



Реализация принципа ООП: полиморфизм

Полиморфизм. Раннее и позднее связывание

- ❑ **Полиморфизм** – это способность объекта принимать различные формы.

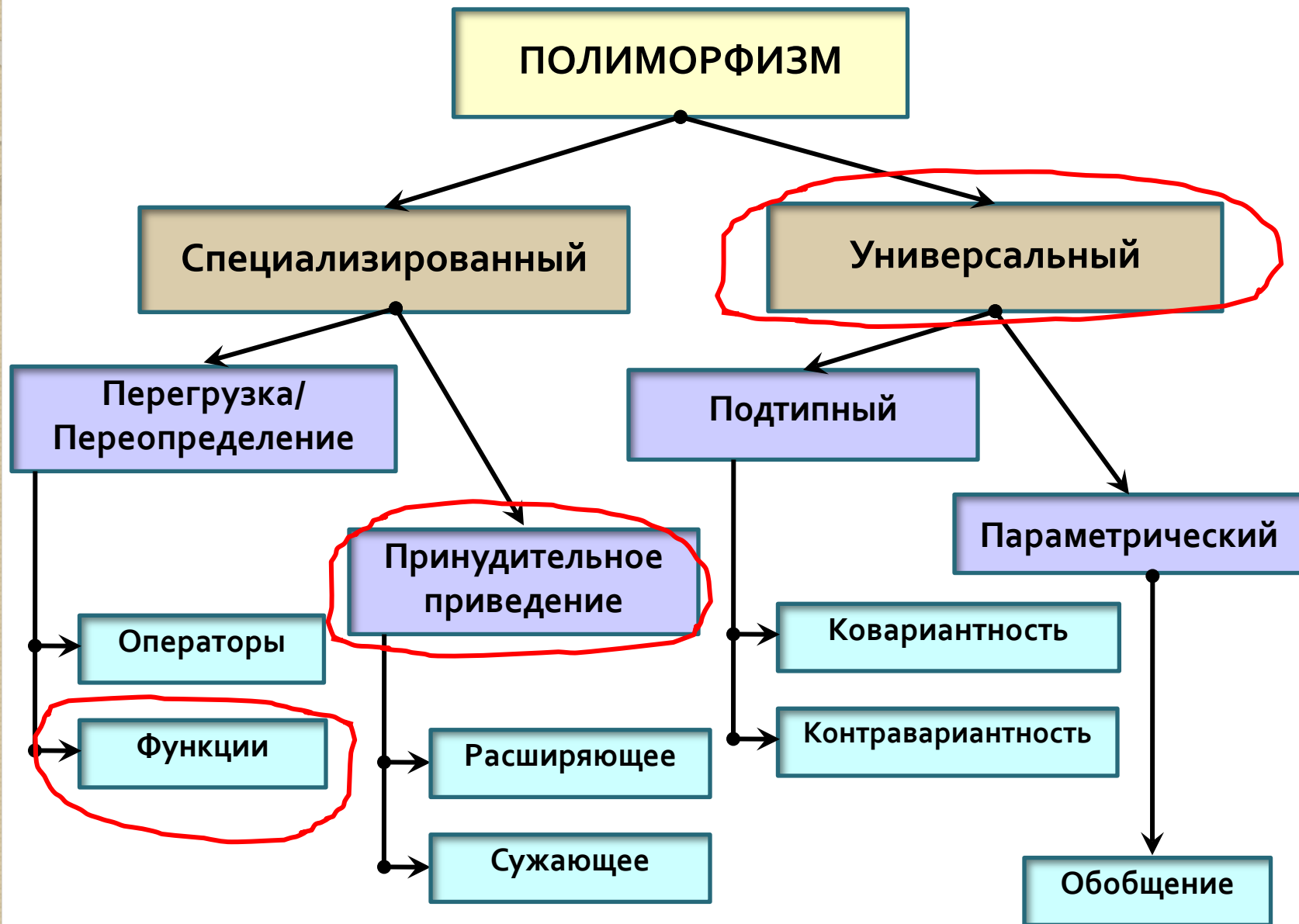
(наиболее распространенное использование полиморфизма в ООП: когда ссылка на суперкласс используется, чтобы обратиться к объекту подкласса).



Полиморфизм - это один интерфейс для множества реализаций

➤ **Подклассы некоторого класса могут определить своё собственное уникальное поведения и все же при этом разделять некоторую или полную функциональность суперкласса.**

Полиморфизм. Раннее и позднее связывание



ДОСТОИНСТВА ПОЛИМОРФИЗМА

- ❑ Повторное использование;
- ❑ Надежность;
- ❑ Более модульная структура;
- ❑ Гибкость в расширении.

СВЯЗЫВАНИЕ

- Существует два типа связывания методов в языке Java:
 - ✓ **ранее** (статическое) связывание – early binding;
 - ✓ **позднее** (динамическое) связывание – late binding.



РАНЕЕ СВЯЗЫВАНИЕ

- ❑ Происходит во время компиляции, т.е. код «знает», какой метод вызывать после компиляции исходного кода на Java в файлы классов:
 - основывается на типе ссылочной переменной (вызов статических методов);
 - используется для разрешения перегруженных методов.

