2. Интерфейсы

В языке Java реализация принципа наследования имеет нюансы. Одним из них является **запрет множественного наследования**.

Создавать для классов наследование от нескольких суперклассов **нельзя**.

На это есть несколько причин:

Из-за множественного наследования может возникнуть ситуация, при которой несколько родительских классов имеют методы с одинаковым названием. В этом случае класс-наследник просто не поймет, к какому из методов нужно обращаться.

Также может возникнуть ситуация, при которой мы получим **ромбовидное наследование**. Это ситуация, когда два родительских класса наследуют свойства одного и того же родительского класса.

Ромбовидное наследование получается, если:

изначальный метод определен в самом старшем классе;

его реализация делится;

дальше опять вступает в силу неоднозначность, которую невозможно разрешить.

Именно **ромбовидное наследование** является **основной причиной** того, почему в Java нет множественного наследования.

Однако нам, как разработчикам, всё равно может понадобиться общий для нескольких классов функционал, который мы не можем добавить в единственный

родительский класс. Для решения этой проблемы в Java есть **интерфейсы**.

Интерфейсы в данном случае не имеют ничего общего с визуальной частью программы.

> **Интерфейсы** — так же, как и абстрактные классы — **могут содержать в себе абстрактные одноименные методы**, т. е. методы без конкретной реализации. Мы можем обращаться к любому из этих методов в классах, имплементирующих данный интерфейс.

Интерфейсы в Java реализовывают свойства **полиморфизма**.

> **Интерфейс** содержит только константы и методы, актуальные для классов, реализующих данный интерфейс. Иначе говоря — это особый класс, который определяет только поведение.

Реализовать методы необходимо в классах, применяющих данный интерфейс. Один класс может реализовывать сразу несколько интерфейсов.

Для создания интерфейса необходимо использовать ключевое слово

```
interface :
```

```
interface Printable {
   void print();
}
```

Мы создали интерфейс. Теперь в нем можно определить методы или переменные.

Мы добавили в интерфейс метод, который требует реализации в каждом классе, который будет применять наш интерфейс.

> **Что нужно помнить:**

Интерфейсы могут содержать также методы **с реализацией**.

Невозможно создавать экземпляры интерфейсов.

Интерфейс **определяет только поведение и не определяет свойства!**

Модификаторы доступа у членов интерфейса указывать не нужно — все методы интерфейса имеют доступ `public` по умолчанию, а переменные имеют модификатор `public static final`, то есть являются **константами**.

Для того чтобы применить интерфейс в каком-либо классе, нужно добавить в строке объявления класса ключевое слово `implements` и название интерфейса.

```
public class Test { // Класс с методом main
   public static void main(String[] args) {
      Book book = new Book("War and Peace", "Lev Tolstoy");
      book.print();
   }
}

// Создадим интерфейс и определим в нем абстрактный метод print
interface Printable {
   void print();
}
```

Создадим новый класс `Book`, который имплементирует интерфейс `Printable`.

```
class Book implements Printable {
   String name;
   String author;

   Book(String name, String author) {
      this.name = name;
      this.author = author;
   }

   // Добавим геттеры для полей
   public String getName() {
      return name;
   }

   public String getAuthor() {
```

```
        return author;
    }

    // Реализуем в нем метод из интерфейса
    public void print() {
        System.out.println(name + " - " + author);
    }
}
```

В данном случае интерфейс `Printable` реализован классом `Book`, в котором реализован метод `print()`. Если класс, реализовывающий интерфейс, не является абстрактным, он должен также реализовывать **все методы интерфейса**.

Мы можем реализовать любой интерфейс сразу в нескольких классах. Давайте добавим еще один класс, который будет реализовывать наш интерфейс.

Создадим класс `Magazine`, в котором также имплементируем интерфейс `Printable`.

```
class Magazine implements Printable {
    private String name;

    String getName() {
        return name;
    }

    Magazine(String name) {
        this.name = name;
    }

    // Тут тоже необходимо реализовать метод из интерфейса
    public void print() {
        System.out.println(name);
    }
}
```

Оба созданных класса объединяет то, что они реализуют один интерфейс. В связи с этим мы можем создавать экземпляры этих классов, определяя их тип как `Printable`.

```
public class Test {
   public static void main(String[] args) {

      // В данном случае мы можем определить тип ссылки на объект
      // равным типу интерфейса
      Printable printable = new Book("War and Peace", "Lev Tolstoy");
      printable.print(); //  Lev Tolstoy – War and Peace

      printable = new Magazine("National Geografic");
      printable.print(); // National Geografic
    }
}

interface Printable {
   void print();
}

class Book implements Printable {
   String name;
   String author;

   Book(String name, String author) {
      this.name = name;
      this.author = author;
   }

      public String getName() {
      return name;
   }

   public String getAuthor() {
      return author;
   }

   public void print() {
      System.out.println(name + " - " + author);
   }
}

class Magazine implements Printable {
   private String name;

   String getName() {
      return name;
   }

   Magazine(String name) {
      this.name = name;
   }
```

```
    public void print() {
        System.out.println(name);
    }
}
```

Как мы упоминали в предыдущем уроке, интерфейс может быть реализован в том числе и в **анонимном классе**.

```
public interface Moto {
    void ride();
    void stop();
}

public class MotoTest {
    public static void main(String[] args) {
        Moto moto = new Moto() {

            @Override
            public void ride() {
                System.out.println("ride a motorcycle!!!");
            }

            @Override
            public void stop() {
                System.out.println("stop motorcycle");
            }
        };

        moveable.ride();
        moveable.stop();
    }
}
```

Интерфейсы в преобразованиях типов

В случае нашего примера можно сказать, что на объект класса `Book` указывает ссылка типа `Printable`. И так как класс `Book` реализует этот интерфейс, он реализовывает и методы из этого интерфейса.

