class Object

```
public class Object {
   public final Class getClass();
   public String toString();
   public boolean equals(Object obj);
   public int hashCode();
   protected Object clone() throws CloneNotSupportedException;
   public final void wait() throws IllegalMonitorStateException,
   InterruptedException;
   public final void wait(long millis)
       throws IllegalMonitorStateException, InterruptedException;
   public final void wait(long millis, int nanos)
       throws IllegalMonitorStateException, InterruptedException;
   public final void notify()
       throws IllegalMonitorStateException;
   public final void notifyAll()
          throws IllegalMonitorStateException;
    protected void finalize()throws Throwable;
}
```

class Object

• public final Class **getClass**()

Этот метод возвращает ссылку на уникальный объект типа Class, который представляет класс этого объекта.

• public String **toString**()

Возвращает строковое представление класса. По умолчанию возвращает:
имя класса + @+Хеш

•public boolean **equals**(Object obj)

Проверка на то, что текущий объект эквивалентен объекту obj. По умолчанию, true будет только в том случае, если obj ссылка на текущий объект. Реализация метода equals должна удовлетворять определенным требованиям

1. **Рефлективность:** Для любого ссылочного значения x, x.equals(x) должен возвратиться true.
2. **Симметричность:** Для любых ссылочных значений x и y, x.equals(y) должен возвратиться true только если y.equals(x) возвращает true.
3. **Транзитивность:** для любых ссылочных значений x, y, и z, если x.equals(y) возвращает true и y.equals(z) возвраты true, тогда x.equals(z) должен возвратить true.
4. **Непротиворечивость:** для любых ссылочных значений x и y если объекты не меняются то и резултата сравнения должен оставаться тем же самым.
5. Для любого ненулевого ссылочного значения x, x.equals(null) должен возвратиться false.

class Object

public int **hashCode**()

Возвращает хеш код объекта. Используется, например, в HashMap

Общие требования к hashCode следующие:

- Если объект не меняется то и хеш код не должен меняться в процессе работы программы. Целое число может быть положительным, отрицательным, или нуль.
- Если два объекта равны согласно equals, то hashCode должен привести к тому же самому целочисленному результату.
- При этом не требуется, чтобы если объекты были различными то были и различными хеш коды этих объектов. Однако, чем реже возникают коллизии, тем лучше для использования этого метода.

Класс String

Класс String предназначен для хранения строк. Основной особенностью класса является то, что при изменении начальной строки каждый раз создается новый объект.

```
String st = "Маша";
st += "Саша";
```

Создаст новый объект содержащий строку «МашаСаша» а исходные объекты будут уничтожены сборщиком мусора.

Если операций конкатенации над одним и тем же строковым объектом производится много, это приводит к интенсивному процессу порождения новых объектов и добавляет работы сборщику мусора.

По определению каждый объект класса String не может изменять-ся. Нельзя ни вставить новые символы в уже существующую строку, ни поменять в ней одни символы на другие. И добавить одну строку в конец другой тоже нельзя.

Класс String

```
public static void main(String args[]) {
  String test1 = "Today is ";
  appendTodaysDate(test1);
  System.out.println(test1); }
public static void appendTodaysDate(String line) {
    line = line + (new Date()).toString(); }
```

Правильный вариант

```
public static String appendTodaysDate(String line) {
   return (line + (new Date()).toString()); }
```

Правильный вариант №2

```
public static void main(String args[]) {
  StringBuffer test1 = new StringBuffer("Today is ");
  appendTodaysDate(test1);
  System.out.println(test1.toString()); }
public static void appendTodaysDate(StringBuffer line) {
  line.append((new Date()).toString()); }
```

Класс String

Получение символов и подстрок

```
String anotherPalindrome = "Niagara. O roar again!";
char aChar = anotherPalindrome.charAt(9);
```

Метод	Описание
String substring(int beginIndex, int endIndex)	Возвращает подстроку данной строки, начиная с символа с индексом beginIndex, заканчивая endIndex — 1.
String substring(int beginIndex)	Возвращает подстроку данной строки, начиная с символа под номером beginIndex и до конца строки.

```
String anotherPalindrome = "Niagara. O roar again!";
String roar = anotherPalindrome.substring(11, 15);
```

Класс String

Метод	Описание
String[] split(String regex) String[] split(String regex, int limit)	Ищет совпадения в строке согласно заданному регулярному выражению и разбивает строку на массив. Необязательный аргумент limit задает максимальный размер возвращаемого массива.
CharSequence subSequence(int beginIndex, int endIndex)	Возвращает последовательность символов, начиная с beginIndex, заканчивая endIndex - 1.
String trim()	Возвращает строку, в которой удалены лишние пробелы в начале строки и в конце.
String toLowerCase() String toUpperCase()	Возвращает копию строки, символы которой переведены в нижний или верхний регистр. Если преобразований не требуется возвращается оригинальная строка.

