Язык Java

Глава III. Основы объектно-ориентированного программирования

Виталий Витальевич Перевощиков

Осенний семестр 2021

Содержание

Объектно-ориентированное программирование

- Объектно-ориентированное программирование (ООП):
  - Представление программы в виде совокупности объектов
  - Объект включает в себя:
    - данные (поля)
    - программный код (методы)
  - Объекты взаимодействуют друг с другом (посредством обмена сообщений)
- Виды ООП:
  - Прототипное программирование (JavaScript)
  - Класс-ориентированное программирование (Java, C++, C#):
    - **Класс** - шаблон для создания объектов

Принципы ООП

- **Абстракция**
  - Использование только тех свойств объекта, которые нужны для решения поставленной задачи
  - Пример: объект "студент":
    - Важные свойства: имя, специальность, СНИЛС, ...
    - Игнорируемые свойства: любимая еда, хобби, ...
- **Инкапсуляция**
  - Ограничение доступа одних компонентов программы к другим
  - Пример: приватные поля и методы
- **Наследование**
  - Построение новых подклассов из существующих классов за счет расширения их функциональности
  - Пример: класс "студент"
    - Подкласс: "студент-программист"(профиль на Github, знание Git, C, Java, ...)
    - Подкласс: "студент-математик"(знание дифф. геометрии)
- **Полиморфизм**
  - Способность метода обрабатывать разные типы данных

Классы и объекты в Java

- Определение класса

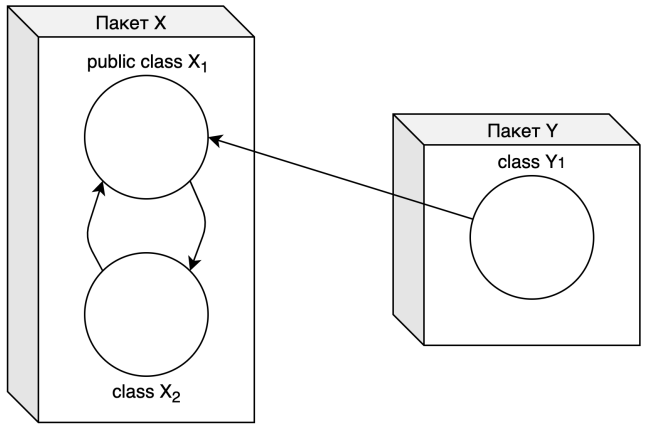
```java
class Student {
    // статические поля (не привязаны к конкретному объекту)
    static String role = "Student";

    // поля конкретного объекта
    String name;         // Иванов Иван Иванович
    String studyProgram; // математическое обеспечение

    // конструкторы
    Student() {}  // пустой конструктор
    Student(String name, String studyProgram) {  // конструктор с параметрами
        this.name = name;
        this.studyProgram = studyProgram;
    }

    // статические методы (не привязаны к конкретному объекту)
    static void printRole() {
        System.out.println(role);
    }

    // методы конкретного объекта
    String getInfo() {
        return name + ", " + studyProgram;
    }
    void setName(String name) {
        this.name = name;
    }
}
```

- Работа с объектами

```java
// создание экземпляра класса
Student student = new Student("Иванов И.И.", "МО");

String info = student.getInfo(); // "Иванов И.И., МО"
student.setName("Петров И.И.");
String infoAfterChange = student.getInfo(); // "Петров И.И., МО"
```

Инкапсуляция: доступ к классам

- Модификаторы доступа:
  - `private`: доступен из всех классов
  - по умолчанию (без ключевого слова): доступен только из классов своего пакета



- Пример:

```
package com.company;

import ru.bfu.ipmit.Student;

class ExternalStudentService {

    Student getStudent() {
        r
    }

}
```

'ru.bfu.ipmit.Student' is not public in 'ru.bfu.ipmit'. Cannot be accessed from outside package

Create type parameter 'Student' ⌥⇧⏎ More actions... ⌥⏎

Инкапсуляция: доступ к полям, конструкторам и методам

```java
package ru.bfu.ipmit;

public class Teacher {

    // поля
    public String name;           // доступно для всех классов
    Student[] students;           // доступно только внутри пакета ru.bfu.ipmit
    protected String[] examTasks; // доступно только для подклассов Teacher, например, Docent
    private double salary;        // доступно только внутри класса Teacher

    // конструкторы
    public Teacher(String name) {}
    Teacher(String name, Student[] students) {}
    protected Teacher(String name, Student[] students, String[] examTasks) {}
    private Teacher(String name, Student[] students, String[] examTasks, double salary) {}

    // методы
    public String getName() { return name; }
    Student[] getStudents() { return students; }
    protected String[] getExamTasks() { return examTasks; }
    private double getSalary() { return salary; }

}
```

