Consumer<T>, IntConsumer

Интерфейс: `java.util.function.Consumer`

Интерфейс: `java.util.function.IntConsumer`

Описание

Используются для передачи в метод блока кода, который принимает один параметр и ничего не возвращает. Таким образом, эти интерфейсы подходят для описания методов с семантикой "применить указанное действие к каждому значению из определённого набора" — например, при обходе списков, деревьев и других структур данных.

Интерфейс `Consumer<T>` используется для описания действия, применяемого к значениям указанного параметром интерфейса объектного типа `T`. Например, `Consumer<String>` выполняет определённое действие со строковым значениями.

Интерфейс `IntConsumer` описывает действие, применяемое к значениям типа `int`. Существуют также аналогичные интерфейсы `LongConsumer` и `DoubleConsumer`, работающие с примитивными типами `long` и `double` соответственно.

Объявление лямбда-выражения

Семейство интерфейсов `Consumer` является особой разновидностью интерфейсов, называемых *функциональными интерфейсами*; это значит, что у них объявлен только один абстрактный метод. Приятным свойством функциональных интерфейсов является то, что они могут быть реализованы не только полноценными классами, но и лямбда-выражениями — причём совершенно прозрачно для методов, в которые они передаются.

Понятно, что объявлять на каждое действие типа `Consumer` полноценный класс было бы как минимум громоздко. На помощь приходят лямбда-выражения, которые решают одну конкретную проблему: *как передать в чужой код свой блок кода, вызываемый из чужого кода*.

В синтаксисе лямбда-выражений в общем случае есть несколько тонкостей, на которых мы сейчас останавливаться не будем. Сейчас покажем лишь, как объявлять конкретную разновидность лямбда-выражений, совместимых с семейством интерфейсов `Consumer`.

```
IntConsumer printer = x -> {
    System.out.println("Число: " + x);
};

IntStream.of(13, 42, 666).forEach(printer);
```

Результат:

```
Число: 13
Число: 42
```

```
Число: 666
```

Объявление такого лямбда-выражения состоит из трёх частей:

1. Имя параметра, под которым он будет доступен внутри блока кода. В нашем случае это `x`, но, как и для параметров методов, это может быть произвольный идентификатор.
2. Стрелка, составленная из двух символов `->`.
3. Тело лямбда-выражения — сам блок кода, ограниченный фигурными скобками `{ }`, внутри которого реализуются производимые с параметром действия.

Важно! Поскольку присваивание лямбда-выражения — это инструкция, в этом случае после закрывающей фигурной скобки должна стоять точка с запятой. Но если лямбда-выражение используется внутри другого выражения (например, вызова метода с параметром — лямбда-выражением), точка с запятой уже не ставится.

Лямбда-выражение, присвоенное переменной типа `IntConsumer` — это вполне себе полноценный объект, реализующий интерфейс `IntConsumer`.

Аналогично объявляется и лямбда-выражение типа `Consumer<T>`:

```
Consumer<String> printer = str -> {
    System.out.print(str.toUpperCase() + " ");
};

Arrays.asList("Hello", "World").forEach(printer);
// HELLO WORLD
```

Внутри лямбда-выражения доступны все поля класса, в методе которого оно объявлено:

```
public class StringPrinter {
    private String adjective = "beautiful";

    public void doPrint() {
        List<String> list = Arrays.asList("Hello world", "Goodbye world");

        Consumer<String> action = str -> {
            String inserted = str.replace(" ", " " + adjective + " ");
            System.out.println(inserted);
        };

        list.forEach(action);
    }
}
```

Результат:

```
Hello beautiful world
Goodbye beautiful world
```

Более того, в лямбда-выражении можно даже использовать локальные переменные метода:

```
String greeting = "Hello ";

Consumer<String> greeter = name -> {
```

```
    System.out.println(greeting + name);
};

Arrays.asList("Alice", "Bob").forEach(greeter);
```

Результат:

```
Hello Alice
Hello Bob
```

На локальные переменные, в отличие от полей класса, накладывается ограничение: им должно присваиваться значение один и только один раз во всём методе, как если бы они были объявлены как `final`. Следующий код не скомпилируется:

```
String greeting = "Hello ";

Consumer<String> greeter = name -> {
    System.out.println(greeting + name);
};

greeting = "Goodbye "; // Ошибка компиляции
```

Конечно, это ограничение не распространяется на локальные переменные, объявленные внутри самого лямбда-выражения (вроде переменной `inserted` в примере с классом `StringPrinter`).

Важно! Хотя это и не обязательно, считается правилом хорошего тона объявлять все локальные переменные метода, используемые в лямбда-выражении, как `final`, чтобы подчеркнуть невозможность переприсваивания.

Прямая передача лямбда-выражения в метод

Итак, любой объект типа `Iterable<T>` (включая все реализации `List`, такие, как `ArrayList`) содержит метод `forEach`, принимающий единственный параметр типа `Consumer`:

```
void forEach(Consumer<? super T> action)
```

Здесь запись `<? super T>` означает "тип `T` или любой его супертип" — то есть тип `T`, любой его суперкласс сколь угодно высоко по иерархии или любой интерфейс, реализованный классом T.

