Модуль #6
ООП в Java

Модуль 6

- **Обертки примитивных типов**
- ООП в Java
- Перегрузка методов
- Наследование и полиморфизм
- Модификаторы final и static
- Модификаторы доступа
- Конструкторы
- Перечисления
- Абстрактные классы
- Интерфейсы
- Принципы проектирования

Обертки примитивов

- Для всех примитивных типов существуют классы обертки.
- Обертки находятся в пакете `java.lang`.

`Byte, Short, Integer, Long, Float, Double, Character`

- В новых версиях java можно использовать обертки абсолютно прозрачно.

```
Integer count   = 1;
Boolean isReady = false;
```

Обертки примитивов

- Java автоматически производит преобразовние из объекта в примитивный тип, если это нужно.

- Объект Boolean будет содержать `true`, если параметр конструктора будет равен строке “true” в любом регистре.

```
Boolean isReady = new Boolean(“TRue”); //true

Boolean isReady = new Boolean(“Yes”); //false
```

Обертки примитивов

- Каждый класс содержит набор констант с максимальными и минимальными значениями.

```
Integer.MIN_VALUE     Integer.MAX_VALUE
```

- Каждый класс содержит статические методы для преобразования типа из строки.

```
Double.parseDouble(String s)
```

- Числовые типы наследуются от класса `Number`.

Обертки примитивов

- Класс Double содержит метод проверки на бесконечность.

```
Double.isInfinity(double d)
```

- Класс Integer содержит полезные методы по работе с бинарным представлением целых числами.

```
Integer.reverse(int i)

Integer.bitCount(int i)

Integer.numberOfLeadingZeros(int i)
```

Обертки примитивов

- Для работы с большими числами можно использовать классы из пакета `java.math`:

  `BigInteger` и `BigDecimal`

- Числа хранятся в строковом виде.

```
BigInteger number = new BigInteger("33");
BigInteger big = number.pow(10000);
```

Модуль 6

- Обертки примитивных типов
- **ООП в Java**
- Перегрузка методов
- Наследование и полиморфизм
- Модификаторы final и static
- Модификаторы доступа
- Конструкторы
- Перечисления
- Абстрактные классы
- Интерфейсы
- Принципы проектирования

ООП в Java

- Поддержка ООП заложена в Java изначально
  (инкапсуляция, наследование, полиморфизм)
- В Java все является объектом, кроме примитивных типов.
- Исполняемый код может находиться только в классе.
- Стандартная библиотека представляет огромное количество классов, но можно создавать свои.

Модуль 6

- Обертки примитивных типов
- ООП в Java
- **Перегрузка методов**
- Наследование и полиморфизм
- Модификаторы final и static
- Модификаторы доступа
- Конструкторы
- Перечисления
- Абстрактные классы
- Интерфейсы
- Принципы проектирования

Перегрузка методов

- По мере создания класса часто возникает необходимость повторного использования имени метода.
- Использование одного и того же имени метода с различным набором параметров называется перегрузкой(Overload).
- Фактически, перегруженные методы могут иметь различные типы возвращаемых значений и списки возбуждаемых исключений.

Перегрузка методов

```
public void aMethod(String s) { }
public void aMethod() { }
public void aMethod(int i, String s) { }
public void aMethod(String s, int i) { }
```

- Рассматривается только тип аргумента, а не название параметра, поэтому следующий метод не считается перегруженным:

```
public void aMethod(int j, String name) { }
```

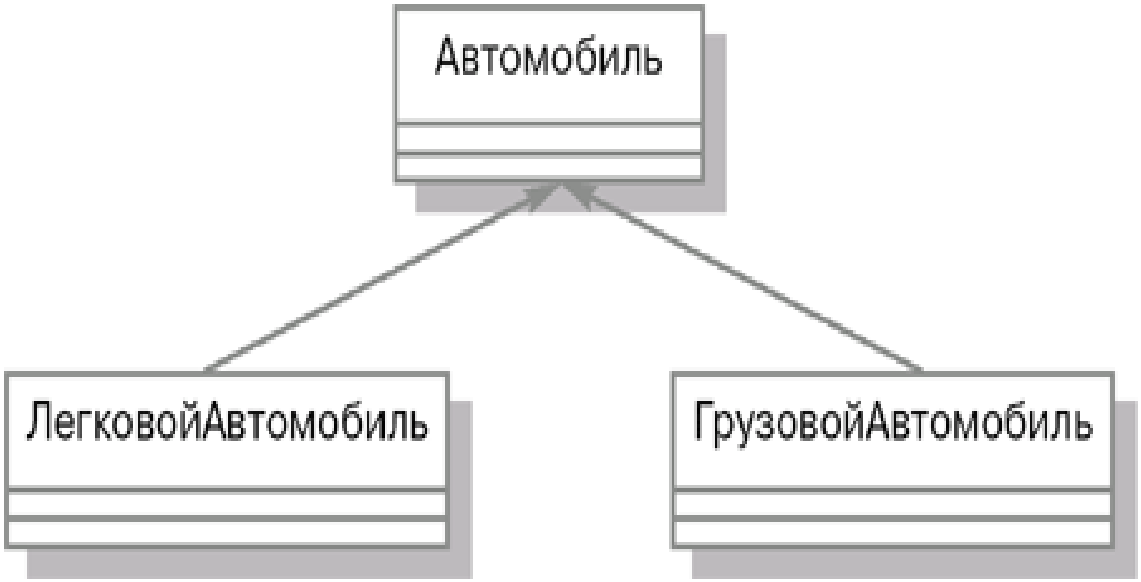
Внимание! Перегруженные мотоды не могут отличаться только возвращаемым значением.

