**#1. Вводный курс по языку программирования JAVA. Hello World**

**Java** — объектно-ориентированный язык программирования. В нем существуют классы (**class**) и объекты (**object**). Объекты в **Java** представляют собой экземпляры класса.

Например, модель android смартфона **Samsung Galaxy s6** вообще — это класс, а экземпляр Galaxy s6, котрый вы закажете на Ebay и он придет к вам по почте — это конкретный объект, экземпляр класса, и вы можете делать с ним все, что хотите.

Давайте напишем программу **Hello World**, которая просто выводит на экран надпись **«Hello, World!».**

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Main {      public static void main(String[] args) {          System.out.println("Hello, World!");      }  } |
|  |  |

Первая строка объявляет класс под названием **Main**.

|  |  |
| --- | --- |
| **1** | **public class Main {** |

В **Java** каждая строка кода, которая может выполняться должна находиться внутри класса. Эта строка объявляет класс **Main**, модификатор доступа **public** означает что класс общедоступен и любой другой класс может получить доступ к нему. На данный момент это не важно, так что не волнуйтесь. Для начала просто напишем наш код в классе **Main**, а про объекты поговорим позже.

Обратите внимание, что, когда мы объявляем общедоступный класс (**public**), мы должны объявить его в файле с тем же именем (**Main.java**), иначе мы получим ошибку при компиляции.

Следующая строка:

|  |  |
| --- | --- |
| **1** | **public static void main(String[] args) {** |

Это точка входа нашей **Java** программы. Метод **main** должен иметь точно такую же сигнатуру, как показано, иначе программа не будет работать.

* public снова же означает, что метод общедоступен
* static означает что вы можете выполнить этот метод без создания экземпляра класса Main
* void означает, что метод не возвращает никакого значения
* main — имя метода

При помощи этой строки мы выводим на экран «Hello, World!».

|  |  |
| --- | --- |
| **1** | **System.out.println("Hello, World!");** |

Это массив строк. Мы будем использовать его в [следующем](http://www.fandroid.info/tutorial-po-osnovam-yazyka-programmirovaniya-java-dlya-nachinayushhih/) уроке, так что не волнуйтесь, если сейчас вы не все понимаете.

Пока потренируйтесь выводить различный текст, ибо только практика сделает из вас программиста! =)

# #2. Вводный курс по языку программирования JAVA. Переменные и типы

Несмотря на то, что язык **Java** объектно-ориентирован, не все типы — объекты. Существуют так называемые примитивы (**primitives**). Вот список всех примитивов в **Java**:

* byte (число, 1 байт)
* short (число, 2 байта)
* int (число, 4 байта)
* long (число, 8 байтов)
* float (число с плавающей точкой, 4 байта)
* double (число с плавающей точкой, 8 байтов)
* char (символ, 2 байта)
* boolean (true (истина) или false (ложь), 1 байт)

**Java** — строго типизированный язык, это означает, что мы должны объявить переменные , прежде чем будем их использовать.

### Числа

Чтобы объявить и присвоить число используйте следующий синтаксис:

|  |  |
| --- | --- |
| **1**  **2** | **int myNumber;**  **myNumber = 5;** |

**=** это оператор присваивания.

Вы можете объединить эти операции:

|  |  |
| --- | --- |
| **1** | **int myNumber = 5;** |

Чтобы объявить число с плавающей точкой, используйте следующий синтаксис:

|  |  |
| --- | --- |
| **1**  **2** | **double d = 4.5;**  **d = 3.0;** |

Если вы хотите использовать float, то:

|  |  |
| --- | --- |
| **1** | **float f = (float) 4.5;** |

Или:

|  |  |
| --- | --- |
| **1** | **float f = 4.5f** |

(f — более короткий способ объявить float)

### Символы и строки

В Java символ — свой собственный тип, и это не просто число. Синтаксис:

|  |  |
| --- | --- |
| **1** | **char c = 'g';** |

String — не примитив. Это реальный тип. Вот несколько способов использования строки:

Создание строки с помощью конструктора

|  |  |
| --- | --- |
| **1** | **String s1 = new String("Who let the dogs out?");** |

С помощью двойных кавычек (» «).

|  |  |
| --- | --- |
| **1** | **String s2 = "Who who who who!";** |

В Java присутсвует конкатенация (объединение) строк при помощи оператора +.

|  |  |
| --- | --- |
| **1** | **String s3 = s1 + s2;** |

В Java нет перегрузки операторов! Оператор + определен только для строк, вы никогда не увидите его с другими объектами, только с примитивами.

|  |  |
| --- | --- |
| **1**  **2** | **int num = 5;**  **String s = "I have " + num + " cookies";** |

Заметьте, что кавычки с примитивами не используются.

### boolean

Каждый оператор сравнения в Java возвращает булевскую переменную (boolean), которая может принять только два значения: true (истина) или false (ложь).

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8** | **boolean b = false;**  **b = true;**    **boolean toBe = false;**  **b = toBe || !toBe;**  **if (b) {**  **System.out.println(toBe);**  **}** |

Оператор || - это логическое «или».

А например, такой код не будет работать по причине несовместимости типов:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **int children = 0;**  **b = children;  // Не будет работать, требуется boolean, а найден int**  **if (children) {  // Не будет работать, требуется boolean, а найден int**  **// Не будет работать, требуется boolean, а найден int**  **}** |

# #3. Вводный курс по языку программирования JAVA. Условные операторы

**Java** использует булевские (логические) переменные, чтобы оценивать условия. Значение **true** или **false** возвратится, после того как выражение будет оценено. Например:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **int a = 4;**  **boolean b = a == 4;**  **if (b) {**  **System.out.println("It's true!");**  **}** |

Конечно, мы обычно не присваиваем условное выражение булевской переменной, мы просто используем короткую версию:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4** | **int a = 4;**  **if (a == 4) {**  **System.out.println("Ohhh! So a is 4!");**  **}** |

### Логические операторы

Есть не так много операторов, которые можно использовать в условиях. Вот они:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10** | **int a = 4;**  **int b = 5;**  **boolean result;**  **result = a < b;  // истина result = a > b;  // ложь**  **result = a <= 4; // меньше или равно - истина result = b >= 6;  // больше или равно - ложь**  **result = a == b;  // равно - ложь**  **result = a != b;  // неравно - истина**  **result = a > b || a < b;  // логическое ИЛИ - истина**  **result = 3 < a && a < 6;  // логическое И - истина**  **result = !result;  // Логическое НЕ - ложь** |

### Оператор if — else

Синтаксис оператора **if — else** довольно прост:

|  |  |
| --- | --- |
| **1**  **2**  **3** | **if (a == b) {**  **// Тело метода. Выполняется если a и b равны.**  **}** |

Так же мы можем добавить еще одно выражение, на случай, если условие не выполняется:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **if (a == b) {**  **// Мы уже знаем эту часть**  **} else {**  **// a и b не равны... :/**  **}** |

Если тело метода можно разместить в одну строку, можно не использовать { }

|  |  |
| --- | --- |
| **1**  **2** | **if (a == b) System.out.println("Yeah!");**  **else System.out.println("Ohhh...");** |

Или

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4** | **if (a == b)**  **System.out.println("Another line Wow!");**  **else**  **System.out.println("Double rainbow!");** |

Несмотря на то, что такой метод мог бы сделать ваш код короче, мы строго рекомендуем новичкам не использовать короткую версию условного оператора

### Другая сторона if

Есть еще один способ записать **if — else** в одну строку — с помощью оператора **?** :

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10** | **int a = 4;**  **int result = a == 4 ? 1 : 8;**  **// result будет равен 1**  **// Или обычная форма записи:**  **int result;**  **if (a == 4) {**  **result = 1;**  **} else {**  **result = 8;**  **}** |

Опять же, мы не рекомендуем новичкам использовать эту версию if.

