HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

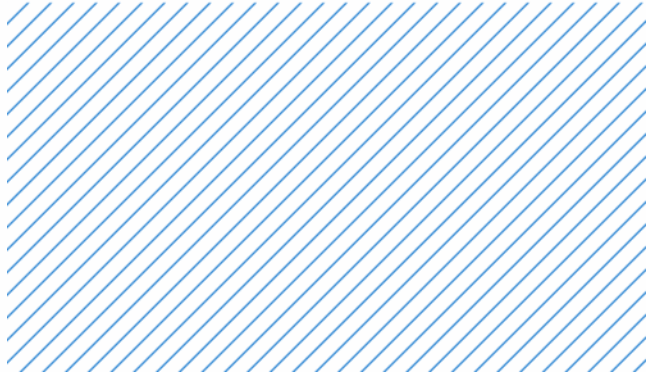
Лекция 8

Наследование и абстракции

Программирование на языке Java

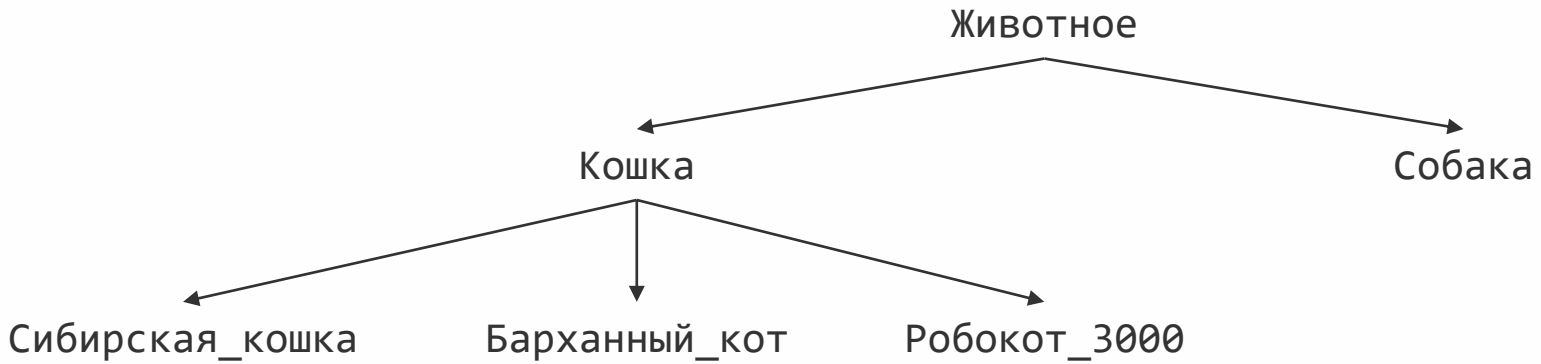
Роман Гуров

ВШЭ БИ 2021



Наследование классов

Наследование (англ. inheritance) — концепция ООП, согласно которой класс может наследовать данные и функциональность некоторого другого класса, способствуя повторному использованию общего кода