Модуль 6

- Обертки примитивных типов
- ООП в Java
- Перегрузка методов
- **Наследование и полиморфизм**
- Модификаторы final и static
- Модификаторы доступа
- Конструкторы
- Перечисления
- Абстрактные классы
- Интерфейсы
- Принципы проектирования

Наследование

- Наследование – принцип, позволяющий описать новый класс на основе уже существующего.



- Основная цель наследования — повторное использование кода.

Наследование

- Пример:
  - Базовый класс «сотрудник вуза», от которого наследуется классы «аспирант», «профессор» и т.д. (в программе создаются объекты на основе классов “аспирант”, “профессор”, но нет смысла создавать объект на основе класса “сотрудник вуза”).

```
public class UniversityEmployee {
    Date dob;
}

class Professor extends UniversityEmployee {

}

class Student extends UniversityEmployee {

}
```

Наследование

- Модель наследования Java предполагает простое (НЕ множественное) наследование.

- Задача наследования поведения 2-х объектов решается с помощью реализации 2-х интерфейсов.

Наследование

```
class Rectangle {
    int x, y, w, h;
    public void setSize(int w, int h) {
        this.w = w;
        this.h = h;
    }
}
class DisplayedRectangle extends Rectangle {
    public void setSize(int w, int h) {
        this.w = w;
        this.h = h;
        redisplay(); // implementation
    }
    public void redisplay() {
        // implementation not shown
    }
}
```

Наследование

- Для того, чтобы не дублировать код родительского метода, его можно вызвать из замещаемого метода с помощью ключевого слова `super`:

```
super.parentMethod(args1, ...)
```

- С помощью `super()` можно вызвать только метод прямого родителя.

Наследование

```
class Rectangle {
    int x, y, w, h;
    public void setSize(int w, int h) {
        this.w = w;
        this.h = h;
    }
}
class DisplayedRectangle extends Rectangle {
    public void setSize(int w, int h) {
        super.setSize(w, h);
        redisplay(); // implementation
    }
    public void redisplay() {
        // implementation not shown
    }
}
```

Наследование

- Замещающий метод должен:
  - Иметь такое же имя и список аргументов, что и метод родительского класса.
  - Возвращаемый тип должен быть тем же или его подклассом.
- Требования к переопределяемому методу:
  - `final` не может быть замещен.
  - Модификатор доступа должен быть не уже.
  - Замещяемый метод может возбуждать исключения того же типа или подкласса.

Наследование

```
class TheSuperclass {
    Number getValue() throws Exception{
        return new Long(33);
    }
}

class TheSubclass extends TheSuperclass {
    Float getValue() throws IOException {
        return new Float(1.23f);
    }
}
```

Наследование

```
public static void main(String[] args) {
    Rectangle[] recs = new Rectangle[4];
    recs[0] = new Rectangle();
    recs[1] = new DisplayedRectangle();
    recs[2] = new DisplayedRectangle();
    recs[3] = new Rectangle();
    for (int r = 0; r < 4; r++) {
        int w = ((int) (Math.random() * 400));
        int h = ((int) (Math.random() * 200));
        recs[r].setSize(w, h);
    }
}
```

Полиморфизм

- Полиморфизм – взаимозаменяемость объектов с одинаковым контрактом («интерфейсом»).

- т.о. мы формально работаем с разными формами базового класса через ссылку данного типа (интерфейс).

- В примере, элемент массива `rect[i]` – есть ссылка базового `Rectangle`.

- При этом должен быть механизм, который при вызове базового класса фактически вызывает метод наследника.

