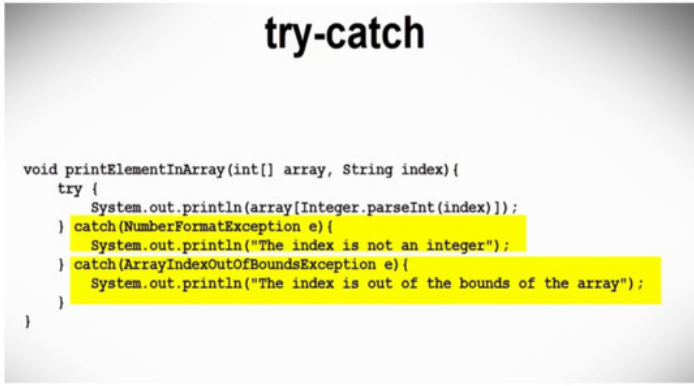
Также вы можете сами, специально, не виртуальная машина, а вы – выбросить исключение с помощью ключевого слова throw, указав тип исключения.



Например, чтобы предотвратить выполнение кода, когда параметр метода не является возможным входным значением.

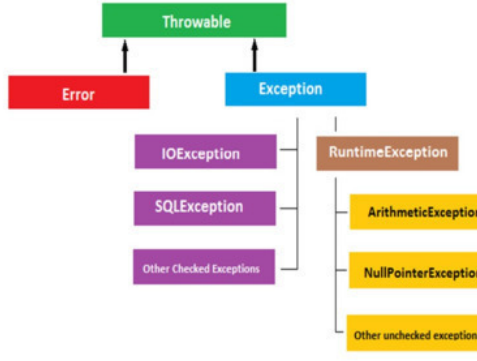
Таким образом, ключевое слово throw – служит для генерации исключений.

Блок try может иметь несколько блоков catch, каждый из которых имеет дело с конкретным исключением.



Если блок try генерирует исключение, то соответствующий блок catch обработает исключение, и программа будет продолжена.

Встроенные исключения Java имеют определенную иерархию.



Все классы, представляющие ошибки являются наследниками класса java.lang.Throwable.

Только объекты этого класса или его наследников могут быть «выброшены» JVM при возникновении какой-нибудь исключительной ситуации, а также только эти исключения могут быть «выброшены» во время выполнения программы с помощью ключевого слова throw.

Поэтому, если вы хотите создать свой класс исключения, он должен происходить от класса Throwable, или более точнее от класса Exception.

Также нужно учитывать, что все исключения делятся на «проверяемые» (checked) и «непроверяемые» (unchecked).

checked exception – проверяемое исключение, которое

проверяется компилятором.

Throwable и Exception и все их наследники, за исключением наследников Error-а и RuntimeException – проверяемые.

Error и RuntimeException и все их наследники – не проверяемые компилятором исключения.

Компилятор при компиляции проверяет код на возможность выброса при выполнении кода проверяемого исключения.

И так как проверяемое исключение проверяется во время компиляции, возникнет ошибка компиляции, если проверяемое исключение не обработано блоком try-catch, или оно не объявлено в заголовке или сигнатуре метода с помощью ключевого слова throws.

```
public class App {
   public static void main(String[] args) {
      f(); // тут ошибка компиляции
   }

   public static void f() throws Exception {
   }
}
```

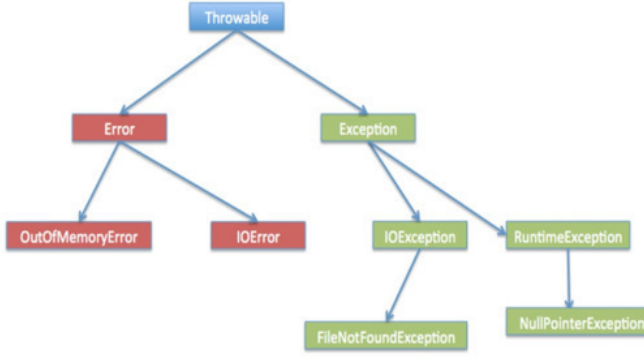
Так почему не все исключения являются проверяемыми?

Дело в том, что если проверять каждое место, где теоретически может быть ошибка, то ваш код сильно разрастется, и станет плохо читаемым.

И язык Java будет полностью непригодным для использования в качестве языка программирования.

Например, в любом месте, где происходит деление чисел, нужно было бы проверять на исключение ArithmeticException, потому что возможно деление на ноль.

Эту проверку создатели языка оставили программисту на его усмотрение.



Таким образом, исключение RuntimeException является не проверяемым и выбрасывается во время выполнения Java

кода, и его дочерние исключения также являются не проверяемыми.

Это исключение IndexOutOfBoundsException – выбрасывается, когда индекс некоторого элемента в структуре данных не попадает в диапазон имеющихся индексов.

Исключение NullPointerException – выбрасывается, когда ссылка на объект, к которому вы обращаетесь, хранит null.

Исключение ClassCastException – это ошибка приведения типов.

И исключение ArithmeticException – выбрасывается, когда выполняются недопустимые арифметические операции, например, деление на ноль.

Исключение Error также является не проверяемым, которое показывает серьезные проблемы возникающие во время выполнения приложения. Исключение Error сигнализирует о ненормальном ходе выполнения программы, т.е. о каких-то критических проблемах.

И его дочерние исключения, также не проверяемые, ThreadDeath – вызывается при неожиданной остановке потока.

Исключение StackOverflowError – ошибка переполнение стека. Часто возникает в рекурсивных функциях из-за неправильного условия выхода.

И исключение OutOfMemoryError – ошибка переполнения памяти.

Из описания этих не проверяемых исключений видно, что

обработать все эти возможные ситуации в коде невозможно, иначе весь код – это будет сплошной try-catch.

Теперь, при использовании множественных операторов catch обработчики подклассов исключений должные находиться выше, чем обработчики их суперклассов.

Иначе, суперкласс будет перехватывать все исключения, имея большую область перехвата.

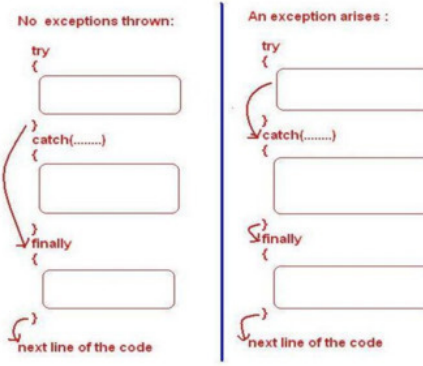
Иными словами, Exception не должен находиться выше ArithmeticException и ArrayIndexOutOfBoundsException.

И еще, операторы try могут быть вложенными.

Если вложенный оператор try не имеет своего обработчика catch для определения исключения, то идёт поиск обработчика catch у внешнего блока try и т. д.

Если подходящий catch не будет найден, то исключение обработает сама система завершением программы.

Таким образом, проверка на проверяемые исключения происходит в момент компиляции, а перехват исключений блоком catch происходит в момент выполнения кода.



Теперь, есть еще одна конструкция в обработке исключений, это блок finally.

Когда исключение передано, выполнение метода направляется по нелинейному пути.

Это может стать источником проблем.

Например, при входе метод открывает файл и закрывает при выходе.

Чтобы закрытие файла не было пропущено из-за обработки исключения, используется блок finally.

Ключевое слово finally создаёт блок кода, который будет выполнен после завершения блока try/catch, но перед кодом, следующим за ним.

Блок будет выполнен, независимо от того, передано исключение или нет.

Оператор finally не обязателен, однако каждый оператор try требует наличия либо catch, либо finally.

Таким образом, блок finally всегда выполняется, когда блок try завершается.

Это гарантирует, что блок finally будет выполнен, даже если произойдет непредвиденное исключение.

Также блок finally позволяет программисту избежать случайного обхода нужного кода.

Включение необходимого для выполнения кода в блок finally всегда является хорошей практикой, даже если не ожидается никаких исключений.

Однако блок finally не всегда может выполняться.

Если виртуальная машина JVM завершает работу во время выполнения кода try или catch, блок finally может не выполняться.

Аналогично, если поток, выполняющий код try или catch, прерывается или убивается, блок finally может не выполняться, даже если программа в целом продолжается.

Блок finally – это ключевой инструмент для предотвращения утечек ресурсов.

Закрывая файл или восстанавливая ресурсы, поместите код в блок finally, чтобы гарантировать, выполнение необходимых операций.

Рассмотрим этот пример.

```
public static void main(String[] args) {
    System.out.println("sout");
    throw new Error();
}
```

Каким здесь может быть вывод в консоль?

Здесь вполне возможна ситуация, когда в консоль сначала будет выведено сообщение об ошибке, а только потом вывод System.out.println.

Так как вывод System. out является буферизированным, то есть сообщения сначала помещаются в буфер, прежде они будут выведены в консоль.

А сообщение необработанного исключение выводится через не буферизированный вывод System.err.

Как уже было сказано, каждый оператор try требует наличия либо catch, либо finally.

```
public static void main(String[] args) {
    try {
      System.err.println("try");
    } finally {
      System.err.println("finally");
    }
  }
```

```
public static void main(String[] args) {
    try {
      return;
    } finally {
      System.err.println("finally");
    }
  }
```

```
public static void main(String[] args) {
    try {
      System.exit(42);
    } finally {
      System.err.println("finally");
    }
  }
```

Поэтому возможна конструкция try – finally.

И блок finally получит управление, даже если try-блок завершится исключением.

И блок finally получит управление, даже если try-блок завершится директивой выхода из метода.

Однако блок finally НЕ будет вызываться, если мы убъем виртуальную машину JVM.

При всем при этом, надо отметить, что блок finally не перехватывает исключение, и программа завершиться ошибкой при возникновении в блоке try исключения.

Исключение перехватывает только блок catch.

Таким образом мы разобрали почти все случаи работы операторов try, catch, throws, throw, и finally.

Рекурсия

Объектно-ориентированное программирование на Java

Основные конструкции языка Java

Лекция 15

Рекурсия

В некоторых случаях нам нужно выполнять повторные вычисления.

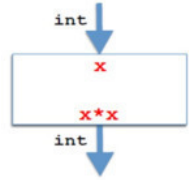
И мы видели циклы for и while, которые выполняют повторные вычисления.

Теперь мы увидим гораздо более мощный механизм повторных вычислений, который называется рекурсией.

Ранее мы определили метод square, который, принимая целое число, возвращает квадрат числа.

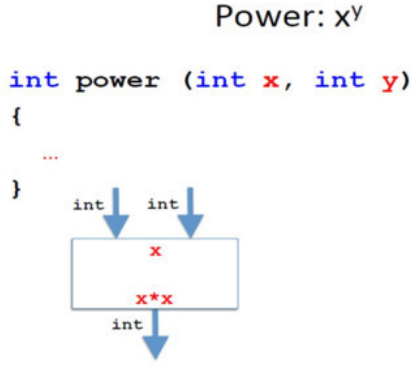
Square: x^2

```
int square (int x)
{
  return x*x;
}
```



Теперь мы хотели бы определить метод, который возводит в степень.

Мы хотим определить метод, который, учитывая базу x и показатель y, вычисляет x в степени y.



Поэтому, если y равно 2, мы вычисляем квадрат числа, как и раньше.

Вы видите, что в этом методе мы имеем два аргумента, целые числа x и y.

Давайте сначала попытаемся определить этот метод.

Давайте проанализируем несколько случаев.

Если y равно 0, то результат x равен степени 0, т. е. 1.

Если y равно 1, результат будет сразу x.

Если y равно 2, результатом является квадрат x.

Мы можем вызвать метод square, который мы определили ранее.

Если y равно 3, мы имеем x в кубе, предполагая, что у нас есть метод, называемый cube, определенный заранее.

И далее нам понадобятся другие методы для всех различ-

ных значений y, которые могут быть приняты.

Power: x^y

```
int power (int x, int y)
{
  if (y==0)       {return 1;}
  else if (y==1) {return x;}
  else if (y==2) {return square(x);}
  else if (y==3) {return cube(x);}
  … // until when?
}
```

Теперь мы можем заменить вызовы методов square, cube, и т. д. следующим кодом.

Таким образом, мы будем иметь x умножить на x, x умножить на x умножить на x и т. д.

Power: x^y

```
int power (int x, int y)
{
  if (y==0)       {return 1;}
  else if (y==1) {return x;}
  else if (y==2) {return x*x;}
  else if (y==3) {return x*x*x;}
  … // not very intelligent!
}
```

Сейчас это немного лучше, но все же очень плохо, потому что порождает бесконечный код.

Но мы все же кое-чему научились.

Чтобы вычислить x в степени y, мы должны умножить x y раз.

Но мы должны учитывать, является ли эта процедура применима для всех целых чисел y?

Нет.

Только для y больше или равно 0.

Для отрицательного y нам понадобится другой способ умножения.

Если у нас есть повторное умножение, мы можем использовать цикл.

Power: x^y

```
int power (int x, int y)
{ // y>=0
  int z=1;
  for (int i=1; i<=y; i++)
    {z=x*z;}
  return z;
}
```

Вот пример того, как мы можем это сделать.

Мы инициализируем целочисленную переменную z в 1, а затем вводим цикл.

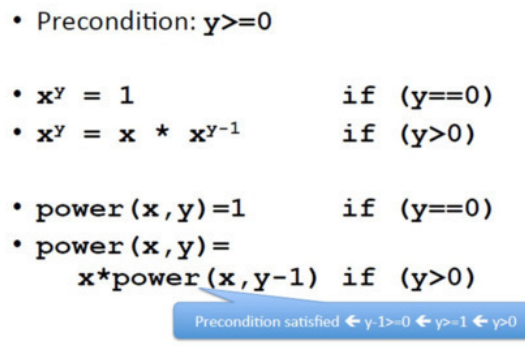
Счетчик i инициализируется 1 и увеличивается на 1 при каждом прогоне цикла.

Этот счетчик отслеживает, сколько x мы умножаем и накапливаем с помощью z.

И мы должны выполнять тело цикла ровно y раз, пока i не станет равен y.

Затем мы выходим и возвращаем накопленное значение в z.

Давайте проанализируем это снова.



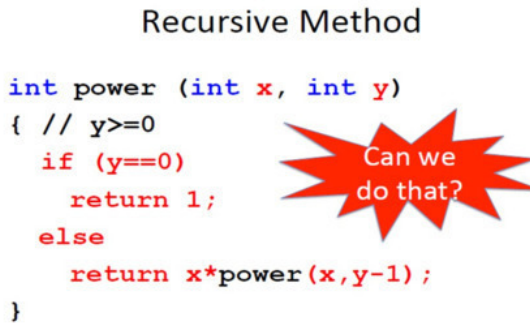
x в степени y равно 1, если y равно 0.

А если y строго больше 0, то x в степени y равно x умножить на x в степени y минус 1.

Это то, что в математике называется рекуррентным уравнением.

И мы можем написать это на Java в виде вызова функции power.

Если y равно 0, возвращаем 1.



Иначе, возвращаем x умножить на вызов этой же функции с x и y минус 1.

Таким образом, тот же метод, который мы определили с помощью цикла, может быть определен с помощью рекурсии.

Оба эти способа эквивалентны.

Но рекурсия позволяет записать сложное поведение простым способом, который потребует довольно сложного программирования при использовании циклов.

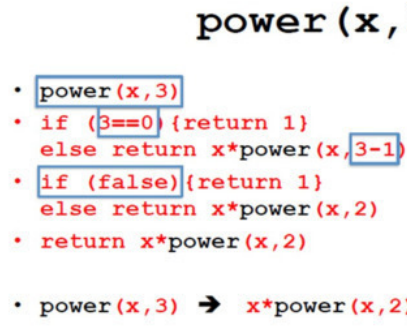
Рекурсию можно сравнить с матрешкой.

Чтобы понять это вернемся к рекурсивному методу, который мы определили.

И давайте упростим последовательно вызов этого метода для небольшой степени, чтобы увидеть, что происходит.

Начнем с x в 3 степени.

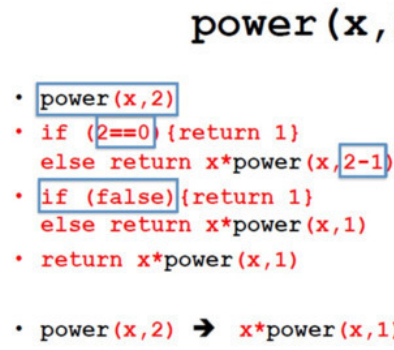
Мы можем заменить вызов метода, используя определение метода.



