Конспект лекции
Stream API и лямбда выражения

Цель и задачи лекции

Цель – изучить принципы работы Stream API и лямбда выражений.

Задачи:

1. Дать понятие лямбда выражениям и функциональному программированию.
2. Изучить основные принципы работы с Java Stream API.

План занятия

1. Лямбда выражения
2. Stream API

Лямбда выражения

Лямбда-выражения и связанные с ними средства, внедренные в версии JDK 8, значительно усовершенствовали язык Java по следующим причинам. Во-первых, они вводят в синтаксис новые элементы, повышающие выразительную силу языка. Они, по существу, упрощают порядок реализации некоторых общих языковых конструкций. И, во-вторых, внедрение лямбда-выражений позволило наделить новыми возможностями библиотеку прикладного программного интерфейса API. К их числу относится возможность упростить параллельную обработку в многоядерных средах, особенно в циклических операциях, выполняемых в стиле for each, а также в новом прикладном программном интерфейсе API потоков ввода-вывода, где поддерживаются конвейерные операции с данными.

Помимо преимуществ, которые лямбда-выражения вносят в Java, имеется еще одна причина, по которой они являются очень важным дополнением этого языка программирования. За последние несколько лет лямбда-выражения стали предметом главного внимания в области разработки языков программирования. В частности, они были внедрены в такие язык, как C# и C++. А их включение в состав версии JDK8 позволило сохранить живой, новаторский характера языка Java, в которому уже привыкли те, кто на нем программирует.

В конечном итоге лямбда-выражения изменили современный вид языка Java таким же образом, как это сделали обобщения несколько лет назад. Проще говоря, внедрение лямбда-выражений окажет влияние практически на всех программирующих на Java. И поэтому они действительно являются очень важным нововведением в Java.

Лямбда-выражение, по существу, является анонимным (т.е. безымянным) методом. Но этот метод не выполняется самостоятельно, а служит для реализации метода, определяемого в

функциональном интерфейсе. Таким образом, лямбда-выражение приводит к некоторой форме анонимного класса. Нередко лямбда-выражения называют также замыканиями.

Функциональным называется такой интерфейс, который содержит один и только один абстрактный метод. Как правило, в таком методе определяется предполагаемое назначение интерфейса. Следовательно, функциональный интерфейс представляет единственное действие. Например, стандартный интерфейс Runnable является функциональным, поскольку в нем определяется единственный метод run(), который, в свою очередь, определяет действие самого интерфейса Runnable. Кроме того, в функциональном интерфейсе определяется целевой тип лямбда-выражения. В связи с этим необходимо подчеркнуть следующее: лямбда-выражение можно использовать только в том контексте, в котором определен его целевой тип. И еще одно замечание: функциональный интерфейс иногда еще называют SAM-типом, где SAM обозначает Single Abstract Method - единственный абстрактный метод.

В функциональном интерфейсе можно определить любой открытый метод, определенный в классе Object, например, метод equals(), не воздействуя на состояние его функционального интерфейса, поскольку они автоматически реализуются экземпляром функционального интерфейса.

Основные положения о лямбда-выражениях

Лямбда-выражение вносит новый элемент в синтаксис и оператор в язык Java. Этот новый оператор называется лямбда-оператором, или операцией "стрелка")(->). Он разделяет лямбда выражение на две части. В левой части указываются любые параметры, требующиеся в лямбда-выражении. (Если же параметры не требуются, то они указываются пустым списком.) А в правой части находится тело лямбда-выражения, где указываются действия, выполняемые лямбда-выражением. Операция -> буквально означает "становиться" или "переходить".

В Java определены две разновидности тел лямбда-выражений. Одна из них состоит из единственного выражения, а другая - из блока кода. Рассмотрим сначала лямбда-выражения, в теле которых определяется единственное выражение. А лямбда-выражения с блочными телами обсудим далее в этой главе.

Прежде чем продолжить дальше, имеет смысл обратится к некоторым примерам лямбда-выражений. Рассмотрим сначала самое простое лямбда-выражение, какое только можно написать. В приведенном ниже лямбда-выражении вычисляется значение константы.

```
() -> 123.45
```

Это лямбда-выражение не принимает никаких параметров, а следовательно, список его параметров оказывается пустым. Оно возвращает значение константы 123,45. Следовательно, это выражение аналогично вызову следующего метода:

```
double myMeth() { return 123.45 }
```

Разумеется, метод, определяемый лямбда-выражением, не имеет имени. Ниже приведено более интересное лямбда-выражение.

```
() -> Math.random() * 100
```

В этом лямбда-выражении из метода Math.random() получается псевдослучайное значение, которое умножается на 100 и затем возвращается результат. И это лямбда-выражение не требует параметров. Если же лямбда-выражению требуются параметры, они указываются списком в левой части лямбда-оператора. Ниже приведен простой пример лямбда-выражения с одним параметров.

```
(n) -> (n % 2)==0
```

Это выражение возвращает логическое значение true, если числовое значение параметра n оказывается четным. Тип параметра (в данном случае n) можно указывать явно, но зачастую в этом нет никакой нужды, поскольку его тип в большинстве случаев выводится. Как и в именованном методе, в лямбда-выражении можно указывать столько параметров, сколько требуется.

Функциональные интерфейсы

Как пояснялось ранее, функциональным называется такой интерфейс, в котором определяется единственный абстрактный метод. Те, у кого имеется предыдущий опыт программирования на Java, могут возразить, что все методы интерфейса неявно считаются абстрактными, но так было до внедрения лямбда-выражений. Как пояснялось в главе 9, начиная с версии JDK 8 для метода, объявляемого в интерфейсе, можно определить стандартное поведение по умолчанию, и поэтому он называется методом по умолчанию. Отныне интерфейсный метод считается абстрактным лишь в том случае, если у него отсутствует реализация по умолчанию. А поскольку интерфейсные методы, не определяемые по умолчанию, неявно считаются абстрактными, то их не обязательно объявлять с модификатором доступа abstract, хотя это и можно сделать при желании.

Ниже приведен пример объявления функционального интерфейса.

```
interface MyNumber {

      double getValue();

}
```

В данном случае метод getValue() неясно считается абстрактным и единственным определяемым в интерфейсе MyNumber. Следовательно, интерфейс MyNumber является функциональным, а его функция определяется методом getValue().

Как упоминалось ранее, лямбда-выражение не выполняется самостоятельно, а скорее образует реализацию абстрактного метода, определенного в функциональном интерфейсе, где указывается его целевой тип. Таким образом, лямбда-выражение может быть указано только в том контексте, в котором определен его целевой тип. Один из таких контекстов создается в том случае, когда лямбда-выражение присваивается ссылке на функциональный интерфейс. К числу других контекстов целевого типа относятся инициализация переменных, операторы return и аргументы методов.

Рассмотрим пример, демонстрирующий применение лямбда-выражения в контексте присваивания. С этой целью сначала объявляется ссылка на функциональный интерфейс MyNumber, как показано ниже.

```
// создать ссылку на функциональный интерфейс MyNumber

MyNumber myNum;
```

Затем лямбда-выражение присваивается этой ссылке на функциональный интерфейс следующим образом:

```
// использовать лямбда-выражение в контексте присваивания

myNum = () -> 123.45;
```

Когда-лямбда-выражение появляется в контексте своего целевого типа, автоматически создается экземпляр класса, реализующего функциональный интерфейс, причем лямбда-выражение определяет поведение абстрактного метода, объявляемого в функциональном интерфейсе. А когда этот метод вызывается через свой адресат, выполняется лямбда-выражение. Таким образом, лямбда-выражение позволяет преобразовать сегмент кода в объект.

В предыдущем примере лямбда-выражение становится реализацией метода getValue(). В итоге получается значение константы 123.45, которое выводится на экран следующим образом:

```
// вызвать метод getValue(), реализуемый

// присвоенным ранее лямбда-выражением

System.out.println(myNum.getValue());
```

Лямбда-выражение было ранее присвоено переменной myNum ссылки функциональный интерфейс MyNumber. Оно возвращает значение константы 123,45, которое получается в результате вызова метода getValue().

Для того чтобы лямбда-выражение использовалось в контексте своего целевого типа, абстрактный метод и лямбда-выражение должны быть совместимыми по типу. Так, если в абстрактном методе указываются два параметра типа int, то и в лямбда-выражении должны быть указаны два параметра, тип которых явно обозначается как int или неявно выводится как int из самого контекста. В общем, параметры лямбда-выражения должны быть совместимыми по типу и количеству с параметрами абстрактного метода. Это же относится и к возвращаемым типам. А любые исключения, генерируемые в лямбда-выражении, должны быть приемлемы для абстрактного метода.

