**Programming with classes**

1. Опишите процедуру инициализации полей класса и полей экземпляра класса. Когда инициализируются поля класса, а когда – поля экземпляров класса. Какие значения присваиваются полям по умолчанию? Где еще в классе полям могут быть присвоены начальные значения?
2. Дайте определение перегрузке методов. Как вы думаете, чем удобна перегрузка методов? Укажите, какие методы могут перегружаться, и какими методами они могут быть перегружены? Можно ли перегрузить методы в базовом и производном классах? Можно ли private метод базового класса перегрузить public методов производного? Можно ли перегрузить конструкторы, и можно ли при перегрузке конструкторов менять атрибуты доступа у конструкторов?
3. Объясните, что такое раннее и позднее связывание? Перегрузка – это раннее или позднее связывание? Объясните правила, которым следует компилятор при разрешении перегрузки; в том числе, если методы перегружаются примитивными типами, между которыми возможно неявное приведение или ссылочными типами, состоящими в иерархической связи.
4. Объясните, как вы понимаете, что такое неявная ссылка this? В каких методах эта ссылка присутствует, а в каких – нет, и почему?
5. Что такое финальные поля, какие поля можно объявить со спецификатором final? Где можно инициализировать финальные поля?
6. Что такое статические поля, статические финальные поля и статические методы. К чему имеют доступ статические методы? Можно ли перегрузить и переопределить статические методы? Наследуются ли статические методы?
7. Что такое логические и статические блоки инициализации? Сколько их может быть в классе, в каком порядке они могут быть размещены и в каком порядке вызываются?
8. Что представляют собой методы с переменным числом параметров, как передаются параметры в такие методы и что представляет собой такой параметр в методе? Как осуществляется выбор подходящего метода, при использовании перегрузки для методов с переменным числом параметров?
9. Чем является класс Object? Перечислите известные вам методы класса  Object, укажите их назначение.
10. Что такое хэш-значение? Объясните, почему два разных объекта могут сгенерировать одинаковые хэш-коды?
11. Что такое объект класса Class? Чем использование метода getClass() и последующего сравнения возвращенного значения с Type.class отличается от использования оператора instanceof?
12. Укажите правила переопределения методов equals(), hashCode() и toString().

***1. Опишите процедуру инициализации полей класса и полей экземпляра класса. Когда инициализируются поля класса, а когда – поля экземпляров класса. Какие значения присваиваются полям по умолчанию? Где еще в классе полям могут быть присвоены начальные значения?***

Как известно, в Java поля (fields) могут принадлежать классу или объекту. Поля, принадлежащие классу, являются статическими, а поля, принадлежащие объекту, - нестатическими. Статические поля доступны без создания объекта класса. Соответственно инициализироваться статические и нестатические поля должны в разное время: одни до создания объекта класса, а другие после.

## Типы инициализации полей объектов и классов

Существуют следующие методы инициализации полей:

|  |  |  |
| --- | --- | --- |
| **Название** | **Применимость** | **Описание** |
| Инициализация в месте объявления поля | Поля класса, поля объекта | Применяется, если инициализация может быть произведена коротким выражением и доступен контекст, необходимый для ее проведения |
| Инициализационный блок | Поля класса, поля объекта | Применяется, если инициализационный код неудобно записывать одним выражением или же, например, нужна обработка проверяемых исключений. В случае объектов может применяться для инициализации полей объектов анонимных классов. |
| Конструктор класса | Поля объекта | Применяется, если для инициализации нужны параметры конструктора |

Далее мы рассмотрим каждый тип инициализации подробнее.

## Инициализация статических полей в месте объявления

class Integer {

...

public static final int SIZE = 32;

...

}

Здесь инициализируется статическое поле SIZE класса Integer. Сама инициализация произойдет во время загрузки класса.

## Инициализация в статическом блоке

В некоторых случаях инициализацию неудобно проводить в месте объявления переменной. Например, если в результате выполнения инициализирующего выражения происходит проверяемое исключение. Или же, если инициализация производится путем выполнения кода, который не может быть представлен в виде выражения. Для таких случаев в Java предусмотрен специальный языковой элемент - инициализационный статический блок. Покажем на примере:

static List<Character> alphabet;

static {

alphabet = new ArrayList<Character>();

for (char c='a'; c<='z'; c++) alphabet.add(c);

}

Переменная alphabet инициализируется в статическом блоке. Инициализация происходит во время загрузки класса аналогично как и в предыдущем примере.

Инициализация статических полей в месте объявления и статические блоки выполняются в порядке их объявления в классе

Инициализация полей объекта

В отличии от полей класса, поля объекта инициализируются во время конструирования экземпляра класса. В Java существует 3 типа такой инициализации:

* инициализация в месте объявления
* инициализация в нестатическом блоке
* инициализация в конструкторе

Инициализация полей объекта в месте объявления

Покажем на примере:

public class Blog {

...

private List<Post> posts = new ArrayList<Post>

...

}

Инициализация поля posts будет произведена во время конструирования объекта Blog.

## Инициализация полей объекта в нестатическом блоке

Использование инициализационных блоков является альтернативой предыдущему способу инициализации. Данный тип инициализации может использоваться, если:

* во время инициализации необходимо обработать проверяемое исключение
* значение поля не удобно вычислять с помощью выражения (например, для этого нужен специально созданный класс или метод, который не хочется создавать только для этих целей)
* необходимо инициализировать поле анонимного класса (в анонимном классе невозможно объявить конструктор)

Пример обработки проверяемого исключения:

class Year2000Problem {

Date start;

{

try { start = new SimpleDateFormat("dd.MM.yyyy").parse("01.01.2000"); }

catch (ParseException impossible) {}

}

}

## Инициализация полей объекта в конструкторе

Часто инициализацию полей объекта имеет смысл проводить только с учетом значений параметров конструктора. В таких случаях ее производят в самом конструкторе. Пример:

class User {

...

String login;

User(String login) { this.login = login; }

...

}

Инициализация в конструкторе и наследование

Выполним следующий код:

public class InheritanceInitOrder {

static class A {

String a;

A() {

a = "a";

System.out.println("a initialized");

System.out.println("b=" + ((B)this).b);

}

}

static class B extends A {

String b;

B() {

b = "b";

System.out.println("b initialized");

System.out.println("b=" + b);

}

}

public static void main(String[] args) throws ClassNotFoundException {

new B();

}

}

В out будет выведено:

a initialized

b=null

b initialized

b=b

Вывод свидетельствует о том, что инициализация выполнялась следующим образом:

* конструктор B первым делом вызвал конструктор предка - класса A
* конструктор A проинициализировал поле a объекта А
* при возврате из конструктора A, конструктор B проинициализировал поле b объекта B

## Порядок инициализации полей объекта

Давайте теперь проанализируем порядок выполнения инициализаторов полей объекта. Для этого выполним код, включающий все типы инициализаторов:

public class ObjectFieldsInitOrder {

static int initialize(String message) {

System.out.println(message);

return 0;

}

static class A {

int i0 = initialize("i0");

int i1;

{ i1 = initialize("i1"); }

int i2 = initialize("i2");

int i3;

A() { i3 = initialize("i3"); }

}

static class B extends A {

int i4 = initialize("i4");

int i5;

{ i5 = initialize("i5"); }

int i6;

B() { i6 = initialize("i6"); }

}

public static void main(String[] args) {

new B();

}

}

В результате выполнения получим:

i0

i1

i2

i3

i4

i5

i6

## Вывод свидельствует о том что:

## инициализация полей в месте объявления и в инициализационном блоке происходит до инициализации в конструкторе

## инициализации полей в месте объявления и в инициализационных блоках выполняются в порядке их объявления в классе

## инициализация полей базового класса происходит полностью до инициализации производного класса, т.е. сначала выполняются все инициализаторы базового класса, а потом все инициализаторы производного класса.

**Значения полей по умолчанию:**

* ссылочные типы – null;
* int, byte, short, long – 0;
* double, float – 0.0;
* char – ‘\u0000’;
* boolean – false;

Явная инициализация означает установление начального (нужного) значения переменной при ее объявлении в классе. **Например.** В классе MyClass реализована инициализация начальными значениями переменных разных типов.

**public** **class** CTrain01

{

// явная инициализация начальных значений переменных

**int** d = 25; // явная инициализация переменной d типа int значением 25

**float** y = 3.885f;

**public** **double** x = -129.48;

**boolean** b = **true**;

**char** c = 'Z';

**long** l = 0xF3309FA;

// ...