### Операторы == и equals

Оператор **==** работает немного по-другому на объектах, нежели на примитивах. Когда вы используем объекты и хотите проверить, равны ли они, оператор **==** скажет что они равны, только если объекты одинаковы, но если вы хотите проверить их на логическое соответствие, используйте метод **equals**. Например:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | **String a = new String("Wow");**  **String b = new String("Wow");**  **String sameA = a;**    **boolean r1 = a == b;      // Ложь, так как a и b не один и тот же объект**  **boolean r2 = a.equals(b); // Истина, так как a и b логически равны**  **boolean r3 = a == sameA;  // Истина, так как a и sameA действительно один и тот же объект** |

# #4. Вводный курс по языку программирования JAVA. Массивы

### Массивы

**Массив** **(англ. Array)** это объект, хранящий в себе фиксированное количество значений одного типа. Другими словами, массив — это нумерованный набор переменных. Переменная в массиве называется элементом массива, а ее позиция в массиве задается индексом.

Массивы в Java тоже представляют собой объекты. Они должны быть объявлены, а затем созданы. Чтобы объявить переменную, которая будет содержать массив целых чисел, мы используем следующий синтаксис:

|  |  |
| --- | --- |
| **1** | **int[] arr;** |

Обратите внимание, размер не указан, так что мы еще не создали массив.

|  |  |
| --- | --- |
| **1** | **arr = new int[10];** |

Теперь мы создали новый массив размером 10. Мы можем проверить размер массива, выведя на экран его длину:

|  |  |
| --- | --- |
| **1** | **System.out.println(arr.length);** |

Так же мы можем получить доступ к массиву и установить значения:

|  |  |
| --- | --- |
| **1**  **2** | **arr[0] = 4;**  **arr[1] = arr[0] + 5;** |

Счет элементов массива в Java начинается с нуля, то есть доступ к первому элементу можно получить по индексу 0 (например, arr[0]). Кроме того, как показано на примере, массив размером 5 будет заканчиваться на индексе 4, так как счет начинается с нуля.

|  |  |
| --- | --- |
| **1**  **2** | **int[] arr = new int[5]**  **arr[4] = 4;  // Получение доступа и присвоение значения последнему элементу** |

Есть также возможность создать массив с указанием значений в одну строку:

|  |  |
| --- | --- |
| **1** | **int[] arr = {1, 2, 3, 4, 5};** |

Кстати, если вы попытаетесь вывести массив целиком на экран, получите что-то в этом роде: [I@f7e6a96.

Для вывода всех значений массива используйте метод Arrays.toString(), преобразующий массив в строку.

|  |  |
| --- | --- |
| **1** | **System.out.println(Arrays.toString(arr));** |

Или напишите цикл, выводящий последовательно элементы массива.

|  |  |
| --- | --- |
| **1**  **2**  **3** | **for(int i=0; i<arr.length; i++) {**  **System.out.println(arr[i]);**  **}** |

IntellijIdea комбинация **CTRL + SHIFT + /** - выделяем текст, нажимаем эту комбинацию и текст у нас становится комментарием.

**ALT + Enter** - импортирует нужный нам класс

# #5. Вводный курс по JAVA. Циклы (Loops)

Есть два вида циклов в Java, for и while.

### For

Цикл for состоит из трех секций:

|  |  |
| --- | --- |
| **1** | **for (int i = 0; i < 3; i++) {}** |

Первая секция выполняется один раз, когда мы входим в цикл.  В нашем примере здесь задается начальное значение переменной i. Вторая секция проверяет логическое условие, если оно возвращает true, выполняются операторы в цикле, если false, выход из цикла. Вторая секция в первый раз запускается сразу после первой секции, и выполняется каждый раз, пока условие верно, вызывая третью секцию. Третья секция — заключительный оператор, его действие выполняется каждый раз при выполнении цикла. В нашем примере это инкремент, который при каждом выполнении увеличивает значение переменной на единицу.

Таким образом, цикл будет работать 3 раза. Вот порядок команд:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12** | **int i = 0;**  **i < 3 // 0 < 3 = true**  **// Inside of loop**  **i++ // i is now 1**  **i < 3 // 1 < 3 = true**  **// Inside of loop**  **i++ // i is now 2**  **i < 3 // 2 < 3 = true**  **// Inside of loop**  **i++ // i is now 3**  **i < 3 // 3 < 3 = false**  **// Loop is done...** |
|  |  |

Мы можем опустить первую и третью секции цикла (как бы странно это ни выглядело), и цикл все еще будет работать:

|  |  |
| --- | --- |
| **1** | **for (;i < 5;) {}** |

Для случаев, где нужно использовать цикл схожих повторяющихся действий, мы используем цикл while

### While

Синтаксис похож на предыдущий:

|  |  |
| --- | --- |
| **1** | **while (condition) {}** |

Условие будет работать впервые при вводе и каждый раз, когда вызывается цикл. Если условие возвратит **false**, то цикл не будет работать.  
Если мы хотим, чтобы цикл всегда выполнял по крайней мере одно действие, мы можем использовать **do-while**:

|  |  |
| --- | --- |
| **1**  **2**  **3** | **do {**    **} while(condition);** |

Не забудьте точку с запятой в конце.

### Foreach

Другая версия **for**, это **foreach**. Но в Java решили не добавлять новое ключевое слово **each**. Ключевое слово, которое мы используем, все еще **for**, но когда мы хотим выполнить действия над элементами массива, делаем так:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4** | **int[] arr = {2, 0, 1, 3};**  **for (int el : arr) {**  **System.out.println(el);**  **}** |

Это была короткая версия, эквивалентная следующей записи:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **int[] arr = {1, 9, 9, 5};**  **for (int i = 0; i < arr.length; i++) {**  **int el = arr[i];**  **System.out.println(el);**  **}** |

Заметьте, что, если вы хотите использовать индекс элемента в цикле, Вы должны использовать более длинную версию и не можете использовать **foreach**.

### break and continue

Эти два ключевых слова помогают нам управлять циклом из него. Оператор **break** останавливает цикл и переходит к оператору, следующему за ним:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12** | **int i;**  **for (i = 0; i < 5; i++) {**  **if (i >= 2) {**  **break;**  **}**  **System.out.println("Yuhu");**  **}**  **System.out.println(i);**  **// Output:**  **// Yuhu**  **// Yuhu**  **// 2** |

Оператор **continue** остановит текущую итерацию и переместится в следующую. Заметьте, что в цикле **for** действие в третьей секции будет выполнено при этом.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18** | **int i;**  **for (i = 0; i < 5; i++) {**  **if (i >= 3) {**  **break;**  **}**  **System.out.println("Yuhu");**  **if (i >= 1) {**  **continue;**  **}**  **System.out.println("Tata");**  **}**  **System.out.println(i);**  **// Yuhu**  **// Tata**  **// Yuhu**  **// Yuhu**  **// 3** |

### Упражнение

Используя цикл, выведите на экран все четные числа из списка чисел в порядке получения. Не выводите числа, идущие после числа 237 в последовательности.

# #6. Вводный курс по JAVA. Методы

Методы в Java — это законченная последовательность действий (инструкций), направленных на решение отдельной задачи. По сути, это функции (они же процедуры, подпрограммы) более ранних, не ООП языков. Только эти функции являются членами классов и для различия с обычными функциями, согласно терминологии объектно-ориентированного программирования, называются методами.

Методы определяются всегда внутри классов:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **public class Main {**  **public static void foo() {**  **// Тело метода**  **}**  **}** |

**foo** -  это метод, который мы определили в классе Main, давайте его рассмотрим.

* **public** — тип доступа (метод может вызываться из другого класса). Существуют и другие типы доступа, к примеру ***private*** (метод доступен только внутри класса) и ***protected*** (о нем мы будем говорить позже).
* **static**  означает что  метод статический, он принадлежит классу Main, а не конкретному экземпляру класса Main. Мы можем вызвать этот метод из другого класса так:  Main.foo().
* **void** значит, что этот метод не возвращает значение. Методы могут возвращать значение в Java и оно должно быть определено при объявлении метода. Однако, вы можете использовать return просто для выхода из метода.
* Этот метод не получает никаких аргументов, но методы java могут получать аргументы, как мы увидим далее на примерах.