Таким образом, мы пишем весь код метода, подставляя вместо y 3.

И в этой последовательности выражений мы переходим от вызова метода с параметрами (x, 3) к вызову метода с параметрами (x, 2).

Пишем весь код метода, подставляя вместо y 2.



И в этой последовательности выражений, мы перешли от вызова метода с параметрами (x, 2) к вызову метода с параметрами (x, 1).

И переходим к вызову метода с параметрами (x, 0).

power(x,1)

- power(x,1)
- if (1==0){return 1}
  else return x*power(x,1-1)
- if (false){return 1}
  else return x*power(x,0)
- return x*power(x,0)

- power(x,1) → x*power(x,0)

x в степени 0 равно 1.

power(x,0)

- power(x,0)
- if (0==0){return 1}
  else return x*power(x,0-1)
- if (true){return 1}
  else return x*power(x,-1)
- return 1

- power(x,0) → 1

Теперь нам нужно собрать все вместе.

power (x, 3) равно x умножить на power (x, 2).

- `power(x,3)`
- `x*power(x,2)`
- `x*(x*power(x,1))`
- `x*(x*(x*power(x,0)))`
- `x*(x*(x*1))`
- `x*(x*x)`

A power (x, 2) равно x умножить на power (x, 1).

A power (x, 1) равна x умножить на power (x, 0), что равно 1.

Таким образом, мы получаем x умножить на x умножить на x умножить на 1.

Так работает рекурсия – сначала мы спускаемся как по лестнице вниз, а затем поднимаемся опять наверх.

Это изображение коробки с медсестрой, держащей меньшую коробку с тем же изображением.

Так что в теории, могут быть бесконечные медсестры и бесконечные коробки.

Но на практике нет бесконечных коробок, потому что изображение имеет некоторое разрешение, и мы не можем опуститься ниже 1 пикселя.

Таким образом, существует конечное число коробок.

Когда мы что-то вычисляем, мы должны заботиться о том, чтобы не создавать нежелательные бесконечные вычисления, которые нарушают нормальный поток вычислений.

Давайте посмотрим, что произойдет, когда мы что-то неправильно программируем.

Давайте рассмотрим, опять наш рекурсивный метод вычисления степени числа.

И давайте вызовем power (x, -2) для некоторого заданно-

го x.

power(x,y)

```
int power (int x, int y){ // y>=0
  if (y==0)
    return 1;
  else
    return x*power(x,y-1);
}
```

Для этого мы можем заменить вызов метода кодом.

power(x,-2)

- `power(x,-2)`
- `if (-2==0){return 1}`
  `else return x*power(x,-2-1)`
- `if (false){return 1}`
  `else return x*power(x,-3)`
- `return x*power(x,-3)`

- `power(x,-2)` → `x*power(x,-3)`

В результате мы перейдем к вызову метода power (x, -3).

В методе power (x, -3) мы перейдем к вызову метода power (x, -4).

power(x,-3)

- `power(x,-3)`
- `if (-3==0){return 1}`
  `else return x*power(x,-3-1)`
- `if (false){return 1}`
  `else return x*power(x,-4)`
- `return x*power(x,-4)`

- `power(x,-3)` → `x*power(x,-4)`

И так далее. Без конца.

- `power(x,-2)` → `x*power(x,-3)`
- `power(x,-3)` → `x*power(x,-4)`
- `power(x,-4)` → `x*power(x,-5)`
- `power(x,-5)` → `x*power(x,-6)`
- …

Мы получим бесконечные вычисления в теории.

На практике мы получим переполнение в какой-то момент и ошибку.

Что же мы сделали не так?

В этом случае мы не соблюдали комментарий, что у должно быть больше или равно 0.

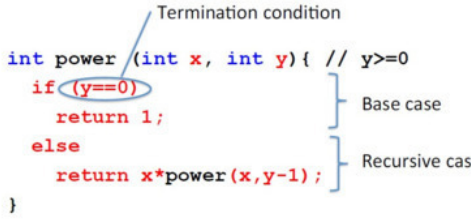
Поэтому мы должны учитывать две важные вещи.

Во-первых, рекурсия хороша, но мы можем перейти к бесконечным вычислениям.

И во-вторых, чтобы избежать этого, мы должны понять условия, при которых рекурсивный метод фактически завершается.

Может быть определенное количество рекурсивных вызовов, но в какой-то момент, нам нужно достичь не рекурсивного случая.

Поэтому при определении рекурсивного метода, всегда должны быть некоторые значения, для которых метод не вызывается рекурсивно.



Существует два способа чтения и понимания рекурсивных методов.

Один из них – это тот способ, который мы видели.

Другой, математический или нотационный способ, которые мы рассмотрим.

Предположим, нам дана задача написать рекурсивный метод.

Начнем с относительно простой задачи – написать метод на Java для вычисления факториала натурального числа.

В общем случае факториал натурального числа n вычисляется умножением всех натуральных чисел, начиная с 1 до n.

```
fac(n) = n! = n*(n-1)*…*2*1

fac(5) = 5! = 5*4*3*2*1
fac(4) = 4! =   4*3*2*1
```

Чтобы решить эту задачу, мы будем использовать следующую стратегию.

Первая часть состоит в том, что мы предполагаем, что задача решена для более простой задачи того же рода.

Предположим, что нам нужно вычислить факториал натурального числа n, но мы уже знаем, как вычислить факториал n минус 1.

```
fac(n)   =     n! = n*(n-1)*...*2*1
fac(n-1) = (n-1)! =   (n-1)*...*2*1
                n! = n*(n-1)!
           fac(n) = n*fac(n-1)
```

Если бы у нас был факториал n минус 1, мы просто бы умножили это число на n, чтобы получить факториал n.

Вторая часть стратегии – выявить случай, когда предыдущее рассуждение не выполняется.

Факториал 0 нельзя свести к более простому случаю, как мы это делали ранее.

```
fac(0) = 0! = 1
```

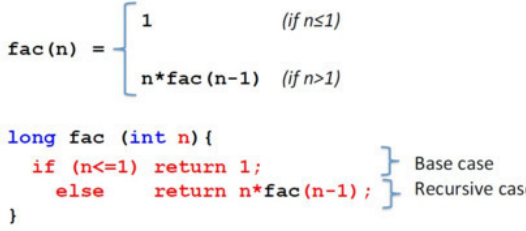
Так что это базовый случай.

Мы просто говорим, что факториал 0 равен 1.

Таким образом, факториал n равен 1, если n равно 0, и факториал n равен n умножить на факториал n минус 1, если n больше 0.

Теперь у нас есть основа для записи рекурсивного метода.

Из математического уравнения легко написать рекурсивный метод.

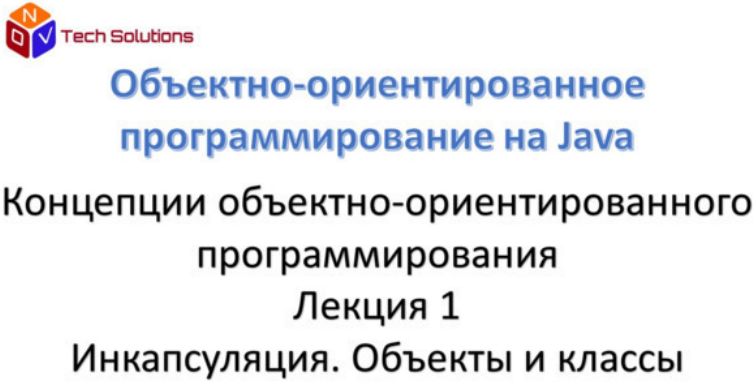


Там мы видим базовый случай, в котором нет рекурсивного вызова.

Базовый случай получается из пограничного случая.

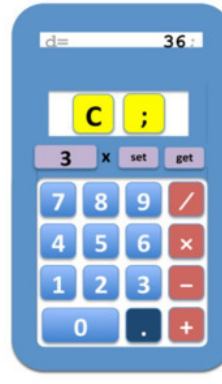
И мы также видим рекурсивный случай, вытекающий из приведения общего случая к более простому.

Инкапсуляция. Объекты и классы



Давайте посмотрим на вычислительные возможности калькулятора.

Как правило, калькулятор может делать две вещи: запомнить значения и вычислить новые значения.



Запомнить значения можно с помощью переменных.

И затем мы можем вычислять новые значения с помощью методов.

Например, мы можем сложить два значения, вычесть или умножить.

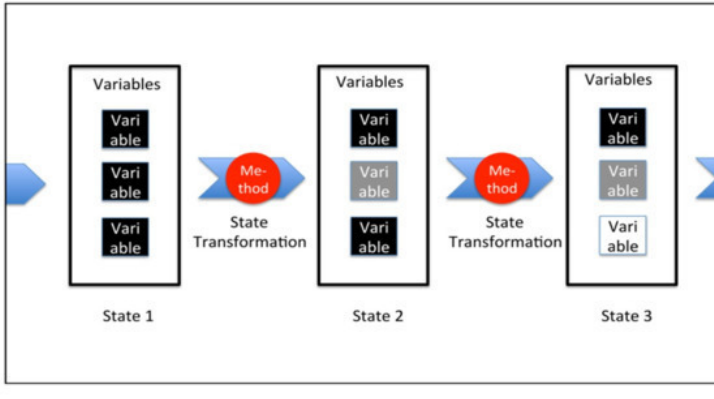
Таким образом, у нас есть методы, соответствующие арифметике, а также методы, чтобы получить или установить переменную x.

Когда мы пишем программу для моделирования этого калькулятора, и мы определяем для него переменные и методы, мы поместим, с одной стороны, все переменные вместе, а с другой стороны – все методы вместе.

Значения всех переменных в конкретный момент времени будут составлять состояние калькулятора.

И набор методов будет определять поведение калькулятора.

Наша модель будет меняться от одного состояния в другое со временем.



При этом состояние будет определяться значениями переменных.

А методы будут отвечать за изменение состояния.

На самом деле, определение переменных и методов – это общий способ моделирования объектов.

Эти объекты могут соответствовать физическим объектам, например, калькулятору.

Или эти объекты могут быть концептуальными, когда ваш код должен моделировать что-то новое.

Таким образом, это разделение состояния и поведения очень важно.

Представьте себе автомобиль, который моделируется в программе, которую вы пишете для игры.

Состоянием этого объекта может быть местоположение, цвет, включены ли фары или нет.

И методы могут быть изменением положения, включить свет фар и т. д.

Помните, что методы часто связаны с глаголами, потому что они подразумевают действие.

Теперь мы собираемся инкапсулировать переменные и методы в новую для нас конструкцию программирования, называемую объектом.

Эта концепция инкапсуляции является одной из ключевых концепций в так называемой объектно-ориентированной парадигме программирования.

Поэтому помните, что объекты имеют состояние, представленное отдельными переменными, которые также называются полями или атрибутами.

И поведение, то, что может делать наш объект, представлено методами.

Эти два компонента: состояние и поведение, не разбросаны по программе, а собраны и инкапсулированы в объекты.

Разные объекты могут иметь одинаковую структуру, и отличаться друг от друга только значениями переменных.

Поэтому мы можем сказать, что такие объекты принадле-

жат одному и тому же классу.

И наоборот, чтобы создать объект, сначала нам нужно сначала определить класс, который является шаблоном для создания объектов.

Рассмотрим пример с различными автомобилями, которые представлены различными объектами.

Все эти объекты принадлежат классу автомобилей Car, который имеет ряд атрибутов или переменных, или полей и ряд методов.

Давайте посмотрим на возможное определение, как мы можем записать этот класс на Java.

```
class Car {
  boolean lights;                    Attributes
  String color;
…
  void turnHeadlightsOn(){ligths = true;}
  void turnHeadlightsOff(){ligths = false;}
  void moveForward(){…;}
  void moveBackward(){…;}            Methods
  void turnRight(){…;}
  void turnLeft (){…;}
}
```

Здесь вы можете увидеть определение класса Car.

Вы можете увидеть зарезервированное ключевое слово

class.

Затем имя, которое мы хотим дать классу.

Обратите внимание, что мы пишем его с заглавной буквы.

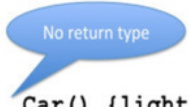
Затем мы указываем переменные с соответствующим типом.

Наконец, мы определяем методы, которые мы хотим дать всем объектам этого класса.

Но как только мы определили класс, как сконструировать объект для этого класса?

Для этого у нас есть конструкторы.

Конструкторы – это специальные методы, которые также включены в тело определения класса.



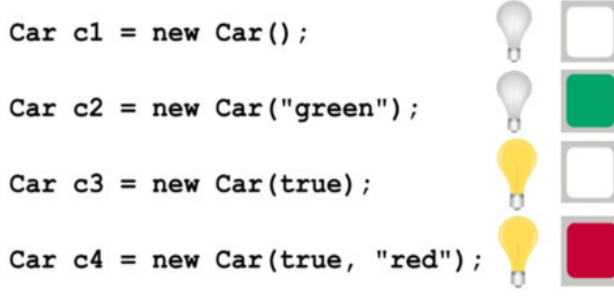
Constructors

```
Car() {lights=false; color="white";}

Car(String c)  {lights=false; color=c;}
Car(boolean b) {lights=b; color="white";}

Car(boolean b, String c)
  {lights=b; color=c;}
```

И они имеют имя класса.

Обратите внимание, что здесь не указан тип возвращаемого результата.

Используя конструкторы, мы можем создавать разные объекты класса.



Заметьте, что может быть не один, а несколько конструкторов.

Эти конструкторы отличаются списком параметров.

Здесь вы можете увидеть несколько возможных конструкторов для класса Car.

Также мы можем вообще не определять конструктор, и в этом случае при вызове конструктора объект создается со значениями по умолчанию.

Здесь мы видим несколько вызовов конструкторов, опре-

деленных ранее.

Посмотрите на объявление.

Сначала мы определяем объект с именем и обратите внимание, что классы работают как типы.

Сначала мы указываем Car, чтобы указать, что объект имеет тип или класс Car.

Затем знак равенства, зарезервированное ключевое слово new и вызов конструктора.

Таким образом, в итоге, чтобы определить объект, мы должны сначала определить класс, предоставляя набор полей и набор методов.

После определения класса мы можем создать объект как экземпляр класса, используя конструктор, предоставляемый классом.

Мы можем создать много объектов одного класса, каждый из которых будет со своим собственным состоянием.

Классы и типы

Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 2

Классы и типы

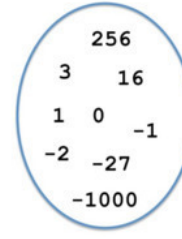
Классы – это шаблоны, из которых мы строим объекты.

И все объекты имеют одинаковую структуру, определенную классом.

Давайте сравним класс, который мы определили, со встроенным Java типом.

Так, например, с одной стороны, у нас есть класс «Car», который мы определили с такими методами, как «двигаться вперед» или «включать фары» и поля, такие как «свет» и «местоположение».

class Car



type int

И, с другой стороны, у нас есть целые числа типа «int».

И для этих целых чисел у нас есть ряд определенных операций или методов, таких как «сложение» или «умножение».

Давайте сосредоточимся на методах.

В обоих случаях методы связаны с объектами в классе или значениями данного типа.

Таким образом, классы похожи на типы, и объекты похожи на сложные значения.

Фактически, вы можете рассматривать классы как типы.

Типы, которые не являются встроенными Java типами, а типы, которые вы определили для решения какой-либо конкретной задачи.

При определении методов и конструкторов классы принимают роль типов.

Действительно, мы использовали строки так же как целые числа, для определения методов и переменных.

```
int a = 3;
String b = "abc";


String repeat(String s, int n){
  String t="";
  for (int i=0; i<n; ++i){t=t+s;}
}
```

И String- это класс, а «int» – это примитивный тип данных.

Здесь мы видим объявление переменной целого числа и переменной строки.

Иногда мы говорим о «ссылках», в случае объектов.

