



Объектно-ориентированное программирование

1. Абстракция
 2. Инкапсуляция
 3. Наследование
 4. Полиморфизм
-



Абстракция

- Представление важных аспектов предметной области в виде совокупности взаимодействующих друг с другом объектов



Абстракция

- **Цель:** представить предметную область в виде совокупности классов/объектов, выделив у них важные свойства (атрибуты, параметры, поля) и определив методы (операции, функции) для работы с ними.
-



Инкапсуляция

- Объединение данных и кода, относящихся к объекту
- Скрытие реализации
- Предоставление пользователю интерфейса для работы с объектом



Инкапсуляция

- **Цель:** Повысить надежность кода, скрыв реализацию сложных бизнес-правил и элементы, доступ к которым может быть небезопасным, предоставив пользователю простой и безопасный интерфейс.



Наследование

- Более общие вещи объявляются в родительском (базовом) классе
 - Более конкретные вещи уточняются в классе-наследнике
 - В Java есть только единичное наследование
 - В Java есть интерфейсы
-



Наследование

Цели:

- обеспечение требуемого уровня абстракции (абстрактные предки, конкретные потомки);
 - повторное использование кода;
 - основа для полиморфизма.
-



Полиморфизм

- возможность объектов с одинаковой спецификацией иметь различную реализацию;
- возможность обращаться к объектам-наследникам, используя ссылку на базовый класс или интерфейс



Полиморфизм

Цели:

- однотипная работа с объектами различных
ТИПОВ
 - работа с объектами, конкретные
тип/реализация которых на этапе
КОМПИЛЯЦИИ еще неизвестны
-



Классы в Java

Объявление класса

```
[public] class <ИмяКласса>  
[extends <ИмяБазовогоКласса>]  
[implements <ИмяИнтерфейса1>[, <ИмяИнтерфейса2> [, ...]]] {  
    <объявление полей, конструкторов и методов класса>  
}
```

Объявление поля

```
[<модификаторы>] <тип> <имяПоля> [=<значение_по_умолчанию>] ;
```

Объявление метода

```
[<модификаторы>] <тип возвращаемых данных> <имяМетода> (  
    [список аргументов])  
    [throws <список классов исключительных ситуаций>] {  
    <блок программного кода>  
}
```



Пример 1. Абстракция

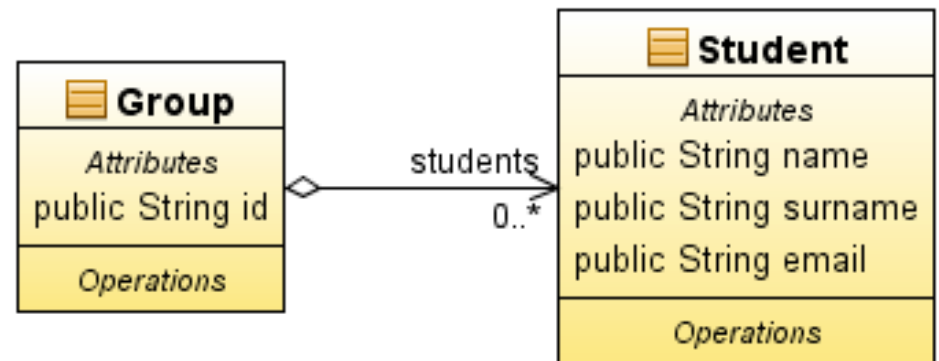
Предметная область: Студенты учатся в группе

Student.java:

```
public class Student {  
    public String name;  
    public String surname;  
    public String email;  
}
```

Group.java:

```
public class Group {  
    public String id;  
    public Student[] students;  
}
```





Пример 2. Инкапсуляция

Предметная область: часы

Атрибуты: часы, минуты

Операции:

- выставить заданное время
- узнать время
- увеличить время на 1 минуту



Бизнес правила:

- часы принимают значение 0..23
- минуты принимают значение 0..59
- если при увеличении времени на 1 минуту получилось 60, выставить значение минуты в 0 и увеличить количество часов на 1
- если при увеличении часов получилось 24, выставить значение часов в 0




```
public class Clock {
```

```
    private int hours;  
    private int minutes;
```

```
    public int getHours() {  
        return hours;  
    }
```

```
    public int getMinutes() {  
        return minutes;  
    }
```

```
// Про public, private будет рассказано позже
```