}

Если членом данных класса есть переменная-ссылка на некоторый класс (объект класса), то она инициализируется стандартным способом с помощью оператора new:

**class** InnerClass

{

// ...

}

**class** MyClass

{

// ...

// явная инициализация переменной obj в классе MyClass

InnerClass obj = **new** InnerClass();

// ...

}

В классе  MyClass  переменная-ссылка obj перед ее использованием обязательно должна быть инициализирована оператором new. Если попробовать использовать неинициализированную переменную, которая есть ссылкой на класс, возникнет критическая ситуация (исключение).

При объявлении, члены данных класса могут быть инициализированы вызовом некоторого метода.

Например. Пусть задан класс CRadius. В классе CRadius инициализируются члены данных len, area, volume с помощью вызова методов Length(), Area(), Volume(). Эти методы вычисляют соответственно длину окружности, площадь круга и объем шара радиуса r, который есть входным параметром методов.

**public** **class** CRadius

{

// скрытые переменные

**private** **double** Pi = 3.1415;

**private** **double** radius=1;

// общедоступные переменные

**public** **double** len = Length(radius); // инициализация переменной len методом Length()

**public** **double** area = Area(radius); // инициализация переменной area методом Area()

**public** **double** volume = Volume(radius); // инициализация переменной volume методом Volume()

// методы вычисления

**double** Length(**double** r)     { **return** 2\*Pi\*r; }

**double** Area(**double** r)   { **return** Pi\*r\*r; }

**double** Volume(**double** r) { **return** 4.0/3.0\*Pi\*r\*r\*r; }

**public** **static** **void** main(String[] args)

{

// демонстрация инициализации методами класса CRadius

CRadius r1 = **new** CRadius(); // происходит явная инициализация членов данных объекта r1

**double** l, a, v;

l = r1.len; // l = 6.283;

a = r1.area; // a = 3.1415

v = r1.volume; // v = 4.1886666666

}

}

***2. Дайте определение перегрузке методов. Как вы думаете, чем удобна перегрузка методов? Укажите, какие методы могут перегружаться, и какими методами они могут быть перегружены? Можно ли перегрузить методы в базовом и производном классах? Можно ли private метод базового класса перегрузить public методов производного? Можно ли перегрузить конструкторы, и можно ли при перегрузке конструкторов менять атрибуты доступа у конструкторов?***

Перегрузка методов

В Java разрешается в одном и том же классе определять два или более метода с одним именем, если только объявления их параметров отличаются. Это называется *перегрузкой методов.*Например, в следующем примере один класс объявляет два метода с одним именем test, но разными параметрами:

public class Overloading1 {

void test(int a) {

System.out.println("a: " + a);

}

void test(int a, int b) {

System.out.println("a и b: " + a + " " + b);

}

public static void main(String[] args) {

Overloading1 ob = new Overloading1();

ob.test(10);

ob.test(10, 20);

}

}

Как же JVM различает какой метод необходимо вызвать? Для этого в Java используется тип и/или количество аргументов метода.

*Перегрузка методов* является одним из способов поддержки полиморфизма в Java.

Возвращаемые типы перегружаемых методов могут отличаться, но самого возвращаемого типа недостаточно для того, чтобы отличать два разных варианта метода.

В следующем примере методы void test() и double test(double a) возвращают значения разного типа. Это допустимо, но при условии, что параметры методов будут отличаться. Закомментированный метод int test() отличается от void test() только типом - это недопустимо для перегруженного метода. Если его добавить, будет ошибка компиляции:

public class Overloading2 {

void test() {

System.out.println("Без параметров");

}

//неправильная перегрузка методов

/\* int test() {

System.out.println("Без параметров");

return 1;

}\*/

double test(double a) {

System.out.println("double a: " + a);

return a \* a;

}

public static void main(String[] args) {

Overloading2 ob = new Overloading2();

ob.test();

double result = ob.test(123.25);

System.out.println("Результат вызова метода ob.test(123.25): " + result);

}

}

Если имя метода перегружено в классе, то параметры этого метода могут отличаться:

* количеством параметров;
* типами параметров, которые получает метод.

В классе может быть перегружено любое количество методов.

Переопределение наследованных методов в производном классе – overriding.

Переопределять можно любые видимые в производном классе не финальные методы. Исключение - переопределение конструкторов базового класс невозможно!

«Переопределение» полей – разрешается создавать два одноименных поля: *this.n   vs  super.n*

В производном классе разрешается перегружать (overload) любые видимые методы.

При наследовании, Вы берете существующий класс и создаете специальную его версию. В основном это означает, что Вы берете главный, целевой класс и приспосабливаете его для частных нужд.

Дочерний класс может переопределить методы экземпляра своего родительского класса. Это называется переопределением метода. Сигнатура (тип возврата, тип параметров, количество параметров и порядок параметров) должна быть такой же, какой была определена в родительском классе. Переопределение метода выполняется для достижения полиморфизма во время выполнения программы.

Перегрузка конструкторов

Конструкторы похожи на методы, поэтому они тоже могут быть перегружены - вы можете объявлять в одном классе несколько конструкторов, которые различаются количеством и типом переменных. В следующем примере добавлены три конструктора в класс Box6. Конечно же, при создании объекта вызывается только один из них - тот, который кажется вам наиболее подходящим:

public class Box6 {

double width;

double height;

double depth;

Box6(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

Box6() {

width = -1; // используем -1 для

height = -1; // обозначения непроинициализированной

depth = -1; // коробки

}

Box6(double len) {

width = len;

height = len;

depth = len;

}

double getVolume() {

return width \* height \* depth;

}

}

public class OverloadCons {

public static void main(String[] args) {

Box6 myBox1 = new Box6(10, 20, 15);

Box6 myBox2 = new Box6();

Box6 myBox3 = new Box6(7);

System.out.println("Объем myBox1: " + myBox1.getVolume());

System.out.println("Объем myBox2: " + myBox2.getVolume());

System.out.println("Объем myBox3: " + myBox3.getVolume());

}

}

***3. Объясните, что такое раннее и позднее связывание? Перегрузка – это раннее или позднее связывание? Объясните правила, которым следует компилятор при разрешении перегрузки; в том числе, если методы перегружаются примитивными типами, между которыми возможно неявное приведение или ссылочными типами, состоящими в иерархической связи.***

Связывание означает наличие связи между ссылкой и кодом. Например, переменная, на которую вы ссылаетесь, привязана к коду, в котором она определена. Аналогично, вызываемый метод привязан к месту в коде, где он определен.

Существует два типа связывания методов в языке Java: ранее связывание (его ещё называют статическим) и позднее (соответственно, динамическое) ***связывание***. Вызов метода в Java означает, что этот метод привязывается к конкретному коду или в момент компиляции, или во время выполнения, при запуске программы и создании объектов. Можно понять из названия, статическое связывание носит более статический характер, так как происходит во время компиляции, то есть код «знает», какой метод вызывать после компиляции исходного кода на Java в файлы классов. А поскольку это относится к ранней стадии жизненного цикла программы, то называется также ранним связыванием (early binding). С другой стороны, динамическое связывание происходит во время выполнения, после запуска программы виртуальной машиной Java. В этом случае то, какой метод вызвать, определяется конкретным объектом, так что в момент компиляции информация недоступна, ведь объекты создаются во время выполнения. А поскольку это происходит на поздней стадии жизненного цикла программы, то называется в языке Java поздним связыванием (late binding).

***Фундаментальное различие между статическим и динамическим связыванием в Java*** состоит в том, что первое происходит рано, во время компиляции на основе типа ссылочной переменной, а второе – позднее, во время выполнения, с использованием конкретных объектов.

Ключевые различия между ранним и поздним связыванием в языке Java:

1. Статическое связывание происходит во время компиляции, а динамическое – во время выполнения.

2. Поскольку статическое связывание происходит на ранней стадии жизненного цикла программы, его называют ранним связыванием. Аналогично, динамическое связывание называют также поздним связыванием, поскольку оно происходит позже, во время работы программы.

3. Статическое связывание используется в языке Java для разрешения перегруженных методов, в то время как динамическое связывание используется в языке Java для разрешения переопределенных методов.

4. Аналогично, приватные, статические и терминальные методы разрешаются при помощи статического связывания, поскольку их нельзя переопределять, а все виртуальные методы разрешаются при помощи динамического связывания.

5. В случае статического связывания используются не конкретные объекты, а информация о типе, то есть для обнаружения нужного метода используется тип ссылочной переменной. С другой стороны, при динамическом связывании для нахождения нужного метода в Java используется конкретный объект.

