**OOP, Classes and Objects, Inheritance, Class Object, Interfaces**

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().
13. Дайте развернутое объяснение трем концепциям ООП.
14. Приведите правила, которым должен следовать компонент java-bean.
15. Как вы думаете, для чего используется наследование классов в java-программе? Приведите пример наследования. Как вы думаете, поля и методы, помеченными модификатором доступа private, наследуются?
16. Укажите, как вызываются конструкторы при создании объекта производного класса? Что в конструкторе класса делает оператор super()? Возможно ли в одном конструкторе использовать операторы super() и this()?
17. Объясните, как вы понимаете утверждения: “ссылка базового класса может ссылаться на объекты своих производных типов” и “объект производного класса может быть использован везде, где ожидается объект его базового типа”. Верно ли обратное и почему?
18. Что такое переопределение методов? Как вы думаете, зачем они нужны? Можно ли менять возвращаемый тип при переопределении методов? Можно ли менять атрибуты доступа при переопределении методов? Можно ли переопределить методы в рамках одного класса?
19. Определите правило вызова переопределенных методов. Можно ли статические методы переопределить нестатическими и наоборот?
20. Какие свойства имеют финальные методы и финальные классы? Как вы думаете, зачем их использовать?
21. Укажите правила приведения типов при наследовании. Напишите примеры явного и неявного преобразования ссылочных типов. Объясните, какие ошибки могут возникать при явном преобразовании ссылочных типов.
22. Что такое абстрактные классы и методы? Зачем они нужны? Бывают ли случаи, когда абстрактные методы содержат тело? Можно ли в абстрактных классах определять конструкторы? Могут ли абстрактные классы содержать неабстрактные методы? Можно ли от абстрактных классов создавать объекты и почему?
23. Что такое интерфейсы? Как определить и реализовать интерфейс в java-программе? Укажите спецификаторы, которые приобретают методы и поля, определенные в интерфейсе. Можно ли описывать в интерфейсе конструкторы и создавать объекты? Можно ли создавать интерфейсные ссылки и если да, то на какие объекты они могут ссылаться?
24. Для чего служит интерфейс Clonable? Как правильно переопределить метод clone() класса Object, для того, что объект мог создавать свои адекватные копии?
25. Для чего служат интерфейсы Comparable и Comparator? В каких случаях предпочтительнее использовать первый, а когда – второй? Как их реализовать и использовать?

***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!*

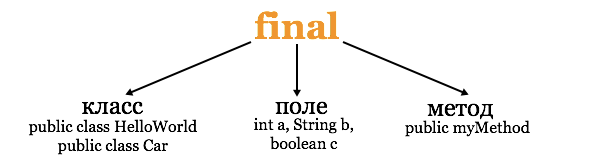
Как всегда, this связан с его экземпляром, и это не будет работать в статических методах.

***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.



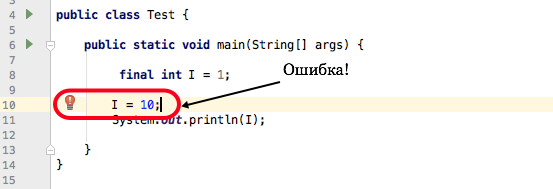
Если вы хотите, чтобы **после инициализации** никто **не мог бы изменить** вашу переменную, напишите слово "final":

|  |  |
| --- | --- |
| 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:

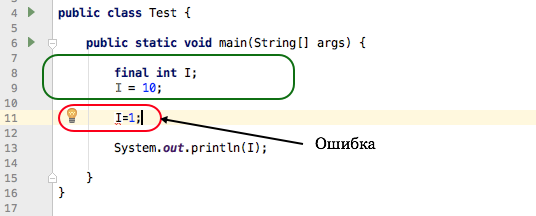
|  |  |
| --- | --- |
| 6  7  8  9  10 | *public class Test {*    *final int I = 1;*    *public static void main(String[] args) {*    *System.out.println(I);*    *}*  *}* |

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

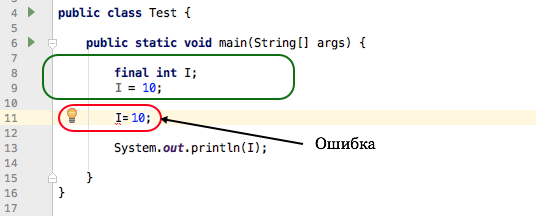


Тем не менее, вы не должны сразу задавать значение переменной. **Суть в том, что первое заданное значение меняться не будет.** Например, такой код будет работать:

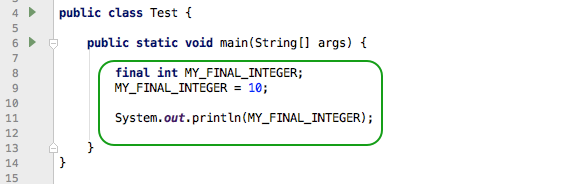
|  |  |
| --- | --- |
| *1*  *2*  *3*  *4*  *5*  *6*  *7*  *8*  *9*  *10*  *11* | *public class Test {*    *public static void main(String[] args) {*    *final int I;*  *I = 10;*    *System.out.println(I);*    *}*  *}* |



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".**

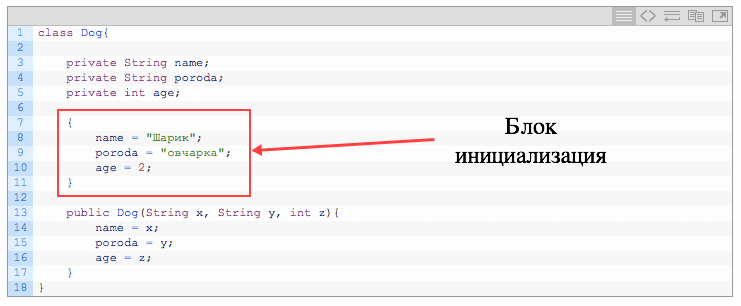
***3. Статическим методам нужен "статический контекст".***

**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:**

|  |
| --- |
| **{**  **//...**  **}**  **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;**

**}**

**}**

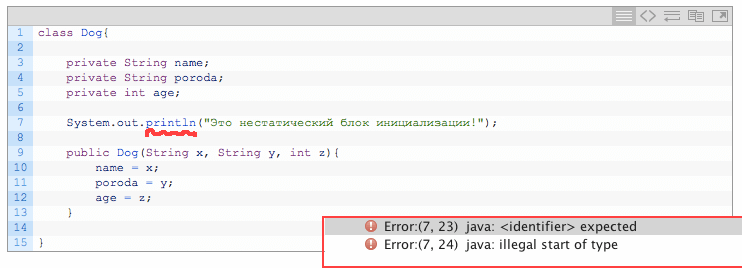
**Теперь в нашем блоке инициализации есть только System.out.println("Это нестатический блок инициализации!"). Запустим его со следующим main-ом:**

|  |  |
| --- | --- |
| **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:**

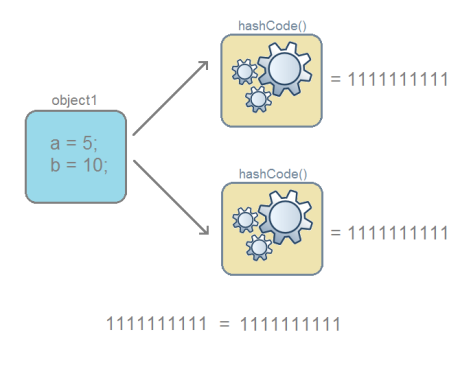
* **Если хеш-коды разные, то и входные объекты гарантированно разные.**
* **Если хеш-коды равны, то входные объекты не всегда равны.**

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

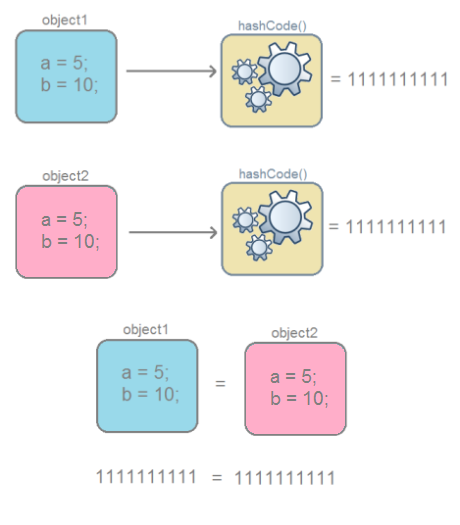
**Подведём итог:**

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

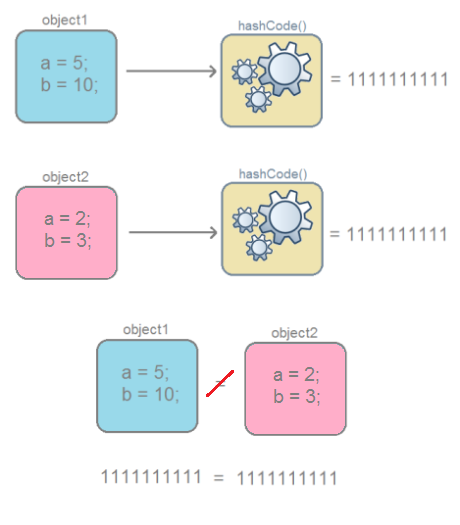
**для одного и того-же объекта, хеш-код всегда будет одинаковым;**

****

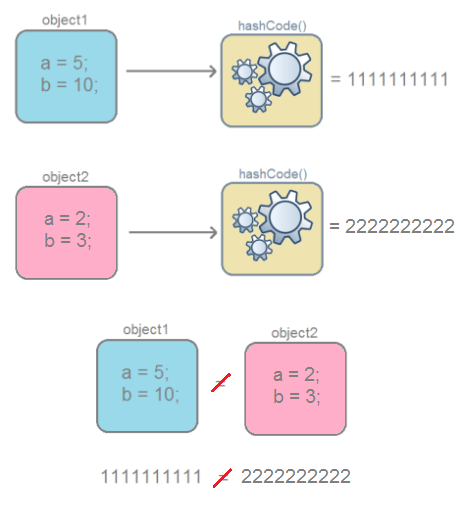
**если объекты одинаковые, то и хеш-коды одинаковые (но не наоборот, см. правило 3).**

****

**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.

## 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.

## 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).
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, в этом случае можно выводить только обязательные поля, тем более, что у нас есть необязательные.

**13. Дайте развернутое объяснение трем концепциям ООП.**

JAVA основывается на концепциях объектно-ориентированного программирования, что позволяет перейти на более высокий уровень абстракции, чтобы разрешить любую проблему реалистичным путем. Объектно-ориентированный подход концептуализирует решение проблемы в плоскости объектов реального мира, которые легче повторно использовать в приложении. В JAVA класс представляет собой макет, шаблон или прототип, который определяет общее поведение объекта данного типа. Экземпляр — это отдельная реализация класса, и все экзепляры класса имеют одинаковые свойства, которые описаны в определении класса. Например, вы можете опрделить класс с именем House (дом) с количеством комнат в качестве атрибута и создать экземпляры класса, такие как дом с двумя комнатами, дом с тремя комнатами и так далее.

**Преимущества:** Ниже перечислены некоторые плюсы объектно-ориентированной разработки программного обеспечения (ПО).

* Снижение затрат на поддержку ПО, в основном за счет того, что она осуществляется модульно.
* Усовершенствованное повторное использование кода благодаря таким качествам, как наследование, и, как результат, более быстрая разработка ПО.
* Повышенные надежность и гибкость кода.
* Легкость понимания вследствие моделирования реального мира.
* Лучшая абстракция на уровне объекта.
* Уменьшение сложности перехода от одной фазы разработки к другой.

Есть четыре основные характеристики ООП:

* Инкапсуляция
* Наследование
* Полиморфизм
* Абстракция

**ИНКАПСУЛЯЦИЯ**

Инкапсуляция выступает договором для объекта, что он должен скрыть, а что открыть для доступа другими объектами. В JAVA мы используем модификатор доступа private для того, чтобы скрыть метод и ограничить доступ к переменной из внешнего мира. JAVA также располагает различными модификаторами доступа: public, по умолчанию, protected, private, которые используются для ограничения видимости на разных уровнях. Но конечной целью является инкапсуляция тех вещей, которые не должны быть изменены. Лучше всего работает подход, при котором, у класса должна быть только одна причина для изменения, и инкапсулирование воплощает в реальность проектирование этой “одной причины”. Правильным в инкапсуляции считается сокрытие часто изменяющихся вещей во избежание повреждения других классов.

**Преимущества:** Ниже представлены некоторые преимущества инкапсуляции:

* Мы можем защитить внутреннее состояние объекта с помощью сокрытия его атрибутов.
* Это улучшает модульное построение кода, так как предотвращает взаимодействие объектов неожиданными способами.
* Повышается практичность кода.
* Это поддерживает договорные отношения конкретного объекта.
* Инкапсуляция облегчает поддержку ПО.
* Изменения в коде могут производиться независимо друг от друга.

**ПОЛИМОРФИЗМ**

Полиморфизм в программировании — это способность предоставлять один и тот же интерфейс для различных базовых форм (типов данных). Это означает, что классы, имеющие различную функциональность, совместно используют один и тот же интерфейс и могут быть динамически вызваны передачей параметров по ссылке. Классический пример — это класс Shape (фигура) и все классы, наследуемые от него: square (квадрат), circle (круг), dodecahedron (додекаэдр), irregular polygon (неправильный многоугольник), splat (клякса) и так далее. В этом примере каждый класс будет иметь свой собственный метод Draw() и клиентский код может просто делать:

Shape shape = **new** Shape();

Shape.area() чтобы получить корректное поведение любой фигуры Красота полиморфизма заключается в том, что код, работая с различными классами, не должен знать, какой класс он использует, так как все они работают по одному принципу. Процесс, применяемый объектно-ориентированными языками программирования для реализации динамического полиморфизма, называется динамическим связыванием. *Примечание:* Полиморфизм — это способность выбирать более конкретные методы для исполнения в зависимости от объекта. Полиморфизм осуществляется тогда, когда не задействованы абстракные классы.

**Преимущества:**

* Создание повторно используемого кода. То есть, как только класс создан, реализован и протестирован, он может свободно использоваться без заботы о том, что конкретно в нем написано.
* Это обеспечивает более универсальный и слабосвязанный код.
* Понижается время компиляции, что ускоряет разработку.
* Динамическое связывание.
* Один и тот же интерфейс может быть использован для создания методов с разными реализациями.
* Вся реализация может быть заменена с помощью использования одинаковых сигнатур метода.

**Переопределение методов как часть полиморфизма.** Переопределение взаимодействует с двумя методами: методом родительского класса и методом производного класса. Эти методы имеют одинаковые имя и сигнатуры. Переопределение позволяет вам производить одну и ту же операцию различными путями для разных типов объектов. Например:

**while**(it.hasNext()) {

Shape s = (Shape) it.next();

totalArea += s.area(dim); //будет применен полиморфизм и вызван нужный метод для каждого объекта.

}

**Перезагрузка методов или ad-hoc полиморфизм или статический полиморфизм** Перезагрузка взаимодействует с несколькими методами одного класса, которые одинаково названы, но имеют разные сигнатуры методов. Перезагрузка позволяет вам описать одну и ту же операцию различными путями для разных данных. Иногда ее называют статическим полиморфизмом, но фактически полиморфизмом она не является. Это ничто иное, как просто наличие двух методов с одинаковыми именами, но разным списком аргументов. Перезагрузка не имеет ничего общего с наследованием и полиморфизмом. И перезагруженный метод совсем не то же самое, что переопределенный метод.

**Параметрический полиморфизм через обобщение в JAVA** При объявлении класса поле имени может ассоциироваться с различными типами, а имя метода может ассоциироваться с различными параметрами и возвращаемыми типами. JAVA поддерживает параметрический полиморфизм, применяя обобщение (дженерики).