Класс String

Метод	Описание
int indexOf(int ch) int lastIndexOf(int ch)	Возвращает индекс первого(последнего) вхождения символа в строке.
int indexOf(int ch, int fromIndex) int lastIndexOf(int ch, int fromIndex)	Возвращает индекс первого(последнего) вхождения символа в строке, начиная поиск с указанного индекса.
int indexOf(String str) int lastIndexOf(String str)	Возвращает индекс первого(последнего) вхождения подстроки в строке.
int indexOf(String str, int fromIndex) int lastIndexOf(String str, int fromIndex)	Возвращает индекс первого(последнего) вхождения подстроки в строке, начиная поиск с указанного индекса.

Класс String

Метод	Описание
String replace(char oldChar, char newChar)	Возвращает новую строку, в которой все oldChar заменены на newChar.
String replace(CharSequence target, CharSequence replacement)	Заменяет все вхождения подстроки target на строку replacement.
String replaceAll(String regex, String replacement)	Заменяет все подстроки, которые описывает заданное регулярное выражение на replacement
String replaceFirst(String regex, String replacement)	Заменяет только первую подходящую подстроку.

Конструкторы классов

Конструктор - специальный метод, который вызывается при создании нового объекта.

В *Java* имя конструктора совпадает с именем класса.

```
class MyClass {
   MyClass() {
      System.out.print(“MyClass");
} }
public class Test {
  public static void main(String[] args)
   {
    for(int i = 0; i < 10; i++)
        new MyClass();
} }
```

```
class MyClass {
   int i;
   MyClass(int myInt) {
       i=myInt;
       System.out.print(“MyClass“+i);
} }
public class Test {
  public static void main(String[] args)
   { for(int i = 0; i < 10; i++) new MyClass(i); } }
```

Все необходимые для существования класса операции должны проводиться в конструкторе и не должны выноситься в методы, которые нужно вызывать отдельно. В *Java* создание и инициализация являются неразделимыми понятиями — одно без другого невозможно.

Перегрузка методов

Сигнатурой метода называется совокупность его имени и набора формальных параметров. Java позволяет создавать несколько методов с одинаковыми именами, но разными сигнатурами. Создание метода с тем же именем, но с другим набором параметров называется **перегрузкой**. Какой из перегруженных методов должен выполняться при вызове, Java определяет на основе фактических параметров:

```
void pr( double a) {
  System.out.println(a);
}
void pr (String a) {
  System.out.println(a);
}
void pr(int[] a) {
  for (int i=0; i<a.length; i++) {
    System.out.print(a[i]+" ")
  }
  System.out.println();
}
```

```
int a = 5;
int [] m = {1, 2, 8, 3}
String s = "Мир";
pr (a) //работает исходный метод
pr (a+s); // 5 мир, работает первая перегрузка
pr (m); // 1 2 8 3
```

Перегрузка по возвращаемым значениям не допускается

Конструкторы классов

Конструктором по умолчанию называется конструктор без аргументов, применяемый для создания «типового» объекта. Если созданный класс не имеет конструктора, компилятор автоматически добавит конструктор по умолчанию.

```
class MyClass{}
…..
MyClass mc = new MyClass();
```

```
class MyClass{ Myclass(int i){….}}
…..
MyClass mc = new MyClass();
```

Если существует хоть один конструктор(в том числе и с параметрами) то конструктор по умолчанию создаваться не будет.

Ключевое слово ***this*** может использоваться только внутри не-статического метода и предоставляет ссылку на объект, для которого был вызван метод.

```
class MyClass{
  int i;
  MyClass inc() {i++; return this}
}
```

```
class MyClass{
  int i;
  public void inc(){Increaser.inc(this);}}
class Increaser{
  public static void inc(MyClass mc) {mc.i++;}}
```

```
class MyClass{
  Myclass(int i){….}
  MyClass(String s, int i){this(i);…}}
```

Очистка памяти

В Java действует механизм сборки мусора.