Если тип возвращаемого значения не **void**, в теле метода должен быть хотя бы один оператор -

**return *выражение*;**

- где тип выражения должен совпадать с типом возвращаемого значения. Этот оператор возвращает результат вычисления выражения в точку вызова метода.  
Если тип возвращаемого значения – **void**, возврат из метода выполняется либо после выполнения последнего оператора тела метода, либо в результате выполнения оператора   
**return;**  
(таких операторов в теле метода может быть несколько).  
Пример объявления метода, возвращающего значение типа int – сумму двух своих параметров типа int:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **int sum(int a, int b){**  **int x;**  **x = a + b;**  **return x;**  **}** |

При вызове метода, например, sum(5, 3), параметры 5 и 3 передаются в метод, как значения соответственно a и b, и оператор вызова метода sum(5, 3)– заменяется значением, возвращаемым методом (8).

В отличие от языка C, в котором тип параметра, задаваемого при вызове, приводится к типу параметра в объявлении функции, тип задаваемого параметра в Java должен строго соответствовать типу параметра в объявлении метода, поэтому вызов метода **sum(1.5, 8)** приведет к ошибке при компиляции программы.

#### Не статические методы

Не статические методы в Java используются чаще, чем статические методы. Эти методы могут принадлежать любому объекту, экземпляру класса, а не всему классу.

Не статические методы могут получать доступ и изменять поля объекта.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9** | **public class Student {**  **private String name;**  **public String getName() {**  **return name;**  **}**  **public void setName(String name) {**  **this.name = name;**  **}**  **}** |

Вызов методов требует экземпляра класса Student.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6** | **Student s = new Student();**  **s.setName("Danielle");**  **String name = s.getName();**    **Student.setName("Bob"); // Не будет работать!**  **Student.getName(); // Не будет работать!** |

#### Перегруженные методы

В языке Java в пределах одного класса можно определить два или более ме­тодов, которые совместно используют одно и то же имя, но имеют разное количество параметров. Когда это имеет место, методы называют **перегру­женными**, а о процессе говорят как о **перегрузке метода** (method overloading).

Когда метод вызывается, то по количеству параметров и/или их типам среда выполнения Java определяет, какую именно версию перегруженного метода надо вызывать (тип возвращаемого значения во внимание не принимается, хотя, в принципе, он тоже может отличаться у разных версий перегруженных методов).

Например, метод

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **double sum(double a, double b) {**  **double x;**  **x = a + b;**  **return x;**  **}** |

вместе с объявленным ранее методом **int** **sum(int** **a, int** **b)**составляют пару перегруженных методов и при вызове **sum(5, 8)** будет вызван первый метод, а при вызове **sum(5.0, 8.0)** будет вызван второй метод.

По умолчанию метод, как и переменная, доступен только классам в том же пакете (наборе классов), что и исходный класс. Если перед возвращаемым типом задан модификатор доступа **public**, то метод является глобальным и доступен любым объектам, а модификатор **private** означает, что метод доступен в том классе, в котором он был объявлен, т.е. метод инкапсулирован в данном классе.

#### Переопределение методов

Кроме перегрузки существует также замещение, или **переопределение методов** (англ. overriding). Замещение происходит, когда класс потомок (подкласс) определяет некоторый метод, который уже есть в родительском классе(суперклассе), таким образом новый метод заменяет метод суперкласса.  У нового метода подкласса должны быть те же параметры или сигнатура, тип возвращаемого результата, что и у метода родительского класса.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12** | **public class Thought {**  **public void message() {**  **System.out.println("Я себя чувствую как стрекоза, попавшая в параллельную вселенную.");**  **}**  **}**  **public class Advice extends Thought {**  **@Override  // Аннотация @Override с Java 5 является необязательной, но весьма полезной**  **public void message() {**  **System.out.println("Внимание: Даты в календаре ближе, чем кажутся.");**  **}**  **}** |

 Класс **Thought** представляет собой суперкласс и обеспечивает вызов метода **message().** Подкласс, называемый **Advice**, наследует каждый метод класса **Though**t. Однако, класс **Advice** переопределяет метод **message()**, замещая функционал, описанный в классе **Thought**.

В Java, когда подкласс содержит метод, переопределяющий метод суперкласса, то он может помимо своего метода вызывать и метод суперкласса при помощи ключевого слова **super**.  
Например, нижеследующий вариант выводит оба сообщения при вызове метода подкласса:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | **public class Advice extends Thought {**  **@Override**  **public void message() {**  **System.out.println("Внимание: Даты в календаре ближе, чем кажутся.");**  **super.message();  // Вызов версии метода родительского класса**  **}**  **}** |

 Существуют методы, которые подкласс не может переопределять. Например, в Java метод, объявленный с ключевым словом **final**, не может быть переопределён. Методы, объявленные как **private** или **static** не могут быть переопределены, поскольку это соответствует неявному использованию **final**.

#### Резюме

* Каждый java-метод должен быть внутри класса
* Статические методы принадлежат классу, а не статические методы принадлежат объектам, экземплярам класса
* В пределах одного класса может быть два и более методов с одинаковыми именами, но разным набором параметров (перегрузка метода)
* Класс-потомок может обеспечивать свою реализацию метода, уже реализованного в одном из родительских классов (переопределение метода)

#### Упражнение

Написать метод **printFullName** класса **Student,** который выводит полное ФИО студента.

**#7. Вводный курс по JAVA. Объекты, конструкторы**

## Объекты

Почти весь код, написанный на языке Java — это классы и объекты, т.к. java — объектно ориентированный язык. Java-объекты хранят свое состояние в переменных, эти  переменные еще называют полями или членами экземпляра класса.

Давайте начнем с примера:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **class Point {**  **int x;**  **int y;**  **}** |

Этот класс определяет точку с координатами X и Y.

Для того, чтобы создать экземпляр этого класса, мы должны использовать ключевое слово **new**.

|  |  |
| --- | --- |
| **1**  **2** | **Point p = new Point();** |

При этом используется так называемый конструктор по умолчанию (или конструктор без параметров) — это специальный метод класса, мы его не определяли явно, но даже если его не определить, он создаётся автоматически, выполняется при создании каждого нового объекта и присваивает первоначальные значения его свойствам (инициализирует их).

От методов в java конструктор отличается тем, что имеет то же самое имя, что и класс, в котором он определен, а также не имеет типа возвращаемого значения. Конструктор в java возвращает новый объект — экземпляр родительского класса).

Мы можем определить наш собственный конструктор. Поскольку методы можно перегружать, а конструктор является методом, то с помощью перегрузки можно создать дополнительные варианты конструкторов. Например, удобно иметь конструктор, который позволит при создании объекта Point явно указывать его координаты:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10** | **class Point {**  **int x;**  **int y;**    **Point(int x, int y) {**  **this.x = x;**  **this.y = y;**  **}**  **}** |

Это означает, что мы не можем больше использовать конструктор по умолчанию new Point(). Теперь мы можем пользоваться только перегруженным конструктором  с указанием начальных значений new Point(4, 1).

Мы можете определить более чем один конструктор, так что объект класса Point может быть создан несколькими способами. Давайте создадим еще один перегруженный конструктор:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13** | **class Point {**  **int x;**  **int y;**    **Point() {**  **this(0, 0);**  **}**    **Point(int x, int y) {**  **this.x = x;**  **this.y = y;**  **}**  **}** |

Обратите внимание на ключевое слово this. Мы можем использовать его внутри конструктора для вызова другого конструктора (для того, чтобы избежать дублирования кода).

Мы также используем ключевое слово this в качестве ссылки на текущий объект.

