



Объектно-ориентированные особенности языка Java

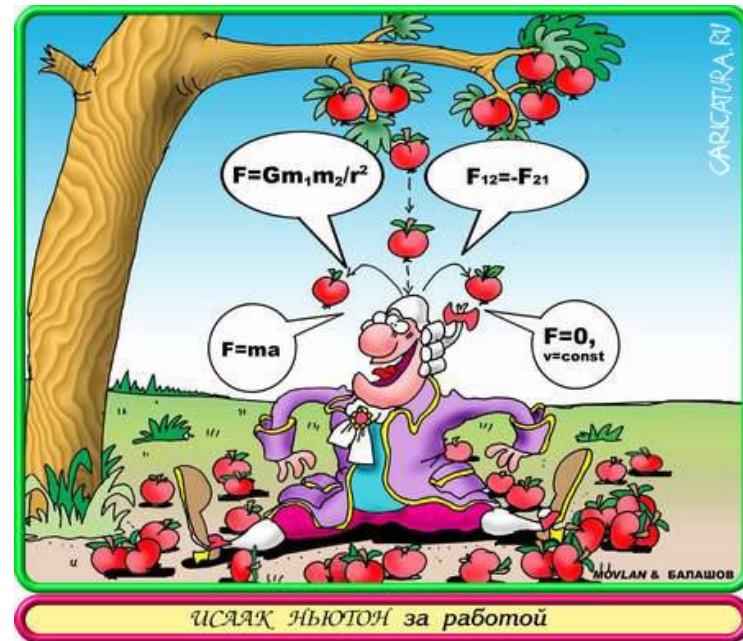
Java for autotesters

- ООП - методология программирования
 - Программа представляется совокупностью объектов
 - Каждый объект - экземпляр класса
 - Классы образуют иерархию наследования

- ООП использует в качестве базовых элементов объекты, а не алгоритмы

Абстракция

- Выделяйте только те факторы, которые нужны для решения задачи
- Отсекайте все лишнее

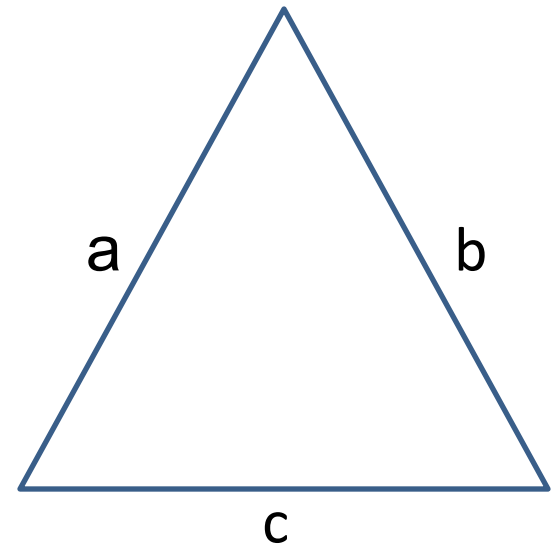
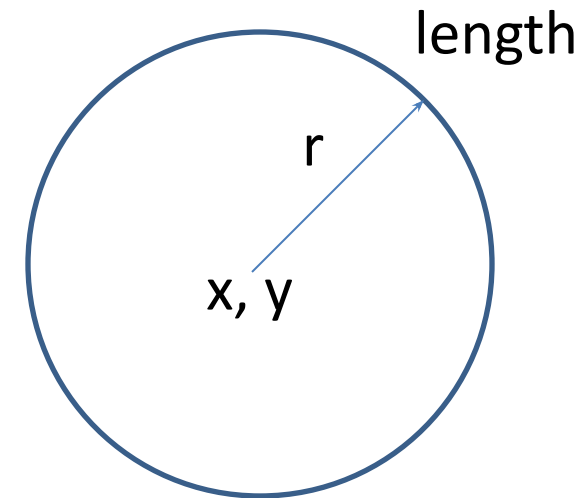


Инкапсуляция

- Скрытие реализации объекта
- У объекта есть внутреннее состояние, недоступное для изменения извне
- У объекта есть интерфейс, с помощью которого с объектом могут взаимодействовать внешние объекты

Инкапсуляция

- Пользователь (прикладной программист) не должен менять внутреннее состояние объекта – он не знает, как это делается!!!
- Поля и методы делятся на внутренние и интерфейсные.



