Урок 5

Абстрактные классы, интерфейсы и полиморфизм

Завершаем знакомство с основными принципами ООП — рассматриваем полиморфизм. Но прежде изучим несколько инструментов, которые помогают Java в реализации этого принципа.

Абстрактные классы

Класс, который содержит хотя бы один абстрактный метод, должен быть определен как абстрактный. Нельзя создать экземпляр такого класса. Методы, объявленные абстрактными, только описывают смысл и не могут включать реализации.

Это класс-шаблон: реализует функциональность только на том уровне, на котором она известна на данный момент. Производные классы ее дополняют.

При наследовании от абстрактного класса все методы, помеченные абстрактными в родительском классе, должны быть переопределены в классе-потомке. Область видимости этих методов должна совпадать (или быть менее строгой). Контроль типов (который мы рассмотрим позже) и количество обязательных аргументов должно быть одинаковым.

Абстрактные классы нужны, чтобы собирать общее поведение будущих дочерних классов, но не давать конкретики. Нельзя представить конкретную реализацию класса **Animal** в жизни. Обычно наши представления — на уровне **Cat** или **Dog**.

```
public abstract class Animal {
    public abstract void voice();
    public void jump() {
        System.out.println("Животное подпрыгнуло");
    }
}
public class Cat extends Animal {
    @Override
    public void voice() {
        System.out.println("Кот мяукнул");
    }
}
```

Несмотря на то, что абстрактные классы не позволяют получать экземпляры объектов, их все же можно применять для создания ссылок на объекты подклассов.

```
Animal a = new Cat();
```

Интерфейсы

С помощью интерфейсов можно описать методы, которые должны быть реализованы в классе без описания функционала.

Интерфейсы объявляются так же, как обычные классы, но с использованием ключевого слова **interface**. Тела методов интерфейсов должны быть пустыми. Методы внутри интерфейса должны быть определены как публичные. Интерфейсы предназначены для того, чтобы определять протокол общения с семейством классов, реализующих их. Пример описания интерфейса:

```
interface CarTemplate
{
    public int getId(); // Получить id автомобиля
    public String getName(); // Получить название
    public void move(); // Ехать
}
```

Для реализации интерфейса используется оператор **implements**. Класс должен реализовать все методы, описанные в интерфейсе, иначе произойдет фатальная ошибка. При необходимости классы могут реализовывать более одного интерфейса (они должны разделяться запятой).

Пример:

```
class Audi implements CarTemplate {
     int getId() {
          return "1-ATHD98";
    }

    String getName() {
         return "Audi";
    }

    void move() {
        System.out.println(this.name + " is moving");
    }
}
```

И абстрактный класс, и интерфейс имеют свои области применения:

Abstract Class	Interface
Абстрактные методы нужно указывать явно с помощью ключевого слова **abstract**	Все методы являются абстрактными
Абстрактные методы можно объявлять с идентификаторами доступа (**public, protected, private**). При реализации в классе-потомке методы должны иметь такой же модификатор (или менее ограниченный)	Все методы — публичные
Может содержать методы с реализацией, поля, константы	Может содержать константы
Нет множественного наследования, поэтому класс может наследовать только один абстрактный класс	Класс может наследовать много интерфейсов

Когда использовать классы, а когда — интерфейсы? Нужно иметь в виду, что интерфейс описывает поведение и возможности своих реализаций. Обратите внимание на классические названия интерфейсов: **Throwable**, **Countable**, **Comparable**, **Iterable**. Рассмотрим, к примеру, интерфейс **Rollable** («катящийся»), и **Foldable** («складывающийся») — они описывают характеристики. А абстрактный класс излагает суть. Например, стол — **Table_Abstract** — может быть деревянным (**Table_Wood extends Table_Abstract**) или хирургическим (**Table_Surgical extends Table_Abstract**). В данном случае **Table_Abstract** объединяет свойства всех столов (площадь поверхности, наличие ножек), а конкретный класс описывает сущность определенного типа предметов мебели. Связь интерфейсов и классов описывает свойства: стол можно катить (**Table_Abstract implements Rollable**), деревянный стол можно сложить (**Table_Wood implements Foldable**).

Полиморфизм

Полиморфизм — это способность объекта использовать методы производного класса, которого не существует на момент создания базового. Рассмотрим пример того, как применяется полиморфизм.
Предположим, на сайте нужны публикации двух видов — новости и анонсы. У обоих есть заголовок, текст, дата. Но есть и отличия: у новостей есть источники, а у анонсов — краткие описания.
Простой вариант — написать два отдельных класса и работать с ними; или один, в который ввести все свойства, присущие типам публикаций, а задействовать только нужные. Но для разных типов аналогичные по логике методы должны работать по-разному. Делать несколько однотипных методов для разных типов (**get_news**, **get_announcements**) — совсем неграмотно. Решение — использовать полиморфизм:

```
abstract class Publication
{
    protected String title;

    public Publication(String title){
        this.title = title
    }

    public void setTitle(String title){
        this.title = title;
    }

    public String getTitle(){
        return title;
    }

    // Этот метод должен напечатать публикацию, но мы не знаем, как именно это
сделать, и потому объявляем его абстрактным
    abstract public void render();
}
```

Теперь можно перейти к созданию производных классов, которые и реализуют недостающую функциональность:

```
class News extends Publication
{
    public void render(){
        System.out.println("Вывод на экран новости");
    }
}
```

```
class Announcement extends Publication
{
    public void render(){
        cutText();
        System.out.println("Вывод на экран анонса");
    }

    private String cutText(){
        // Укорачивание текста для анонса
    }
}
```

Теперь при использовании можно заменить конкретный класс его абстрактным родителем или интерфейсом:

```
Publication[] publications = new Publication[2];

publication[0] = new News();
publication[1] = new Announcement();

for(Publication pub : publications){
    pub.render();
}
```