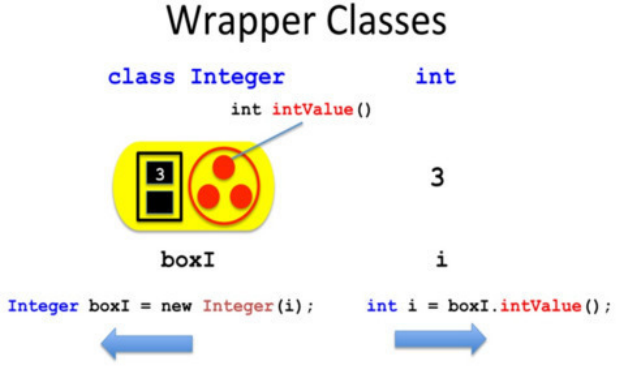
В нижней части мы видим объявление метода со String и «int» в качестве параметров.

Таким образом, вы можете рассматривать классы как типы – типы, определенных вами в соответствии с вашими потребностями.

На самом деле для каждого примитивного типа существует соответствующий класс, называемый «классом-оболоч-

кой».

Например, у нас есть тип «int» и класс «Integer».



И этот класс Integer является классом-оболочкой.

Объект класса «Integer» содержит поле с числом «int» и метод, который возвращает число «int», сохраненное в этом объекте.

Кроме того, там есть другие поля и методы, которые используются для разных целей.

Как вы можете видеть, для преобразования числа «int» в объект «Integer», мы можем использовать конструктор «Integer».

И для преобразования объекта Integer в значение «int», мы используем метод «intValue» класса «Integer».

Представление просто «int» в компьютере намного эффективнее, чем соответствующего объекта, так как существует много вещей, которые нужно хранить в объекте.

Класс – это не просто тип.

Во-первых, потому что он может содержать более одного поля.

Это можно понимать, как составное значение – значение с несколькими компонентами – например, тремя целыми числами.

Поэтому, классы – это хороший способ собрать несколько значений вместе в полях объекта.

И эти компоненты могут быть нескольких типов или классов.

В частности, они могут быть довольно сложными объектами, определенными как части данного объекта.

Представьте, что вы определили класс «Двигатель» с набором полей, которые определяют состояние двигателя и набором методов, которые определяют то, что вы можете делать с двигателем.

```
class Car {
  boolean lights;
  String color;
  Engine eng;
…
  void turnHeadlightsOn(){lights = true;}
  void moveForward(){…}
  void tuneEngine(int n){…}
…
}
```

У нас может быть объект класса «Двигатель» как атрибут или поле класса «Автомобиль».

Это дает большие возможности, так как концепция класса позволяет структурировать вашу программу или систему, определяя различные подсистемы.

Вы можете создавать объекты, используя другие объекты.

Но концепция объекта класса еще богаче.

Мы могли бы рассматривать тип данных как набор значений, вместе с некоторыми методами для них.

И у нас есть переменные, которые могут хранить эти значения.

Но значение само по себе не имеет состояния, и сам по себе тип данных не имеет состояния.

Напротив, объект имеет состояние.

Так как он имеет внутри переменные, и он может запоминать значения.

Классы можно рассматривать как типы, классы определяют типы.

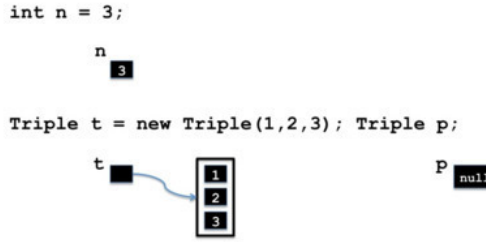
Но, кроме того, объекты имеют состояние.

Когда мы создаем новый объект – и мы делаем это с помощью ключевого слова «new» и конструктором – строки здесь являются исключением – и мы резервируем пространство в памяти для хранения значений полей.

После создания мы можем ссылаться на этот объект, создавая ссылку с именем.

Переменная и именем объекта будет хранить ссылку на объект.

Но в этих объяснениях мы не будем акцентировать внимание на идеи ссылки или указателя, поскольку Java как-то не поддерживает эту идею.



Другие языки программирования делают это.

Мы также не будем говорить об уничтожении объектов и освобождении памяти.

Потому что Java делает это автоматически с помощью так называемого «сборщика мусора», который автоматически освобождает память для объектов без какой-либо ссылки.

Когда ссылка указывает на отсутствие объекта, мы говорим, что его содержимое равно нулю.

Когда вы создали новый объект и дали ему имя – или, вернее, определили ссылку для него – как вы получаете доступ к полю или атрибуту?

Ответ заключается в использовании точечной нотации.

Вы хотите ссылаться на поле «n1» объекта «t».

```
Triple t = new Triple(1,2,3);

t.n1

t.get1()

get1(t)
```

Вы пишете «t.n1.»

И для методов мы делаем что-то подобное.

Чтобы вызвать метод «get1 ()» для объекта «t», мы пишем «t. get1 ()».

Область видимости

Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 3

Область видимости

Классы имеют двойную цель.

Они могут использоваться для определения новых типов данных, которые связаны с решением задачи.

И также они служат для структурирования кода.

У нас есть инкапсулированные переменные и методы в классах.

Однако любая другая часть программы может изменять переменные и вызывать методы.

И иногда важно скрыть доступ к некоторым переменным или методам, чтобы контролировать возникновение возмож-

ных проблем.

```
class Car {
          int gas; // from 0 to 100
  …
        void move(int n){…; gas-=n; …;}
        void fill(int n){…, gas+=n; …;}
}
```

Представьте себе, что в нашей модели автомобиля у нас есть поле gas, которое служит индикатором оставшегося топлива в машине.

Представьте, что эта переменная должна содержать значение от 0 до 100, 0 означает пустой бак, а 100 – полный.

Теперь, когда автомобиль тратит при движении n единиц топлива, эти n единиц вычитаются из переменной gas.

Мы также можем заполнить бак на АЗС, и в этом случае переменная gas увеличивается.

Таким образом, топливо уменьшается с помощью метода перемещения и увеличивается с помощью метода заполнения.

Однако у нас может быть проблема, поскольку любая часть программы имеет доступ к этой переменной gas.

Кто-то может даже изменить переменную на отрицательное число, что не имеет смысла.

Таким образом, два метода, которые должны изменять и должны иметь доступ к этой переменной, не являются единственным контролем этой переменной, и мы должны каким-то образом ограничить этот доступ.

```
class Car {
  private int gas; // from 0 to 100
  …
  public void move(int n){…; gas-=n; …;}
  public void fill(int n){…, gas+=n; …;}
}
```

Давайте посмотрим на эти два модификатора доступа, public и private.

На данный момент мы будем использовать их только для переменных и методов класса.

Здесь мы пишем private до объявления переменной gas.

Это означает, что мы можем получить доступ к ней только в классе, а не вне класса.

Два метода, move и fill, определяются как public, и поэтому могут быть вызваны вне класса.

Это типичная ситуация, чтобы иметь приватные переменные и публичные методы.

У нас также могут быть приватные методы, которые определены, например, как вспомогательные методы для других публичных методов.

```
public class Car {
   private int gas; // from 0 to 100
   …
   public Car () {gas=0;}
   public void move(int n){…; gas-=n; check();}
   public void fill(int n){…, gas+=n; check();}
   private void check()
    {if (gas<0) gas=0; if (gas>100) gas=100;}
}
```

Здесь мы видим метод check, который вызывается из move и fill, но нам не нужно вызывать этот метод вне класса.

Наконец, мы также ставим ключевое слово public перед классом.

Его смысл станет понятным позже.

Таким образом, извне класса, как правило, мы имеем доступ только к методам, а не к переменным.

Доступ к переменным имеют только методы.

Здесь мы разделили понятия инкапсуляции и сокрытие информации.

Хотя для некоторых эти две концепции идут вместе, то есть инкапсуляция всегда подразумевает сокрытие информации.

Как правило, мы хотим иметь приватные переменные экземпляра и публичные методы, которые получают доступ к этим переменным.

Но мы должны запрограммировать это явно с помощью ключевых слов «private» и «public».

Всегда рекомендуется делать переменные приватными.

А затем определять публичные методы для установки значений переменных и получения значений переменных.

```
public class Car {
  private int gas; // from 0 to 100
  …
  public Car () {gas=0;}
  public int getGas() {return gas;}
  public void setGas(int g) {gas=g;}
  …
}
```

Как правило, название этих двух типов методов соответствует одному и тому же шаблону:

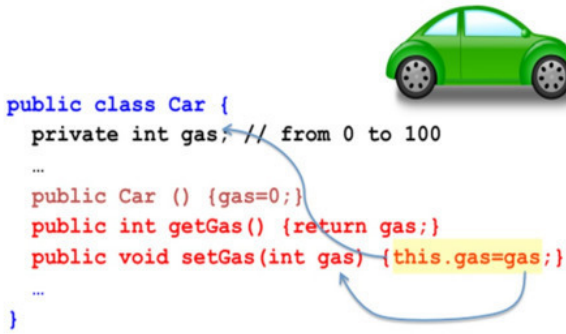
Как правило, имена этих методов начинаются со слова «set» и начинаются со слова «get».

Поэтому эти методы иногда называют сеттеры и геттеры.

Заметим, что в методе setGas мы имеем параметр g, который присваивается полю gas.

Иногда, мы хотим назвать параметр setGas тем же именем, что и переменную экземпляра.

И с этим не возникает никаких проблем.



Однако, если мы хотим отличить визуально параметр от поля, мы можем использовать ключевое слово this и точку перед именем.

Это означает, что это имя относится к полю класса.

В некоторых случаях нам нужно иметь поля класса, которые имеют общее значение для всех объектов в классе.

```
public class Car {
  static int noWheels=4;
  private int gas; // from 0 to 100
  …
  public Car () {gas=0;}
  public int getGas() {return gas;}
  public void setGas(int gas) {this.gas=gas;}
  …
}
```

Эти переменные называются переменными класса, а не переменными экземпляра класса, и они объявляются с помощью ключевого слова «static».

Эти переменные не создаются для каждого созданного объекта класса.

Они создаются только один раз для всех объектов класса.

И если мы изменим это значение, оно будет изменено для всех объектов.

Если мы не хотим, чтобы эта переменная менялась,

Мы можем сделать ее константой, добавив ключевое слово «final».

```
public class Car {
  static final int NOWHEELS=4;
  private int gas; // from 0 to 100
  …
  public Car () {gas=0;}
  public int getGas() {return gas;}
  public void setGas(int gas) {this.gas=gas;}
  …
}
```

Мы можем также сделать это и для переменных экземпляра.

По соглашению, имена таких переменных пишутся в верхнем регистре, заглавными буквами.

Как показано здесь.

Значения финальных переменных могут быть установлены только один раз.

Таким образом, теперь у нас есть разные виды переменных.

С одной стороны, у нас есть локальные переменные.

Затем у нас есть переменные экземпляра, которые создаются для каждого объекта или экземпляра класса.

Каждый объект может иметь свое значение, хранящееся в этой переменной.

Мы можем использовать ключевое слово «this» для обозначения этих переменных.

И у нас есть переменные класса, которые создаются только один раз для всех объектов одного класса.

Они объявляются с ключевым словом «static».

Статические переменные инициализируются только один раз, при запуске выполнения кода, при загрузке класса.

Эти переменные будут инициализированы первыми, прежде чем будут инициализированы любые переменные экземпляра.

И если вы хотите сделать переменную экземпляра или переменную класса неизменной, вы добавляете ключевое слово «final».

Наследование

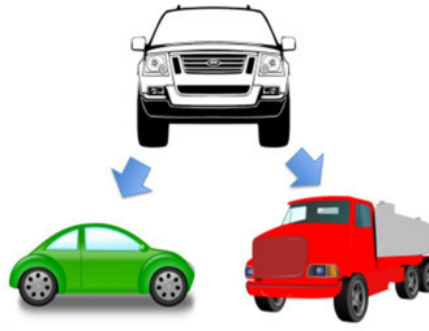
Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 4

Наследование

Рассмотрим две машины, принадлежащие к одному классу.

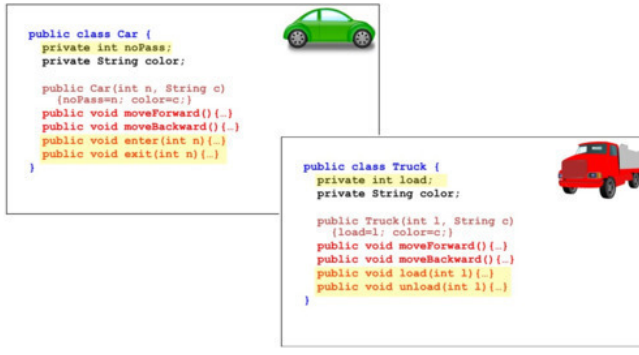


У них есть общие методы и поля, но в тоже время есть отличающиеся особенности.

И вместо того, чтобы создавать два разных объекта одного класса, а потом пытаться учесть их отличающиеся особенности с помощью отдельного кода, или создавать два разных несвязанных между собой класса, давайте сначала смоделируем общий класс автомобилей, а затем создадим класс легковых автомобилей и класс грузовиков, которые унаследуют общие поля и общие методы от общего класса автомобилей, но у них также будут и свои собственные поля, и методы.

Давайте посмотрим, как мы это делаем на Java.

Представьте, что у нас есть класс Car с этими полями и методами.



В частности, есть приватное поле количество пассажиров, noPass, которое содержит количество пассажиров в данный момент времени.

enter и exit- это методы, которые изменяют это число пассажиров.

Другой класс грузовиков имеет переменную загрузки, которая может быть изменена с помощью методов load и unload.

Имейте в виду, что не стоит называть переменную и метод одним и тем же именем.

Затем оба класса используют переменную цвет, а также методы для движения вперед

и назад.

Что мы можем сделать для упрощения кода, так это сна-

чала определить универсальный класс для транспортных средств.

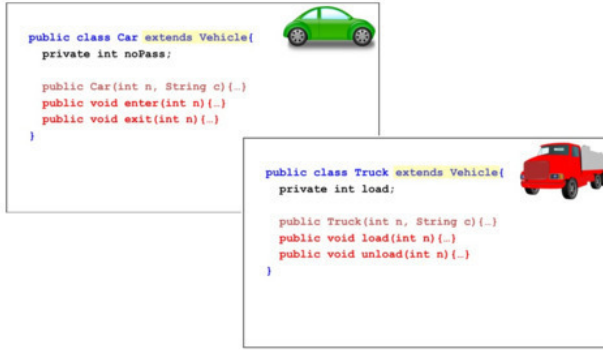
Этот класс будет иметь поля и методы, общие для всех автомобилей – в нашем случае – для легковых автомобилей и грузовиков.

```
public class Vehicle{
  private String color;

  public Vehicle(String c) {color=c;}
  public void moveForward(){…}
  public void moveBackward(){…}
}
```

Затем мы можем определить классы, car и truck, которые наследуют поля и методы от этого общего для них класса.

Vehicle будет называться суперклассом классов car и truck, и классы car и truck являются подклассами класса Vehicle.

Теперь мы можем определить класс car, расширив класс Vehicle, и добавить дополнительные поля и методы, которые может иметь легковой автомобиль.



А для грузовых автомобилей мы делаем то же самое: расширяем класс Vehicle такими полями и методами, которые необходимы.

Все остальные поля и методы унаследованы от класса Vehicle.

Обратите внимание, что мы не раскрыли тело конструктора.

Это требует дальнейшего объяснения и новых концепций.

Но вы должны знать, что класс может иметь несколько подклассов, тогда как класс не может быть подклассом более чем одного класса.

У одного класса не может быть двух суперклассов, не может быть двух родителей.

Таким образом, мы знаем, что один класс может расширить другой класс.

Например, если класс B расширяет класс A, это означает, что он наследует его поля и методы.

И это можно сделать многократно.

То есть класс B может быть расширен, например, классом C.

Теперь мы хотим проанализировать вопрос о том, как определить конструктор класса A, который расширяет другой класс.

В нашем определении класса vehicle и класса car, где класс car расширяет класс vehicle, мы определяем конструктор для класса vehicle, который инициализирует приватное поле color.

```
public class Vehicle{
  private String color;
  public Vehicle(String c) {color=c;}
  public void moveForward(){…}
  public void moveBackward(){…}
}
public class Car extends Vehicle{
  private int noPass;
  public Car(int n, String c)
   {color=c; noPass=n;}
  public void enter(int n){…}
  public void exit(int n){…}
}
```

И с этим не никаких проблем.

Но как мы можем определить тело конструктора car, с учетом двух аргументов, целого числа для количества пассажиров и строки для цвета?

Класс car наследует все методы от класса vehicle – перемещение вперед и назад, и все его поля, в данном случае, только color.