Модификаторы доступа

	В классе	В пакете	В подклассах	Везде
<code>public</code>	+	+	+	+
<code>protected</code>	+	+	+	-
(по умолчанию)	+	+	-	-
<code>private</code>	+	-	-	-

Пример ограничения доступа

```
Point point = new Point();
```

```
point.setX(0.0);
```

```
point.setY(1.0);
```

```
point.x = 1.0; // Ошибка
```

```
System.out.println("(" + point.getX() + ", " +  
    point.getY() + ")");
```

Public
МЕТОДЫ
ДОСТУПНЫ

Ошибка!

- **Интерфейс** – это ссылочный тип, аналогичный классу, в котором могут быть *только* константы, сигнатуры методов и вложенные типы
 - Нельзя создать экземпляр
 - Нет тел методов
 - Можно только **реализовать** в классах или **расширить** в других интерфейсах

Интерфейсы как API

- API - Application Programming Interface
Интерфейс прикладного программирования
- API делается открытым, а его реализация может храниться в секрете
- Реализация может пересматриваться, но она должна по-прежнему реализовывать контракт, на который полагаются клиенты

Пример интерфейса

Модификатор
доступа

Ключевое
слово

Имя
интерфейса

```
public interface Relatable {
```

```
    int isLargerThan(Relatable other);  
}
```

Все методы
по умолчанию
public abstract

Декларация
метода
(Методов может
быть несколько)

Реализация интерфейса

```
public class Rectangle implements Relatable {
    private int width;
    private int height;

    public int getArea() {
        return width * height;
    }

    @Override
    public int isLargerThan(Relatable other) {
        Rectangle otherRect = (Rectangle) other;
        return this.getArea() - otherRect.getArea();
    }
}
```

Использование интерфейса как типа

- Интерфейс определяет новый ссылочный тип данных
- Переменная такого типа может ссылаться на объект любого класса, реализующего интерфейс
- Класс может реализовывать несколько интерфейсов

Изменение интерфейсов

- Есть интерфейс:

```
public interface DoIt {  
    void doSomething(int i, double x);  
    int doSomethingElse(String s);  
}
```

- Нужно добавить еще один метод
- Не очень хорошее решение:

```
public interface DoIt {  
    void doSomething(int i, double x);  
    int doSomethingElse(String s);  
    boolean didItWork(int i, double x, String s);  
}
```

- Более надежное решение – наследование интерфейсов:

```
public interface DoItPlus extends DoIt {  
    boolean didItWork(int i, double x, String s);  
}
```