Помимо динамической памяти, объект может владеть и другими ресурсами — подчас более ценными, чем память. Если объект при создании открывает файл, по завершении использования он должен его закрыть, если подключается к СУБД — должен отключиться. В системах с ручным управлением памятью это делается непосредственно перед удалением объекта из памяти, чаще всего — в деструкторах соответствующих объектов. В системах со сборкой мусора обычно есть возможность выполнить некоторый код непосредственно перед удалением объекта, так называемые финализаторы

Для таких ситуаций в *Java* предусмотрен метод ***finalize()***, который можно определить в классе.

Объекты могут быть и не переданы сборщику мусора.

Процесс сборки мусора относится только к памяти.

Инициализация

- Локальные переменные не инициализируются
- Поля класса инициализируются значениями по умолчанию
- Поля класса можно инициализировать вручную

```
class DataOnly {
  int i =15;
  double d =0.123;
  boolean b = true;}
```

```
class DataOnly {
  int i; double d;
  boolean b;
  DataOnly(){i=10;b=false;}}
```

- Поля класса можно инициализировать в конструкторе.
- Важно знать порядок инициализации:

```
class ClassA{ ClassA(int i){
      System.out.println("ClassA constructor " +i);}}
class ClassB{
   ClassA a3;
   ClassA a = new ClassA(1);
   ClassB(){
     System.out.println("ClassB constructor");
     a3= new ClassA(3);}
   ClassA a1= new ClassA(2);}
public class InitTest {
   public static void main(String[] args) throws IOException {
       ClassB b= new ClassB();}}
```

Инициализация

```
import java.io.IOException;
import java.util.Random;
class Initializer{
public static int initint(String text){
System.out.println(text);  return new Random(100).nextInt();}}
class ClassA{
public static int i4 = Initializer.initint("Init I4");
ClassA(int i){
  System.out.println("ClassA constructor " +i);}}
class ClassB{
public static int i1 = Initializer.initint("Init I1");
public static int i2;
static{
  i2 =Initializer.initint("Init I2"); }
ClassA a3;
ClassA a = new ClassA(1);
ClassB(){ System.out.println("ClassB constructor"); a3= new
ClassA(3);}
ClassA a1= new ClassA(2);
public static int i3 = Initializer.initint("Init I3");}
public class InitTest {
public static void main(String[] args) throws IOException {
ClassB b= new ClassB();}}
```

Инициализация массивов

Чтобы резервировать память для массива, необходимо записать некоторое выражение инициализации.

```
int[] a1 = { 1, 2, 3, 4, 5 };
int[] a2; a2 = a1;
```

Все массивы (как массивы примитивов, так и массивы объектов) содержат поле которое можно прочитать (но не изменить!) для получения количества элементов в массиве. Это поле называется ***length***. Нумерация элементов начинается с нуля, последнему элементу массива соответствует индекс ***length—1***. В *Java* при выходе за рамки массива происходит ошибка времени исполнения

```
int[] a = new int[rand.nextInt(20)];
Integer[] b = new Integer[]{
   new Integer(1), new Integer(2), 3,};
```

```
static void printArray(Object[] args) {
for(Object obj: args) System.out.print(obj + "
");}

printArray(new Object[]{ new Integer(47),
new Float(3.14), new Double(11.11) });
```

```
static void printArray(Object... args) {
for(Object obj : args)
System.out.print(obj + " ");}

printArray(47, 3.14F, 11.11);
printArray("one", "two", "three");
```

Спецификаторы доступа *Java*

В *Java* спецификаторы доступа ***public, protected*** и ***private*** располагаются перед определением членов классов — как полей, так и методов. Каждый спецификатор доступа управляет только одним отдельным определением.

Если спецификатор доступа не указан, используется **«пакетный» уровень доступа.** Это значит, что член класса доступен для всех остальных классов текущего пакета, но для классов за пределами пакета он воспринимается как приватный (private). Так как компилируемый модуль — файл — может принадлежать лишь одному пакету, все классы одного компилируемого модуля автоматически открыты друг для друга в границах пакета. Доступ в пределах пакета и является основной причиной для группировки классов в пакетах.