Некоторые использования лямбда-выражений

Принимая во внимание все сказанное выше, рассмотрим ряд простых примеров, демонстрирующих основные принципы действия лямбда-выражений. В первом примере программы все приведенные ранее фрагменты кода собраны в единое целое:

```
// Функциональный интерфейс
interface MyNumber {
    double getValue();
}
```

```
class LambdaDemo {
    public static void main(String args[]) {
        MyNumber myNum; // объявить ссылку на функциональный интерфейс

        // Здесь лямбда-выражение просто является константным выражением.
        // Когда оно присваивается ссылочной переменной myNum, получается
        // экземпляр класса, в котором лямбда-выражение реализует
        // метод getValue() из функционального интерфейса MyNumber
        myNum = () -> 123.45;

        // вызвать метод getValue(), предоставляемый
        // присвоенным ранее лямбда-выражением
        System.out.println("Фиксированное значение: " + myNum.getValue());

        // А здесь используется более сложное выражение
        myNum = () -> Math.random() * 100;

        // В следующих строках кода вызывается лямбда-выражение
        // из предыдущей строки кода
        System.out.println("Случайное значение: " + myNum.getValue());
        System.out.println("Еще одно случайное значение: " + myNum.getValue());

        // Лямбда-выражение должно быть совместимо с абстрактным методом,
        // определяемым в функциональном интерфейсе. Поэтому следующая
        // строка кода ошибочна:
        // myNum = () -> "123.03"; // ОШИБКА!
    }
}
```

Как упоминалось ранее, лямбда-выражение должно быть совместимо по типу с абстрактным методом, для реализации которого оно предназначено. Именно поэтому последняя строка кода в приведенном выше примере закомментирована. Ведь значение типа String несовместимо с типом double, возвращаемым методом getValue().

В следующем примере программы демонстрируется применение лямбда-выражения с параметром:

```
//функциональный интерфейс
interface NumericTest {
    boolean test(int n);
}

class LambdaDemo2 {
    public static void main(String args[]) {
        NumericTest isEven = (n) -> (n % 2)==0;

        if(isEven.test(10)) System.out.println("Число 10 четное");
        if(!isEven.test(9)) System.out.println("Число 9 нечетное");

        // А теперь воспользоваться лямбда-выражением, в котором
        // проверяется, является ли число неотрицательным
        NumericTest isNonNeg = (n) -> n >= 0;

        if(isNonNeg.test(1)) System.out.println("Число 1 неотрицательное");
        if(!isNonNeg.test(-1)) System.out.println("Число -1 отрицательное");
    }
}
```

В данном примере программы демонстрируется главная особенность лямбда выражений, требующая более подробного рассмотрения. Обратите особое внимание на лямбда-выражение, выполняющее проверку на равенство:

```
(n) -> (n % 2)==0;
```

Обратите внимание на то, что тип переменной n не указан, но выводится из контекста. В данном случае тип переменной n выводится из типа int параметра метода test(), определяемого в функциональном интерфейсе NumericTest. Впрочем, ничто не мешает явно указать тип параметра в лямбда-выражении. Например, следующее лямбда-выражение так же достоверно, как и предыдущее:

```
(int n) -> (n % 2)==0
```

где параметр n явно указывается как int. Как правило, явно указывать тип параметров лямбда-выражений необязательно, хотя в некоторых случаях это может все же потребоваться.

В данном примере программы демонстрируется еще одна важная особенность лямбда-выражений. Ссылка на функциональный интерфейс может быть использована для выполнения любого совместимого с ней лямбда-выражения. Обратите внимание на то, что в данной программе определяются два разных лямбда-выражения, совместимых с методом test() из функционального интерфейса NumericTest. В первом лямбда-выражении isNonNeg проверяется, является ли числовое значение отрицательным, а во втором лямбда-выражении isNonNeg - является ли оно отрицательным. Но в любом случае проверяется значение параметра n. А поскольку каждое из этих лямбда-выражений совместимо с методом test(), то оно выполняется по ссылке на функциональный интерфейс NumericTest.

Прежде чем продолжить дальше, следует сделать еще одно замечание. Если у лямбда-выражения имеется единственный параметр, его совсем не обязательно заключать в круглые скобки в левой части лямбда-оператора. Например, приведенный ниже способ написания лямбда-выражения также допустим в программах.

```
n -> (n % 2)==0
```

Ради согласованности в представленных далее примерах программ списки параметров всех лямбда-выражений заключаются в круглые скобки - даже если они содержат единственный параметр. Разумеется, вы вольны выбрать тот способ указания параметров лямбда-выражений, который вам больше по душе.

В приведенном ниже примере программы демонстрируется лямбда-выражение, принимающее два параметра. В данном случае в лямбда-выражении проверяется, является ли одно число множителем другого.

```
interface NumericTest2 {
    boolean test(int n, int d);
}

class LambdaDemo3 {
    public static void main(String args[]) {

        // В этом лямбда-выражении проверяется, является ли
        // одно число множителем другого
        NumericTest2 isFactor = (n, d) -> (n % d) == 0;


        if(isFactor.test(10, 2))
            System.out.println("Число 2 является множителем числа 10");
```

```
        if(!isFactor.test(10, 3))
            System.out.println("Число 3 не является множителем числа 10");
    }
}
```

В данном примере программы метод test() определяется в функциональном интерфейсе NumericTest2 следующим образом:

```
boolean test(int n, int d);
```

При объявлении метода test() указываются два параметра. Следовательно, в лямбда-выражении, совместимом с методом test(), следует также указать два параметра. Ниже показано, как это делается.

```
(n, d) -> (n % d) == 0
```

Оба параметра, n и d, указываются списком через запятую. Данный пример можно обобщить. Всякий раз, когда в лямбда-выражении требуется больше одного параметра, их следует указать списком через запятую, заключив в круглые скобки в левой части лямбда-оператора.

Следует, однако, иметь в виду, что если требуется явно объявить тип одного из параметров лямбда-выражения, то это следует сделать и для всех остальных параметров. Например, следующее лямбда-выражение достоверно:

```
(int n, int d) -> (n % d) == 0
```

А это лямбда-выражение недостоверно:

```
(int n, d) -> (n % d) == 0
```

Блочные лямбда-выражения

Тело лямбда-выражений в предыдущих примерах состояло из единственного выражения. Такая разновидность тел называется телом выражения, а лямбда-выражения с телом выражения иногда еще называют одиночными. В теле выражения код, указываемый в правой части лямбда-оператора, должен состоять из одного выражения. Несмотря на все удобство одиночных лямбда-выражений, иногда в них требуется вычислять не одно выражение. Для подобных случаев в Java предусмотрена вторая разновидность лямбда-выражений, где код, указываемый в правой части лямбда-оператора, может состоять из нескольких операторов. Такие лямбда-выражения называются блочными, а их тело - телом блока.

Блочное лямбда-выражение расширяет те виды операций, которые могут выполняться в лямбда-выражении, поскольку оно допускает в своем теле наличие нескольких операторов. Например, в блочном лямбда-выражении можно объявлять переменные, организовывать циклы, указывать операторы выбора if и switch, создавать вложенные блоки и т.д. Создать блочное лямбда-выражение совсем не трудно. Для этого достаточно заключить тело выражения в фигурные скобки таким образом, как и любой другой блок кода.

Кроме наличия в теле выражения нескольких операторов, блочные лямбда выражения применяются точно так же, как и упоминавшиеся ранее одиночные лямбда-выражения. Однако для возврата значения из блочных лямбда-выражений нужно явно указывать оператор return. Это нужно делать потому, что тело блочного лямбда-выражения не представляет одиночное выражение.

Ниже приведен пример программы, в котором блочное лямбда-выражение применяется для вычисления и возврата факториала целочисленного значения.

```
interface NumericFunc {
    int func(int n);
}

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

        // Это блочное лямбда-выражение вычисляет
        // факториал целочисленного значения
        NumericFunc factorial = (n) -> {
            int result = 1;

            for(int i=1; i <= n; i++)
                result = i * result;

            return result;
        };

        System.out.println("Факториал числа 3 равен " + factorial.func(3));
        System.out.println("Факториал числа 5 равен " + factorial.func(5));
    }
}
```

В данном примере программы обратите внимание на то, что в блочном лямбда выражении объявляется переменная result, организуется цикл for и указывается оператор return. Все эти действия вполне допустимы в теле блочного лямбда выражения. По существу, тело блока такого выражения аналогично телу метода. Следует также иметь в виду, что когда в лямбда-выражении оказывается оператор return, он просто вызывает возврат из самого лямбда-выражения, но не из объемлющего его метода.