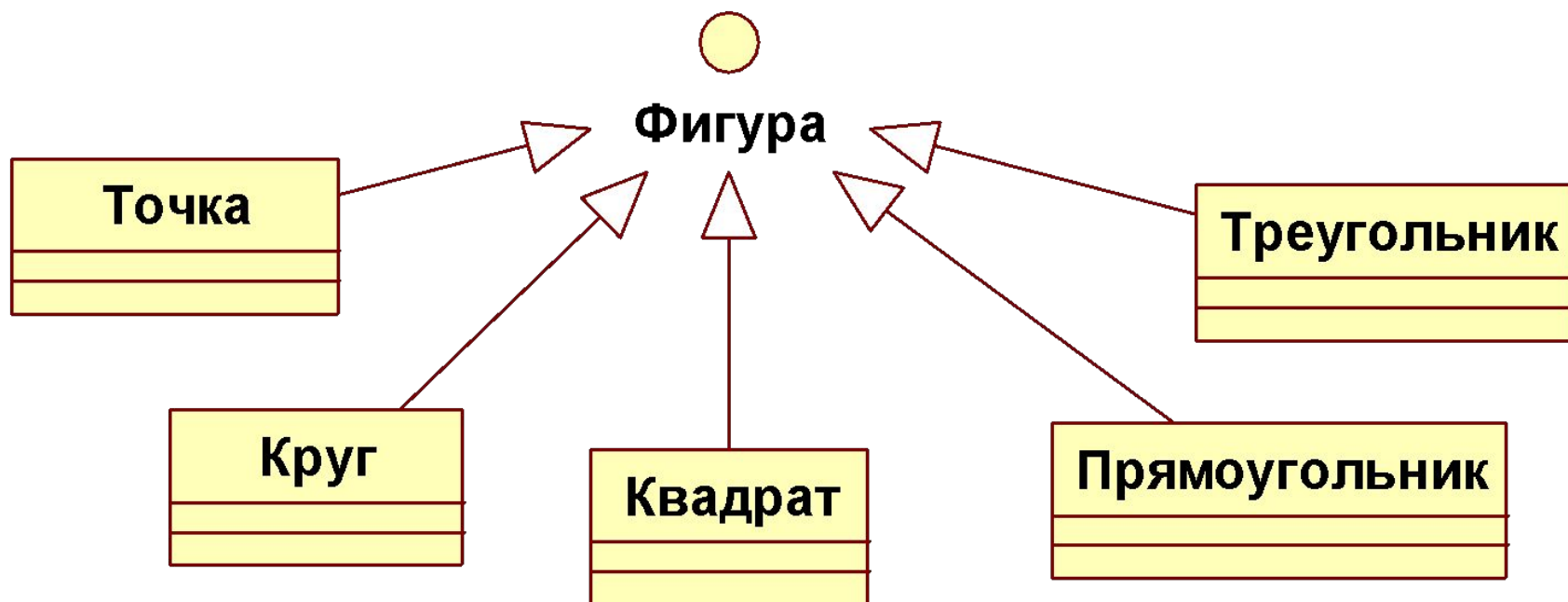
Наследование

- Механизм получения нового класса на основе уже существующего
 - Существующий класс можно дополнять или изменять, и получать новый класс
- Отношение IS-A («является»)
 - Наследник ЯВЛЯЕТСЯ разновидностью родителя
- Существующий – **суперкласс** (базовый, родительский)
- Новый – **подкласс** (производный, дочерний)

