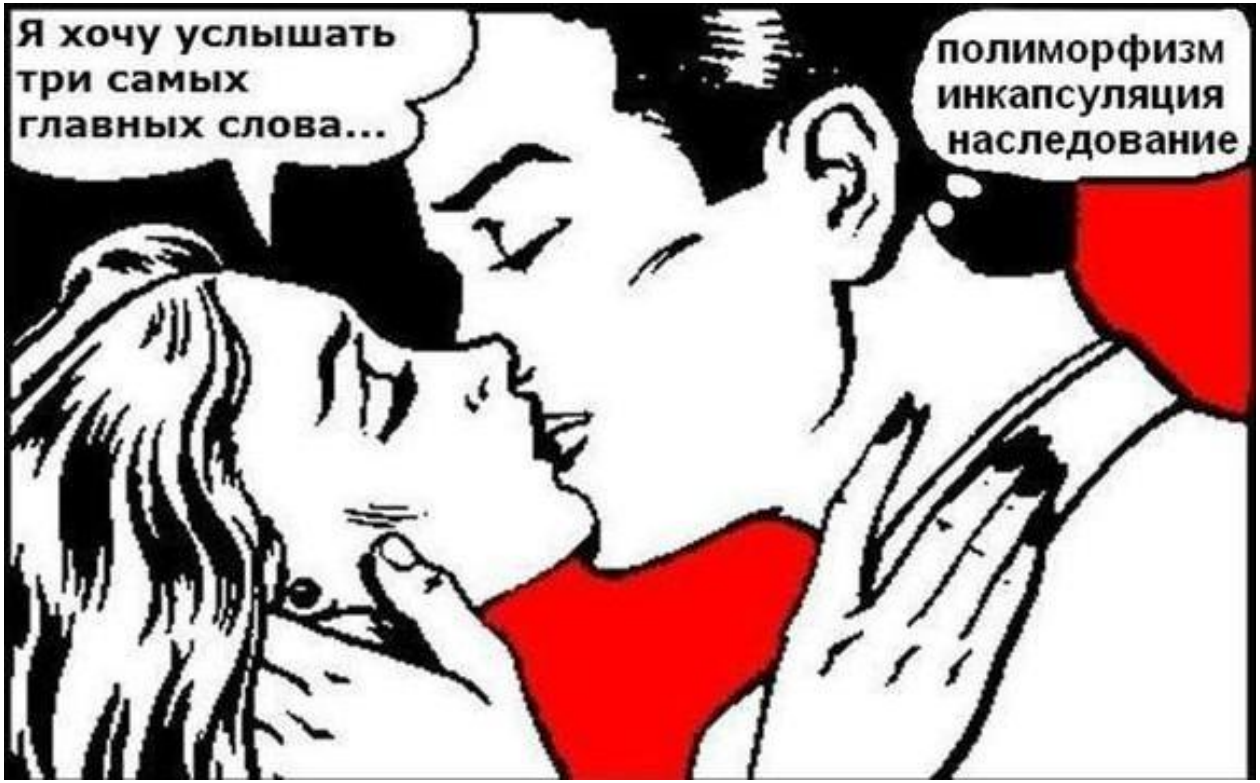
Core Java, осень

ООП


Я хочу услышать три самых главных слова...
полиморфизм
инкапсуляция
наследование

Все есть класс

- Любой код - часть класса
- Любые данные - часть некоторого класса
- Любые данные (кроме примитивов) - объекты