 Clock
<i>Attributes</i> private int hours private int minutes
<i>Operations</i> public int getHours() public int getMinutes() public void setHours(int hours) public void setMinutes(int minutes) public void increaseMinutes() public void increaseMinutesTryInvestigateHowItsWork()



```
public void setHours(int hours) {  
    if (hours >= 0 && hours <= 23) {  
        this.hours = hours;  
    } else {  
        throw new IllegalArgumentException("hours=" +  
hours);  
    }  
}
```

```
public void setMinutes(int minutes) {  
    if (minutes >= 0 && minutes <= 59) {  
        this.minutes = minutes;  
    } else {  
        throw new IllegalArgumentException("minutes=" +  
minutes);  
    }  
}
```

```
// Про this и new будет рассказано позже  
// Про throw и Exception будут через несколько лекций
```



```
public void increaseMinutes() {  
    minutes++;  
    if (minutes >= 60) {  
        minutes = 0;  
        hours++;  
        if (hours >= 24) {  
            hours = 0;  
        }  
    }  
}
```

```
public void increaseMinutesTryInvestigateHowItsWork() {  
    minutes++;  
    hours += minutes / 60;  
    minutes %= 60;  
    hours %= 24;  
}  
}
```

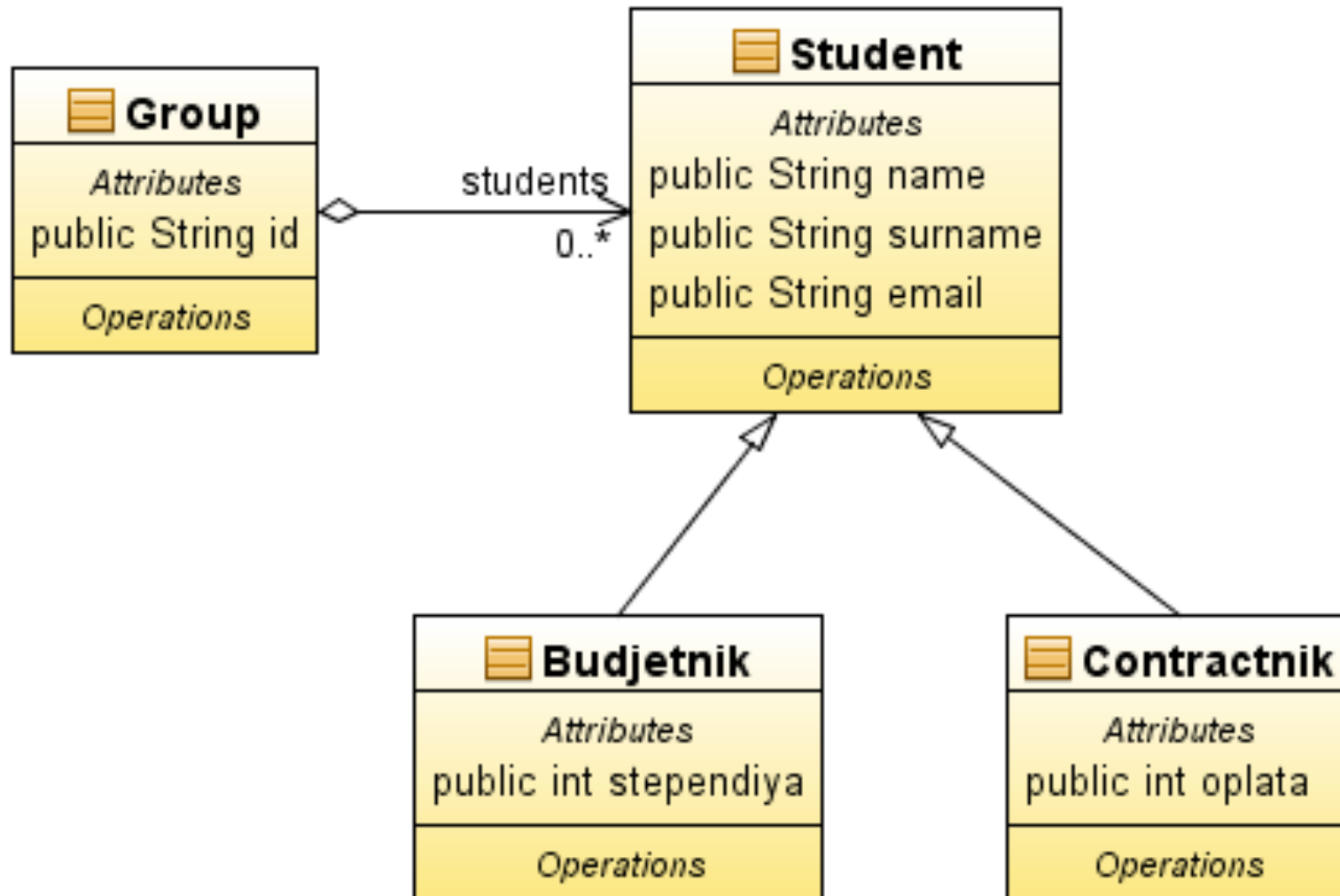



Пример 3. Наследование

```
// Взяли готовый класс  
// и расширили его в соответствии со своими потребностями
```

```
public class Budjetnik extends Student {  
    public int stependiya;  
}
```

```
public class Contractnik extends Student {  
    public int oplata;  
}
```





IS-A, HAS-A

IS-A: Бюджетник **является** Студентом

HAS-A: Группа **содержит** Студентов



Создание объектов

*<Тип> <ссылка> = **new** <ИмяКласса>([<параметры_конструктора>]);*
<Тип> = <ИмяКласса> | <ИмяБазовогоКласса> | <ИмяИнтерфейса>

Пример.

```
Clock c = new Clock();  
System.out.println(c.getHours()+" "+c.getMinutes());  
c.setHours(12);  
c.setMinutes(19);  
c.increaseMinutes();  
System.out.println(c.getHours()+" "+c.getMinutes());  
}
```



Создание объектов

- Новые экземпляры объектов создаются в «куче» (heap) оператором **new**
- Копирование ссылки (присвоение, передача в качестве параметра) не приводит к созданию нового объекта!!! (*см. пример*)



```
// Почувствуйте разницу :)
```

```
public class Main {  
    public static void main(String[] args) {  
        int a1;  
        a1=1;  
        int a2=a1;  
        a2=2;  
        System.out.println(a1 + " " + a2);  
  
        Clock c1=new Clock();  
        c1.setHours(1);  
        Clock c2=c1;  