List<String> list = **new** ArrayList<String>();

**Почему мы не можем переопределить статический метод в JAVA?** Переопределение зависит от наличия экземпляра класса. Идея полиморфизма состоит в том, что вы можете создать подкласс, и объекты, реализуемые теми подклассами, будут вести себя по-другому с теми же методами родителького класса (переопределенными в подклассах). Статический метод не ассоциируется ни к каким экземпляром класса, таким образом, сама концепция переопределения не может быть применена. Создателями JAVA руководили два соображения, которые повлияли на такой подход. Во-первых, это проблемы исполнения кода: лилось очень много критики в адрес Smalltalk из-за медленной работы (сборщик мусора и полиморфизм были частью этой проблемы), и в проектировании JAVA старались этого избежать. Вторым соображением было решение, что целевой аудиторией JAVA станут С++ разработчики. То, что статические методы работают именно таким образом, было очень знакомо C++ программистам, а так же ускоряло работу, так как не было необходимости проходить вверх по иерархии классов, чтобы выяснить, какой метод вызывать. Вы идете прямо к классу и вызываете конкретный метод.

**НАСЛЕДОВАНИЕ**

Наследование — это включение поведения (т.е. методов) и состояния (т.е. переменных) базового класса в производный класс, таким образом они становятся доступны в этом производном классе. Главное преимущество наследования в том, что оно обеспечивает формальный механизм повторного использования кода и избегает дублирования. Унаследованный класс расширяет функциональность приложения благодаря копированию поведения родительского класса и добавлению новых функций. Это делает код сильно связанным. Если вы захотите изменить суперкласс, вам придется знать все детали подклассов, чтобы не разрушить код. Наследование — это форма повторного использования программного обеспечения, когда из уже существующего класса (суперкласса) создается новый класс (подкласс), который расширяет свою функциональность и при этом использует некоторые свойства суперкласса. Так что, если у вас есть класс-родитель, а потом появляется класс-наследник, то наследник наследует все вещи, которыми обладает родитель.

**Преимущества:**

* Усовершенствованное повторное использование кода.
* Устанавливается логическое отношение «is a» (является кем-то, чем-то). Например: Dog is a*n* animal. (Собака является животным).
* Модуляризация кода.
* Исключаются повторения.

**Недостаток:**

* **Сильная связанность:** подкласс зависит от реализации родительского класса, что делает код сильно связанным.

**АБСТРАКЦИЯ**

Абстракция означает разработку классов исходя из их интерфейсов и функциональности, не принимая во внимание реализацию деталей. Абстрактный класс представляет собой интерфейсы без включения фактической реализации. Он отличает реализацию объекта от его поведения. Абстракция упрощает код, скрывая несущественные детали.

**Преимущества:**

* Применяя абстракцию, мы можем отделить то, что может быть сгруппировано по какому-либо типу.
* Часто изменяемые свойства и методы могут быть сгруппированы в отдельный тип, таким образом основной тип не будет подвергаться изменениям. Это усиливает принцип ООП: *«Код должен быть открытым для Расширения, но закрытым для Изменений»*.
* Абстракция упрощает представление доменных моделей.

**Отличие между абстракцией и инкапсуляцией** Инкапсуляция — это стратегия, используемая как часть абстракции. Инкапсуляция относится к структуре объекта: объекты инкапсулируют свои свойства и скрывают их от доступа извне. Пользователи класса взаимодействуют с ним с помощью его методов, но не имеют доступа напрямую к структуре класса. Таким образом класс абстрагирует детали реализации, относящиеся к его строению. Абстракция является более общим термином. Она также может достигаться среди прочего с помощью подклассов. Например, класс List (список) в стандартной библиотеке является абстракцией для последовательности элементов, проиндексированных согласно их места в списке. Конкретными примерами списка List являются ArrayList или LinkedList. Код, который взаимодействует со списком List абстрагируется от деталей, какой именно список он использует. Часто абстракция невозможна без сокрытия основного состояния с помощью инкапсуляции. Если класс раскрывает свою внутреннюю структуру, он не может изменить свои внутренние операции, а, следовательно, не может абстрагироваться.

**Что такое абстрактный класс и абстрактный метод?** Случается, что во время разработки вы хотите, чтобы базовый класс представлял только интерфейс для его производных классов. То есть вы не хотите, чтобы кто-либо создавал экземпляры базового класса. Вам необходимо использовать интерфейс таким образом, чтобы только приводить объекты к нему (это неявное приведение, которое обеспечивает полиморфное поведение). Это достигается путем создания данного класса абстрактным с помощью ключевого слова abstract. Это накладывает некоторые ограничения, такие как невозможность создавать экземпляры абстрактного класса, при использовании абстрактного класса необходимо реализовывать абстрактные методы. Этим обеспечивается полиморфизм. Абстрактный класс может содержать и абстрактные и конкретные методы. Если хоть один метод в классе объявлен абстрактным, весь класс должен так же быть объявлен абстрактным. Тем не менее, в обратную сторону правило не обязано соблюдаться. Если класс объявлен абстрактным, он может и не содержать абстрактные методы. Метод, который всего лишь определяет свои сигнатуры и не обеспечивает реализацию, называется абстрактным. Фактическая его реализация оставлена его подклассам, которые расширяют абстрактный класс. Абстрактный метод не может быть использован объектом, только другой класс может его расширить.

**Когда необходимо использовать абстрактный класс?** Абстрактные классы позволяют вам определить некоторое поведение по умолчанию и заставить подклассы обеспечить любое конкретное поведение. Например: List (список) является интерфейсом, в свою очередь AbstractList определяет основное поведение Списка, которое может быть использовано как есть или уточнено в подклассе, например, в ArrayList (списочный массив).

**Что такое интерфейс?** В концепции интерфейса лежит абстрактный класс, но интерфейс (определяется ключевым словом interface) шагнул дальше. Он предотвращает вообще любую реализацию метода или функции. Вы можете только объявлять метод или функцию, но не обеспечивать их реализацию. Класс, который реализует данный интерфейс, должен как раз и позаботиться о фактической реализации. Интерфейсы очень полезны и повсеместно используются в ООП. Так как они разделяют сам интерфей и реализацию, они предоставляют много преимуществ своего использования:

1. **Множественное наследование**.
2. **Слабая связанность**. Происходит абстракция операции, такая как разделение на уровни, а конкретной реализацией может быть что угодно: JDBC, JPA, JTA и т.д.
3. **Программа-интерфейс не реализуется**.
4. **Полиморфизм с динамическим связыванием**: раскрывается програмный интерфейс объекта без раскрытия его фактической реализации.
5. **Абстрактные уровни**, разделение функциональностей.

**Разница между интерфейсом и абстрактным классом.**

* Интерфейс — это договорные отношения с классами, которые этот интерфейс реализуют, о том, что реализация происходит путём, обозначенным интерфейсом. Это пустая оболочка с объявленными методами.
* Абстрактный класс определяет некоторое общее поведение и просит свои подклассы определить нетипичное или конкретное поведение для своего класса.
* Методы и члены абстрактного класса могут быть обозначены любым модификатором доступа, в свою очередь все методы интерфейса обязаны быть открытыми (public).
* Когда происходит наследование абстрактного класса, класс-наследник должен определить абстрактные методы, в то время как интерфейс может наследовать другой интерфейс и при этом не обязательно определять его методы.
* Класс-наследник может расширять только один абстрактный класс, а интерфейс может расширять или класс может реализовывать множество других интерфейсов.
* Класс-наследник может определять абстрактные методы с тем же или менее ограниченным модификатором доступа, при этом класс, реализующий интерфейс, должен определять методы с тем же уровнем видимости.
* Интерфейс не содержит конструкторы, в том время, как они есть в абстрактном классе.
* Переменные, объявленные в Java-интерфейсе по умолчанию являются final. Абстрактный класс может содержать переменные, которые не являются final.
* Все участники Java-интерфейса по умолчанию являются public. Участники абстрактного класса могут позволить себе быть public, protected и др.

**ПРИМЕР**

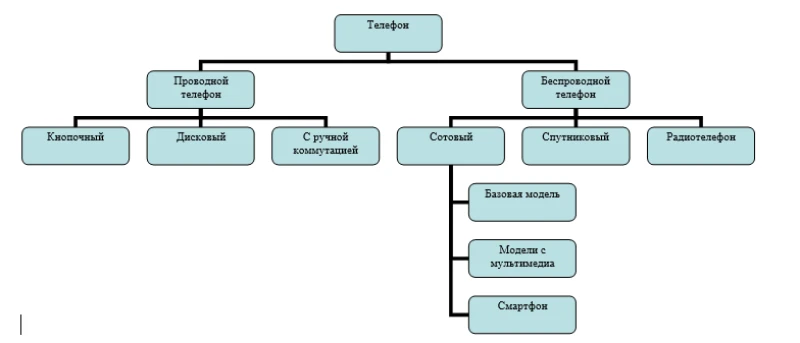
Попробуем с помощью примеров разобраться, что такое классы и объекты, а также с тем, как применять на практике основные принципы ООП: абстракцию, наследование, полиморфизм и инкапсуляцию.

### Что такое объект?

Мир, в котором мы живем, состоит из объектов. Если мы посмотрим вокруг, то увидим, что нас окружают дома, деревья, автомобили, мебель, посуда, компьютеры. Все эти предметы являются объектами, и каждый из них обладает набором определенных характеристик, поведением и назначением. Мы привыкли к объектам, и мы их используем всегда для вполне конкретных целей. Например, если нам необходимо доехать до работы, мы пользуемся автомобилем, если захотим поесть – посудой, а если отдохнуть – нам понадобится удобный диван. Человек привык мыслить объектно для решения задач в повседневной жизни. Это послужило одной из причин использования объектов в программировании, а такой подход к созданию программ назвали объектно-ориентированным. Приведём пример. Представьте, что вы разработали новую модель телефона и хотите наладить её серийное производство. Как разработчик телефона, вы знаете для чего он нужен, как он будет функционировать, и из каких деталей он будет состоять (корпус, микрофон, динамик, провода, кнопки и т.д.). При этом только вы знаете, как соединить эти детали. Однако вы не планируете выпускать телефоны лично, для этого у вас есть целый штат работников. Чтобы вам не пришлось каждый раз объяснять, как соединить детали телефона, и чтобы все телефоны при производстве получались одинаковыми, прежде чем начать их выпуск, вам понадобиться сделать чертеж в виде описания устройства телефона. **В ООП такое описание, чертеж, схема или шаблон называется классом, из которого при выполнении программы создается объект.** Класс — это описание еще не созданного объекта, как бы общий шаблон, состоящий из полей, методов и конструктора, а объект – экземпляр класса, созданный на основе этого описания.

## АБСТРАКЦИЯ ООП

Давайте теперь подумаем, как нам перейти от объекта из реального мира к объекту в программе на примере телефона. История этого средства связи превышает 100 лет и современный телефон, в отличие от своего предшественника из 19 века, представляет собой куда более сложное устройство. Когда мы пользуемся телефоном, то не задумываемся о его устройстве и процессах, происходящих внутри него. Мы просто используем функции, предоставленные разработчиками телефона — кнопки или сенсорный экран для выбора номера и совершения вызовов. Одним из первых интерфейсов телефона была рукоятка, которую нужно было вращать, чтобы сделать вызов. Разумеется, это было не очень удобно. Тем не менее, свою функцию рукоять исправно выполняла. Если посмотреть на самый современный и на самый первый телефон, можно сразу выделить самые важные детали, которые важны и для устройства конца 19-го века, и для суперсовременного смартфона. Это совершение вызова (набор номера) и приём вызова. По сути это то, что делает телефон телефоном, а не чем-то другим. Сейчас мы применили принцип в ООП — выделение наиболее важных характеристик и информации об объекте. **Этот принцип ООП называется абстракцией.** Абстракцию в ООП можно также определить, как способ представления элементов задачи из реального мира в виде объектов в программе. Абстракция всегда связана с обобщением некоторой информации о свойствах предметов или объектов, поэтому главное — это отделить значимую информацию от незначимой в контексте решаемой задачи. При этом уровней абстракции может быть несколько. Попробуем применить принцип абстракции к нашим телефонам. Для начала выделим наиболее распространённые типы телефонов от самых первых и до наших дней. Например, их можно представить в виде диаграммы, приведенной на рисунке 1.



Теперь с помощью абстракции мы можем выделить в этой иерархии объектов общую информацию: общий абстрактный тип объектов — телефон, общую характеристику телефона — год его создания, и общий интерфейс — все телефоны способны принимать и посылать вызовы. Вот как это выглядит на Java:

**public** **abstract** **class** AbstractPhone {

**private** **int** year;

**public** AbstractPhone(**int** year) {

**this**.year = year;

}

**public** **abstract** **void** call(**int** outputNumber);

**public** **abstract** **void** ring (**int** inputNumber);

}

На основании этого абстрактного класса мы сможем создавать в программе новые типы телефонов с использованием других базовых принципов ООП Java, которые рассмотрим ниже.

## ИНКАПСУЛЯЦИЯ

**С помощью абстракции мы выделяем общее для всех объектов.** Однако каждая модель телефона — индивидуальна и чем-то отличается от других. Как же нам в программе провести границы и обозначить эту индивидуальность? Как сделать так, чтоб никто из пользователей случайно или преднамеренно не смог сломать наш телефон, или попытаться переделать одну модель в другую? Для мира реальных объектов ответ очевиден: нужно поместить все детали в корпус телефона. Ведь если этого не сделать и оставить все внутренности телефона и провода, соединяющие их снаружи, обязательно найдется любознательный экспериментатор, который захочет “улучшить” работу нашего телефона. Для исключения подобного вмешательства в конструкцию и работу объекта **в ООП используют принцип инкапсуляции** – еще один базовый принцип ООП, при котором атрибуты и поведение объекта объединяются в одном классе, внутренняя реализация объекта скрывается от пользователя, а для работы с объектом предоставляется открытый интерфейс. Задача программиста — определить, какие атрибуты и методы будут доступны для открытого доступа, а какие являются внутренней реализацией объекта и должны быть недоступны для изменений.

## Инкапсуляция и управление доступом

Допустим, при производстве на тыльной стороне телефона гравируется информация о нем: год его выпуска или логотип компании производителя. Эта информация вполне конкретно характеризует данную модель — его состояние. Можно сказать, разработчик телефона позаботился о неизменности этой информации — вряд ли кому-то придет в голову удалять гравировку. В мире Java состояние будущих объектов описывается в классе с помощью полей, а их поведение – с помощью методов. Возможность же изменения состояния и поведения осуществляется с помощью модификаторов доступа к полям и методам – private, protected, public, а также default (доступ по умолчанию). Например, мы решили, что год создания, название производителя телефона и один из методов относятся к внутренней реализации класса и не подлежат изменению другими объектами в программе. С помощью кода класс можно описать так:

**public** **class** SomePhone {

**private** **int** year;

**private** String company;

**public** SomePhone(**int** year, String company) {

**this**.year = year;

**this**.company = company;

}

**private** **void** openConnection(){

//findComutator

//openNewConnection...

}

**public** **void** call() {

openConnection();

System.out.println("Вызываю номер");

}

**public** **void** ring() {

System.out.println("Дзынь-дзынь");

}

}

Модификатор private делает доступными поля и методы класса только внутри данного класса. Это означает, что получить доступ к private полям из вне невозможно, как и нет возможности вызвать private методы. Сокрытие доступа к методу openConnection, оставляет нам также возможность к свободному изменению внутренней реализации этого метода, так как этот метод гарантированно не используется другими объектами и не нарушит их работу. Для работы с нашим объектом мы оставляем открытыми методы call и ring с помощью модификатора public. **Предоставление открытых методов для работы с объектом также является частью механизма инкапсуляции, так как если полностью закрыть доступ к объекту – он станет бесполезным.**

## НАСЛЕДОВАНИЕ

Давайте посмотрим еще раз на диаграмму телефонов. Можно заметить, что она представляет собой иерархию, в которой модель, расположенная ниже обладает всеми признаками моделей, расположенных выше по ветке, плюс своими собственными. Например, смартфон, использует сотовую сеть для связи (обладает свойствами сотового телефона), является беспроводным и переносным (обладает свойствами беспроводного телефона) и может принимать и делать вызовы (свойствами телефона). В этом случае мы можем говорить о наследовании свойств объекта. **В программировании наследование заключается в использовании уже существующих классов для описания новых.** Рассмотрим пример создания класса смартфон с помощью наследования. Все беспроводные телефоны работают от аккумуляторных батарей, которые имеют определенный ресурс работы в часах. Поэтому добавим это свойство в класс беспроводных телефонов:

**public** **abstract** **class** WirelessPhone **extends** AbstractPhone {

**private** **int** hour;

**public** WirelessPhone(**int** year, **int** hour) {

**super**(year);

**this**.hour = hour;

}

}

Сотовые телефоны наследуют свойства беспроводного телефона, мы также добавили в этот класс реализацию методов call и ring:

**public** **class** CellPhone **extends** WirelessPhone {

**public** CellPhone(**int** year, **int** hour) {

**super**(year, hour);

}

@Override

**public** **void** call(**int** outputNumber) {

System.out.println("Вызываю номер " + outputNumber);

}

@Override

**public** **void** ring(**int** inputNumber) {

System.out.println("Вам звонит абонент " + inputNumber);

}

}

И, наконец, класс смартфон, который в отличие от классических сотовых телефонов имеет полноценную операционную систему. В смартфон можно добавлять новые программы, поддерживаемые данной операционной системой, расширяя, таким образом, его функциональность. С помощью кода класс можно описать так:

**public** **class** Smartphone **extends** CellPhone {

**private** String operationSystem;

**public** Smartphone(**int** year, **int** hour, String operationSystem) {

**super**(year, hour);

**this**.operationSystem = operationSystem;

}

**public** **void** install(String program){

System.out.println("Устанавливаю " + program + "для" + operationSystem);

}

}

Как видите, для описания класса Smartphone мы создали совсем немного нового кода, но получили новый класс с новой функциональностью. Использование принципа наследование ООП позволяет значительно уменьшить объем кода, а значит, и облегчить работу программисту.