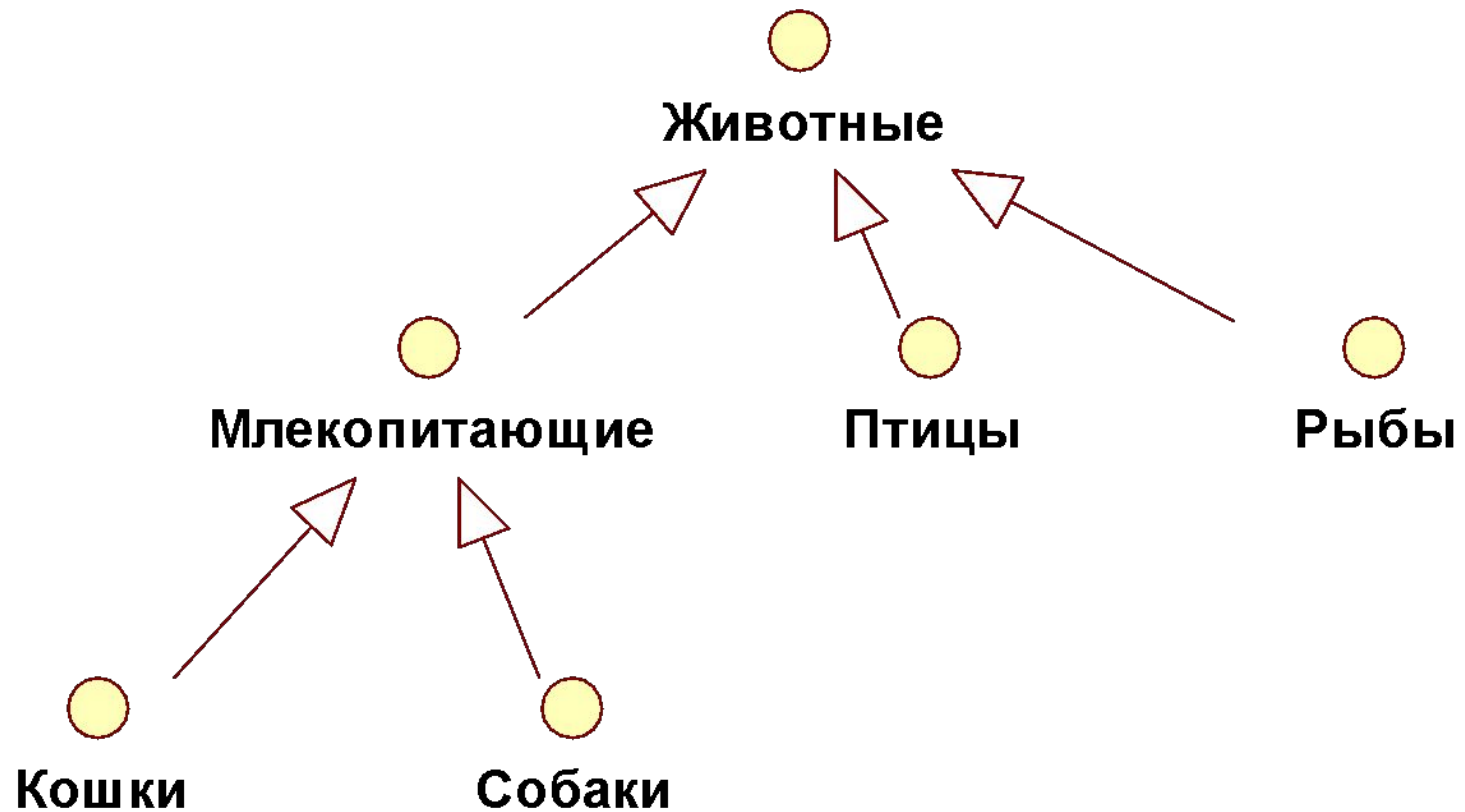
Наследование

- **Одиночное**
 - У класса есть только один предок
- **Множественное**
 - У класса может быть несколько предков
 - Java не поддерживает множественное наследование
- В Java у класса может быть только один суперкласс, но класс может реализовывать несколько интерфейсов