Class

```
public class Person {

 private String name;
 private int age;

}
```

Class

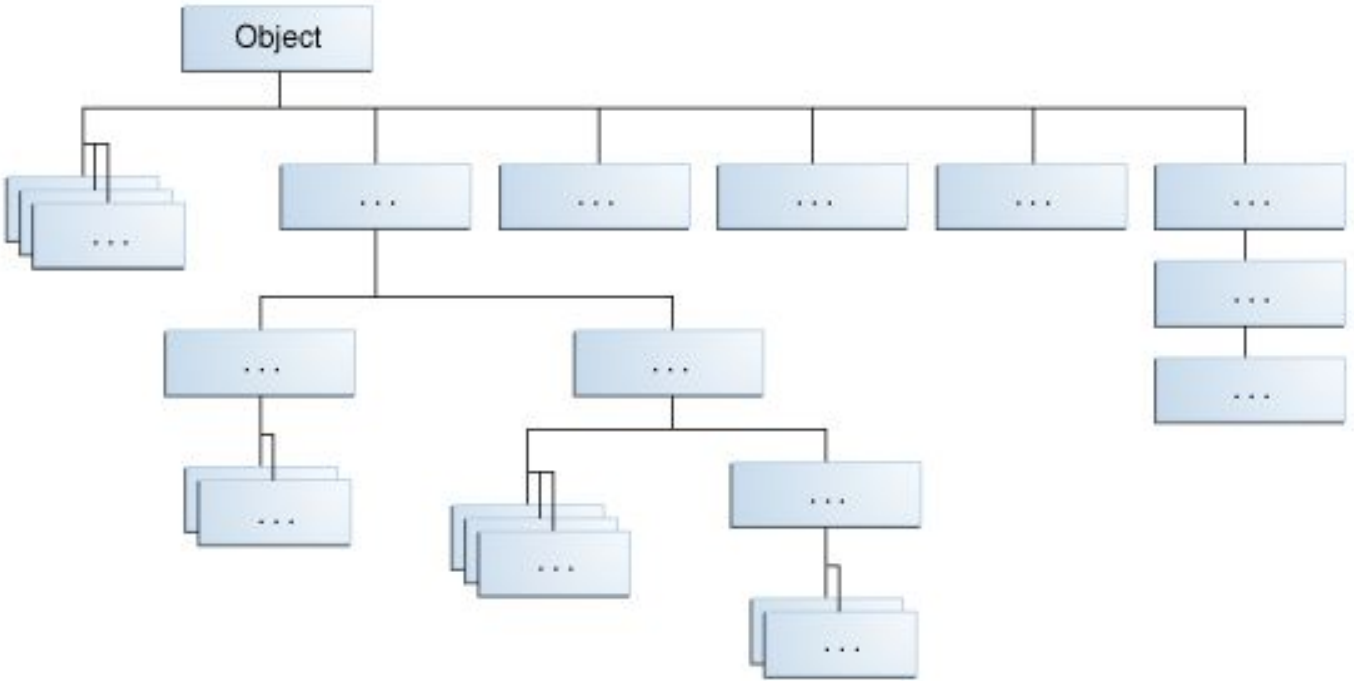
```
public class Person {

 private String name;
 private int age;

 // конструктор
 public Person(String name, int age) {
   this.name = name;
   this.age = age;
 }
}
```

Методы

```
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;
}
```

```java
public class Person {

 private String name;
 private int age;

 public Person(String name, int age) {
   this.name = name;
   this.age = age;
 }

 public void sayHi() {
   System.out.println("Hi, " + name);
 }

 public void birthday() {
   makeOlder();
   System.out.println("Oh, I've gotten older! I'm " + age + " years.");
 }

 private void makeOlder() {
   age += 1;
 }
}
```

В java все - Object



Static, final, strictfp

```
public class Employee extends Person {

 public static long generalId = 0;
 private long id;
 private String company;
 private int salary;

 // smth code

 public void setId() {
   id = ++generalId;
 }
}
```

Static, final, strictfp

```
public class Employee extends Person {

 public static long generalId;
 private long id;
 private String company;
 private int salary;

 static {
   generalId = 1;
 }

 // smth code

 public void setId() {
   id = generalId++;
 }
}
```

Static, final, strictfp

```
public final class Person{
}

public class Employee extends Person{
 // ошибка
}
```