## ПОЛИМОРФИЗМ

Если мы посмотрим на все модели телефонов, то, несмотря на различия во внешнем облике и устройстве моделей, мы можем выделить у них некое общее поведение – все они могут принимать и совершать звонки и имеют достаточно понятный и простой набор кнопок управления. Применяя известный нам уже один из основных принципов ООП абстракцию в терминах программирования можно сказать, что объект телефон имеет один общий интерфейс. Поэтому пользователи телефонов могут вполне комфортно пользоваться различными моделями, используя одни и те же кнопки управления (механические или сенсорные), не вдаваясь в технические тонкости устройства. Так, вы постоянно пользуетесь сотовым телефоном, и без труда сможете совершить звонок с его стационарного собрата. Принцип в ООП, когда программа может использовать объекты с одинаковым интерфейсом без информации о внутреннем устройстве объекта, называется **полиморфизмом**. Давайте представим, что нам в программе нужно описать пользователя, который может пользоваться любыми моделями телефона, чтобы позвонить другому пользователю. Вот как можно это сделать:

**public** **class** User {

**private** String name;

**public** User(String name) {

**this**.name = name;

}

**public** **void** callAnotherUser(**int** number, AbstractPhone phone){

// вот он полиморфизм - использование в коде абстактного типа AbstractPhone phone!

phone.call(number);

}

}

}

Теперь опишем различные модели телефонов. Одна из первых моделей телефонов:

**public** **class** ThomasEdisonPhone **extends** AbstractPhone {

**public** ThomasEdisonPhone(**int** year) {

**super**(year);

}

@Override

**public** **void** call(**int** outputNumber) {

System.out.println("Вращайте ручку");

System.out.println("Сообщите номер абонента, сэр");

}

@Override

**public** **void** ring(**int** inputNumber) {

System.out.println("Телефон звонит");

}

}

Обычный стационарный телефон:

**public** **class** Phone **extends** AbstractPhone {

**public** Phone(**int** year) {

**super**(year);

}

@Override

**public** **void** call(**int** outputNumber) {

System.out.println("Вызываю номер" + outputNumber);

}

@Override

**public** **void** ring(**int** inputNumber) {

System.out.println("Телефон звонит");

}

}

И, наконец, крутой видеотелефон:

**public** **class** VideoPhone **extends** AbstractPhone {

**public** VideoPhone(**int** year) {

**super**(year);

}

@Override

**public** **void** call(**int** outputNumber) {

System.out.println("Подключаю видеоканал для абонента " + outputNumber );

}

@Override

**public** **void** ring(**int** inputNumber) {

System.out.println("У вас входящий видеовызов..." + inputNumber);

}

}

Создадим объекты в методе main() и протестируем метод callAnotherUser:

AbstractPhone firstPhone = **new** ThomasEdisonPhone(1879);

AbstractPhone phone = **new** Phone(1984);

AbstractPhone videoPhone=**new** VideoPhone(2018);

User user = **new** User("Андрей");

user.callAnotherUser(224466,firstPhone);

// Вращайте ручку

//Сообщите номер абонента, сэр

user.callAnotherUser(224466,phone);

//Вызываю номер 224466

user.callAnotherUser(224466,videoPhone);

//Подключаю видеоканал для абонента 224466

Используя вызов одного и того же метода объекта user, мы получили различные результаты. Выбор конкретной реализации метода call внутри метода callAnotherUser производился динамически на основании конкретного типа вызывающего его объекта в процессе выполнения программы. В этом и заключается основное преимущество полиморфизма – выбор реализации в процессе выполнения программы. В примерах классов телефонов, приведенных выше, мы использовали переопределение методов – прием, при котором изменяется реализация метода, определенная в базовом классе, без изменения сигнатуры метода. По сути это является заменой метода, и именно новый метод, определенный в подклассе, вызывается при выполнении программы. Обычно, при переопределении метода, используется аннотация @Override, которая подсказывает компилятору о необходимости проверить сигнатуры переопределяемого и переопределяющего методов. В итоге, чтобы стиль вашей программы соответствовал концепции ООП и принципам ООП java следуйте следующим советам:

* выделяйте главные характеристики объекта;
* выделяйте общие свойства и поведение и используйте наследование при создании объектов;
* используйте абстрактные типы для описания объектов;
* старайтесь всегда скрывать методы и поля, относящиеся к внутренней реализации класса.

***14. Приведите правила, которым должен следовать компонент java-bean.***

**JavaBeans** — классы в языке [Java](https://ru.wikipedia.org/wiki/Java), написанные по определённым правилам. Они используются для объединения нескольких [объектов](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) в один ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *bean* — фасоль, кофейное зерно) для удобной передачи данных.

Спецификация [Sun Microsystems](https://ru.wikipedia.org/wiki/Sun_Microsystems) определяет JavaBeans как повторно используемые программные компоненты, которыми можно управлять, используя графические конструкторы и средства [IDE](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8).

JavaBeans обеспечивают основу для многократно используемых, встраиваемых и модульных [компонентов](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) [ПО](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5). Компоненты JavaBeans могут принимать различные формы, но наиболее широко они применяются в элементах графического пользовательского [интерфейса](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81). Одна из целей создания JavaBeans — взаимодействие с похожими компонентными [структурами](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0). Например, [Windows-программа](https://ru.wikipedia.org/w/index.php?title=Windows-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0&action=edit&redlink=1), при наличии соответствующего [моста](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D1%81%D1%82_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) или [объекта-обёртки](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), может использовать компонент JavaBeans так, будто бы он является компонентом [COM](https://ru.wikipedia.org/wiki/Component_Object_Model) или [ActiveX](https://ru.wikipedia.org/wiki/ActiveX).

## Правила описания JavaBean.

Чтобы [класс](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) мог работать как bean, он должен соответствовать определённым соглашениям об именах методов, конструкторе и поведении. Эти соглашения дают возможность создания инструментов, которые могут использовать, замещать и соединять JavaBeans.

Правила описания гласят:

* Класс должен иметь [конструктор](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) без параметров, с модификатором доступа public. Такой конструктор позволяет инструментам создать объект без дополнительных сложностей с параметрами.
* Свойства класса должны быть доступны через get, set и другие методы (так называемые [методы доступа](https://ru.wikipedia.org/wiki/Accessor)), которые должны подчиняться стандартному соглашению об именах. Это легко позволяет инструментам автоматически определять и обновлять содержание bean’ов. Многие инструменты даже имеют специализированные редакторы для различных типов свойств.
* Класс должен быть [сериализуем](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F#Java). Это даёт возможность надёжно сохранять, хранить и восстанавливать состояние bean независимым от платформы и виртуальной машины способом.
* Класс должен иметь переопределенные методы equals(), hashCode() и toString().
* *// PersonBean.java*
* 3**public** **class** **PersonBean** **implements** java.io.Serializable {
* 4 **private** String name;
* 5 **private** boolean deceased;
* 6 *// Методы геттеры (get) и сеттеры (set)*
* 8 **public** String getName() {
* 9 **return** name;
* 10 }
* 11 **public** void setName(String name) {
* 12 **this**.name = name;
* 13 }
* 14 **public** boolean isDeceased() {
* 15 **return** deceased;
* 16 }
* 17 **public** void setDeceased(boolean deceased) {
* 18 **this**.deceased = deceased;
* 19 }
* 20 *//Переопределенные методы equals() и hashCode()*
* 22 @Override
* 23 **public** boolean equals(Object o) {
* 24 **if** (**this** == o) {
* 25 **return** **true**;
* 26 }
* 27 **if** (o == **null** || getClass() != o.getClass()) {
* 28 **return** **false**;
* 29 }
* 30 PersonBean that = (PersonBean) o;
* 32 **if** (deceased != that.deceased) {
* 34 **return** **false**;
* 35 }
* 36 **return** !(name != **null** ? !name.equals(that.name) : that.name != **null**);
* 37 }
* 39 @Override
* 41 **public** int hashCode() {
* 42 int result = name != **null** ? name.hashCode() : 0;
* 43 result = 31 \* result + (deceased ? 1 : 0);
* 44 **return** result;
* 45 }
* 46 *//Переопределенный метод toString()*
* 48 @Override
* 49 **public** String toString() {
* 50 **return** "PersonBean{" +
* 51 "name='" + name + '\'' +
* 52 ", deceased=" + deceased +
* 53 '}';
* 54 }
* 55}
* *// TestPersonBean.java*
* **public** **class** **TestPersonBean** {
* **public** **static** void main(String[] args) {
* PersonBean person = **new** PersonBean();
* person.setName("Bob");
* person.setDeceased(**true**);
* *// Результат: "Bob [deceased]"*
* System.out.print(person.getName());
* System.out.println(person.isDeceased() ? " [deceased]" : " [alive]");
* }
* }

***15. Как вы думаете, для чего используется наследование классов в java-программе? Приведите пример наследования. Как вы думаете, поля и методы, помеченными модификатором доступа private, наследуются?***

**Наследование** — это процесс перенимания классом свойств (методов и полей) другого класса. С использованием в Java наследования информация становится управляемой в иерархическом порядке.

Класс, который наследует свойства другого класса, называется подклассом (производным классом, наследующим классом), а класс, свойства которого наследуются, известен как суперкласс (базовый класс, родительский класс)

**extends** — это кодовое слово, используемое для наследования свойств класса.

Дальше приведён пример процесса наследования на Java. На этом примере Вы можете рассмотреть два класса с именами Calculator и My\_Calculator.

Используя ключевое слово extends в Java, My\_Calculator перенимает методы addition() и subtraction() класса Calculator.

class Calculator {

int c;

public void addition(int a, int b) {

c = a + b;

System.out.println("Сумма чисел: " + c);

}

public void subtraction(int a, int b) {

c = a - b;

System.out.println("Разность чисел: " + c);

}

}

public class My\_Calculator extends Calculator {

public void multiplication(int a, int b) {

c = a \* b;

System.out.println("Произведение чисел: " + c);

}

public static void main(String args[]) {

int a = 10, b = 20;

My\_Calculator cal = new My\_Calculator();

cal.addition(a, b);

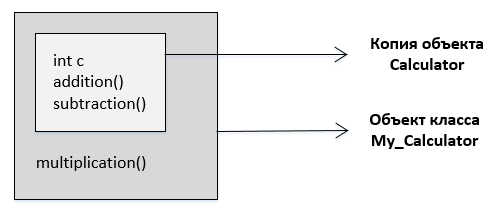
cal.subtraction(a, b);

cal.multiplication(a, b);

}

}

В данной программе, при создании объекта классу **My\_Calculator**, копия содержимого суперкласса создаётся в нём же. Поэтому, используя объект подкласса, Вы можете получить доступ к членам суперкласса.



Ссылочная переменная суперкласса может содержать объект подкласса, но, используя эту переменную, Вы можете иметь доступ только к членам суперкласса, поэтому, чтобы иметь доступ к членам обоих классов, рекомендуется всегда создавать ссылочную переменную к подклассу.

Обращаясь к программе выше, Вы можете создать экземпляр класса, как в примере ниже. Но, используя ссылочную переменную суперкласса, Вы не можете вызвать метод **multiplication()**, который принадлежит подклассу My\_Calculator.

Calculator cal = new My\_Calculator();

cal.addition(a, b);

cal.subtraction(a, b);

**Примечание:** подкласс наследует все члены (поля, методы, вложенные классы) из суперкласса. в Java конструкторы не являются членами, поэтому они не наследуются подклассом, но конструктор суперкласса может быть вызван из подкласса.

## КЛЮЧЕВОЕ СЛОВО SUPER

Ключевое слово **super** схоже с ключевым словом **this**. Ниже приведены случаи, где используется **super** в Java.

* Для **дифференциации членов** суперкласса от членов подкласса, если у них есть одинаковые имена.
* Для **вызова конструктора суперкласса** из подкласса.

### **Дифференциация членов**

Если класс перенимает свойства другого класса, и члены суперкласса имеют те же имена, что и в подклассе, для их разделения мы используем ключевое слово super.

Этот раздел содержит программу, которая демонстрирует использование ключевого слова **super** в Java.

В предложенной программе у вас есть два класса с именами Sub\_class и Super\_class, оба имеющие метод display() с разными реализациями и переменную с именем num с разными значениями. Вы можете увидеть, что мы использовали ключевое слово **super** для дифференциации членов суперкласса из подкласса.

// Интерфейс

class Super\_class {

int num = 88;

// Метод display() суперкласса

public void display() {

System.out.println("Это метод display() суперкласса");

}

}

public class Sub\_class extends Super\_class {

int num = 77;

// Метод display() субкласса

public void display() {

System.out.println("Это метод display() подкласса");

}

public void my\_method() {

// Инициализация подкласса

Sub\_class sub = new Sub\_class();

// Вызываем метод display() подкласса

sub.display();

// Вызываем метод display() суперкласса

super.display();

// Выводим значение переменной num подкласса

System.out.println("Значение переменной num в подклассе: " + sub.num);

// Выводим значение переменной num суперкласса

System.out.println("Значение переменной num в суперклассе: " + super.num);

}

public static void main(String args[]) {

Sub\_class obj = new Sub\_class();

obj.my\_method();

}

}

### **Вызов конструктора суперкласса**

Если класс перенимает свойства другого класса, подкласс автоматически получается стандартный конструктор суперкласса. Но если Вы хотите вызвать параметризованный конструктор суперкласса, Вам нужно использовать ключевое слово super.

В предложенной программе демонстрируется использование в Java ключевого слова super для вызова параметризованного конструктора. В этой программе содержится суперкласс и подкласс, где суперкласс содержит параметризованный конструктор, который принимает строковое значение, а мы используем ключевое слово super для вызова параметризованного конструктора суперкласса.

class Superclass {

int age;

Superclass(int age) {

this.age = age;

}

public void getAge() {

System.out.println("Значение переменной age в суперклассе равно: " + age);

}

}

public class Subclass extends Superclass {

Subclass(int age) {

super(age);

}

public static void main(String args[]) {

Subclass s = new Subclass(24);

s.getAge();

}

}

## ОТНОШЕНИЕ IS-A

**IS-A** — это способ сказать «Этот объект является типом этого объекта». Давайте посмотрим, как ключевое слово **extends** используется для достижения наследования.

public class Animal {

}

public class Mammal extends Animal {

}

public class Reptile extends Animal {

}

public class Dog extends Mammal {

}

Теперь, основываясь на примере выше, в объектно-ориентированных терминах, следующие утверждения верны

* Animal является суперклассом класса Mammal.
* Animal является суперклассом класса Reptile.
* Mammal и Reptile являются подклассами класса Animal.
* Dog одновременно является подклассом классов Mammal и Animal.

Теперь, используя отношение IS-A, мы можем сказать так:

* Mammal IS-A Animal.
* Reptile IS-A Animal.
* Dog IS-A Mammal.

Таким образом, Dog IS-A тоже Animal.