**приватные**, **статические** и **final-методы** связываются при помощи **статического связывания**, а **виртуальные – динамического**. Аналогично, лучший пример статического связывания – перегрузка методов, а переопределение – динамического.

При вызове перегруженного метода для определения нужной версии Java использует тип и/или количество аргументов метода. Следовательно, перегруженные методы должны различаться по типу и/или количеству их параметров. Хотя возвращаемые типы перегруженных методов могут быть различны, самого возвращаемого типа не достаточно для различения двух версий метода. Когда Java встречает вызов перегруженного метода, она просто выполняет ту его версию, параметры которой соответствуют аргументам, использованным в вызове.

При вызове перегруженного метода Java ищет соответствие между аргументами, которые были использованы для вызова метода, и параметрами метода. Однако это соответствие не обязательно должно быть полным. В некоторых случаях к разрешению перегрузки может применяться автоматическое преобразование типов Java.

Java может автоматически преобразовывать тип integer в тип double, и это преобразование может использоваться для разрешения вызова. Java будет использовать автоматическое преобразование типов только при отсутствии полного соответствия.

Передать методу с аргументом int можно значение типа byte, short и float. Значения типа byte и short исполняющая система Java автоматически преобразует в тип int. В результате будет вызван вариант метода f(int). А если параметр имеет значение типа float, то оно преобразуется в тип doube, и далее вызывается вариант метода f(double).Важно понимать, что автоматическое преобразование типов выполняется лишь в от­сутствие прямого соответствия типов параметра и аргумента.

***4. Объясните, как вы понимаете, что такое неявная ссылка this? В каких методах эта ссылка присутствует, а в каких – нет, и почему?***

Как правило, применять this нужно в двух случаях:

1. Когда у переменной экземпляра класса и переменной метода/конструктора одинаковые имена;
2. Когда нужно вызвать конструктор одного типа (например, конструктор по умолчанию или параметризированный) из другого. Это еще называется явным вызовом конструктора.

Иногда требуется, чтобы метод ссылался на вызвавший его объект. Ключевое слово *this* в Java используется в теле любого метода для ссылки на текущий объект.

Рассмотрим конструктор, в котором параметры имеют те же имена, что и переменные класса. В этом случае параметры перекрывают область видимости переменных класса и мы не можем напрямую обратится к переменным класса. Чтобы это сделать используется ключевое слово *this*.

Второй вариант использования ключевого слова *this() -*с его помощью можно вызвать один конструктор из другого. Вызов *this()* может находиться только в первой строчке конструктора:

**class** Human{

**int** age;

**int** weight;

**int** height;

Human(**int** age, **int** weight){

**this**.age = age;

**this**.weight = weight;

}

Human(**int** age, **int** weight, **int** height){

//вы вызываете конструктор с двумя параметрами

**this**(age, weight);

//и добавляете недостающую переменную

**this**.height = height;

}

}

Ключевое слово this в Java используется только в составе методов либо конструкторов экземпляра класса. Но неявно ключевое слово this передается во все методы, кроме статических (поэтому this часто называют неявным параметром) и может быть использовано для обращения к объекту, вызвавшему метод. Бояться этого ключевого слова не нужно, потому что This не страшно.

Ключевое слово Javas this используется для ссылки на текущий экземпляр метода, на котором он используется.

Ниже приведены способы использования этого:

Чтобы специально указать, что вместо переменной или локальной переменной используется переменная экземпляра. То есть

*private String javaFAQ;*

*void methodName(String javaFAQ) {*

*this.javaFAQ = javaFAQ;*

*}*

Здесь это относится к переменной экземпляра. Здесь приоритет высокий для локальной переменной. Поэтому отсутствие this обозначает локальную переменную. Если локальная переменная, которая является именем параметра, не такая же, как переменная экземпляра, то независимо от this используется или нет, она обозначает переменную экземпляра.

*this используется для ссылки на конструкторы*

*public JavaQuestions(String javapapers) {*

*this(javapapers, true);*

*}*

Это вызывает конструктор того же класса java, который имеет два параметра.

this используется для передачи текущего экземпляра java в качестве параметра

*obj.itIsMe(this);*

Как и выше, это также можно использовать для возврата текущего экземпляра

*CurrentClassName startMethod() {*

*return this;*

*}*

Примечание. Это может привести к нежелательным результатам при использовании во внутренних классах в двух указанных выше пунктах. Поскольку это будет относиться к внутреннему классу, а не к внешнему экземпляру.

this можно использовать для получения дескриптора текущего класса

*Class className = this.getClass(); // this methodology is preferable in java*

*Хотя это можно сделать с помощью*

*Class className = ABC.class; // here ABC refers to the class name and you need to know that!*

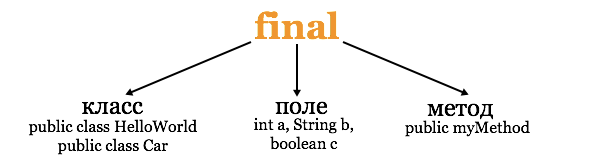
Как всегда, this связан с его экземпляром, и это не будет работать в статических методах.

***5. Что такое финальные поля, какие поля можно объявить со спецификатором final? Где можно инициализировать финальные поля?***

**Модификатор final** - это способ, с помощью которого вы можете контролировать работу своей программы и ее составных частей. Это один из Ваших инструментов.

Суть **модификатора final** - сделать дальнейшее изменение объекта невозможным. С английского "final" можно перевести как **"последний, окончательный"**.

Можно применять этот модификатор **тремя способами**: для класса, для поля (переменной) и для метода.



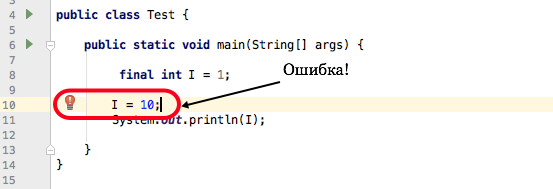
Если вы хотите, чтобы **после инициализации** никто **не мог бы изменить** вашу переменную, напишите слово "final":

|  |  |
| --- | --- |
| 2  3  4  5  6  7  8  9  10 | *public class Test {*    *public static void main(String[] args) {*    *final int I = 1;*    *System.out.println(I);*    *}*  *}* |

Или так:

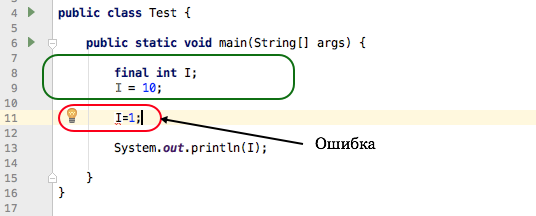
|  |  |
| --- | --- |
| 6  7  8  9  10 | *public class Test {*    *final int I = 1;*    *public static void main(String[] args) {*    *System.out.println(I);*    *}*  *}* |

Теперь, изменить переменную нельзя. Если вы попробуете поменять значение, то получите ошибку:

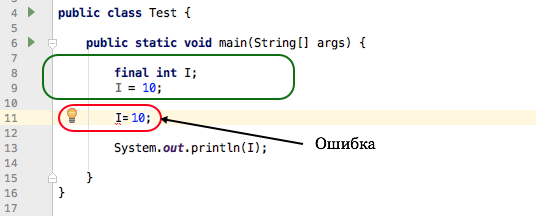


Тем не менее, вы не должны сразу задавать значение переменной. **Суть в том, что первое заданное значение меняться не будет.** Например, такой код будет работать:

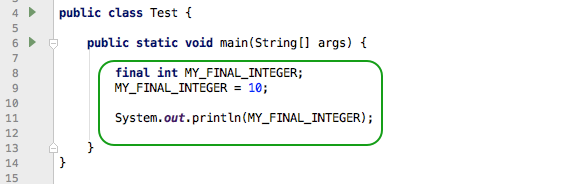
|  |  |
| --- | --- |
| *1*  *2*  *3*  *4*  *5*  *6*  *7*  *8*  *9*  *10*  *11* | *public class Test {*    *public static void main(String[] args) {*    *final int I;*  *I = 10;*    *System.out.println(I);*    *}*  *}* |



И этот тоже



**\*Обратите внимание:** переменные с final - это константы. При этом их принято писать **заглавными буквами** - тут [CamelStyle](https://vertex-academy.com/tutorials/ru/chto-takoe-camelstyle/)не работает :