- Также final методы нельзя переопределять

Вложенные и внутренние классы

```
class OuterClass {
 //...
 static class StaticNestedClass {
 //...
 }
 class InnerClass {
 //...
 }
}
```

Фабричная инициализация и builder

Статические методы вместо конструктора

```
public static Boolean valueOf(boolean b) {
   return (b ? TRUE : FALSE);
}
```

```
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance();
```

Фабричная инициализация и builder

Много параметров в конструкторе

Можно заменить на шаблон JavaBeans, но теряется читаемость и потокобезопасность, также нельзя сделать неизменяемый объект

Можно заменить шаблоном Builder

Builder

```
// “плохой” вариант
public class NutritionFacts {

 private final int servingSize = -1; // необходимо
 private final int servings = -1; // необходимо
 private final int calories; // опционально
 private final int fat; // опционально
 private final int sodium; // опционально
 private final int carbohydrate; // опционально

 // 100500 вариантов конструкторов...
}
```

Builder

```
// тоже “плохой” вариант
public class NutritionFacts {

 private int servingSize = -1;
 private int servings = -1;
 private int calories;
 private int fat;
 private int sodium;
 private int carbohydrate;

 // getters, setters...
}
```

Builder

```
// просто пример использования
NutritionFacts cola = new  NutritionFacts.Builder(240, 8)
    .calories(100)
    .fat(8)
    .sodium(14)
    .carbohydrate(30)
    .build();
```

Инкапсуляция

Модификаторы доступа

- private - доступно только классу
- package-private (no modifier) - доступно только пакету
- protected - доступно классу, пакету и классам наследникам
- public - доступно всем

Инкапсуляция

```
public class Person {

 private String name;
 private int age;
 private String city;

 public Person() {
 }

 public Person(String name, int age, String city) {
   this.name = name;
   this.age = age;
   this.city = city;
 }
}
// геттеры и сеттеры для всего кроме city
```

Наследование

Наследование

```
public class Employee extends Person{
 private String company;
 private int salary;

 public Employee() {
 }

 public Employee(String name, int age, String city, String company, int salary) {
   super(name, age, city);
   this.company = company;
   this.salary = salary;
 }
}
```

Перегрузка и переопределение

```
// overload
public void calculate(){
 // smth
}

public void calculate(int i){
 // smth
}

public void calculate(int i, double d){
 // smth
}
```

Перегрузка и переопределение

```
public class Person {

 // smth
 public void sayHi() {
   System.out.println("Hi, " + name);
 }
}

public class Employee extends Person {

 // smth
 @Override
 public void sayHi() {
   System.out.println("Hi, " + /*super.*/getName() + ". I'm employee");
 }
}
```

Оператор instanceof

```
Object string = "this is string!";

// было давно
if (string instanceof String) {
 String realString = (String) string;
 System.out.println(realString);
}

// было недавно
if (string instanceof String) {
 var realString = (String) string;
 System.out.println(realString);
}

// стало сейчас
if (string instanceof String realString) {
 System.out.println(realString);
}
```

Неизменяемые классы

Чтобы класс был неизменяемым, он должен соответствовать следующим требованиям:

- Должен быть объявлен как final, чтобы от него нельзя было наследоваться. Иначе дочерние классы могут нарушить иммутабельность.
- Все поля класса должны быть приватными и неизменяемыми.
- Для корректного создания экземпляра в нем должны быть параметризованные конструкторы, через которые осуществляется первоначальная инициализация полей класса.
- Для исключения возможности изменения состояния после инстанцирования, в классе не должно быть сеттеров.
- Для полей-коллекций необходимо делать глубокие копии, чтобы гарантировать их неизменность.