С использованием ключевого слова **extend**, подклассы могут наследовать все свойства суперкласса кроме его приватных свойств (private).

Так как у нас есть хорошее понимание принципа работы ключевого слова **extends**, давайте рассмотрим, как используется ключевое слово **implements** для получения отношения IS-A.

В общем, ключевое слово **implements** в Java используется с классами для перенятия свойств интерфейса. Интерфейсы никогда не могут быть переняты классом с помощью **extends**.

## Ключевое Слово instanceof

Давайте использует оператор instanceof в Java с целью проверки, являются ли Mammal и Dog на самом деле Animal.

interface Animal{}

class Mammal implements Animal{}

public class Dog extends Mammal {

public static void main(String args[]) {

Mammal m = new Mammal();

Dog d = new Dog();

System.out.println(m instanceof Animal);

System.out.println(d instanceof Mammal);

System.out.println(d instanceof Animal);

}

}

## ОТНОШЕНИЕ HAS-A

Эти отношения в основном основаны на обращении. Они определяют, является ли определенный класс HAS-A определенным случаем. Эта взаимосвязь помогает уменьшить дублирование кода, а также баги. Взглянем на пример.

public class Vehicle{}

public class Speed{}

public class Van extends Vehicle {

private Speed sp;

}

Мы видим, что у класса Van HAS-A (есть) Speed. Имея отдельный класс Speed, нам не нужно вставлять код, принадлежащий Speed в класс Van, что позволяет нам использовать класс Speed в нескольких приложениях.

В особенности объектно-ориентированного программирования, пользователям не нужно волноваться о том, какой объект выполняет текущую работу. Для достижения этого, класс Van скрывает детали реализации от пользователей класса Van. Таким образом, пользователи, должны попросить класс Van выполнить определенное действие, и класс Van либо выполнит работу сам по себе, либо попросит другой класс выполнить действие.

## ВИДЫ НАСЛЕДОВАНИЯ

Есть различные способы наследования, как показано ниже.

|  |  |  |
| --- | --- | --- |
| **Вид** | **Схема** | **Пример** |
| Одиночное наследование | Одиночное наследование в Java | public class A { ... }  public class B extends A { ... } |
| Многоуровневое наследование | Многоуровневое наследование в Java | public class A { ... }  public class B extends A { ... }  public class C extends B { ... } |
| Иерархическое наследование | Иерархическое наследование в Java | public class A { ... }  public class B extends A { ... }  public class C extends A { ... } |
| Множественное наследование | Множественное наследование в Java | public class A { ... }  public class B { ... }  public class C extends A, B { ... }  // Java не поддерживает множественное наследование |

Очень важно запомнить, что Java не поддерживает множественное наследование. Это значит, что класс не может продлить более одного класса. Значит, следующее утверждение НЕВЕРНО:

public class extends Animal, Mammal{}

Тем не менее, класс может реализовать один или несколько интерфейсов, что и помогло Java избавиться от невозможности множественного наследования.

**Динамическая диспетчеризация методов**

Наследование и возможность переопределения методов открывают нам большие возможности. Прежде всего мы можем передать переменной суперкласса ссылку на объект подкласса:

|  |  |
| --- | --- |
| 1 | Person sam = new Employee("Sam", "Oracle"); |

Так как Employee наследуется от Person, то объект Employee является в то же время и объектом Person. Грубо говоря, любой работник предприятия одновременно является человеком.

Однако несмотря на то, что переменная представляет объект Person, виртуальная машина видит, что в реальности она указывает на объект Employee. Поэтому при вызове методов у этого объекта будет вызываться та версия метода, которая определена в классе Employee, а не в Person. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class Program{      public static void main(String[] args) {          Person tom = new Person("Tom");          tom.display();          Person sam = new Employee("Sam", "Oracle");          sam.display();      }  }  class Person {      private String name;      public String getName() { return name; }      public Person(String name){          this.name=name;      }      public void display(){          System.out.printf("Person %s \n", name);      }  }  class Emplo extends Person{      private String company;      public Employee(String name, String company) {          super(name);          this.company = company;      }      @Override      public void display(){          System.out.printf("Employee %s works in %s \n", super.getName(), company);      }  } |

Консольный вывод данной программы:

Person Tom

Employee Sam works in Oracle

При вызове переопределенного метода виртуальная машина динамически находит и вызывает именно ту версию метода, которая определена в подклассе. Данный процесс еще называется dynamic method lookup или динамический поиск метода или динамическая диспетчеризация методов.

***16. Укажите, как вызываются конструкторы при создании объекта производного класса? Что в конструкторе класса делает оператор super()? Возможно ли в одном конструкторе использовать операторы super() и this()?***

Производный класс имеет доступ ко всем методам и полям базового класса (даже если базовый класс находится в другом пакете) кроме тех, которые определены с модификатором private. При этом производный класс также может добавлять свои поля и методы:

|  |  |
| --- | --- |
| 1  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class Program{      public static void main(String[] args) {          Employee sam = new Employee("Sam", "Microsoft");          sam.display();  // Sam          sam.work();     // Sam works in Microsoft      }  }  class Person {      private String name;      public String getName(){ return name; }      public Person(String name){          this.name=name;      }      public void display(){          System.out.println("Name: " + name);      }  }  class Employee extends Person{       private String company;      public Employee(String name, String company) {          super(name);          this.company=company;      }      public void work(){          System.out.printf("%s works in %s \n", getName(), company);      }  } |

В данном случае класс Employee добавляет поле company, которое хранит место работы сотрудника, а также метод work.

Если в базовом классе определены конструкторы, то в конструкторе производного классы необходимо вызвать один из конструкторов базового класса с помощью ключевого слова super. Например, класс Person имеет конструктор, который принимает один параметр. Поэтому в классе Employee в конструкторе нужно вызвать констуктор класса Person. После слова super в скобках идет перечисление передаваемых аргументов. Таким образом, установка имени сотрудника делегируется конструктору базового класса.

При этом вызов конструктора базового класса должен идти в самом начале в конструкторе производного класса.

**Запрет наследования**

Хотя наследование очень интересный и эффективный механизм, но в некоторых ситуациях его применение может быть нежелательным. И в этом случае можно запретить наследование с помощью ключевого слова final. Например:

|  |  |
| --- | --- |
| 1  2 | public final class Person {  } |

Если бы класс Person был бы определен таким образом, то следующий код был бы ошибочным и не сработал, так как мы тем самым запретили наследование:

|  |  |
| --- | --- |
| 1  2 | class Employee extends Person{  } |

Кроме запрета наследования можно также запретить переопределение отдельных методов. Например, в примере выше переопределен метод display(), запретим его переопределение:

|  |  |
| --- | --- |
| 3  4  5  6  7  8  9 | public class Person {       //........................           public final void display(){                   System.out.println("Имя: " + name);      }  } |

В этом случае класс Employee не сможет переопределить метод display.

***17. Объясните, как вы понимаете утверждения: “ссылка базового класса может ссылаться на объекты своих производных типов” и “объект производного класса может быть использован везде, где ожидается объект его базового типа”. Верно ли обратное и почему?***

***- «ссылка базового класса может ссылаться на объекты своих производных типов»:***

Есть класс Person (см. выше п.13), от него наследует класс Employee. Т.к. Employee наследует от Person, то объект Employee одновременно является и объектом Person.

Person sam = new Employee("Sam", "Oracle");

Таким образом у объекта sam, хоть он и является объектом Person, будут использоваться методы и переменные Employee.

***- «объект производного класса может быть использован везде, где ожидается объект его базового типа»:***

Как говорилось чуть выше, объект производного класса (например, Employee), также является объектом базового класса (Person). Поэтому, везде, где используется базовый объект, можно использовать его наследуемый объект.

В обратном порядке это не работает, т.к. взаимосвязь идет сверху вниз.

***18. Что такое переопределение методов? Как вы думаете, зачем они нужны? Можно ли менять возвращаемый тип при переопределении методов? Можно ли менять атрибуты доступа при переопределении методов? Можно ли переопределить методы в рамках одного класса?***

**Переопределение метода** (англ. Method overriding) в объектно-ориентированном программировании — одна из возможностей языка программирования, позволяющая подклассу или дочернему классу обеспечивать специфическую реализацию метода, уже реализованного в одном из суперклассов или родительских классов.

Переопределение (**Override в Java)** необходимо для того, чтобы один общий метод мог делать разные вещи в зависимости от того, в каком классе он был вызван. Переопределение позволяет взять какой-то метод родительского класса и написать в каждом классе-наследнике свою реализацию этого метода. Новая реализация «заменит» родительскую в дочернем классе. **Переопределение методов в Java** — один из инструментов для реализации идеи полиморфизма (принципа ООП, о котором мы рассказывали в прошлой лекции). Поэтому главным преимуществом его использования будет та же гибкость, о которой мы говорили ранее. Мы можем выстроить простую и логичную систему классов, каждый из которых будет обладать специфическим поведением (собаки лают, кошки мяукают), но единым интерфейсом — один метод voice() на всех вместо кучи методов voiceDog(), voiceCat() и т.д.

У переопределения есть ряд ограничений:

1. У переопределенного метода должны быть те же аргументы, что и у метода родителя.

Если метод voice родительского класса принимает на вход String, переопределенный метод в классе-потомке тоже должен принимать на вход String, иначе компилятор выдаст ошибку

**2. У переопределенного метода должны быть тот же тип возвращаемого значения, что и у метода родителя.**

**3. Модификатор доступа у переопределенного метода также не может отличаться от «оригинального».**

***19. Определите правило вызова переопределенных методов. Можно ли статические методы переопределить нестатическими и наоборот?***

Чтобы задать нужное нам поведение, мы сделали несколько вещей:

1. Создали в каждом классе-наследнике метод с названием родительского класса.

2. Сообщили компилятору, что мы не просто так назвали метод так же, как в классе-родителе: хотим переопределить его поведение. Для этого «сообщения» компилятору мы поставили над методом **аннотацию @Override** («переопределен»).  
Проставленная над методом аннотация @Override сообщает компилятору (да и читающим твой код программистам тоже): «Все ок, это не ошибка и не моя забывчивость. Я помню, что такой метод уже есть, и хочу переопределить его».

3. Написали нужную нам реализацию для каждого класса-потомка. Змея при вызове voice() должна шипеть, медведь — рычать и т.д.

Статические методы не могут быть переопределены. Класс наследник может объявлять метод с такой же сигнатурой, что и супер класс, но это не будет переопределением. При вызове переопределенного метода JVM выбирает нужный вариант основываясь на типе объекта. Вызов же статического метода происходит без объекта. Версия вызываемого статического метода всегда определяется на этапе компиляции.

При использовании ссылки для доступа к статическому члену компилятор при выборе метода учитывает тип ссылки, а не тип объекта, ей присвоенного.

***20. Какие свойства имеют финальные методы и финальные классы? Как вы думаете, зачем их использовать?***

Обозначая метод класса модификатором **final**, мы имеем ввиду, что ни один производный класс не в состоянии переопределить этот метод, изменив его внутреннюю реализацию. Другими словами, речь идет о финальной версии метода. Класс в целом также может быть помечен как **final**.

Класс, помеченный как **final**, не поддается наследованию и все его методы косвенным образом приобретают свойство **final**.

Применение признака **final** в объявлениях классов и методов способно повысить уровень безопасности кода. Если класс снабжен модификатором **final**, никто не в состоянии расширить класс и, вероятно, нарушить при этом его контракт. Если признаком **final** обозначен метод, вы можете полностью доверять его внутренней реализации во всех ситуациях, не опасаясь "подделки". Уместно применить **final**, например, в объявлении метода, предусматривающего проверку пароля, вводимого пользователем, чтобы гарантировать точное исполнение того, что методом предусмотрено изначально. Возможному злоумышленнику не удастся изменить исходную реализацию такого метода, "подсунув" программе его переопределенную версию, которая, скажем, всегда возвращает значение true, свидетельствующее об успешной регистрации пользователя, независимо от того, какой пароль он ввел на самом деле. Вы вправе, если позволяет конкретная ситуация, пойти дальше и объявить как **final** класс целиком; метод **ValidatePassword** приобретёт то же свойство косвенным путём.

Употребление модификатора **final** в объявлении метода или класса накладывает серьезные ограничения на возможность дальнейшего использования и развития кода. Применение **final** в объявлении метода – это верный показатель того, что реализация метода самодостаточна и полностью завершена. Другие программисты, которые захотят воспользоваться вашим классом, расширив его функции в угоду собственным потребностям, будут стеснены в выборе средств достижения цели либо полностью лишены таковых. Пометив признаком **final** класс в целом, вы запретите возможность его наследования и, вероятно, существенно снизите его практическую ценность для других. Собравшись применить модификатор **final**, убедитесь, готовы ли ВЫ к подобным жертвам и стоит ли их принести.

Во многих случаях для достижения достаточного уровня безопасности кода вовсе нет необходимости обозначать весь класс как **final** – вполне возможно сохранить способность класса к расширению, пометив модификатором **final** только его "критические" структурные элементы. В этом случае вы оставите в неприкосновенности основные функции класса и одновременно разрешите его наследование с добавлением новых членов, но без переопределения "старых". Разумеется, поля, к которым обращается код методов **final**, должны быть в свою очередь обозначены как **final** либо **private**, поскольку в противном случае любой производный класс получит возможность изменять их содержимое, воздействуя на поведение соответствующих методов.

Еще один эффект применения модификатора **final** связан с упрощением задачи оптимизации кода, решаемой компилятором. Вот что происходит, когда вызывается метод, не помеченный как **final**, исполняющая система определяет фактический класс объекта, связывает вызов с наиболее подходящим кодом из группы перегруженных методов и передает управление этому коду. Но если бы, например, метод **getName** в примере класса **Attr**, рассмотренном раньше, был обозначен как **final**, операция обращения к нему, возможно, была бы заметно упрощена. В самом тривиальном случае, подобном тому, который касается **getName**, компилятор может попросту заменить вызов метода кодом его тела. Такой механизм носит название встраивания кода (inlining). При использовании inline-версии метода **getName** два следующих выражения выполняются совершенно одинаково:

Хотя приведенные выражения равнозначны, второе все-таки обладает преимуществом, поскольку метод **getName** позволяет придать полю name свойство "только для чтения", а коду класса – некую степень абстракции, которая обеспечивает возможность более свободного изменения реализации класса. Та же схема оптимизации может быть применена компилятором и по отношению к методам **private** и **statiс**, так как и они не допускают переопределения. Использование модификатора final в объявлениях классов способствует также повышению эффективности некоторых операций проверки типов. В этом случае многие подобные операции могут быть выполнены уже на стадии компиляции и поэтому потенциальные ошибки выявляются гораздо раньше. Если компилятор встречает в исходном тексте ссылку на класс **final**, он может быть "уверен", что соответствующий объект относится именно к тому типу, который указан. Компилятор в состоянии сразу определить место, занимаемое классом в общей иерархии классов, и проверить, верно тот используется или нет. Если модификатор **final** не применяется, соответствующие проверки осуществляются только на стадии выполнения программы.

***21. Укажите правила приведения типов при наследовании. Напишите примеры явного и неявного преобразования ссылочных типов. Объясните, какие ошибки могут возникать при явном преобразовании ссылочных типов.***

public class Program{

    public static void main(String[] args) {

        Person tom = new Person("Tom");

        tom.display();

        Person sam = new Employee("Sam", "Oracle");

        sam.display();

        Person bob = new Client("Bob", "DeutscheBank", 3000);

        bob.display();

    }

}

// класс человека

class Person {

    private String name;

    public String getName() { return name; }

    public Person(String name){

        this.name=name;

    }

    public void display(){

        System.out.printf("Person %s \n", name);

    }

}

// служащий некоторой компании

class Employee extends Person{

    private String company;

    public Employee(String name, String company) {

        super(name);

        this.company = company;

    }

    public String getCompany(){ return company; }

    public void display(){

        System.out.printf("Employee %s works in %s \n", super.getName(), company);

    }

}

// класс клиента банка

class Client extends Person{

    private int sum; // Переменная для хранения суммы на счете

    private String bank;

    public Client(String name, String bank, int sum) {

        super(name);

        this.bank=bank;

        this.sum=sum;

    }

    public void display(){

        System.out.printf("Client %s has account in %s \n", super.getName(), bank);

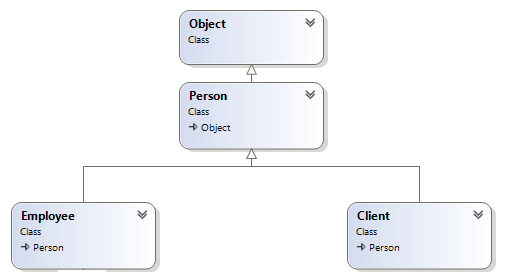
    }

    public String getBank(){ return bank; }

    public int getSum(){ return sum; }

}

В этой иерархии классов можно проследить следующую цепь наследования: Object (все классы неявно наследуются от типа Object) -> Person -> Employee|Client.