После того, как мы определили объект ***р*** мы можем получить доступ к X и Y.

|  |  |
| --- | --- |
| **1**  **2** | **p.x = 3;**  **p.y = 6;** |

### Методы

Теперь мы можем определить методы класса Point.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12** | **class Point {**  **... // Наш код ранее**  **void printPoint() {**  **System.out.println("(" + x + "," + y + ")");**  **}**    **Point center(Point other) {**  **// Возвращает центр между этой и другой точками**  **// Заметьте, мы используем целое число, поэтому не получим точное значение**  **return new Point((x + other.x) / 2, (y + other.y) / 2);**  **}**  **}** |

### Public и Private

На прошлом уроке мы уже рассматривали модификаторы доступа. Ключевое слово private перед переменной или методом  означает, что только сам класс может получить доступ к переменной или методу, а когда мы используем public это значит, что любой объект может получить к нему доступ. Конструкторы обычно определяют как **public**, переменные определяют как **private,** а методы разделяются, некоторые **public,** а некоторые **private,**в зависимости от задач.

## Упражнение

Напишите новый метод класса Point с названием scale, который сделает точку наполовину ближе к началу координат (0,0). Например, точка с координатами (8, 4) после выполнения метода  scale будет иметь координаты (4, 2).

**#8. Компиляция и выполнение java программы с командной строки**

### Компиляция и выполнение java программы

Продолжаем курс программирования java для android-разработчиков. Данный урок научит вас запускать java программы из командной строки, для общего понимания [процесса компиляции и запуска](http://www.fandroid.info/sborka-i-vypolnenie-java-programm/) программ на языке java.

После создания простого приложения, которое выводит что-то на экран, вы должны компилировать ваш код и запустить его.

Независимо от того, какую операционную систему вы используете, Linux, Mac или Windows, если на вашем компьютере установлен [JDK (Java Development Kit)](http://www.oracle.com/technetwork/java/javase/downloads/index.html), вы можете в консоли набрать следующие команды чтобы скомпилировать и запустить программу:

* **javac (или  javac.exe)**
* **java (или  java.exe)**

В первом случае будет вызван компилятор javac.exe, а во втором случае — запускалка java.exe, которая стартует нашу программу. Эти файлы лежат в папке bin  вашего JDK.

Рассмотрим на примере. Вспомним код из первого урока — создадим файл с названием Main.java.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **public class Main {**  **public static void main(String[] args) {**  **System.out.println("Hello, World!");**  **}**  **}** |

Идем в папку, куда среда  разработки сохранила проект. Я работаю в IntelliJIDEA, и мой проект лежит в C:\Users\имя пользователя\IdeaProjects\название проекта\src. Находим там наш файл Main.java. Консоль вызывается так: щелкаем правой клавишей мыши с зажатой клавишей Shift на пустом месте в папке, где лежит файл нашей программы, и выбираем пункт контекстного меню «Открыть окно команд».  
Для того, чтобы скомпилировать его нужно набрать в консоли команду **javac** и в качестве параметра передать имя нашего файла:

|  |  |
| --- | --- |
| **1** | **javac Main.java** |

Эта команда вызовет компилятор, который создаст файл **Main.class**, содержащий скомпилированный код нашей java программы.

Чтобы запустить ее, нужно ввести команду **java** с именем класса (не файла!) в качестве параметра:

|  |  |  |
| --- | --- | --- |
| **1** | **java Main.class //так неправильно** | |
|  |  | |
| **1** | **java Main //так правильно** |

### Аргументы

В главном классе нашей программы есть метод public static void main(...), который в качестве аргумента принимает массив String[] args.

Массив строк в качестве аргумента можно передать в программу при запуске из командной строки .

Любой массив в Java имеет переменную длину, это число элементов в этом массиве.

Добавим такой код в класс **Main.java**:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8** | **public class Main {**  **public static void main(String[] args) {**  **for (int i = 0; i < args.length; i++) {**  **System.out.println(args[i]);**  **}**  **}**  **}** |

И чтобы скомпилировать и запустить программу с аргументами, пишем в консоль:

|  |  |
| --- | --- |
| **1**  **2** | **javac Main.java**  **java Main arg0 arg1 arg2** |

### Упражнение

Создайте программу, которая выводит аргументы, переданные при запуске, в одну строку.

**# 9. Основы языка JAVA. Наследование в java**

## Наследование в java

Наследование в Java позволяет повторно использовать код одного класса в другом классе, то есть вы можете унаследовать новый класс от уже существующего класса. Главный наследуемый класс в Java называют родительским классам, или суперклассом. Наследующий класс называют дочерним классом, или подклассом. Подкласс наследует все поля и свойства суперкласса, а также может иметь свои поля и свойства, отсутствующие в классе-родителе.

### Пример наследования

Рассмотрим класс под названием Shape (Форма). Shape является базовым классом, от которого наследуются другие формы, таких как прямоугольник, квадрат, круг и т.д.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6** | **public class Shape {**  **public double area ()**  **{**  **return 0;**  **}**  **}** |

Поскольку это просто общая «форма», метод  вычисления площади area() будет возвращать ноль.  
Чтобы узнать площадь конкретной фигуры, нужно создать подкласс, унаследованный от класса Shape, и в нем переопределить метод area().

От класса Shape  наследуется класс Circle, который тоже представляет собой форму.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14** | **class Circle extends Shape {  // ключевое слово "extends" означает наследование**    **private static final double PI = Math.PI;   // константа**  **private double diameter; //любое число, представляющее диаметр этого круга**    **public Circle(double diameter) { // конструктор**  **this.diameter = diameter;**  **}**    **public double area(){**  **double radius = diameter / 2.0;**  **return PI \* radius \* radius;**  **}**  **}** |

Метод area() базового класса наследуется классом Circle и становится доступен в нем, но нам нужно  переопределить метод area() в классе Circle, таким образом, чтобы он вычислял площадь круга.

Преимущество использования наследования в том, что вы можете написать код, который можно применить к ряду классов, расширяющих более общий класс.

Создадим  класс Main, и в нем напишем метод, который вычисляет большую площадь двух фигур:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16** | **public class Main {**  **public static void main(String[] args) {**  **Shape s1 = new Circle (5.0);**  **Shape s2 = new Rectangle (5.0, 4.0);**  **Shape larger = getLargerShape(s1,s2);**    **System.out.println("The area of the larger shape is: "+larger.area());**  **}**    **public static Shape getLargerShape(Shape s1, Shape s2) {**  **if(s1.area() > s2.area())**  **return s1;**  **else**  **return s2;**  **}**  **}** |

Как вы можете видеть, метод getLargerShape() не требует указания определенного типа фигуры для его двух параметров. В качестве параметров для этого метода можно использовать экземпляр любого класса, который наследует тип Shape. Можно использовать экземпляр класса круг, прямоугольник, треугольник, трапеция, и т.д. — до тех пор, как они наследуют класс формы.

### Резюме:

Наследование классов ( inheritance ) один из существенных атрибутов ООП (объектно-ориентированного программирования). Оно позволяет строить новые классы на базе существующих, добавляя в них новые возможности или переопределяя существующие.

Унаследованные поля могут быть использованы непосредственно, как и любые другие поля. Вы можете объявить поле в дочернем классе с тем же именем, что и в суперклассе, скрывая его таким образом (но это не рекомендуется). Вы можете объявить новые поля в подклассе, которых нет в суперклассе. Унаследованные методы тоже можно использовать непосредственно. Вы можете написать в подклассе новый метод, который имеет ту же сигнатуру, что и метод суперкласса, это называется переопределением метода. Вы можете написать новый статический метод в подклассе, который имеет ту же сигнатуру, что и метод суперкласса, таким образом, скрывая его (то есть метод суперкласа будет виден внутри подкласса только через ключевое слово super). Вы можете объявить новые методы в подклассе, которых нет в суперклассе. Вы можете написать конструктор подкласса, который вызывает конструктор суперкласса, неявно или с помощью ключевого слова super. Подкласс не наследует закрытые члены родительского класса, например, обозначенные модификатором private.