***6. Что такое статические поля, статические финальные поля и статические методы. К чему имеют доступ статические методы? Можно ли перегрузить и переопределить статические методы? Наследуются ли статические методы?***

**Обычно при создании класса вы описываете, как объекты этого класса ведут себя и как они выглядят. Объект появляется только после того, как он будет создан ключевым словом new, и только начиная с этого момента для него выделяется память и появляется возможность вызова методов.**

**Но есть две ситуации, в которых такой подход недостаточен. Первая — это когда некоторые данные должны храниться «в единственном числе» независимо от того, сколько было создано объектов класса. Вторая — когда вам потребуется метод, не привязанный ни к какому конкретному объекту класса (то есть метод, который можно вызвать даже при полном отсутствии объектов класса). Такой эффект достигается использованием ключевого слова static, делающего элемент класса статическим. Когда вы объявляете что-либо как static, это означает, что данные или метод не привязаны к определенному экземпляру этого класса. Поэтому, даже если вы никогда не создавали объектов класса, вы можете вызвать статический метод или получить доступ к статическим данным. С обычным объектом вам необходимо сначала создать его и использовать для вызова метода или доступа к информации, так как нестатические данные и методы должны точно знать объект, с которым работают.**

**Некоторые объектно-ориентированные языки используют термины данные уровня класса и методы уровня класса, подразумевая, что данные и методы существуют только на уровне класса в целом, а не для отдельных объектов этого класса. Иногда эти термины встречаются в литературе по Java.**

**Чтобы сделать данные или метод статическими, просто поместите ключевое слово static перед их определением. Например, следующий код создает статическое поле класса и инициализирует его:**

**class StaticTest {**

**static int i =47;**

**}**

**Теперь, даже при создании двух объектов StaticTest, для элемента StaticTest.i выделяется единственный блок памяти. Оба объекта совместно используют одно значение i. Пример:**

**StaticTest stl = new StaticTestО;**

**StaticTest st2 = new StaticTestO;**

**В данном примере как stl.i, так и st2.i имеют одинаковые значения, равные 47, потому что расположены они в одном блоке памяти.**

**Существует два способа обратиться к статической переменной. Как было видно выше, вы можете указать ее с помощью объекта, например st2.i. Также можно обратиться к ней прямо по имени класса (для нестатических членов класса такая возможность отсутствует):**

**StaticTest.i++,**

**Оператор ++ увеличивает значение на единицу (инкремент). После выполнения этой команды значения stl.i и st2.i будут равны 48.**

**Синтаксис с именем класса является предпочтительным, потому что он не только подчеркивает, что переменная описана как static, но и в некоторых случаях предоставляет компилятору больше возможностей для оптимизации.**

**Та же логика верна и для статических методов. Вы можете обратиться к такому методу или через объект, как это делается для всех методов, или в специальном синтаксисе имяКласса.метод(). Статические методы определяются по аналогии со статическими данными:**

**class Incrementable {**

**static void increment ) { StaticTest.i++; }**

**}**

**Нетрудно заметить, что метод increment() класса Incrementable увеличивает значение статического поля i. Метод можно вызвать стандартно, через объект:**

**Incrementable sf = new IncrementableO.**

**sf.incrementO;**

**Или, поскольку increment() является статическим, можно вызвать его с прямым указанием класса:**

**Incrementable.increment();**

**Применительно к полям ключевое слово static радикально меняет способ определения данных: статические данные существуют на уровне класса, в то время как нестатические данные существуют на уровне объектов, но в отношении изменения не столь принципиальны. Одним из важных применений static является определение методов, которые могут вызываться без объектов. В частности, это абсолютно необходимо для метода main(), который представляет собой точку входа в приложение.**

**В Java смысл ключевого слова final зависит от контекста, но в основном оно означает: «Это нельзя изменить». Запрет на изменения может объясняться двумя причинами: архитектурой программы или эффективностью. Эти две причины основательно различаются, поэтому в программе возможно неверное употреб­ление ключевого слова final.**

**В следующих разделах обсуждаются три возможных применения final: для данных, методов и классов.**

**Неизменные данные**

**Во многих языках программирования существует тот или иной способ сказать компилятору, что частица данных является «константой». Константы полезны в двух ситуациях:**

**• константа времени компиляции, которая никогда не меняется;**

**• значение, инициализируемое во время работы программы, которое нельзя изменять.**

**Компилятор подставляет значение константы времени компиляции во все выражения, где оно используется; таким образом предотвращаются некоторые издержки выполнения. В Java подобные константы должны относиться к примитивным типам, а для их определения используется ключевое слово final. Значение такой константы присваивается во время определения.**

**Поле, одновременно объявленное с ключевыми словами static и final, существует в памяти в единственном экземпляре и не может быть изменено.**

**При использовании слова final со ссылками на объекты его смысл не столь очевиден. Для примитивов final делает постоянным значение, но для ссылки на объект постоянной становится ссылка. После того как такая ссылка будет связана с объектом, она уже не сможет указывать на другой объект. Впрочем, сам объект при этом может изменяться; в Java нет механизмов, позволяющих сделать произвольный объект неизменным. (Впрочем, вы сами можете напи­сать ваш класс так, чтобы его объекты фактически были константными.) Данное ограничение относится и к массивам, которые тоже являются объектами.**

**Модификатор static для методов**

***1. Метод вызывается без создания объекта класса.***

**Как и в случае с полями, статические методы можно вызывать без создания объекта. Если обращаться к статическим методам и через название объекта, и название класса, код будет работать. К нестатическим методам нужно обращаться исключительно через название объектов класса.**

***2. Статические методы нельзя переопределять.***

**Как Вы помните, один из фундаментальных**[**принципов ООП**](https://vertex-academy.com/tutorials/ru/chto-takoe-oop/)**- это "наследование". Дело в том, что в случаях, когда нам нужно создать новый класс, который имеет много общих свойств с каким-то уже существующим, вместо того, чтобы писать все заново, можно сделать "наследника" существующего класса. Этот "наследник" будет иметь те же самые метод и переменные, что и "родитель".**

**Тем не менее, в классе-наследнике обычно можно переопределять существующие методы. Это значит, что можно создать метод с таким же названием, только заменить его "внутренности". Так вот, статический метод нельзя переопределить. По аналогии с переменными, можно сказать, что этот метод "один для класса и его наследников" - так же, как статическая переменная "одна для класса и всех его объектов".**

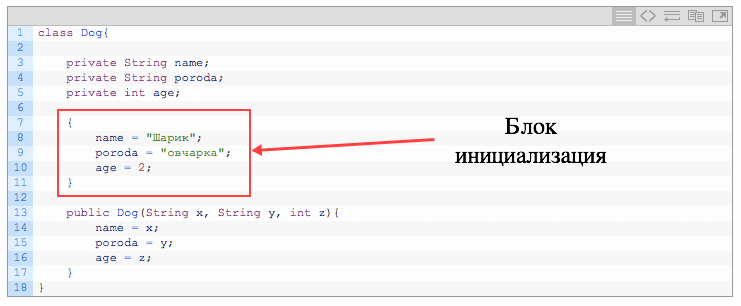
***3. Статическим методам нужен "статический контекст".***

**Есть такое правило: статический метод не может ссылаться на нестатическую переменную. Что это значит?**

**Представьте, что у нас в каком-то классе есть статический метод. То есть это метод, к которому, как Вы знаете, можно обращаться без создания объекта класса. Это значит, что если статический метод будет обращаться к нестатическим переменным (которые попросту "не будут существовать", потому что объект не объявлен), то возникнет ошибка. Поэтому, статические методы могут ссылаться только на статические переменные. Это гарантирует, что во время выполнения нашего метода все элементы будут инициализированы и будут работать. Именно это и называется "статическим контекстом".**

***7. Что такое логические и статические блоки инициализации? Сколько их может быть в классе, в каком порядке они могут быть размещены и в каком порядке вызываются?***