Наследование

- В ОО языках версия метода, который необходимо вызвать, определяется классом объекта, для которого происходит вызов.
- Это “отложенное” решение называется позднее связывание.
- Поиск и выполнение фактического кода – задача JVM.

Наследование

- Аргументы метода также полиморфны.

```
public class TaxService {
   public TaxRate findTaxRate(Employee e) {
      // calculate the employee's tax rate
   }
}

// Meanwhile, elsewhere in the application class
TaxService taxSvc = new TaxService();
Manager m = new Manager();
TaxRate t = taxSvc.findTaxRate(m);
```

Наследование

- Коллекции объектов с одним и тем же типом называются гомогенными колекциями.

```
MyDate[] dates = new MyDate[2];
dates[0] = new MyDate(22, 12, 1964);
dates[1] = new MyDate(22, 7, 1964);
```

- Коллекции объектов с различными типами называются гетерогенными.

```
Employee [] staff = new Employee[1024];
staff[0] = new Manager();
staff[1] = new Employee();
staff[2] = new Engineer();
```

Наследование

- Оператор **instanceof** служит для проверки фактического типа объекта.

```
public class Employee extends Object
public class Manager extends Employee
public class Engineer extends Employee

public void doSomething(Employee e) {
   if ( e instanceof Manager ) {
   // Process a Manager
   } else if ( e instanceof Engineer ) {
   // Process an Engineer
   } else {
   // Process any other type of Employee
   }
}
```

Наследование

- Из определения полиморфизма, ссылка – это спецификация базового типа. Фактически объект может являться наследником этого типа.

- Оператор **cast**() позволяет восстановить ссылку на фактический тип объекта.

```
public void doSomething(Employee e) {
    if ( e instanceof Manager ) {
        // Восстановление типа Manager
        Manager m = (Manager) e;
        System.out.println(m.getDepartment());
    }
    // rest of operation
}
```

Модуль 6

- Обертки примитивных типов
- ООП в Java
- Перегрузка методов
- Наследование и полиморфизм
- **Модификаторы final и static**
- Модификаторы доступа
- Конструкторы
- Перечисления
- Абстрактные классы
- Интерфейсы
- Принципы проектирования

Наследование

- Модификатор `final` применим к классам, переменным и методам.

Наследование

- Финальный класс:
  - не может быть расширен.

```
public final class java.lang.Math { }
```

- Финальная переменная:
  - не может быть изменена после того как ей было присвоено значение.
  - играет роль константы в Java.
  - Если переменная – это ссылка, то изменению не подлежит значение ссылки.

Наследование

- Финальный метод:
  - не может быть замещен в подклассе

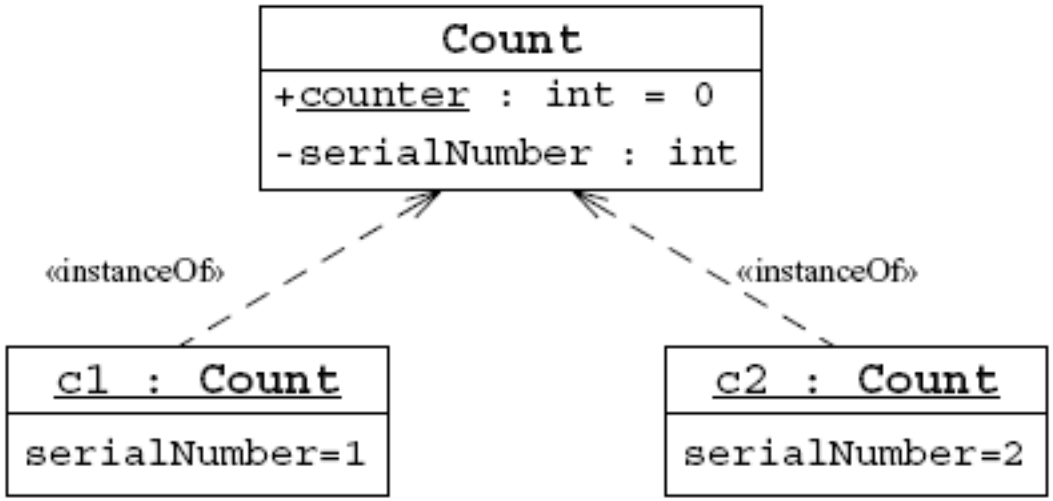
```
class Mammal {
    final void getAround() {
    }
}

class Dolphin extends Mammal {
    void getAround() {
        //Ошибка компиляции: Cannot override the 
method from Test.Mammal
    }
}
```

Модификатор static

- Ключевое слово `static` используется как модификатор для переменных, методов и вложенных классов.
- Декларирует, что атрибут или метод, ассоциированы с целым классом, а не с конкретным экземпляром.
- Статические члены часто называются членами класса.