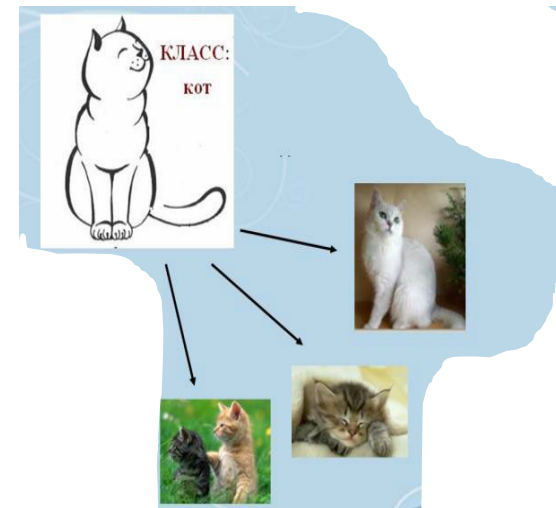
ПОЗДНЕЕ СВЯЗЫВАНИЕ

- ❑ Происходит во время выполнения, т.е. компилятор не имеет возможности определить какой вариант метода с одинаковым названием следую вызвать:
 - проявляется в возможности вызова переопределенного метода подкласса через ссылку на суперкласс;
 - основывается на типе объекта.

Полиморфизм. Раннее и позднее связывание

Пример 1: позднее связывание

```
class Cat {  
    public void move() {  
        System.out.println("Cat move");  
    }  
}  
class BritishCat extends Cat {  
    @Override  
    public void move() {  
        System.out.println("British cat move");  
    }  
}  
class PersianCat extends Cat {  
    @Override  
    public void move() {  
        System.out.println("Persian cat move");  
    }  
}
```



Продолжение примера 1,

```
public class Main {  
    public static void main(String[] arg) {  
        Cat[] myCats = { new Cat(),  
                        new BritishCat(), new PersianCat() };  
        for (Cat cat : myCats) {  
            cat.move();  
        }  
    }  
}
```



Выполнение варианта
метода зависит от типа
объекта, а не ссылки

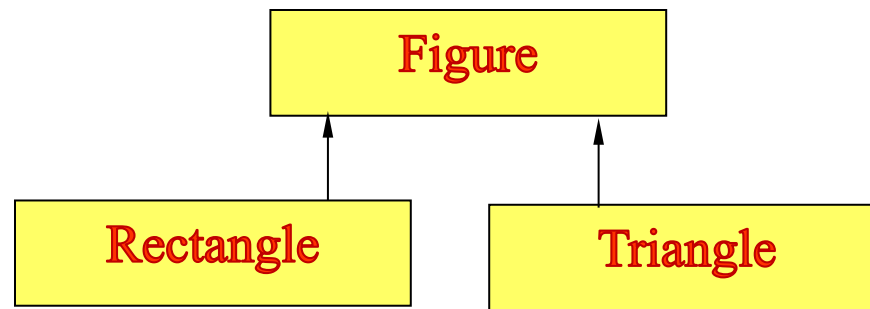
Вывод в консоль:

Cat move
British cat move
Persian cat move

Полиморфизм. Раннее и позднее связывание

Пример 2:

```
class Figure {  
    double dim1;  
    double dim2;  
  
    Figure(double a, double b) {  
        dim1 = a;  
        dim2 = b;  
    }  
  
    double area() {  
        System.out.println("Площадь фигуры не определена.");  
        return 0;  
    }  
}
```



Продолжение примера 2.

```
class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // переопределение метода area для четырехугольника
    double area() {
        System.out.println("В области четырехугольника.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // переопределение метода area для прямоугольного треугольника
    double area() {
        System.out.println("В области треугольника.");
        return dim1 * dim2 / 2;
    }
}
```

Полиморфизм. Раннее и позднее связывание

Продолжение примера 2,

```
class FindAreas {  
    public static void main(String args[]) {  
        Figure f = new Figure(10, 10);  
        Rectangle r = new Rectangle(9, 5);  
        Triangle t = new Triangle(10, 8);  
        Figure figref;  
  
        figref = r;  
        System.out.println("Площадь равна " + figref.area());  
  
        figref = t;  
        System.out.println("Площадь равна " + figref.area());  
  
        figref = f;  
        System.out.println("Площадь равна " + figref.area());  
    }  
}
```

Вывод в консоли:

```
В области четырехугольника.  
Площадь равна 45  
В области треугольника.  
Площадь равна 40  
Область фигуры не определена.  
Площадь равна 0
```

Полиморфизм. Раннее и позднее связывание

- Раннее связывание для статических методов – вызов варианта метода зависит не от типа объекта, а от типа ссылки на объект.