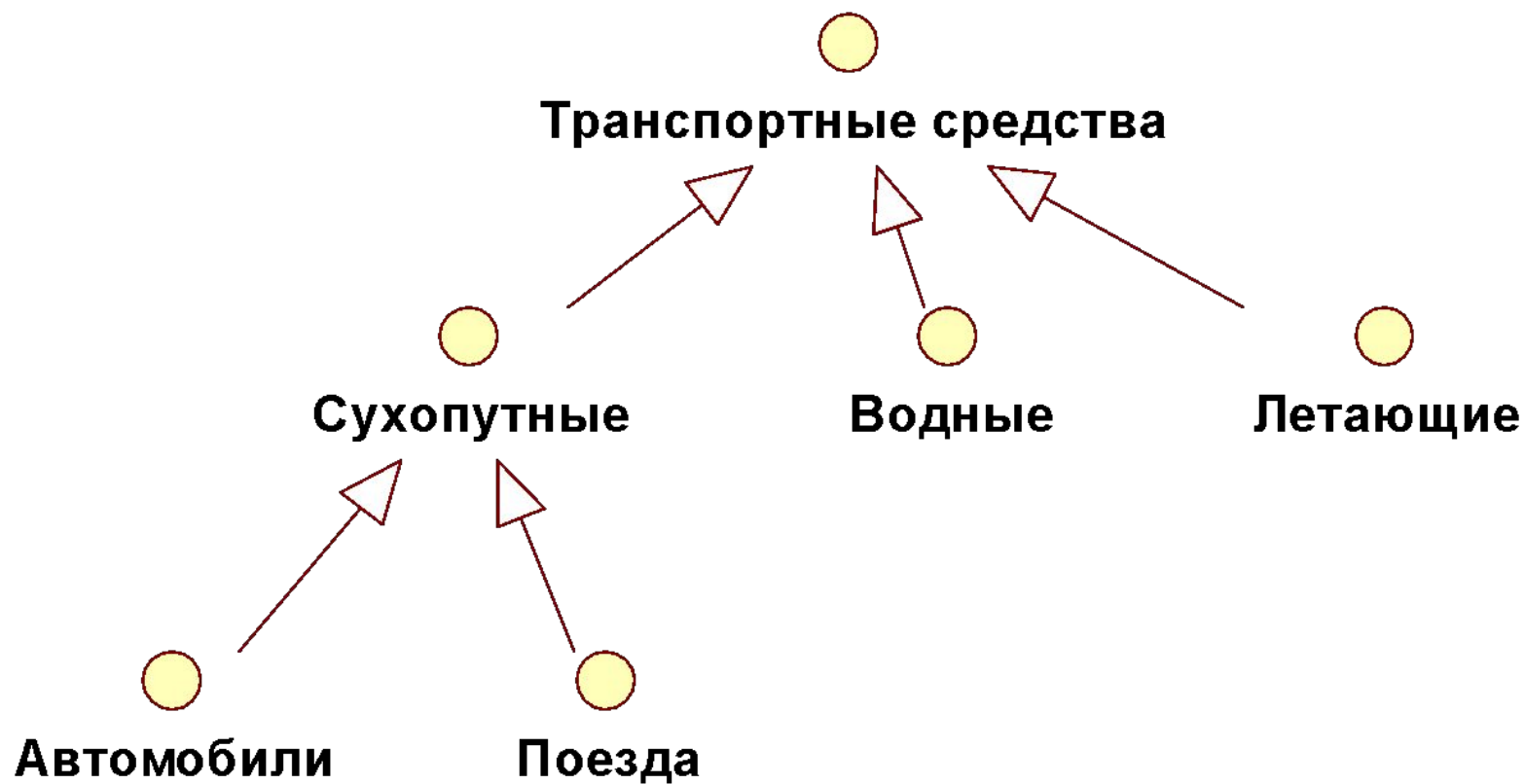
Примеры наследования



Примеры наследования



Примеры наследования



Наследование

- За исключением `Object`, у каждого класса есть один непосредственный суперкласс
- Класс `Object` находится во главе иерархии
- Интерфейсы не являются частью иерархии классов
- Наследуются все компоненты с модификаторами `public` и `protected` (поля, методы и вложенные классы)
- Конструкторы не наследуются подклассами, но могут быть вызваны в подклассе

Методы класса Object

```
public final native Class<?> getClass();
```

```
public native int hashCode();
```

```
public boolean equals(Object obj)
```

```
protected native Object clone() throws CloneNotSupportedException;
```

```
public String toString()
```

```
public final native void notify();
```

```
public final native void notifyAll();
```

```
public final native void wait(long timeout)
```

```
    throws InterruptedException;
```

```
public final void wait(long timeout, int nanos)
```

```
    throws InterruptedException
```

```
public final void wait() throws InterruptedException
```

```
protected void finalize() throws Throwable
```

Типы объекта

- В Java класс может наследоваться только от одного класса, но может реализовывать более одного интерфейса
- У объекта может быть несколько типов:
 - тип собственного класса
 - типы всех суперклассов
 - типы всех реализованных им интерфейсов

Приведение типа объектов

- **Приведение типа** показывает использование объекта одного типа вместо другого типа
- **Неявное** приведение типа

```
String s = "some text";  
Object o = s;
```

- **Явное** приведение типа

```
String t = (String) o;
```

Что можно делать в подклассах

- Поля
 - Унаследованные использовать как есть
 - Объявить новые
 - Скрыть поле (hide) (не рекомендуется)
- Методы
 - Унаследованные использовать как есть
 - Объявить новые
 - Переопределить (override)
 - Скрыть статический (hide)
- Конструкторы
 - В конструкторе подкласса вызвать конструктор суперкласса

Переопределение и сокрытие методов

- Метод с сигнатурой, совпадающей с сигнатурой метода суперкласса

	Нестатический метод суперкласса	Статический метод суперкласса
Нестатический метод подкласса	Переопределяет	Ошибка компиляции
Статический метод подкласса	Ошибка компиляции	Скрывает

private в суперклассе

- Не наследуются у своего суперкласса
- Доступны через унаследованные `public` или `protected` методы
- У вложенного класса есть доступ к `private` полям и методам своего внешнего класса
- `private` компоненты неявно доступны через унаследованный вложенный класс

private в суперклассе

```
public class Superclass {  
    private int value;  
    public int getValue() {  
        return value;  
    }  
}
```

```
public class Subclass extends Superclass {  
    public void method() {  
        System.out.println(value);  
        System.out.println(getValue());  
    }  
}
```

ОШИБКА!

Ключевые слова super

- Доступ к членам суперкласса


```
super.superclassMethod();
```

- Вызов конструкторов суперкласса

```
super();
```

```
super(parameters);
```

Конструктор без параметров
обычно нет смысла вызывать явно



- В конструкторе вызов super – всегда первый оператор

- Final классы
 - От `final` класса (неизменяемого) невозможно наследоваться
 - Например, от класса `String`
- Final методы
 - Нельзя переопределить в подклассах
 - Методы, вызываемые из конструкторов должны обычно быть объявлены как `final`
 - Необходимы, если реализацию метода не должна изменяться и важна для сохранения непротиворечивого состояния объекта

Абстрактные методы и классы

- **Абстрактный класс** – это класс, объявленный с модификатором `abstract`
 - могут быть, а могут и не быть абстрактные методы
 - нельзя создать экземпляр, но можно создать подклассы
- **Абстрактный метод** – это метод, объявленный без реализации:

```
abstract void moveTo(double deltaX,  
                    double deltaY);
```

Интерфейсы vs Абстрактные классы

- Абстрактные классы могут содержать поля (которые не `static` и не `final`)
- Абстрактные классы могут содержать реализации методов
- Разные интерфейсы могут независимо реализовываться классами в разных местах иерархии
- Абстрактные классы в большинстве случаев наследуются для использования части реализации