Модификатор static

- Статический атрибут класса разделяется среди всех атрибутов этого класса.




```
public class Count {
    private int serialNumber;
    public static int counter = 0;
    public Count() {
        counter++;
        serialNumber = counter;
    }
}
```

Модификатор static

- Статический метод класса принадлежит классу.
- Статический метод может и должен вызываться через сам класс, а не через экземпляр класса.
- Может осуществлять доступ к статическим переменным, в котором он объявлен.
- Не имеет ссылки `this`.
- Не может быть замещен.

Модификатор static

```
public class Count3 {
    private int serialNumber;
    private static int counter = 0;
    public static int getSerialNumber() {
        return serialNumber; // COMPILER ERROR!
    }
}
```

Статические инициализаторы

- Класс может содержать код в статическом блоке, который не принадлежит никакому методу.
- Статический блок выполняется ровно один раз в момент загрузки класса.
- Статический блок используется для инициализации статических атрибутов класса.

Статические инициализаторы

```
public class StaticExample {
    static double d = 1.23;
    static {
        System.out.println("Static code: d=" + d
    }

    public static void main(String args[]) {
        System.out.println("main: d = " + d++);
    }
}
```

Статический импорт

- Статический импорт импортирует статические атрибуты и методы класса.

```
import static <pkg_list>.<class_name>.<member_name>;
```

или

```
import static <pkg_list>.<class_name>.*;
```

Статический импорт

```
public cards.tests;

import cards.domain.PlayingCard;
import static cards.domain.Suit.*;

public class TestPlayingCard
{
   public static void main(String... args)
   {
     PlayingCard card1 = new PlayingCard(Spades);
     System.out.println("Card1 is the " +
        card1.getRank());
   }
}
```

Модуль 6

- Обертки примитивных типов
- ООП в Java
- Перегрузка методов
- Наследование и полиморфизм
- Модификаторы final и static
- **Модификаторы доступа**
- Конструкторы
- Перечисления
- Абстрактные классы
- Интерфейсы
- Принципы проектирования

Модификаторы доступа

- Модификаторы доступа определяют какой класс может использовать ту или иную функциональность.

- Под функциональностью будем понимать:
  - Сам класс
  - Член класса
  - Метод или конструктор
  - Вложенный класс

Модификаторы доступа

- Модификаторы доступа:
  - public
  - protected
  - *default (friendly, package)*
  - private
- Возможен максимум один модификатор:

```
class Parser {…}
public class EightDimensionalComples { … }
private int i;
protected double getChiSquared() {…}
private class Horse {…}
Button getBtn(){…}
```

Модификатор public

- Наиболее открытый модификатор доступа – **`public`**.
- **`public`** класс, переменная или метода могут использоваться в программе без всяких ограничений.
- **`public`** метод может быть замещен в подклассе.
- Каждый java файл обязан иметь один **`public`** класс.

Модификатор private

- Наиболее закрытый модификатор доступа – **`private`**.
- Классы верхнего уровня не могут быть объявлены как **`private`**.
- **`private`** метод или переменная может использоватьсятолько в классе, который ее декларировал.
- Доступ к **`private`** переменной или методу запрещен из подкласса (класса-наследника).

Модификатор private

```
class Complex {
    private double real, imaginary;
    public Complex(double r, double i) {
        real = r;
        imaginary = i;
    }
    public Complex add(Complex c) {
        return new Complex(real + c.real, imaginary +
c.imaginary);
    }
}
class Client {
    void useThem() {
        Complex c1 = new Complex(1, 2);
        Complex c2 = new Complex(3, 4);
        Complex c3 = c1.add(c2);
        double d = c3.real; // Illegal!
    }
}
```

Модификатор default

- Default – способ доступа к классам, переменным и методам, когда модификатор доступа не указывается.
- Данные класса, методы и сам класс могут быть default.
- Default элементы класса A могут быть доступны любому классу, находящемуся в том же пакете что и класс A.
- Default метод может быть замещен в любом подклассе того же пакета.

Модификатор protected

- Только переменные и методы могут быть декларированы как `protected`.

