3. Дженерики, или параметрический полиморфизм

Главным инструментом полиморфизма являются **дженерики**.

> **Дженерики (или обобщения)** — это параметризованные типы. Они помогают компилятору определить, с каким типом данных будет производиться работа.

Благодаря дженерикам мы можем создавать **интерфейсы, классы и методы**, у которых в качестве параметра в треугольных скобках определен тип данных, которым они оперируют. Такой подход уменьшает вероятность ошибок, делает код более универсальным и емким.

Также дженерики используются при хранении данных в различных структурах, чтобы точно определить, объекты какого типа нужно хранить в том или ином массиве.

Кроме того, если кто-то из других разработчиков решит положить в массив элемент неправильного типа, ошибка станет заметной сразу — на этапе компиляции.

Параметризованные классы

> **Параметризованный класс** — класс, при объявлении которого явно указывается тип данных, с которым он работает.

Давайте для наглядности поработаем с массивами в параметризованном классе.

```
// Параметризуем наш класс типом T
public class ClassArrays<T> {

   // Объявим в классе поле — массив типа T
   private T[] array;

   // Создадим конструктор на основе нашего массива
   public ClassArrays(T[] array) {
      this.array = array;
   }

   // Создадим в методе main несколько объектов нашего типа
   public static void main(String[] args) {
      ClassArrays<Byte> byteArray = new ClassArrays<>(new Byte[4]);
      ClassArrays<Double> doubleArray = new ClassArrays<>(new Double[3]);
      ClassArrays<Integer> integerArray = new ClassArrays<>(new Integer[9]);

   // Благодаря тому, что мы использовали универсальный тип данных — T,
   // мы можем передать в конструктор нашего объекта массив, который
   // может содержать объекты любого типа, что существенно сокращает
   // количество написанного нами кода

    }
}
```

Для создания конкретных объектов класса `ClassArrays` мы указываем `Byte`, `Integer` и `Double` в качестве аргумента типа, с которым мы хотим работать.

Благодаря тому что при объявлении класса `ClassArrays` мы указали параметр `<T>`, нет необходимости создавать объекты разных классов. В данном случае у нас только одна версия класса, которая корректно будет работать с каждым из типов.

> Имейте в виду, что в качестве параметра в дженериках **нельзя** указывать примитивы. Это ограничение обусловлено тем, что разработчиками языка Java в дженериках не были реализованы **Автоупаковка** и **Автораспаковка**.

Если мы указываем в качестве заполнителя какой-то универсальный символ, например `<T>`, это будет означать, что при выполнении какой-либо работы необходимо будет подставить конкретный тип данных вместо этого символа. Соответственно, этот тип будет учитываться при выполнении логики, в которой задействован параметр типа.

Ограниченные типы

В Java есть возможность ограничить параметр типа.

Если нам необходимо ограничить параметр типа классом-предком, то в таком случае мы должны использовать ключевое слово `extends` .

```
class SomeClass <T extends SomeSuperClass>
```

В данном случае нашим параметром типа может быть любой класс-наследник класса `SomeSuperClass` , включая его самого.

Также в качестве типов ограничений мы можем использовать и интерфейсы.

```
public class SomeClass<T extends AutoCloseable> {
    ... // Реализация класса
}
```

И даже можем комбинировать типы ограничений.

```
class SomeClass <T extends SomeSuperClass & AutoCloseable> {
    ... // Реализация класса
}
```

В таком случае необходимо перечислять их через разделитель-амперсанд `&` , причем на первой позиции должен быть указан именно тип класса, а уже за ним можно перечислять интерфейсы.

В данном случае мы можем использовать в качестве параметра типа класс, который является классом `SomeSuperClass` или его наследником, а также реализовывать интерфейс `Autoclosable` .

Wildcard

> **Wildcard** — это неизвестный тип данных, который обозначается знаком <?>. На месте этого символа можно использовать любой тип данных в одной реализации метода.

Wildcard, по сути, — дженерик-метасимвол, который мы используем в случаях, когда нет четкого понимания, какой тип данных мы будем использовать. Условно `<?>` можно представить как `<Object>` .

```
// Создадим класс Average,
// который типизируем классом Number и его наследниками
public class Average<T extends Number> {

    // Создадим поле в виде массива
    private T[] array;

    // Сформируем конструктор на основе поля
    public Average(T[] array) {
        this.array = array;
    }

    // Создадим и реализуем метод average, который подсчитывает
    // среднее арифметическое среди всех членов массива
    public double average() {
        double sum = 0.0;

        for (T value : array) {
            sum += value.doubleValue();
        }

        return sum / array.length;
    }
}
```

Предположим, что нам нужно сравнить средние значения массивов в методе `sameAvg()` (дальше нужно будет создать этот метод в классе `Average`) при условии, что типы этих массивов отличаются.