Но поле color является приватным полем и не может быть доступно извне класса vehicle.

Это относится также и к подклассам, и это очень важно.

Поэтому неправильно присваивать значение «c» полю color в классе car.

Мы не можем получить к этому полю доступ, потому что оно является приватным.

Мы можем использовать только публичный метод, например, конструктор.

Теперь, если мы хотим вызвать конструктор суперкласса, мы используем ключевое слово super.

```
public class Vehicle{
  private String color;
  public Vehicle(String c) {color=c;}
  public void moveForward(){…}
  public void moveBackward(){…}
}
public class Car extends Vehicle{
  private int noPass;
  public Car(int n, String c)
   {super(c); noPass=n;}
  public void enter(int n){…}
  public void exit(int n){…}
}
```

Здесь вы это видите.

super (c) – вызов конструктора vehicle (c).

Таким образом, мы сможем инициализировать поле color из подкласса.

Вызов конструктора суперкласса должен быть перед любым другим кодом в теле конструктора подкласса.

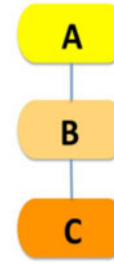
Например, сначала установить количество пассажиров, а затем вызвать супер будет неправильным.

Вы должны сначала вызвать супер, а затем включить любой другой вызов, который вам может понадобиться.

```
public class Vehicle{
  private String color;
  public Vehicle(String c) {color=c;}
  public void moveForward(){…}
  public void moveBackward(){…}
}
public class Car extends Vehicle{
  private int noPass;
  public Car(int n, String c)
   {          noPass=n; super(c);}
  public void enter(int n){…}
  public void exit(int n){…}
}
```

Здесь мы видим другой пример.

```
public class A {
  public A() {System.out.print("A, ");}
}

public class B extends A {
  public B() {super(); System.out.print("B, ");}
}

public class C extends B {
  public C() {super(); System.out.print("C.");}
}

C c = new C();
```



A, B, C.

У нас есть класс A с подклассом B, а класс B с подклассом C.

Диаграмма справа от вас показывает отношения наследования.

Класс A имеет конструктор без аргументов, который печатает строку A, пробел.

В классе B мы видим, что есть также конструктор без аргументов, который правильно вызывает сначала конструктор суперкласса A, затем печатает строку B, пробел.

В классе C конструктор без аргументов сначала вызывает конструктор его суперкласса B, а затем печатает строку C точка.

Теперь, что происходит, когда мы создаем новый объект класса C?

Конструктор C вызывает конструктор B, который в свою очередь, вызывает конструктор A.

Таким образом, печатается: A, пробел, B, пробел, C точка.

Подводя итог, первое, что нам нужно сделать в конструкторе подкласса, это вызвать конструктор суперкласса.

Приведение типов

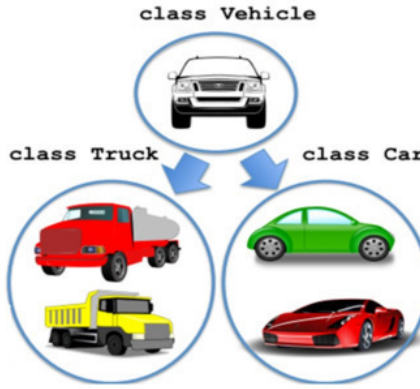
Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 5

Приведение типов

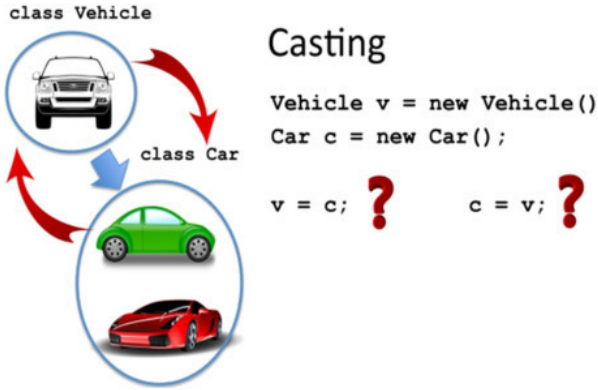
Давайте посмотрим снова на эту иерархию классов.



Легковой автомобиль и грузовик являются подклассами или производными классами класса vehicle.

Вопрос в том, если ли у нас есть объект класса car, мы можем использовать его там, где должны быть объекты класса vehicle?

Например, в переменной vehicle?

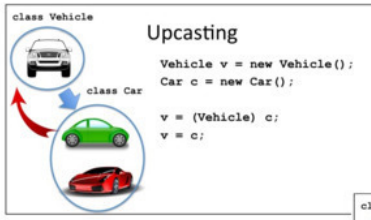


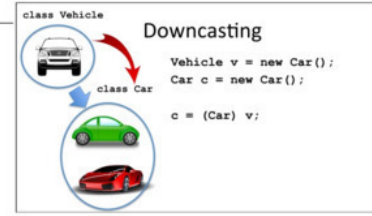
И наоборот, можем ли мы поместить объекты суперкласса там, где должны быть объекты подкласса?

И если да, то при каких обстоятельствах?

Мы говорим о кастинге или приведении при преобразовании объекта из одного класса к другому связанному классу.

Представьте себе, что у нас есть переменная vehicle, которая хранит объект vehicle, и переменная car, с сохраненным в нем объектом car.





Можем ли мы присвоить объект car переменной vehicle и наоборот?

Мы говорим о приведение к базовому типу при преобразовании объекта из класса в суперкласс.

И переход от подкласса к суперклассу всегда возможен.

Объекты подкласса наследуют все от суперкласса.

Поэтому все, что вы хотите сделать с переменной суперкласса, применимо к объекту подкласса.

Чтобы привести к базовому типу объект, вы можете указать суперкласс в круглых скобках, как вы здесь видите.

Но вы также можете не делать это, как вы видите в последней строке.

Мы говорим о понижающем приведении при конвертации объекта от класса к его подклассу.

Теперь мы хотим заставить vehicle стать car.

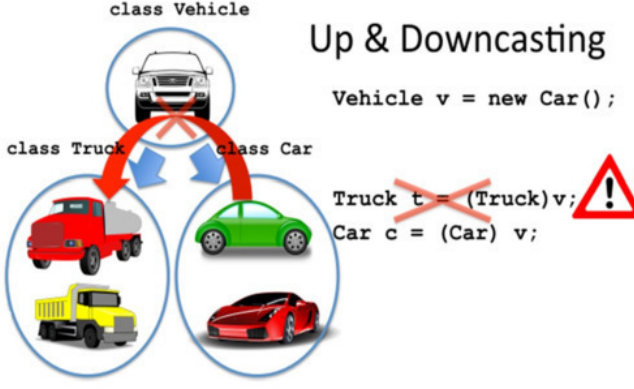
Мы переходим от общего класса к более конкретному классу, и это должно быть сделано явно.

В этом примере мы объявляем переменную типа vehicle, но храним в ней car.

Таким образом, мы можем явно понизить эту переменную для хранения car, который находится в переменной v.

Вы должны быть очень осторожны при кастинге вверх и вниз.

Мы объявляем переменную v, и мы храним в ней car.



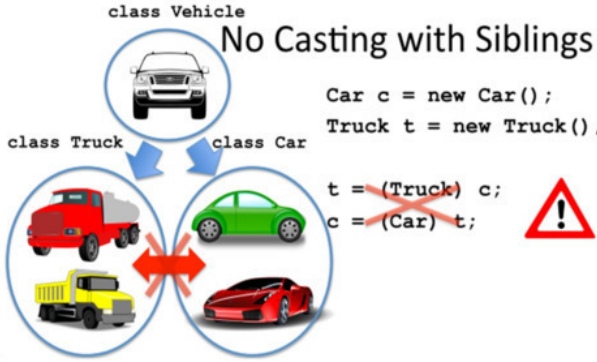
Мы можем это сделать, поскольку car является vehicle.

Однако вы не можете привести v в переменную truck.

Вы не можете сделать приведение между классами, полу-

ченными из одного класса.

Вы не можете превратить car в truck или truck в car.



У них разные поля и методы.

Преобразование применимо не только для классов.

Это также возможно с примитивными типами и между примитивными типами.

```
double d = (double) 3;          3.0

double e = 1.0 * 3;             3.0


int n = (int) 3.6;                  3

int m = 1234567890; float f = m;
int dif = m - (int)f;             -46
```

Мы видели несколько примеров со строками и целыми числами.

Это особый случай, когда нет необходимости явного преобразования числа в строку, а можно сделать это используя оператор плюс.

При кастинге вверх мы не теряем информацию о числовом значении.

Поэтому мы можем делать это преобразование неявно.

Кастинг вниз более опасен, поскольку мы можем потерять информацию о числовом значении.

При преобразовании double в int мы получаем усеченное целочисленное значение, поэтому это преобразование нужно указывать явно.

Полиморфизм

Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 6

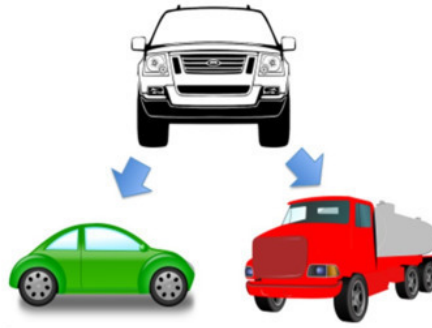
Полиморфизм

В объектно-ориентированном программировании мы организуем объекты в классы.

Объекты в одном классе имеют одинаковые поля, и одни и те же методы.

Можно сказать, что объекты в классе имеют одну и ту же форму, они могут просто отличаться значениями полей в определенном состоянии.

Когда мы ввели наследование, мы ввели семейства связанных классов.



Класс может наследовать поля и методы из базового класса и добавить дополнительные свои поля и методы.

Теперь мы хотим настроить возможности в классах такой иерархии.

Представьте, что мы хотим иметь одни и те же методы в базовом классе и в производном классе, но мы хотим сделать что-то другое в зависимости от класса, к которому принадлежит объект.

```
public class Vehicle{
  private String color;
  public Vehicle(String c) {color=c;}
  public String toString()
    {return "I am a vehicle";}
}
public class Car extends Vehicle{
  private int noPass;
  public Car(int n, String c)
   {super(c); noPass=n;}
  public String toString ()
    {return
   "I am a car. I am carrying "+noPass+" passengers";}
}
```

Здесь мы видим, что в методе toString подкласса car определено другое поведение, отличное от того, которое определено в суперклассе.

Поэтому поведение считается переопределенным.

Этот же метод может делать что-то совершенно отличное от метода суперкласса, с тем же именем и теми же функциональными возможностями.

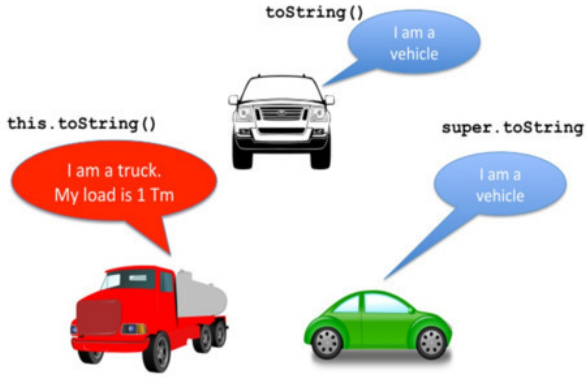
Таким образом, мы видим, что метод с тем же именем и одинаковой функциональностью может иметь разный код в разных классах иерархии.

Это называется переопределением.

Однако при необходимости можно вызвать метод суперкласса.

Для этого нам просто нужно вызвать метод с префиксом

супер.



Здесь также может использоваться ключевое слово this, чтобы обратиться к методу, который определен в соответствующем классе.

Это переопределение методов называется полиморфизмом.

Слово полиморфизм происходит от греческого, что означает многие формы.

И в контексте объектно-ориентированного программирования, полиморфизм позволяет нам иметь методы с одним и тем же именем, и одинаковой функциональностью, но разным поведением в группе классов, связанных отношением наследования.

Другими словами, полиморфизм позволяет использовать наследников, как родителей. При этом, если в классе-наследнике был переопределен какой-то метод, то вызовется он.

Переопределение и перегрузка

Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 7

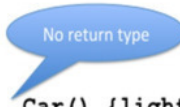
Переопределение и перегрузка

Теперь давайте рассмотрим две концепции, которые выглядят взаимосвязанными, но на самом деле являются разными, это перегрузка и переопределение.

Обе эти концепции применяются к методам.

Ранее мы говорили о конструкторах.

Помните, что у нас был автомобиль с двумя полями, lights и color.



Constructors

```
Car() {lights=false; color="white";}

Car(String c)  {lights=false; color=c;}
Car(boolean b) {lights=b; color="white";}

Car(boolean b, String c)
  {lights=b; color=c;}
```

И мы определили в одном классе не один, а несколько конструкторов.

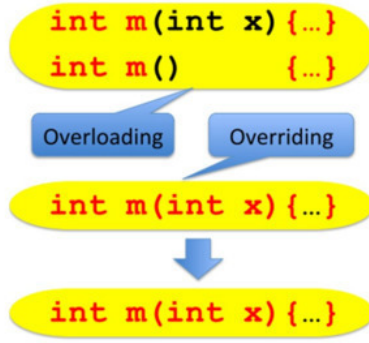
Имена этих конструкторов были одинаковыми, но параметры были разные.

И это важно, чтобы список параметров был другим.

Вы не можете определить два конструктора с одним и тем же именем, и одним и тем же списком параметров.

Фактически, Java понимает, какой конструктор вызвать, просматривая параметры.

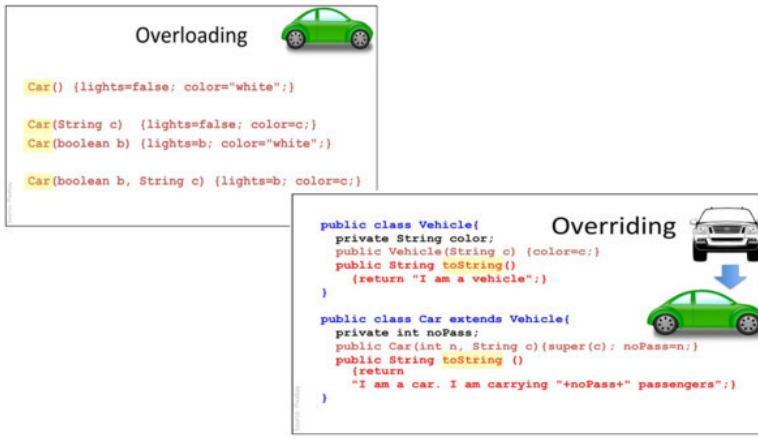
И то, что мы делали для конструкторов, также применимо для методов.



Мы говорим о перегрузке, когда у нас есть разные методы с тем же именем, но разным списком параметров.

С другой стороны, мы ввели переопределение, когда мы хотели изменить поведение метода, унаследованного от суперкласса.

В этом примере метод toString суперкласса переопределяется в подклассе с помощью метода с тем же именем, и теми же параметрами, и возвращаемым типом, но другим телом метода.



Важно, чтобы параметры и возвращаемый тип были одинаковыми.

Отличалось только тело метода.

И в пределах одного класса мы можем перегрузить метод.

В этом случае имя и возвращаемый тип совпадают, но список параметров будет другим.

Компилятор будет различать, какой вызывается метод, сравнивая списки параметров.

Неправильно пытаться перегрузить метод, просто изменив возвращаемый тип.

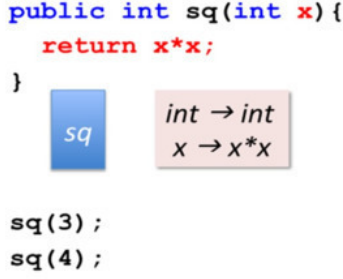
Если мы это сделаем, мы получим ошибку компилятора.

То же самое произойдет, если мы просто изменим имена параметров.

В этом случае определенный метод не изменится вообще.

И мы также получим ошибку компилятора.

Когда мы определяем метод, мы связываем идентификатор – имя метода – с некоторым кодом – телом метода.



Всякий раз, когда мы вызываем это имя метода с некоторыми значениями, мы знаем, какой код нужно выполнить.

Например, используя объявление метода, мы связываем идентификатор sq с методом, который отображает целые числа в целые числа, возводя число в квадрат.

Идентификатор sq всегда привязан к методу в соответствующей области кода.