Пример 3,

```
class Insurance {
```

```
    public static final int LOW = 100;
```

```
    public int premium() {
```

```
        return LOW;
```

```
    }
```

```
    public static String category() {
```

```
        return "Insurance";
```

```
    }
```

```
}
```

Метод экземпляра

Метод класса

Продолжение примера 3:

```
class CarInsurance extends Insurance {  
    public static final int HIGH = 200;  
    public int premium() {  
        return HIGH;  
    }  
    public static String category() {  
        return "Car Insurance";  
    }  
}
```

Переопределение
метода экземпляра

Переопределение
метода класса

Продолжение примера 3:

```
public class Main {  
    public static void main(String[] args) {  
        Insurance current = new CarInsurance();  
        int premium = current.premium();  
        String category = current.category();  
        System.out.println("premium : " + premium);  
        System.out.println("category : " + category);  
    }  
}
```

Раннее (статическое)
связывание на
основе типа
Insurance ссылки
current

Позднее
(динамическое)
связывание на основе
типа объекта
CarInsurance

Вывод в консоли:

premium : 200
category : Insurance



ВЫВОДЫ:

- ✓ Суперклассы и подклассы образуют иерархию по степени увеличения специализации.
- ✓ Такая иерархия позволяет подклассу определять собственные методы (поведение) при сохранении единообразия интерфейса.
- ✓ Суперкласс может определять общую форму методов, которые будут использоваться всеми его подклассами.
- ✓ Тип объектной ссылки определяет разновидность объектов, на которые через нее можно ссылаться.
- ✓ Через ссылку на суперкласс можно вызывать только унаследованные методы суперкласса (т.е. методы, определенные только в подклассе, нельзя вызвать через объектную ссылку на суперкласс).



Преобразование ТИПОВ ССЫЛКИ

Преобразование типов ссылки

- ❑ В Java приведение типов объектов – это когда ссылка на объект может быть приведена к типу другой ссылки на объект (*приведение возможно к своему типу класса или к одному из его подклассов, или суперклассов, или интерфейсов*):

(ReferenceType)RelationalExpression

Тип, к которому
приводится ссылка

Ссылка, которая
приводится

Например,

```
Object obj = "abracadabra";
```

```
String str = (String)obj;
```



*Если попытаться выполнить приведение класса, а объект не входит эту ветку в иерархии наследования, то происходит ошибка типа **ClassCastException***

Пример 4, ошибочное приведение типов объектов на базе иерархии из примера 1:

```
public class Main {  
    public static void main(String[] arg) {  
        Cat myCat = new BritishCat();  
        BritishCat myCat2 = (BritishCat)myCat;  
        PersianCat myCat3 = (PersianCat)myCat;  
    }  
}
```

OK!

OK!

Ошибка времени выполнения
ClassCastException!



Во избежание такой ошибки можно проверить корректность приведения с помощью оператора *instanceof*.

- ❑ Оператора **instanceof** вырабатывает **true**, если значение *RelationalExpression* не нулевое (null) и может быть приведено к *ReferenceType* не выбрасывая **ClassCastException**.

В противном случае результат - **false**.

➤ Синтаксис:

< RelationalExpression> **instanceof** < ReferenceType >

Например,

```
if (myCat instanceof PersianCat) {  
    System.out.println("Persian cat!");  
    PersianCat myCat3 = (PersianCat) myCat;  
} else {  
    System.out.println("Not Persian cat!");  
}
```



Класс типа `final` и модификатор `protected`

FINAL-КЛАССЫ

- ❑ Если класс объявить как **final**, то он не может иметь подкласса.

```
final class Cat {      //... }
```

```
final class Cat {  
    //...
```

```
}
```

```
class    BritishCat extends Cat {  
    //...
```

```
}
```



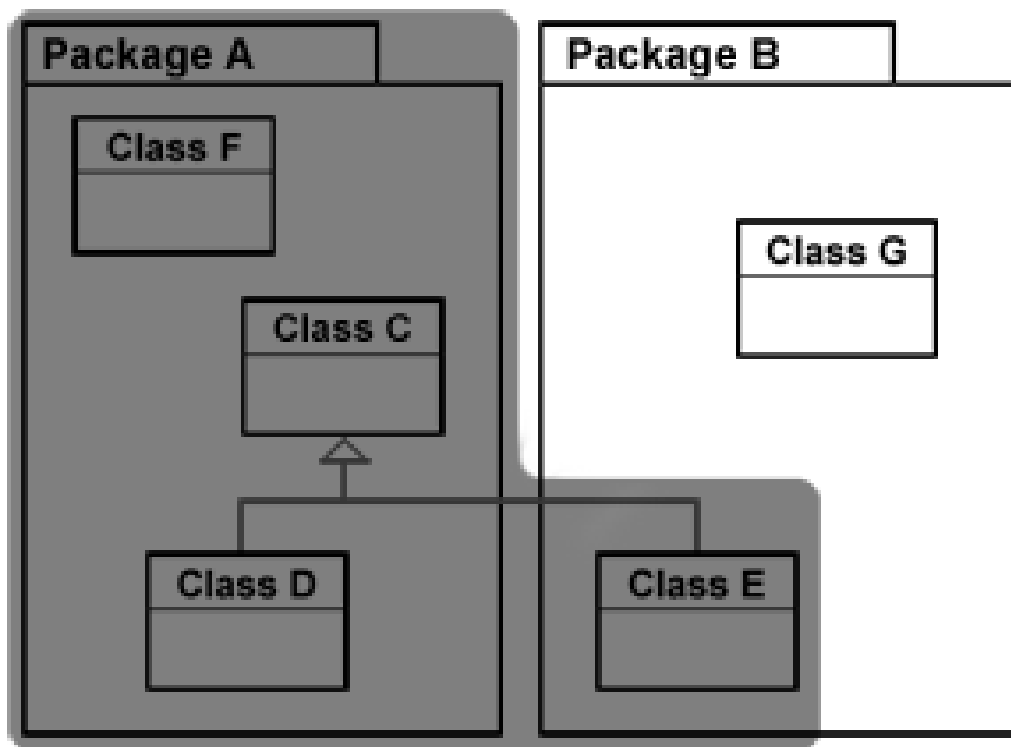
Это полезно при
создании
неизменяемого
класса.