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

      Integer intArray[] = {1, 5, 2, 4, 3};
      Double doubleArray[] = {2.4, 7.3, 4.4, 15.1};

      Average<Integer> aveInt = new Average<>(intArray);
      Average<Double> aveDouble = new Average<>(doubleArray);

      if (aveInt.sameAvg(aveDouble)) {
         System.out.println("are the same.");}
      else {
         System.out.println("differ.");
      }
   }
}
```

Отталкиваясь от того, что мы изучили ранее, и от того, что `Average` — параметризованный класс, кажется очевидной следующая реализация метода `sameAvg()` :

```
boolean sameAvg(Average<T> object) {
    return average() == object.average();
}
```

Но такой код работать не будет.

Такая запись подразумевает использование массивов с идентичными параметрами типов. А в нашем случае типы разнятся. Для того чтобы данный метод корректно работал в наших условиях, необходимо использовать **wildcard**.

```
boolean sameAvg(Average<?> object) {
    return average() == object.average();
}
```

Wildcard можно ограничивать так же, как мы изучали ранее. Только вместо записи `<T extends SomeSuperClass>` можно использовать запись `<? extends SomeSuperClass>` .

Например:

```
public static void test(SomeClass <? extends SomeSuperClass> obj) {
    ... // Реализация метода
}

class SomeYoungestClass {
    ... // Реализация класса
}

class SomeClass<T> {
    ... // Реализация класса
}
```

Если же у нас есть необходимость ограничить параметр типа классом-потомком, тогда нам нужно использовать ключевое слово super.

```
public static void test(SomeClass <? super SomeYoungestClass> obj) {
    ... // Реализация метода
}

class SomeYoungestClass {
    ... // Реализация класса
}

class SomeClass<T> {
    ... // Реализация класса
}
```

В данном случае нашим параметром типа может быть любой класс-предок класса SomeYoungestClass, в том числе и класс Object, включая его самого.

Параметризованные методы и конструкторы

Не только классы и интерфейсы могут быть параметризованными. У нас есть возможность использовать параметр типа еще и в **методах** параметризованного или непараметризованного класса.

Мы можем создать метод, который будет параметризован одним или несколькими параметрами типа.

```
public class SomeMethodTest {

   public static <T, V> boolean isContain(T obj, V[] array) {
      for (V value : array) {
         if (obj.equals(value)) {
            return true;
         }
      }
      return false;
   }

   public static void main(String[] args) {
      Integer[] intArray = {1, 7, 2, 14, 9};

      if (isContain(2, intArray)) {
         System.out.println("2 входит в массив intArray");
      }

      if (!isContain(8, intArray)) {
         System.out.println("8 не входит в intArray");
      }
      System.out.println();

      String[] strArray = {"one", "four", "five"};

      if (isContain("five", strArray)) {
         System.out.println("five входит в массив strArray");
      }

      if (!isContain("ten", strArray)) {
         System.out.println("ten не входит в массив strArray");
      }
   }
}
```

Параметризованными могут быть даже конструкторы.

```
public class SomeConstructorClassEx {
   private double value;

   public <T extends Number> SomeConstructorClassEx(T arg) {
      value = arg.doubleValue();
   }

   public void printValue() {
      System.out.println("value: " + value);
   }
}

public class SomeConstructorClassExTest {
   public static void main(String[] args) {
      SomeConstructorClassEx constr1 = new SomeConstructorClassEx(100);
      SomeConstructorClassEx constr2 = new SomeConstructorClassEx(123.5F);

      constr1.printValue();
      constr2.printValue();
   }
}
```

Параметризованные интерфейсы

Интерфейсы также могут быть параметризованными. Логика реализации дженериков в интерфейсах не отличается от их реализации в классах.

```
public interface SomeInterface<T> {
   T someMethod(T t);
}

public class SomeClass<T> implements SomeInterface<T> {
   @Override
   public T someMethod(T t) {
      return t;
   }

   public static void main(String[] args) {
      SomeInterface<String> obj = new SomeClass<>();
      String string = obj.someMethod("some string");
   }
}
```

Иерархии параметризованных классов

Так же как и непараметризованные классы, классы, объявленные с параметром типа, могут строить иерархию. Разница в том, что параметр должен передаваться по всем поколениям классов.

```
public class SomeSuperClass<T> {
   private T obj;

   public SomeSuperClass(T obj) {
      this.obj = obj;
   }

   private T getObj() {
      return obj;
   }
}

public class SomeYoungerClass<T> extends SomeSuperClass<T> {
   public SomeYoungerClass(T obj) {
      super(obj);
   }
}
```

Подкласс параметризованного суперкласса необязательно должен быть параметризованным. Но в нем должны быть указаны параметры типа, требующиеся его параметризованному суперклассу.

Подкласс может быть дополнен своими параметрами типа, если требуется.

Суперклассом для параметризованного класса может быть непараметризованный класс.