Суперклассы обычно размещаются выше подклассов, поэтому на вершине наследования находится класс Object, а в самом низу Employee и Client.

Объект подкласса также представляет объект суперкласса. Поэтому в программе мы можем написать следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Object tom = new Person("Tom");  Object sam = new Employee("Sam", "Oracle");  Object kate = new Client("Kate", "DeutscheBank", 2000);  Person bob = new Client("Bob", "DeutscheBank", 3000);  Person alice = new Employee("Alice", "Google"); |

Это так называемое восходящее преобразование (от подкласса внизу к суперклассу вверху иерархии) или upcasting. Такое преобразование осуществляется автоматически.

Обратное не всегда верно. Например, объект Person не всегда является объектом Employee или Client. Поэтому нисходящее преобразование или downcasting от суперкласса к подклассу автоматически не выполняется. В этом случае нам надо использовать операцию преобразования типов.

|  |  |
| --- | --- |
| 1  2  3  4  5 | Object sam = new Employee("Sam", "Oracle");  // нисходящее преобразование от Object к типу Employee  Employee emp = (Employee)sam;  emp.display();  System.out.println(emp.getCompany()); |

В данном случае переменная sam приводится к типу Employee. И затем через объект emp мы можем обратиться к функционалу объекта Employee.

Мы можем преобразовать объект Employee по всей прямой линии наследования от Object к Employee.

Примеры нисходящих перобразований:

|  |  |
| --- | --- |
| 1  2  3  4 | Object kate = new Client("Kate", "DeutscheBank", 2000);  ((Person)kate).display();  Object sam = new Employee("Sam", "Oracle");  ((Employee)sam).display(); |

Но рассмотрим еще одну ситуацию:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Object kate = new Client("Kate", "DeutscheBank", 2000);  Employee emp = (Employee) kate;  emp.display();  // или так  ((Employee)kate).display(); |

В данном случае переменная типа Object хранит ссылку на объект Client. Мы можем без ошибок привести этот объект к типам Person или Client. Но при попытке преобразования к типу Employee мы получим ошибку во время выполнения. Так как kate не представляет объект типа Employee.

Здесь мы явно видим, что переменная kate - это ссылка на объект Client, а не Employee. Однако нередко данные приходят извне, и мы можем точно не знать, какой именно объект эти данные представляют. Соответственно возникает большая вероятность столкнуться с ошибкой. И перед тем, как провести преобразование типов, мы можем проверить, а можем ли мы выполнить приведение с помощью оператора **instanceof**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Object kate = new Client("Kate", "DeutscheBank", 2000);  if(kate instanceof Employee){      ((Employee)kate).display();  }  else{      System.out.println("Conversion is invalid");  } |

Выражение kate instanceof Employee проверяет, является ли переменная kate объектом типа Employee. Но так как в данном случае явно не является, то такая проверка вернет значение false, и преобразование не сработает.

***22. Что такое абстрактные классы и методы? Зачем они нужны? Бывают ли случаи, когда абстрактные методы содержат тело? Можно ли в абстрактных классах определять конструкторы? Могут ли абстрактные классы содержать неабстрактные методы? Можно ли от абстрактных классов создавать объекты и почему?***

**АБСТРАКТНЫЙ КЛАСС**

Класс, который во время объявления содержит в себе ключевое слово abstract, известен как абстрактный класс.

* Абстрактные классы в Java могут содержать или не содержать абстрактные методы, т.е. методы без тела (public void get();)
* Если класс имеет хотя бы один абстрактный метод, то класс должен быть объявлен абстрактным.
* Если класс объявлен абстрактным, его нельзя реализовать и создать экземпляр класса.
* Чтобы использовать абстрактный класс, вам нужно наследовать его из другого класса, обеспечить реализацию абстрактных методов в нём.
* Если вы наследуете абстрактный класс в Java, вам нужно обеспечить реализацию всех абстрактных методов в нём.

**АБСТРАКТНЫЙ МЕТОД**

Если вы хотите, чтобы класс содержал конкретный метод, но вы желаете, чтобы фактическая реализация этого метода определялась дочерними классами, вы можете объявить метод в родительском классе как абстрактный

* Ключевое слово abstract используется для объявления метода абстрактным.
* Вам нужно разместить ключевое слово abstract перед именем метода во время его объявления.
* Абстрактный метод в Java содержит сигнатуру метода, но не содержит тела метода.
* Вместо фигурных скобок у абстрактного метода будет точка с запятой (;) на конце.

Объявление метода абстрактным ведёт к двум последствиям:

* Класс, содержащий его, должен быть объявлен абстрактным.
* Любой класс, наследующий текущий класс, должен либо переопределить абстрактный метод, либо объявить себя абстрактным.

**Примечание.** В конце концов, класс-потомок должен реализовать абстрактный метод, иначе у вас будет иерархия абстрактных классов, экземпляры которых нельзя будет создать.

* /\* File name : Employee.java \*/
* public abstract class Employee {
* private String name;
* private String address;
* private int number;
* public Employee(String name, String address, int number) {
* System.out.println("Собираем данные о работнике");
* this.name = name;
* this.address = address;
* this.number = number;
* }
* public abstract double computePay();
* public void mailCheck() {
* System.out.println("Отправляем чек " + this.name + " " + this.address);
* }
* public String toString() {
* return name + " " + address + " " + number;
* }
* public String getName() {
* return name;
* }
* public String getAddress() {
* return address;
* }
* public void setAddress(String newAddress) {
* address = newAddress;
* }
* public int getNumber() {
* return number;
* }
* }
* /\* File name : Salary.java \*/
* public class Salary extends Employee {
* private double salary; // Годовая заработная плата
* public double computePay() {
* System.out.println("Вычисляем заработную плату для " + getName());
* return salary/52;
* }
* // Остаток определения класса
* }

***23. Что такое интерфейсы? Как определить и реализовать интерфейс в java-программе? Укажите спецификаторы, которые приобретают методы и поля, определенные в интерфейсе. Можно ли описывать в интерфейсе конструкторы и создавать объекты? Можно ли создавать интерфейсные ссылки и если да, то на какие объекты они могут ссылаться?***

Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.

Чтобы определить интерфейс, используется ключевое слово interface. Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Методы без реализации похожи на абстрактные методы абстрактных классов. Так, в данном случае объявлен один метод, который не имеет реализации.

Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Чтобы класс применил интерфейс, надо использовать ключевое слово implements.

Если класс применяет интерфейс, то он должен реализовать все методы интерфейса. Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный, а его неабстрактные классы-наследники затем должны будут реализовать эти методы.

В тоже время мы не можем напрямую создавать объекты интерфейсов, поэтому следующий код не будет работать.

Одним из преимуществ использования интерфейсов является то, что они позволяют добавить в приложение гибкости. Например, класс Book и класс Journal связаны тем, что они реализуют интерфейс Printable. Поэтому мы динамически в программе можем создавать объекты Printable как экземпляры обоих классов.

**Интерфейсы в преобразованиях типов**

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Например, так как класс Journal реализует интерфейс Printable, то переменная типа Printable может хранить ссылку на объект типа Journal:

|  |
| --- |
| Printable p =new Journal("Foreign Affairs");  p.print();  // Интерфейс не имеет метода getName, необходимо явное приведение  String name = ((Journal)p).getName();  System.out.println(name); |

И если мы хотим обратиться к методам класса Journal, которые определены не в интерфейсе Printable, а в самом классе Journal, то нам надо явным образом выполнить преобразование типов: ((Journal)p).getName();

**Методы по умолчанию**

Ранее до JDK 8 при реализации интерфейса мы должны были обязательно реализовать все его методы в классе. А сам интерфейс мог содержать только определения методов без конкретной реализации. В JDK 8 была добавлена такая функциональность как методы по умолчанию. И теперь интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Например, создадим метод по умолчанию в интерфейсе Printable:

|  |
| --- |
| interface Printable {      default void print(){          System.out.println("Undefined printable");      }  } |

Метод по умолчанию - это обычный метод без модификаторов, который помечается ключевым словом default. Затем в классе Journal нам необязательно этот метод реализовать, хотя мы можем его и переопределить.

**Статические методы**

Начиная с JDK 8 в интерфейсах доступны статические методы - они аналогичны методам класса. Чтобы обратиться к статическому методу интерфейса также, как и в случае с классами, пишут название интерфейса и метод.

**Приватные методы**

По умолчанию все методы в интерфейсе фактически имеют модификатор public. Однако начиная с Java 9 мы также можем определять в интерфейсе методы с модификатором private. Они могут быть статическими и нестатическими, но они не могут иметь реализации по умолчанию.

Подобные методы могут использоваться только внутри самого интерфейса, в котором они определены. То есть к примеру нам надо выполнять в интерфейсе некоторые повторяющиеся действия, и в этом случае такие действия можно выделить в приватные методы

interface Calculatable{

    default int sum(int a, int b){

        return sumAll(a, b);

    }

    default int sum(int a, int b, int c){

        return sumAll(a, b, c);

    }

    private int sumAll(int... values){

         int result = 0;

         for(int n : values){

             result += n;

         }

         return result;

    }

**Константы в интерфейсах**

Кроме методов в интерфейсах могут быть определены статические константы. Хотя такие константы также не имеют модификаторов, но по умолчанию они имеют модификатор доступа public static final, и поэтому их значение доступно из любого места программы.

**Множественная реализация интерфейсов**

Если нам надо применить в классе несколько интерфейсов, то они все перечисляются через запятую после слова implements:

**Наследование интерфейсов**

Интерфейсы, как и классы, могут наследоваться:

***24. Для чего служит интерфейс Clonable? Как правильно переопределить метод clone() класса Object, для того, что объект мог создавать свои адекватные копии?***

Иногда необходимо получить копию объекта, которая не зависела бы от оригинала. С которой можно было бы производить манипуляции, при этом, не изменяя оригинал. При обыкновенном присваивание объектов (obj1 = obj2;) передаются ссылки на объект. В итоге два экземпляра ссылаются на один объект, и изменение одного приведет к изменению другого. Как мы видим это не то, что нам нужно. И в данном случае, нам на помощь придет интерфейс Cloneable и метод clone() класса Object.

И так, если нам необходимо получить независимый клон объекта, то необходимо вызвать метод clone(). Данный метод объявлен, как protected, а это значит, что метод защищен, и может быть доступен только при наследовании объекта. Как выясняется, это не является проблемой, потому как любой класс, является потомком класса Object. Однако при защищенном методе класс может клонировать только свои собственные объекты. Чтобы клонировать другие объекты, метод clone() необходимо расширить до public.

Пример расширения метода clone().

**public** User clone() **throws** CloneNotSupportedException {

**return** (User)**super**.clone();

       }

Как мы можем видеть метод clone() может выбрасывать исключение CloneNotSupportedException. Данное исключение возникает в случае, когда клонируемый класс не имеет реализации интерфейса Cloneable. Интерфейс Cloneable не реализует ни одного метода. Он является всего лишь маркером, говорящим, что данный класс реализует клонирование объекта.

Само клонирование осуществляется вызовом родительского метода clone(). Данный вид клонирования называется поверхностным клонированием. Его можно использовать только в том случае, если у клонируемого класса объявлены  неизменяемые типы объекты.

Пример реализации клонирования объекта.

**package** my.cloneable;

**class** User **implements** Cloneable {

**private** String name;

**private** **int** age;

**public** String getName() {

**return** name;

       }

**public** **void** setName(String name) {

**this**.name = name;

       }

**public** **int** getAge() {

**return** age;

       }

**public** **void** setAge(**int** age) {

**this**.age = age;

       }

**public** User clone() **throws** CloneNotSupportedException {

**return** (User)**super**.clone();

       }

}

**public** **class** App {

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

             User user = **new** User();

             user.setName("Иванов");

             user.setAge(25);

             System.*out*.println("Данные до клонирования: " +

user.getName() + " - " + user.getAge() + "лет");

             User clone;

**try** {

                    clone = user.clone();

                    clone.setName("Петров");

                    clone.setAge(30);

                    System.*out*.println("Клон после изменения данные: " +

                           clone.getName() + " - " + clone.getAge() + "лет");

             } **catch** (CloneNotSupportedException e) {

                    System.*out*.println("Объект не может быть клонированным.");

             }

             System.*out*.println("Оригинал, после манипуляций с клоном: " +

 user.getName() + " - " + user.getAge() + "лет");

       }

} /\* результат:

Данные до клонирования: Иванов - 25лет

Клон после изменения данные: Петров - 30лет

Оригинал, после манипуляций с клоном: Иванов - 25лет

 \*/

Как видно из данного примера, мы сначала создаем экземпляр объект User, затем передаем ему значения. После этого присваиваем клон объекта User переменной clone. Проводим манипуляции с клонированным объектом. На всем этапе выводим промежуточные значения оригинального экземпляра объекта User и его клона. Как можно заметить, из результата примера, оригинальный экземпляр объекта остался с первоначальными значениями, а значения клон экземпляра были изменены.

Существует так же второй вид клонирования объекта, который называется глубокое клонирование. Его используют в тех случаях, когда в клонируемом классе есть изменяемые объекты.

**package** my.cloneable;

**import** java.util.Calendar;

**import** java.util.Date;

**import** java.util.GregorianCalendar;

**class** User **implements** Cloneable {

**private** String name;

**private** **int** age;

**private** GregorianCalendar birthday;

**public** String getName() {

**return** name;

       }

**public** **void** setName(String name) {

**this**.name = name;

       }

**public** **int** getAge() {

**return** age;

       }

**public** **void** setAge(**int** age) {

**this**.age = age;

       }

**public** String getBirthday() {

**return** birthday.get(Calendar.*DAY\_OF\_MONTH*) + "." +

                           birthday.get(Calendar.*MONTH*) + "." +

                           birthday.get(Calendar.*YEAR*);

       }

**public** **void** setBirthday(**int** day, **int** month, **int** year) {

             birthday = **new** GregorianCalendar();

             birthday.set(Calendar.*DAY\_OF\_MONTH*, day);

             birthday.set(Calendar.*MONTH*, month);

             birthday.set(Calendar.*YEAR*, year);

       }

**public** User clone() **throws** CloneNotSupportedException {

             User clone = (User)**super**.clone();

             clone.birthday = (GregorianCalendar) birthday.clone();

**return** clone;

       }

}

**public** **class** App {

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

             User user = **new** User();

             user.setName("Иванов");

             user.setAge(25);

             user.setBirthday(12, 03, 1975);

             System.*out*.println("Данные до клонирования: " + user.getName() + " - " + user.getAge() + "лет, день рождение: " + user.getBirthday());

             User clone;

**try** {

                    clone = user.clone();

                    clone.setName("Петров");

                    clone.setAge(30);

                    clone.setBirthday(15, 11, 1992);

                    System.*out*.println("Клон после изменения данные: " + clone.getName() + " - " + clone.getAge() + "лет, день рождение: " + clone.getBirthday());

             } **catch** (CloneNotSupportedException e) {

                    System.*out*.println("Объект не может быть клонированным.");

             }

             System.*out*.println("Оригинал, после манипуляций с клоном: " + user.getName() + " - " + user.getAge() + "лет, день рождение: " + user.getBirthday());

       }

}

В данном примере, в классе User, добавлена изменяемая переменная birthday, в которой хранится дата рождения. Чтобы она нормально клонировалась необходимо переопределить метод clone() следующим образом :

**public** User clone() **throws** CloneNotSupportedException {

             User clone = (User)**super**.clone();

             clone.birthday = (GregorianCalendar) birthday.clone();

**return** clone;

       }

Как вы видите, добавилась одна строчка

             clone.birthday = (GregorianCalendar) birthday.clone();

в которой напрямую клонируется поле birthday.