Во многих языках, которые не являются объектно-ориентированными, эта привязка выполняется обычно во время компиляции.

Во время выполнения эта привязка зафиксирована.

И это называется «ранним» или «статическим» связыванием.

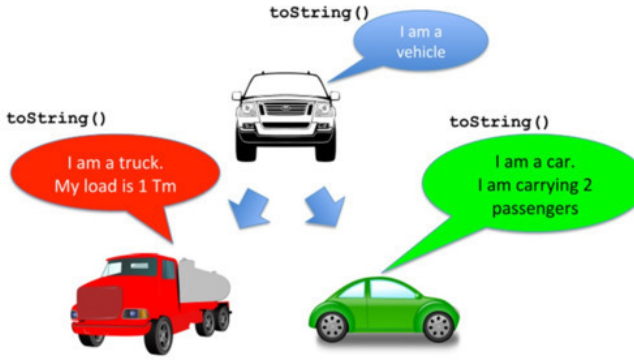
Но этот способ не соответствует концепции полиморфизма и переопределения методов в производных классах.

Здесь мы хотим точно противоположного – чтобы часть кода была не привязана статически к имени метода, а, чтобы зависела от объекта, вызванного во время выполнения.

Поведение, которое нам нужно, называется «динамическим» связыванием.

Поэтому нам нужно различать статическое или раннее связывание, которое выполняется во время компиляции, от динамического или позднего связывания, которое выполняется во время выполнения кода.

В отличие от переопределения перегруженные методы разрешаются во время компиляции.



При этом информация предоставляется классом.

Когда код программы доходит до имени метода, компилятор знает, какое тело метода выполнить – по крайней мере в случае перегруженных методов.

Но это не относится к переопределению.

Здесь разрешение имен выполняется во время выполнения программы.

Динамическое связывание используется для переопределенных методов.

Здесь информация задается объектом, а не классом.

Предположим, мы объявили массив транспортных средств под названием «гараж» для хранения четырех автомобилей.

```
Vehicle[] garage = new Vehicle[4];
garage[0] = new Car(…);
garage[1] = new Car(…);
garage[2] = new Truck(…);
garage[3] = new Truck(…);
```

```
for (int i=0; i<4; i=i+1){
  System.out.println(garage[i].toString());
}
```

И предположим также, что у нас есть автомобили и грузовики, которые стоят в разных позициях.

Теперь в цикле for мы применяем методы toString,

Которые мы определили ранее, ко всем элементам массива.

Что происходит?

Свой метод применяется к каждому из этих элементов.

Таким образом, мы можем иметь единую форму объектов, но разнообразие в том, что выполняется.

Возможно даже, в случае компиляции мы не знаем классов элементов массива.

Это будет считываться во время выполнения программы.

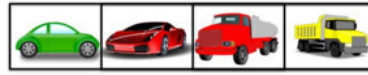
Поэтому динамическое связывание является необходимым поведением для переопределения метода.

Теперь посмотрим на другой пример.

Давайте теперь определим несколько перегруженных методов с именем p.

```
public void p(Vehicle v){
   System.out.println("I am a vehicle");
}
public void p(Car c){
   System.out.println("I am a car");
}
public void p(Truck t){
   System.out.println("I am a truck");
}
```

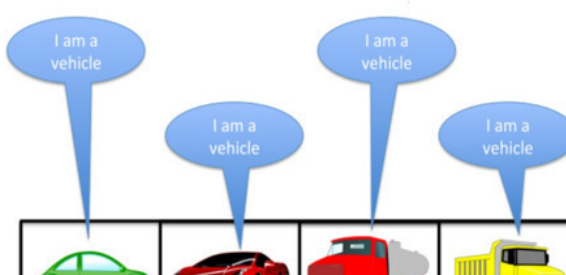
```
for (int i=0; i<4; i=i+1){
   p(garage[i]);
}
```



У них есть один параметр, который является объектом разных классов.

И теперь мы вызываем метод p для всех элементов этого массива.

Помните, что аргумент метода p – это vehicle в массиве vehicle.



Поскольку каждый элемент является vehicle, строка будет напечатана для vehicle, так как метод p привязывается к телу во время компиляции.

Помимо примера, который мы видели, private, final, и static методы также привязываются статически.

Кроме того, атрибуты всегда привязываются статически.

Возникает вопрос, почему все не привязывать динамически?

Имеет смысл связывать идентификаторы с данными или кодом во время компиляции по двум причинам.

Во-первых, чтобы выполнить первую проверку кода и выявить ошибки, а во-вторых, оптимизировать генерируемый код.

Вот почему эта стратегия используется чаще в языках

программирования.

Однако это не работает, когда мы переопределяем метод.

Во время компиляции мы можем даже не знать, какой объект мы получим.

Тогда имеет смысл применить динамическое связывание.

Динамическое связывание также называется «поздним связыванием».

Первое приближение к классу выполняется во время компиляции, но нужный класс окончательно определяется во время выполнения.

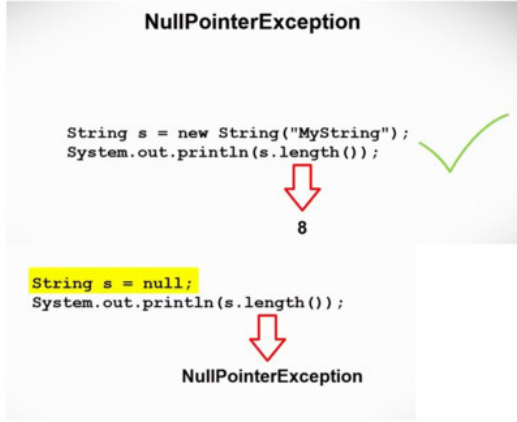
Теперь вернемся к исключениям, чтобы объяснить некоторые дополнительные исключения, которые вы должны знать и которые связаны с объектами и классами.

Небольшое напоминание, исключения – это события, которые происходят во время выполнения программы и которые нарушают нормальный поток выполнения инструкций программы.

Мы уже видели три исключения: ArithmeticException, ArrayIndexOutOfBoundsException
и NumberFormatException.

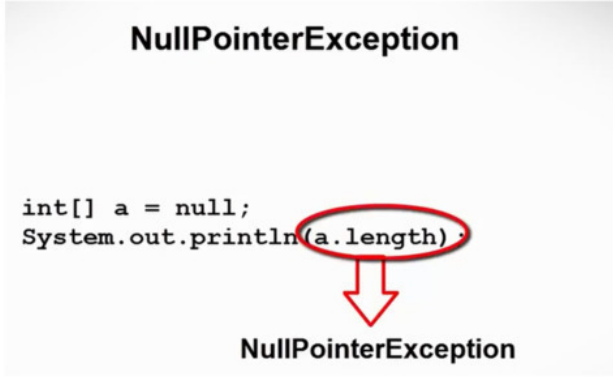
Следующее исключение, которое мы увидим, – это исключение NullPointerException.

Это исключение возникает при попытке программы использовать переменную, которая не имеет примитивного типа, и которая еще не была инициализирована.



Т. е. мы пытаемся использовать переменную, которая должна указывать на объект, который еще не был создан.

Представьте, что мы хотим напечатать длину массива, который мы объявили, но который мы еще не инициализировали.

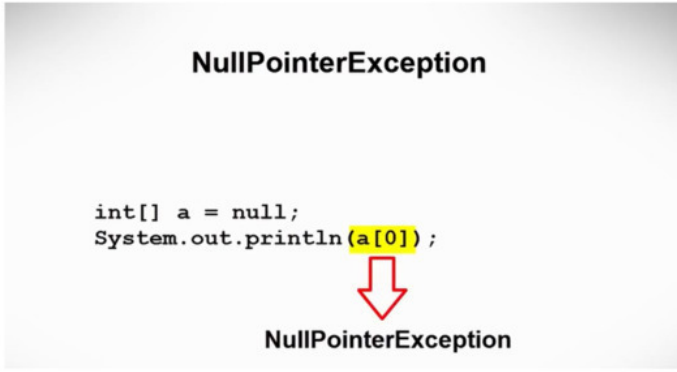


Тогда мы получим такое же исключение NullPointerException.

Имейте в виду, что «length» – это метод в случае класса String, но поле в случае массива.

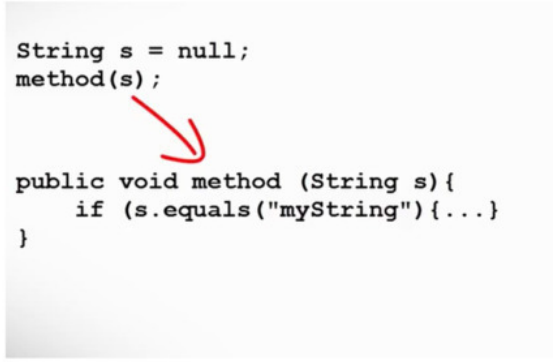
И, если мы попытаемся получить доступ к позиции в массиве, который не был инициализирован, программа будет генерировать исключение NullPointerException, а не исключение ArrayIndexOutOfBoundsException.

В этих примерах очень легко обнаружить, что мы пытаемся использовать переменную, которая не была инициализирована.



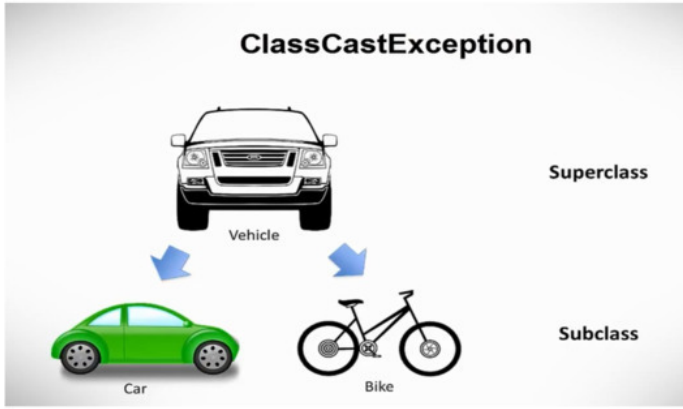
Однако, когда мы программируем, мы определяем методы, которые получают аргументы и которые вызываются из других объектов.

В этих случаях обнаружение переменных, которые не были инициализированы, не так очевидно.



Второе исключение, которое связано с объектами и классами, и которое мы увидим, это ClassCastException.

Чтобы проиллюстрировать это исключение, рассмотрим эту иерархию классов, где Vehicle является суперклассом, и Car и Bike – это подклассы.



Согласно этой иерархии, можно создать экземпляр класса Car и присвоить его переменной типа Vehicle, потому что Car также является Vehicle.

Это приведение правильное и позволяет нам воспользоваться свойством полиморфизма, сохраняя в одном массиве Vehicle набор объектов классов Car и Bike.

```
Vehicle v = new Car();
```

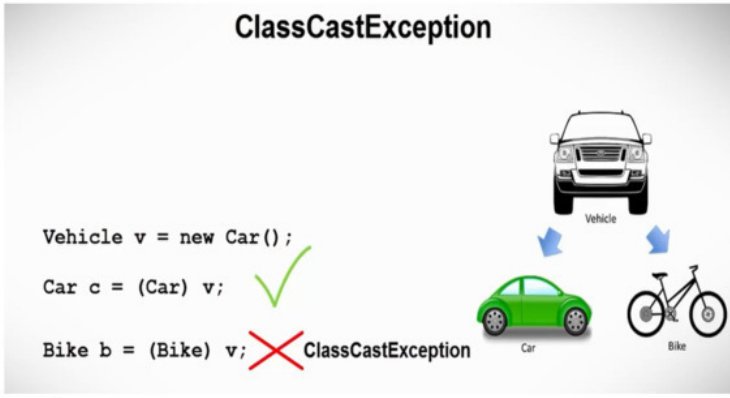
```
Vehicle[] arrayV = {new Car(), new Car(), new Bike()};
```

```
Vehicle v = new Car();
Car c = (Car) v;
```

Позже в программе нам может понадобиться привести этот экземпляр к объекту класса Car.

Единственное условие, которое налагает Java, – это сделать это приведение явным.

Однако, если мы попытаемся применить этот экземпляр к объекту класса Bike, программа выбросит ClassCastException во время выполнения, потому что объект в переменной «v» не является байком.



Мы уже видели, как обрабатывать исключения, которые выбрасываются, когда в программах происходят определенные события, используя конструкцию «try-catch».

Однако мы также можем программировать методы, которые при определенных обстоятельствах должны выбрасывать исключения.

Чтобы явно выбросить исключение в методе, нам нужно использовать ключевое слово «throw» и создать экземпляр конкретного исключения, которое метод должен выбросить.

```
public void method (String s){
    if (s == null){
        throw new IllegalStateException("s is null");
    }
    ...
}
```

Один и тот же метод может выбросить несколько исключений в зависимости от конкретных обстоятельств.

Примитивы и объекты

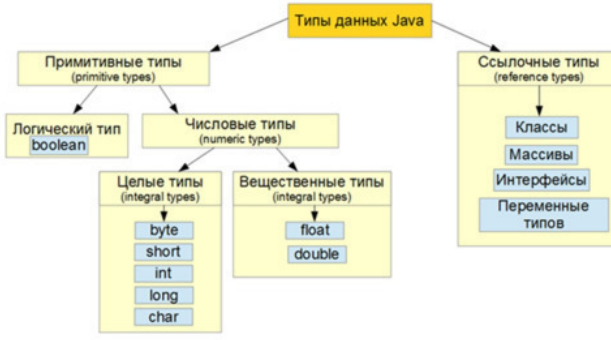
Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 8

Примитивы и объекты

Теперь в качестве обобщения.



В Java есть два общих типа данных: примитивы и объекты.

Примитив – это тип данных Java, которые считаются простейшей формой данных.

Данные этого типа хранятся непосредственно в памяти.

Это данные типа int, char, double и boolean.

И когда вы создаете новую переменную типа int, которая является примитивом, компьютер выделяет область в памяти с именем и значением этого int прямо там.

Поэтому всякий раз, когда вы передаете переменную в качестве параметра или копируете ее, вы копируете значение этой переменной.

Поэтому вы создаете совершенно новую версию этой переменной каждый раз, когда вы манипулируете ей.

Так как примитивы такие простые, мы можем выполнять

с ними прямые математические операции, такие как сложение, вычитание, деление, и так далее.

Теперь, что такое объект?

Объектом является гораздо более сложный тип данных, потому что на самом деле это способ хранения нескольких фрагментов связанной информации и различных вещей, которые вы можете делать с этой информацией под одним типом данных.

Такие вещи, как String, Array, Scanner и ArrayList считаются объектами.

И все они начинаются с большой буквы в Java, чтобы обозначить их как объекты.

Когда вы создаете новую переменную типа объект, например, для массива, компьютер выделяет область памяти для ссылки на то, где этот код на самом деле собирается хранить эти данные.

Затем, когда вы передаете это значение в качестве параметра, вы передаете ссылку, а не фактические данные.

И это потому, что объекты намного больше примитивов, и постоянно копировать их очень затратно.

Поэтому вам всегда нужно понимать, когда вы копируете ссылку на объект или сами данные объекта.

Поскольку объекты сложнее примитивов, вы не можете выполнять такие вещи, как сложение и вычитание, как с простыми числами.

Но, поскольку объекты имеют свое поведение, вам просто

нужно взглянуть на методы объекта, чтобы узнать, что вы можете с этим объектом сделать.

Например, если вы хотите узнать, сколько символов в строке, вы вызываете метод length.

Каждый объект имеет свой собственный набор моделей поведения.

И есть одна вещь, о которой нужно знать.

Это специальное ключевое слово null.

Null – это просто слово, которое означает отсутствие объекта.

По сути, это значение 0 для объекта.

Точно так же, как 0 – это значение 0 для int или 0.0 – это значение 0 для double.

Null – это значение 0 для всех типов объектов.

Предположим, мы создаем новый массив строк.

Если мы создадим новый массив символов, мы знаем, что он хранит значения нулей по умолчанию.

Но что он хранит в случае, когда мы создаем массив строк?

Это Null.

Это то, что автоматически заполняется в массив, что означает, объект может быть здесь, но его нет здесь и сейчас.

Это важно знать, потому что вы можете столкнуться с очень распространенным типом исключения Null Pointer.

Обычно это происходит, когда вы пытаетесь выполнить метод объекта, который является нулевым.

Например, мы хотим получить длину строки, которая хранится в этом массиве.