        c2.setHours(2);  
        System.out.println(c1.getHours() + " " +  
c2.getHours());  
    }  
}
```



Инициализация объектов

При создании нового объекта его поля, для которых в описании класса не были указаны значения по-умолчанию, принимают такие значения:

byte, char, short, int, long, float, double	0
boolean	false
ссылки	null

* Примечание: это справедливо только для полей объектов и классов. Локальные переменные автоматически не инициализируются!!!

```
int a;  
System.out.println(a); // ОШИБКА!!!
```

Конструкторы

- Конструкторы позволяют совместить создание и инициализацию объекта
 - Имя конструктора совпадает с именем класса (с учетом регистра)
 - В описании конструктора отсутствует тип возвращаемой величины
 - Можно объявлять несколько конструкторов, отличающихся количеством или типом параметров
 - Если у класса нет ни одного конструктора, компилятор создает конструктор по-умолчанию без параметров. Такой конструктор не делает ничего, кроме вызова конструктора без параметров базового класса
 - Если у класса явно объявлен хотя бы один конструктор, конструктор по-умолчанию автоматически не создается
-



Пример:

```
//Clock.java
public class Clock {

    private int hours;
    private int minutes;

    public Clock(int hours) {
        this.hours = hours;
    }

    public Clock(int hours, int
minutes) {
        this.hours = hours;
        this.minutes = minutes;
    }

    ...
}
```

```
//Main.java
...
//OK
Clock c1 = new Clock(12);

//OK
Clock c2 = new Clock(12, 20);

//ОШИБКА!!!
Clock c = new Clock();
// Т.к. мы явно объявили
// конструкторы, конструктора
// по умолчанию больше нет!

...
```



Ссылка **this**

Данное ключевое слово используется в качестве ссылки на объект, в котором в данный момент происходит выполнение программного кода.

Чаще всего применяется для:

- передачи другому объекту ссылку на себя (для создания связей между объектами)
- для обращения к свойствам объекта, если их область видимости перекрыта другими переменными с такими же именами



Примеры использования **this**

```
public class Student {  
    String name;  
    ...
```

```
    public Student(String name) {  
        this.name = name;  
    }
```

```
    ...
```

```
    void addToGroup(Group group) {  
        group.addStudent(this);  
    }
```

```
    ...
```

```
}
```



Ссылка **super**

Данное ключевое слово используется в качестве ссылки на объект суперкласса (базового класса) объекта, в котором в данный момент происходит выполнение программного кода.