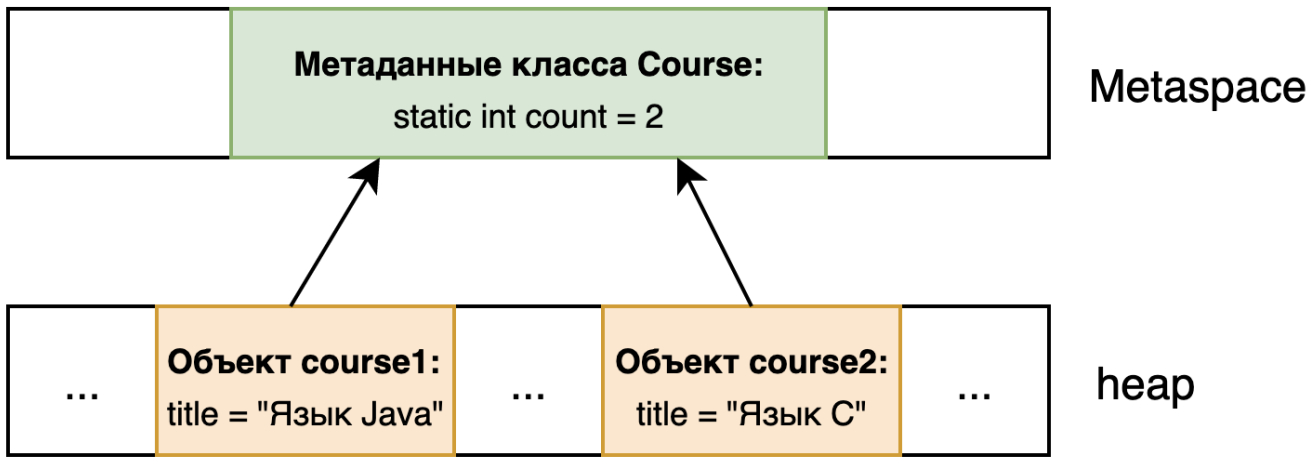
Статические поля и методы

```java
package ru.bfu.ipmit;

public class Course {
    // статические поля
    private static int count;  // общее количество курсов

    // поля экземпляра класса
    private String title;

    public Course(String title) {
        this.title = title;
        count++;
    }

    // 2 способа доступа к статическому полю
    public static int getStaticCount() { return count; } // статический метод
    public int getInstanceCount() { return count; } // метод экземпляра класса

    public String getTitle() { return title; }
}
```

```java
Course courseInstance = new Course("Язык Java");
int staticCount = Course.getStaticCount(); // 1
int instanceCount = courseInstance.getInstanceCount(); // 1
```

Статические поля и методы

- Хранятся в области памяти "Metaspace":



Наследование

```java
// класс-родитель
class Student {
    private String name;
    private String studyProgram;

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

    public String getInfo() {
        return name + ", " + studyProgram;
    }
}
```

```java
// класс-потомок
class ProgrammingStudent extends Student {
    private String githubLink;

    public ProgrammingStudent(String name, String githubLink) {
        // вызов конструктора класса-родителя Student
        super(name, "Программирование");
        // определение собственных методов класса-потомка
        this.githubLink = githubLink;
    }

    public String getGithubLink() { return githubLink; }
}
```

```java
ProgrammingStudent student = new ProgrammingStudent("Иванов И.И.", "github.com/ivanov");
// Метод класса-родителя
String info = student.getInfo(); // "Иванов И.И., Программирование"
String link = student.getGithubLink(); // "github.com/ivanov"
```

```java
ProgrammingStudent progStudent = new ProgrammingStudent("Иванов И.И.", "github.com/ivanov");

Student student = progStudent; // неявное преобразование к классу-предку
String githubLink = student.getGithubLink(); // метод недоступен

progStudent = student; // неявное преобразование невозможно
progStudent = (ProgrammingStudent)student; // явное преобразование к классу-потомку
githubLink = progStudent.getGithubLink();
```

Композиция

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

```java
// наследование
class ProgrammingStudent extends Student {
    private String githubLink;

    public ProgrammingStudent(String name, String githubLink) {
        super(name, "Программирование");
        this.githubLink = githubLink;
    }

    // переопределение метода getInfo
    @Override
    public String getInfo() { return super.getInfo() + ", " + githubLink; }
}
```

```java
// композиция
public class ProgrammingStudent {

    private Student student;
    private String githubLink;

    public ProgrammingStudent(String name, String githubLink) {
        this.student = new Student(name, "Programming");
        this.githubLink = githubLink;
    }

    public String getInfo() { return student.getInfo() + ", " + githubLink; }
}
```