Ниже приведен еще один пример блочного лямбда-выражения. В данном примере программы изменяется на обратный порядок следования символов в строке.

```
interface StringFunc {
    String func(String n);
}

class BlockLambdaDemo2 {
    public static void main(String args[]) {

        // Это блочное выражение изменяет на обратный
        // порядок следования символов в строке
        StringFunc reverse = (str) -> {
            String result = "";
            int i;

            for(i = str.length()-1; i >= 0; i--)
                result += str.charAt(i);

            return result;
        };
```

```
        System.out.println("Лямбда обращается на " + reverse.func("Лямбда"));

        System.out.println("Выражение обращается на " + reverse.func("Выражение"));

    }
}
```

В данном примере программы в функциональном интерфейсе StringFunc объявляется метод func(), принимающий параметр типа String и возвращающий значение типа String. Следовательно, в лямбда-выражении reverse тип параметра str должен быть выведен как String. Обратите внимание на то, что метод charAt() вызывается для параметра str как для объекта. И это вполне допустимо, поскольку этот параметр имеет тип String благодаря выведению типов.

Обобщенные функциональные интерфейсы

Указывать параметры типа в самом лямбда-выражении нельзя. Следовательно, лямбда-выражение не может быть обобщенным. (Безусловно, все лямбда-выражения проявляют в той или иной мере свойства, подобные обобщениям, благодаря выведению типов.) А вот функциональный интерфейс, связанный с лямбда-выражением, может быть обобщенным. В этом случае целевой тип лямбда-выражения отчасти определяется аргументом типа или теми аргументами, которые указываются при объявлении ссылки на функциональный интерфейс.

Чтобы понять и оценить значение обобщенных функциональных интерфейсов, вернемся к двум примерам из предыдущего раздела. В них применялись два разных функциональных интерфейса: NumericFunc и StringFunc. Но в обоих этих интерфейсах был определен метод func(), возвращавший результат. В первом случае - значение типа String. Следовательно, единственное отличие обоих вариантов этого метода состояло в типе требовавших данных. Вместо того чтобы объявлять два функциональных интерфейса, методы которых отличаются только типом данных, можно объявить один обобщенный интерфейс, который можно использовать в обоих случаях. Именно такой подход и принят в следующем примере программы:

```
// Обобщенный функциональный интерфейс
interface SomeFunc<T> {
    T func(T t);
}

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

        // использовать строковый вариант интерфейса SomeFunc
        SomeFunc<String> reverse = (str) -> {
            String result = "";
            int i;

            for(i = str.length()-1; i >= 0; i--)
                result += str.charAt(i);
            return result;
        };

        System.out.println("Лямбда обращается на " +
                reverse.func("Лямбда"));
        System.out.println("Выражение обращается на " +
                reverse.func("Выражение"));

        // а теперь использовать целочисленный вариант интерфейса SomeFunc
        SomeFunc<Integer> factorial = (n) -> {
```

```
            int result = 1;

            for(int i=1; i <= n; n++)
                result = i * result;

            return result;
        };

        System.out.println("Факториал числа 3 равен " + factorial.func(3));
        System.out.println("Факториал числа 5 равен " + factorial.func(5));
    }
}
```

В данном примере программы обобщенный функциональный интерфейс SomeFunc объявляется следующим образом:

```
interface SomeFunc<T> {

      T func(T t);

}
```

где T обозначает как возвращаемый тип, так и тип параметра метода func(). Это означает, что он совместим с любым лямбда-выражением, принимающим один параметр и возвращающим значение того же самого типа.

Обобщенный функциональный интерфейс SomeFunc служит для предоставления ссылки на два разных типа лямбда-выражений. В первом их них используется тип String, а во втором - тип Integer. Таким образом, один и тот же интерфейс может быть использован для обращения к обоим лямбда-выражениям - reverse и factorial. Отличается лишь аргумент типа, передаваемый обобщенному функциональному интерфейсу SomeFunc.

Передача лямбда-выражений в качестве аргументов

Как пояснялось ранее, лямбда-выражение может быть использовано в любом контексте, предоставляющем его целевой тип. Один из таких контекстов возникает при передаче лямбда-выражения в качестве аргумента. В действительности передача лямбда-выражений в качестве аргументов является весьма распространенным примером их применения. Более того, это весьма эффективное их применение, поскольку оно дает возможность передать исполняемый код метода в качестве его аргумента. Благодаря этому значительно повышается выразительная сила языка Java.

Для передачи лямбда-выражения в качестве аргумента параметр, получающий это выражение в качестве аргумента, должен иметь тип функционального интерфейса, совместимого с этим лямбда-выражением. Несмотря на всю простоту применения лямбда-выражений в качестве передаваемых аргументов, полезно все же показать, как это происходит на практике. В следующем примере программы демонстрируется весь этот процесс:

```
interface StringFunc {
    String func(String n);
}

class LambdasAsArgumentsDemo {
```

```
    // Первый параметр этого метода имеет тип функционального
    // интерфейса. Следовательно, ему можно передать ссылку на
    // любой экземпляр этого интерфейса, включая экземпляр,
    // создаваемый в лямбда-выражении. А второй параметр
    // обозначает обрабатываемую символьную строку

    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }

    public static void main(String args[])
    {
        String inStr = "Лямбда-выражения повышают эффективность Java";
        String outStr;

        System.out.println("Это исходная строка: " + inStr);

        // Ниже приведено простое лямбда-выражение, преобразующее
        // в верхний регистр букв все символы исходной строки,
        // передаваемой методу stringOp()
        outStr = stringOp((str) -> str.toUpperCase(), inStr);
        System.out.println("Эта строка в верхнем регистре: " + outStr);

        // А здесь передается блочное лямбда-выражение, удаляющее
        // пробелы из исходной символьной строки
        outStr = stringOp((str) -> {
            String result = "";
            int i;

            for(i = 0; i < str.length(); i++)
                if(str.charAt(i) != ' ')
                    result += str.charAt(i);
            return result;
        }, inStr);

        System.out.println("Это строка с удаленными пробелами: " + outStr);

// Конечно, можно передать и экземпляр интерфейся StringFunc,
// созданный в предыдущем лямбда выражении. Например, после
// следующего объявления ссылка reverse делается на экземпляр
// интерфейса StringFunc
        StringFunc reverse = (str) -> {
            String result = "";
            int i;

            for(i = str.length()-1; i >= 0; i--)
                result += str.charAt(i);
            return result;
        };

        // А теперь ссылку reverse можно передать в качестве первого
        // параметра методу stringOp()
        // since it refers to a StringFunc object.
        System.out.println("Это обращенная строка: " +
                stringOp(reverse, inStr));
    }
}
```

Прежде всего обратите внимание в данном примере программы на метод stringOp(), у которого имеются два параметра. Первый параметр относится к типу StringFunc, т.е. к функциональному интерфейсу. Следовательно, этот параметр может получать ссылку на любой экземпляр функционального интерфейса StringFunc, в том числе и создаваемый в лямбда-выражении. А второй параметр метода, stringOp(), относится к типу String и обозначает

обрабатываемую символьную строку. Затем обратите внимание на первый вызов метода stringOp():

```
outStr = stringOp((str) -> str.toUpperCase(), inStr);
```

где в качестве аргумента данному методу передается простое лямбда-выражение. При этом создается экземпляр функционального интерфейса StringFunc и ссылка на данный объект передается первому параметру метода stringOp(). Таким образом, код лямбда-выражения, встраиваемый в экземпляр класса, передается данному методу. Контекст целевого типа лямбда-выражения определяется типом его параметра. А поскольку лямбда-выражение совместно с этим типом, то рассматриваемый здесь вызов достоверен. Встраивать в метод такие простоты лямбда-выражения, как упомянутое выше, нередко оказывается очень удобно, особенно когда лямбда-выражение предназначается для однократного употребления.

Далее в рассматриваемом здесь примере программы методу stringOp() передается блочное лямбда-выражение. Оно удаляет пробелы из исходной символьной строки и еще раз показано ниже.

```
outStr = stringOp((str) -> {
        String result = "";
        int i;

        for(i = 0; i < str.length(); i++)
            if(str.charAt(i) != ' ')
                result += str.charAt(i);
            return result;
        }, inStr);
```

И хотя здесь указывается блочное лямбда-выражение, описанный процесс передачи лямбда-выражения остается тем же самым и для простого одиночного лямбда-выражения. Но в данном случае некоторым программистам синтаксис может показаться несколько неуклюжим.