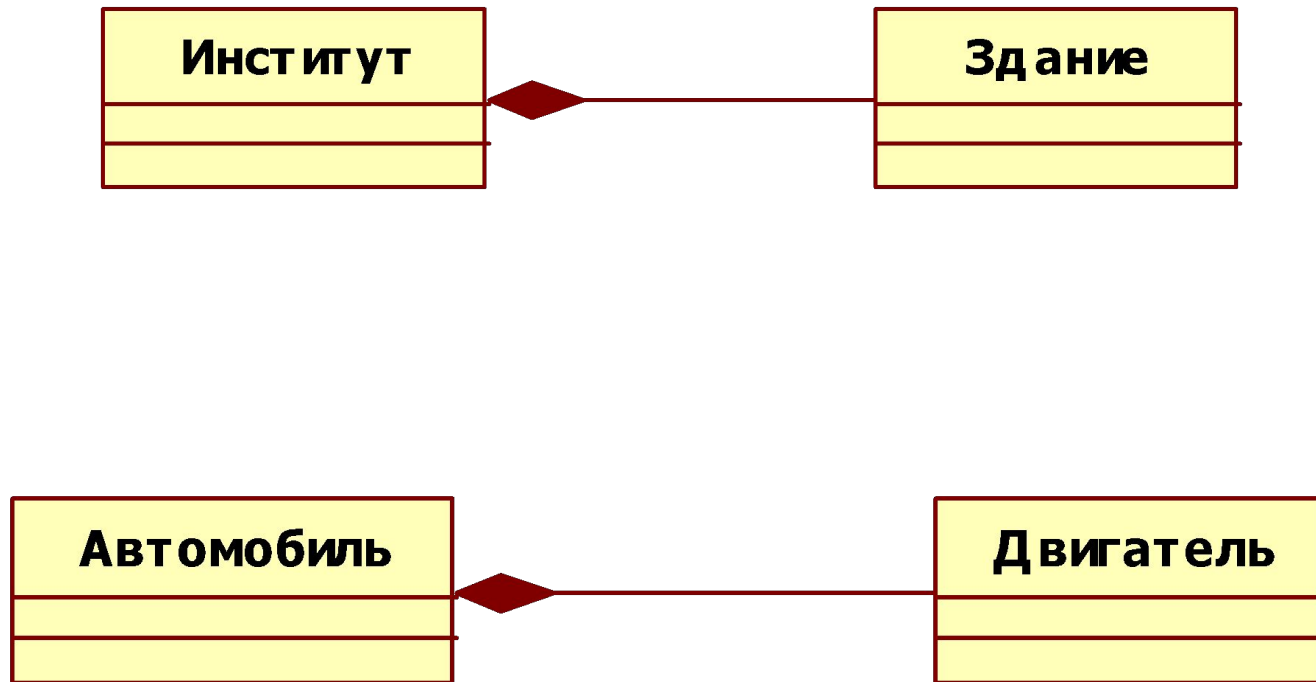
Абстрактные классы

- Может реализовывать интерфейс
 - При этом не обязательно реализовывать все методы интерфейса
- Может содержать статические компоненты
 - Их можно использовать с именем класса – как и в случае любого другого класса

- Существуют различные варианты включения объектов
 - Композиция
 - Агрегация
 - Ассоциация (Осведомленность)

- Самый строгий вид включения
- Времена жизни внешнего и включаемого объекта совпадают
- Включаемый объект может существовать только как часть внешнего
- Отношение «целое-часть» (HAS-A)