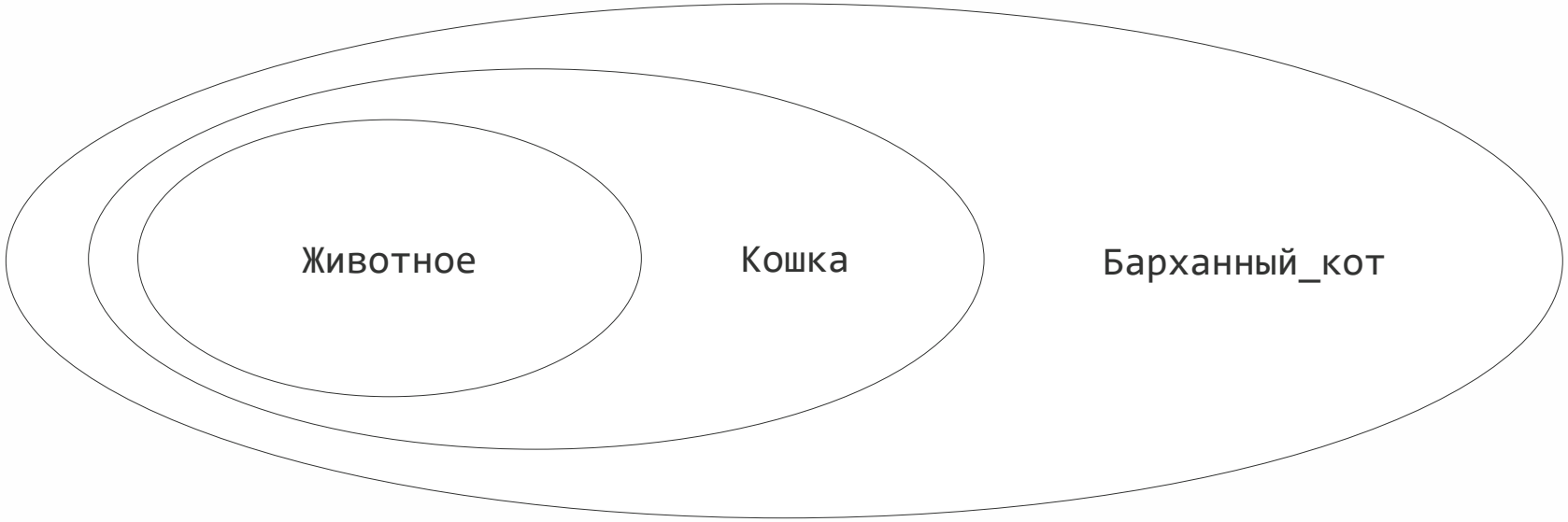
- Суперкласс (англ. super class), родительский класс (англ. parent class), предок, родитель или надкласс — класс, от которого наследуется некоторый другой класс
- Подкласс (англ. subclass), производный класс (англ. derived class), дочерний класс (англ. child class), класс потомок, класс наследник — класс, наследуемый от некоторого суперкласса
- Базовый класс (англ. base class) — это класс, находящийся на вершине иерархии наследования классов

Кошка – суперкласс для Сибирская_кошка

Животное – базовый класс в данной иерархии

Наследование классов

Класс-наследник сохраняет все методы своего родителя



Можно представлять, что подкласс полностью содержит в себе содержимое (методы и поля) своего родителя

При этом, подкласс может переопределить метод суперкласса, изменяя, таким образом, его поведение

Наследование в Java

Для наследования в Java используется ключевое слово **extends**

```
class Base {
    public void based_method() {
        System.out.println("I'm based!");
    }
}


class Derived extends Base {
    public void derived_method() {
        System.out.println("I'm derived!");
    }
}
```

Наследник будет содержать в себе все поля и методы родительского класса

```
public static void main(String[] args) {
    Base obj1 = new Base();
    Derived obj2 = new Derived();
    obj1.based_method();
    obj2.based_method();
    obj2.derived_method();
}
```

Модификаторы доступа при наследовании

Модификаторы доступа при наследовании работают по знакомым правилам

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Приватные поля и методы родителя недоступны из класса-наследника (по сути наследник – другой класс)

`protected` – недоступен снаружи (не считая package), но доступен внутри всех наследников

```java
class Base {
    private int a;
    protected int b;
    public int c;
}


public class Derived extends Base {
    public int test() {
        return a + b + c;
    }
}
```

```java
package ru.hse.lecture8.subpackage;

import ru.hse.lecture8.Derived;

public class Main {
    public static void main(String[] args) {
        Derived obj = new Derived();
        int temp;
        temp = obj.a;
        temp = obj.b;
        temp = obj.c;
    }
}
```

Переопределение методов

```java
class Base {
    public void present() {
        System.out.println("It's me, Base!");
    }
}


class Derived extends Base {
    @Override
    public void present() {
        System.out.println("It's me, Derived!");
    }
}
```

```java
public static void main(String[] args) {
    Base base = new Base();
    Derived derived = new Derived();
    base.present();
    derived.present();
}
```

```
It's me, Base!
It's me, Derived!
```

Для переопределения метода необходимо, чтобы метод наследника удовлетворял требованиям:

- Идентичные название и набор аргументов
- Тот же или более открытый модификатор доступа
- Видимость базового метода в классе-наследнике
- Возвращаемый тип совпадает с или является наследником возвращаемого типа базового метода

Аннотация @Override проверяет, что метод действительно переопределен

Её использование необязательно, но улучшает читаемость кода и позволяет избежать опечаток в сигнатуре переопределяемого метода

Конструктор базового класса

Создание объекта класса-наследника обязательно требует вызова конструктора базового класса

По-умолчанию, вызывается *конструктор по-умолчанию* (то есть, с пустыми скобками):

```
class Base {
    Base() {
        System.out.println("Base default constructor");
    }
}


class Derived extends Base {
    Derived() {
        System.out.println("Derived default constructor");
    }
}
```

```
public static void main(String[] args) {
    new Derived();
}
```

```
Base default constructor
Derived default constructor
```