Спецификаторы доступа *Java*

Получить доступ можно лишь несколькими способами:

- Объявить член класса открытым (***public***), то есть доступным для кого угодно и откуда угодно.
- Сделать член класса доступным в пакете, не указывая другие спецификаторы доступа, и разместить другие классы в этом же пакете.
- Производный класс может получить доступ к защищенным (***protected***) членам базового класса вместе с открытыми членами ***public*** (но не к приватным членам ***private***). Такой класс может пользоваться доступом в пределах пакета только в том случае, если второй класс принадлежит тому же пакету (впрочем, пока на наследование и доступ protected можно не обращать внимания).
- Предоставить «методы доступа», то есть методы для чтения и модификации значения. С точки зрения ООП этот подход является предпочтительным, и именно он используется в технологии *JavaBeans*.

Спецификаторы доступа *Java*

При использовании ключевого слова ***public*** фактически объявляется, что следующее за ним объявление члена класса доступно для всех, и прежде всего для клиентских программистов, использующих библиотеку.

Ключевое слово ***private*** означает, что доступ к члену класса не предоставляется никому, кроме методов этого класса. Другие классы того же пакета также не могут обращаться к ***private***-членам. Все «вспомогательные» методы классов стоит объявить как ***private***, чтобы предотвратить их случайные вызовы в пакете; тем самым вы фактически запрещаете изменение поведения метода или его удаление.

```
class Sunday {
private Sunday() {}
 static Sundae makeASunday()
 { return new Sunday(); }
Sunday x = Sunday.makeASunday();
}
```

Композиция

При композиции ссылка на внедряемый объект просто включается в новый класс.

```
class Engine{
       private double vol;
       public Engine(){}
       void start(){System.out.print("Driver is ready ");}}
class Wheel{
       private double dim;
       public Wheel(){}}
class Carr{
   Wheel[] wheels;
   String model = “911”;
   int maxspeed;
   Engine engine;
   public Carr(){wheels = new Wheel[4];
   maxspeed=12;
   for(int i=0;i<4;i++){wheels[i]=new Wheel();}}
   public void start(){if (engine==null){engine=new Engine();}
      engine.start();
       System.out.print("car is ready");}}
```

Композиция

Примитивные типы, определенные в качестве полей класса, автоматически инициализируются нулевыми значениями. Однако ссылки на объекты заполняются значениями ***null***, и при попытке вызова метода по такой ссылке произойдет исключение.

Компилятор не создает объекты для ссылок «по умолчанию», и это логично, потому что во многих случаях это привело бы к лишним затратам ресурсов. Если вам понадобится проинициализировать ссылку, сделайте это самостоятельно:

- в точке определения объекта. Это значит, что объект всегда будет инициализироваться перед вызовом конструктора;
- в конструкторе данного класса;
- непосредственно перед использованием объекта. Этот способ часто называют отложенной инициализацией. Он может сэкономить вам ресурсы в ситуациях, где создавать объект каждый раз необязательно и накладно;
- с использованием инициализации экземпляров.

Наследование

Наследование является неотъемлемой частью *Java* (и любого другого языка ООП). Фактически оно всегда используется при создании класса, потому что, даже если класс не объявляется производным от другого класса, он автоматически становится производным от корневого класса *Java* ***Object***.

В программе наследование выражается перед фигурной скобкой, открывающей тело класса: сначала записывается ключевое слово ***extends***, а затем имя базового (*base*) класса. Тем самым вы автоматически получаете доступ ко всем полям и методам базового класса.

```
class ClassA{
  int i;
  ClassA(){System.out.print(“A”);}
}
class ClassB extends ClassA{
  int b;
  ClassB()[System.out.print(”B”);}
}
```

Инициализация родителя

Наследование не просто копирует интерфейс базового класса. Когда вы создаете объект производного класса, внутри него содержится подобъект базового класса. Этот подобъект выглядит точно так же, как выглядел бы созданный обычным порядком объект базового класса.

Очень важно, чтобы подобъект базового класса был правильно инициализирован, и гарантировать это можно только одним способом: выполнить инициализацию в конструкторе, вызывая при этом конструктор базового класса, у которого есть необходимые знания и привилегии для проведения инициализации базового класса. *Java* автоматически вставляет вызовы конструктора базового класса в конструктор производного класса.