Например, как
класс **String**.

Не может быть
суперклассом

ОСОБЕННОСТИ МОДИФИКАТОРА `PROTECTED`

- ❑ Модификатор доступа **`protected`** указывает, что элемент может быть доступен только в его собственном пакете и внутри подкласса, который находится в другом пакете.



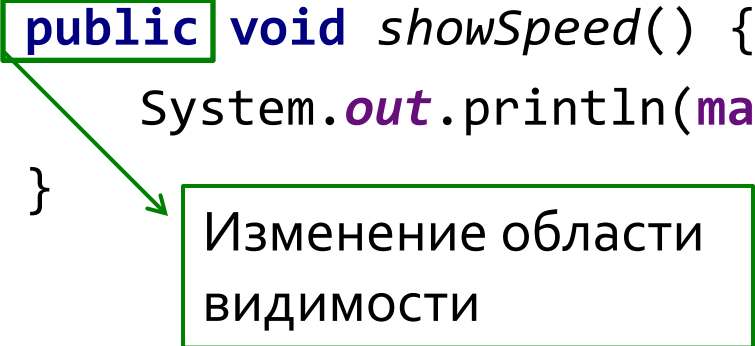
i При переопределении метода можно изменить модификатор доступа. Однако, он не должен стать более ограничивающим. Например, `public` менее ограничивающий, чем `protected`.

Класс типа `final` и модификатор `protected`

Пример 5:

```
package com.knu.basic_transport;
class Vehicle {
    protected int maxSpeed = 230;
    protected void showSpeed() {
        System.out.println(maxSpeed);
    }
}

package com.knu.derived_transport;
import com.knu.basic_transport.Vehicle;
class Bicycle extends Vehicle {
    public void showSpeed() {
        System.out.println(maxSpeed);
    }
}
```



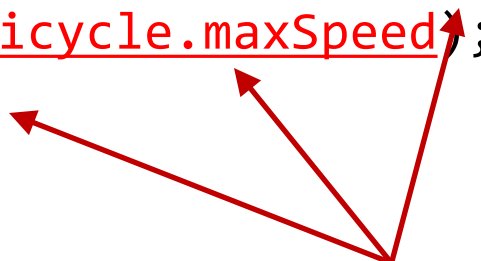
Изменение области
видимости

Класс типа `final` и модификатор `protected`


Продолжение примера 5,

```
package com.knu.test_transport;
import com.knu.basic_transport.Vehicle;
import com.knu.derived_transport.Bicycle;

class Demo {
    public static void main(String[] args) {
        Vehicle vehicle = new Vehicle();
        Bicycle bicycle = new Bicycle();
        System.out.println(vehicle.maxSpeed);
        System.out.println(bicycle.maxSpeed);
        vehicle.showSpeed();
        bicycle.showSpeed();
    }
}
```