### Упражнение:

##### Создайте недостающий класс Rectangle, который наследует класс Shape и находит площадь.

**# 10. Основы JAVA. Исключения (Exception) и методы их обработки. Операторы try…catch**

**CTRL + ALT + T** - можно выделить кусок кода и с помощью этой комбинации вызвать **try/catch**

### Что такое исключения (Exception) в языке java, причины их возникновения  и методы обработки.

Исключениями или исключительными ситуациями (состояниями) называются ошибки, возникшие в программе во время её работы. В языке Java все исключения являются объектами и могут быть вызваны не только системой, но и создаваться самим разработчиком.

Исключения делятся на несколько классов, которые имеют общего предка — класс Throwable. Его потомками являются подклассы Exception и Error. Исключения (Exceptions) являются результатом проблем в программе, которые в принципе решаемы и предсказуемы. Например, произошло деление на ноль в целых числах.

Ошибки (Errors) представляют собой более серьёзные проблемы, которые, согласно спецификации Java, не следует пытаться обрабатывать в собственной программе, поскольку они связаны с проблемами уровня JVM. Например, исключения такого рода возникают, если закончилась память, доступная виртуальной машине.

В Java все исключения делятся на три типа: контролируемые исключения (checked) и неконтролируемые исключения (unchecked), к которым относятся ошибки (Errors) и исключения времени выполнения (RuntimeExceptions, потомок класса Exception).

Контролируемые исключения представляют собой ошибки, которые можно и нужно обрабатывать в программе, к этому типу относятся все потомки класса Exception (кроме RuntimeException).

**Общие сведения об исключениях**

Исключения представляют собой объекты, которые позволяют обработать исключительную ситуацию. Исключительная ситуация— это ошибка, происходящая во время выполнения программы (так называемая ошибка времени выполнения).

Исключительные ситуации можно обрабатывать с помощью блока try/catch/finally.

Рассмотрим, как он работает. В блок try помещается код, в котором может содержаться исключительная ситуация. Он работает либо до возникновения исключительной ситуации, либо при успешном выполнении до конца блока try.

В блоке catch выполняются некие операции при возникновении исключительной ситуации определенного типа (в параметрах блока — в круглых скобках — указывается тип возможного исключения — т.е. класс, который обрабатывает исключительную ситуацию).

Блок catch может делать все, что угодно: вызывать необходимый класс-исключение при помощи создания его объектов с использованием ключевого слова throw (это фактически искусственная передача выполнения нужному нам блоку catch), а может и сам производить какие-либо действия.

В блоке finally выполняются определенные восстановительные операции. Он обычно предназначен для освобождения памяти, выделенной под объект. Восстановительные операции будут выполняться независимо от того, возникло исключение (теперь мы будем именно так называть исключительные ситуации) в блоке try или нет.

Блок finally может отсутствовать.  
Конструкция try/catch/finally может быть вложенной. Покажем использование этой конструкции на примере:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49** | **class MyException extends Exception {**    **public void MyException() {**  **System.out.println("Возникло исключение!!!");**  **}**  **}**  **class Toys {**  **public static short numbers = 0;**    **public static short newNumber() {**  **return ++numbers;**  **}**    **public short number;**  **public String name, size;**    **public void setParam(String name, String size) {**  **try {**  **this.name = name;**  **this.size = size;**    **if (this.number != 0) {**  **throw new MyException(); // Вызов метода создания объекта MyException**  **}**    **this.number = Toys.newNumber();**  **} catch (MyException e) {**  **System.err.println("Error!");**  **new MyException().MyException();**  **} catch (Exception e) {**  **// Блок обработки всех остальных исключений блока try.**  **} finally {**  **System.out.println("Итого: \nИмя игрушки: " + this.name**  **+ "\nРазмер игрушки: " + this.size + "\nНомер игрушки: "**  **+ this.number);**  **}**  **}**  **public static void main(String[] args) {**  **Toys car = new Toys();**  **try {**  **car.setParam("Car", "Large");**  **// Какие-то действия над объектом car.**  **car.setParam("Car", "Small");**  **// Вот это приведет к возникновению исключительной ситуации.**  **} catch (Exception e) {**  **// ...**  **}**  **}**  **}** |

Запомните важное правило: если необходимо, чтобы были отдельные обработчики исключения для базового класса и его производных, всегда в начале нужно помещать обработчик для производных классов. Если же сделать наоборот, то всегда будет выполняться обработчик для базового класса, так как он обрабатывает исключения своего типа и производных классов, а это противоречит желанию создать отдельный обработчик для производного класса.Естественно, вы заметили, что мы создали класс-исключение как производный от класса Exception.

Подробнее создание классов-исключений мы рассмотрим позднее, а пока давайте рассмотрим использование стандартного класса ArithmeticException:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17** | **class Program {**  **public static void main(String[] args) {**  **int a = 10;**  **try {**  **a = a / Integer.parseInt(args[0]);**  **System.out.println("10 / " + args[0] + " = " + a);**  **} catch (ArithmeticException e) {**  **System.out.println("Error!");**  **} catch (ArrayIndexOutOfBoundsException e) {**  **System.out.println("Выходза пределы массива");**  **// А если пользователь не ввел значения в коммандной строке?**  **} finally {**  **System.out.println("That\'s all!!!");**  **}**  **}**    **}** |

Классы-исключения в большинстве своем наследуются от производного класса Exception. Можно использовать явное (прямое) обращение к объекту исключений, чтобы вызвать необходимые действия. Напомним о классе из первого примера, который позволяет обрабатывать исключение.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4** | **class MyException extends Exception {**  **public void MyException() {**  **System.out.println ("Возникло исключение!!!"); }**  **}** |

Если необходимо, чтобы с помощью некоего класса обрабатывались какие-либо ошибки в методе, надо после круглых скобок, где указываются параметры, написать ключевое слово throws, а после него— имя класса, который должен обрабатывать какую-либо ошибку.

Если же нужно использовать несколько классов, лучше всего создать базовый класс — и тогда метод будет обрабатывать ошибки с помощью базового класса и всех его производных. Искусственно создать исключения можно с помощью оператора throw. Приведем пример использования класса MyException:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32** | **class Toys {**  **public static short numbers = 0;**    **public static short newNumber() {**  **return ++numbers;**  **}**    **public short number;**  **public String name, size;**    **public void setParam(String name, String size) {**  **try {**  **this.name = name;**  **this.size = size;**    **if (this.number != 0) {**  **throw new MyException(); // Вызов метода создания объекта MyException**  **}**    **this.number = Toys.newNumber();**  **} catch (MyException e) {**  **System.err.println("Error!");**  **new MyException().MyException();**  **} catch (Exception e) {**  **// Блок обработки всех остальных исключений блока try.**  **} finally {**  **System.out.println("Итого: \nИмя игрушки: " + this.name**  **+ "\nРазмер игрушки: " + this.size + "\nНомер игрушки: "**  **+ this.number);**  **}**  **}**  **}** |

Так как мы пытаемся вторично вызвать статический метод создания номера данного объекта, вызовем класс-исключение, который выдает несколько строк и продолжает выполнение программы, не выполнив код приращения номера.