Создадим еще один класс с методом `main`. Реализуем в нем следующий код.

```
public class Test2 {
   public static void main(String[] args) {

      Printable printable = new Book("War and Peace", "Lev Tolstoy");
      printable.print();

      // Тут используется явное приведение,
      // так как в интерфейсе нет метода getName
      String name = ((Book) printable).getName();
      System.out.println(name);
    }
}
```

Явное приведение типов — `(Book)printable.getName();` — нужно для того, чтобы через ссылку интерфейса вызывать методы класса, к которому относится наш объект.

Методы по умолчанию

В восьмой версии Java была добавлена возможность прописывать реализацию методов прямо в интерфейсе. Данная реализация распространяется на все классы, имплементирующие интерфейс.

Однако ничего не мешает нам переопределить эти методы в реализующих классах. Такие методы называются **дефолтными**, или **методами по умолчанию**. Обозначаются они ключевым словом `default` .

Давайте сделаем ранее созданный нами абстрактный метод в интерфейсе **дефолтным**, то есть добавим ему реализацию.

```
interface Printable {
   default void print() {
      System.out.println("Default method");
   }
}
```

По сути, дефолтный метод — это обычный метод, который без переопределения будет одинаково работать во всех имплементирующих его классах.

```
// Класс Magazine в данном случае реализует интерфейс Printable,
// и объекты этого класса имеют метод print, созданный в интерфейсе,
// с той реализацией, которая обозначена в самом интерфейсе

class Magazine implements Printable {
   private String name;

   String getName() {
      return name;
   }

   Magazine(String name) {
      this.name = name;
   }
}
```

Статические методы

В восьмой версии Java также была добавлена возможность реализации статических методов в интерфейсах. Статические интерфейсы используются исключительно для обслуживания самого интерфейса.

```
interface Printable {
   void print();

   static void read() {
      System.out.println("Static method");
   }
}
```

Для того чтобы вызвать этот метод, нужно прописать сначала имя интерфейса, а потом, через точку, название метода. То есть сделать всё так же, как при вызове статических методов в классе.

```
public class Test3 {

   public static void main(String[] args) {
      Printable.read();
   }
}
```

Константы в интерфейсах

Интерфейсы могут иметь переменные. И, хотя они не имеют явных модификаторов доступа, все они имеют тип доступа `public static final`, что делает их **константами**.

```
interface SomeInterface {
   int const1 = 1;
   int const2 = 0;

   void printConst(int value);
}
```

Константы можно использовать в любой точке программы.

```
public class Program {
   public static void main(String[] args) {
      SomeClass object1 = new SomeClass();
      object1.printConst(1);
   }
}

class SomeClass implements SomeInterface {
   public void printConst(int value) {
      if(value == const1) {
         System.out.println(const1);
      } else if(value == const2) {
         System.out.println(const2);
      } else {
         System.out.println("nothing");
      }
   }
}

interface SomeInterface {
   int const1 = 1;
   int const2 = 0;

   void printConst(int value);
}
```

Множественная реализация интерфейсов

Чтобы расширить свой функционал, один класс может имплементировать сразу несколько интерфейсов. Для этого после слова `implements` достаточно перечислить все необходимые интерфейсы через запятую.

```
interface Printable {
   ... // Методы интерфейса
}

interface Searchable {
   ... // Методы интерфейса
}

class Book implements Printable, Searchable{
   ... // Реализация класса
}
```

Наследование интерфейсов

Чтобы расширить функционал применяемых интерфейсов, мы можем наследовать интерфейсы аналогично классам.

```
interface MagazinePrintable extends Printable {
   void paint();
}
```

Классы, имплементирующие самый младший интерфейс (последний по старшинству в череде наследующих друг друга интерфейсов), должны будут реализовывать методы всех интерфейсов, наследующих друг друга.

Интерфейсы как параметры и результаты методов

У интерфейсов есть еще одно свойство — их можно использовать как тип возвращаемого значения или передавать в качестве типа параметра метода.

```
public class Test {
   public static void main(String[] args) {

      Printable printable = createPrintable("National Geographic",false);
      printable.print();

      read(new Book("Dead Souls", "N. Gogol'"));
      read(new Magazine("New York Times"));
   }

   static void read(Printable p) {
      p.print();
   }

   static Printable createPrintable(String name, boolean option) {
      if(option) {
         return new Book(name, "Undefined");
      } else {
         return new Magazine(name);
      }
   }
}

interface Printable {
   void print();
}

class Book implements Printable {
   String name;
   String author;

   Book(String name, String author) {
      this.name = name;
      this.author = author;
   }

   public void print() {
      System.out.println(name + " - " + author);
   }
}

class Magazine implements Printable {
   private String name;

   String getName() {
      return name;
   }

   Magazine(String name) {
```

```
        this.name = name;
    }

    public void print() {
        System.out.println(name);
    }
}
```

Так как метод `read()` принимает объект, а метод `createPrintable()` возвращает объект интерфейса `Printable`, мы в обоих случаях можем использовать типы классов, реализующих данный интерфейс.

На выходе получим:

```
National Geographic

N. Gogol' - Dead Souls

New York Times
```