Если блочное выражение кажется слишком длинным для встраивания в вызов метода, то его можно просто присвоить переменной ссылки на функциональный интерфейс, как это делалось в предыдущих примерах. И тогда остается только передать эту ссылку вызываемому методу. Такой прием показан в конце рассматриваемого здесь примера программы, где определяется блочное лямбда-выражение, изменяющее порядок следования символов в строке на обратный. Это лямбда-выражение присваивается переменной reverse, ссылающейся на функциональный интерфейс StringFunc. Следовательно, переменную reverse можно передать в качестве аргумента первому параметру метода stringOp(). Именно так и делается в конце данной программы, где методу stringOp() передаются переменная reverse и обрабатываемая символьная строка. Экземпляр, получаемый в результате вычисления каждого лямбда-выражения, является реализацией функционального интерфейса StringFunc, поэтому каждое из этих выражений может быть передано в качестве первого аргумента вызываемому методу stringOp().

И последнее замечание: помимо инициализации переменных, присваивания и передачи аргументов, следующие операции образуют контекст целевого типа лямбда-выражений: приведение типов, тернарная операция ?, инициализация массивов, операторы return, а также сами лямбда-выражения.

Лямбда-выражения и исключения

Лямбда-выражение может генерировать исключение. Но если оно генерирует проверяемое исключение, то последнее должно быть совместимо с исключениями, перечисленными в выражении throws из объявления абстрактного метода в функциональном интерфейсе. Эта особенность демонстрируется в приведенном ниже примере, где вычисляется среднее числовых значений типа double в массиве. А если лямбда-выражению передается массив нулевой длины, то генерируется исключение типа EmptyArrayException. Как следует из данного примера, это исключение перечислено в выражении throws из объявления метода func() в функциональном интерфейсе DoubleNumericArrayFunc.

```
interface DoubleNumericArrayFunc {
    double func(double[] n) throws EmptyArrayException;
}

class EmptyArrayException extends Exception {
    EmptyArrayException() {
        super("Массив пуст");
    }
}

class LamdaExceptionDemo {
    public static void main(String args[]) throws EmptyArrayException {

        double[] values = { 1.0, 2.0, 3.0, 4.0 };

// В этом лямбда выражении вычисляется среднее числовых
// значений типа double в массиве
        DoubleNumericArrayFunc average = (n) -> {
            double sum = 0;

            if(n.length == 0)
                throw new EmptyArrayException();

            for(int i=0; i < n.length; i++)
                sum += n[i];

            return sum / n.length;
        };

        System.out.println("Среднее равно " + average.func(values));

// Эта строка кода приводит к генерированию исключения
        System.out.println("Среднее равно " + average.func(new double[0]));
    }
}
```

В результате первого вызова метода average.func() возвращается среднее значение 2,5. А при втором вызове этому методу передается массив нулевой длины, что приводит к генерированию исключения типа EmptyArrayException. Напомним, что наличие выражения throws в объявлении метода func() обязательно. Без этого программа не будет скомпилирована, поскольку лямбда-выражение перестанет быть совместимым с методом func().

Данный пример демонстрирует еще одну важную особенность лямбда-выражений. Обратите внимание на то, что параметр указываемый при объявлении метода func() в функциональном интерфейсе DoubleNumericArrayFunc, обозначает массив, тогда как параметр лямбда-выражения просто указан как n, а не n[]. Напомним, что тип параметра лямбда-выражения выводится из целевого контекста. В данном случае целевым контекстом является

массив типа double[], поэтому и параметр n[] совсем не обязательно и даже не допустимо. И хотя его можно было бы явно указать как double[] n, это не дало бы в данном случае никаких преимуществ.

Лямбда-выражения и захват переменных

Переменные, определяемые в объемлющей области действия лямбда-выражения, доступны в этом выражении. Например, в лямбда-выражении можно использовать переменную экземпляра или статическую переменную, определяемую в объемлющем его классе. В лямбда-выражении доступен также по ссылке this(явно или неявно) вызывающий экземпляр объемлющего его класса. Таким образом, в лямбда-выражении можно получить или установить значение переменной экземпляра или статической переменной и вызвать метод из объемлющего его класса.

Но если в лямбда-выражении используется локальная переменная из объемлющей его области действия, то возникает особый случай, называемый захватом переменной. В этом случае в лямбда-выражении можно использовать только те локальные переменные, которые действительно являются завершенными. Действительно завершенной считается такая переменная, значение которой не изменяется после ее первого присваивания. Такую переменную совсем не обязательно объявлять как final, хотя это и не считается ошибкой.(Параметр this в объемлющей области действия автоматически оказывается действительно завершенным, а у лямбда-выражений собственный параметр this отсутствует.)

Следует, однако, иметь в виду, что локальная переменная из объемлющей области действия не может быть видоизменена в лямбда-выражении. Ведь это нарушило бы ее действительно завершенное состояние, а следовательно, привело бы к недопустимому ее захвату.

В следующем примере программы демонстрируется отличие действительно конечных переменных от изменяемых локальных переменных:

```
interface MyFunc {
    int func(int n);
}

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

        // Локальная переменная, которая может быть захвачена
        int num = 10;

        MyFunc myLambda = (n) -> {
            // Такое применение переменной num допустимо, поскольку
            // она не видоизменяется
            int v = num + n;

            // Но следующая строка кода недопустима, поскольку в ней
            // предпринимается попытка видоизменить значение num
            // num++;
            return v;
        };

        // И следующая строка кода приведет к ошибке, поскольку в ней
        // нарушается действительно завершенное состояние переменной num
        // num = 9;
    }
}
```

Как следует из комментариев к данному примеру программы, переменная num является действительно завершенной, и поэтому ее можно использовать в лямбда-выражении myLambda. Но если попытаться видоизменить переменную num как в самом лямбда-выражении, так и за его пределами, то она утратит свое действительно завершенное состояние. Это привело бы к ошибке, а программа не подлежала бы компиляции.

Следует особо подчеркнуть, что в лямбда-выражении можно использовать и видоизменять переменную экземпляра из вызывающего его класса. Но нельзя использовать локальную переменную из объемлющей его области действия, если только эта переменная не является действительно завершенной.

Ссылки на методы

С лямбда-выражениями связано еще одно очень важное средство, называемое ссылкой на метод. Такая ссылка позволяет обращаться к методу, не вызывая его. Она связана с лямбда-выражениями потому, что ей также требуется контекст целого типа, состоящий из совместного функционального интерфейса. Имеются разные виды ссылок на методы. Рассмотрим сначала ссылки на статические методы.

Ссылки на статические методы

Для создания ссылки на статический метод служит следующая общая форма:

```
имя_класса::имя_метода
```

Обратите внимание на то, что имя класса в этой форме отделяется от имени метода двумя двоеточиями(::). Этот новый разделитель внедрен в версии JDK 8 специально для данной цели. Такой ссылкой на метод можно пользоваться везде, где она совместима со своим целевым типом.

В следующем примере программы демонстрируется применение ссылки на статический метод:

```
// Функциональный интерфейс для операций с символьными строками
interface StringFunc {
    String func(String n);
}

// В этом интерфейсе определяется статический метод strReverse()
class MyStringOps {
    // Статический метод, изменяющий порядок
    // следования символов в строке
    static String strReverse(String str) {
        int i;
        String result = "";

        for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);

        return result;
    }
}

class MethodRefDemo {
    // В этом методе функциональный интерфейс указывается в качестве
    // типа первого его параметра. Следовательно, ему может быть передан
```

```
    // любой экземпляр этого интерфейса, включая и ссылку на метод
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }

    public static void main(String args[]) {
        String inStr = "Лямбда-выражения повышают эффективность Java";
        String outStr;

        // Здесь ссылка на метод strReverse() передается методу stringOp()
        outStr = stringOp(MyStringOps::strReverse, inStr);

        System.out.println("Исходная строка: " + inStr);
        System.out.println("Обращенная строка: " + outStr);
    }
}
```

В данной программе особое внимание обратите на следующую строку кода:

```
outStr = stringOp(MyStringOps::strReverse, inStr);
```

В этой строке кода ссылка на статический метод strReverse(), объявляемый в классе MyStringOps, передается первому аргументу метода stringOp(). И это вполне допустимо, поскольку метод strReverse() совместим с функциональным интерфейсом StringFunc. Следовательно, в выражении MyStringOps::strReverse() предоставляет реализацию метода func() из функционального интерфейса StringFunc.