Однако с клонированием объектов необходимо быть очень аккуратным, потому как возникает много ошибок из-за неправильной передачи объектов класса. По этому используйте данный механизм только в тех случаях, когда это необходимо.

**Общий контракт для метода clone**

Метод clone создает и возвращает копию этого объекта. Точное значение слова "копия" может зависеть от класса объекта. Общее намерение состоит в том, чтобы для любого объекта x выражение

x.clone() != x

будет true, и выражение

x.clone().getClass() == x.getClass()

будет true, но это не абсолютные требования. Хотя обычно

x.clone().equals(x)

будет true, это не абсолютное требование. Копирование объекта обычно влечет за собой создание нового экземпляра его класса, но также может потребоваться копирование внутренних структур данных. Никакие конструкторы не вызываются.

Если вы переопределите метод clone в нефинальном классе, вы должны вернуть объект, полученный с помощью вызова super.clone()

Методы super.clone() автоматически копируют примитивные значения

На практике ожидается, что класс, реализующий Cloneable, предоставит правильно работающий метод public clone().

метод clone() функционирует как другой конструктор; вы должны убедиться, что он не причиняет вреда исходному объекту и правильно устанавливает инварианты для клона.

Все классы, реализующие Cloneable, должны переопределять clone

Метод clone() должен сначала вызвать super.clone, а затем исправить все поля, которые необходимо исправить.

Не создавайте интерфейс расширяющий Cloneable

Не реализуйте Cloneable для любого класса, предназначенного для наследования

Альтернативы использования clone():

Конструктор копирования: public Example(Example example);

Фабрика копирования: public static Example newInstance(Example example);

***25. Для чего служат интерфейсы Comparable и Comparator? В каких случаях предпочтительнее использовать первый, а когда – второй? Как их реализовать и использовать?***

**Интерфейс Comparable**

Для того, чтобы объекты Person можно было сравнить и сортировать, они должны применять интерфейс Comparable<E>. При применении интерфейса он типизируется текущим классом. Применим его к классу Person:

class Person implements Comparable<Person>{

    private String name;

    Person(String name){

        this.name = name;

    }

    String getName(){return name;}

    public int compareTo(Person p){

        return name.compareTo(p.getName());

    }

}

Интерфейс Comparable содержит один единственный метод int compareTo(E item), который сравнивает текущий объект с объектом, переданным в качестве параметра. Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвратит ноль, значит, оба объекта равны.

В данном случае мы не возвращаем явным образом никакое число, а полагаемся на встроенный механизм сравнения, который есть у класса String. Но мы также можем определить и свою логику, например, сравнивать по длине имени.

**Интерфейс Comparator**

Однако перед нами может возникнуть проблема, что если разработчик не реализовал в своем классе, который мы хотим использовать, интерфейс Comparable, либо реализовал, но нас не устраивает его функциональность, и мы хотим ее переопределить? На этот случай есть еще более гибкий способ, предполагающий применение интерфейса Comparator<E>.

Интерфейс Comparator содержит ряд методов, ключевым из которых является метод compare():

public interface Comparator<E> {

    int compare(T a, T b);

    // остальные методы

}

Метод compare также возвращает числовое значение - если оно отрицательное, то объект a предшествует объекту b, иначе - наоборот. А если метод возвращает ноль, то объекты равны. Для применения интерфейса нам вначале надо создать класс компаратора, который реализует этот интерфейс:

class PersonComparator implements Comparator<Person>{

    public int compare(Person a, Person b){

        return a.getName().compareTo(b.getName());

    }

}

Сортировка по нескольким критериям

Начиная с JDK 8 в механизм работы компараторов были внесены некоторые дополнения. В частности, теперь мы можем применять сразу несколько компараторов по принципу приоритета. Например, изменим класс Person следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Person{      private String name;      private int age;      public Person(String n, int a){            name=n;          age=a;      }      String getName(){return name;}      int getAge(){return age;}  } |

Здесь добавлено поле для хранения возраста пользователя. И, допустим, нам надо отсортировать пользователей по имени и по возрасту. Для этого определим два компаратора:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class PersonNameComparator implements Comparator<Person>{      public int compare(Person a, Person b){            return a.getName().compareTo(b.getName());      }  }  class PersonAgeComparator implements Comparator<Person>{      public int compare(Person a, Person b){          if(a.getAge()> b.getAge())              return 1;          else if(a.getAge()< b.getAge())              return -1;          else              return 0;      }  } |

Интерфейс компаратора определяет специальный метод по умолчанию thenComparing, который позволяет использовать цепочки компараторов для сортировки набора:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Comparator<Person> pcomp = new PersonNameComparator().thenComparing(new PersonAgeComparator());  TreeSet<Person> people = new TreeSet(pcomp);  people.add(new Person("Tom", 23));  people.add(new Person("Nick",34));  people.add(new Person("Tom",10));  people.add(new Person("Bill",14));   for(Person  p : people){       System.out.println(p.getName() + " " + p.getAge());  } |

Comparable делает наши объекты «сравнимыми» и создает для них наиболее естественный порядок сортировки, который будет использоваться в большинстве случаев.

Comparator — это отдельный класс-«сравниватель» (перевод немного корявый, но зато понятный). Если нам нужно реализовать какую-то специфическую сортировку, нам необязательно лезть в класс и менять логику compareTo(). Вместо этого мы можем создать отдельный класс-comparator в нашей программе и научить его делать нужную нам сортировку!

**Generic classes and Interfaces, Enums**

1. Что такое перечисления в Java. Как объявить перечисление? Чем являются элементы перечислений? Кто и когда создает экземпляры перечислений?
2. Можно ли самостоятельно создать экземпляр перечисления? А ссылку типа перечисления? Как сравнить, что в двух переменных содержится один и тот же элемент перечисления и почему именно так?
3. Что такое анонимные классы?
4. Что такое параметризованные классы? Для чего они необходимы? Приведите пример параметризованного класса и пример создания объекта параметризованного класса? Объясните, ссылки какого типа могут ссылаться на объекты параметризованных классов? Можно ли создать объект, параметризовав его примитивным типом данных?

**1. Что такое перечисления в Java. Как объявить перечисление? Чем являются элементы перечислений? Кто и когда создает экземпляры перечислений?**

Кроме отдельных примитивных типов данных и классов в Java есть такой тип как enum или перечисление. Перечисления представляют набор логически связанных констант.

Перечисление в Java относится к типу класса, но перечисление НЕ может наследоваться от другого класса и НЕ может быть супер классом. Все перечисления автоматически наследуют от класса java.lang.Enum. В этом классе определяется ряд методов, доступных для использования во всех перечислениях: ordinal(), compareTo(), equals(), values() и valueOf(). Перечисления также неявно наследуют интерфейсы Serializable и Comparable.

Объявление перечисления происходит с помощью оператора enum, после которого идет название перечисления. Затем идет список элементов перечисления через запятую:

enum Day{

    MONDAY,

    TUESDAY,

    WEDNESDAY,

    THURSDAY,

    FRIDAY,

    SATURDAY,

    SUNDAY

}

Перечисление фактически представляет новый тип, поэтому мы можем определить переменную данного типа и использовать ее. Мы можем присвоить переменной только те значения, которые определены в перечислении:

public class Program{

    public static void main(String[] args) {

        Day current = Day.MONDAY;

        System.out.println(current);    // MONDAY

    }

}

Перечисления могут использоваться в классах для хранения данных.

**МЕТОДЫ ПЕРЕЧИСЛЕНИЙ**

**Метод values().**

Каждое перечисление имеет статический метод values(). Он возвращает массив всех констант перечисления:

public class Program{

    public static void main(String[] args) {

        Type[] types = Type.values();

        for (Type s : types) { System.out.println(s); }

    }

}

enum Type{

    SCIENCE,

    BELLETRE,

    PHANTASY,

    SCIENCE\_FICTION

}

**Метод ordinal()**

Метод ordinal() возвращает порядковый номер определенной константы (нумерация начинается с 0):

System.out.println(Type.BELLETRE.ordinal());  //1

Автоматически предопределённый метод для перечисления **valueOf()** возвращает константу перечисления, значение которой соответствует строке, переданной в параметре.

enum Cat {

Leopard, Puma, Lion, Tiger, Manul

}

public void onClick(View v) {

Cat cat;

cat = Cat.valueOf("Puma");

infoTextView.setText(cat.toString());

}

**Метод compareTo().**

Также можно сравнивать порядковые значения констант одного и того же перечисления с помощью метода **compareTo().** С помощью метода int compareTo(типПеречисления e) можно сравнить порядковые значения двух констант одного и того же перечислимого типа. Метод возвращает значение типа int.

Если порядковое значение вызывающей константы меньше, чем у константы е (this < e), то метод compareTo() возвращает отрицательное значение.

Если порядковые значения обеих констант одинаковы (this == e), возвращается нуль.

Если порядковое значение вызывающей константы больше, чем у константы е (this > e), то возвращается положительное значение.

**Метод equals()**

Вызвав метод equals(), переопределяющий аналогичный метод из класса Object, можно сравнить на равенство константу перечисления с любым другим объектом. Но оба эти объекты будут равны только в том случае, если они ссылаются на одну и ту же константу из одного и того же перечисления. Простое совпадение порядковых значений не вынудит метод equals() возвратить логическое значение true, если две константы относятся к разным перечислениям.

При сравнении констант перечислений, можно использовать оператор "==" - он будет работать так же, как и метод equals().

**ВОЗМОЖНОСТИ ПЕРЕЧИСЛЕНИЯ**

Создать экземпляр перечисления с помощью оператора new нельзя, но в остальном перечисление обладает всеми возможностями, которые имеются у других классов. А именно - в перечисления можно добавлять конструкторы, переменные и методы. Конструкторы перечисления являются private по умолчанию.

Допустим мы хотим задать размер нашей чашки кофе в миллилитрах. Для этого введем переменную ml в перечисление и геттер метод getMl(). Добавим конструктор, на вход которого будем задавать значение для миллилитров.

Обратите внимание, что при объявлении конструктора не указан модификатор доступа - он private по умолчанию. Уже говорилось, что нельзя создавать объекты перечисления используя оператор new. Как же тогда вызвать наш конструктор? Для вызова конструктора перечисления после указания константы ставятся круглые скобки, в которых передается нужное значение.

public enum CoffeeSize2 {

// 100, 150 и 200 передаются в конструктор

BIG(100), HUGE(150), OVERWHELMING(200);

private int ml;

CoffeeSize2(int ml) {

this.ml = ml;

}

public int getMl() {

return ml;

}

}

Методы для перечисления вызываются так же, как и для обычного объекта. В следующем классе мы перебираем все константы нашего перечисления и для каждого вызываем метод getMl():

public class CoffeeSizeMlDemo {

public static void main(String[] args) {

for (CoffeeSize2 coffeeSize : CoffeeSize2.values()) {

System.out.println(coffeeSize + " " + coffeeSize.getMl());

}

}

}

Конструкторы в перечислении могут быть перегружены, как показано в следующем примере. Для вызова конструктора без параметров просто не пишите скобки после константы:

public enum CoffeeSize3 {

BIG(100), HUGE, OVERWHELMING(200);

private int ml;

CoffeeSize3(int ml) {

this.ml = ml;

}

CoffeeSize3() {

this.ml = -1;

}

public int getMl() {

return ml;

}

}

**ОБЪЯВЛЕНИЕ ПЕРЕЧИСЛЕНИЙ**

Перечисления могут быть объявлены: отдельным классом или как член класса. Но НЕ могут быть объявлены внутри метода.

В этом пример перечисление CoffeeSize объявлено внутри класса Coffee3:

public class Coffee3 {

enum CoffeeSize { BIG, HUGE, OVERWHELMING }

CoffeeSize size;

}

Для обращения к такому перечислению необходимо использовать имя внешнего класс:

public class CoffeeTest2 {

public static void main(String[] args) {

Coffee3 drink = new Coffee3();

drink.size = Coffee3.CoffeeSize.BIG; // имя внешнего класса необходимо

}

}

public class CoffeeTest3 {

public static void main(String[] args) {

// Неправильно! Нельзя объявлять перечисления внутри метода!

/\*enum CoffeeSize {BIG, HUGE, OVERWHELMING}

Coffee drink = new Coffee();

drink.size = CoffeeSize.BIG;\*/

}

}

**ПЕРЕОПРЕДЕЛЕНИЕ МЕТОДОВ**

Для перечислений можно переопределять методы, но это не совсем обычное переопределение.

Добавим в наше перечисление метод getLid(), который возвращает код крышки для чашки кофе. Для всех констант подходит код B, который возвращает этот метод, кроме константы OVERWHELMING. Для OVERWHELMING чашки нужен код A. Переопределим метод getLid() для этой константы. Как это делается? После объявления константы открываем фигурные скобки, в которых и переопределяем этот метод. Если необходимо переопределить несколько методов, это делается в этих же фигурных скобках.

public enum CoffeeSize4 {

BIG(100),

HUGE(150),

OVERWHELMING(200) {

@Override

public String getLidCode() {

return "A";

}

};

private int ml;

CoffeeSize4(int ml) {

this.ml = ml;

}

public int getMl() {

return ml;

}

public String getLidCode() {

return "B";

}

}

ПЕРЕЧИСЛЕНИЯ И ИНТЕРФЕЙСЫ

Перечисления не могут наследовать другие классы, но могут реализовывать интерфейсы. Например, следующее перечисление реализует интерфейс *Runnable*:

public enum Currency implements Runnable {

PENNY(1), NICKLE(5), DIME(10), QUARTER(25);

private int value;

Currency(int value) {

this.value = value;

}

@Override

public void run() {

System.out.println("Перечисления в Java могут реализовывать интерфейсы");

}

}

**2. Можно ли самостоятельно создать экземпляр перечисления? А ссылку типа перечисления? Как сравнить, что в двух переменных содержится один и тот же элемент перечисления и почему именно так?**

Попытка явного создания экземпляра типа enum является ошибкой времени компиляции. Метод окончательного клонирования в Enum гарантирует, что константы enum никогда не могут быть клонированы, а специальная обработка механизмом сериализации гарантирует, что повторяющиеся экземпляры никогда не будут созданы в результате десериализации. Отражательная инстанциация типов перечисления запрещена. Вместе эти четыре вещи гарантируют, что никакие экземпляры типа перечисления не существуют за пределами тех, которые определены константами перечисления.

enum Cat {

Leopard, Puma, Lion, Tiger, Manul

}

public void onClick(View v) {

Cat[] allcats = Cat.values();

for(Cat cat : allcats) {

System.out.println(cat);

}

}

Для примера использовалась дополнительная переменная allcats, которой присваивается ссылка на массив перечислимых значений. Можно обойтись без дополнительной переменной.

for(Cat cat : Cat.values()) {

System.out.println(cat);

}

При сравнении констант перечислений, можно использовать оператор "==" - он будет работать так же, как и метод equals().

**3. Что такое анонимные классы?**

Анонимные классы - это классы, что не имеют имени и их создание происходит в момент инициализации объекта.  Достаточно интересная возможность, которой многие пользуются. В этом случае вы сразу создаете объект и класс. Еще раз использовать этот класс внутри своего кода вы не сможете. Вообще, анонимный класс создается на основе какого-то класса или интерфейса, и сразу в этом же кусочке кода вы переопределяете (в случае с интерфейсом – реализуете) нужный метод.

Анонимный класс — это полноценный внутренний класс. Поэтому у него есть доступ к переменным внешнего класса, в том числе к статическим и приватным. Есть у них кое-что общее и с локальными классами: они видны только внутри того метода, в котором определены. И еще одно важное ограничение, которое досталось анонимным классам от их «предков» — внутренних классов: анонимный класс не может содержать статические переменные и методы.