Композиция – пример



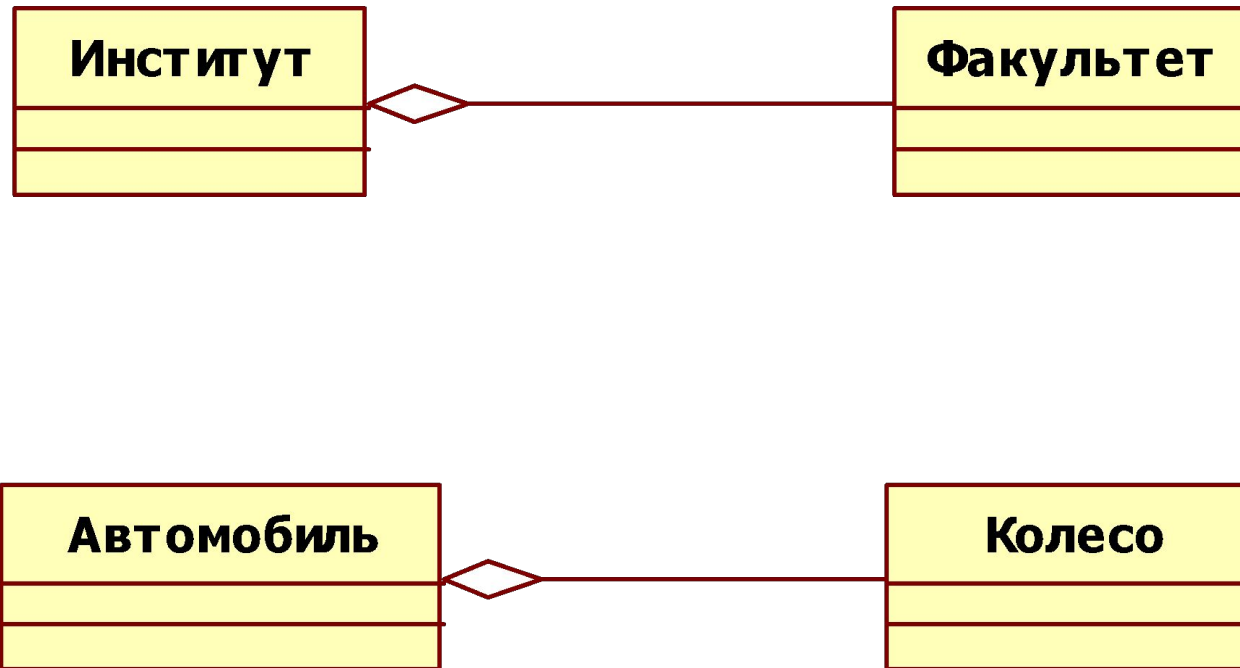
Композиция - пример

```
public class Car {  
  
    private Engine engine;  
  
    public Car() {  
        this.engine = new Engine();  
    }  
  
    public Car(Engine engine) {  
        this.engine = engine;  
    }  
  
    public Engine getEngine() {  
        return engine;  
    }  
}
```

Время жизни
объектов Car и Engine
совпадает

- Отношение «целое-часть» (HAS-A)
- Но объекты могут существовать независимо
- Включаемый объект может существовать и без внешнего

Агрегация – Пример



Агрегация - пример

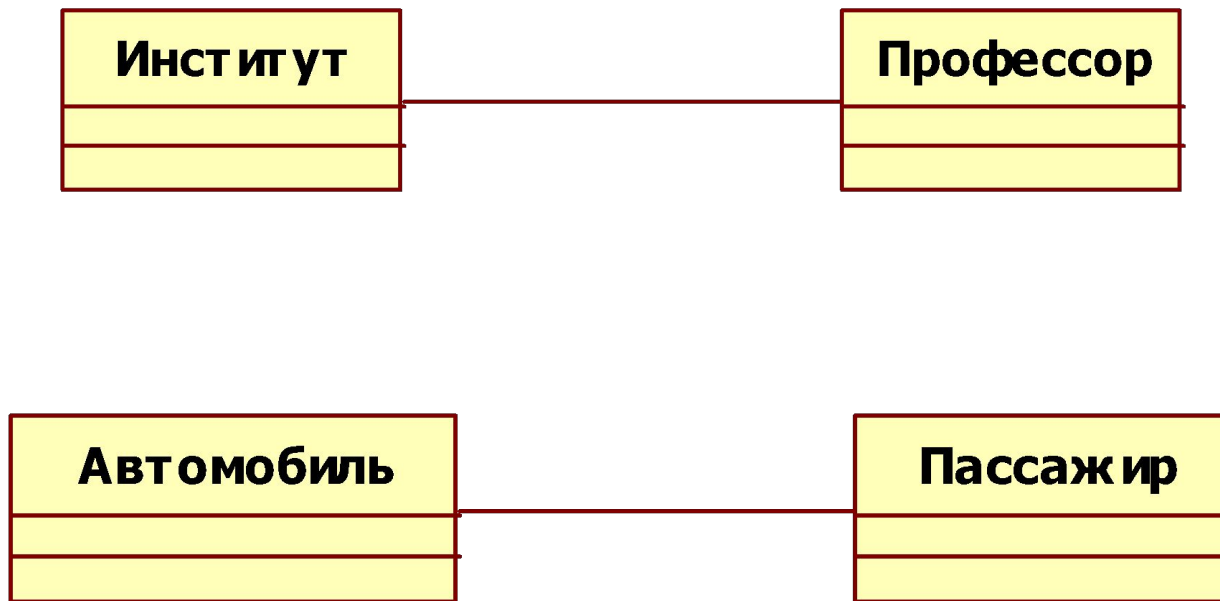
```
public class Car {  
  
    private Wheel[] wheels;  
  
    public Car(Wheel[] wheels) {  
        this.wheels = wheels;  
    }  
  
    public Wheel[] getWheels() {  
        return wheels;  
    }  
  
    public void setWheels(Wheel[] wheels) {  
        this.wheels = wheels;  
    }  
}
```