Ссылки на методы экземпляра

Для передачи ссылки на метод экземпляра для конкретного объекта служит следующая форма:

```
ссылка_на_объект::имя_метода
```

Как видите, синтаксис этой формы ссылки на метод экземпляра похож на тот, что используется для ссылки на статический метод, за исключением того, что вместо имени класса в данном случае используется ссылка на объект. Ниже приведен переделанный вариант программы из предыдущего примера, чтобы продемонстрировать применение ссылки на метод экземпляра.

```
// Функциональный интерфейс для операций с символьными строками
interface StringFunc {
    String func(String n);
}

// Теперь в этом классе определяется метод экземпляра strReverse()
class MyStringOps {
    String strReverse(String str) {
        String result = "";
        int i;

        for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);
        return result;
    }
}

class MethodRefDemo2 {
    // В этом методе функциональный интерфейс указывается в качестве
```

```
    // типа первого его параметра. Следовательно, ему может быть передан
    // любой экземпляр этого интерфейса, включая и ссылку на метод
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }

    public static void main(String args[]) {

        String inStr = "Лямбда-выражения повышают эффективность Java";
        String outStr;

        // создать объект типа MyStringOps
        MyStringOps strOps = new MyStringOps();

        // А теперь ссылка на метод экземпляра strReverse()
        // передается методу stringOp()
        outStr = stringOp(strOps::strReverse, inStr);

        System.out.println("Исходная строка " + inStr);
        System.out.println("Обращенная строка " + outStr);
    }
}
```

Эта версия программы выводит такой же результат, как и предыдущая ее версия. В данном примере программы обратите внимание на то, что метод strReverse() теперь объявляется в классе MyStringOps как метод экземпляра. А в теле метода main() создается экземпляр strOps класса MyStringOps. Этот экземпляр служит для создания ссылки на свой метод strReverse() при вызове метода stringOp(), как еще раз показано ниже. В данном примере метод экземпляра strReverse() вызывается для объекта strOps.

```
outStr = stringOp(strOps::strReverse, inStr);
```

Возможны и такие случаи, когда требуется указать метод экземпляра, который будет использоваться вместе с любым объектом данного класса, а не только с указанным объектом. В подобных случаях можно создать ссылку на метод экземпляра в следующей общей форме:

```
имя_класса::имя_метода_экземпляра
```

В этой форме имя класса указывается вместо имени конкретного объекта, несмотря на то, что в ней указывается и метод экземпляра. В соответствии с этой формой первый параметр метода из функционального интерфейса совпадает с вызывающим объектом, а второй - с параметром, указанным в методе экземпляра. Рассмотрим пример программы, в которой определяется метод counter(), подсчитывающий количество объектов в массиве, удовлетворяющих условию, определяемому в методе func() из функционального интерфейса MyFunc. В данном случае подсчитываются экземпляры класса HighTemp.

```
// Функциональный интерфейс с методом, принимающим два ссылочных
// аргумента и возвращающим логическое значение
interface MyFunc<T> {
    boolean func(T v1, T v2);
}

// Класс для хранения максимальной температуры за день
class HighTemp {
    private int hTemp;

    HighTemp(int ht) { hTemp = ht; }
```

```
    // возвратить логическое значение true, если вызывающий объект
    // типа HighTemp имеет такую же температуру, как и у объекта ht2
    boolean sameTemp(HighTemp ht2) {
        return hTemp == ht2.hTemp;
    }

    // возвратить логическое значение true, если вызывающий объект
    // типа HighTemp имеет температуру ниже, чем у объекта ht2
    boolean lessThanTemp(HighTemp ht2) {
        return hTemp < ht2.hTemp;
    }
}

class InstanceMethWithObjectRefDemo {

    // Метод, возвращающий количество экземпляров объекта,
    // найденных по критериям, задаваемых параметром
    // функционального интерфейса MyFunc
    static <T> int counter(T[] vals, MyFunc<T> f, T v) {
        int count = 0;

        for(int i=0; i < vals.length; i++)
            if(f.func(vals[i], v)) count++;
        return count;
    }

    public static void main(String args[])
    {
        int count;

        // создать массив объектов типа HighTemp
        HighTemp[] weekDayHighs = {
                new HighTemp(89), new HighTemp(82),
                new HighTemp(90), new HighTemp(89),
                new HighTemp(89), new HighTemp(91),
                new HighTemp(84), new HighTemp(83) };

        // Использовать метод counter() вместе с массивами объектов
        // типа HighTemp. Обратите внимание на то, что ссылка на метод
        // экземпляра sameTemp() передается в качестве второго параметра
        count = counter(weekDayHighs, HighTemp::sameTemp,
                new HighTemp(89));
        System.out.println(
                "Дней, когда максимальная температура была 89: " + count);

        // А теперь создать и использовать вместе с данным
        // методом еще один массив объектов типа HighTemp
        HighTemp[] weekDayHighs2 = {
                new HighTemp(32), new HighTemp(12),
                new HighTemp(24), new HighTemp(19),
                new HighTemp(18), new HighTemp(12),
                new HighTemp(-1), new HighTemp(13) };

        count = counter(weekDayHighs2, HighTemp::sameTemp,
                new HighTemp(12));
        System.out.println(
                "Дней, когда максимальная температура была 12: " + count);

        // А теперь воспользоваться методом lessThanTemp(), чтобы
        // выяснить, сколько дней температура была меньше заданной
        count = counter(weekDayHighs, HighTemp::lessThanTemp,
                new HighTemp(89));
        System.out.println("Дней, когда максимальная температура была меньше 89: " +
count);
```

```
        count = counter(weekDayHighs2, HighTemp::lessThanTemp,
                new HighTemp(19));
        System.out.println("Дней, когда максимальная температура была меньше 19: " +
count);
    }
}
```

В данном примере программы обратите внимание на то, что в классе HighTemp объявлены два метода экземпляра: sameTemp() и lessThanTemp(). Первый метод возвращает логическое значение true, если оба объекта типа HighTemp содержат одинаковую температуру. А второй метод возвращает логическое значение true, если температура в вызывающем объекте меньше, чем в передаваемом. Каждый из этих методов принимает параметр типа HighTemp и возвращает логическое значение. Следовательно, каждый из них совместим с функциональным интерфейсом MyFunc, поскольку тип вызывающего объекта может быть приведен к типу первого параметра метода func(), а тип его аргумента - к типу второго параметра этого метода. Таким образом, когда следующее выражение:

```
HighTemp::sameTemp
```

передается методу counter(), то создается экземпляр функционального интерфейса MyFunc, где тип первого параметра func() соответствует типу объекта, вызывающего метод экземпляра, т.е. типу HighTemp. А тип второго параметра метода func() также соответствует типу HighTemp, поскольку это тип параметра метода экземпляра sameTemp(). Это же справедливо и для метода экземпляра lessThanTemp().

И последнее замечание: используя оператор super, можно обращаться к варианту метода из суперкласса, как показано ниже, где имя обозначает имя вызываемого метода.

Ссылки на обобщенные методы

Ссылками на методы можно также пользоваться для обращения к обобщенным классам и/или методам. В качестве примера рассмотрим следующую программу:

```
// Функциональный интерфейс для обработки массива значений
// и возврата целочисленного результата
interface MyFunc<T> {
    int func(T[] vals, T v);
}

// В этом классе определяется метод countMatching(), возвращающий
// количество элементов в массиве, равных указанному значению.
// Обратите внимание на то, что метод countMatching() является
// обобщенным, тогда как класс MyArrayOps - необобщенным
class MyArrayOps {
    static<T> int countMatching(T[] vals, T v) {
        int count = 0;

        for(int i=0; i < vals.length; i++)
            if(vals[i] == v) count++;

        return count;
    }
}

class GenericMethodRefDemo {
    // В качестве первого параметра этого метода указывается
```

```
    // функциональный интерфейс MyFunc, а в качестве двух других
    // параметров - массив и значение, причем оба типа T
    static <T> int myOp(MyFunc<T> f, T[] vals, T v) {
        return f.func(vals, v);
    }

    public static void main(String args[]) {
        Integer[] vals = { 1, 2, 3, 4, 2, 3, 4, 4, 5 };
        String[] strs = {"Один", "Два", "Три", "Два" };
        int count;

        count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);
        System.out.println("Массив vals содержит " + count + " числа 4");

        count = myOp(MyArrayOps::<String>countMatching, strs, "Два");
        System.out.println("Массив str содержит " + count + " числа два");
    }
}
```

В данном примере программы необобщенный класс MyArrayOps содержит обобщенный метод countMatching(). Этот метод возвращает количество элементов в массиве, совпадающих с указанным значением. Обратите внимание на порядок указания аргумента обобщенного типа. Например, при первом вызове из метода main() этому методу передается аргумент типа Integer следующим образом:

```
count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);
```

Обратите также внимание на то, что происходит после разделителя ::. Этот синтаксис можно обобщить. Когда обобщенный метод указывается как метод экземпляра, его аргумент типа указывается после разделителя :: и перед именем этого метода. Следует, однако, заметить, что явно указывать аргумент типа в данном случае (и во многих других) совсем не обязательно, поскольку тип этого аргумента выводится автоматически. А в тех случаях, когда указывается обобщенный класс, аргумент типа следует после имени этого класса и перед разделителем ::.