Иначе говоря, в метод `List<String>.forEach` мы можем передать значение типа `Consumer<String>`, или `Consumer<Object>`, или `Consumer<CharSequence>`, и так далее.

Аналогично в классе `IntStream` есть метод:

```
void forEach(IntConsumer action)
```

Здесь наше действие, передаваемое аргументом метода, работает не с объектом, а с числом типа `int`.

Присваивать лямбда-выражение переменной необязательно. Можно объявить его напрямую внутри вызова метода, принимающего объект функционального интерфейса:

```
IntStream.of(1, 2, 3, 4).forEach(i -> {
    System.out.print((i * i) + " ");
});
// 1 4 9 16
```

Обратите внимание, что сначала пишется фигурная скобка, закрывающая тело лямбда-выражения, а затем — круглая скобка, закрывающая вызов метода.

Важно! Внутри лямбда-выражения инструкция `return` имеет несколько другой смысл, чем внутри метода. Она возвращает управление *из тела лямбда-выражения*, а не из метода, в котором оно объявлено. Таким образом, если передать лямбда-выражение в метод `forEach`, инструкция `return` внутри него будет работать аналогично `continue` при использовании цикла:

```
// Печатаем только чётные числа
int[] lost = { 4, 8, 15, 16, 23, 42 };

IntStream.of(lost).forEach(i -> {
    if (i % 2 != 0) {
        return;  // переходим к следующему элементу
    }

    System.out.print(i + " ");
});

// Это аналогично циклу...
for (int i : lost) {
    if (i % 2 != 0) {
        continue;  // переходим к следующему элементу
    }

    System.out.print(i + " ");
}
```

Результат в обоих случаях один и тот же:

```
4 8 16 42
```

Использование объектов `Consumer` и `IntConsumer`

Лямбда-выражения — это синтаксическая конструкция, не влияющая на само использование функциональных интерфейсов. Коду, использующему интерфейс, совершенно всё равно, как он был реализован: с помощью лямбда-выражения или обычного класса.

`Consumer<T>` — это обычный интерфейс с одним абстрактным методом:

```
void accept(T t)
```

Аналогично и `IntConsumer` имеет один абстрактный метод:

```
void accept(int value)
```

При использовании лямбда-выражения в качестве реализации интерфейса `Consumer` или `IntConsumer` оно просто становится реализацией метода `accept`:

```
IntConsumer hexPrinter = i -> {
    System.out.printf("%x\n", i);
};

hexPrinter.accept(255);  // FF
```

```
Consumer<int[]> intArrayPrinter = arr -> {
    System.out.println(Arrays.toString(arr));
};

int[] lost = { 4, 8, 15, 16, 23, 42 };
intArrayPrinter.accept(lost);  // [ 4, 8, 15, 16, 23, 42 ]
```

Аналогичным образом можно использовать и объекты интерфейсов `Consumer` и `IntConsumer`, передаваемые как параметры методов. Для примера рассмотрим стандартную реализацию метода `forEach`. При реализации интерфейса `Iterable` и метода `iterator` мы "бесплатно" получаем также и реализацию метода `forEach` по умолчанию, определённую в объявлении самого интерфейса `Iterable<T>`:

```
default void forEach(Consumer<? super T> action) {
    // если action == null, явно бросаем NullPointerException
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

Как видно, по умолчанию метод `forEach` реализуется через цикл for-each, что логично. При желании её всегда можно переопределить, если реализация класса допускает более эффективную реализацию операции "применить действие `action` для всех элементов" без использования итератора.

В качестве ещё одного примера можно привести обобщённую реализацию рекурсивного обхода дерева файлов в глубину. Более подробно классы `Path` и `Files` будут рассмотрены позже.

```
void walkDir(Path dir, Consumer<Path> action) throws IOException {
    try (DirectoryStream<Path> dirStream = Files.newDirectoryStream(dir)) {
        // Обходим список всех файлов и подкаталогов в каталоге
        for (Path dirEntry: dirStream) {
            if (Files.isDirectory(dirEntry)) {
                // Рекурсия для подкаталогов
                walkDir(dirEntry, action);
            } else {
                // Вызываем action для файлов
                action.accept(dirEntry);
            }
        }
    }

    // Вызываем action для dir в последнюю очередь
    action.accept(dir);
}
```

Один раз реализовав этот алгоритм, мы получаем возможность на его основе писать любые рекурсивные действия над файловой системой — например, рекурсивное удаление каталога вместе со всем содержимым:

```
void deleteDirRecursively(Path dir) {
    walkDir(dir, dirEntry -> {
```

```
        try {
            Files.delete(dirEntry);
        } catch (IOException e) {
            e.printStackTrace();
        }
    });
}

void deleteAll() {
    deleteDirRecursively(Paths.get("C:"));  // Не делайте так :(
}
```

Важно! Показанный здесь пример — учебный. Писать самостоятельно рекурсивный обход каталога не нужно — для этого существуют стандартные и более мощные методы `Files.walk` и `Files.walkFileTree`.