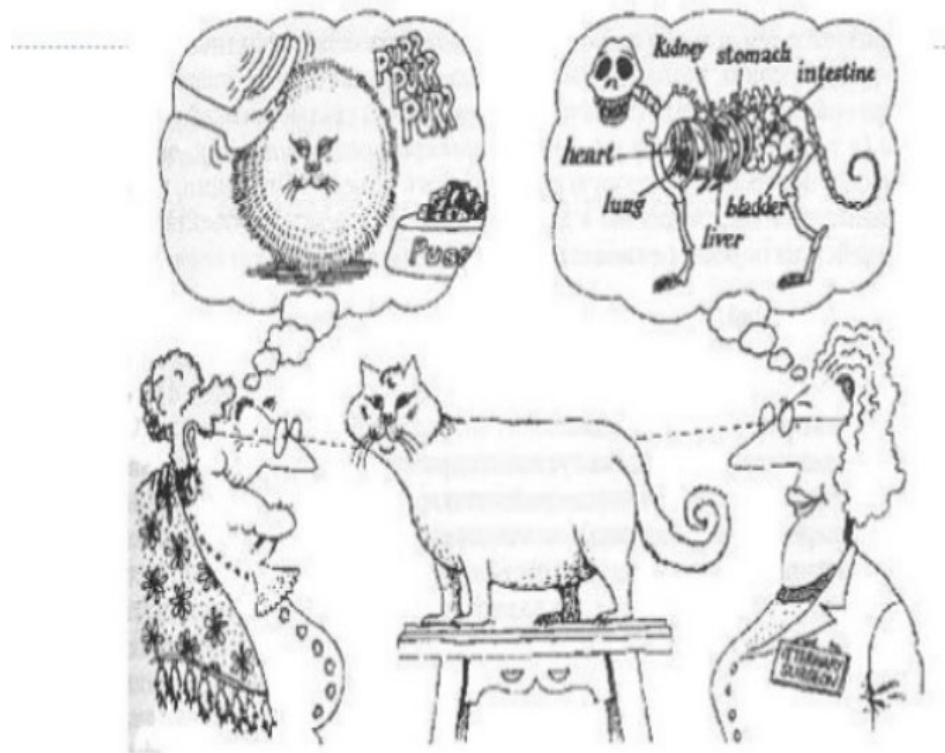
Ошибка
компиляции



Абстракция. Абстрактные классы

Понятие абстракции

- **Абстракция** заключается в том, что сущность произвольной сложности можно рассматривать, а также производить определенные действия над ней, как над *единым целым*, не вдаваясь в *детали внутреннего построения и функционирования*.



Абстракция фокусируется на существенных с точки зрения наблюдателя характеристиках объекта

Понятие абстракции

- ❑ **Абстрагирование** — это способ выделить набор значимых характеристик объекта, исключая из рассмотрения не значимые.
- ❑ Существует понятие — степень абстракции:
 - высокая степень дает только Приблизительное описание объекта и не позволяет правильно моделировать его поведение;
 - низкая степень абстракции сделает модель очень сложной, перегруженной деталями, и потому непригодной для использования.



Понятие абстракции

- ❑ **Абстрагирование** необходимо как способ познания и описания окружающего мира для обмена информацией:
 - абстракции позволяют провести **декомпозицию** предметной области на набор понятий и связей между ними.

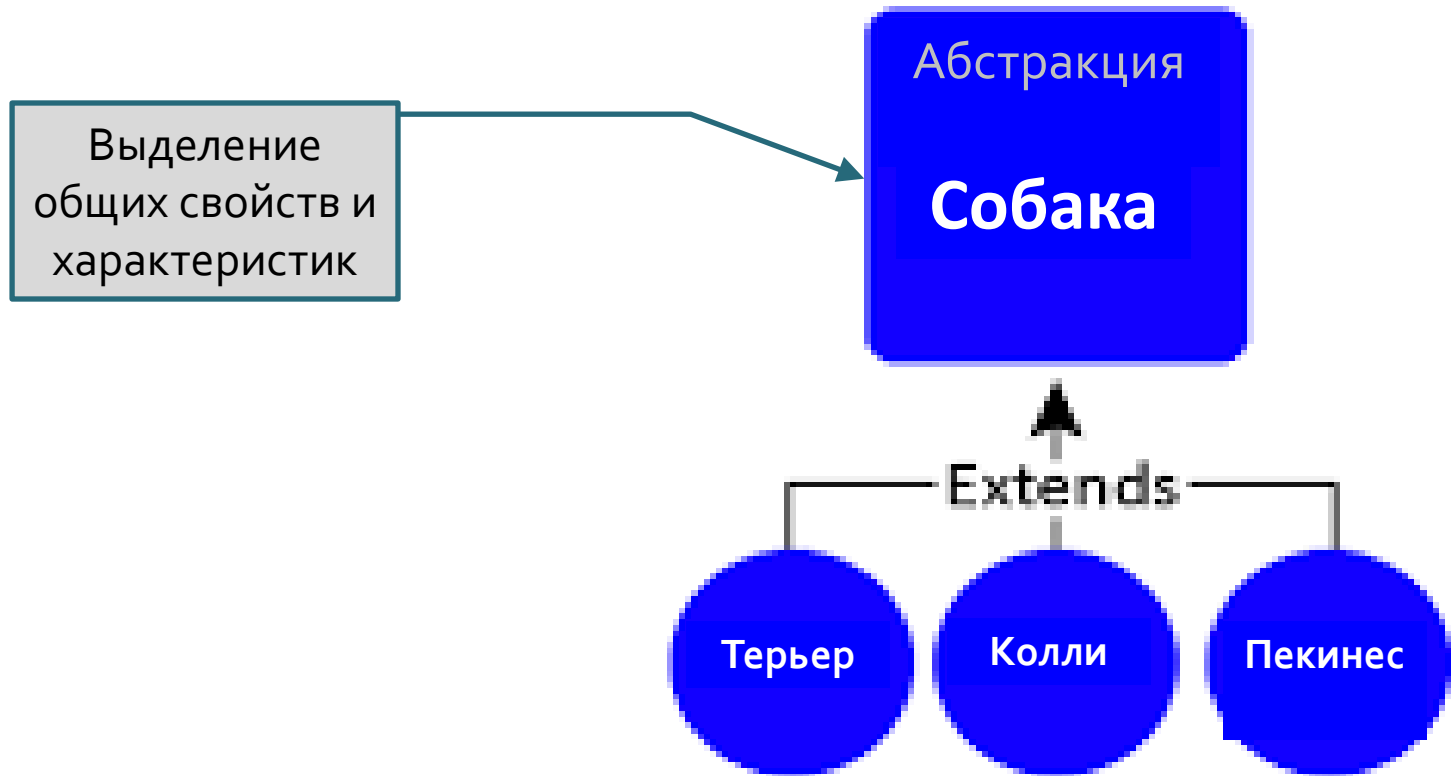
Например,



Все предметы собраны из детского конструктора, но без труда в них можно узнать дома, окна, двери, городские кварталы, людей.

