



Лямбда выражения в Java

Лямбда выражения в Java

- ❑ **Лямбда-выражение** представляет собой блок кода, который можно передать в другое место, поэтому он может быть выполнен позже, один или несколько раз.
 - Лямбда-выражение является блоком кода с параметрами
- ❑ **Где применяются лямбда-выражения?**
 - Когда пытаются передать *функциональность* в качестве *аргумента* другому методу (например, метод ***sort(...)*** получает фрагмент кода, необходимый для сравнения элементов, и этот код встраивается в остальную часть логики сортировки, которую не нужно переопределять);
 - Лямбда-выражения позволяют рассматривать функциональность как аргумент метода или код как данные

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

Лямбда выражения в Java

Например,

```
List<String> names = Arrays.asList("peter", "anna", "mike",  
                                   "xenia");
```

- через анонимный класс

```
Collections.sort(names, new Comparator<String>() {  
    @Override  
    public int compare(String a, String b) {  
        return b.compareTo(a);  
    } });
```

- через лямбда-выражения

```
Collections.sort(names, (a, b) -> b.compareTo(a) );
```

Передача
функциональности
как аргумента
метода

Лямбда выражения в Java

❑ Лямбда-выражение состоит из трех компонентов:

- список аргументов: `(int x, int y)`;
- стрелки: `->`;
- тела: `x + y`.

Например,

`(int x, int y) -> x + y`

Метод получает два аргумента и возвращает их сумму

`() -> 123`

Метод без аргументов и возвращает число 123

`(String s) -> { System.out.println(s); }`

Метод получает один аргумент типа String и отображает строку в консоли

Лямбда выражения в Java

Пример 1:

```
List<String> strings = new ArrayList<String>();  
strings.add("abc");  
strings.add("cba");  
strings.add("test");  
strings.add("hello");  
//...
```

➤ Без лямбда-выражения

```
Collections.sort(strings, new Comparator<String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

Лямбда выражения в Java

Продолжение примера 1, с лямбда-выражением:

- если код занимает более одной строки, то выражение берется в фигурные скобки и при необходимости указывается **return**

```
Collections.sort(strings, (String s1,String s2) -> {  
    if (s1.length() < s2.length()) return -1;  
    else if (s1.length() > s2.length()) return 1;  
    else return 0;  
});
```

Лямбда выражения в Java

Продолжение примера 1, с лямбда-выражением:

- если код занимает одну строку, то в выражении можно опустить скобки { } и оператор **return**

```
Collections.sort(strings, (String s1,String s2) ->  
                                s1.length() - s2.length() );
```

- если типы параметров можно вывести из контекста, то их можно опустить

```
Collections.sort(strings, (s1,s2) -> s1.length() - s2.length() );
```

Лямбда выражения в Java

Пример 2:

- анонимный **Runnable**

```
Runnable r1 = new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Hello world!");  
    }  
};
```



- через лямбда-выражение (если лямбда-выражение не имеет параметров, все равно ставятся пустые скобки, так же, как и с методом без параметров):

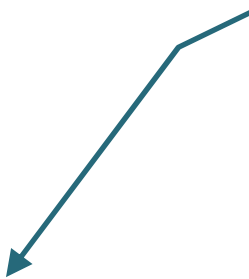
```
Runnable r2 = () -> System.out.println("Hello lambda!");
```


Лямбда выражения в Java

Пример 3:

- использование метода с одним параметром

```
interface MyFunc {  
    public int calc(int x);  
}
```



Метод получает
экземпляр типа
MyFunc

```
public static int testLmb(int[] arr, MyFunc fnc) {  
    int fsum = 0;  
    for (int i = 0; i < arr.length; i++) {  
        fsum += fnc.calc(arr[i]);  
    }  
    return fsum;  
}
```

Лямбда выражения в Java

Продолжение примера 3, с лямбда-выражением:

- если метод имеет один параметр выводимого типа, можно даже опустить скобки () в компоненте «параметр»:

```
public static void main(String[] args) {  
    int[] arr = {1, 2, 3, 4, 5, 5, 7, 8, 9};  
    System.out.println( testLmb(arr, x -> x) );  
    System.out.println( testLmb(arr, x -> x + 1) );  
    System.out.println( testLmb(arr, x -> x * x) );  
    System.out.println( testLmb(arr, x -> (x > 2) ? x : 0) );  
    System.out.println( testLmb(arr, x -> x%3) );  
}
```

Реализация метода
интерфейса
MyFunc

Вывод в консоли:

44
53
274
41
11

Никогда не указывается тип
результата лямбда-выражения.
Это всегда выясняется из
контекста

Лямбда выражения в Java

Пример 4:

```
interface MyCalcInterface {  
    public void doCalc(int value1, int value2);  
}
```



```
MyCalcInterface calc = (v1, v2) -> {  
    v1++;  
    int result = v1 * v2;  
    System.out.println("The result is: " + result);  
};  
calc.doCalc(10, 5);
```

Вывод в консоли:
The result is: 55



Функциональные интерфейсы

Функциональные интерфейсы

- ❑ Каждому лямбда-выражению соответствует тип, представленный *функциональным интерфейсом*.
 - *интерфейс*, который содержит **один абстрактный метод**;
 - каждое лямбда-выражение этого типа будет сопоставлено объявленному методу.



Чтобы не добавлять в язык тип функции

@FunctionalInterface

```
public interface MyFuncInterface {  
    public int calc(int x, int y);  
}  
  
public static void main(String[] args) {  
    MyFuncInterface mfi = (v1, v2) -> v1 * v2;  
    System.out.println(mfi.calc(10, 20));  
}
```

<https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html>

Функциональные интерфейсы

- ❑ Пакет **java.util.function** содержит 43 функциональных интерфейсов общего назначения.
- ❑ Пакет **java.util.function** определяет 4 семейства функциональных интерфейсов.

Function (Функция)	Получает аргументы и что-то возвращает
Predicate (Предикаты)	Получает аргументы и возвращает boolean
Consumer (Потребитель)	Получает аргументы и ничего не возвращает
Supplier (Поставщик)	Не получает аргументы и что-то возвращает

- Функциональные интерфейсы названы в соответствии с аргументами и возвращаемым значением;
- Некоторые содержат один аргумент, некоторые - два аргумента.

Функциональные интерфейсы

Общее описание (1/2)

Функциональный интерфейс	Параметры	Возвращаемый тип	Абстрактный метод	Описание
Supplier<T>	0	T	get	Возвращает объект типа T. Содержит метод get()
Consumer<T>	T	void	accept	Выполняет операцию над объектом типа T. Содержит метод accept()
BiConsumer<T, U>	T, U	void	accept	Выполняет операцию заданную в методе accept() над объектами T и U
Predicate<T>	T	boolean	test	Определяет, удовлетворяет ли объект типа T некоторому условию. Возвращает логическое значение, обозначающее результат. Содержит метод test()
BiPredicate<T, U>	T, U	boolean	test	Возвращает логическое значение true, если оба аргумента удовлетворяют условию, заданному методом test(), иначе false

<https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

Функциональные интерфейсы

Общее описание (2/2)

Функциональный интерфейс	Параметры	Возвращаемый тип	Абстрактный метод	Описание
Function<T, R>	T	R	apply	Выполняет операцию над объектом типа T и возвращает в результате объект типа R. Содержит метод apply()
BiFunction<T, U, R>	T, U	R	apply	Выполняет операцию над объектами T и U и возвращает результат R. Содержит метод apply
UnaryOperator<T>	T	T	apply	Выполняет унарную операцию над объектом типа T и возвращает результат того же типа. Содержит метод apply()
BinaryOperator<T, T>	T, T	T	apply	Выполняет логическую операцию над двумя объектами типа T и возвращает результат того же типа. Содержит метод apply()

Функциональные интерфейсы

Supplier (Поставщик)

- ❑ используется для создания какого-либо объекта без использования входных параметров.
- ❑ синтаксис:

@FunctionalInterface

```
public interface Supplier<T> {  
    T get();  
}
```

Пример 5:

```
public static void main(String[] args) {  
    Supplier< ArrayList<String> > supList = () ->  
                                                new ArrayList<String>();  
    System.out.println( supList.get() );  
}
```

Функциональные интерфейсы

Consumer (Потребитель)

- ❑ используется в том случае, если нужно применить какое-то действие/операцию к параметру (или к двум параметрам для **BiConsumer**) и при этом в возвращаемом значении нет необходимости.
- ❑ синтаксис:

@FunctionalInterface

```
public interface Consumer<T> {  
    void accept(T t);  
}
```

@FunctionalInterface

```
public interface BiConsumer<T, U> {  
    void accept(T t, U u);  
}
```

Функциональные интерфейсы

Пример 6:

```
public static void main(String[] args) {  
    Consumer<String> printer = s -> System.out.println(s);  
    printer.accept("Функциональные интерфейсы в Java");  
}
```

Пример 7:

```
public static void main(String[] args) {  
    Map<Integer, String> map = new HashMap<>();  
    BiConsumer<Integer, String> con = (i, s) -> map.put(i, s);  
    con.accept(1, "item one");  
    con.accept(2, "item two");  
    System.out.println(map);  
}
```

Функциональные интерфейсы

Predicate (Утверждение)

- ❑ используется для проверки соблюдения некоторого условия в (применяется обычно в фильтрах и сравнении);
- ❑ синтаксис:

@FunctionalInterface

