

Функциональное программирование в Java. Часть 1

Даниил Пилипенко

Директор центра подбора
IT-специалистов SymbioWay

Skillbox

Дженерики

Java generics (обобщённые типы) в Java

- 1 Обобщённые типы (классы и интерфейсы)
- 2 Обобщённые методы
- 3 Обобщённые конструкторы

Коллекции в Java.

Пример использования обобщённых типов

- `List<String> list = new ArrayList<>()`
- `Set<Integer> set = new HashSet<>()`
- `Map<String, String> map = new HashMap<>()`

Объявляем дженерик-классы и создаём их экземпляры



Пример кода

```
class Box<T> { // обозначение типа - T
    // переменная с типом T
    private T item;

    //параметр метода типа T
    public void putItem(T item) {
        this.item = item;
    }

    // возвращает объект типа T
    public T getItem() {
        return item;
    }
}
```

Создание параметризованных объектов класса

Пример полной записи:

```
class Paper {}  
Box<Paper> boxForPaper = new Box<Paper>();
```

Пример короткой записи:

```
Box<Paper> boxForPaper = new Box<>();
```

Рекомендации Oracle по названию параметров типов

- **E** — **element**, для элементов параметризованных коллекций
- **K** — **key**, для ключей map-структур
- **V** — **value**, для значений map-структур
- **N** — **number**, для чисел
- **T** — **type**, для обозначения типа параметра в произвольных классах
- **S**, **U**, **V** и так далее — применяются, когда в дженерик-классе несколько параметров

Использование различных параметров для создания объектов

```
class Plastic {}  
class Glass {}
```

```
Box<Plastic> boxForPlastic = new Box<>();  
Box<Glass> boxForGlass = new Box<>();
```

Создание классов с несколькими параметрами

```
class TwoCellsBox<T, S> {  
    private T firstCellItem;  
    private S secondCellItem;  
    //...  
}
```

```
TwoCellsBox<Plastic, Glass> plasticGlassBox =  
    new TwoCellsBox<>();
```

Создание параметризованных интерфейсов

```
interface GarbageHandler<T, S> {  
    void handle(T what, S how);  
}
```

Создание параметризованных интерфейсов

```
interface GarbageHandler<T, S> {  
    void handle(T what, S how);  
}
```

```
class MyPaperHandleMethod {  
}  
  
class MyNonGenericPaperHandler implements  
GarbageHandler<Paper, MyPaperHandleMethod> {  
    @Override  
    public void handle(Paper what,  
        MyPaperHandleMethod how) {  
        // здесь что-то делается с бумажным  
        // мусором способом MyPaperHandleMethod  
    }  
}
```

Создание классов с двумя параметрами

```
class GarbageHandlerImpl<T, S>
    implements GarbageHandler<T, S> {
    @Override
    public void handle(T what, S how) {
        // здесь что-то делается с
        // мусором типа T способом S
    }
}
```

Создание классов с двумя параметрами

```
class GarbageHandlerImpl<T, S>
    implements GarbageHandler<T, S> {
    @Override
    public void handle(T what, S how) {
        // здесь что-то делается с
        // мусором типа T способом S
    }
}
```

```
class PaperHandler<T>
    implements GarbageHandler<Paper, T> {
    @Override
    public void handle(Paper what, T how) {
        // здесь что-то делается
        // с бумагой способом T
    }
}
```

Raw type

Можно создавать экземпляры дженерик-типов «без расшифровки», то есть никто не запретит вам объявить переменную типа Box:

```
Box box = new Box<>();
```

Параметризованные методы

У метода **transfer** есть свой личный параметр для типа, который не обязан совпадать ни с типом **T**, ни с типом **S**.

При первом упоминании новый параметр, как и в случае с заголовком класса или интерфейса, пишется в угловых скобках:

```
interface GarbageHandler<T, S> {  
    void handle(T what, S how);  
    <E> void transfer(E dangerousWaste);  
}
```


Параметризованные методы

Дженерик-методы можно объявлять и в обычных (не дженерик) классах и интерфейсах. Ваш класс для переработки мог бы выглядеть так. Здесь дженерики используются только в методе.

Обратите внимание на синтаксис: параметры типов объявляются после модификатора доступа (**public**), но перед возвращаемым типом (**void**). Они перечисляются через запятую в общих угловых скобках:

```
class GarbageHandlerImpl {  
  
    public <T, S> void handle(T what, S how) {  
        // здесь что-то делается  
        // с мусором типа T способом S  
    }  
}
```

Ограничение дженериков сверху и снизу

```
abstract class Garbage{
    public abstract double getWeight();
}

class Paper extends Garbage{
    @Override
    public double getWeight() {
        return 0.01;
    }
}

class Plastic extends Garbage{
    @Override
    public double getWeight() {
        return 0.3;
    }
}
```

Ограничение дженериков сверху и снизу

```
class Box<T> {  
  
    private T item;  
  
    public double getItemWeight() {  
        // не скомпилируется  
        return item == null ? 0 :  
            item.getWeight();  
    }  
    // ... остальные методы  
}
```

Ограничение дженериков сверху и снизу

```
class Box<T extends Garbage> {  
    // методы класса  
}
```

Ограничение дженериков сверху и снизу

Для одного класса или интерфейса можно добавить сразу несколько ограничений. Вспомните про интерфейс для пункта приёма мусора и введите класс для метода переработки — **HandleMethod**.

Тогда **GarbageHandler** можно переписать так:

```
class HandleMethod {  
  
}  
  
interface GarbageHandler<T extends Garbage,  
    S extends HandleMethod> {  
    void handle(T what, S how);  
}
```

Wildcards

Wildcards нельзя подставлять везде, где до этого были написаны буквенные обозначения. Не получится, например, объявить класс **Box<?>** или дженерик-метод, который принимает такой тип:

```
class Box<?>{ // не скомпилируется
    ? variable; // не скомпилируется
    public <?> void someMethod(? param) {
        // не скомпилируется
        //...
    }
}
```

Wildcards

В этом примере вы можете подставить вместо «?» любой тип, в том числе Paper, поэтому строка успешно скомпилируется:

```
List<?> example1 =  
    new ArrayList<Paper>();
```

Wildcards

Это уже знакомое вам ограничение сверху, upper bounding, — вместо «?» допустим Garbage или любой его класс-наследник, то есть Paper подходит.

```
List<? extends Garbage> example2 =  
    new ArrayList<Paper>();
```


Wildcards

Ограничение сверху, upper bounding, — вместо «?» допустим `Garbage` или любой его класс-наследник, то есть `Paper` подходит.

Но можно ограничить тип и снизу. Это называется lower bounding.

Здесь `<? super Garbage>` означает, что вместо «?» можно подставить `Garbage` или любой класс-предок `Garbage`.

Все ссылочные классы неявно наследуют класс `Object`, так что в правой части ещё может быть `ArrayList<Object>`.

```
List<? super Garbage> example3 =  
    new ArrayList<Garbage>();
```

Выводы

Вы узнали, что **дженерики в Java** — это **механизм, который позволяет работать с типом класса** как с переменной.

Они используются для создания классов, интерфейсов и методов, которые могут работать с различными типами данных. Некоторые примеры их использования были представлены в данном материале.