Понятие абстракции

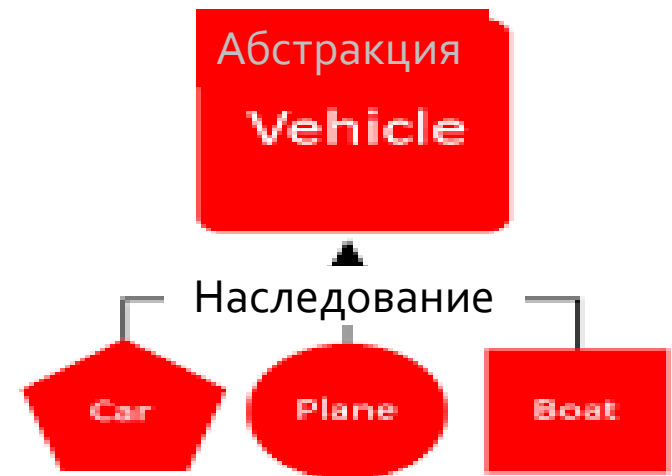
- ❑ **Классы** — это абстракции механизмов группировки и обобщения (при этом группировка достигается тем, что похожим объектам сопоставляется один класс, а обобщение достигается за счет иерархии классов).
- ❑ *Абстракция представляет не весь объект целиком, а только лишь его существенный набор характеристик:*



Зачем нужны абстрактные типы данных

- Группировка родственных операций и данных
- Упрощение за счет построения более высокого уровня абстракции
- Возможность моделировать сущности реального мира
- Изоляция сложности. Упрощение за счет сокрытия деталей реализации
- Повышение удобочитаемости и понятности кода
- Ограничение влияния изменений
- Локальность изменений кода

Следствие: снижение сложности!



Абстрактные классы

- ❑ **Абстрактный класс** – это класс, который объявлен как абстрактный, при этом он может и не содержать абстрактные методы.
- ❑ Абстрактные классы не могут быть использованы для создания экземпляров, но они могут быть суперклассами.
- ❑ Можно сказать, что это класс, который определяет обобщенную форму, отдельно используемую всеми его подклассами (т.е. такой класс определяет только природу методов).

```
abstract class Vehicle {  
    protected void move() {  
        System.out.println("Vehicle move");  
    }  
}
```

Абстрактные классы

- ❑ **Абстрактный метод** – это метод, который объявлен без реализации (без тела, и заканчивается точкой с запятой), например:

```
public abstract void move();
```



Особенности:

- Если класс включает *абстрактные методы*, то он обязательно должен быть объявлен как *абстрактный*;
- Если абстрактный класс – это суперкласс, то подкласс обеспечивает реализацию всех абстрактных методов своего родительского класса;
- Если *подкласс не обеспечивает реализацию* всех абстрактных методов своего родительского класса, то он также должен быть объявлен как *абстрактный*.

Абстрактные классы

Пример 6:

```
abstract class Animal {  
    public abstract void move();  
}  
class Reptiles extends Animal {  
}
```

Подкласс должен обеспечить реализацию метода move() или быть абстрактным

```
abstract class Animal {  
    public abstract void move();  
}  
abstract class Reptiles extends Animal {           // OK  
}
```

Подкласс
объявлен
абстрактным

Абстрактные классы

Пример 7:

```
abstract class Animal {  
    public abstract void move();  
}
```

Абстрактный метод

```
abstract class Reptiles extends Animal {  
}
```

Абстрактный класс

```
class Boa extends Reptiles {  
    @Override  
    public void move(){  
        System.out.println("Boa move");  
    }  
}
```

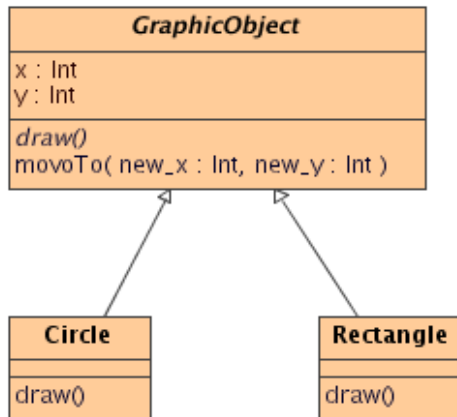
Реализация абстрактного метода

```
public class Main {  
    public static void main(String[] arg) {  
        Animal myAnimal = new Boa();  
        myAnimal.move();  
    }  
}
```

Ссылке на абстрактный класс
можно присваивать объект
подкласса, реализующего его
абстрактные методы