Неизменяемые классы

Неизменяемые классы:

- легко конструировать, тестировать и использовать
- автоматически потокобезопасны и не имеют проблем синхронизации
- не требуют конструктора копирования
- позволяют выполнить «ленивую инициализацию» хэшкода и кэшировать возвращаемое значение
- не требуют защищенного копирования, когда используются как поле
- делают хорошие `Map` ключи и `Set` элементы (эти объекты не должны менять состояние, когда находятся в коллекции)
- делают свой класс постоянным, единожды создав его, а он не нуждается в повторной проверке
- всегда имеют «атомарность по отношению к сбою» (failure atomicity, термин применил Джошуа Блох): если неизменяемый объект бросает исключение, он никогда не останется в нежелательном или неопределенном состоянии.

Singleton

```
public class Singleton {

 // не потокобезопасно
 private static Singleton instance;

 private Singleton() {
 }

 public static Singleton getInstance() {
   if (instance == null) {
     instance = new Singleton();
   }
   return instance;
 }
}
```

Record

```
public record StudentRecord(String name, CourseType courseType) {}
```

У класса record есть ограничения и особенности:

- все объявленные поля получают модификатор final;
- все поля класса объявляются в заголовке, дополнительные объявить нельзя
- можно объявлять static-поля класса;
- класс record неявно объявлен как final, поэтому его нельзя наследовать;
- он не может быть абстрактным и наследовать другие классы;
- можно добавлять свои конструкторы;
- для конструктора можно использовать проверку аргументов;
- можно переопределить стандартные методы — геттеры, toString(), equals()и hashCode();
- можно добавлять статические и нестатические методы.

Record

```
public static List<Student> findTreeStudentsByAlphabet(List<Student> students) {
 record CourseTypeToFirstLetter(Student student, char firstLetter) {}

 return students.stream()
     .map(st -> new CourseTypeToFirstLetter(st, st.getName().charAt(0)))
     .sorted((Comparator.comparing(CourseTypeToFirstLetter::firstLetter)))
     .map(CourseTypeToFirstLetter::student)
     .limit(3)
     .collect(Collectors.toList());
}
```

Sealed, non-sealed

```
sealed class Person permits Student, Teacher, Curator {}
```

Можем ограничивать наследование, но не так строго, как final

Композиция и агрегирование

Полиморфизм

- Статический (раннее связывание) = перегрузка методов
- Динамический (позднее связывание) = переопределение методов

Полиморфизм

```
// статический
public class Person {

 // smth code

 public void sayHi() {
   System.out.println("Hi, " + name);
 }

 public void sayHi(String name) {
   System.out.println("Hi, " + name);
 }
}
```

```
Person person = new Person("Alex", 21,
"Moscow");
person.sayHi();
```

Полиморфизм

```
// динамический
Person person = new Person("Alex", 21, "Moscow");
Person employee = new Employee("Mike", 24, "London", "Oracle", 10_000);
Person lead = new Lead("Jeremy", 38, "NY", "Google", 100_000, "main team");

List<Person> stuff = Arrays.asList(person, employee, lead);
for (Person p: stuff) {
 p.sayHi();
}
```

Upcast и downcast

```
public static class Animal {

 public void doSound() {
   System.out.println("I'm an animal! I say arrrrgh");
 }
}

public static class Cat extends Animal {
 @Override
 public void doSound() {
   System.out.println("I'm a cat! Meow! Meow!");
 }
}

public static void main(String[] args) {
 Animal animal = new Cat();
 Cat cat = (Cat) animal;
 Animal newAnimal = (Animal) cat;
}
```

Enum

```
public enum DayOfWeek {
 MONDAY, TUESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Object

```
public String toString()
public native int hashCode()
public boolean equals(Object obj)
```

Object