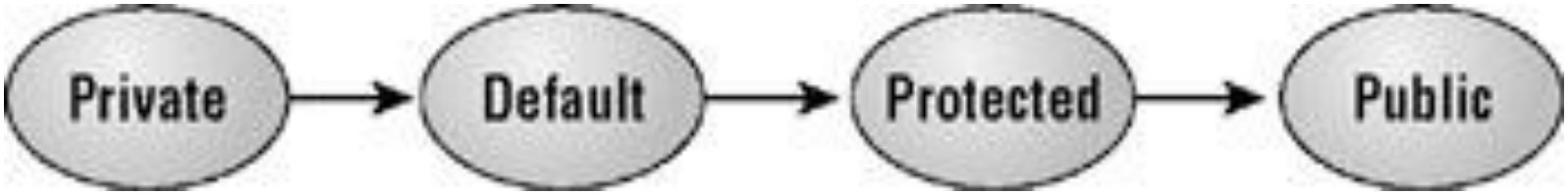
- `protected` элемент класса доступен всем классам того же пакета, точно так же как и default.

- `protected` элемент класса A может быть доступен подклассам класса A, размещенным в произвольных пакетах.

Модификаторы доступа

- При замещении метода нельзя делать область его видимости более закрытой.

- Область видимости замещенного метода может быть такой же, что и область видимости замещаемого метода или шире.



Модуль 6

- Обертки примитивных типов
- ООП в Java
- Перегрузка методов
- Наследование и полиморфизм
- Модификаторы final и static
- Модификаторы доступа
- **Конструкторы**
- Перечисления
- Абстрактные классы
- Интерфейсы
- Принципы проектирования

Конструкторы

- Конструктор – специальный блок инструкций, вызываемый при создании объекта.

- Конструкторы не наследуются обычным способом и должны быть определены для каждого класса самостоятельно.

- Пример вызова конструктора:

```
new MyClass(arg1, arg2, …);
```

Конструкторы

- Необходимо гарантировать, что конструктор потомка вызывется после конструктора родителя.
- Ключевое слово `super(arg1, …)` служит для вызова конструктора предка.
- Нужный конструктор выбирается исходя из списка параметров.

Конструктор по умолчанию

- Конструктор у которого нет аргументов, называется конструктором по умолчанию.

- Если в теле класса явно не указан ни один конструктор, то компилятор создаст его автоматически.

- В этом случае модификатор доступа совпадает с модификатором доступа класса.

Конструкторы

- Конструкторы можно перегружать.
- Список аргументов должен отличаться.
- Можно использовать ссылку **this** для того, чтобы вызвать перегруженный конструктор.

```
public Employee(String name, double salary, Date DoB){
    // реализация опущена
}
public Employee(String name, double salary) {
    this(name, salary, null);
}
public Employee(String name, Date DoB) {
    this(name, 0, DoB);
}
```

Конструкторы

```
public class Employee extends Object {
    private String name;
    private double salary = 15000.00;
    private Date birthDate;

    public Employee(String n, Date DoB) {
        // implicit super();
        name = n;
        birthDate = DoB;
    }
    public Employee(String n) {
        this(n, null);
    }
}
public class Manager extends Employee {
    private String department;
    public Manager(String n, String d) {
        super(n);
        department = d;
    }
}
```

Модуль 6

- Обертки примитивных типов
- ООП в Java
- Перегрузка методов
- Наследование и полиморфизм
- Модификаторы final и static
- Модификаторы доступа
- Конструкторы
- **Перечисления**
- Абстрактные классы
- Интерфейсы
- Принципы проектирования

Перечисления

- Очень часто требуется вводить перечислимые типы:

```
public class PlayingCard {
   public static final int SUIT_SPADES = 0;
   public static final int SUIT_HEARTS = 1;
   public static final int SUIT_CLUBS = 2;
   public static final int SUIT_DIAMONDS = 3;
   private int suit;
   public PlayingCard(int suit) {
      this.suit = suit;
   }

   public String getSuitName() {
      String name = "";
      switch (suit) {
      case SUIT_SPADES:
         name = "Spades";
         break;
      case SUIT_HEARTS:
         name = "Hearts";
         break;
      ...
```

Перечисления

- В Java 1.5 появился новый механизм перечислений (enum).

- Перечисление – это подкласс класс `java.lang.Enum`. Перечисление решает указанную проблему и может быть использовано в операторе switch.

- Перечисление – это обычный класс, с некоторыми ограничениями.

Отличие перечислений от классов

- Декларируются с помощью ключевого слова **`enum`**
- Нельзя явно создавать экземпляр перечисления.
- Нельзя расширять перечисление.
- Перечисление может быть аргументом в **`switch`**.
- Имеет встроенный метод `name()`, возвращающий значение перечисления.