Там нет строки, поэтому мы получаем так называемое исключение Null Pointer.

Вы не можете назвать длину того, чего не существует.

Имейте в виду, что null означает объект, а не пустой объект.

Например, вы можете вызвать метод length для пустой String.

Это длина равна нулю.

Но нет такой длины, как длина того, чего не существует.

Просто важно знать, что null означает, что здесь нет объекта.

И нам нужно туда его поместить.

Теперь, когда мы понимаем, что такое примитив и что такое объект, важно понять, как компьютер рассматривает эти два типа переменных в своей собственной памяти.

Потому что это оказывает огромное влияние на то, как вы их программируете.

```
int x = 10;
```

	0	1	2	3
0				
1				
2				
3				

Представим себе, что это память компьютера.

На самом деле это похоже на то, как выглядит память компьютера.

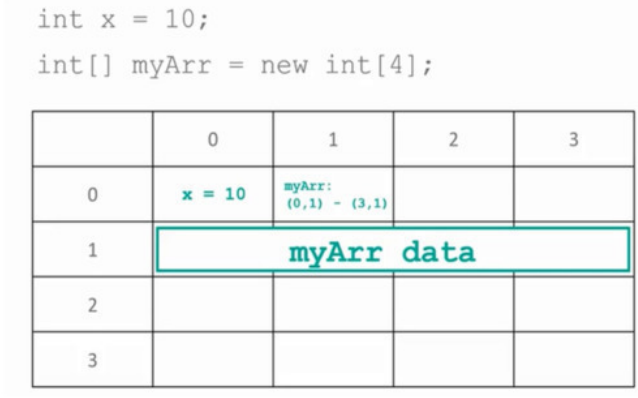
Это общая сетка с адресами для каждого отдельного местоположения, очень похожая на массив.

Когда вы создаете новую переменную примитивного типа, компьютер занимает одно место в памяти и просто помещает эту информацию прямо там, имя и значение переменной.

Когда вы создаете новый объект, он является динамическим.

Он может расти и сокращаться, он может быть большим.

Поэтому компьютер должен придумать особый способ поддерживать этот объект в собственной памяти.



Поэтому, при создании, например, нового массива, компьютер сначала находит место в своей памяти для хранения адреса, где он будет хранить этот массив.

И затем он занимает целую секцию памяти для какого-либо большого объекта.

И, теперь у нас есть массив, находящийся в памяти, где одна из ячеек хранит адрес, где находятся реальные данные.

Это и есть ссылка.

Таким образом существует большое различие между примитивами и объектами.

Примитивы хранятся непосредственно в памяти, как только вы создаете примитив.

Они настолько малы, что это имеет смысл.

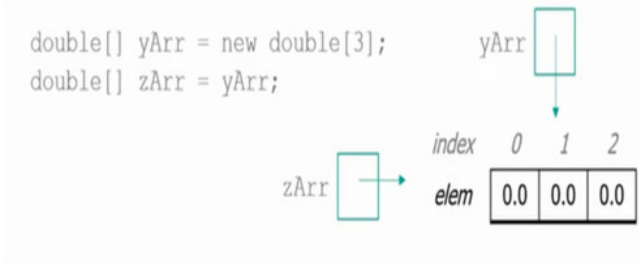
Когда вы копируете переменную примитивного типа и меняете ее значение, первоначальное значение никак не меняется.

Между ними нет реальной связи.

Это будут две совершенно разные переменные.

Как вы можете себе представить, объекты функционируют по-другому.

Вместо того, чтобы выделять пространство для фактического значения, объекты занимают пространство в памяти для ссылки на место, где хранится информация объекта.



Поэтому, если я создаю новый массив, а затем создаю другой массив, и устанавливаю его равным первому массиву, что копируется?

Компьютер копирует ссылку.

Теперь у меня есть две переменные, которые указывают на одну и ту же информацию.

Поэтому, если я что-то изменяю в массиве z, изменится и массив y, и наоборот.

Вы просто скопировали адрес, где находится информация.

Поэтому, если я создам объект и передам его как параметр в метод, я передам ссылку или адрес.

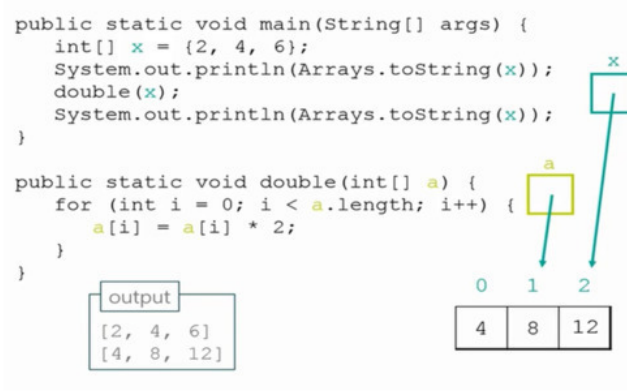
И любые изменения, которые я сделаю в этом методе с объектом, будут отражены в первоначальном объекте.

Мне даже не нужно возвращать его в методе.

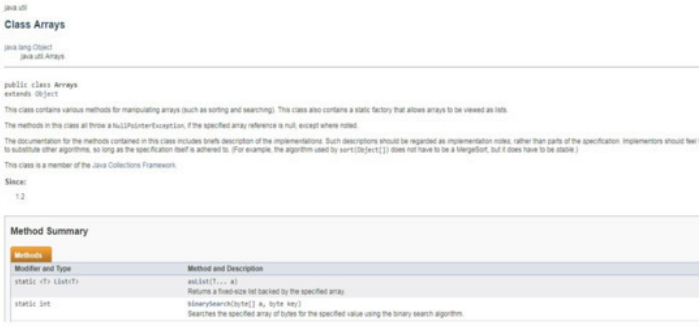
Как было сказано ранее, массивы – это объекты. Однако, у них нет полезных методов внутри объекта Array.

Для этого в Java есть класс Arrays,

который содержит набор статических вспомогательных методов для работы с числами, схожих с тем, как в классе Math есть набор статических вспомогательных методов для работы с числами.



Вот несколько популярных методов из класса Arrays.



java.util

Class Arrays

java.lang.Object
  java.util.Arrays

public class **Arrays**
extends Object

This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

The methods in this class all throw a `NullPointerException`, if the specified array reference is null, except where noted.

The documentation for the methods contained in this class includes briefs description of the implementations. Such descriptions should be regarded as *implementation notes*, rather than parts of the specification. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by `sort(Object[])` does not have to be a MergeSort, but it does have to be stable.)

This class is a member of the Java Collections Framework.

Since:

1.2

Method Summary

Methods

Modifier and Type	Method and Description
static <T> List<T>	asList(T... a) Returns a fixed-size list backed by the specified array.
static int	binarySearch(byte[] a, byte key) Searches the specified array of bytes for the specified value using the binary search algorithm.

Метод toString возвращает строковое представление массива.

Метод equals определяет, одинаковы ли два массива.

Метод fill присваивает новое значение всем элементам массива.

Метод sort сортирует элементы.

Метод binarySearch выполняет поиск элемента по значению и возвращает индекс элемента в случае успеха, или отрицательное целое в случае, если такого элемента нет.

Для работы метода binarySearch необходимо, чтобы массив был уже отсортирован.

Method name	Description
`toString(arr)`	returns a string representing the array inside [], e.g. `"[7, 33, 51, 14]"`
`equals(arr1, arr2)`	returns `true` if the two arrays are equal, that is they contain the same elements in the same order
`fill(arr, val)`	sets every element in the array to have the specified value
`sort(arr)`	sorts the elements in the array into ascending order
`binarySearch(arr, val)`	returns the index of the given value in an array (< 0 if not found); the array must be sorted

```
import java.util.*;
```

Класс Arrays находится в пакете java. util, и если вы хотите его использовать, вы должны добавить строку import java. util.* в начало Java файла.

Давайте рассмотрим пример использования пары методов из класса Arrays.

```
//Return the median value of an array of numbers
//without changing the parameter array
public static double median(int[] numbers) {
  int[] tmp = Arrays.copyOf(numbers, numbers.length);
  Arrays.sort(tmp);
  int mid = tmp.length/2;   // Note: int division
  if  (tmp.length%2 == 0) { // even length?
    return (tmp[mid-1]+tmp[mid])/2.0; //float division
  } else {
    return tmp[mid];
  }
}
```

В этой задаче мы хотим вернуть медианное значение для множества чисел, где медиана – это среднее значение, когда числа отсортированы.

Для решения задачи, сначала мы создадим копию массива, т.о. мы не изменим оригинальный массив.

После создания копии, мы отсортируем массив. Потом мы просто сможем получить медианное значение, которое представляет собой просто средний элемент массива нечетной длины, или арифметическое среднее двух средних элементов массива четной длины.

Вот наш метод median, который принимает массив целых чисел в качестве аргумента, и возвращает значение типа double.

Мы возвращаем тип double, т.к. у нас может быть усред-

нение двух целых чисел.

Метод начинается с создания копии массива-аргумента вызовом метода copyOf класса Arrays.

Этот метод создаст копию массива с количеством элементов, которое указанно вторым аргументом.

В данном случае, мы создаем полную копию массива numbers.

После того, как копия сделана, мы сортируем ее, вызывая метод Arrays.sort.

Мы находим средний элемент массива, используя целочисленное деление, и затем определяем, четная ли длина у массива или нечетная.

Если длина четная, мы возвращаем среднее значение двух центральных элементов.

В этом случае, мы делим на 2.0, чтобы получить число с плавающей запятой.

Если длина нечетная, мы просто возвращаем центральный элемент отсортированного массива.

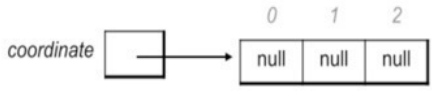
В заключение, давайте коротко обсудим массивы объектов.

Как упоминалось ранее, когда массив создан, его элементы инициализируются нулем 0 такого же типа, что и базовый тип массива.

Для массивов объектных типов, значение при инициализации – это специальное значение null.

Значение null просто означает, что там не пока объекта.

```
Point[] coordinate = new Point[3];
```



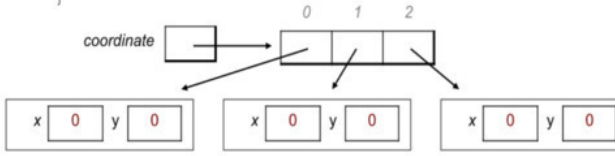
Например, если мы создадим массив coordinate, это массив трех элементов типа Point.

Все три элемента будут проинициализированы значением null.

Перед тем, как пользоваться этим массивом, нам нужно заменить все значения null реальными объектами Point.

Т.о. массивы объектного типа требуют инициализации в два этапа.

```
Point[] coordinate = new Point[3];          // step 1
for (int i=0; i<coordinate.length; i++) {
   coordinate[i] = new Point(0, 0);         // step 2
}
```



На первом тапе, вы создаете объект массива, а на втором этапе, вы создаете объект базового типа для каждого элемента массива.

В образце кода, первым шагом является создание массива coordinate.

Затем, мы выполняем второй шаг с помощью цикла, в котором создается реальный объект класса Point для элемента 0, 1 и 2.

Массивы – полезный инструмент.

Однако они имеют некоторые ограничения.

Когда вы сначала создаете массив, вам нужно выбрать его размер.

И как только вы выберете размер массива, его нельзя изменить.

Это усложняет ситуацию, если у вас есть динамический набор информации, входящий и выходящий из вашей структуры данных.

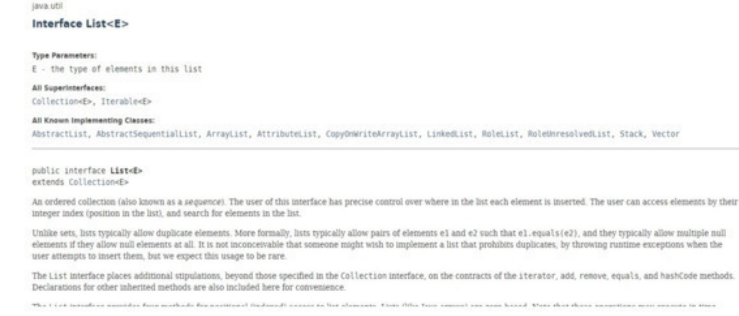
Что, если вы не знаете, сколько всего будет элементов в конце концов?

Кроме того, если вы захотите, скажем, вставить что-то в середину массива, вы должны освободить место для этого.

Это означает, что вы должны сдвинуть все остальные элементы дальше по массиву.

Было бы неплохо, если бы существовала структура данных, которая обеспечивала бы легкий доступ и организацию массива, но при этом предоставляла бы всю гибкость, которая вам нужна.

Такая структура данных в Java есть и это список.



java.util

Interface List<E>

Type Parameters:
`E` - the type of elements in this list

All SuperInterfaces:
`Collection<E>, Iterable<E>`

All Known Implementing Classes:
`AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector`

`public interface List<E>`
`extends Collection<E>`

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements e1 and e2 such that `e1.equals(e2)`, and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

The `List` interface places additional stipulations, beyond those specified in the `Collection` interface, on the contracts of the `iterator`, `add`, `renove`, `equals`, and `hashCode` methods. Declarations for other inherited methods are also included here for convenience.

Список представляет собой упорядоченную последовательность элементов, как и массив.

При этом, он добавляет функциональность, позволяющую ему расти и уменьшаться и вставлять элемент в середину без необходимости делать какие-либо изменения.

Также вы можете удалить элемент внутри списка.

Первый тип списка, так как в Java существует много типов списков, это ArrayList.

ArrayList хранит информацию в массиве, но при этом предоставляет дополнительную функциональность списка.

Вот несколько сравнений использования ArrayList и простого массива.

java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:
AttributeList, RoleList, RoleUnresolvedList

---

public class **ArrayList<E>**
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added

```
String[] names = new String[5];
ArrayList<String> names = new ArrayList<String>();
```

```
names[0] = "Jessica";
names.add("Jessica");
```

```
String s = names[0];
String s = names.get(0);
```

С массивом вы начнете с типа и затем набор скобок, а затем его размер.

С ArrayList, вам просто нужно знать, какой тип информации вы собираетесь хранить в нем, а затем вы создаете новый ArrayList.

И он будет расти и сокращаться по мере необходимости.

Не нужно передавать его длину.

Чтобы добавить значение в массив вы должны найти в нем место и добавить в это место значение.

В ArrayList вы можете просто сказать add и затем добавить все, что захотите, в ArrayList.

Он сам знает, где находится свободное пространство.

Вы также можете получить элемент, как и массив, используя индекс.

ArrayList поддерживает индексы для каждого из элементов, как и массив.

В ArrayList вы должны передать тип информации, которую он собирается хранить, в качестве параметра.

И это отлично подходит для объектов.

Но как насчет примитивов?

К сожалению, вы не можете просто создать ArrayList из, например, int.

Поэтому вам нужно использовать так называемый класс-оболочку, который является простым классом, хранящим только int внутри него.

Это класс Integer.

То же самое существует для double и char.

ArrayList поставляется с огромным набором методов, чтобы сделать жизнь проще.

```
ArrayList<DataType> name = new ArrayList<DataType>();
```

int => Integer	char => Character
double => Double	boolean => Boolean

add(**value**)	myList.add("hello")
add(**index, value**)	myList.add(0, "hello")
clear()	myList.clear()
indexOf(**value**)	myList.indexOf("hello")
get(**index**)	myList.get(0)
remove(**index**)	myList.remove(0)
set(**index, value**)	myList.set(0, "goodbye")
size()	myList.size()
toString()	myList.toString()

Вы не только можете добавить элемент в самом конце, но вы также можете добавить элемент по определенному ин-

дексу.

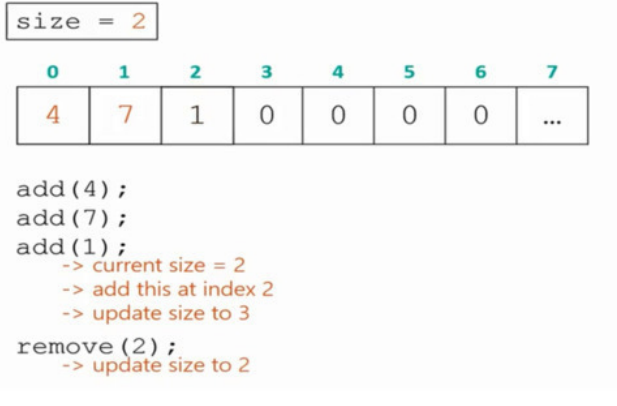
Вы можете очистить массив, вы можете выполнить поиск по массиву.