Реализация интерфейса:

**public** **class** **Foo** **{**

*// Анонимный класс, который реализует интерфейс SayHello*

**static** **SayHello** h **=** **new** **SayHello()** **{**

**@Override**

**public** **void** **say()** **{**

**System.**out**.**println**(**"Метод внутреннего анонимного класса"**);**

**}**

**};**

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

h**.**say**();**

**}**

**}**

*// somewhere*

**interface** **SayHello** **{**

**void** **say();**

**}**

Наследник определенного класса:

**public** **class** **External** **{**

*// Анонимный класс наследуется от класса Foo*

**static** **Foo** foo **=** **new** **Foo()** **{**

**@Override**

**public** **void** **show()** **{**

**super.**show**();**

**System.**out**.**println**(**"Метод внутреннего анонимного класса"**);**

**}**

**};**

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

foo**.**show**();**

**}**

**}**

***4. Что такое параметризованные классы? Для чего они необходимы? Приведите пример параметризованного класса и пример создания объекта параметризованного класса? Объясните, ссылки какого типа могут ссылаться на объекты параметризованных классов? Можно ли создать объект, параметризовав его примитивным типом данных?***

Дженерики (или обобщения) - это параметризованные типы.

Параметризованные типы позволяют объявлять классы, интерфейсы и методы, где тип данных, которыми они оперируют, указан в виде параметра. Используя дженерики, можно создать единственный класс, например, который будет автоматически работать с разными типами данных.

Классы, интерфейсы или методы, имеющие дело с параметризованными типами, называются параметризованными или обобщениями, параметризованными (обобщенными) классами или параметризованными (обобщёнными) методами.

Обобщения добавили в язык безопасность типов.

**ПАРАМЕТРИЗОВАННЫЕ КЛАССЫ**

Следующий пример демонстрирует использование параметризованного класса, который описывает матрицу:

public class Matrix<T> {

private T[] array;

public Matrix(T[] array) {

this.array = array.clone();

}

public static void main(String[] args) {

Matrix<Double> doubleMatrix = new Matrix<>(new Double[2]);

Matrix<Integer> integerMatrix = new Matrix<>(new Integer[4]);

Matrix<Byte> byteMatrix = new Matrix<>(new Byte[7]);

}

}

В объявлении Matrix<Integer> integerMatrix Integer является аргументом типа.

Java не создает разные версии класса Matrix или любого другого параметризованного класса. Имеется только одна версия класса Matrix, которая существует в прикладной программе.

Дженерики работают только с объектами! Следующий код является неправильным:

Gen<int> strOb = new Gen<int> (53);

// Ошибка, нельзя использовать примитивные типы

Т обозначает имя параметра типа. Это имя используется в качестве заполнителя вместо которого в дальнейшем подставляется имя конкретного типа, передаваемого классу Matrix при создании объекта. Это означает, что обозначение Т применяется в классе Matrix всякий раз, когда требуется параметр типа. Всякий раз, когда объявляется параметр типа, он указывается в угловых скобках.

Обобщенные типы отличаются в зависимости от типов-аргументов. Следующий код не допустим:

doubleMatrix = integerMatrix; // Не верно!

Несмотря на то, что doubleMatrix и integerMatrix имеют тип Matrix<T>, они являются ссылками на разные типы, потому что типы их параметров отличаются.

Обобщенный класс может быть объявлен с любым количеством параметров типа. Например:

public class TwoGen<T, V> {

private T obT;

private V obV;

public TwoGen(T obT, V obV) {

this.obT = obT;

this.obV = obV;

}

public void showTypes() {

System.out.println("Тип T: " + obT.getClass().getName());

System.out.println("Тип V: " + obV.getClass().getName());

}

public T getObT() {

return obT;

}

public V getObV() {

return obV;

}

}

public class SimpleGen {

public static void main(String[] args) {

TwoGen<Integer, String> twoGen = new TwoGen<>(88, "Generics");

twoGen.showTypes();

System.out.println("Значение T: " + twoGen.getObT());

System.out.println("Значение V: " + twoGen.getObV());

}

}

**ОГРАНИЧЕННЫЕ ТИПЫ**

Указывая параметр типа, можно наложить ограничение сверху в виде верхней границы, где объявляется супер класс, от которого должны быть унаследованы все аргументы типов. С этой целью вместе с параметром указывается ключевое слово extends:

class Gen <Т extends Superclass>

Параметр типа Т может быть заменен только указанным супер классом или его подклассами.

Рассмотрим пример использования ограниченного типа:

public class Average<T extends Number> {

private T[] array;

public Average(T[] array) {

this.array = array;

}

public double average() {

double sum = 0.0;

for (T value : array) {

sum += value.doubleValue();

}

return sum / array.length;

}

}

public class AverageDemo {

public static void main(String[] args) {

Integer[] intArray = {1, 2, 3, 4, 5};

Average<Integer> integerAverage = new Average<>(intArray);

System.out.println("Среднее значения для Integer " + integerAverage.average());

Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5};

Average<Double> doubleAverage = new Average<>(doubleArray);

System.out.println("Среднее значения для Double " + doubleAverage.average());

// Не откомпилируется,

// потому что String не является наследником Number

/\* String[] strArray = {"1", "2", "3", "4", "5"};

Average<String> strAverage = new Average<>(strArray);

System.out.println("Среднее значения для String " + strAverage.average());\*/

}

}

В виде ограничения можно накладывать не только тип класса, но и тип интерфейса:

public class MyClass<T extends Serializable>

Ограничение может включать в себя как тип класса, так и типы одного или нескольких интерфейсов:

class Gen <T extends MyClass & MyInterface1 & MyInterface2>

Тип класса должен быть задан первым. Накладывая на обобщенный тип ограничение, состоящее из класса и одного или нескольких интерфейсов, для их объединения следует воспользоваться логической операцией &: Таким образом, любой тип, передаваемый параметру Т, должен быть подклассом, производным от класса MyClass и реализующим интерфейсы MyInterface1 и MyInterface2.

**ИЕРАРХИИ ПАРАМЕТРИЗОВАННЫХ КЛАССОВ**

Параметризованные классы могут быть частью иерархии классов так же, как и любые другие не параметризованные классы. То есть параметризованный класс может выступать в качестве супер класса или подкласса.

Ключевое отличие между параметризованными и не параметризованными иерархиями состоит в том, что в параметризованной иерархии любые аргументы типов, необходимые параметризованному супер классу, всеми подклассами должны передаваться по иерархии вверх.

Например:

public class GenericSuper<T> {

private T ob;

public GenericSuper(T ob) {

this.ob = ob;

}

private T getOb() {

return ob;

}

}

public class GenericSub<T> extends GenericSuper<T> {

public GenericSub(T ob) {

super(ob);

}

}

Подкласс параметризованного супер класса необязательно должен быть параметризованным, но в нем все же должны быть, указаны параметры типа, требующиеся его параметризованному супер классу. Подкласс может, если требуется, быть, дополнен и своими параметрами типа. Супер классом для параметризованного класса может быть класс не параметризованный.

**ОГРАНИЧЕНИЯ ПРИСУЩИЕ ОБОБЩЕНИЯМ**

Обобщениям присущи некоторые ограничения. Рассмотрим их:

1. Нельзя создавать экземпляр по параметру типа. Ни обычный объект, ни массив:

public class GenRestriction<T> {

private T ob;

private T[] array;

public GenRestriction(T ob, T[] array) {

// Недопустимо!!!

//оb = new Т();

//array = new Т[10];

this.ob = ob;

this.array = array;

}

}

2. Нельзя создать массив специфических для типа обобщенных ссылок:

public class GenArrays {

public static void main(String[] args) {

// Нельзя создать массив специфичных для типа обобщенных ссылок.

// GenericSub<Integer>[] gens = new GenericSub<Integer>[10];

GenericSub<?>[] gens = new GenericSub<?>[10];

gens[0] = new GenericSub<>(34);

}

}

3. Нельзя создавать обобщенные статические переменные и методы. Но объявить статические обобщенные методы со своими параметрами типа все же можно:

public class GenericWrongStatic<T> {

// Неверно, нельзя создать статические переменные типа Т.

//public static Т оb;

// Неверно, ни один статический метод не может использовать Т.

/\* public static T getOb() {

return оb;

}\*/

//Но объявить статические обобщенные методы со своими параметрами типа можно

public static <V> void getOb(V v) {

System.out.println(v);

}

}

**Exceptions and Errors**

1. Что для программы является исключительной ситуацией? Какие способы обработки ошибок в программах вы знаете?
2. Что такое исключение для Java-программы? Что значит “программа выбросила исключение”? Опишите ситуации, когда исключения выбрасываются виртуальной машиной(автоматически), и когда необходимо их выбрасывать вручную?
3. Приведите иерархию классов-исключений, делящую исключения на проверяемые и непроверяемые. В чем особенности проверяемых и непроверяемых исключений?
4. Объясните работу оператора try-catch-finally. Когда данный оператор следует использовать? Сколько блоков catch может соответствовать одному блоку try? Можно ли вкладывать блоки try друг в друга, можно ли вложить блок try в catch или finally? Как происходит обработка исключений, выброшенных внутренним блоком try, если среди его блоков catch нет подходящего? Что называют стеком операторов try? Как работает блок try с ресурсами.
5. Укажите правило расположения блоков catch в зависимости от типов перехватываемых исключений. Может ли перехваченной исключение быть сгенерировано снова, и, если да, то как и кто в этом случае будет обрабатывать повторно сгенерированное исключение? Может ли блок catch выбрасывать иные исключения, и если да, то опишите ситуацию, когда это может быть необходимо.
6. Когда происходит вызов блока finally? Существуют ли ситуации, когда блок finally не будет вызван? Может ли блок finally выбрасывать исключений? Может ли блок finally выполнится дважды?

**1. Что для программы является исключительной ситуацией? Какие способы обработки ошибок в программах вы знаете?**

Исключительная ситуация— это ошибка, происходящая во время выполнения программы (так называемая ошибка времени выполнения). [Основные типы исключений (Exception) в java](https://www.fandroid.info/osnovnye-tipy-isklyuchenij-exception-v-java/).

Обработка ошибок в Java основана на использовании в программе следующих ключевых слов:

* try – определяет блок кода, в котором может произойти исключение;
* catch – определяет блок кода, в котором происходит обработка исключения;
* finally – определяет блок кода, который является необязательным, но при его наличии выполняется в любом случае независимо от результатов выполнения блока try.

Эти ключевые слова используются для создания в программном коде специальных обрабатывающих конструкций: try{}catch, try{}catch{}finally, try{}finally{}.

* throw – используется для возбуждения исключения;
* throws – используется в сигнатуре методов для предупреждения, о том что метод может выбросить исключение.

**2. Что такое исключение для Java-программы? Что значит “программа выбросила исключение”? Опишите ситуации, когда исключения выбрасываются виртуальной машиной(автоматически), и когда необходимо их выбрасывать вручную?**

В мире программирования возникновение ошибок и непредвиденных ситуаций при выполнении программы называют **исключением**. В программе исключения могут возникать в результате неправильных действий пользователя, отсутствии необходимого ресурса на диске, или потери соединения с сервером по сети. Причинами исключений при выполнении программы также могут быть ошибки программирования или неправильное использование API. В отличие от нашего мира, программа должна четко знать, как поступать в такой ситуации. Для этого в Java предусмотрен механизм исключений.

Возможность предупреждения и разрешения исключительной ситуации в программе для ее продолжения – одна из причин использования исключений в Java. Механизм исключений также позволяет защитить написанный вами код (программный интерфейс) от неправильного использования пользователем за счет валидации (проверки) входящих данных.

На стадии разработки программы мы «ограждаем» опасные участки кода в отношении исключений с помощью блока try{}, предусматриваем «запасные» пути с помощью блока catch{}, в блоке finally{} мы пишем код, который выполняется в программе при любом исходе. В случаях, когда мы не можем предусмотреть «запасной путь» или намеренно хотим предоставить право его выбора пользователю, мы должны, по крайней мере, предупредить его об опасности.

В программировании при написании своих классов и методов мы не всегда можем предвидеть контекст их использования другими разработчиками в своих программах, поэтому не можем предвидеть на 100% правильный путь для разрешения исключительной ситуации. В то же время, правило хорошего тона — предупредить пользователей нашего кода о возможности исключительной ситуации. Механизм исключений Java позволяет нам сделать это с помощью throws – по сути, объявления общего поведения нашего метода, заключающееся в выбрасывании исключения, и предоставляя, таким образом, написание кода по обработке исключения в Java пользователю метода.

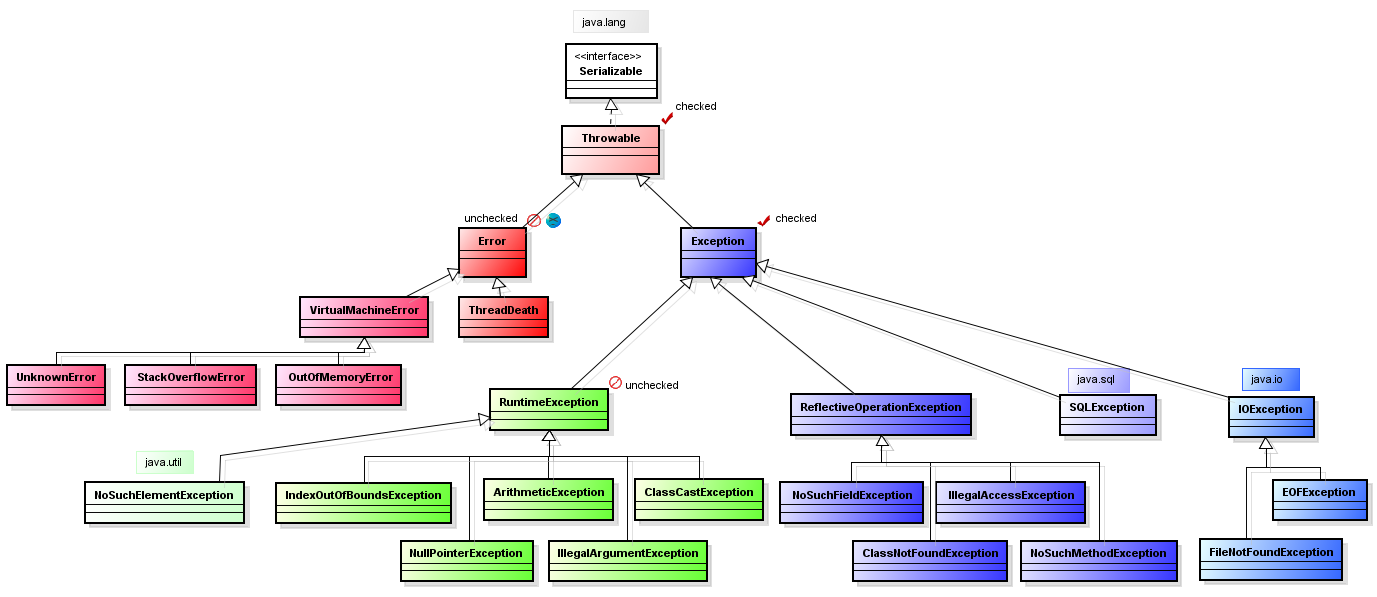
**“программа выбросила исключение” –** возможные исключения были обработаны и одно из них наступило, тогда в консоли выводится сожержание ощшибки/исключения.

При исполнении программы исключение генерируется JVM или вручную, с помощью оператора throw. При этом в памяти создается объект исключения и выполнение основного кода программы прерывается, а обработчик исключений JVM пытается найти способ обработать исключение.

Автоматическая генерация. Если Java-машина обнаруживает некоторую ошибку, например, деление на ноль или ошибку приведения типов, то она сама генерирует соответствующее исключение.  Error и RuntimeException и все их наследники — unchecked checked exception — проверяемое исключение, проверяемое компилятором.

Обработка исключений вручную происходит по средства try{}catch, try{}catch{}finally, try{}finally{} или throws.

***3. Приведите иерархию классов-исключений, делящую исключения на проверяемые и непроверяемые. В чем особенности проверяемых и непроверяемых исключений?***



У всех исключений есть общий класс-предок Throwable. От него происходят две большие группы — исключения (Exception) и ошибки (Error).

Error — это критическая ошибка во время исполнения программы, связанная с работой виртуальной машины Java. В большинстве случаев Error не нужно обрабатывать, поскольку она свидетельствует о каких-то серьезных недоработках в коде.

Exceptions — это, собственно, исключения: исключительная, незапланированная ситуация, которая произошла при работе программы. Это не такие серьезные ошибки, как Error, но они требуют нашего внимания.

Все исключения делятся на 2 вида — проверяемые (checked) и непроверяемые (unchecked).