Перечисления

```
public enum LightState {
    RED, YELLOW, GREEN;
}

public static void main(String[] args) {
    switch (nextTrafficLight.getState()) {
    case LightState.RED:
        stop();
        break;
    case LightState.YELLOW:
        floorIt();
        break;
    ...
}
```

Перечисления

```
enum Suit {
    DIAMOND(true), HEART(true), CLUB(false),
SPADE(false);

    private boolean red;

    Suit(boolean b) {
        red = b;
    }
    public boolean isRed() {
        return red;
    }
    public String toString() {
        String s = name();
        s += red ? ":red" : ":black";
        return s;
    }
}
```

Модуль 6

- Обертки примитивных типов
- ООП в Java
- Перегрузка методов
- Наследование и полиморфизм
- Модификаторы final и static
- Модификаторы доступа
- Конструкторы
- Перечисления
- **Абстрактные классы**
- Интерфейсы
- Принципы проектирования

Абстракция

- Абстракция – это существенные характеристики объекта, которые отличают его от всех других объектов, четко определяя его концептуальные границы.

- Абстракция позволяет работать с объектами, не вдаваясь в реализации, которую можно гибко менять, не затрагивая клиентский код.

- Абстрактный класс – это класс, определяющий некоторый контракт, но не являющийся законченной реализацией.

Абстракция

- Абстрактый метода – метод не имеющий реализации на этом уровне, а лишь объявляющий контракт.

- Абстрактный метод не реализуется для класса, в котором описан, однако, должен быть реализован для его неабстрактных потомков.

```
public abstract class Bill
{
    public abstract int getSum();
}
```

Абстракция

- Абстрактый класс может (но в общем случае не должен) содержать абстрактные методы.
- Абстрактный класс может содержать реализацию не абстрактных методов.
- Если класс содержит хотя бы один абстрактный метод, он считается абстрактным.

Абстракция

- Нельзя создать объект абстрактного класса, но можно создать экземпляр не абстрактного потомка.

```
public abstract class Shape {
      int color;
      Coordinates startPoint;

      abstract public void draw();
}

class Circle extends Shape {
      public void draw() {
         // Здесь рисуется круг
      }
}
```

Модуль 6

- Инкапсуляция
- Coupling and cohesion
- Перегрузка методов
- Наследование
- Модификаторы final и static
- Модификаторы доступа
- Конструкторы
- Перечисления
- Абстрактные классы
- **Интерфейсы**
- Принципы проектирования

Интерфейсы

- Интерфейс – это контракт между клиентским кодом и классом, котороый реализует (имплементирует) этот интерфейс.

- Полиморфизм – это прицип ООП. В Java этот принцип реализуется с помощью абстрактных классов и интерфейсов.

- Можно считать что интерфейс – это абстрактный класс, у которого все методы – абстрактные.

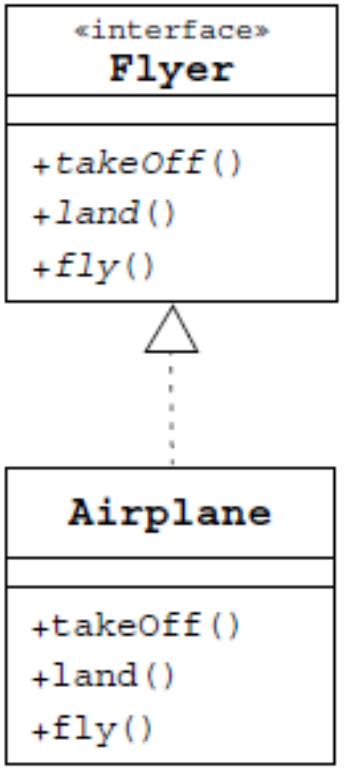
Интерфейсы

- Интерфейс Java определяется с помощью ключевого слова `interface`.

- Класс, который реализует интерфейс указывает это с помощью слова `implements`.