Если требуется вызвать конструктор с параметрами, то нужно использовать ключевое слово **super**:

```
class Base {
    Base(int value) {
        System.out.println("Base constructor with value=" + value);
    }
}


class Derived extends Base {
    Derived() {
        super(10);
        System.out.println("Derived default constructor");
    }
}
```

```
public static void main(String[] args) {
    new Derived();
}
```

```
Base constructor with value=10
Derived default constructor
```

Ещё о super

Ключевое слово `super` может также использоваться для вызова базового метода из переопределённого:

```
class Base {
    private static final int value = 5;

    String tell() {
        return "I'm base with value " + value;
    }
}

class Derived extends Base {
    @Override
    String tell() {
        return "I'm derived and my superclass tells that \"" + super.tell() + '"';
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println( (new Derived()).tell() );
    }
}
```

```
I'm derived and my superclass tells that "I'm base with value 5"
```

Вызов `tell()` без `super` привел бы к бесконечной рекурсии

Класс `Object`

Если при объявлении класса не указать его надкласс, то он будет наследован от стандартного класса `Object`

Таким образом, транзитивно, ***все*** классы в Java являются наследниками `Object`

А значит, каждый класс имеет некоторый унаследованный набор основных методов:

```
public class Object {
    public String toString() {
        return this.getClass().getName() + "@" + Integer.toHexString(this.hashCode());
    }

    public boolean equals(Object obj) {
        return this == obj;
    }

    public native int hashCode();

    // ...
}
```

Переопределение методов класса Object

Попробуем создать класс `ComplexNumber` и переопределить для него методы базового класса `Object`

```java
import java.util.Objects;

class ComplexNumber {
    public ComplexNumber(double real, double imaginary) {
        real_ = real;
        imaginary_ = imaginary;
    }

    @Override
    public String toString() {
        return real_ + " + i * " + imaginary_;
    }

    @Override
    public boolean equals(Object other) {
        if (this == other) return true;
        if (!(other instanceof ComplexNumber)) return false;

        ComplexNumber comp = (ComplexNumber) other;
        return comp.real_ == real_ && comp.imaginary_ == imaginary_;
    }

    @Override
    public int hashCode() {
        return Objects.hash(real_, imaginary_);
    }

    private final double real_;
    private final double imaginary_;
}
```

```java
public static void main(String[] args) {
    ComplexNumber comp1 = new ComplexNumber(-10.5, 20.4);
    ComplexNumber comp2 = new ComplexNumber(-10.5, 20.4);
    System.out.println(comp1);

    System.out.println(comp1 == comp2);
    System.out.println(comp1.equals(comp2));
    System.out.println(comp1.hashCode() + " " + comp2.hashCode());
}
```

```
-10.5 + i * 20.4
false
true
1791820737 1791820737
```

Полиморфизм в Java

Полиморфизм – ещё один основополагающий принцип ООП – в общем смысле, означает возможность обращаться с разными объектами одинаковым образом

Примером *статического* полиморфизма является уже знакомое нам понятие – перегрузка методов

Статический полиморфизм происходит на этапе компиляции

Но куда более интересным является *динамический* полиморфизм, работающий на этапе выполнения:

```
class Base {
    public String tell() {
        return "I'm base";
    }
}

class Derived extends Base {
    @Override
    public String tell() {
        return "I'm derived";
    }
}
```

```
public static void main(String[] args) {
    Base base = new Derived();
    System.out.println(base.tell());
}
```

Можно преобразовать объект к своему родителю и использовать его так, будто бы это и есть родитель

Полиморфизм в Java

```java
class Base {
    public String tell() {
        return "I'm base";
    }
}

class Derived extends Base {
    @Override
    public String tell() {
        return "I'm derived";
    }
}
```

```java
public static void main(String[] args) {
    Base base = new Derived();
    System.out.println(base.tell());
}
```

```
I'm derived
```

Можно преобразовать объект к своему родителю и использовать его так, будто бы это и есть родитель

Но объект при этом всё так же остаётся объектом класса-наследника

Если в наследнике переопределён какой-то метод, то даже после преобразования к родителю будет вызываться его переопределённая версия

Для запрета дальнейшего переопределения метода в наследниках, можно обозначить метод как `final`