Если класс не имеет конструктора по умолчанию или вам понадобится вызвать конструктор базового класса с аргументами, этот вызов придется оформить явно, с указанием ключевого слова super и передачей аргументов

```
class ClassA{
  int i;
 // ClassA(){System.out.print(“A”);}}
 // ClassA(int i){System.out.print(“A”);}}
class ClassB extends ClassA{
  int b;
  ClassB(){System.out.print(”B”);}}
```

Делегирование

В системах программирования с отсутствующим множественным наследованием задачи, требующие множественного наследования, всегда могут быть решены композицией (агрегированием) с последующим делегированием полномочий.

С помощью *делегирования* композицию можно сделать столь же мощным инструментом повторного использования, сколь и наследование. При делегировании в процесс обработки запроса вовлечено *два* объекта: получатель поручает выполнение операций другому объекту — *уполномоченному.*

Делегирование занимает промежуточное положение между наследованием и композицией: экземпляр существующего класса включается в создаваемый класс (как при композиции), но в то же время все методы встроенного объекта становятся доступными в новом классе (как при наследовании).

```
public class SpaceShipControls {
   void up(int velocity) {}
   void down(int velocity) {}
   void left(int velocity) {}
   void right(int velocity) {}
   void forward(int velocity) {}
   void back(int velocity) {}
   void turboBoost() {} }
public class SpaceShipDelegation {
  private String name;
  private SpaceShipControls controls =
    new SpaceShipControls();
  public SpaceShipDelegation(String name) {
    this.name = name;}
  // Делегированные методы:
  public void back(int velocity) {controls.back(velocity);}
  public void down(int velocity) {controls.down(velocity);}
  public void forward(int velocity) {controls.forward(velocity);}
  public void left(int velocity) {controls.left(velocity);}
  public void right(int velocity) {controls.right(velocity);}
  public void turboBoost() {controls.turboBoost();}
  public void up(int velocity) {controls.up(velocity);}
}
```

Завершение

В *Java* отсутствует понятие деструктора

В *Java* программисты просто «забывают» об объектах, не уничтожая их самостоятельно, так как функции очистки памяти возложены на сборщика мусора. Во многих случаях эта модель работает, но иногда класс выполняет некоторые операции, требующие завершающих действий.

Во многих случаях завершающие действия не являются проблемой; достаточно дать сборщику мусора выполнить свою работу. Но уж если понадобилось провести их явно, сделайте это со всей возможной тщательностью и вниманием, так как в процессе сборки мусора трудно в чем-либо быть уверенным. Сборщик мусора вообще может не вызываться, а если он начнет работать, то объекты будут уничтожаться в произвольном порядке. Лучше не полагаться на сборщик мусора в ситуациях, где дело не касается освобождения памяти. Если вы хотите провести завершающие действия, создайте для этой цели свой собственный метод и не полагайтесь на метод ***finalize()***.

Перегрузка методов базового класса

Если какой-либо из методов базового класса *Java* был перегружен несколько раз, переопределение имени этого метода в производном классе не скроет ни одну из базовых версий

Поэтому перегрузка работает вне зависимости от того, где был определен метод — на текущем уровне или в базовом классе:

```
class Base{
public void doSomething(int i){
System.out.print(i);
}
}
class Child extends Base{
public void doSomething(String s){
System.out.print(s);
}
}
….
Child c = new Child();
c.doSomething(1);
c.doSomething("sdfsdf");
```

В *Java* SE5 появилась запись ***@Override***; она не является ключевым словом, но может использоваться так, как если бы была им. Если вы собираетесь переопределить метод, используйте ***@Override***, и компилятор выдаст сообщение об ошибке, если вместо переопределения будет случайно выполнена перегрузка:

Композиция VS Наследование

И композиция, и наследование позволяют вам помещать подобъекты внутрь вашего нового класса (при композиции это происходит явно, а в наследовании — опосредованно).

Композиция в основном применяется, когда в новом классе необходимо использовать функциональность уже существующего класса, но не его интерфейс. То есть вы встраиваете объект, чтобы использовать его возможности в новом классе, а пользователь класса видит определенный вами интерфейс, но не замечает встроенных объектов. Для этого внедряемые объекты объявляются со спецификатором ***private***. Когда пользователь знает, что класс собирается из составных частей, ему значительно легче понять его интерфейс.