Например, вы ищете конкретное слово, но вы не знаете, в каком индексе оно находится.

Это метод indexOf.

Вы также можете удалить и установить определенный индекс.

По сути, ArrayList это массив внутри класса, который имеет большой размер 2^32—1, так что вы не сможете использовать всю длину массива.



ArrayList имеет переменную размера, которую он всегда поддерживает.

Вы добавляете элемент в массив и удаляете, при этом изменяется переменная размера.

Абстракция

Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 9

Абстракция

Абстракция в объектно-ориентированном программировании помогает скрыть сложные детали объекта.

Абстракция является одним из ключевых принципов OOAD (объектно-ориентированный анализ и дизайн).

И абстракция достигается композицией (разделением) и агрегацией (объединением).

Например, автомобиль оснащен двигателем, колесами и многими другими деталями.

```
public class Car {

  int price;
  String name;
  String color;

  int engineCapacity;
  int engineHorsePower;

  String wheelName;
  int wheelPrice;

  void move(){
  //move forward
  }
  void rotate(){
   //Wheels method
  }
  void internalCombustion(){
   //Engine Method
  }

}
```

И мы можем записать все свойства автомобиля, двигателя и колеса в одном классе.

В этом примере атрибуты колеса и двигателя добавляются к типу автомобиля.

При программировании это не вызовет каких-либо проблем.

Но когда дело доходит до поддержки приложения, это становится более сложным.

Теперь, используя абстракцию, мы можем отделить вещи, которые можно сгруппировать в другом типе.

Часто изменяющиеся свойства и методы можно сгруппировать в отдельный тип, чтобы основной тип не нуждался в изменении.

Это добавляет силы принципу OOAD – «Код должен быть

открыт для расширения, но закрыт для модификации».

И это упрощает представление модели.

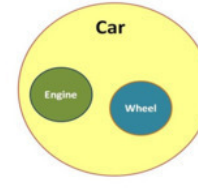
Применяя абстракцию с композицией (разделением) и агрегацией (объединением), приведенный пример может быть изменен следующим образом.

Вы можете видеть, что атрибуты и методы, связанные с двигателем и колесом, перемещаются в соответствующие классы.

```
public class Car {

   Engine engine = new Engine();
   Wheel wheel = new Wheel();

   int price;
   String name;
   String color;

   void move(){
   //move forward
   }
}
```

```
public class Engine {
  int engineCapacity;
  int engineHorsePower;

  void internalCombustion(){
    //Engine Method
  }
}
```

```
public class Wheel {
  String wheelName;
  int wheelPrice;

  void rotate(){
    //Wheels method
  }
}
```



Двигатель и колесо относятся к типу автомобиля.

Когда создается экземпляр автомобиля, оба – двигатель и колесо будут доступны для автомобиля, а когда будут изменения этих типов (двигателя и колеса), изменения будут ограничиваться только этими классами и не будут влиять

на класс Car.

Абстракция известна как отношение Has-A, например, у студента есть имя, у ученика есть ручка, у машины есть двигатель, то есть у каждого есть отношение Has-A.

И используя это отношение, мы разделяем на части, а затем одна часть может использовать другие части в виде объектов.

Абстракция является одним из основополагающих принципов языков объектно-ориентированного программирования.

И абстракция помогает снизить сложность, а также улучшает поддерживаемость системы.

В сочетании с концепциями инкапсуляции и полиморфизма абстракция дает больше возможностей объектно-ориентированным языкам программирования.

Абстракция – это принцип обобщения.

Абстракция принимает множество конкретных экземпляров объектов и извлекает их общую информацию и функции для создания единой обобщенной концепции, которая может использоваться для описания всех конкретных экземпляров как одно.

При этом мы переходим от конкретного экземпляра к более обобщенному понятию, думая о самой базовой информации и функции объекта.

Тем самым абстракция помогает скрыть сложные детали объекта.

Например, когда вы используете электронную почту, сложные детали того, что происходит, когда вы отправляете электронное письмо, например, протокол, используемый вашим почтовым сервером, скрыт от пользователя.

Поэтому, чтобы отправить электронное письмо, вам просто нужно ввести текст, указать адрес получателя и нажать «Отправить».

Аналогично в объектно-ориентированном программировании абстракция – это процесс скрытия деталей реализации от пользователя, и пользователю предоставляется только функциональность.

Другими словами, пользователь будет иметь информацию о том, что делает объект, а не о том, как он это делает.

И в Java это свойство абстракции реализуется с использованием абстрактных классов и интерфейсов, которые мы рассмотрим на следующей лекции.

Интерфейсы. Абстрактные методы и классы

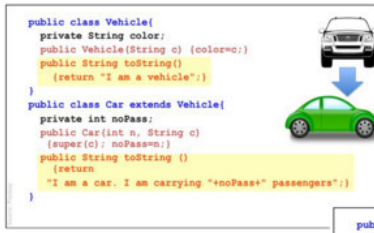
Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования
Лекция 10
Интерфейсы. Абстрактные методы и классы

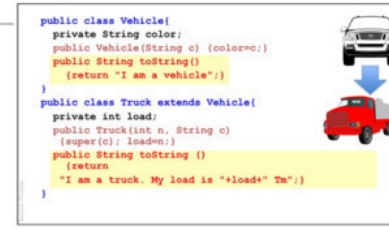
Ранее мы определяли метод в одном классе и переопределяли этот метод в производных классах.

Таким способом можно делать разные вещи в зависимости от класса.

Здесь мы видим разные строки, возвращаемые методом toString.



```
public class Vehicle{
  private String color;
  public Vehicle(String c) {color=c;}
  public String toString()
    {return "I am a vehicle";}
}
public class Car extends Vehicle{
  private int noPass;
  public Car(int n, String c)
   {super(c); noPass=n;}
  public String toString ()
    {return
   "I am a car. I am carrying "+noPass+" passengers";}
}
```



```
public class Vehicle{
  private String color;
  public Vehicle(String c) {color=c;}
  public String toString()
    {return "I am a vehicle";}
}
public class Truck extends Vehicle{
  private int load;
  public Truck(int n, String c)
   {super(c); load=n;}
  public String toString ()
    {return
   "I am a truck. My load is "+load+" Tm";}
}
```

Так как класс Vehicle является общим для этих классов, нам может вообще не понадобится строка, которая возвращается его методом toString.

В этом случае мы можем вообще не определять тело для метода toString в классе Vehicle.

Мы можем это сделать, и метод без тела называется «абстрактным».

```
public abstract class Vehicle{
  private String color;
  public Vehicle(String c) {color=c;}
  public abstract String toString();
  … // other methods
}
public class Car extends Vehicle{
  private int noPass;
  public Car(int n, String c)
   {super(c); noPass=n;}
  public String toString ()
    {return
   "I am a car. I am carrying "+noPass+" passengers";}
}
```

Абстрактные методы обозначаются с помощью ключевого слова «абстрактный» в определении.

И класс с хотя бы одним абстрактным методом называется «абстрактным классом».

Здесь мы видим ключевое слово абстрактный в определении абстрактного класса.

Таким образом, абстрактный метод – это метод без тела.

Конструкторы, статические методы и финальные методы не могут быть абстрактными.

Абстрактный класс – это класс, в котором некоторые методы абстрактные, а некоторые – нет.

Абстрактный класс – это незавершенный класс, и мы не можем создать его объекты.

Чтобы получить объект, мы сначала должны определить

производный класс, где тела определены для всех абстрактных методов.

Абстрактный класс может быть расширен до класса, или до другого абстрактного класса.

Кроме того, вы можете определить класс как абстрактный даже без абстрактного метода.

Это допускается, и вы можете это сделать для предотвращения возможности создания экземпляра класса.

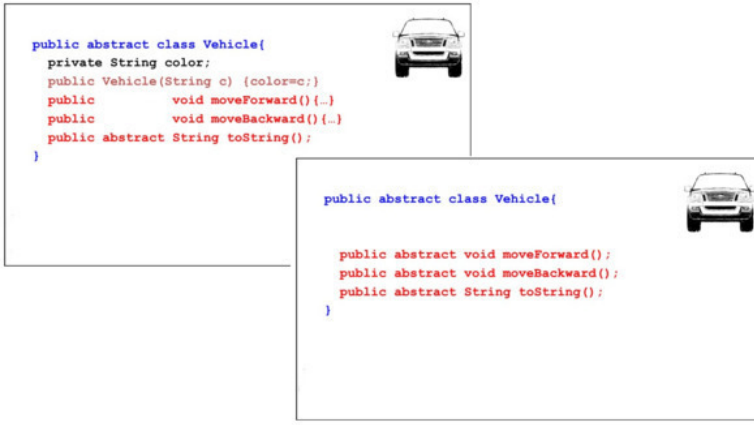
Однако, если есть абстрактный метод, вы получите ошибку, если вы не добавите ключевое слово «abstract» в определение класса.

Таким образом, абстрактные методы помогают разделить определение метода от поведения метода.

А абстрактные классы – это незавершенные классы, которые содержат абстрактные методы.

В абстрактном классе могут быть и абстрактные методы, а также обычные методы.

Теперь вопрос в том, что, если мы сделаем все методы абстрактными.



Но класс со всеми абстрактными методами уже не является абстрактным классом.

Это нечто другое.

Если все методы абстрактны, мы называем это интерфейсом.

Обратите внимание, что во всех методах нет тел.

```
public interface VehicleIF {

  public          void moveForward();
  public          void moveBackward();
  public          String toString();
}
```

Здесь мы не пишем ключевое слово abstract для методов, но здесь нет путаницы, поскольку все методы в интерфейсе абстрактные.

Кроме того, мы объявляем методы публичными, но нам не нужно писать это явно.

```
public interface VehicleIF {


  void moveForward();
  void moveBackward();
  String toString();
}
```

Таким образом, все методы автоматически объявляются публичными, даже если мы не укажем ключевое слово public.

На самом деле не совсем верно, что все методы в интерфейсе должны быть абстрактными.

В Java 8 могут быть методы с телом, но они должны быть статическими методами или методами по умолчанию.

Но мы не будем усложнять, чтобы подчеркнуть концепцию интерфейса в его самой чистой форме.

Итак, для вас, в интерфейсе, все методы абстрактны.

Но что насчет полей?

В интерфейсе могут быть поля.

Но все они автоматически статические и финальные.

То есть, они являются константами.

Они также автоматически публичные, поэтому ключевое слово public не требуется явно указывать.

Таким образом, концепция интерфейса – это полезная абстракция.

Тогда как абстрактный класс реализует абстракцию, показывая некоторую общую функциональность для семейства классов без ее конкретной реализации, интерфейс, например, как физический интерфейс в радио или музыкальном проигрывателе, демонстрирует сервис снаружи и скрывает реализацию, которую мы можем определить.

То есть, в случае интерфейса, абстракция скрывает реализацию объекта от пользователя и предоставляет только интерфейс.

На самом деле мы можем изменить эту реализацию без изменения интерфейса, и, следовательно, предоставляемых нами услуг.

Интерфейсы обеспечивают уровень абстракции.

Можно использовать предоставленные методы без знания того, как они реализованы.

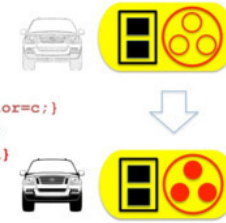
Но в какой-то момент эти методы должны быть реализованы.

Представьте, что у нас есть интерфейс под названием VehicleIF.

```
public interface VehicleIF {
  int SOS_PHONE = 112;

  void moveForward();
  void moveBackward();
  String toString();
}
```

И в этом интерфейсе указан ряд методов.
Помните, что они публичные.

```
public class Vehicle
  implements VehicleIF {
  private String color;
  public Vehicle(String c){color=c;}
  public void moveForward(){…}
  public void moveBackward(){…}
  public String toString(){…}
}
```



Мы могли бы реализовать класс для этого интерфейса и назвать его Vehicle.

Обратите внимание, что теперь используется ключевое слово implements вместо ключевого слова extends.

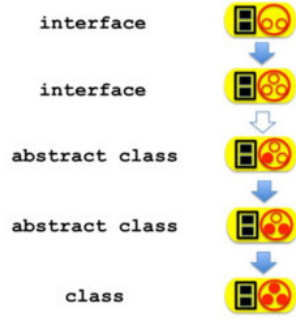
Для реализации интерфейса подразумевается определение класса, где для всех методов, дается реализация.

Мы можем также дополнительно определить поля и конструкторы, а также другие методы, возможно приватные.

Класс следует спецификации, заданной интерфейсом, и добавляет конкретные детали о том, как эти методы могут работать.

Однако нам не нужно идти от спецификации интерфейса к реализации класса за один шаг.

Мы могли бы также действовать поэтапно.



Путь от интерфейса к классу, который может быть создан, может быть короче или длиннее.

Во-первых, мы можем расширить один интерфейс от другого интерфейса, например, путем добавления абстрактных методов.

Мы можем частично реализовать интерфейс, путем реализации некоторых методов.

В этом случае мы получаем в результате не класс, а абстрактный класс, потому что не все методы реализованы.

И, наконец, мы можем перейти непосредственно от интерфейса к классу, путем реализации всех методов.

Если у нас есть абстрактный класс, мы можем расширить его другим абстрактным классом, например, если мы реализуем некоторые абстрактные методы, но не все из них.

Интерфейсы помогают нам моделировать системы, которые позволяют нам повторно использовать не только просто код, но и целиком концепции.

Теперь важно то, что класс может реализовать не только один, но и несколько интерфейсов.

В этом случае класс должен реализовать все методы от всех интерфейсов.

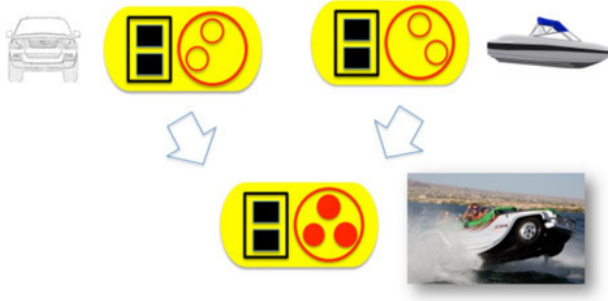
Помните, что класс не может расширять несколько классов.

В Java нет множественного наследования, как в других языках программирования, таких как C ++.

Класс не может расширять два класса.

Однако в Java класс может реализовать два интерфейса.

Это способ сказать, что A является B и C.



Например, мы можем определить амфибию как реализацию интерфейса автомобиля и интерфейса лодки.

Этот амфибия-автомобиль будет реализовывать методы обоих интерфейсов.

Теперь, при этом, могут возникнуть конфликты имен.

Что произойдет, если у нас есть одно и тоже имя метода в обоих интерфейсах?

Если у нас есть одинаковое имя метода в обоих интерфейсах, но разные возвращаемые типы, тогда возникает ошибка.

Если имя метода и тип возвращаемого значения совпадают, тогда все в порядке.

Если, кроме того, совпадают и параметры методов, тогда класс должен реализовать этот метод один раз.

Они неразличимы.

Если типы параметров методов не совпадают, мы имеем случай перегрузки, который обрабатывается таким же образом, как будто эти методы принадлежат одному интерфейсу.

Оба эти методы должны быть реализованы.

Мы можем также определить класс путем реализации интерфейса и наследования от класса одновременно.

Также нужно сказать, что помимо абстрактных классов и интерфейсов для структурирования кода используются классы-утилиты, в которых определены только статические члены.

Как правило, такие классы используются для объедине-

ния родственных алгоритмов.

Примером такого класса может служить класс Math, предоставляющий реализации различных математических функций.

Таким образом, отношения реализации и наследования помогают нам определить структуры связанных интерфейсов и классов, которые помогают нам масштабировать и упорядочить код.

Проектирование интерфейсов всегда было непростой задачей, потому что, если мы хотим добавить дополнительные методы в интерфейсы, это потребует изменений во всех классах реализации.

По мере старения интерфейса количество реализующих его классов может увеличиться настолько, что будет невозможно расширить интерфейс, изменяя все классы реализации.

Вот почему большинство библиотек сначала обеспечивают базовый класс реализации, а затем они расширяют его и переопределяют его методы, чтобы при изменении интерфейса, изменить только базовый класс реализации.