Полиморфизм в Java

В этом и заключается полиморфизм – мы обрабатываем объекты разных типов так, будто бы тип один:

```
class Base {
    public String tell() {
        return "I'm base!";
    }
}

class Foo extends Base {
    public String tell() {
        return "I'm foo!";
    }
}
class Bar extends Base {
    public String tell() {
        return "I'm bar!";
    }
}
class Baz extends Base {
    public String tell() {
        return "I'm baz!";
    }
}
```

```
public class Main {
    public static Base[] generateArray(int n) {
        Base[] result = new Base[n];
        for (int i = 0; i < n; ++i) {
            if (Math.random() < 0.33) {
                result[i] = new Foo();
            } else if (Math.random() < 0.5) {
                result[i] = new Bar();
            } else {
                result[i] = new Baz();
            }
        }
        return result;
    }

    public static void main(String[] args) {
        int n = (new Scanner(System.in)).nextInt();

        Base[] bases = generateArray(n);

        for (int i = 0; i < bases.length; ++i) {
            System.out.println(bases[i].tell());
        }
    }
}
```

```
6
I'm bar!
I'm baz!
I'm bar!
I'm baz!
I'm foo!
I'm baz!
```

Абстрактные классы

Рассмотрим цепочку наследования из первого слайда: `Животное->Кошка->Робокот_3000`

Что из себя должны представлять объекты классов `Животное` и `Кошка`?

На самом деле, ничего! Они являются лишь абстрактными понятиями, которые может и имеют некоторую функциональность и данные, но не могут быть использованы без конкретной реализации в наследнике

Поэтому нужен способ запретить инстанциировать такие абстрактные классы, с чем нам помогает ключевое слово `abstract`

```
abstract class AbstractNamePrinter {
    public void printName() {
        System.out.println(name);
    }
    protected String name;
}
```

```
new AbstractNamePrinter();
```

'AbstractNamePrinter' is abstract; cannot be instantiated

Make 'AbstractNamePrinter' not abstract Alt+Shift+Enter

Абстрактные классы

```java
abstract class AbstractNamePrinter {
    public void printName() {
        System.out.println(name);
    }
    protected String name;
}
```

```java
class NamePrinter extends AbstractNamePrinter {
    public NamePrinter(String name) {
        this.name = name;
    }
}
```

```java
class PrettyNamePrinter extends AbstractNamePrinter {
    public PrettyNamePrinter(String name) {
        this.name = name;
    }
    @Override
    public void printName() {
        System.out.println("[{(" + name + ")}]");
    }
}
```

Может быть и так, что реализация некоторого метода в абстрактном классе бессмысленна,
но хочется потребовать, чтобы каждый наследник предоставлял свою реализацию данного метода

Такой метод помечается ключевым словом `abstract` и не имеет тела

```java
abstract class AbstractCat {
    public abstract String meow();
}
```

```java
class MaineCoon extends AbstractCat {
    @Override
    public String meow() {
        return "MEOW";
    }
}
```

```java
class RoboCat extends AbstractCat {
    @Override
    public String meow() {
        return "M3-0w!!1";
    }
}
```

Пример иерархии

Рассмотрим пример иерархии классов, реализующей геометрические фигуры

```
public class Point {
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() { return x; }

    public double getY() { return y; }

    @Override
    public String toString() { return "(" + x + ", " + y + ")"; }

    private final double x;
    private final double y;
}
```

Пример иерархии

```
public enum Color {
    BLACK,
    WHITE,
    RED,
    GREEN,
    BLUE
}
```

```
public abstract class Shape {
    public Shape(Color color) { this.color = color; }

    public Color getColor() { return color; }

    public abstract double getArea();

    private final Color color;
}
```

```
public class Triangle extends Shape {
    private final Point a;
    private final Point b;
    private final Point c;

    public Triangle(Point a, Point b, Point c, Color color) {
        super(color);
        this.a = a;
        this.b = b;
        this.c = c;
    }

    @Override
    public double getArea() {
        return Math.abs((a.getX() - c.getX()) * (b.getY() - c.getY())
                - (b.getX() - c.getX()) * (a.getY() - c.getY())) / 2;
    }
}
```

```
public class Circle extends Shape {
    private final Point center;
    private final double radius;

    public Circle(Point center, double radius, Color color) {
        super(color);
        this.center = center;
        this.radius = radius;
    }

    public Point getCenter() { return center; }

    public double getRadius() { return radius; }

    @Override
    public double getArea() { return radius * radius * Math.PI; }
}
```

```
public class Square extends Shape {
    private final Point center;
    private final double size;

    public Square(Point corner, double size, Color color) {
        super(color);
        this.center = corner;
        this.size = size;
    }

    public Point getCenter() { return center;}

    public double getSize() { return size; }

    @Override
    public double getArea() { return size * size; }
}
```