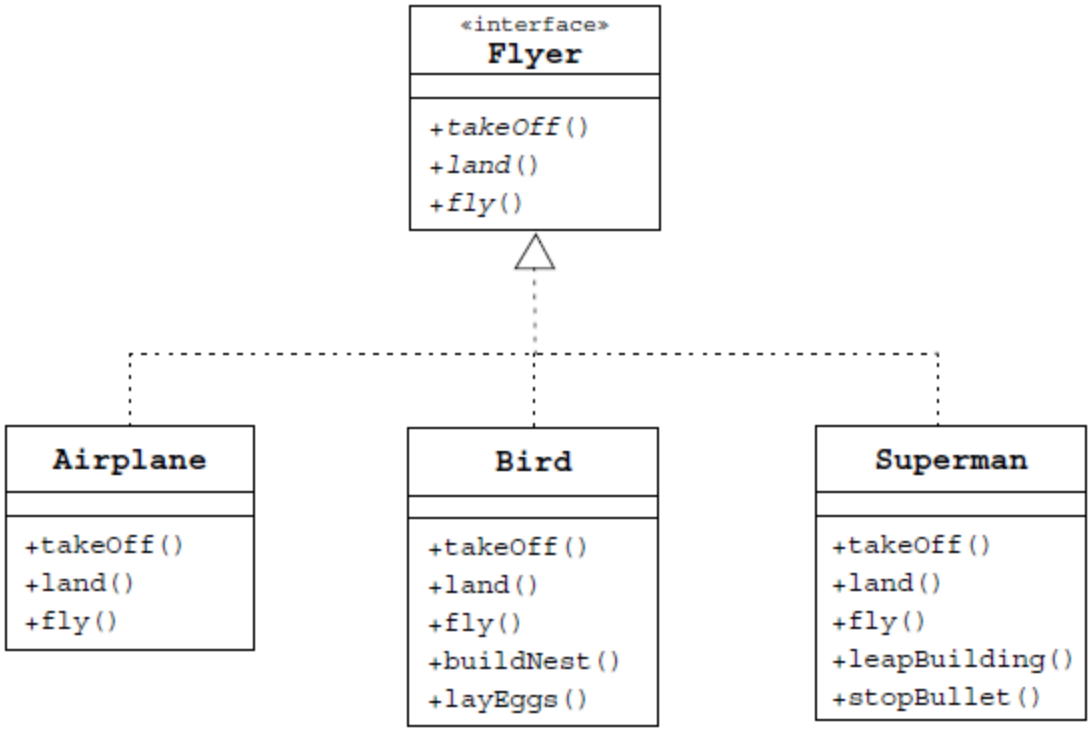
Интерфейсы

```
public interface Flyer {
    public void takeOff();
    public void land();
    public void fly();
}

public class Airplane implements Flyer {
    public void takeOff() {
        // accelerate until lift-off
        // raise landing gear
    }
    public void land() {
    }
    public void fly() {

    }
}
```



Интерфейсы

- т.к. интерфейс – это только спецификация некоторого поведения, то класс может наследовать несоколько интерфейсов.



Интерфейсы

```
public interface Flyer {
    public void fly();
}
public interface Swimer {
    public void swim();
}
public class Penguin implements Swimer {
    public void swim() {
        // Пингвин умеет плавать, но не умеет летать
    }
}
public class Duck implements Flyer, Swimer {
    public void fly() {

    }
    public void swim() {

    }
}
```

Интерфейсы

- т.к. интерфейс – это соглашение, а не реализация, то:
  - Нельзя создать экземпляр
  - Нет конструкторов
  - Нет данных экземпляра

Внимание! Интерфейс может содержать статические финальные данные

Интерфейсы

- модификтор **`public static final`** опционален.

```
interface Flyer {
    public final static int NB_WINGS = 2;
    void fly();
}
// Это эквивалентно:
interface Flyer {
    int NB_WINGS = 2;
    void fly();
}
```

Интерфейсы

- Невложенный интерфейс не может быть `private`.
- Интерфейс не может быть `protected`.
- `public` интерфейс может быть реализован любым классом.
- default интерфейс может быть реализован любым классом того же пакета.

Интерфейсы

- Подразумевается, что все методы интерфейса объявлены как **`public abstract`**.

```
interface Flyer {
    public final static int NB_WINGS = 2;
    void fly();
}
// Это эквивалентно:
interface Flyer {
    int NB_WINGS = 2;
    void fly();
}
```

Интерфейсы

- При замещении метода нельзя сужать область видимости.

```
public class Penguin implements Swimer {
    void swim() {
        // Ошибка компилятора: Cannot reduce the
visibility of the inherited method from Swimer
    }
}
```

