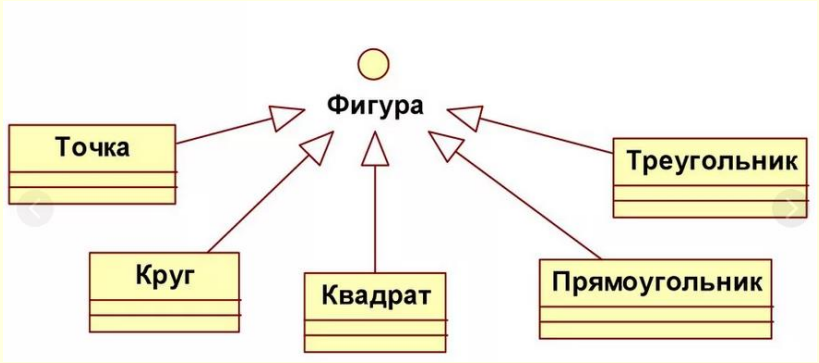
Классы: наследование и интерфейсы



Пакеты

- Аналогия с библиотеками
  - Средство объединения классов
  - Механизм распространения группы классов, связанных с решением одной задачи
- Стандартная библиотека java – набор пакетов
- Иерархическая вложенность пакетов
- Пакеты обеспечивают уникальность имени класса
- Чтобы обеспечить абсолютную уникальность имени пакета, компания Sun рекомендует использовать доменное имя вашей компании в Интернет (которое по определению уникально), записанное в обратном порядке.
- Единственная цель вложенных пакетов— гарантия уникальности имен. С точки зрения компилятора между вложенными пакетами нет абсолютно никакой связи.

Пакеты

- Класс может использовать все классы из собственного пакета и все *открытые* классы из других пакетов.

```
java.util.Date today = new java.util.Date( );
```

Или

```
import java.util. * ;   / / import  java.util.Date;
```

```
Date today = new Date();
```

Пакеты

```
package com.horstmann.corejava;

public class Employee{
. . .
}
```

- Каталог пакета

com\horstmann\corejava

- Пакет «по умолчанию» - текущий каталог

Поиск пакета

- Каталог пакета находится внутри каталога, где лежит компилируемый файл

(рабочий каталог)

PackageTest.Java

PackageTest.class

com/

horstmann/

corejava/

Employee.Java

Employee.class

Поиск пакета

- Используя переменную classpath

classpath = .; c:\classes;c:\com\horstmann\corejava.

Архивы

```
rt.jar
classpath =c:\class;.;
    c:\archives\archive.jar
```

У javac и java есть ключ «-classpath»

Область видимости пакета

- Если ни один модификатор доступа не указан, то сущность (т.е. класс, метод или переменная) является доступной всем методам в том же самом *пакете*

Классы (краткое повторение)

- Все типы данных, кроме примитивных реализуются в виде классов
- Объекты классов создаются динамически (ссылочная модель)
- Уничтожать объекты не требуется

Доступ

- Классы, поля и методы имеют модификатор доступа (public, private, protected, в пределах пакета)
- Конструктор гарантирует инициализацию объекта
- Завершающие действия следует прописывать в специальном методе и вызывать явно, т.к. деструктор отсутствует, а метод finalize() служит другим целям

Методы

- Допустима перегрузка методов (требуется разная сигнатура)
- Для обращения к объекту, вызвавшему метод – this
- Для вызова конструктора из другого конструктора – this()

Повторное использование классов

- Композиция
- Наследование

Наследование

- class Child extends Parent
- любой класс – наследник Object
- допустимо только одиночное наследование

```
class Base {
 .....
   .....
}
class Derived extends Base {
   .....
   .....
}
```

Наследование

- в конструкторе производного класса автоматически вызывается конструктор без параметров базового класса
- для вызова другого конструктора используется super()
- вызов конструктора базового класса должен быть первой командой в конструкторе производного класса
- в классе-наследнике допустимы и переопределение и перегрузка методов базового класса
- для обращения к методам базового класса используется super
- для всех методов, кроме final, используется позднее связывание

Полиморфизм

```
class Employee {
   public Employee(String n, double s, int year)   {. . .}
. . .
}
class Manager extends Employee {
public Manager(String n, double s, int year)
   {
      super(n, s, year);
      bonus = 0;
   }
. . .
}
```

```
public static void main(String[] args)    {
     Manager boss = new Manager("Carl Cracker", 80000, 1987);
      boss.setBonus(5000);
      Employee[] staff = new Employee[3];
      staff[0] = boss;
      staff[1] = new Employee("Harry Hacker", 50000, 1989);
      staff[2] = new Employee("Tommy Tester", 40000, 1990);
      for (Employee e : staff)
         System.out.println("name=" + e.getName() +
                                  ",salary=" + e.getSalary());
   }
```

Абстрактные классы

- Абстрактным называется класс, у которого хотя бы один метод не имеет реализации
- Метод, не имеющий реализации, называется абстрактным методом

```
abstract class Instrument  {
        . . .
                public abstract void sound() ;
}
```

- Можно описывать переменные объекты абстрактного класса
- Нельзя создавать экземпляры объектов абстрактного класса
- Наследник абстрактного класса должен переопределить все его абстрактные методы или он тоже будет абстрактным классом

Интерфейсы

- Интерфейс описывает протокол доступа к классу
- В интерфейсе определяются только заголовки методов класса
- Если в интерфейсе определены поля, то они автоматически являются статическими. Для инициализации используется статическая секция
- Классы могут реализовывать один или несколько интерфейсов
- Если класс реализует интерфейс он должен определить реализацию всех методов этого интерфейса
- Если хотя бы один из методов остается неопределенным, класс должен быть описан как абстрактный
- Начиная с Java 9 введены методы статические (static) и по умолчанию (default)

