УДК 004.9

КОЛЛЕКЦИИ В JAVA 8

Земляков Геннадий Евгеньевич

студент
Самарский государственный аэрокосмический университет, Самара

author@apriori-journal.ru

Аннотация. В данной статье рассматриваются нововведения языка программирования Java 8 (методы по умолчанию в интерфейсах и функциональные интерфейсы, лямбда-выражения, пакет java.util.stream), а также их применение в коллекциях.

Ключевые слова: Java 8; коллекции; лист; итерация; лямбда-выражение.

COLLECTIONS IN JAVA 8

Zemlyakov Gennady Evgenievich

student
Samara State Aerospace University, Samara

Abstract. Java 8 offered new facilities for working with collections (default methods in interfaces and functional interfaces, lambda-expressions, java.util.stream package) which made code concise, expressive and elegant.

Key words: Java 8; collections; list; iteration; lambda-expression.

Коллекции применяются для хранения чисел, строк или объектов. Они настолько часто используются, что даже небольшое уменьшение количества кода необходимого для их обработки приводит к значительному сокращению всей программы. Перебор элементов коллекции, преобразовании коллекции, поиск элементов в коллекции являются наиболее часто применяемыми операциями при работе с коллекциями. Рассмотрим данные операции с применением новых средств, ставших доступными с выходом восьмой версии языка программирования Java.

Перебор элементов

Перебор элементов коллекции является базовой операцией при работе с ней. Создание коллекции в Java можно осуществить с помощью следующего кода:

```
List<String> languages = Arrays.asList(“Java”, “JavaScript”, “Ruby”, “Pytnon”, “C”);
```

Рассмотрим операцию перебора элементов коллекции в старых версиях Java:

```
for (int i=0; i<languages.length;i++) {
    System.out.println(languages.get(i))
}
```

Java также предлагает более современный подход:

```
for (String language : languages) {
    System.out.println(language);
}
```

Оба подхода являются внешними итераторами. Мы точно контролируем перебор элементов коллекции, сообщая где начинать итерацию и где ее заканчивать.

Рассмотрим недостатки таких подходов:

1) Цикл `for` являются последовательным и его достаточно сложно сделать параллельным;

2) Циклы не являются полиморфными. Мы используем коллекцию в цикле for вместо вызова метода (полиморфная операция) коллекции.

Рассмотрим операцию перебора элементов коллекции с применением внутреннего итератора. Интерфейс Iterable был расширен в Java 8 методом forEach(), который принимает параметр типа Consumer.

```
languages.forEach(new Consumer<String>() {
    public void accept(final String language) {
        System.out.println(language);
    }
});
```

В данном примере мы вызываем метод forEach() коллекции languages, передавая объект анонимного класса Consumer. Метод forEach() вызовет метод accept() объекта Consumer для каждого элемента в коллекции. Преимуществом данного подхода является то, что мы сосредотачиваем внимание не на том как нам делать перебор элементов коллекции, а на том что мы хотим сделать с каждым элементом коллекции. Недостаток данного подхода заключается в том что код выглядит еще более громоздким по сравнению с первыми двумя примерами. Мы можем легко устранить его добавив лямбда-выражения.

```
languages.forEach(language ->
    System.out.println(language));
```

Переменной language присваивается значение каждого элемента коллекции. Библиотеки Java берут на себя контроль за выполнением лямбда – выражений. Однако данный подход имеет ограничение. Если мы вызвали метод forEach() мы не можем прервать его выполнение, в отличие от первых двух примеров.

Рассмотрим последний, еще более короткий пример.

```
friends.forEach(System.out::println);
```

В данном примере мы используем ссылку на метод. Java позволяет нам заменить тело метода его именем.

Лямбда-выражения позволяют осуществлять перебор коллекции, используя более короткий код в отличии от классических подходов.

Преобразование коллекции

Преобразование коллекции является такой же простой операцией как и перебор элементов. Предположим мы хотим преобразовать все буквы строк, содержащихся в коллекции, в верхний регистр. Рассмотрим этот пример.

Объекты строк в Java не могут быть изменены. Мы можем создать новые строки и заменить соответствующие элементы в существующей коллекции. У такого подхода есть два недостатка: оригинальная коллекция будет стерта, сложность реализации параллельных вычислений.

Оптимальным решением будет создание новой коллекции со строками в верхнем регистре.

```
List<String> uppercaseLanguages = new
    ArrayList<String>();

for(String language : languages) {

    uppercaseLanguages.add(language.toUpperCase());

}
```

В этом примере мы создали пустую коллекцию и заполнили ее строками в верхнем регистре, перебирая строки в оригинальной коллекции.

Рассмотрим пример с использованием внутреннего итератора Java 8.

```
List<String> uppercaseLanguages = new
    ArrayList<String>();

languages.forEach(name ->
    uppercaseLanguages.add(language.toUpperCase()));

System.out.println(uppercaseLanguages);
```

Мы использовали внешний итератор, но нам по-прежнему приходится создавать новую коллекцию.

Рассмотрим пример с использованием лямбда-выражений. Метод map() нового интерфейса Stream позволит избежать нам создание коллекции и сделает код короче. Stream похож на итератор коллекции и обеспечивает потоковые функции. Метод map()позволяет преобразовать входящую последовательность.

```
languages.stream().map(language -> name.toUpperCase())
.forEach(language -> System.out.print(language));
```

Метод map() достаточно полезен для преобразования элементов коллекции. В последнем примере мы не создавали новой коллекции и код получился коротким.

Мы можем сделать код еще более коротким используя ссылку на метод. С помощью данного средства мы можем заменить language -> name.toUpperCase() на String::toUpperCase.

Рассмотрим пример:

```
languages.stream().map(String::toUpperCase)
.forEach(language -> System.out.println(language));
```

Лямбда-выражения помогают нам преобразовывать коллекцию без создания новой коллекции, а также делают код более коротким и читабельным.

Поиск элементов

Из списка языков программирования коллекции languages выберем язык назнание которого начинается с буквы “J”. Так как в коллекции может отсутствовать элемент начинающийся с буквы “J”, то результатом поиска может быть пустая коллекция. Рассмотрим пример с применением старых средств.

```
List<String> startsWithJ = new ArrayList<String>();
for(String language : languages) {
    if(language.startsWith("J")) {
        startsWithN.add(language);
    }
}
```

В данной примере мы создаем пустую коллекцию `startsWithJ`. Затем мы перебираем элементы исходной коллекции `languages` в поисках строки начинающейся с буквы “J”. В случае если мы находим такую строку, мы добавляем ее в коллекцию `startsWithJ`.

Изменим предыдущий пример, добавив вызов метода `filter()` интерфейса `Stream`.

```
List<String> startsWithJ =
     languages.stream().filter(language ->
     language.startsWith("J"))
     .collect(Collectors.toList());
```

Метод `filter()` принимает в качестве параметра лямбда-выражение, возвращающее `boolean` результат. Если лямбда-выражение возвращает `true` элемент в контексте лямбда-выражения добавляется в результирующую коллекцию. В итоге метод возвращает `Stream` содержащий элементы для которых лямбда-выражение вернуло `true`.

Применение лямбда-выражений делает Java код более коротким, выразительным и открывает широкие возможности для дальнейших улучшений кода.

Список использованных источников

1. Venkat Subramaniam. Functional Programming in Java. The Pragmatic Bookshelf. Dallas, 2014.
2. Richard Warburton. Java 8 Lambdas: Pragmatic Functional Programming. O'Reilly Media, 2014.
3. Cay S. Horstmann. Java SE 8 for the Really Impatient. Addison-Wesley. NJ, 2014.