Несмотря на то что в предыдущих примерах был продемонстрирован механизм применения ссылок на методы, эти примеры все же не раскрывают в полной мере их преимуществ. Ссылки на методы могут, в частности, оказаться очень полезными в сочетании с каркасом коллекций Collections Framework. И ради полноты изложения ниже приведен краткий, но наглядный пример применения ссылки на метод, чтобы определить наибольший элемент в коллекции.

Обнаружить в коллекции наибольший элемент можно, в частности, вызвав метод max(), определенный в классе Collections. При вызове варианта метода max(), применяемого в рассматриваемом здесь примере, нужно передать ссылку на коллекцию и экземпляр объекта, реализующего интерфейс Comparator<T>. В этом интерфейсе определяется порядок сравнения двух объектов. В нем объявляется единственный абстрактный метод compare(), принимающий два аргумента, имеющих типы сравниваемых объектов. Этот метод должен возвратить числовое значение больше нуля, если первый аргумент больше второго; нулевое значение, если оба аргумента равны; и числовое значение меньше нуля, если первый объект меньше второго.

Прежде для вызова метода max() с двумя определяемыми пользователем объектами экземпляра интерфейса Comparator<T> приходилось получать, реализовав сначала этот интерфейс явным образом в отдельном классе, а затем создав экземпляр данного класса. Далее

этот экземпляр передавался в качестве компаратора методу max(). В версии JDK 8 появилась возможность просто передать методу max() ссылку на сравнение, поскольку в этом случае компаратор реализуется автоматически. Этот процесс демонстрируется ниже на простом примере создания коллекции типа ArrayList объектов типа MyClass и поиска в ней наибольшего значения, определяемого в методе сравнения.

```
class MyClass {
    private int val;

    MyClass(int v) { val = v; }

    int getVal() { return val; }
}

class UseMethodRef {
    // Метод compare(), совместимый с аналогичным методом,
// определенным в интерфейсе Comparator<T>
    static int compareMC(MyClass a, MyClass b) {
        return a.getVal() - b.getVal();
    }

    public static void main(String args[]) {
        List<MyClass> al = new ArrayList<MyClass>();

        al.add(new MyClass(1));
        al.add(new MyClass(4));
        al.add(new MyClass(2));
        al.add(new MyClass(9));
        al.add(new MyClass(3));
        al.add(new MyClass(7));

// Найти максимальное значение, используя метод CompareMC()
        MyClass maxValObj = Collections.max(al, UseMethodRef::compareMC);

        System.out.println("Максимальное значение равно: " + maxValObj.getVal());
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Максимальное значение равно: 9
```

В данном примере программы обратите внимание на то, что в самом классе MyClass не определяется метод сравнения и не реализуется интерфейс Comparator. Тем не менее максимальное значение в списке объектов типа MyClass может быть получено в результате вызова метода max(), поскольку в классе UseMethodRef определяется статический метод compareMC(), совместимый с методом compare(), определенным в интерфейсе Comparator. Таким образом, отпадает необходимость явным образом реализовать и создавать экземпляр интерфейса Comparator.

Ссылки на конструкторы

Ссылки на конструкторы можно создать таким же образом, как и ссылки на методы. Ниже приведена общая форма синтаксиса, которую можно употреблять для создания ссылок на конструкторы.

имя_класса::new

Эта ссылка может быть присвоена любой ссылке на функциональный интерфейс, в котором определяется метод, совместимый с конструктором. Ниже приведен пример применения ссылки на конструктор.

```
// В функциональном интерфейсе MyFunc определяется метод,
// возвращающий ссылку на класс MyClass
interface MyFunc {
    MyClass func(int n);
}

class MyClass {
    private int val;

    // Этот конструктор принимает один аргумент
    MyClass(int v) { val = v; }

    // А этот конструктор по умолчанию
    MyClass() { val = 0; }

    //...

    int getVal() { return val; }
}

class ConstructorRefDemo {
    public static void main(String args[]) {
        // Создать ссылку на конструктор класса MyClass.
        // Метод func() из интерфейса MyFunc принимает аргумент,
        // поэтому оператор new обращает к параметризированному
        // конструктору класса MyCklass, а не к его конструктору по умолчанию
        MyFunc myClassCons = MyClass::new;

        // создать экземпляр класса MyClass по ссылке на его конструктор
        MyClass mc = myClassCons.func(100);

        // Использовать только что созданный экземпляр класса Myclass
        System.out.println("Значение val в объекте mc равно " + mc.getVal());
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Значение val в объекте mc равно 100
```

В данном примере программы обратите внимание на то, что метод func() из интерфейса MyFunc возвращает ссылку на тип MyClass определяются два конструктора. В первом конструкторе указывается параметр типа int, а второй является конструктором по умолчанию и поэтому не имеет параметров. А теперь проанализируем следующую строку кода:

```
MyFunc myClassCons = MyClass::new;
```

В этой строке кода создается ссылка на конструктор класса MyClass в выражении MyClass::new. В данном случае ссылка делается на конструктор MyClass(int v), поскольку метода func() из интерфейса MyFunc принимает параметр типа int, а с ним совпадает именно этот конструктор. Обратите также внимание на то, что ссылка на этот конструктор присваивается переменной myClasCons ссылки на функциональный интерфейс MyFunc. После выполнения данной строки кода переменную myClasCons можно использовать для создания экземпляра

класса MyClass, как показано ниже. По существу, переменная myClasCons предоставляет еще один способ вызвать конструктор MyClass(int v).

```
MyClass mc = myClasCons.func(100);
```

Аналогичным образом создаются ссылки на конструкторы обобщенных классов. Единственное отличие состоит в том, что в данном случае может быть указан аргумент типа. И делается это после имени класса, как и при создании ссылки на обобщенный метод. Создание и применение ссылки на конструктор обобщенного класса демонстрируется на приведенном ниже примере, где функциональный интерфейс MyFunc и класс MyClass объявляются как обобщенные.

```
// Теперь функциональный интерфейс MyFunc обобщенный
interface MyFunc<T> {
    MyClass<T> func(T n);
}

class MyClass<T> {
    private T val;

    // Этот конструктор принимает один аргумент
    MyClass(T v) { val = v; }

    // А этот конструктор по умолчанию
    MyClass( ) { val = null; }

    // ...
    T getVal() { return val; };
}

class ConstructorRefDemo2 {
    public static void main(String args[]) {

        // Создать ссылку на конструктор обобщенного класса MyClass<T>
        MyFunc<Integer> myClassCons = MyClass<Integer>::new;

        MyClass<Integer> mc = myClassCons.func(100);

        // воспользоваться только что созданным
        // экземпляром класса MyClass<T>
        System.out.println(
                "Значение val в объекте mc равно " + mc.getVal());
    }
}
```

Эта версия программы выводит такой же результат, как и предыдущая ее версия, только теперь функциональный интерфейс MyFunc и класс MyClass являются обобщенными. Следовательно, в последовательность кода, создающего ссылку на конструктор, можно включить аргумент типа, как показано ниже, хотя это требуется далеко не всегда.

```
MyFunc<Integer> myClassCons = MyClass<Integer>::new;
```

Аргумент типа Integer уже указан при создании переменной myClassCons, поэтому его можно использовать для создания объекта типа MyClass<Integer>, как показано в следующей строке кода:

```
MyClass<Integer> mc = MyClassCons.func(100);
```