Абстрактные классы

Пример 8:



```
public abstract class GraphicObject {
    //абстрактный метод
    public abstract void draw();
    //движение центра фигуры
    public void moveTo(int x, int y) { }
}

class Circle extends GraphicObject {
    public void draw() {
        //рисует круг
    }
}

class Runner {
    public static void main(String[] args) {
        // можно объявить ссылку
        GraphicObject mng;
        //mng = new GraphicObject();
        //нельзя создать объект!
        mng = new Circle();
        mng.draw();
    }
}
```



Интерфейсы

Интерфейсы

- ❑ **Java интерфейс** – это тип, похожий на класс, который может содержать только константы, объявления методов и вложенные типы.
 - ✓ Начиная с Java 8, методы по умолчанию и статические методы могут иметь реализацию в определении интерфейса.

- ❑ **Интерфейс** – это аналог абстрактного класса и представляет собой множество требований, предъявляемых к соответствующему классу.
 - ✓ Объекты интерфейсов не могут быть созданы;
 - ✓ Интерфейсы могут быть только реализованы в классах;
 - ✓ Интерфейсы могут быть расширены в других интерфейсах.

Интерфейсы

ОПИСАНИЕ ИНТЕРФЕЙСА:

`{public}01 interface <идентификатор> { тело интерфейса }`

РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА:

`{<доступ>}02 class <идентификатор> implements
 <интерфейс> {, <интерфейс>}0N
 { <реализация класса> }`

- Интерфейсы образуют "контракт" между классом и внешним миром, и этот контракт настраивается во время сборки компилятором.
- Если ваш класс утверждает реализацию интерфейса, то все методы, определенные этим интерфейсом, должны появиться в его исходном коде для успешной компиляции.

Интерфейсы

Например,

<pre>public interface BehaviorAnimal { int SIGNAL = 5; void move(); void vocalize(); void preen(); }</pre>	<pre>public static final int SIGNAL = 5; public abstract void move(); public abstract void vocalize(); public abstract void preen();</pre>
Неявно	Явно

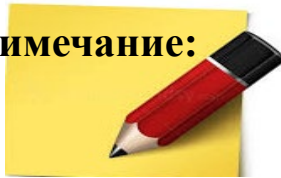


Особенности:

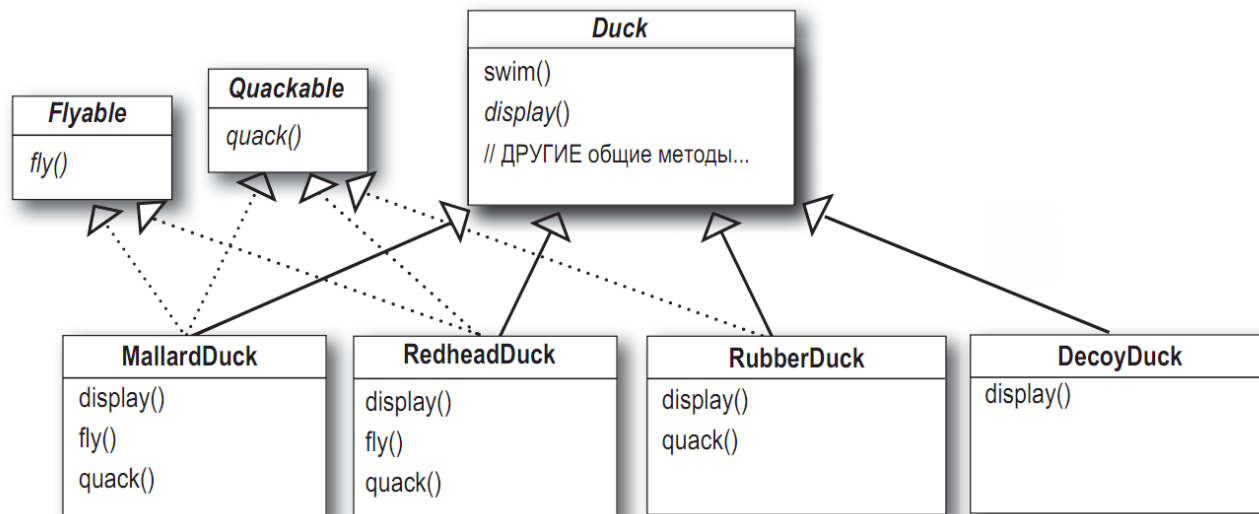
- Методы в интерфейсе (которые не были объявлены как **default** или **static**) неявно являются абстрактными и открытыми, поэтому модификатор **abstract** и **public** не нужно приводить в описании;
- Все константные величины, определенные в интерфейсе, неявно являются **public static final**.

Интерфейсы

Примечание:



- ❑ Класс, который реализует (расширяет) интерфейс, должен представить полную реализацию всех методов, объявленных в интерфейсе
 - ✓ Класс может содержать и собственные методы.
- ❑ Если класс расширяет интерфейс, но полностью не реализует его методы, то этот класс должен быть объявлен как *абстрактный*.
- ❑ При реализации методы интерфейса должны быть описаны со модификатором доступа **public**.