Класс Object

- Все пользовательские классы в Java являются потомками класса `java.lang.Object`

```java
class Empty /* extends Object */{
}
```

```java
// Методы toString, hashCode и equals наследуются от класса Object
Empty empty = new Empty();
String toString = empty.toString(); // "ru.bfu.ipmit.Empty@4617c264"
int hash = empty.hashCode(); // 1175962212

Empty anotherEmpty = new Empty();
boolean equals = empty.equals(anotherEmpty); // false, т.к. сравнение ссылок
```

Полиморфизм: переопределение метода

```java
// класс-родитель
class Student {
    private String name;
    private String studyProgram;

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

    public String getInfo() {
        return name + ", " + studyProgram;
    }
}
```

```java
// класс-потомок
class ProgrammingStudent extends Student {
    private String githubLink;

    public ProgrammingStudent(String name, String githubLink) {
        super(name, "Программирование");
        this.githubLink = githubLink;
    }

    // переопределение метода getInfo
    @Override
    public String getInfo() {
        return super.getInfo() + ", " + githubLink;
    }
}
```

```java
ProgrammingStudent student = new ProgrammingStudent("Иванов И.И.", "github.com/ivanov");
// Вызывается переопределенный метод класса-потомка
String info = student.getInfo(); // "Иванов И.И., Программирование, github.com/ivanov"
```

”Магия” полиморфизма

- Полиморфизм позволяет использовать экземпляр класса-потомка вместо экземпляра класса-предка

```java
public class StudentService {

    public static String getStudentInfo(Student student) {
        return student.getInfo();
    }

}
```

```java
Student mathStudent = new Student("Петров П.П.", "Математика");
ProgrammingStudent progStudent = new ProgrammingStudent("Иванов И.И.", "github.com/ivanov");

String mathStudentInfo = StudentService.getStudentInfo(mathStudent); // "Петров П.П., Математика"
String progStudentInfo = StudentService.getStudentInfo(progStudent); // Иванов И.И., Программирование, github.com/ivanov
```

Статический полиморфизм: перегрузка метода

- Возможность определения нескольких методов с одинаковым именем, но разными параметрами

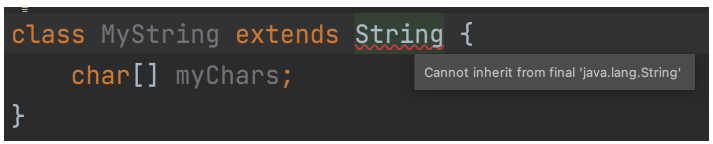
```
class Student {
    private String name;
    private String studyProgram;

    public void initialize() {
        initialize("", "");
    }

    public void initialize(String name) {
        initialize(name, "");
    }

    public void initialize(String name, String studyProgram) {
        this.name = name;
        this.studyProgram = studyProgram;
    }
}
```

- Перегрузка невозможна, если методы отличаются только типом возвращаемого значения:

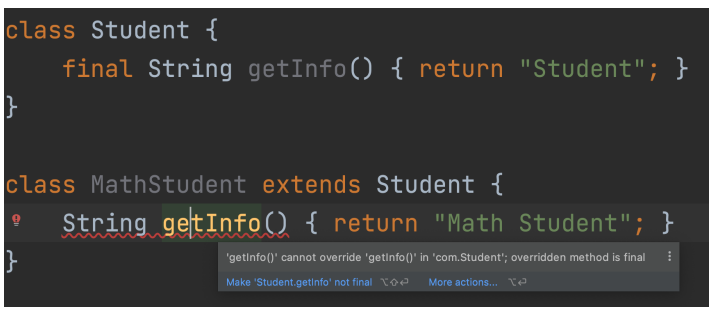
```
public boolean initialize() {
    initializ  'initialize()' is already defined in 'ru.bfu.ipmit.Student'
    return true;
}
```

Ключевое слово `final`

- Для **классов**: запрещено наследование



- Для **методов**: запрещено переопределение



Ключевое слово `final`

- Для **переменных**:
  - Для переменных **примитивного** типа: нельзя изменить значение после инициализации

```java
class Student {
    final int age;

    Student(int age) {
        this.age = age; // инициализация переменной
    }

    void updateAge(int age) {
        this.age = age; // изменение переменной
    }
}
```

Cannot assign a value to final variable 'age'

Make 'Student.age' not final ⌥⇧⏎ More actions... ⌥⏎

  - Для переменных **ссылочного** типа:
    - нельзя изменить ссылку на объект
    - можно изменить состояние объекта

```java
class Student {
    // инициализация переменной ссылочного типа
    final String[] courseNames = new String[10];

    void changeCourses() {
        courseNames[0] = "Язык Java"; // МОЖНО изменять состояние
        courseNames = new String[20]; // НЕЛЬЗЯ изменять ссылку
    }
}
```

