

# Язык Java

Осенний семестр 2023, магистратура, 1 курс

Лекция 3: Объектно-ориентированное программирование (ООП)

# Содержание

1. Принципы ООП
2. Классы и объекты
3. Инкапсуляция
4. Наследование и композиция
5. Полиформизм
6. Ключевое слово `static`
7. Ключевое слово `final`
8. Перечисления `enum`
9. Абстрактные классы и интерфейсы

# Объектно-ориентированное программирование (ООП)

- Представление программы в виде совокупности *объектов*
- Объект содержит:
  - данные (*поля*): переменные примитивных типов, другие объекты, ...
  - программный код (*методы*)
- ООП в Java: *класс-ориентированное программирование*
  - Использование *классов* (шаблонов) для создания объектов

# Принципы ООП

- **Инкапсуляция**

- Ограничение доступа одних компонентов программы к другим
- Пример: *приватные* поля и методы объекты недоступны другим объектам

- **Наследование**

- Создание новых классов (*подклассов*) на основе существующих (*суперклассов*)
- Расширение функциональности: добавление новых полей и методов

- **Полиморфизм**

- Способность функции обрабатывать данные разных типов (суперклассов и подклассов)

# Определение класса

## Базовая структура класса

```
class Student {  
    // поля  
  
    // конструкторы (методы для создания объектов)  
  
    // методы  
}
```

Класс может иметь модификатор доступа `public` , но об этом позже

## Пример класса

```
class Student {  
    String name; // поле  
  
    Student(String name) { this.name = name; } // конструктор  
  
    String getName() { return name; } // метод (getter)  
    void setName(String name) { this.name = name; }  
}
```

`this` указывает на объект класса (использовать необязательно, если нет конфликта имен с локальными переменными)

Пример создания и использования объекта:

```
Student student = new Student("Дейенерис");  
String name = student.getName();           // "Дейенерис"
```

# Конструктор по умолчанию

- Конструктор по умолчанию (отсутствует)

```
class Student {  
    int age;  
}
```

**Вопрос:** Что будет записано в переменную `result` ?

```
Student student = new Student();  
int result = student.age;
```

# Перегрузка конструктора

Класс может иметь любое количество конструкторов, которые различаются в списке параметров

```
class Student {  
    int age;  
    Student(int age) { // конструктор 1  
        this.age = age;  
    }  
    Student(long age) { // конструктор 2  
        this((int)age); // вызов конструктора 1  
    }  
}
```

```
Student intStudent = new Student(10); // конструктор 1  
Student longStudent = new Student(10L); // конструктор 2
```

**Вопрос:** Какой конструктор будет вызван при `new Student((short)10)` ?



# Инициализация полей объекта

```
class Student {  
    String firstName = "Дейенерис"; // инициализация при объявлении  
    String lastName;  
    String patronym;  
  
    { lastName = "Таргариен"; } // инициализатор объекта  
  
    Student(){  
        this.patronym = "Эйрисовна"; // инициализация в конструкторе  
    }  
}
```

Инициализатор может быть определен и при создании объекта.

```
Student student1 = new Student(); // Таргариен Дейенерис Эйрисовна  
Student student2 = new Student(){ // Дрого Дейенерис Эйрисовна  
    lastName = "Дрого";           // инициализатор объекта  
};
```

## Инициализация полей: упражнение

```
class ValueContainer {  
    String value = "a";  
    ValueContainer() { value += "c"; }  
    { value += "b"; }  
}
```

Что будет записано в переменную `result` ?

```
String result = new ValueContainer(){  
    value += "d";  
}.value;
```

# Инкапсуляция: доступ к классам

Модификаторы доступа:

- `public` : доступен из абсолютно всех классов
- по умолчанию (без ключевого слова): доступен только внутри своего пакета

```
package opensource;  
  
public class A {} // доступен везде  
class B {}        // доступен только внутри пакета opensource
```

```
package myproject;  
  
import opensource.A;  
import opensource.B; // ошибка компиляции
```

# Инкапсуляция: доступ к полям, конструкторам, методам

Обсудим на примере полей (для конструкторов и методов аналогично)

```
package student;

public class Student {
    public String name;        // доступно абсолютно везде
    String dateOfBirth;       // доступно в пакете student
    protected double gpa;     // доступно только внутри Student и в подклассах
    private String password;   // доступно только внутри Student
}
```

**Вопрос:** а можно ли узнать пароль следующим образом?

```
new Student(){ System.out.println(password); }
```

**Внимание:** `private` не гарантирует, что данные не смогут быть прочитаны и изменены другими классами (см. Java Reflection API).