Интерфейсы

Пример 9:

```
interface Shape {  
    double PI = Math.PI;  
    double getSquare();  
}  
  
class Circle implements Shape {  
    double radius;  
    Circle(double x) { radius = x; }  
    public double getSquare() {  
        return PI * radius * radius;  
    }  
}  
  
public class Demo1 {  
    public static void main(String [] a) {  
        Shape object = new Circle(7.01);  
        System.out.print("Square: " + object.getSquare());  
    }  
}
```

Абстрактный метод

Реализация метода

Вывод в консоли:

Square: 22.022564502

Интерфейсы



- ❑ Объектную ссылку на интерфейсный тип можно связывать с любым объектом, чей класс реализует этот интерфейс:
 - ✓ однако через эту ссылку можно вызывать только методы, объявленные в интерфейсе;
 - ✓ собственные методы класса недоступны.

- ❑ Класс может реализовать более одного интерфейса
 - ✓ объекты могут иметь несколько типов: тип своего класса и типы всех интерфейсов, которые они реализуют.

Например,

```
class Car implements OperateCar, Parkable, Moveable {...}
```

```
// ...
```

```
OperateCar myOpCar = new Car();
```

```
Parkable myPark = new Car();
```

```
Moveable myMov = new Car();
```

В зависимости от ссылки
поведение объекта отражает
тот или иной контракт



Интерфейс может расширять любое число интерфейсов.

```
interface Interface1{/*...*/}
```

```
interface Interface2{/*...*/}
```

```
interface GroupedInterface extends Interface1,  
                                     Interface2 {
```

```
    //...
```

```
}
```



Класс может расширять только один класс и при этом реализовывать множество интерфейсов.

- Если в суперклассе и интерфейсе окажутся методы с одинаковой сигнатурой, то реализованный метод интерфейса перекроет видимость метода суперкласса;
- Для вызова метода суперкласса необходимо использовать обращение **super**.

Интерфейсы

Пример 10:

```
interface Call {
    int NUM = 10;
    void call();
}

class Base {
    int i = -5;
    public void call() {
        System.out.println("call() of class BASE:
                           i = " + i);
    }
}
```

Интерфейсы

Продолжение примера 10:

```
class Client extends Base implements Call {  
    public void call() {  
        System.out.println ("call() of class Client:  
                               NUM = " + NUM);  
        super.call();  
    }  
}  
  
public class Demo2 {  
    public static void main(String [] args) {  
        Call object = new Client();  
        object.call();  
    }  
}
```

Реализация метода интерфейса

Вызов метода суперкласса

Вывод в консоли:

```
call() of class Client: NUM = 10  
call() of class BASE: i = -5
```

Интерфейсы

- ❑ При расширении классом нескольких интерфейсов возможна ситуация когда два и более интерфейса имеют поля с одинаковыми именами (*но, возможно, разными значениями*) и методы с одинаковыми именами:
 - обратиться к полю напрямую без указания того, какой из интерфейсов имеется в виду нельзя;
 - к полю можно обратиться как к статическому элементу одного из интерфейсов;
 - или привести объект к одному из родительских интерфейсов;
 - создать отдельные реализации для одноимённых методов из разных интерфейсов в классе наследнике нельзя (*можно задавать лишь одну общую для всех реализацию этих методов*).

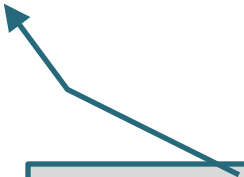
Пример 11:

```
public interface Interface1 {  
    int someField = 100;  
    String someMethod();  
}
```

Интерфейсы

Продолжение примера 11:

```
public interface Interface2 {  
    double someField = 200.5;  
    String someMethod();  
}  
  
public class SomeClass implements Interface1,  
                                   Interface2 {  
    public String someMethod() {  
        System.out.println("In interface2 = "  
                             + Interface2.someField);  
        return "It Works";  
    }  
}
```



Обращение к
полю конкретного
интерфейса

Интерфейсы

Продолжение примера 11:

```
public class Main {  
    public static void main(String[] args) {  
        SomeClass a = new SomeClass();  
        System.out.println( a.someMethod() );  
        System.out.println( ((Interface2)a).someMethod() );  
        System.out.println( ((Interface1)a).someField );  
        System.out.println( Interface1.someField );  
    }  
}
```

Одна и та же реализация

Обращение к одному и тому же полю

Вывод в консоли:

```
In interface2 = 200.5  
It Works  
In interface2 = 200.5  
It Works  
100  
100
```



Итоги

- ❑ Интерфейсы могут содержать только неизменяемые переменные и методы.
- ❑ Все методы интерфейса должны быть определены как **public** (т.к. по умолчанию они определены как общедоступные).
- ❑ Интерфейс, расширяющий другой интерфейс, не отвечает за реализацию методов из родительского интерфейса (т.к. интерфейсы не могут определять какую-либо реализацию).
- ❑ Интерфейс может расширять или класс может реализовать несколько других интерфейсов.
- ❑ Класс, реализующий интерфейс, должен определять методы с одинаковой видимостью (**public**).