Интерфейсы

```
interface StackInt {
    void push (int  value);
     bool isEmpty();
     int  pop();
     int  peek();
}
```

Интерфейсы

```
class Array100StackInt implements StackInt {
   private int array [ ];
   private int top;
   public Array100StackInt () {
            array = new int[100]; top = 0;
   }
    public void push (int  value) { array [top++]=value;}
    public bool isEmpty() { return (top ==0);}
    public int  pop() { top--; return array [top];}
    public int  peek() { return array[top-1];}
    public bool isFull() {return top==100;} //расширение
}
```

Интерфейсы

```
Array100StackInt  s1 = new  Array100StackInt();

StackInt   s2 = new   Array100StackInt();
```

для s2 нельзя использовать вызов

s2.isFull()

Внутренние классы

- **Inner classes** — внутренние классы (они же — non static nested classes, нестатические вложенные классы)
- Это классы для выделения в программе некой сущности, которая неразрывно связана с другой сущностью

```
public class OuterClass {
...
   private InnerClass obj;

  class InnerClass {
     ... }
}
```

Внутренние классы

Объект внутреннего класса не может существовать без объекта «внешнего» класса

У объекта внутреннего класса есть доступ к полям и методам «внешнего» класса

При создании объекта внутреннего класса, в него незаметно передается ссылка на объект «внешнего» класса (this)

Локальные классы

```
public class OuterClass {
  public void someMethod(){
        class LocalClass{ . . .
                    }
   . . .
  }
}
```

Доступен только в методе, где он определен

У объекта локального класса есть доступ к полям и методам «внешнего» класса

Анонимный класс

Локальный класс без имени. Наследует какой-то класс, или имплиментирует какой-то интерфейс

Доступен только в методе, где он определен

У него есть доступ к полям и методам «внешнего» класса

```
public class OuterClass {
    public void someMethod(){
        Callable callable = new Callable() {
        @Override
        public Object call() throws Exception { return null; }
        };
}
}
```

Статический внутренний класс

Объект статического класса **не хранит ссылку на конкретный экземпляр внешнего класса**

Объект статического вложенного класса может существовать сам по себе.

В этом плане статические классы более «независимы», чем нестатические.

Единственный момент — при создании такого объекта нужно указывать название внешнего класса

```
public class OuterClass {
  public static class StaticInnerClass{
   }
}

OuterClass.StaticInnerClass     obj =
            new OuterClass.StaticInnerClass();
```

ИНТЕРФЕЙСЫ И ЛЯМБДА-ВЫРАЖЕНИЯ

Функциональные интерфейсы

- Интерфейс с единственным методом

```
public interface Comparator <T>{
    int compare(T first, T second);
}
public interface Runnable {
    void run();
}
public interface ActionListener {
    void ActionPerformed (ActionEvent e);
}
```

Функциональные интерфейсы

- Аннотация

  @FunctionalInterface

  Чаще всего для реализации интерфейса используют анонимные классы

Без использования лямбда-выражения

```
String [ ] words;
. . .
Arrays.sort (words); // стандартное сравнение
Arrays.sort(words, new Comparator<String>(){
  public int compare(String first, String second){
            return first.length() – second.length();
}});
```

Лямбда-выражение

Можно использовать там, где ожидается объект класса, реализующего функциональный интерфейс

```
String [ ] words;
. . .
Arrays.sort (words,
      (first,second)-> first.length() – second.length());

Comparator<String> comp =
      (first,second)-> first.length() – second.length();

Arrays.sort (words, comp );
```

Лямбда-выражение

- Лямбда-выражение можно присваивать переменной, типом которой является функциональный интерфейс
- Лямбда-выражение можно использовать там, где требуется объект класса, реализующего функциональный интерфейс.

Формат записи

```
Runnable runn = () -> System.out.println(“!!!!!”);
ActionListener lsnr =
        event ->System.out.println(“BUTTON”);
Runnable mult = () -> {
        System.out.println(“!!!!!”);
        for ( ; ;) System.out.println(“.”);
}
BinaryOperator <Long> add = (x,y) -> x+y;
```

Использование значений

При реализации анонимным классом захват контекста – финальные переменные

```
final String name = getUserName();
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Hi "+name);
    }
});
```

Использование значений

В лямбда-выражении – эффективно финальные переменные
(становятся такими, если попадают в лямбда-выражение)

```
String name = getUserName();
//не допустимо name = getOtherName();
button.addActionListener( event ->
                System.out.println("Hi "+name);
);
```

Функциональные интерфейсы

- Пакет

  java.util.function,*

Интерфейс	Метод
Predicat <T>	boolean test (T t)
BinaryOperator<T>	T apply (T f, T s)
Consumer <T>	void accept (T t)
Supplier <T>	T get()

Ссылки на методы

- В лямбда-выражении можно использовать имеющиеся методы классов

```
Arrays.sort(words,(x,y)->x.compareToIgnoreCase(y));
```

- Для упрощения можно использовать ссылку на метод класса String

```
Arrays.sort(words,String::compareToIgnoreCase);
```

Ссылки на методы

```
List <String> ws =Arrays.asList(words);
. . .
// forEach – параметр Consumer<? super T>

ws.forEach(x->System.out.println(x));
ws.forEach(System.out::println);

// replaceAll –параметр UnaryOperator<E>
ws.replaceAll(x->x.toUpperCase());
ws.replaceAll(String::toUpperCase());
```