1. Checked исключения, это те, которые должны обрабатываться блоком catch или описываться в сигнатуре метода. Unchecked могут не обрабатываться и не быть описанными.

2. Unchecked исключения в Java – наследованные от RuntimeException, checked – от Exception (не включая unchecked).

***4. Объясните работу оператора try-catch-finally. Когда данный оператор следует использовать? Сколько блоков catch может соответствовать одному блоку try? Можно ли вкладывать блоки try друг в друга, можно ли вложить блок try в catch или finally? Как происходит обработка исключений, выброшенных внутренним блоком try, если среди его блоков catch нет подходящего? Что называют стеком операторов try? Как работает блок try с ресурсами.***

Общая форма:

public class ExceptionDemo1 {

public static void main(String[] args) {

try {

// блок кода, в котором отслеживаются ошибки

} catch (ExceptionType1 e) {

// обработчик исключений типа ExceptionType1

} catch (ExceptionType2 e) {

// обработчик исключений типа ExceptionType2

} finally {

// блок кода, который должен быть выполнен после завершения блока try

}

}

}

В блоке try размещаются операторы программы, которые нужно проконтролировать и, в случае возникновения исключительной ситуации, сгенерировать исключение. Внутри блока try могут быть вызваны различные методы, способные сгенерировать то или иное исключение. Однако, обработчик исключения будет только один.

В некоторых случаях один фрагмент кода может инициировать более одного исключения. Используется два или более операторов catch, каждый для перехвата своего типа исключений.

Когда возбуждается исключение, каждый оператор catch проверяется по порядку, и первый из них, чей тип соответствует исключению, выполняется.

После того как выполнится один из операторов catch, все остальные пропускаются, и выполнение программы продолжается с места, следующего за блоком try-catch.

Когда используются множественные операторы catch, важно помнить, что подклассы исключений должны следовать перед любыми их супер классами. Это потому, что оператор catch, который использует супер класс, будет перехватывать все исключения этого супер класса плюс всех его подклассов. То есть подкласс исключения никогда не будет обработан, если вы попытаетесь его перехватить после его супер класса. Более того, в Java недостижимый код является ошибкой.

**Объявление нескольких исключений в одном блоке catch (multi-catch блок)**

try {

...

} catch(IllegalStateException | SQLException | ContextException e){

System.out.println(e.getMessage());

}

Параметр оператора catch должен быть фактически завершенным (effective final). Это означает, что ему нельзя присваивать новое значение в блоке оператора catch или же он должен быть явно объявлен как final.

Multi-catch блок может содержать только исключения не входящие в одну иерархию.

В блоке finally указывается код, который должен быть обязательно выполнен после завершения блока try. Блок операторов finally выполняется независимо от того, будет ли сгенерировано исключение или нет. Если исключение сгенерировано, блок операторов finally выполняется даже при условии, что ни один из операторов catch не совпадает с этим исключением.

Операторы try и catch составляют единое целое. Оператор finally может отсутствовать.

Операторы try можно вкладывать друг в друга. Если у оператора try низкого уровня нет раздела catch, соответствующего возникшему исключению поиск будет развернут на одну ступень выше, и будут проверены разделы catch внешнего оператора try.

**package** by.bsu.exception;

**public class** NestedTryCatchRunner {

**public void** doAction() {

**try** { *// внешний блок*

**int** a = (**int**) (Math.*random*() \* 2) — 1;

System.*out*.println("a = " + a);

**try** { *// внутренний блок*

**int** b = 1/a;

StringBuilder sb = **new** StringBuilder(a);

} **catch** (NegativeArraySizeException e) {

System.*err*.println("недопустимый размер буфера: " + e);

}

} **catch** (ArithmeticException e) {

System.*err*.println("деление на 0: " + e);

}

}

}

В результате запуска приложения при a=0 будет сгенерировано исключение ArithmeticException, а подходящий для его обработки блок try–catch является внешним по отношению к месту генерации исключения. Этот блок и будет задействован для обработки возникшей исключительной ситуации. Вкладывание блоков try-catch друг в друга загромождает код, поэтому такими конструкциями следует пользоваться с осторожностью.

При использовании блока try-catch вначале выполняются все инструкции между операторами try и catch. Если в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается и переходит к инструкции сatch.

Выражение catch имеет следующий синтаксис: catch (тип\_исключения имя\_переменной). Объявляется переменная ex, которая имеет тип Exception. Но если возникшее исключение не является исключением типа, указанного в инструкции сatch, то оно не обрабатывается, а программа просто зависает или выбрасывает сообщение об ошибке.

Но так как тип Exception является базовым классом для всех исключений, то выражение catch (Exception ex) будет обрабатывать практически все исключения. Обработка же исключения в данном случае сводится к выводу на консоль стека трассировки ошибки с помощью метода printStackTrace(), определенного в классе Exception.

После завершения выполнения блока catch программа продолжает свою работу, выполняя все остальные инструкции после блока catch.

Операторы try могут быть вложенными. То есть оператор try может находиться внутри блока другого try. Всякий раз, когда управление попадает в блок try, контекст этого исключения заталкивается в стек. Если вложенный оператор try не имеет обработчика catch для определенного исключения, стек "раскручивается" и проверяются на соответствие обработчики catch следующего (внешнего) блока try. Это продолжается до тех пор, пока не будет найден подходящий оператор catch либо пока не будут проверены все уровни вложенных try. Если подходящий оператор catch не будет найден, то исключение обработает система времени выполнения Java.

Начиная с седьмой версии Java предлагает улучшенное управление ресурсами, которые должны быть закрыты после окончания работы с ними. К таким ресурсам относятся, например, файлы потоки, соединения с базами данных и сокетами. Этой цели служит специальная языковая конструкция try-with-resources. Для того чтобы это автоматическое закрытие работало создан специальный интерфейс AutoCloseable.

В Java 7 все классы ресурсов реализуют этот интерфейс. Его сигнатура выглядит следующим образом:

**public** **interface** AutoCloseable {

**void** close() **throws** Exception;

}

Интерфейс объявляет метод close(), который автоматически вызывается для объектов, обслуживаемых конструкцией try-with-resources.

Оператор try-c-ресурсами реализует принцип **автоматического управления ресурсами**, целью которого является избежать, например, утечек памяти, в случаях когда ресурс по каким-то причинам не освобождается, если он больше не нужен.

Неудачный исход закрытия файла может привести к "утечкам памяти", поскольку неиспользуемые ресурсы оперативной памяти останутся выделенными.([стр 365](http://www.ozon.ru/context/detail/id/31249554/))

*try ( FileInputStream res = new FileInputStream(args[O])) {*

*//использование ресурса*

*}*

Оператор try-c-ресурсами позволяет объявить и проинициализировать ресурс (в круглых скобках после оператора try), создав переменной ресурса локальный контекст в блоке try. По завершении этого блока переменная удаляется, а значит и ресурс автоматически закрывается.

Отсюда отпадает необходимость явного закрытия ресурса методом close() в блоке оператора finally.

**5. Укажите правило расположения блоков catch в зависимости от типов перехватываемых исключений. Может ли перехваченной исключение быть сгенерировано снова, и, если да, то как и кто в этом случае будет обрабатывать повторно сгенерированное исключение? Может ли блок catch выбрасывать иные исключения, и если да, то опишите ситуацию, когда это может быть необходимо.**

В некоторых случаях один фрагмент кода может инициировать более одного исключения. Используется два или более операторов catch, каждый для перехвата своего типа исключений.

Когда возбуждается исключение, каждый оператор catch проверяется по порядку, и первый из них, чей тип соответствует исключению, выполняется.

После того как выполнится один из операторов catch, все остальные пропускаются, и выполнение программы продолжается с места, следующего за блоком try-catch.

Когда используются множественные операторы catch, важно помнить, что подклассы исключений должны следовать перед любыми их супер классами. Это потому, что оператор catch, который использует супер класс, будет перехватывать все исключения этого супер класса плюс всех его подклассов. То есть подкласс исключения никогда не будет обработан, если вы попытаетесь его перехватить после его супер класса. Более того, в Java недостижимый код является ошибкой.

**Повторная генерация исключений**

Внутри определенного фрейма стека может понадобиться перехватить все исключения или же определенное их подмножество, выполнить некоторую очистку и затем заново сгенерировать исключение, чтобы позволить ему дальше распространяться по стеку.

Для реализации сказанного используется оператор throw без параметров:

|  |
| --- |
| using System;  using System.Collections;  public class Entrypoint  {  static void Main() {  try {  try {  ArrayList list = new ArrayList ();  list. Add ( 1 ) ;  Console.WriteLine( "Элемент 10 = {0}", list[10] );  }  catch( ArgumentOutOfRangeException ) {  Console.WriteLine ( "Выполнить полезную работу и" +  " повторить исключение" );  // Заново сгенерировать перехваченное исключение, throw;  }  finally {  Console.WriteLine ( "Очистка..." );  }  }  catch {  Console.WriteLine ( "Готово" );  }  }  } |

Обратите внимание, что любые блоки finally, связанные с фреймом исключения, с которым ассоциирован блок catch, будут выполнены перед выполнением обработчиков любых исключений более высокого уровня. Это видно в выводе показанного выше кода:

Такой стиль работы довольно неуклюж, поскольку при этом требуется следить за правильной повторной генерацией исключений. Если вдруг вы нечаянно забудете это сделать, то вряд ли справитесь с исключительной ситуацией.

Продемонстрированные приемы помогут достичь цели, когда единственным местом, где помещается блок catch, будет то, где может возникнуть действие по исправлению ситуации.

Иногда необходимо “транслировать” исключение внутри обработчика исключений. В этом случае перехватывается исключение одного типа, но затем генерируется исключение другого типа — возможно, более точного — в блоке catch для передачи его на обработку на следующем уровне.

Рассмотрим приведенный ниже пример:

|  |
| --- |
| using System;  using System.Collections;  public class MyException : Exception  {  public MyException ( String reason. Exception inner ) :base( reason, inner ) {  }  }  public class Entrypoint {  static void Main() {  try {  try {  ArrayList list = new ArrayList ();  list. Add ( 1 ) ;  Console.WriteLine( "Элемент 10 = {0}", list[10] );  }  catch( ArgumentOutOfRangeException x ) {  Console.WriteLine ( "Выполнить полезную работу и " + " повторить исключение" );  throw new MyException ( "Лучше сгенерировать исключение", х ) ;  }  finally {  Console.WriteLine( "Очистка..." );  }  }  catch ( Exception x ) {  Console.WriteLine ( x );  Console.WriteLine ( "Готово" );  }  }  } |

Одним особым качеством типа System.Exception является способность включать в себя ссылку на вложенное исключение через свойство Exception. InnerException. Таким образом, когда генерируется новое исключение, вы можете предохранить цепочку исключений для исключений, обрабатывающих их. Я рекомендую использовать это полезное свойство стандартного типа исключений С# для трансляции ваших исключений.

Вывод предыдущего кода выглядит следующим образом:

Выполнить полезную работу и повторить исключение

Очистка…

MyException: Лучше сгенерировать исключение —> System.ArgumentOutOfRangeException:

Index was out of range.

Must be non-negative and less than the size of the collection.

Parameter name: index

at System.Collections.ArrayList.get\_Item(Int32 index)

at Entrypoint.Main()

– End of inner exception stack trace

at Entrypoint.Main()

*System.ArgumentOutOfRangeException:*

*Выход индекса за пределы диапазона.*

*Значение должно быть положительным и меньше размера коллекции.*

*Имя параметра: index*

*в System.Collections.ArrayList.get\_Item (Int32 index)*

*в Entrypoint.Main ()*

*– Конец трассировки стека внутреннего исключения*

*–в Entrypoint.Main()*

Готово

Имейте в виду, что трансляции исключений по возможности следует избегать. Чем больше исключений перехватывается и генерируется заново в стеке, тем больше код, обрабатывающий исключение, будет изолирован от кода, его генерирующего. То есть становится трудным сопоставить точку перехвата с исходной точкой генерации исключения. Да, свойство Exception.

InnerException помогает смягчить проблему, но все равно трудно найти изначальную причину проблемы, если исключения транслируются по пути.

**Исключение можно генерировать** в блоке catch, создавая тем самым цепочку исключений. Обычно разработчики поступают так в том случае, если им надо изменить тип исключения. Если вы создаете подсистему, используемую другими программистами, имеет смысл генерировать такие исключения, которые дали бы возможность сразу определить, что ошибка возникла именно в ней. В качестве примера подобного исключения можно привести **ServletException**.

Возможно, что контейнеру сервлетов и не обязательно иметь информацию о том, какая именно возникла ошибка, а важно лишь знать, что некорректно работает **сервлет**.

Ниже показан пример перехвата исключения и повторной его генерации.

|  |
| --- |
| try  {  Обращение к базе данных  }  catch(SQLException e)  {  throw new ServletException("database error: " + e.getMessage());  } |

В данном случае в конструкторе ServletException формируется сообщение. В Java SE 1.4 вы можете даже указать исходное исключение, сообщив, что она стало причиной нового.

|  |
| --- |
| try  {  access the database  }  catch(SQLException e)  {  Throwable se = new ServletException("database error");  se.setCause(e);  throw se;  } |

Теперь при перехвате исключения можно извлечь исходное исключение так, как показано ниже:

|  |  |
| --- | --- |
| 1 | Throwable e = se.getCause(); |

Специалисты настоятельно рекомендуют использовать при разработке именно такой подход. Он позволяет генерировать исключение более высокого уровня, не теряя деталей исходного исключения.

**Совет!** Описанный подход удобен в том случае, если перехват исключения осуществляется в методе, который не имеет права генерировать контролируемые исключения. Вы можете перехватить контролируемое исключение и инкапсулировать его в составе исключения времени выполнения.

В ряде классов, описывающих исключения, например, в **ClassNotFoundException**, **InvocationTargetException** и **RuntimeException**, поддерживается своя схема создания цепочки исключений. Начиная с версии [Java](http://java-code.ru/) SE 1.4, в них параллельно поддерживается стандартный механизм инкапсуляции исходного исключения. Таким образом, вы можете либо применять методы, типичные для этих классов, либо использовать метод getCause().

**6. Когда происходит вызов блока finally? Существуют ли ситуации, когда блок finally не будет вызван? Может ли блок finally выбрасывать исключений? Может ли блок finally выполнится дважды?**

Возможна ситуация, при которой нужно выполнить некоторые действия по завершению программы (закрыть поток, освободить соединение с базой данных) вне зависимости от того, произошло исключение или нет. В этом случае используется блок finally, который обязательно выполняется после инструкций try или catch.

Каждому разделу try должен соответствовать по крайней мере один раздел catch или блок finally. Блок finally часто используется для закрытия файлов и освобождения других ресурсов, захваченных для временного использования в начале выполнения метода. Код блока выполняется перед выходом из методадаже в том случае, если перед ним были выполнены инструкции вида return, break, continue.

**Существуют ли ситуации, когда блок finally не будет вызван?**

Блок finally выполняется не всегда, например в такой ситуации:

|  |  |
| --- | --- |
| 1  2  3  4  5 | try {      System.exit(0);  } catch(Exception e) {      e.printStackTrace();  } finally { } |

Здесь finally недостижим, так как происходит системный выход из программы. Общими словами: когда jvm умирает, ей не до finally

Единственными моментами, которые, finally, не будут называться, являются:

* Если вы вызываете System.exit();
* Если сначала произойдет сбой JVM;
* Если JVM достигает бесконечного цикла (или другого не прерывающегося, не заканчивающегося оператора) в блоке try или catch;
* Если ОС принудительно завершает процесс JVM; например, "убить -9" в UNIX.
* Если хост-система умирает; например, сбой питания, аппаратная ошибка, паника ОС и т.д.
* Если, наконец, блок будет выполняться потоком демона, а все остальные не-демонные потоки выходят до того, как, наконец, вызывается.

**Может ли блок finally выбрасывать исключений?**

Ответ аналогичный случаю с двумя return – будет обработано в finally блоке. Если было выброшено два исключения – одно в try, второе в finally, то исключение в finally “проглотит” исключение выше (см. пример). Если до блока finally исключение было обработано, то мы можем получить информацию об исключении в блоке try и тем самым не потерять исключение, которое впоследствии может быть перезаписано в finally другим исключением.