Хотя в языке Java предусмотрен стандартный класс-исключение для ошибок при попытке деления на 0, сейчас мы создадим собственный класс-исключение, который будет обрабатывать подобные ошибки:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34** | **class MyDivByZero extends Exception {**  **public void MyDivByZero() {**  **System.out.println("Что такое? Вы решили делить на 0?");**  **}**  **}**  **class Arithmetic {**  **public static double myDelete(double a, double b) throws MyDivByZero {**  **double c;**  **try {**  **if (b == 0.0) {**  **throw new MyDivByZero();**  **}**  **c = a / b;**  **} catch (MyDivByZero e) {**  **c = 0.0;**  **System.exit(0);**  **}**  **return c;**  **}**    **public static void main(String[] args) {**  **try {**  **System.out.println("5.1, разделить на 1.2 будет "**  **+ myDelete(5.1, 1.2));**  **System.out.println("3.6, разделить на 0 будет "**  **+ myDelete(3.6, 0));**  **System.out.println("7.8, разделить на 3.9 будет "**  **+ myDelete(7.8, 3.9));**    **} catch (MyDivByZero e) {**  **// ...**  **}**  **}**  **}** |

В данном случае мы сами разделили на 0, и проблемы с обработкой исключения не возникло (это всего лишь ошибка программиста — он ее исправит). А вдруг необходимо попросить пользователя ввести число и он введет 0? Тогда-то и будет проблема.

**Основные типы исключений (Exception) в java**

Перечень наиболее часто встречающихся исключений (исключительных ситуаций, ошибок) в языке программирования java с расшифровкой их значения.

### ArithmeticException

Возникла исключительная ситуация, связанная с ошибкой при выполнении арифметического вычисления (например, с попыткой целочисленного деления на нуль). Класс **ArithmeticalException** унаследован от **RuntimeException**.

### ArrayIndexOutOfBoundsException

Задано значение индекса массива, не принадлежащее допустимому диапазону. Имеется дополнительный конструктор, принимающий в качестве параметра ошибочное значение индекса и включающий его в текст описательного сообщения. Класс **ArrayIndexOutOfBoundsException** унаследован от **IndexOutOfBoundException**

### ArrayStoreException

Предпринята попытка сохранения в массиве объекта недопустимого типа. Возникает, если попытаться записать в ячейку массива ссылку на объект неправильного типа.

Класс **ArrayStoreException**унаследован от **RuntimeException.**

### ClassCastException

Выполнена неверная операция преобразования типов (ошибка приведения типов).

Класс **ClassCastException** унаследован от **RuntimeException.**

### ConcurrentModificationException

Осуществлена попытка изменения объекта конкурирующим потоком вычислений (thread) с нарушением контракта класса (тип определен в пакете jav.util).

Также исключение может происходить при работе с коллекциями при обычной однопоточной работе. ConcurrentModificationException возникает когда коллекция модифицируется «одновременно» с проходом по коллекции итератором любыми средствами кроме самого итератора.

Класс **ConcurrentModificationException** унаследован от **RuntimeException.**

### EmptyStackException

Возникает при попытке извлечения объекта из пустого стека. Тип обладает только конструктором без параметров, поскольку причина ситуации очевидна без дополнительных разъяснений (тип определен в пакете java.util).

Класс **EmptyStackException**унаследован от **RuntimeException.**

### IllegalArgumentException

Методу передано неверное значение аргумента (например, отрицательное, когда метод предполагает задание положительных значений).

Класс **IllegalArgumentException**унаследован от **RuntimeException.**

### IllegalMonitorStateException

Выполнено обращение к методу wait, notifyAll или notify объекта, когда текущий поток вычислений не обладает блокировкой (lock) этого объекта.

Класс **IllegalMonitorStateException** унаследован от **RuntimeException.**

### IllegalStateException

Предпринята попытка выполнения операции в то время, когда объект не находится в соответствующем состоянии (например при регистрации или удалении ловушки события закрытия исполняющей системы (shutdown hook) после начала процедуры закрытия).

Класс **IllegalStateException**унаследован от **RuntimeException.**

### IllegalThreadStateException

Предпринята попытка выполнения операции в то время, когда объект потока вычислений не находится в соответствующем состоянии (например, вызван метод start для потока, который уже приступил к работе).

Класс **IllegalThreadStateException** унаследован от **IllegalArgumentException**

### IndexOutOfBoundsException

Задано значение индекса массива или содержимого строки типа String, не принадлежащее допустимому диапазону.

Класс **IndexOutOfBoundsException** унаследован от **RuntimeException**

### MissingResourceException

Не найден требуемый ресурс или пакет ресурсов (resource bundle). Единственный конструктор типа предусматривает задание трех аргументов: строки описательного сообщения, наименования класса ресурсов и объекта ключа, отвечающего отсутствующему ресурсу. Для получения строк наименования класса и ключа применяются методы detClassName и getKey соответственно (тип определен в пакете java.util).

Класс **MissingResourceException**унаследован от **RuntimeException.**

### NegativeArraySizeException

Предпринята попытка создания массива с размером, значение которого задано отрицательным числом.

Класс **NegativeArraySizeException** унаследован от **RuntimeException.**

### NoSuchElementException

Операция поиска элемента в объекте одного из контейнерных классов завершилась неудачей (тип определен в пакете java.util).

Класс **NoSuchElementException** унаследован от **RuntimeException.**

### NullPointerException

Возникает при попытке обращения к полю, методу или объекту по ссылке, равной null. Также исключение выбрасывается, когда метод, не допускающий передачи аргумента null, был вызван с заданием значения null. В последнем случае может быть сгенерировано и исключение типа IllegalArgumentException.

Класс **NullPointerException** унаследован от **RuntimeException.**

### NumberFormatException

Строка, которая, как предполагалось должна содержать представление числа, не отвечает этому требованию. Исключение выбрасывается такими методами, как, например, Integer.parseInt.

Класс **NumberFormatException** унаследован от **IllegalArgumentException.**

### SecurityException

Предпринята попытка выполнения операции, запрещенной системой обеспечения безопасности в соответствии с действующей политикой безопасности.

Класс **SecurityException** унаследован от **RuntimeException.**

### StringIndexOutOfBoundsException

Задано значение индекса содержимого строки типа String, не принадлежащее допустимому диапазону. Имеется дополнительный конструктор, принимающий в качестве параметра ошибочное значение индекса и включающий его в текст описательного сообщения.

Класс **StringIndexOutOfBoundsException** унаследован от **IndexOutOfBoundsException.**

### UndeclaredThrowableException

Выбрасывается при обращении к методу целевого объекта посредством объекта рефлективного класса Proxy, если метод invoke объекта InvocationHandler генерирует объявляемое исключение, которое не допускает присваивания ни одному из типов исключений, упомянутых в предложении throws метода целевого объекта. Рассматриваемое исключение содержит ссылку на исключение, генерируемое методом invoke, которое может быть получено с помощью метода getUndeclaredThrowable. Класс исключений UndeclaredThrowableException поддерживает два конструктора: оба принимают в качестве параметров ссылку на объект Throwable, а один из них, помимо того, строку описания (тип определен в пакете java.lang.reflect).

Класс **UndeclaredThrowableException** унаследован от **RuntimeException.**

### UnsupportedOperationException

Предпринята попытка выполнения операции над объектом, который ее не поддерживает (например, модификация объекта, обозначенного признаком «только для чтения»). используется также классами коллекций из состава пакета java.util как реакция на вызов методов производного класса, реализация которых не обязательна.

Класс **UnsupportedOperationException** унаследован от **RuntimeException.**

**# 11. Основы JAVA. Абстрактные классы и методы в java**

**# 12 Интерфейсы в Java**

**Интерфейсы**

Интерфейсы Java созданы для поддержки динамического выбора (resolution) методов во время выполнения программы. Интерфейсы похожи на классы, но в отличие от последних у интерфейсов нет переменных представителей, а в объявлениях методов отсутствует реализация. Класс может иметь любое количество интерфейсов. Все, что нужно сделать — это реализовать в классе полный набор методов всех интерфейсов. Сигнатуры таких методов класса должны точно совпадать с сигнатурами методов реализуемого в этом классе интерфейса. Интерфейсы обладают своей собственной иерархией, не пересекающейся с классовой иерархией наследования. Это дает возможность реализовать один и тот же интерфейс в различных классах, никак не связанных по линии иерархии классового наследования. Именно в этом и проявляется главная сила интерфейсов.