**Инициализация с помощью блоков**



**Виды блоков инициализации**

**Существует всего два типа блоков:**

**- нестатический (instance initializer)**

**- статический (class initializer)**

**В предыдущем примере Вы видели первый тип - нестатический блок инициализации. Его общую форму можно показать так:**

|  |  |
| --- | --- |
| **1**  **2**  **3** | **{**  **//...**  **}**  **Динамический (нестатический) блок представляет собой дополнение к конструктору. Он ставится между определениями полей и функциями класса. Команды будут выполняться при создании объекта. Динамический блок — это добавка для упрощения написания конструктора, и он не приносит дополнительной функциональности. Он позволяет сэкономить создание функции запуска и добавление её вызова из всех конструкторов. Например, фрагмент кода:**  1**public** **class** **Car** {  2 **static** int count = 0;  3 **public** Car (String model) {  4 init();  5 *// ...*  6 }  7 **public** Car (String model, Double price) {  8 init();  9 *// ...*  10 }  11  12 **private** void init() {  13 count++;  14 System.out.println("Hello everyone, we have " + count + " cars now!");  15 }  16 *// ...*  17}  **равнозначен коду:**  1**public** **class** **Car** {  2 **static** int count = 0;  3 **public** Car (String model) {  4 *// ...*  5 }  6 **public** Car (String model, Double price) {  7 *// ...*  8 }  9  10 {  11 count++;  12 System.out.println("Hello everyone, we have " + count + " cars now!");  13 }  14 *// ...*  15} |

**Второй тип - статический блок. Внешне он совсем немного отличается от первого типа - только нужно дописать слово static:**

**static {**

**//...**

**}**

**Как можно догадаться по названию: статический блок используется для инициализации статических переменных, а "обычный" - для всех остальных.**

**Статический блок — это, в сущности, конструктор для всего класса. Он ставится между определениями полей и функциями класса. Команды будут выполняться в одном из двух случаев, том, который наступит раньше:**

1. **При создании первого объекта класса в процессе работы программы, перед запуском конструктора.**
2. **При первом вызове статической функции, перед выполнением.**

**То есть код выполняется при первой загрузке класса.**

**Зачем используются блоки инициализации**

**Красивый и читабельный код - это конечно хорошо. Но зачем они нужны? Есть ли другие преимущества у блоков инициализации?**

**Есть. И это - больший функционал.**

**Внутри блоков инициализации мы можем не только присваивать значения. Это как метод - тут можно писать любые команды. Например, вывод в консоль:**

**class Dog{**

**private String name;**

**private String poroda;**

**private int age;**

**{**

**System.out.println("Это нестатический блок инициализации!");**

**}**

**public Dog(String x, String y, int z){**

**name = x;**

**poroda = y;**

**age = z;**

**}**

**}**

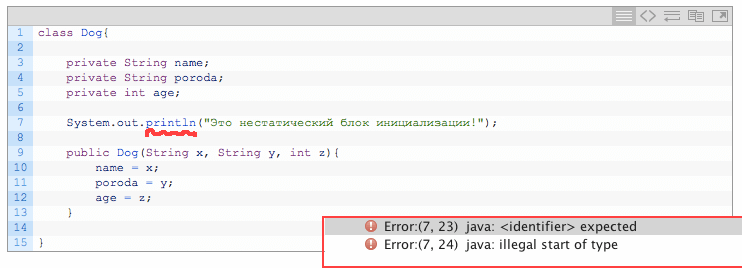
**Теперь в нашем блоке инициализации есть только System.out.println("Это нестатический блок инициализации!"). Запустим его со следующим main-ом:**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6** | **public class Test {**  **public static void main(String[] args) {**  **Dog d = new Dog("Шарик", "овчарка", 2);**  **}**  **}** |

**И получим в консоли:**

**https://vertex-academy.com/tutorials/wp-content/uploads/2018/03/Capture-d%E2%80%99%C3%A9cran-2018-03-05-%C3%A0-14.11.14.png**

**Как видите, работает. Если бы мы просто написали System.out.println, без блока инициализации, программа бы не запустилась:**

****

**Подводим итог:**

**- блоки инициализации используются для инициализации переменных внутри класса;**

**- есть два типа блоков инициализации - статический и нестатический;**

**- синтаксис очень простой - просто пишем выражение внутри блока из двух скобок:**

**- статический блок инициализации используется для статических переменных, нестатический - для всех остальных;**

**- блоки инициализации делают код читабельнее, и позволяют вызывать любые методы.**

**При разработке языка Java был установлен постоянный порядок действий при загрузке. Во время загрузки класса порядок выглядит следующим образом:**

1. **Определения статических полей родительских классов.**
2. **Инициализация статических полей и выполнение статических блоков родительских классов.**
3. **Определения статических полей класса.**
4. **Инициализация статических полей и выполнение статических блоков класса.**

**Затем, при создании объекта, порядок выглядит следующим образом:**

1. **Определения полей объекта из родительских классов.**
2. **Инициализация полей и выполнение динамических блоков из родительских классов.**
3. **Выполнение конструкторов из родительских классов.**
4. **Определения полей объекта из его класса.**
5. **Инициализация полей и выполнение динамических блоков из его класса.**
6. **Выполнение конструктора из его класса.**

**Когда существует цепочка предков, все действия выполняются сначала для самого дальнего предка (класс Object), а затем вниз по цепочке в том же порядке до текущего класса.**

**При наличии более чем одного типа в одном и том же разделе выше, действия выполняются в порядке появления в программе.**

***8. Что представляют собой методы с переменным числом параметров, как передаются параметры в такие методы и что представляет собой такой параметр в методе? Как осуществляется выбор подходящего метода, при использовании перегрузки для методов с переменным числом параметров?***

**В языке Java существуют методы, которые могут принимать переменное количество аргументов. Они называются методами с *аргументами переменной длины* (*var-args*).**

**Для указания аргументов переменной длины служат три точки. Например:**

***static void test(int... array)***

**Наряду с параметром переменной длины у метода могут быть и "обычные" параметры. Но параметр переменной длины должен быть последним среди всех параметров, объявляемых в методе. Например:**

***static void test(double d, int... array)***

**Метод может содержать только один параметр с переменным количеством аргументов.**

**Применение аргументов переменной длины**

**В следующем примере показан метод test(), объявленный с переменным количеством аргументов типа int. Внутри метода мы обращаемся к этим переменным как к массиву. При вызове этого метода можно передать любое, даже нулевое количество аргументов, а также массив:**

public class VarArgs {

static void test(int... array) {

System.out.println("Количество аргументов: " + array.length);

for (int a : array) {

System.out.print(a + " ");

}

System.out.println();

}

public static void main(String[] args) {

test();

test(1);

test(1, 2);

test(new int[]{1, 3});

}

}

**Методы с переменным числом аргументов можно перегружать. Рассмотрим пример:**

**Перегрузка методов с аргументами переменной длины**

**При вызове метода без аргументов, подходят два метода -*test(double... array)* и test(int... array)*.*В этом случае будет вызван метод с меньшим диапазоном значений - test(int... array)*.***

**При вызове метода test() с одним значением типа int*-*test(3)*,*будет выбран метод test(int a)*.***

public class VarArgs2 {

static void test(double... array) {

System.out.println("test(double... array)");

System.out.println("Количество аргументов: " + array.length);

for (double a : array) {

System.out.print(a + " ");

}

System.out.println();

}

static void test(int... array) {

System.out.println("test(int... array)");

System.out.println("Количество аргументов: " + array.length);

for (int a : array) {

System.out.print(a + " ");

}

System.out.println();

}

static void test(int a) {

System.out.println("test(int a)");

}

public static void main(String[] args) {

test();

test(3);

test(1.0);

test(1, 2);

}

}

Результат выполнения:

test(int... array)

Количество аргументов: 0

test(int a)

test(double... array)

Количество аргументов: 1

1.0

test(int... array)

Количество аргументов: 2

1 2

**При перегрузке метода, принимающего аргументы переменной длины, могут происходить непредвиденные ошибки. Они связаны с неоднозначностью, которая может возникать при вызове перегружаемого метода с аргументами переменной длины.**

**В следующем примере метод test перегружен - один вариант принимает значения *var-ags* типа boolean, а второй тоже *var-ags,*но типа int. При вызове метода test() без аргументов, возникает ошибка неоднозначности - JVM не может выбрать необходимый метод.**

**Аргументы переменной длины и неоднозначность**

public class VarArgs3 {

static void test(boolean... array) {

System.out.println("test(boolean... array)");

System.out.println("Количество аргументов: " + array.length);

for (boolean a : array) {

System.out.print(a + " ");

}

System.out.println();

}

static void test(int... array) {

System.out.println("test(int... array)");

System.out.println("Количество аргументов: " + array.length);

for (int a : array) {

System.out.print(a + " ");

}

System.out.println();

}

public static void main(String[] args) {

// test();// ошибка неоднозначности

test(3);

test(1, 2);

}

}

***9. Чем является класс Object? Перечислите известные вам методы класса  Object, укажите их назначение.***

**Хотя мы можем создать обычный класс, который не является наследником, но фактически все классы наследуются от класса Object. Все остальные классы, даже те, которые мы добавляем в свой проект, являются неявно производными от класса Object.**