В представленных выше примерах был продемонстрирован механизм применения ссылки на конструктор, но на практике они подобным образом не используются, поскольку это не приносит никаких выгод. Более того, наличие двух обозначений одного и того же конструктора приводит, по меньшей мере, к конфликтной ситуации. Поэтому с целью продемонстрировать более практический пример применения ссылок на конструкторы в приведенной ниже программе применяется статический метод myClassFactory(), который является фабричным для объектов класса любого типа, реализующего интерфейс MyFunc. С помощью этого метода можно создать объект любого типа, имеющего конструктор, совместимый с его первым параметром.

```
// Реализовать простую фабрику классов, используя ссылку на конструктор
interface MyFunc<R, T> {
    R func (T n);
}

// Простой обобщенный класс
class MyClass<T> {
    private T val;

    // Конструктор, принимающий один параметр
    MyClass(T v) { val = v; }

    // Конструктор по умолчанию. Этот конструктор в
    // данной программе НЕ используется
    MyClass() { val = null; }
    // ...

    T getVal() { return val; };
}

// Простой необобщенный класс
class MyClass2 {
    String str;

    // Конструктор, принимающий один аргумент
    MyClass2(String s) { str = s; }

    // Конструктор по умолчанию. Этот конструктор в
    // данной программе НЕ используется
    MyClass2() { str = ""; }

    // ...

    String getVal() { return str; };
}

class ConstructorRefDemo3 {

    // Фабричный метод для объектов разных классов.
    // У каждого класса должен быть свой конструктор,
    // принимающий один параметр типа T. А параметр R
    // обозначает тип создаваемого объекта
    static <R, T> R myClassFactory(MyFunc<R, T> cons, T v) {
        return cons.func(v);
    }

    public static void main(String args[]) {
        // Создать ссылку на конструктор класса MyClass.
        // В данном случае оператор new обращается к конструктору,
        // принимающему аргумент
        MyFunc<MyClass<Double>, Double> myClassCons = MyClass<Double>::new;

        // создать экземпляр типа класса MyClass, используя фабричный метод
        MyClass<Double> mc = myClassFactory(myClassCons, 100.1);
```

```
        // использовать только что созданный экземпляр класса MyClass
        System.out.println(
                "Значение val в объекте mc равно " + mc.getVal());

        // А теперь создать экзепляр другого класса,
        // используя метод myClassFactory()
        MyFunc<MyClass2, String> myClassCons2 = MyClass2::new;

        // создать экземпляр класса MyClass2, используя фабричный метод
        MyClass2 mc2 = myClassFactory(myClassCons2, "Лямбда");

        // использовать только что созданный экземпляр класса MyClass
        System.out.println(
                "Значение str в объекте mc2 равно " + mc.getVal());

    }
}
```

Ниже приведен результат выполнения данной программы.

```
Значение val в объекте mc равно 100.1

Значение str в объекте mc равно Лямбда
```

Как видите, метод myClassFactory() используется для создания объектов типа MyClass<Double> и MyClass2. Несмотря на отличия в обоих классах, в частности, класс MyClass является обобщенным, а класс MyClass2 - необобщенным, объекты обоих классов могут быть созданы с помощью фабричного метода myClassFactory(), поскольку оба они содержат конструкторы, совместимые с методом func() из функционального интерфейса MyFunc, а методу myClassFactory() передается конструктор того класса, объект которого требуется создать. Можете поэкспериментировать немного с данной программой, попробовав создать объекты разных классов, а также экземпляры разнотипных объектов класса MyClass. При этом вы непременно обнаружите, что с помощью метода myClassFactory() можно создать объект любого типа, в классе которого имеется конструктор, совместимый с методом func() из функционального интерфейса MyFunc. Несмотря на всю простоту данного примера, он все же раскрывает истинный потенциал ссылок на конструкторы в Java.

Прежде чем продолжить дальше, следует упомянуть о второй форме синтаксиса ссылок на конструкторы, в которой применяются массивы. В частности, для создания ссылки на конструктор массива служит следующая форма:

```
тип[]::new
```

где тип обозначает создаваемый объект. Так, если обратится к форме класса MyClass, предоставленной в первом примере применения ссылки на конструктор (ConstructorDemo), а также объявить интерфейс MyArrayCreator следующим образом:

```
interface MyArrayCreator<T> {

      T func(int n);

}
```

то в приведенном ниже фрагменте кода создается двухэлементный массив объектов типа MyClass и каждому из них присваивается начальное значение.

```
MyArrayCreator<MyClass[]> mcArrayCons = MyClass[]::new;
MyClass[] a = mcArrayCons.func(2);
a[0] = new MyClass(1);
a[1] = new MyClass(2);
```

В этом фрагменте кода вызов метода func(2) приводит к созданию двухэлементного массива. Как правило, функциональный интерфейс должен содержать метод, принимающий единственный параметр типа int, если он служит для обращения к конструктору массива.

Предопределенные функциональные интерфейсы

В приведенных до сих пор примерах определялись собственные функциональные интерфейсы для целей наглядной демонстрации основных принципов действия лямбда-выражений и функциональных интерфейсов. Но зачастую определять собственный функциональный интерфейс не нужно, поскольку в версии JDK 8 внедрен новый пакет java.util.function, предоставляющий несколько предопределенных функциональных интерфейсов. Рассмотрим некоторые из них.

- UnaryOperator<T> - принимает в качестве параметра объект типа T, выполняет над ними операции и возвращает результат операций в виде объекта типа T
- BinaryOperator<T> принимает в качестве параметра два объекта типа T, выполняет над ними бинарную операцию и возвращает ее результат также в виде объекта типа T
- Predicate<T> проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение true. В качестве параметра лямбда-выражение принимает объект типа T
- Function<T,R> представляет функцию перехода от объекта типа T к объекту типа R
- Consumer<T> выполняет некоторое действие над объектом типа T, при этом ничего не возвращая
- Supplier<T> не принимает никаких аргументов, но должен возвращать объект типа T

В приведенном ниже переделанном примере представленной ранее программы BlockLamdaDemo демонстрируется применение предопределенного функционального интерфейса Function. Если в предыдущей версии данной программы для демонстрации блочных лямбда-выражений на примере вычисления факториала заданного числа был создан собственный функциональный интерфейс NumericFunc, то в новой ее версии для этой цели применяется встроенный функциональный интерфейс Function, как показано ниже.

```
// импортировать функциональный интерфейс Function
import java.util.function;

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

// Это блочное лямбда-выражение вычисляет факториал
// целочисленного значения. Для это цели на сей раз
// используется функциональный интерфейс Function
        Function<Integer, Integer> factorial = (n) -> {
            int result = 1;
            for(int i=1; i <= n; n++)
                result = i * result;
```

```
            return result;
        };

        System.out.println("Факториал числа 3 равен " + factorial.apply(3));
        System.out.println("Факториал числа 5 равен " + factorial.apply(5));
    }
}
```

Stream API

Само понятие пота данных в потоковом AI определяется как канал передачи данных. Следовательно, поток данных представляет собой последовательность объектов. Поток данных оперирует источником данных, например массивом или коллекцией. В самом потоке данные не хранятся, а только перемещаются и, возможно, фильтруются, сортируются и обрабатываются иным образом в ходе этого процесса. Но, как правило, действие самого потока данных не видоизменяет их источник. Например, сортировка данных в потоке не изменяет их упорядочение в источнике, а скорее приводит к созданию нового потока данных, дающего отсортированный результат.

Начиная с JDK 8 в Java появился новый API - Stream API. Его задача - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся основная функциональность данного API сосредоточена в пакете java.util.stream.

Потоковые интерфейсы

В потоковом API определяется ряд потоковых интерфейсов, входящих в состав пакета java.util.stream. В основание их иерархии положен интерфейс BaseStream, в котором определяются основные функциональные возможности всех потоков данных. Интерфейс BaseStream является обобщенным и определяется следующим образом:

```
interface BaseStream<T, S extends BaseStream<T, S>>
```

где параметр T обозначает тип элементов в потоке данных, а параметр S - тип потока данных, расширяющего интерфейс BaseStream. В свою очередь, интерфейс BaseStream расширяет интерфейс AutoCloseble, а следовательно, потоком данных можно управять в блоке оператора try с ресурсами. Но, как правило, закрывать приходится только те потоки данных, где источники данных требуют закрытия (например, те потоки, которые связаны с файлами).

Производными от интерфейса BaseStream являются несколько типов интерфейсов. Наиболее употребительным из них является интерфейс Stream. Он объявляется следующим образом:

```
interface Stream<T>
```

где параметр T обозначает тип элементов в потоке данных. Интерфейс Stream является обобщенным и поэтому пригоден для всех ссылочных типов.