При использовании наследования вы берете уже существующий класс и создаете его специализированную версию. В основном это значит, что класс общего назначения адаптируется для конкретной задачи.

Композиция VS Наследование

Подмена объекта при композиции

final

По отношению к полям **final** означает, что это поле не может быть изменено. Для примитивов имеет смысл понятие константа времени компиляции. Компилятор подставляет значение константы времени компиляции во все выражения, где оно используется; таким образом предотвращаются некоторые издержки выполнения. Для их определения в *Java* используется ключевое слово ***final***. Значение такой константы присваивается во время определения.

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.

```
class Data{
int i;
String s;
final int d;
Data(){ d=1;}
}
```

```
class TestFinal{
  public final int a=5;
  public final int
            c=(new Random(10)).nextInt(10);
  public final Data val = new Data();
  TestFinal(){val.i=125;//val = new Data();
}}
```

final и инициализация

```
public class Foo {
 private static Foo instance = new Foo();
 private static final int DELTA = 6;
 private static int BASE = 7;
 private int x;
 //private static Foo instance = new Foo();
 private Foo() {
  x = BASE + DELTA;
 }
 public static void main(String... argv) {
  System.out.println(Foo.instance.x);
 }
}
```

final

Java позволяет вам объявлять неизменными аргументы метода, объявляя их с ключевым словом ***final*** в списке аргументов.

```
public void Test(final Data d){
//d = new Data();
d.i=5;}
```

Неизменные методы используются по двум причинам. Первая причина — «блокировка» метода, чтобы производные классы не могли изменить его содержание. Это делается по соображениям проектирования, когда вам точно надо знать, что поведение метода не изменится при наследовании. Любой закрытый (***private***) метод в классе косвенно является неизменным (***final***) методом. Так как вы не в силах получить доступ к закрытому методу, то не сможете и переопределить его. Ключевое слово ***final*** можно добавить к закрытому методу, но его присутствие ни на что не повлияет.

Если метод объявлен как ***private***, он не является частью интерфейса базового класса; это просто некоторый код, скрытый внутри класса, у которого оказалось то же имя. Если вы создаете в производном классе одноименный метод со спецификатором ***public***, ***protected*** или с доступом в пределах пакета, то он никак не связан с закрытым методом базового класса. Так как ***privat***-метод недоступен и фактически невидим для окружающего мира, он не влияет ни на что, кроме внутренней организации кода в классе, где он был описан.

Пример с переопределением

```
package points;
public class Point {
   public int x, y;

/* protected*/ void move(int dx, int dy) { x += dx; y += dy; }

   public void moveAlso(int dx, int dy) { move(dx, dy); }
}
```

```
package morepoints;
public class PlusPoint extends points.Point {
        public void move(int dx, int dy) {
              moveAlso(dx, dy);
        }
}
class Test {
     public static void main(String[] args) {
            PlusPoint pp = new PlusPoint();
            pp.move(1, 1);
        }
}
```

```
class WithFinals {
  // То же, что и просто private:
 private final void f() {
System.out.print("WFinals.f()");}
  // Также автоматически является final:
 private void g() { System.out.print("WithFinals.g()"); }
}
class OverridingPrivate extends WithFinals {
  private final void f() {
  System.out.print("OverridingPrivate.f()");
  }
  private void g() {
  System.out.print("OverridingPrivate.g()");
  }}
class OverridingPrivate2 extends OverridingPrivate {
  public final void f() {
  System.out.print("OverridingPrivate2.f()");
  }
  public void g() {
  System.out.print("OverridingPrivate2.g()");
  }}
```

```
public class FinalOverridingIllusion {
  public static void main(String[] args) {
    OverridingPrivate2 op2 = new OverridingPrivate2();
    op2.f();
    op2.g();
    // Можно провести восходящее преобразование:
    OverridingPrivate op = op2;
    // Но методы при этом вызвать невозможно:
    //! op.f();
    //! op.g();
    // И то же самое здесь:
    WithFinals wf = op2;
    //! wf.f();
    //! wf.g();
  }
}
```

immutable

Литература

1. Брюс Эккель Философия Java. 4-е издание
2. Хорстманн К. С., Корнелл Г. -- Java 2. Том 1. Основы
3. Habrahabr.ru
4. Sql.ru
5. http://grepcode.com/project/repository.grepcode.com/java/root/jdk/openjdk/
6. !!! Google.com