**У класса есть несколько важных методов.**

* **Object clone() - создаёт новый объект, не отличающий от клонируемого**
* **boolean equals(Object obj) - определяет, равен ли один объект другому**
* **void finalize() - вызывается перед удалением неиспользуемого объекта**
* **Class<?> getClass() - получает класс объекта во время выполнения**
* **int hashCode() - возвращает хеш-код, связанный с вызывающим объектом**
* **void notify() - возобновляет выполнение потока, который ожидает вызывающего объекта**
* **void notifyAll() - возобновляет выполнение всех потоков, которые ожидают вызывающего объекта**
* **String toString() - возвращает строку, описывающий объект**
* **void wait() - ожидает другого потока выполнения**
* **void wait(long millis) - ожидает другого потока выполнения**
* **void wait(long millis, int nanos) - ожидает другого потока выполнения**

**Методы getClass(), notify(), notifyAll(), wait() являются финальными и их нельзя переопределять.**

**toString**

**Метод toString служит для получения представления данного объекта в виде строки. При попытке вывести строковое представления какого-нибудь объекта, как правило, будет выводиться полное имя класса. Например:**

public class Program{

    public static void main(String[] args) {

        Person tom = new Person("Tom");

        System.out.println(tom.toString()); // Будет выводить что-то наподобие Person@7960847b

    }

}

class Person {

    private String name;

    public Person(String name){

        this.name=name;

    }

}

**Полученное значение (в данном случае Person@7960847b) вряд ли может служить хорошим строковым описанием объекта. Поэтому метод toString() нередко переопределяют. Например:**

public class Program{

    public static void main(String[] args) {

        Person tom = new Person("Tom");

        System.out.println(tom.toString()); // Person Tom

    }

}

class Person {

    private String name;

    public Person(String name){

        this.name=name;

    }

    @Override

    public String toString(){

        return "Person " + name;

    }

}

**Метод hashCode**

**Метод hashCode позволяет задать некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты.**

**Например, выведем представление вышеопределенного объекта:**

Person tom = new Person("Tom");

System.out.println(tom.hashCode()); // 2036368507

**Но мы можем задать свой алгоритм определения хэш-кода объекта:**

class Person {

    private String name;

    public Person(String name){

        this.name=name;

    }

    @Override

    public int hashCode(){

         return 10 \* name.hashCode() + 20456;

    }

}

**Получение типа объекта и метод getClass**

**Метод getClass позволяет получить тип данного объекта:**

Person tom = new Person("Tom");

System.out.println(tom.getClass()); // class Person

**Метод equals**

**Метод equals сравнивает два объекта на равенство:**

public class Program{

    public static void main(String[] args) {

        Person tom = new Person("Tom");

        Person bob = new Person("Bob");

        System.out.println(tom.equals(bob)); // false

        Person tom2 = new Person("Tom");

        System.out.println(tom.equals(tom2)); // true

    }

}

class Person {

    private String name;

    public Person(String name){

        this.name=name;

    }

    @Override

    public boolean equals(Object obj){

        if (!(obj instanceof Person)) return false;

         Person p = (Person)obj;

        return this.name.equals(p.name);

    }

}

**Метод equals принимает в качестве параметра объект любого типа, который мы затем приводим к текущему, если они являются объектами одного класса.**

**Оператор instanceof позволяет выяснить, является ли переданный в качестве параметра объект объектом определенного класса, в данном случае класса Person. Если объекты принадлежат к разным классам, то их сравнение не имеет смысла, и возвращается значение false.**

**Затем сравниваем по именам. Если они совпадают, возвращаем true, что будет говорить, что объекты равны.**

***10. Что такое хэш-значение? Объясните, почему два разных объекта могут сгенерировать одинаковые хэш-коды?***

**Если очень просто, то хеш-код — это число. На самом деле просто, не так ли? Если более точно, то это битовая строка фиксированной длины, полученная из массива произвольной длины ([википедия](http://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).**

**Пример №1**

**Выполним следующий код:**

**public class Main {**

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

**Object object = new Object();**

**int hCode;**

**hCode = object.hashCode();**

**System.out.println(hCode);**

**}**

**}**

**В результате выполнения программы в консоль выведется целое 10-ти значное число. Это число и есть наша битовая строка фиксированной длины. В java она представлена в виде числа примитивного типа int, который равен 4-м байтам, и может помещать числа от -2 147 483 648 до 2 147 483 647. На данном этапе важно понимать, что хеш-код это число, у которого есть свой предел, который для java ограничен примитивным целочисленным типом int.**

**Вторая часть объяснения гласит:**

**полученная из массива произвольной длины.**

**Под массивом произвольной длины мы будем понимать объект. В 1 примере в качестве массива произвольной длины у нас выступает объект типа Object.**

**В итоге, в терминах Java, хеш-код — это целочисленный результат работы метода, которому в качестве входного параметра передан объект.**

**Этот метод реализован таким образом, что для одного и того-же входного объекта, хеш-код всегда будет одинаковым. Следует понимать, что множество возможных хеш-кодов ограничено примитивным типом int, а множество объектов ограничено только нашей фантазией. Отсюда следует утверждение: “Множество объектов мощнее множества хеш-кодов”. Из-за этого ограничения, вполне возможна ситуация, что хеш-коды разных объектов могут совпасть.**

**Здесь главное понять, что:**

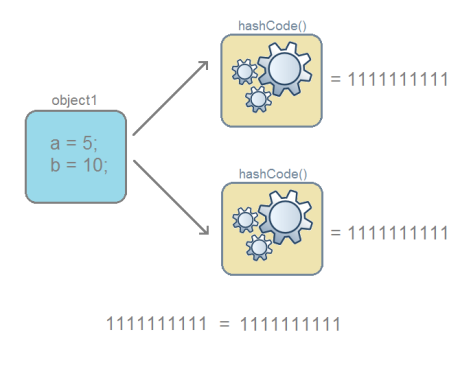
* **Если хеш-коды разные, то и входные объекты гарантированно разные.**
* **Если хеш-коды равны, то входные объекты не всегда равны.**

**Ситуация, когда у разных объектов одинаковые хеш-коды называется — коллизией. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода.**

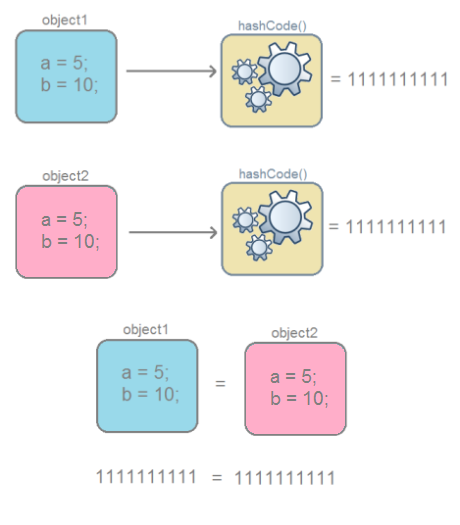
**Подведём итог:**

**Сперва, чтобы избежать путаницы, определимся с терминологией. Одинаковые объекты — это объекты одного класса с одинаковым содержимым полей.**

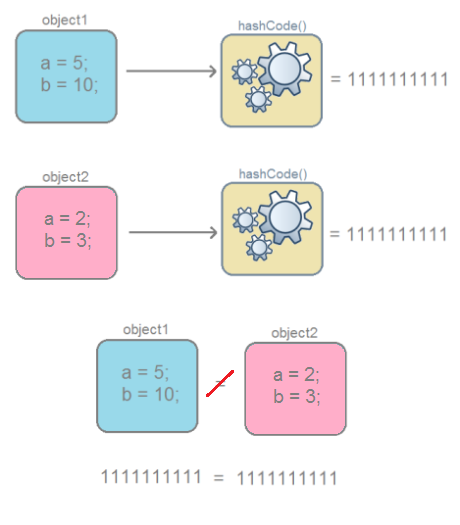
**для одного и того-же объекта, хеш-код всегда будет одинаковым;**