Интерфейс Stream оперирует ссылками на объекты, и поэтому он не может обращаться непосредственно к примитивным типам данных. Для обработки потоков примитивных типов данных в потоковом AI определяются следующие интерфейсы:

- DoubleStream
- IntStream
- LongStream

Все эти интерфейсы расширяют интерфейс BaseStream и обладают теми же функциональными возможностями, что и интерфейс Stream, за искючением того, что они оперируют примитивными, а не ссылочными типами данных.

Получение потока данных

Получить поток данных можно самыми разными способами. Вероятно, самый распространенный способ получения потока данных из коллекции. В версии JDK 8 интерфейс Collection дополнен двумя методами, специально предназначенными для получения потока данных из коллекции. Первый из них называется stream(), а его обща форма показана ниже.

```
default Stream<E> stream()
```

В реализации по умолчанию этот метод возвращает последовательный поток данных. Второй метод называется parallelStream() , а его обща форма выглядит следующим образом:

```
default Stream<E> parallelStream()
```

В реализации по умолчанию этот метод возвращает параллельный поток данных, если это вообще возможно. А если получить параллельный поток данных нельзя, то вместо него может быть возвращен последовательный поток данных.

Параллельные потоки данных поддерживают параллельное выполнение потоковых операций. Благодаря тому что интерфейс Collection может быть реализован в классе каждой коллекции, с помощью упомянутых выше методов можно получить поток из класса любой коллекции, в том числе ArrayList или HashSet.

Рассмотрим на примере, как создать или получить объект java.util.stream.Stream.

- Пустой стрим: Stream.empty() // Stream<String>
- Стрим из List: list.stream() // Stream<String>
- Стрим из Map: map.entrySet().stream() // Stream<Map.Entry<String, String>>
- Стрим из массива: Arrays.stream(array) // Stream<String>
- Стрим из указанных элементов: Stream.of("a", "b", "c") // Stream<String>

Операции Stream API

Рассмотрим основные методы Stream API.

filter(Predicate predicate) - фильтрует стрим, принимая только те элементы, которые удовлетворяют заданному условию. Операция Filter принимает предикат, который фильтрует все элементы потока. Эта операция является промежуточной, т.е. позволяет вызвать другую операцию (например, forEach) над результатом. ForEach принимает функцию, которая вызывается

для каждого элемента в (уже отфильтрованном) поток. ForEach является конечной операцией. Она не возвращает никакого значения, поэтому дальнейший вызов потоковых операций невозможен.

```
Stream.of(1, 2, 3)
    .filter(x -> x == 10)
    .forEach(System.out::print);
    // Вывода нет, так как после фильтрации стрим станет пустым

Stream.of(120, 410, 85, 32, 314, 12)
    .filter(x -> x > 100)
    .forEach(System.out::println);
    // 120, 410, 314
```

map(Function mapper) - применяет функцию к каждому элементу и затем возвращает стрим, в котором элементами будут результаты функции. map можно применять для изменения типа элементов.

- Stream.mapToDouble(ToDoubleFunction mapper)
- Stream.mapToInt(ToIntFunction mapper)
- Stream.mapToLong(ToLongFunction mapper)
- IntStream.mapToObj(IntFunction mapper)
- IntStream.mapToLong(IntToLongFunction mapper)
- IntStream.mapToDouble(IntToDoubleFunction mapper)

Специальные операторы для преобразования объектного стрима в примитивный, примитивного в объектный, либо примитивного стрима одного типа в примитивный стрим другого.

```
Stream.of("3", "4", "5")
        .map(Integer::parseInt)
        .map(x -> x + 10)
        .forEach(System.out::println);
// 13, 14, 15

Stream.of(120, 410, 85, 32, 314, 12)
        .map(x -> x + 11)
        .forEach(System.out::println);
// 131, 421, 96, 43, 325, 23
```

sorted() и sorted(Comparator comparator)

Операция Sorted является промежуточной операцией, которая возвращает отсортированное представление потока. Элементы сортируются в обычном порядке, если вы не предоставили свой компаратор. Причём работает этот оператор очень хитро: если стрим уже помечен как отсортированный, то сортировка проводиться не будет, иначе соберёт все элементы, отсортирует их и вернёт новый стрим, помеченный как отсортированный.

```
IntStream.range(0, 100000000)
        .sorted()
        .limit(3)
        .forEach(System.out::println);
// 0, 1, 2

IntStream.concat(
        IntStream.range(0, 100000000),
```

```
        IntStream.of(-1, -2))
        .sorted()
        .limit(3)
        .forEach(System.out::println);
// Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

Stream.of(120, 410, 85, 32, 314, 12)
        .sorted()
        .forEach(System.out::println);
// 12, 32, 85, 120, 314, 410
```

Краткое описание конвейерных методов работы со стримами

Метод stream	**Описание**	**Пример**
filter	Отфильтровывает записи, возвращает только записи, соответствующие условию	collection.stream().filter(«a1»::equals).count()
skip	Позволяет пропустить N первых элементов	collection.stream() .skip(collection.size() — 1).findFirst().orElse(«1»)
distinct	Возвращает стрим без дубликатов (для метода equals)	collection.stream() .distinct().collect(Collectors.toList())
map	Преобразует каждый элемент стрима	collection.stream() .map((s) -> s + "_1").collect(Collectors.toList())
peek	Возвращает тот же стрим, но применяет функцию к каждому элементу стрима	collection.stream() .map(String::toUpperCase). peek((e) -> System.out.print("," + e)). collect(Collectors.toList())

limit	Позволяет ограничить выборку определенным количеством первых элементов	collection.stream() .limit(2).collect(Collectors.toList())
sorted	Позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator	collection.stream().sorted().collect(Collectors.toList())
mapToInt, mapToDouble, mapToLong	Аналог map, но возвращает числовой стрим (то есть стрим из числовых примитивов)	collection.stream().mapToInt((s) -> Integer.parseInt(s)).toArray()
flatMap, flatMapToInt, flatMapToDouble, flatMapToLong	Похоже на map, но может создавать из одного элемента несколько	collection.stream().flatMap((p) -> Arrays.asList(p.split(",")).stream()).toArray(String[]::new)

Краткое описание терминальных методов работы со стримами

Метод stream	**Описание**	**Пример**
findFirst	Возвращает первый элемент из стрима (возвращает Optional)	collection.stream().findFirst().orElse(«1»)
findAny	Возвращает любой подходящий элемент из стрима (возвращает	collection.stream().findAny().orElse(«1»)

	Optional)	
collect	Представление результатов в виде коллекций и других структур данных	collection.stream() .filter((s) -> s.contains(«1»)).collect(Collectors.toList())
count	Возвращает количество элементов в стриме	collection.stream().filter(«a1»::equals).count()
anyMatch	Возвращает true, если условие выполняется хотя бы для одного элемента	collection.stream().anyMatch(«a1»::equals)
noneMatch	Возвращает true, если условие не выполняется ни для одного элемента	collection.stream().noneMatch(«a8»::equals)
allMatch	Возвращает true, если условие выполняется для всех элементов	collection.stream().allMatch((s) -> s.contains(«1»))
min	Возвращает минимальный элемент, в качестве условия использует компаратор	collection.stream().min(String::compareTo).get()
max	Возвращает максимальный элемент, в качестве условия использует компаратор	collection.stream().max(String::compareTo).get()

forEach	Применяет функцию к каждому объекту стрима, порядок при параллельном выполнении не гарантируется	set.stream().forEach((p) -> p.append("_1"));
forEachOrdered	Применяет функцию к каждому объекту стрима, сохранение порядка элементов гарантирует	list.stream().forEachOrdered((p) -> p.append("_new"));
toArray	Возвращает массив значений стрима	collection.stream() .map(String::toUpperCase).toArray(String[]::new);
reduce	Позволяет выполнять агрегатные функции на всей коллекцией и возвращать один результат	collection.stream().reduce((s1, s2) -> s1 + s2).orElse(0)

Литература и ссылки

1. Спецификация языка Java https://docs.oracle.com/javase/specs/jls/se8/html/index.html
2. Язык программирования Java SE 8. Подробное описание, 5-е издание, Джеймс Гослинг, Билл Джой, Гай Стил, Гилад Брача, Алекс Бакли; 672 стр., с ил.; 2015, 2 кв.; Вильямс.
3. Шилдт Г. Java 8. Полное руководство. 9-е издание. — М.: Вильямс, 2012. — 1377 с.
4. Эккель Б. Философия Java. 4-е полное изд. — СПб.: Питер, 2015. — 1168 с.
5. https://habr.com/ru/company/luxoft/blog/270383/

Вопросы для самоконтроля

1. Опишите основные методы применяемые в Stream API.
2. Какие основные функциональные интерфейсы появились в Java 8?