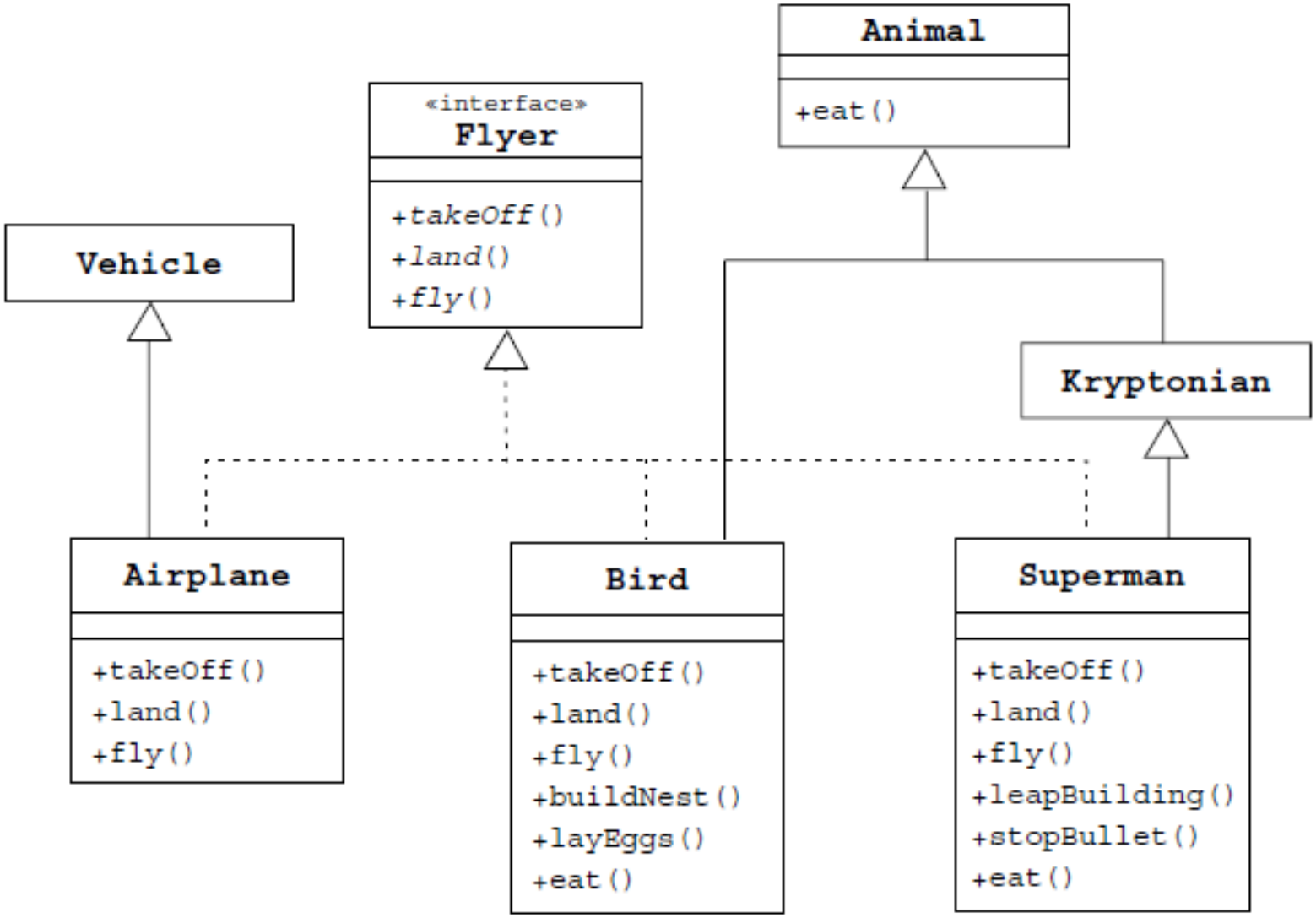
Интерфейсы

- Поведение может расширять другое поведение.

```
public interface Set extends Collection, Comparator
{
   ...
}
```

Внимание! Класс реализующий расширенный интерфейс должен реализовать методы обоих интерфейсов.

Комплексный пример



Комплексный пример

```
public class Bird extends Animal implements Flyer {
   public void takeOff() { /* take-off implementation */
   public void land() { /* landing implementation */ }
   public void fly() { /* fly implementation */ }
   public void buildNest() { /* nest building behavior */
   public void layEggs() { /* egg laying behavior */ }
   public void eat() { /* override eating behavior */ }
}
```

Модуль 6

- Обертки примитивных типов
- Coupling and cohesion
- Перегрузка методов
- Наследование
- Модификаторы final и static
- Модификаторы доступа
- Конструкторы
- Перечисления
- Абстрактные классы
- Интерфейсы
- **Принципы проектирования**

Принципы проектирования

- Два подхода к описанию класса:
  - Is a (является)

    «Дом – *это* помещение, *в котором* живет семья и домашние животные»

  - Has a (имеет)

    В то же время, семья и животное находятся в доме, т.е. дом «содержит» их.

```
public class Home extends House {
    Family inhabitants;
    Pet thePet;
}
```

Принципы проектирования

- При проектировании системы первым шагом является перечисление всех сущностей системы.
- Затем необходимо решить в каком отношении (is a или has a) находятся сущности.
- Затем можно переходить к проектировке используя средства UML.

Упражнение 9

Создание Bank application.