**Интерфейс** — это структура данных, которая может содержать поля, представленные в виде именованных констант (ключевое слово final тут обычно не указывается, так как поля — это всегда именованные константы) и объявления методов. Интерфейсом могут расширяться многие классы. Интерфейс может сам расширяться несколькими интерфейсами. Заметим, что интерфейс могут использовать сразу несколько классов, независимых друг от друга.

У интерфейса могут быть следующие модификаторы.  
• public (если он есть, то интерфейс доступен отовсюду, если его нет — доступен только в данном пакете).  
• abstract (так как интерфейс всегда абстрактный, то модификатор обычно опускается).  
• strictfp— все позже реализуемые методы должны будут работать с числами с плавающей точкой аналогично на всех машинах Java.  
Приведем пример простого интерфейса, а также класса, который его наследует:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60** | **interface Apple {**  **String red = "red";**  **String yellow = "yellow";**  **String green = "green";**  **}**  **interface Tangerine {**  **String orange = "orange";**  **}**  **interface Fruits extends Apple, Tangerine {**  **String fruit1 = "Apple";**  **String fruit2 = "Tangerine";**  **void setFruit(int number1, int number2);**  **String getFruit();**  **}**  **class MyFruits implements Fruits {**  **String color;**  **String name;**  **private String getColor() {**  **return color;**  **}**  **private void setColor(int number) {**  **switch (number) {**  **case 0:**  **this.color = this.red;**  **break;**  **case 1:**  **this.color = this.yellow;**  **break;**  **default:**  **this.color = this.green;**  **}**  **}**    **public String getFruit() {**  **String s = "Fruit: " + this.name + "Color: " + getColor();**  **return s;**  **}**    **public void setFruit(int number1, int number2) {**  **switch (number1) {**  **case 0:**  **this.name = this.fruit1;**  **setColor(number2);**  **break;**  **default:**  **this.name = this.fruit2;**  **this.color = this.orange;**  **break;**  **}**  **}**    **public static void main(String[] args) {**  **MyFruits my1 = new MyFruits();**  **my1.setFruit(0, 2);**  **System.out.println(my1.getFruit());**  **MyFruits my2 = new MyFruits();**  **my2.setFruit(1, 0);**  **System.out.println(my2.getFruit());**  **}**  **}** |

Можно объявить переменную с типом данных, являющимся наименованием интерфейса. Она может ссылаться на объект класса, который реализует интерфейс, и может вызывать методы, которые объявлены в интерфейсе, а реализованы в классе. Чтобы сделать такую ссылку, необходимо в качестве типа данных ссылки указать наименование интерфейса, а после ключевого слова new указать имя конструктора класса:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21** | **interface Fruit {**  **void setName(String s);**  **}**  **class Apple implements Fruit {**  **String name;**    **public String getName() {**  **return name;**  **}**    **public void setName(String s) {**  **this.name = s;**  **}**    **public static void main(String[] args) {**  **Fruit my1 = new Apple();**  **my1.setName("Apple");**  **System.out.println(my1.getName());**  **// A это ошибка! Нет метода getName() в интерфейсе Fruit.**  **}**  **}** |

Локальными переменными называются переменные, объявляемые в каких-либо блоках (повторим еще раз: блоком называется участок кода между двумя фигурными скобками; блоками могут быть: тело метода, блок инициализации и др.). Локальные переменные существуют до тех пор, пока не кончится блок. Как только блок закончится, локальная переменная перестанет существовать.

**Урок 13. Основы JAVA. Методы обратного вызова (callback)**

**Механизм обратного вызова(callbacks)** широко распространен в программировании. При обратном вызове программист задает действия, которые должны выполняться всякий раз, когда происходит некоторое событие. И неважно, будете разрабатывать только java-программы или android-приложения — колбеки будут встречаться вам повсюду.

Для создания методов обратного вызова (callback) в java используются уже знакомые нам из прошлого урока интерфейсы.

Создадим новый класс **SomeClass** с интерфейсом **Callback**:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22** | **import javax.swing.\*;**    **public class SomeClass {**    **// создаем колбек и его метод**  **interface Callback{**  **void callingBack();**  **}**    **Callback callback;**    **public void registerCallBack(Callback callback){**  **this.callback = callback;**  **}**    **void doSomething(){**  **JOptionPane.showMessageDialog(null, "Выполняется работа");**    **// вызываем метод обратного вызова**  **callback.callingBack();**  **}**  **}** |

У интерфейса мы определили один метод **callingBack()**. Далее в классе мы создали объект интерфейса и инициализировали его в методе registerCallBack. В классе также был создан метод **doSomething()**, в котором может содержаться какой-то сложный код. Мы здесь создаем диалоговое окно, используя компонент JOptionPane библиотеки Swing. Кто не в курсе, **Swing** — библиотека для создания графического интерфейса программ на языке Java.

В конце работы метода **doSomething()** вызывается метод интерфейса callingBack() . В данном случае мы сами создали метод и знаем его код. Но во многих случаях, вы будете использовать готовый метод какого-то класса и можете даже не знать, что именно содержится в этом методе. Вам надо только знать, что такой метод существует и выполняет конкретную задачу.

Теперь нам нужен класс, реализующий интерфейс-callback. Создаем класс MyClass и  подключаем интерфейс-колбек через ключевое слово **implements.**Среда разработки заставит нас реализовать шаблон метода интерфейса.

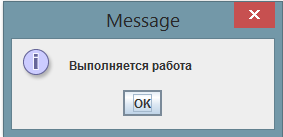
|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9** | **public class MyClass implements SomeClass.Callback {**    **@Override**  **public void callingBack() {**    **System.out.println("Вызов метода обратного вызова");**    **}**  **}** |

Теперь мы можем использовать метод обратного вызова callingBack **()** для решения своих задач. Будем здесь выводить текст в консоль. Создадим класс Main, в котором будет реализован метод main — точка входа в программу.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12** | **public class Main {**    **public static void main(String[] args) {**  **SomeClass someClass = new SomeClass();**  **MyClass myClass = new MyClass();**    **//инициализируем колбек, передавая методу registerCallBack экземпляр MyClass, реализующий интерфейс колбек**  **someClass.registerCallBack(myClass);**  **someClass.doSomething();**    **}**  **}** |

Здесь создаем экземпляры классов, инициализируем колбек, передавая методу registerCallBack экземпляр MyClass, реализующий интерфейс-callback. И вызываем метод doSomething();

При старте программы будет выполняться код метода doSomething(); из класса **SomeClass**.  На экране появилось диалоговое окно.



Когда мы нажмем кнопку, или закроем диалоговое окно,  сработает метод обратного вызова callingBack**()**, который выведет сообщение в консоль.

Через интерфейс-callback также можно передавать какие-то данные. Давайте немного усложним пример.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31** | **package callback;**    **import javax.swing.\*;**    **public class SomeClass {**    **String replyTo;**    **interface Callback{**  **void callingBack(String s);**  **}**    **private Callback callback;**    **public void registerCallBack(Callback callback){**  **this.callback = callback;**  **}**    **void doSomething(){**  **int reply = JOptionPane.showConfirmDialog(null, "Вы программист?", "Опрос", JOptionPane.YES\_NO\_OPTION);**    **if (reply == JOptionPane.NO\_OPTION){**  **replyTo = "Нет";**  **}**  **if (reply == JOptionPane.YES\_OPTION){**  **replyTo = "Да";**  **}**    **callback.callingBack(replyTo);**  **}**  **}** |

Объявим строковую переменную replyTo. Добавим аргумент метода callingBack — строковую переменную.

Изменим диалоговое окно — присвоим его новой целочисленной переменной reply, и изменим на showConfirmDialog, который имеет заголовок окна и больше одной кнопки ответа. В зависимости от нажатой кнопки переменной reply будет присвоено значение. Сравниваем ее значение с константой и присваиваем значение «Да» или «Нет» переменной replyTo.