Ссылка wheels не
должна быть null

Ассоциация

- Самый слабый вид включения
- Один объект знает о существовании другого
- Осведомленность может быть взаимной
- Времена жизни объектов никак не связаны

Ассоциация – пример



Ассоциация - пример

```
public class Car {  
  
    private Passenger passenger;  
  
    public Car(Passenger passenger) {  
        this.passenger = passenger;  
    }  
  
    public Passenger getPassenger() {  
        return passenger;  
    }  
  
    public void setPassenger(Passenger passenger) {  
        this.passenger = passenger;  
    }  
}
```

Ссылка passenger
МОЖЕТ БЫТЬ null

Полиморфизм

- **Полиморфизм** – возможность объектов с одинаковой спецификацией иметь различную реализацию
- «Один интерфейс, множество реализаций»



- Зачем они могут понадобиться?
 - Логическая группировка классов
 - Увеличение инкапсуляции
 - Более легкий для чтения и поддержки код

Статические вложенные классы

```
public class OuterClass {  
    ...  
    public static class NestedClass {  
        ...  
    }  
}
```

```
OuterClass.NestedClass nested  
    = new OuterClass.NestedClass();
```

Внутренние классы

```
public class OuterClass {  
    ...  
    public class InnerClass {  
        ...  
    }  
}
```

```
OuterClass.InnerClass inner  
    = new OuterClass().new InnerClass();
```

Анонимные классы

```
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        System.exit(0);  
    }  
});
```

Перечисления (enum)

- *Перечисление* – это тип, значения которого ограничены фиксированным множеством констант

```
public enum Gender {  
    MALE,  
    FEMALE;  
}
```

```
public enum Season {  
    WINTER,  
    SPRING,  
    SUMMER,  
    FALL  
}
```

Перечисления могут содержать поля и методы

```
public enum Planet {
    MERCURY(3.303e+23, 2.4397e6),
    VENUS  (4.869e+24, 6.0518e6),
    EARTH  (5.976e+24, 6.37814e6),
    ...
    private final double mass;
    private final double radius;
    private Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    public double getMass() {
        return mass;
    }
    public double getSurfaceGravity() {
        return G * mass / (radius * radius);
    }
}
```


Пример использования enum

```
public enum Direction {
    NORTH(0, 1),
    EAST(1, 0),
    SOUTH(0, -1),
    WEST(-1, 0);
    private final int x;
    private final int y;
    private Direction(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
...
public void move(Direction direction) {
    currentX += direction.getX();
    currentY += direction.getY();
}
```

Перечисления (enum)

- Все перечисления неявно наследуются от `java.lang.Enum`
- Все константы перечисления неявно имеют модификаторы `public static final`
- Нельзя создать экземпляр перечисления с помощью оператора `new`
- Нельзя наследоваться от перечисления

Перечисления (enum)

- Некоторые нестатические методы перечисления:
 - `ordinal()` - номер элемента перечисления (номера начинаются с 0)
 - `compareTo()` - элементы перечисления можно сравнивать
- Полезные статические методы перечислений:
 - `values()` – возвращает массив из всех констант перечисления
 - `valueOf(String name)` – ищет константу с заданным именем

- *Аннотации* содержат данные, которые не являются частью программы
- Применения:
 - Информация для компилятора
 - Обработка времени компиляции и размещения
 - Обработка времени выполнения

Предопределенные аннотации

- **@Deprecated**

```
/**  
 * @deprecated explanation of why it was deprecated  
 */  
@Deprecated  
static void deprecatedMethod() {  
}
```

- **@Override**

```
@Override  
int overriddenMethod() {  
}
```

- **@SuppressWarnings**

```
@SuppressWarnings("deprecation")  
void useDeprecatedMethod() {  
    deprecatedMethod();  
}
```

Спасибо за внимание!

Контактная информация:

Денис Мурашев

Инструктор

EPAM Systems, Inc.

Адрес: Саратов, Рахова, 181

Email: Denis_Murashev@epam.com

<http://www.epam.com>