Чаще всего применяется для:

- вызова метода базового класса, который был переопределен в потомке
- вызова конструктора базового класса из конструктора потомка. При этом вызов конструктора базового класса должен быть первым оператором в конструкторе. Если **super** не используется, происходит вызов конструктора без параметров базового класса (если его нет – ошибка компиляции).



Примеры использования **super**

Пример 1

```
public class Employee {  
  
    String name;  
  
    public Employee(String name) {  
        this.name = name;  
    }  
  
    public String toString() {  
        return "Name:" + name;  
    }  
  
}
```



Пример 1 (Продолжение)

```
public class Manager extends Employee {
```

```
    String departament;
```

```
    public Manager(String name, String departament) {  
        super(name);  
        this.departament = departament;  
    }
```

```
    public String toString() {  
        return super.toString() + " is manager of " +  
            departament;  
    }
```

```
}
```



Примеры использования **super**

Пример 2

```
public class Base {  
    private int a;
```

```
    public Base(int a) {  
        this.a=a;  
    }  
}
```



Пример 2 (Продолжение)

```
public class MyClass extends Base{  
    public MyClass() {  
        super(veryComplexAlgorithm());  
        // other inits here  
        // ...  
    }  
}
```

```
    public static int veryComplexAlgorithm() {  
        // very complex algorithm here  
        // ...  
        return 42;  
    }  
}
```




Перегрузка методов

(**Overload**, не путать с **Override** !!!)

Допускается объявлять несколько методов или конструкторов с одинаковыми именами, но разными параметрами

- Количество или типы аргументов должны отличаться
- Возвращаемый тип может отличаться

Сигнатура метода — имя и количество/типы параметров.

Двух методов с одинаковыми сигнатурами в одном классе/интерфейсе быть не должно.



Задания

1. Написать класс Коты. Предусмотреть наличие 4 полей, 2 конструкторов и 2 методов. Предусмотреть инкапсуляцию. Создать несколько экземпляров этого класса и вывести информацию про них на экран, предусмотрев соответствующий метод.
 2. Создать класс-обертку для работы с одномерным массивом. Предусмотреть инкапсуляцию и базовые операции над массивом (сортировка, удаление элемента, добавление элемента, поиск максимального и т.д.).
 3. Написать систему классов, реализующие фигуры на плоскости (точка, отрезок, квадрат, прямоугольник, треугольник). Предусмотреть инкапсуляцию. Предусмотреть несколько конструкторов и методов определения размера. Использовать для связи отношение Has-a.
 4. Поменять предыдущую систему так, чтоб она была связана еще и отношением наследования (is-a). (д/з)
-



Пакеты (**package**)

- Для исключения возможных конфликтов названий классов разных производителей, классы размещаются в пакетах
- Пакеты также могут размещаться в пакетах, образуя иерархию
- В качестве пакета верхнего уровня рекомендуется использовать доменное имя своей организации, составляющие которого записаны в обратном порядке. Это с высокой вероятностью обеспечит отсутствие 2х классов с одинаковыми названиями в масштабах всей планеты:

```
com.pupkin.vasya.mypacket.MyClass;  
com.smith.john.mypacket.MyClass;
```



- Название пакета указывается в самом начале файла

Файл edu/acts/hr/Employee.java

```
// File containing source code for class Mechanism
```

```
package edu.acts.hr;
```

```
public class Employee {
```

```
//...
```

```
}
```

- На уровне файловой системы пакеты представлены папками, в которых хранятся файлы .java (перед компиляцией) и .class (после компиляции).



Импортирование пакетов (**import**)

- Чтобы не писать каждый раз полное имя класса (`edu.acts.hr.Employee`) допускается с помощью **import** указать классы, к которым можно будет обращаться по их имени без указания пакета, в котором они находятся

```
package mypackage;  
import edu.acts.hr.Employee;  
...  
public class MyClass {  
...  
    Employee e = new Employee();  
    // Вместо edu.acts.hr.Employee e = new edu.acts.hr.Employee();  
}
```



- **import** следует располагать после **package**, но перед **class**
- можно импортировать все классы из пакета (при этом классы из вложенных пакетов автоматически не импортируются):

```
import java.io.*;  
import java.net.*;
```

- количество импортированных классов никак не влияет на размер или быстродействие кода, поскольку импортирование является просто разрешением имен и осуществляется на этапе компиляции