# Наследование

## Суперкласс (класс-родитель)

```
class Student {  
    private String name;  
    private String studyProgram;  
  
    Student(String name, String studyProgram) {  
        this.name = name;  
        this.studyProgram = studyProgram;  
    }  
  
    String getInfo() {  
        return name + ", " + studyProgram;  
    }  
}
```

## Подкласс (класс-потомок)

```
class InformaticsStudent extends Student {  
    private String githubUser;  
  
    InformaticsStudent(String name, String user) {  
        // вызов конструктора суперкласса  
        super(name, "Информатика");  
        // инициализация собственных методов  
        githubUser = user;  
    }  
  
    String getGithubUser() {  
        return githubUser;  
    }  
}
```

```
InformaticsStudent student = new InformaticsStudent("Д.Э. Таргариен", "daenerys");  
String info = student.getInfo(); // "Д.Э. Таргариен, Информатика"  
String githubUser = student.getGithubUser(); // "daeneris"
```

# Наследование

Суперкласс (класс-родитель)

```
class Student {  
    // ...  
    Student() {  
        // ...  
    }  
}
```

Подкласс (класс-потомок)

```
class InformaticsStudent extends Student {  
    // ...  
    InformaticsStudent() {  
        // ...  
    }  
}
```

**Вопрос:** Корректен ли следующий код?

```
InformaticsStudent infoStudent = new Student();  
Student student = new InformaticsStudent();
```

# Наследование

Суперкласс (класс-родитель)

```
class Student {  
    // ...  
    Student() {  
        // ...  
    }  
}
```

Подкласс (класс-потомок)

```
class InformaticsStudent extends Student {  
    // ...  
    InformaticsStudent() {  
        // ...  
    }  
}
```

Ответ: Нет, ошибка компиляции в первой строке:

```
InformaticsStudent infoStudent = new Student();  
Student student = new InformaticsStudent();
```

Следующий вопрос: А поможет ли явное преобразование?

```
InformaticsStudent infoStudent = (InformaticsStudent)new Student();
```

# Композиция

- Композиция - альтернатива наследованию:
  - один класс включает в себя другой в качестве одного из полей

```
// наследование
class InformaticsStudent extends Student {
    private String githubUser;

    InformaticsStudent(String name, String user) {
        super(name, "Информатика");
        githubUser = user;
    }
}
```

```
// композиция
class InformaticsStudent {
    private Student student; // !!!
    private String githubUser;

    InformaticsStudent(String name, String user) {
        student = new Student(name, "Информатика");
        githubUser = user;
    }
}
```



# Класс `Object`

- Все пользовательские классы (без указания класса-родителя) являются потомками класса `java.lang.Object`

```
package org.example;  
  
class Empty /* extends Object */ {  
}
```

```
Empty empty = new Empty();  
empty.toString();           // "org.example.Empty@7a81197d"  
empty.hashCode();           // 2055281021  
  
Empty anotherEmpty = new Empty();  
empty.equals(anotherEmpty);  // false, т.к. сравнение ссылок
```

## Упражнение

```
class A {  
    String getClassName() { return "A"; }  
}  
  
class B extends A {  
    String getClassName() { return "B"; }  
}
```

```
A a = new B();  
String className = a.getClassName();
```

Какое значение будет записано в переменную `className` ?

## Полиморфизм: переопределение метода

```
class A {  
    String getClassName() { return "A"; }  
}  
  
class B extends A {  
    // рекомендуется использовать аннотацию @Override при переопределении  
    @Override  
    String getClassName() { return "B extends " + super.getClassName(); }  
}
```

```
A a = new B();  
String className = a.getClassName(); // "B extends A"
```

# Статический полиморфизм: перегрузка метода

- Возможность определения нескольких методов с одинаковым методом, но разными типами аргументов:

```
class Student {  
    private String name;  
  
    void initialize(String name) { this.name = name; }  
  
    void initialize() { initialize("Deyeneris"); }  
}
```

**Вопрос:** Можно ли добавить в класс `Student` следующий метод?

```
public String initialize() {  
    initialize("Дейенерис");  
    return name;  
}
```