Хранение констант

- Пример использования `final`: “статический” класс для хранения констант:

```java
// final для запрета наследования
final class Languages {
    public static final String JAVA = "Java";
    public static final String C_PLUS_PLUS = "C++";
    public static final String PYTHON = "Python";
    public static final String JAVA_SCRIPT = "JavaScript";

    // скрытый конструктор для запрета создания экземпляров класса
    private Languages() {
    }
}
```

```java
String java = Languages.JAVA;
String javaScript = Languages.JAVA_SCRIPT;

Languages.JAVA = "C++"; // нельзя изменить значение константы
Languages instance = new Languages(); // нельзя создать экземпляр класса
```

- **Внимание**: соглашение для имен констант: `JAVA_SCRIPT`

Перечисления enum

- “Усовершенствованный” вариант хранения констант:

```
enum Language {
    JAVA,
    C_PLUS_PLUS,
    PYTHON,
    JAVA_SCRIPT
}
```

- Пример использования перечислений:

```
Language java = Language.JAVA;
int ordinal = java.ordinal(); // 0 (индекс в списке констант)
String name = java.name(); // "JAVA" (имя элемента перечисления)

// обход элементов перечисления
for (Language language : Language.values()) {
    System.out.println(language);
}
```

- **Рекомендация**: если известно, что переменная типа `String` принимает всего несколько значений, то следует использовать enum вместо `String`

Абстрактный класс

- Нельзя создать экземпляр абстрактного класса
- Экземпляр можно создать только в классах-наследниках
- Абстрактный класс может (но не обязан) иметь **абстрактные методы**:
  - объявленные, но не реализованные методы
  - реализация производится в классах-потомках

```java
abstract class Student {
    public String name;

    Student(String name) { this.name = name; }
    // объявление абстрактного метода
    abstract String getInfo();
}
```

```java
class ProgrammingStudent extends Student {
    ProgrammingStudent(String name) { super(name); }

    String getInfo() {
        return name + ", студент-программист";
    }
}
```

```java
class MathStudent extends Student {
    MathStudent(String name) { super(name); }

    String getInfo() {
        return name + ", студент-математик";
    }
}
```

```java
Student mathStudent = new MathStudent("Иванов И.И.");
String mathStudentInfo = mathStudent.getInfo(); // "Иванов И.И., студент-программист"

Student progStudent = new ProgrammingStudent("Петров П.П.");
String progStudentInfo = progStudent.getInfo(); // "Петров П.П., студент-математик"

Student student = new Student("Сидоров С.С.");
```

'Student' is abstract; cannot be instantiated

Implement methods ⌥⇧⏎ More actions... ⌥⏎

Интерфейс

- Ссылочный тип данных
- Содержит только абстрактные (нереализованные методы)
- Может также содержать константы и статические методы

```java
interface FileStorage {
    // абстрактные методы
    void uploadFile(String fileName, String fileContent);
}

class LocalStorage implements FileStorage {
    @Override
    public void uploadFile(String fileName, String fileContent) {
        // Сохранение файла на локальном компьютере
    }
}

class CloudStorage implements FileStorage {
    @Override
    public void uploadFile(String fileName, String fileContent) {
        // загрузка файла в облачное хранилище
    }
}
```

```java
// чтение флага из конфигурационного файла
boolean isCloudStorage = readFromConfiguration("cloudStorage");

FileStorage storage = isCloudStorage ? new LocalStorage() : new CloudStorage();
storage.uploadFile("hello.txt", "hello");
```

Реализация нескольких интерфейсов

- Классы могут реализовывать несколько интерфейсов:

```java
interface Student {
    String getStudyProgram();
}

interface Teacher {
    String[] getCourses();
}

class TeachingStudent implements Student, Teacher {
    @Override
    public String getStudyProgram() {
        return "Программирование";
    }

    @Override
    public String[] getCourses() {
        return new String[]{ "Практикум по программированию" };
    }
}
```

```java
TeachingStudent teachingStudent = new TeachingStudent();
Student student = teachingStudent;
Teacher teacher = teachingStudent;
```

- **Внимание**: множественное наследование (`extends`) классов в Java не поддерживается

Вопрос для самопроверки

Является ли следующая программа корректной?

```
class A {}

class B extends A {}

public class Main {

    public static void main(String[] args) {
        A a = new B();
        B b = new A();
    }
}
```

Варианты ответа:

- да, конечно
- нет, произойдет ошибка компиляции
- нет, произойдет ошибка времени выполнения