Пример иерархии

```
public static void main(String[] args) {
    Circle circle = new Circle(
            new Point(0, 0), 1, Color.BLACK);

    Triangle triangle = new Triangle(
            new Point(0, 0), new Point(1, 0), new Point(0, 1), Color.RED);

    Square square = new Square(
            new Point(5, 5), 2, Color.BLUE);

    Shape shape = triangle;
    Object object = triangle;
    triangle = (Triangle) object;

    Shape[] shapes = {circle, triangle, square};

    Shape maxShape = findShapeWithMaxArea(shapes);
    System.out.println("Shape with max area: " + maxShape);
}

private static Shape findShapeWithMaxArea(Shape[] shapes) {
    Shape maxShape = null;
    double maxArea = Double.NEGATIVE_INFINITY;
    for (Shape shape: shapes) {
        double area = shape.getArea();
        if (area > maxArea) {
            maxArea = area;
            maxShape = shape;
        }
    }
    return maxShape;
}
```

Интерфейсы

Помимо абстрактных классов, существует понятие интерфейса

У интерфейса все методы всегда являются абстрактными и публичными

Объявляется интерфейс ключевым словом `interface` вместо `class`

```
public interface Runnable {
    void run();
}
```

Поля могут быть только `public static final`, то есть константы

Также, могут быть и static методы

Цель интерфейса – определить некий контракт – набор методов, которые обязуется иметь класс, реализующий этот интерфейс

Интерфейсы

Для указания того, что класс реализует интерфейс, используется ключевое слово `implements`

```java
public interface Runnable {
    void run();
}
```

```java
class Program implements Runnable {
    @Override
    public void run() {
        System.out.println("The program is running!");
    }
}
```

Несмотря на все запреты, интерфейс может предоставлять реализацию метода по-умолчанию:

```java
interface StupidRunnable {
    void run();

    default void runWithLog() {
        System.out.println("Started run");
        run();
        System.out.println("Finished run");
    }
}
```

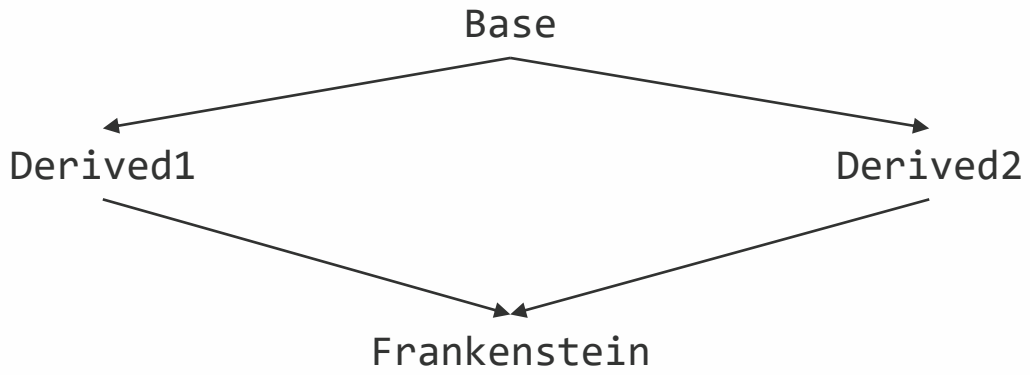
Интерфейсы и множественное наследование

В отличие от суперклассов, ограничений на количество реализуемых интерфейсов нет

```
public final class StringBuilder
        extends AbstractStringBuilder
        implements Serializable, Comparable<StringBuilder>, CharSequence {
    // ...
}
```

Эта возможность позволяет спокойно жить без возможности наследоваться от нескольких классов

Отсутствие множественного наследования спасает нас от проблемы ромбовидного наследования



Примеры интерфейсов

CharSequence – последовательность символов

```
public interface CharSequence {
    int length();

    char charAt(int var1);

    default boolean isEmpty() {
        return this.length() == 0;
    }

    CharSequence subSequence(int var1, int var2);
}
```

CharSequence реализован такими классами, как String и StringBuilder

```
public static void printEverySecondChar(CharSequence seq) {
    for (int i = 1; i < seq.length(); i += 2) {
        System.out.println(seq.charAt(i));
    }
}
```

Примеры интерфейсов

Appendable – нечто, к чему в конец можно дописывать символы

```
public interface Appendable {
    Appendable append(CharSequence var1);

    Appendable append(CharSequence var1, int var2, int var3);

    Appendable append(char var1);
}
```

Appendable реализован тем же StringBuilder, да и многими другими классами