****

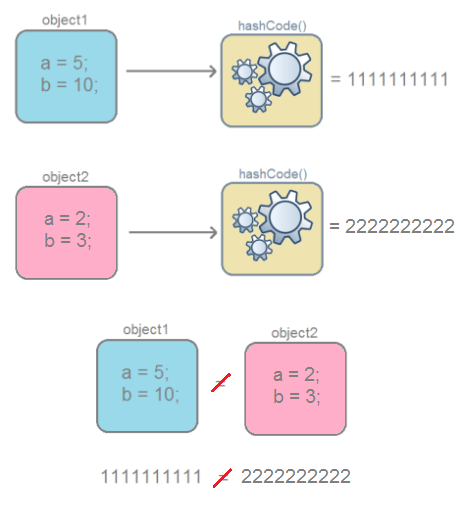
**если объекты одинаковые, то и хеш-коды одинаковые (но не наоборот, см. правило 3).**

****

**если хеш-коды равны, то входные объекты не всегда равны (коллизия);**

****

**если хеш-коды разные, то и объекты гарантированно разные;**

****

**11. Что такое объект класса Class? Чем использование метода getClass() и последующего сравнения возвращенного значения с Type.class отличается от использования оператора instanceof?**

**В Java почти все сущности являются объектами, за исключением примитивных типов. У каждого объекта есть класс. Сами классы тоже является объектами, и они принадлежат классу Class.**

**У класса Class нет публичных конструкторов. Class - это generic тип. Методы Class предназначены для получения информации о классе (объекте типа Class). Например, можно узнать полное имя класса, какие у него аннотации, какие конструкторы и т.п. Эти методы нужны для reflection. С помощью reflection вы можете создавать объекты, которые принадлежат этому классу, и при этом заранее класс объекта вы можете не знать.**

**Существуют библиотеки, которые позволяют создавать объекты типа Class "на лету", т.е. вы можете создать новый класс прямо во время работы программы и так же можете изменить существующий класс.**

**Класс Object , стоящий во главе иерархии классов Java, представляет все объекты, действующие в системе, является их общей оболочкой. Всякий объект можно считать экземпляром класса Object .**

**Класс с именем class представляет характеристики класса, экземпляром которого является объект. Он хранит информацию о том, не является ли объект на самом деле интерфейсом, массивом или примитивным типом, каков суперкласс объекта, каково имя класса, какие в нем конструкторы, поля, методы и вложенные классы.**

**В классе class нет конструкторов, экземпляр этого класса создается исполняющей системой Java во время загрузки класса и предоставляется методом getClass() класса object , например:**

**String s = "Это строка";**

**Class с = s.getClass();**

**Статический метод forName(string class) возвращает объект класса class для класса, указанного в аргументе, например:**

**Class cl = Class.forName("Java,lang.String");**

**Но этот способ создания объекта класса class считается устаревшим (deprecated). В новых версиях JDK для этой цели используется специальная конструкция — к имени класса через точку добавляется слово class :**

**Class c2 = Java.lang.String.class;**

**Логические методы isArray(), isIntetface(), isPrimitive() позволяют уточнить, не является ли объект массивом, интерфейсом или примитивным типом.**

**Если объект ссылочного типа, то можно извлечь сведения о вложенных классах, конструкторах, методах и полях методами getDeciaredClasses(), getDeclaredConstructors(), getDeclaredMethods(), getDeclaredFields() , в виде массива классов, соответствейно, Class, Constructor, Method, Field . Последние три класса расположены в пакете java.lang.reflect и содержат сведения о конструкторах, полях и методах аналогично тому, как класс class хранит сведения о классах.**

**Методы getClasses(), getConstructors(), getlnterfaces(), getMethods(), getFieids() возвращают такие же массивы, но не всех, а только открытых членов класса.**

**Метод getSuperClass() возвращает суперкласс объекта ссылочного типа, getPackage() — пакет, getModifiers() — модификаторы класса В битовой форме. Модификаторы можно затем расшифровать методами класса Modifier из пакета Java.lang.reflect .**

Чем использование метода getClass() и последующего сравнения возвращенного значения с Type.class отличается от использования оператора instanceof?

**instanceof проверяет, является ли ссылка на объект с левой стороны (LHS) экземпляром типа с правой стороны (RHS) *или каким-либо подтипом* .**

**getClass() == ...  проверяет, идентичны ли типы.**

**12. Укажите правила переопределения методов equals(), hashCode() и toString().**

## Правила переопределения equals

Метод equals() необходим в Java для подтверждения или отрицания того факта, что два объекта одного происхождения являются логически равными. То есть, сравнивая два объекта, программисту необходимо понять, эквивалентны ли их значимые поля. Не обязательно все поля должны быть идентичны, так как метод equals() подразумевает именно логическое равенство. Но иногда нет особой необходимости в использовании этого метода. Как говорится, самый легкий путь избежать проблем, используя тот или иной механизм — не использовать его. Также следует заметить, что однажды нарушив контракт equals вы теряете контроль над пониманием того, как другие объекты и структуры будут взаимодействовать с вашим объектом. И впоследствии найти причину ошибки будет весьма затруднительно.

### Когда не стоит переопределять этот метод

* **Когда каждый экземпляр класса является уникальным.**

В большей степени это касается тех классов, которые предоставляют определенное поведение, нежели предназначены для работы с данными. Таких, например, как класс Thread. Для них реализации метода equals, предоставляемого классом Object, более чем достаточно. Другой пример — классы перечислений (Enum).

* **Когда на самом деле от класса не требуется определять эквивалентность его экземпляров.**

Например для класса java.util.Random вообще нет необходимости сравнивать между собой экземпляры класса, определяя, могут ли они вернуть одинаковую последовательность случайных чисел. Просто потому, что природа этого класса даже не подразумевает такое поведение.

* **Когда класс, который вы расширяете, уже имеет свою реализацию метода equals и поведение этой реализации вас устраивает.**

Например, для классов Set, List, Map реализация equals находится в AbstractSet, AbstractList и AbstractMap соответственно.

* **И, наконец, нет необходимости перекрывать equals, когда область видимости вашего класса является private или package-private и вы уверены, что этот метод никогда не будет вызван.**

### Контракт equals

При переопределении метода equals разработчик должен придерживаться основных правил, определенных в спецификации языка Java.

* **Рефлексивность**

для любого заданного значения x, выражение x.equals(x) должно возвращать true.  
Заданного — имеется в виду такого, что x != null

* **Симметричность**

для любых заданных значений x и y, x.equals(y) должно возвращать true только в том случае, когда y.equals(x) возвращает true.

* **Транзитивность**

для любых заданных значений x, y и z, если x.equals(y) возвращает true и y.equals(z) возвращает true, x.equals(z) должно вернуть значение true.

* **Согласованность**

для любых заданных значений x и y повторный вызов x.equals(y) будет возвращать значение предыдущего вызова этого метода при условии, что поля, используемые для сравнения этих двух объектов, не изменялись между вызовами.

* **Сравнение null**

для любого заданного значения x вызов x.equals(null) должен возвращать false.

### Нарушение контракта equals

Многие классы, например классы из Java Collections Framework, зависят от реализации метода equals(), поэтому не стоит им пренебрегать, т.к. нарушение контракта этого метода может привести к нерациональной работе приложения и в таком случае найти причину будет достаточно трудно. Согласно принципу **рефлексивности**, каждый объект должен быть эквивалентен самому себе. Если этот принцип будет нарушен, при добавлении объекта в коллекцию и при последующем поиске его с помощью метода contains() мы не сможем найти тот объект, который только что положили в коллекцию. Условие **симметричности** гласит, что два любых объекта должны быть равны независимо от того, в каком порядке они будут сравниваться. Например, имея класс, содержащий всего одно поле строкового типа, будет неправильно сравнивать в методе equals данное поле со строкой. Т.к. в случае обратного сравнения метод всегда вернет значение false.

Из условия **транзитивности** следует, что если любые два из трех объектов равны, то в таком случае должны быть равны все три. Этот принцип легко нарушить в том случае, когда необходимо расширить некий базовый класс, добавив к нему значимый компонент. Например, к классу Point с координатами x и y необходимо добавить цвет точки, расширив его. Для этого потребуется объявить класс ColorPoint с соответствующим полем color. Таким образом, если в расширенном классе вызывать метод equals родителя, а в родительском будем считать, что сравниваются только координаты x и y, тогда две точки разного цвета, но с одинаковыми координатами будут считаться равными, что неправильно. В таком случае, необходимо научить производный класс различать цвета. Для этого можно воспользоваться двумя способами. Но один будет нарушать правило симметричности, а второй — транзитивности.

И, наконец, следующее правило **согласованности** гласит, что если объекты x и y не меняются, повторный вызов x.equals(y) должен вернуть то же значение, что и ранее. Последнее правило заключается в том, что ни один объект не должен быть равен null. Здесь все понятно, null — это неопределенность, равен ли объект неопределенности? Непонятно, т. е. false.