В Java 8 вводится понятие метода по умолчанию интерфейса.

```
public interface Interface1 {

void method1(String str);

default void log(String str){
System.out.println("I1 logging::"+str);

}
}
```

```
public interface Interface2 {

void method2();

default void log(String str){
System.out.println("I2 logging::"+str);

}
}
```

```
public class MyClass implements Interface1, Interface2 {

    @Override
    public void method2() {
    }

    @Override
    public void method1(String str) {
    }

    @Override
    public void log(String str){
        System.out.println("MyClass logging::"+str);
        Interface1.print("abc");
    }
}
```

Для создания метода по умолчанию в интерфейсе нам нужно использовать ключевое слово «default» в сигнатуре метода.

Теперь, когда класс реализует интерфейс, необязательно предоставлять реализацию для методов по умолчанию интерфейса.

Эта функция помогает, помимо подхода с базовым классом реализации, с расширением интерфейсов с помощью дополнительных методов, и все, что нам здесь нужно, это обеспечить реализацию по умолчанию.

Мы знаем, что Java не позволяет нам расширять несколько классов, потому что это приведет к проблеме, когда компилятор не может решить, какой метод суперкласса использовать.

При использовании методов по умолчанию эта же проблема возникает и для интерфейсов.

Так как, если класс реализует интерфейс 1 и интерфейс 2 и не реализует общий метод по умолчанию, компилятор не может решить, какой из них выбрать.

Поэтому, обязательным является обеспечение реализации общих методов интерфейсов по умолчанию.

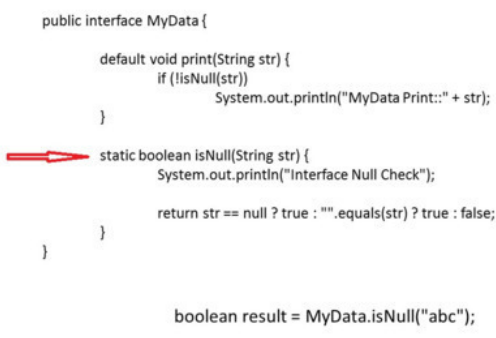
И поэтому, если класс реализует оба вышеупомянутых интерфейса, он должен будет обеспечить реализацию для метода log, иначе компилятор будет выбрасывать ошибку времени компиляции.

Таким образом, методы по умолчанию интерфейса Java помогают расширить интерфейсы, не опасаясь сломать классы реализации.

И методы по умолчанию интерфейса стирают различия между интерфейсами и абстрактными классами.

Если какой-либо класс в иерархии имеет метод с той же сигнатурой, метод по умолчанию теряет смысл для внедрения, чтобы избежать путаницы.

Теперь, статический метод интерфейса похож на метод по умолчанию, за исключением того, что мы не можем переопределить его в классах реализации.



Эта функция помогает избежать нежелательных результатов, связанных с плохой реализацией в классах реализации.

Статический метод интерфейса виден только для методов интерфейса, однако, как и другие статические методы, мы можем использовать статические методы интерфейса, используя его имя.

Статические методы интерфейса хороши для предоставления вспомогательных методов, не требующих реализации в классах.

И можно использовать статические методы интерфейса Java для удаления классов-утилит, и переместить все статические методы классов-утилит в соответствующий интерфейс, который будет легко найти и использовать.

Однако мы не можем определить статический метод ин-

терфейса для методов класса Object, мы получим ошибку компилятора.

Это связано с тем, что Object является базовым классом для всех классов.

В Java 9 вводятся приватные методы и приватные статические методы в интерфейсах.

```
public interface MyInterface {

    default void defaultMethod 1() {
      privateMethod("Hello from the default method 1!");
    }

     default void defaultMethod 2() {
      privateMethod("Hello from the default method 2!");
    }


    private void privateMethod(final String string) {
        System.out.println(string);
    }

    void normalMethod();
}
```

```
interface Example {

    static void doJob1(String arg) {
        verifyArg(arg);
        ...
    }

    static void doJob2(String arg) {
        verifyArg(arg);
        ...
    }

    private static void verifyArg(String arg) {
        ...
    }
}
```

Чтобы был выбор, какие выставлять клиентам методы реализации.

Соответственно приватные методы интерфейса предназначены для использования в методах по умолчанию интерфейса, а приватные статические методы интерфейса предназначены для использования в статических методах интерфейса.

И приватные методы должны иметь реализацию, они не могут быть абстрактными.

И мы не можем получить доступ или наследовать приватные методы от интерфейса к другому интерфейсу или классу.

Пакеты

Объектно-ориентированное программирование на Java

Концепции объектно-ориентированного программирования

Лекция 11

Пакеты

Давайте рассмотрим концепцию пакета, которая позволяет программистам лучше структурировать свои программы, что облегчает их понимание и управление.

В большом приложении классов создается тысячи и десятки тысяч.

Поэтому возникает вопрос: Если классов много, их все в одном каталоге держать? И как потом с ними разбираться?

Конечно же необходим некий механизм упорядочивания.

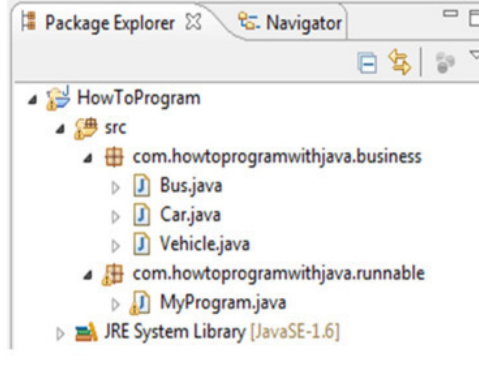
И такой механизм создан.

Причем достаточно простой и очевидный – каталоги.

Мы уже привыкли, что на диске наши файлы лежат в разных каталогах, которые мы сами организовываем в удобном порядке.

В Java сделано тоже самое – физически класс кладется в определенный каталог файловой системы, представляющий собой пакет.

Существует даже некоторые правила именования этих каталогов или пакетов.



Например, для коммерческих проектов каталог должен начинаться сначала с префикса «com» а за ним следует название компании или доменное имя компании – например «mycompany».

Далее следует название проекта.

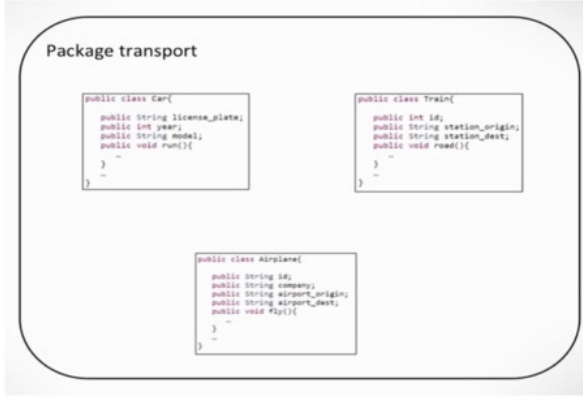
Потом уже идет более точное разделение по какому-либо признаку – чаще всего функциональному.

Пакет в Java представляет собой группу связанных классов и интерфейсов, которые имеют схожие свойства.

Это абстрактная концепция, и это ответственность программиста, чтобы правильно организовать различные классы в пакеты.

Обычно классы, сгруппированные в один и тот же пакет, имеют сходную семантику.

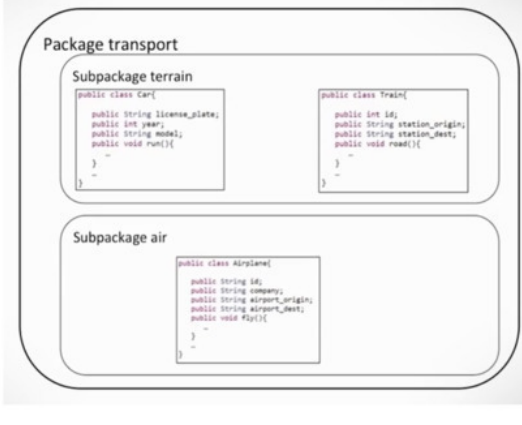
Например, представьте, что у вас есть класс Car.



И он может быть частью пакета с именем transport, который также может содержать другие классы, такие как самолет или поезд.

Пакеты можно подразделить на подпакеты в зависимости от степени абстракции, которую хочет программист, формирующий иерархию пакетов.

Таким образом, наш предыдущий пакет транспорт мог быть разделен на подпакеты наземного транспорта и воздушного транспорта для представления физической среды транспорта.



Подумайте о том, какие классы будут храниться в каждом подпакете.

Когда вы создаете новую программу, очень полезно организовать различные классы и интерфейсы в пакеты, чтобы упростить ваш проект.

Таким образом, вы упрощаете использование классов

и интерфейсов.

Далее, мы рассмотрим стандартную библиотеку Java, которая хорошо структурирована на пакеты и подпакеты.

Хорошо, но когда мы пишем новый класс, как мы можем определить, какой класс принадлежит к какому пакету?

Это очень просто.

В верхней части исходного кода класса вы добавляете слово package, за которым следует имя пакета.

Помните, что определение пакета должно быть первым выражением в исходном файле.

Имя пакета определяется программистом.

Это имя должно быть в нижнем регистре, чтобы избежать конфликтов с именами классов и интерфейсов, и не может быть одним из слов, зарезервированных Java, таким как main, for или string.

И подпакеты задаются с использованием символа точки.

Таким образом, для создания пакета, нам нужно в файловой системе создать каталоги и подкаталоги, например, каталог transport и подкаталог air.

```
package transport.air;

public class Airplane{
. . .
}
```

Затем поместить в них файлы классов и интерфейсов.

И затем указать в каждом классе и интерфейсе вверху директиву package с именами каталогов и подкаталогов, разделенными точками, то есть путем где находится файл класса или интерфейса.

При написании новых программ вам может потребоваться доступ к некоторым классам и интерфейсам из определенного пакета.

В этом случае вы должны заранее импортировать такие классы.

Этот импорт должен быть размещен сверху исходного кода класса, сразу после объявления пакета класса, используя слово импорт.

```
import transport.air.Airplain;

import transport.Car;

import transport.air.*;
```

За оператором импорта должен следовать весь путь пакета, вместе с классом, который вы хотите импортировать.

И это будет полное квалифицированное имя класса – имя класса вместе с именем его пакета.

Если вы хотите импортировать все классы в пакете, вы можете использовать символ звездочки.

Таким образом, вы сможете получить доступ ко всем публичным полям и методам этих классов.

Полное имя класса – весьма важный момент.

Разделение классов по пакетам служит не только для удобства, но решает еще одну важную задачу – уникальность имен классов.

Наверняка в большом проекте будет участвовать много людей и каждый будет писать свои классы.

И наверняка имена этих классов нередко будут одинаковые.

И скорее всего вы будете подключать сторонние библиотеки и в них будут классы, которые будут называться так же как ваши.

Единственным спасением различать их – это поместить в разные пакеты.

Таким образом, как правило, программа состоит из нескольких пакетов.

И каждый пакет имеет собственное пространство имен для классов и интерфейсов, объявленных в пакете.

И пакеты образуют иерархическую структуру имен.

При этом полные имена классов и интерфейсов, то есть их имена с учетом пакетов, должны быть уникальными.

И для доступа из одного пакета к другим пакетам используется ключевое слово import.

Также пакеты могут быть безымянными.

Классы и интерфейсы безымянного пакета не содержат объявления пакета.

И безымянные пакеты следует использовать только в небольших тестовых программах.

Также в Java можно использовать статический импорт для доступа к статическим методам и полям класса.

```
Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2));
```

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

sqrt(Math.pow(side1, 2) + pow(side2, 2));

import static java.lang.Math.*;
```

Например, в этом выражении, методы pow и sqrt являются статическими, поэтому они должны быть вызваны с указанием имени их класса – Math.

И это приводит к достаточно громоздкому коду.

Этих неудобств можно избежать, если воспользоваться статическим импортом.

При этом имена методов sqrt и pow становятся видимыми благодаря оператору статического импорта.

Также, с помощью звездочки, можно импортировать все остальные статические члены класса Math, не указывая их по одному.

Каким бы удобным ни был статический импорт, очень важно не злоупотреблять им, чтобы избежать конфликта имен, например, если вы определите в своем классе свой ме-

тод pow.

Теперь, когда мы узнали, что классы группируются в пакеты, и мы знаем, что методы и поля класса могут быть публичными и приватными, пора сказать, что методы и поля класса также могут быть защищенными, это ключевое слово protected, и приватными в пакете, это отсутствие всякого ключевого слова.

Доступ к членам класса

	Private	Модификатор отсутствует	Protected	Public
Один и тот же класс	Да	Да	Да	Да
Подкласс класса этого же пакета	Нет	Да	Да	Да
Класс этого же пакета, не являющийся подклассом	Нет	Да	Да	Да
Подкласс класса другого пакета	Нет	Нет	Да	Да
Класс другого пакета, не являющийся подклассом класса данного пакета	Нет	Нет	Нет	Да

Публичный член класса виден везде без ограничений.

Защищенный член класса, protected, виден в пределах своего пакета, а также подклассом класса, даже если подкласс принадлежит другому пакету.

Если нет никакого ключевого слова, член класса виден только в пределах своего пакета.

И приватный член класса виден только в пределах своего класса.

Абстрактные классы vs Интерфейсы

Объектно-ориентированное программирование на Java

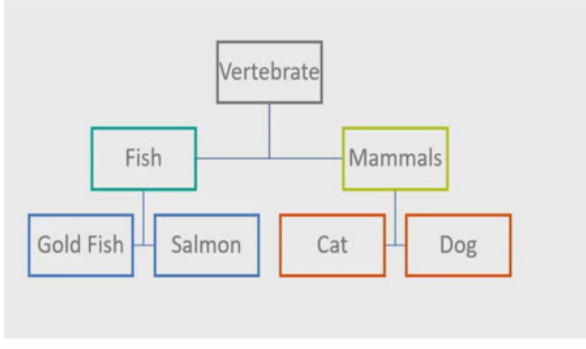
Концепции объектно-ориентированного программирования

Лекция 12

Абстрактные классы vs Интерфейсы

Теперь давайте вернемся немного назад и рассмотрим наследование разных объектов.

Предположим у нас есть эта иерархия.



Вверху иерархии у вас есть что-то более общее, например, позвоночные, а затем, когда вы спускаетесь вниз, все становится более конкретным.

Таким образом, в самом низу у вас есть наиболее специфический уровень.

В некоторых деревьях наследования, поскольку все становится более общим, объекты как бы перестают восприниматься как реальные экземпляры.

Как что такое объект рыбы?

Существует много разных видов рыб, и это слишком общее, чтобы действительно существовало в природе.

Поэтому, для представления таких объектов и вводится понятие абстрактный.

Абстрактные методы – это определение метода в суперк-

лассе, но они не имеют реальной реализации.

Это только заголовок метода.

Что это такое, так это контракт между суперклассом и подклассом.

Как суперкласс, я диктую, что должен реализовывать подкласс.

Чтобы сделать это, вы помещаете ключевое слово abstract и затем заголовок метода с типом возврата и параметрами.

Вам также не нужно устанавливать видимость метода – публичный он или приватный, потому что вы решите это в подклассе.

Как только у вас появятся абстрактные методы в определении класса, значит вы создали абстрактный класс.

Теперь, если у вас есть ключевое слово abstract где угодно, внутри этого класса, вы должны добавить слово abstract в заголовок класса.

Теперь, когда вы используете абстрактный класс и когда вы используете интерфейс?

```
public abstract class Mammal {
   public String furType;
   public String covering() {
      return furtType;
   }
   abstract boolean bornLive();
}
```

В конечном счете, абстрактный класс – это просто обычный суперкласс, к которому вы добавляете некоторое поведение.

Это похоже на добавление интерфейса к существующему суперклассу.

Итак, вы используете абстрактный класс, когда вам нужно совместно использовать код и гарантировать поведение в близко связанных классах.

Так как абстрактный класс объявляет общую функциональность для семейства связанных классов.

Вы также используете абстрактный класс, если у вас есть специфические подклассы, которые должны расширять общие классы.

Вы будете использовать интерфейс, когда вы хотите гаран-

тировать поведение несвязанные классов.

С помощью интерфейсов вы также можете использовать дополнительное наследование, если у вас уже есть расширение класса и вы уже использовали ваше основное наследование.

Конец ознакомительного фрагмента.