```
public String toString()
public native int hashCode()
public boolean equals(Object obj)
public final native Class getClass()
public final native void notify()
public final native void notifyAll()
public final native void wait(long timeout)
public final void wait(long timeout, int nanos)
public final void wait()
protected void finalize()
protected native Object clone()
```

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`.

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 изменились.
- 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.

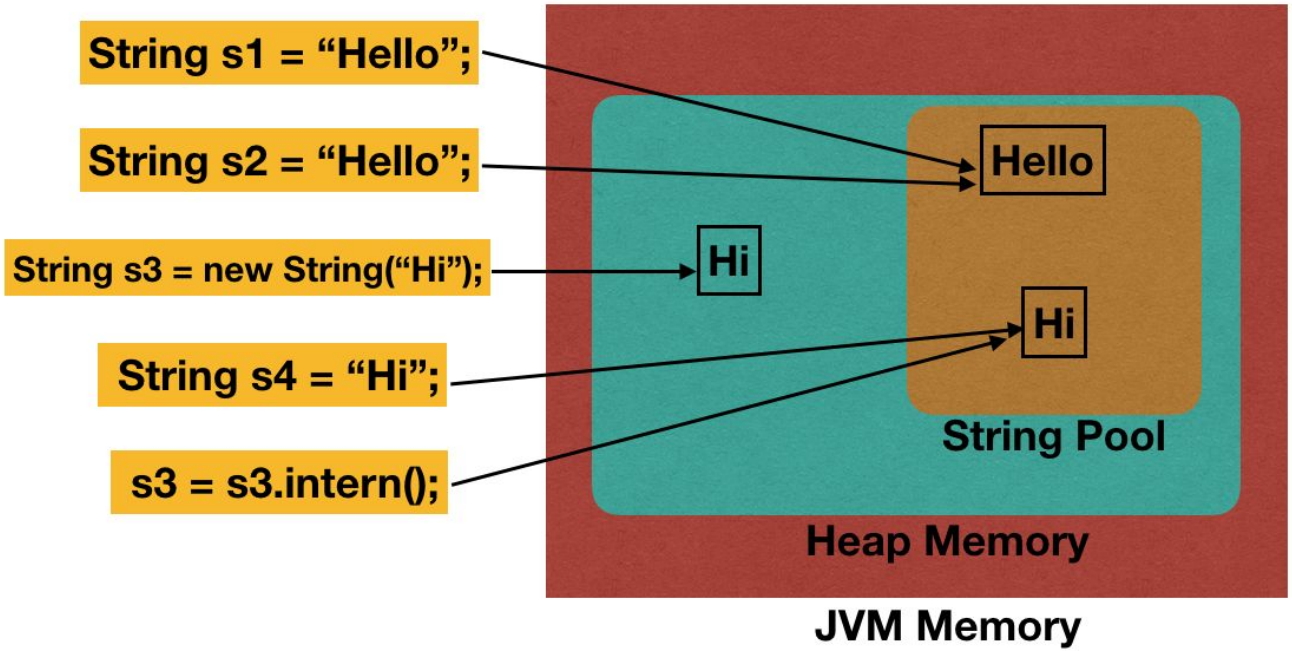
String

```
String s = "I love movies";

String day = "День";
String and = "и";
String night = "Ночь";
String dayAndNight = day + " " + and + " " + night;
```

String

```
String a = String.valueOf(1);
String b = String.valueOf(12.0D);
String c = String.valueOf(123.4F);
String d = String.valueOf(123456L);
String s = String.valueOf(true);
String human = String.valueOf(new Human("Alex"));

System.out.println(a); // 1
System.out.println(b); // 12.0
System.out.println(c); // 123.4
System.out.println(d); // 123456
System.out.println(s); // true
System.out.println(human); // Человек с именем Alex
```

String pool


String s1 = “Hello”;
String s2 = “Hello”;
String s3 = new String(“Hi”);
String s4 = “Hi”;
s3 = s3.intern();
Hello
Hi
Hi
String Pool
Heap Memory
JVM Memory

Обертки над примитивами

```
Integer i = new Integer(682);
Double d = new Double(2.33); Boolean b = new Boolean(false);

//
String s = "1166628";
Integer i = Integer.parseInt(s);

//
int x = 7;
Integer y = 111; x = y; // автораспаковка
y = x * 123; // автоупаковка
```