### Общий алгоритм определения equals

1. Проверить на равенство ссылки объектов this и параметра метода o.  
   if (this == o) return true;
2. Проверить, определена ли ссылка o, т. е. является ли она null.  
   Если в дальнейшем при сравнении типов объектов будет использоваться оператор instanceof, этот пункт можно пропустить, т. к. этот параметр возвращает false в данном случае null instanceof Object.
3. Сравнить типы объектов this и o с помощью оператора instanceof или метода getClass(), руководствуясь описанием выше и собственным чутьем.
4. Если метод equals переопределяется в подклассе, не забудьте сделать вызов super.equals(o)
5. Выполнить преобразование типа параметра o к требуемому классу.
6. Выполнить сравнение всех значимых полей объектов:
   * для примитивных типов (кроме float и double), используя оператор ==
   * для ссылочных полей необходимо вызвать их метод equals
   * для массивов можно воспользоваться перебором по циклу, либо методом Arrays.equals()
   * для типов float и double необходимо использовать методы сравнения соответствующих оберточных классов Float.compare() и Double.compare()
7. И, наконец, ответить на три вопроса: является ли реализованный метод симметричным? Транзитивным? Согласованным? Два других принципа (рефлексивность и определенность), как правило, выполняются автоматически.

### Контракт hashCode

Для реализации хэш-функции в спецификации языка определены следующие правила:

* вызов метода hashCode один и более раз над одним и тем же объектом должен возвращать одно и то же хэш-значение, при условии что поля объекта, участвующие в вычислении значения, не изменялись.
* вызов метода hashCode над двумя объектами должен всегда возвращать одно и то же число, если эти объекты равны (вызов метода equals для этих объектов возвращает true).
* вызов метода hashCode над двумя неравными между собой объектами должен возвращать разные хэш-значения. Хотя это требование и не является обязательным, следует учитывать, что его выполнение положительно повлияет на производительность работы хэш-таблиц.

### Методы equals и hashCode необходимо переопределять вместе

Исходя из описанных выше контрактов следует, что переопределяя в своем коде метод equals, необходимо всегда переопределять и метод hashCode. Так как фактически два экземпляра класса отличаются, потому что находятся в разных областях памяти, сравнивать их приходится по некоторым логическим признакам. Соответственно, два логически эквивалентных объекта, должны возвращать одинаковое значение хэш-функции. **Что произойдет, если будет переопределен только один из этих методов?**

1. equals есть, hashCode нет

Допустим мы правильно определили метод equals в нашем классе, а метод hashCode решили оставить как он есть в классе Object. Тогда с точки зрения метода equals два объекта будут логически равны, в то время как с точки зрения метода hashCode они не будут иметь ничего общего. И, таким образом, помещая некий объект в хэш-таблицу, мы рискуем не получить его обратно по ключу.

1. hashCode есть, equals нет.

Что будет если мы переопределим метод hashCode, а реализацию метода equals унаследуем из класса Object. Как известно метод equals по умолчанию просто сравнивает указатели на объекты, определяя, ссылаются ли они на один и тот же объект. Предположим, что метод hashCode мы написали по всем канонам, а именно — сгенерировали средствами IDE, и он будет возвращать одинаковые хэш-значения для логически одинаковых объектов. Очевидно, что тем самым мы уже определили некоторый механизм сравнения двух объектов.

Следовательно, пример из предыдущего пункта по идее должен выполняться. Но мы по-прежнему не сможем найти наш объект в хэш-таблице. Хотя будем уже близки к этому, потому что как минимум найдем корзину хэш-таблицы, в которой объект будет лежать.

Для успешного поиска объекта в хэш-таблице помимо сравнения хэш-значений ключа используется также определение логического равенства ключа с искомым объектом. Т. е. без переопределения метода equals никак не получится обойтись.

## Общий алгоритм определения hashCode

Здесь, мне кажется, вообще не стоит сильно переживать и выполнить генерацию метода в своей любимой IDE. Потому что все эти смещения битов вправо, влево в поиске золотого сечения, т. е. нормального распределения — это для совсем упоротых чуваков. Лично я сомневаюсь, что смогу сделать лучше и быстрее, чем та же Idea.

## Как переопределить метод toString в Java:

Печать форматированной даты(т.е. день/месяц/год) вместо "сырого" значения. Это очень полезная подсказка при переопределении Java-метода toString. Обычный toString() класса java.util.Date не выводит форматированную дату и включает много деталей, которые не всегда нужны. Если вы используете частичный [DateFormat](https://javarevisited.blogspot.com/2011/09/convert-date-to-string-simpledateformat.html) т.е. dd-MM-yy в вашем приложении, то вы определенно хотели бы видеть этот формат вместо данного по умолчанию. IDE обычно не генерирует форматированный вывод Date и это то, что вам нужно сделать самому, но это того стоит. Вы так же можете использовать [SimpleDataFormat](https://javarevisited.blogspot.com/2012/03/simpledateformat-in-java-is-not-thread.html) класс или библиотеку Joda Date time для этого.

## Документирование формата toString

Если ваш метод toString() не выводит данные в виде поле=значение, то хорошей идеей будет документирование формата вывода toString, особенно для важных объектов типа Работник или Студент. К примеру если метод toString() класса Работник(Employee) печатает "John-101-Sales-9846387321" то хорошей практикой будет указать формат как "имя-ID-отдел-контакт", но в то же время не давайте клиенту возможность получать информацию из метода toString(), вы должны всегда предоставлять соответствующие методы для получения данных, такие как getName(), getId(), getContact() и так далее, поскольку информация полученная из toString() представления объекта является хрупкой и подвержена ошибкам, так что клиент всегда должен иметь чистый путь получения информации.

## Используйте StringBuilder для составления вывода toString()

Если вы пишете код для метода toString() в Java, тогда используйте [StringBuilder](https://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) что бы добавить отдельные атрибуты. Если вы используете IDE вроде [Eclipse](https://javarevisited.blogspot.com/2011/02/how-to-setup-remote-debugging-in.html), Netbeans или IntelliJ тогда использование StringBuilder и метода append() вместо оператора + для составления toString, так же является верным путем. По умолчанию и Eclipse и Netbeans генерируют toString с оператором конкатенации.

## Использование аннотации @Override

Использование @Override при переопределении метода в Java это одна из лучших практик в языке. Но эта подсказка не так важна, как это было бы в случае с переопределением методов equals() и compareTo(), поскольку перегрузка вместо переопределения может создать более тонкие, сложно вычисляемые, ошибки. В любом случае лучше использовать аннотацию @Overrride.

## Печать содержимого массива, вместо вывода объекта массива

Массив это объект в Java но он не переопределяет метод toString и когда вы печатаете массив, то по умолчанию формат вывода не особо полезен, поскольку мы хотели бы видеть содержимое массива. К слову это еще одна причина [почему char[] array предпочтительнее String](https://javarevisited.blogspot.com/2012/03/why-character-array-is-better-than.html) для хранения чувствительных данных, таких как пароль. Найдите время взглянуть помогает ли печать содержимого массива вашим клиентам или нет и если это имеет смысл, то печатайте содержимое массива вместо самого объекта. Из соображений производительности предпочтительнее использовать Collection такие как ArrayList или [HashSet](https://javarevisited.blogspot.com/2012/06/hashset-in-java-10-examples-programs.html) вместо Array для хранения других объектов.

## Бонусные подсказки

Еще несколько бонусных подсказок по переопределению метода toString в Java.

1. Печатать вывод toString в несколько строк или в одну основываясь на его длине.
2. Включать полные имена классов в представление toString, другими словами package.class что бы избежать любого непонимания.
3. Вы можете пропускать значения null или показывать их, но лучше показывать. Иногда они полезны, поскольку показывают какое поле является null во время какого-то инциндента, к примеру [NullPointerException](https://javarevisited.blogspot.com/2012/06/common-cause-of-javalangnullpointerexce.html" \t "_blank).
4. Используйте формат ключ-значение, к примеру member.name=member.value, большинство IDE поддерживают это.
5. Включайте унаследованных членов, если вы считаете, что они должны предоставлять необходимую информацию в класс-наследник.
6. Иногда объект содержит много необязательных и обязательных параметров, как мы показали в нашем примере [шаблона Builder](https://javarevisited.blogspot.com/2012/06/builder-design-pattern-in-java-example.html), когда практически невозможно распечатать все поля, в этом случае можно выводить только обязательные поля, тем более, что у нас есть необязательные.