# Статические поля и методы

```
class Course {  
    private static int courseCount; // статическое поле (поле класса)  
    private String courseTitle;    // поле экземпляра класса  
  
    Course(String courseTitle) {  
        this.courseTitle = courseTitle;  
        courseCount++;  
    }  
  
    static int getCourseCount() { // статический метод  
        return courseCount;  
    }  
}
```

```
Course java = new Course("Java");  
Course c = new Course("C");  
int count = Course.getCourseCount(); // 2
```

# Ключевое слово `final`

- Для классов: запрещено наследование

```
final class A {}  
  
class B extends A {} // ошибка компиляции
```

Пример `final` класса: `String`

- Для методов: запрещено переопределение

```
class A {  
    final String getType() { return "A"; }  
}  
  
class B extends A {  
    @Override  
    String getType() { return "B"; } // ошибка компиляции  
}
```

# Ключевое слово `final`

- Для переменных примитивного типа: нельзя изменить значение после инициализации

```
class Student {  
    final int age;  
    Student(int age) { this.age = age; }  
    void update(int age) { this.age = age; } // ошибка компиляции  
}
```

```
void swap(final int x, final int y) {  
    int tmp = x;  
    x = y;      // ошибка компиляции  
    y = tmp;    // ошибка компиляции  
}
```

```
void method() {  
    final int x = 1;  
    x = 2;      // ошибка компиляции  
}
```

# Ключевое слово `final`

- Для переменных примитивного типа:
  - нельзя изменить ссылку на объект после инициализации
  - МОЖНО ИЗМЕНИТЬ СОСТОЯНИЕ ОБЪЕКТА

```
class Student {  
    final String[] courseNames = new String[10];  
  
    void addCourse() { courseNames[0] = "Java"; } // OK  
  
    void changeReference() {  
        courseNames = new String[20]; // ошибка компиляции  
    }  
}
```



# Перечисления `enum`

```
enum Language {  
    JAVA,  
    PYTHON,  
    C  
}
```

Пример использования перечислений:

```
Language java = Language.JAVA;  
int ordinal = java.ordinal(); // 0 (индекс в списке констант)  
String name = java.name();    // "JAVA" (имя элемента перечисления)  
  
// итерация по элементам перечисления  
for (Language language : Language.values()) {  
    System.out.println(language)  
}
```

# Абстрактный класс

```
abstract class Student {  
    String name;  
  
    // абстрактный метод: объявлен, но не реализован  
    abstract String getInfo();  
}
```

- Нельзя создать объект абстрактного класса:

```
Student student = new Student(); // ошибка компиляции
```

- Абстрактные методы реализуются в классах-потомках:

```
class InformaticsStudent extends Student {  
    @Override  
    String getInfo() { return "Информатик"; }  
}
```

```
class MathematicsStudent extends Student {  
    @Override  
    String getInfo() { return "Математик"; }  
}
```

# Интерфейс

Содержит только абстрактные (нереализованные методы)

```
interface FileStorage {  
    void uploadFile(String fileName, String fileContent);  
}
```

Реализация интерфейсов:

```
// Хранение данных на локальном компьютере  
class LocalStorage implements FileStorage {  
    String folder = "C:\\Users\\Deyeneris\\SecretStorage";  
  
    @Override  
    void uploadFile(String fileName, String fileContent) {  
        // реализация  
    }  
}
```

```
// Хранение данных в облачном сервисе  
class CloudStorage implements FileStorage {  
    String storageUrl = "https://...";  
    String storageUserName = "deyeneris";  
    String storageUserPassword = "qwerty";  
  
    @Override  
    void uploadFile(String fileName, String fileContent) {  
        // реализация  
    }  
}
```

Использование интерфейсов:

```
FileStorage storage = new CloudStorage();  
storage.uploadFile("my-cloud-password.txt", "qwerty");
```

# Реализация нескольких интерфейсов

Классы могут реализовывать несколько интерфейсов

```
interface FileUploader {  
    void uploadFile(String fileName, String fileContent);  
}
```

```
interface FileDownloader {  
    String downloadFile(String fileName);  
}
```

```
class LocalStorage implements FileUploader, FileDownloader {  
    @Override  
    void uploadFile(String fileName, String fileContent) {  
        // реализация  
    }  
  
    @Override  
    String downloadFile(String fileName) {  
        // реализация  
    }  
}
```

```
LocalStorage storage = new LocalStorage();  
FileUploader fileUploader = storage;  
FileDownloader fileDownloader = storage;
```