Теперь внесем изменения в MyClass.

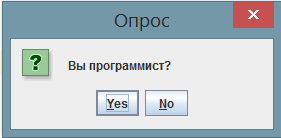
|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13** | **package callback;**    **public class MyClass implements SomeClass.Callback {**  **@Override**  **public void callingBack(String s) {**    **if (s != null) {**  **System.out.println("Ваш ответ: " + s);**  **} else {**  **System.out.println("Вы не ответили на вопрос!");**  **}**  **}**  **}** |

Добавим аргумент в метод. И добавим условие проверки, что строка не пустая — это возможно, если  вместо ответа просто закрыть диалоговое окно.

Класс Main без изменений.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14** | **package callback;**      **public class Main {**    **public static void main(String[] args) {**    **SomeClass someClass = new SomeClass();**  **MyClass myClass = new MyClass();**    **someClass.registerCallBack(myClass);**  **someClass.doSomething();**  **}**  **}** |

Запускаем программу. Теперь текст сообщения в консоли меняется в зависимости от того, выберем мы да, нет или просто закроем окно.



**Примеры методов обратного вызова (callbacks)**

Callback  в чистом виде, это когда вы сформировали задачу и отправили её выполнять какому то там процессу, но не ждёте её завершения и продолжаете  работать, а чтобы к вам мог вернутся результат работы, вы для воркера передаёте ссылку на свой класс с кастом в интерфейс калбека в котором обьявлен метод, при помощи которого воркер может вернуть данные. Вся работа происходит асинхронно без блокировок и ожиданий. В этом смысл калбеков, так как некоторые задачи могут идти долго, а некоторые быстро, но каллер не должен об этом заботиться, он просто предоставляет интерфейс для ответа и продолжает работу.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62**  **63**  **64** | **import java.util.ArrayList;**  **import java.util.Random;**  **/\*\***  **\***  **\* @author mutagen**  **\*/**  **public class CallBackDemo {**  **public static void main(String[] args) {**  **Caller caller = new Caller();**  **for (int i = 0; i < 10; i++) {**  **new Worker(caller).start();**  **}**  **//        итого у нас 9 воркеров отработали и вернули ответ по калбеку в коллекцию**  **System.out.println(caller.getStatuses());**  **}**  **static class Caller implements Callback {**  **private ArrayList<Integer> statuses = new ArrayList<>();**  **public ArrayList<Integer> getStatuses() {**  **return statuses;**  **}**  **@Override**  **public void callMeBack(int status) {**  **//            тут может быть логика куда вернутся данные, может быть коллекция, я выбрал попроще**  **synchronized (statuses) {**  **statuses.add(status);**  **}**  **}**  **}**  **static interface Callback {**  **//        подготовим интерфейс по которому нам будут возвращать данные**  **void callMeBack(int status);**  **}**  **static class Worker extends Thread {**  **private Callback cb;**  **public Worker(Callback cb) {**  **this.cb = cb;**  **}**  **int pleaseDoMeAFavor() {**  **return new Random().nextInt();**  **}**  **@Override**  **public void run() {**  **//            выполним работу**  **int st = pleaseDoMeAFavor();**  **//            и вернём данные вызывающему по калбек интерфейсу**  **cb.callMeBack(st);**  **}**  **}**  **}** |

Допустим, мы разрабатываем фреймворк. У нас есть определенный шаблон действий, в котором только один пункт мы не можем прописать жестко и в этом месте необходимо, чтобы пользователь вставил свой функционал. Это можно сделать абстрактным методом, но в этом случае проблема: во-первых мы не сможем динамически менять такие «вставки», во-вторых, для 1000 функционалов мы получим огромную иерархию наследования. Кроме того, так как множественное наследование в Java не поддерживается, наследование можно использовать только в отдельных случаях. В этом случае можно использовать коллбэки.  
Яркий пример использования колбэков — Spring-овские темплейты для работы с JDBC — они исполняют общие функции самостоятельно, передавая пользователю только такие операции, как обработка ResultSet-ов.

У нас есть темплейтный класс:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35** | **package callbacks;**  **import java.util.Arrays;**  **import java.util.List;**  **public class Template {**  **private ICustomOperation callback;**  **public Template(ICustomOperation callback) {**  **this.callback = callback;**  **}**  **public void setCallback(ICustomOperation callback) {**  **this.callback = callback;**  **}**  **public void processData(List<String> lines) {**  **for (String line : lines) {**  **System.out.println("Input: " + line);**  **line = "[" + line + "]";**  **System.out.println("Standart operation 1 result : " + line);**  **line = line.replaceAll(" ", "\_");**  **System.out.println("Standart operation 2 result : " + line);**  **line = callback.evaluate(line); // здесь мы будем передавать контроль над обработкой пользователю**  **System.out.println("Custom operation result     : " + line);**  **line = line.toLowerCase();**  **System.out.println("Standart operation 3 result : " + line);**  **}**  **}**  **}** |

Есть колбэк-интерфейс:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | **package callbacks;**  **public interface ICustomOperation {**  **public String evaluate(String input);**  **}** |

И различные его реализации:

|  |  |  |
| --- | --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9** | **package callbacks;**  **public class CustomOperation implements ICustomOperation {**  **public String evaluate(String input) {**  **return "CUSTOM: " + input;**  **}**  **}** | |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9** | **package callbacks;**  **public class AnotherCustomOperation implements ICustomOperation {**  **public String evaluate(String input) {**  **return "ANOTHER CUSTOM: " + input;**  **}**  **}** |

Теперь запуск:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6** | **Template template = new Template(new CustomOperation());**  **List<String> data = Arrays.asList(new String[] { "first line", "SEcOnD LiNe" });**  **template.processData(data);**  **template.setCallback(new AnotherCustomOperation());**  **template.processData(data);** |

Есть ещё одно применение: допустим, у нас есть объект, у которого private метод, который вы не хотите делать public или protected. Вы вызваете метод другого класса и изнутри него хотите сделать вызов нашего private-метода.Это можно сделать следующим образом:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39** | **package callbacks;**  **public class PrivateFieldHolder {**  **private OuterCaller outerCaller;**  **public OuterCaller getOuterCaller() {**  **return outerCaller;**  **}**  **public void setOuterCaller(OuterCaller outerCaller) {**  **this.outerCaller = outerCaller;**  **}**  **public void publicMethod() {**  **System.out.println("Holder: public method call");**  **// здесь хитрая штука: мы передаем в вызов метода**  **// внешнего класса коллбэк, который**  **// является анонимным классом внутри данного класса**  **// т.к. этот анонимный класс видит private-метод, то мы**  **// можем поместить внутрь callPrivateMethod вызов этого**  **// private-метода. Таким образом при вызове**  **// callPrivateMethod во внешнем классе позволит обратиться**  **// к private-методу этого класса.**  **outerCaller.callFromOutsidePrivateMethod(new IPrivateMethodCallback() {**  **public void callPrivateMethod(String caller) {**  **privateMethod(caller);**  **}**  **});**  **}**  **private void privateMethod(String caller) {**  **System.out.println(caller + ": private method call");**  **}**  **}** |

Колбэк-интерфейс:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | **package callbacks;**  **public interface IPrivateMethodCallback {**  **public void callPrivateMethod(String caller);**  **}** |

Внешний класс, которому нужно вызвать private-метод:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10** | **package callbacks;**  **public class OuterCaller {**  **public void callFromOutsidePrivateMethod(IPrivateMethodCallback callback) {**  **System.out.println("Outer method call");**  **callback.callPrivateMethod("Outer caller");**  **}**  **}** |

Теперь запуск:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4** | **PrivateFieldHolder pfh = new PrivateFieldHolder();**  **OuterCaller oc = new OuterCaller();**  **pfh.setOuterCaller(oc);**  **pfh.publicMethod();** |