```
public interface Predicate<T> {  
    boolean test(T t);  
}
```

@FunctionalInterface

```
public interface BiPredicate<T, U> {  
    boolean test(T t, U u);  
}
```

Функциональные интерфейсы

Пример 8:

```
public static void main(String[] args) {  
    Predicate<Integer> isPositive = x -> x > 0;  
    System.out.println(isPositive.test(5));        // true  
    System.out.println(isPositive.test(-7));       // false  
}
```

Пример 9:

```
public static void main(String[] args) {  
    BiPredicate<String, String> pred = (s1, s2) -> s1.equals(s2);  
    System.out.println(pred.test("Функциональные интерфейсы",  
                                "Функциональные интерфейсы в Java 8"));  
}
```

Функциональные интерфейсы

Function (Функция)

- используется для преобразования входного параметра или в двух параметров (для **BiFunction**) в какое-либо значение, тип значение может не совпадать с типом входных параметров.
- синтаксис:

@FunctionalInterface

```
public interface Function<T, R> {  
    R apply(T t);  
}
```

Тип входного параметра

Тип результата

@FunctionalInterface

```
public interface BiFunction<T, U, R> {  
    R apply(T t, U u);  
}
```

Функциональные интерфейсы

Пример 10:

```
public static void main(String[] args) {  
    Function<String, String> func = s -> s.toUpperCase();  
    System.out.println(func.apply("Функциональные интерфейсы"));  
}
```

Пример 11:

```
public static void main(String[] args) {  
    BiFunction<String, String, String> func =  
                                                (s1, s2) -> s1.concat(s2);  
    System.out.println(func.apply("Функциональные интерфейсы",  
                                   " в Java 8"));  
}
```

Функциональные интерфейсы

UnaryOperator и Binary Operator

- ❑ разновидность **Function**, в которых входные и выходные обобщенные параметры должны совпадать.
 - UnaryOperator расширяет Function,
 - BinaryOperator расширяет BiFunction .

- ❑ синтаксис:

@FunctionalInterface

```
public interface UnaryOperator<T> extends Function<T, T> { }
```

@FunctionalInterface

```
public interface BinaryOperator<T> extends
```

```
BiFunction<T,T,T> { }
```


Функциональные интерфейсы

Пример 12:

```
public static void main(String[] args) {  
    UnaryOperator<Integer> square = x -> x*x;  
    System.out.println(square.apply(5));  
}
```

Вывод в консоли:
25

Пример 2:

```
public static void main(String[] args) {  
    BinaryOperator<Integer> multiply = (x, y) -> x*y;  
    System.out.println(multiply.apply(3, 5));  
    System.out.println(multiply.apply(10, -2));  
}
```

Значения
должны быть
одного типа

Вывод в консоли:
15
-20

Функциональные интерфейсы

- ❑ Существуют функциональные интерфейсы для работы с примитивными типами данных:
 - функциональный интерфейс **Function** единственный, который возвращает обобщенный тип, все остальные либо ничего не возвращают, либо возвращают примитивные типы;
 - функциональные интерфейсы **BiConsumer**, **BiPredicate** и **BiFunction** не используются для работы с примитивами.

Функциональный интерфейс	Параметр	Возвращаемый тип	Абстрактный метод
Supplier			
IntSupplier	0	int	getAsInt()
LongSupplier	0	long	getAsLong()
DoubleSupplier	0	double	getAsDouble()
BooleanSupplier	0	boolean	getAsBoolean()

Функциональные интерфейсы

Функциональный интерфейс	Параметр	Возвращаемый тип	Абстрактный метод
Consumer			
IntConsumer	1 int	void	accept()
LongConsumer	1 long	void	accept()
DoubleConsumer	1 double	void	accept()
Predicate			
IntPredicate	1 int	boolean	test()
LongPredicate	1 long	boolean	test()
DoublePredicate	1 double	boolean	test()
Function			
IntFunction	1 int	R	apply()
LongFunction	1 long	R	apply()
DoubleFunction	1 double	R	apply()
UnaryOperator			
IntUnaryOperator	1 int	int	applyAsInt()
LongUnaryOperator	1 long	long	applyAsLong()
DoubleUnaryOperator	1 double	double	applyAsDouble()
BinaryOperator			
IntBinaryOperator	2 int	int	applyAsInt()
LongBinaryOperator	2 long	long	applyAsLong()
DoubleBinaryOperator	2 double	double	applyAsDouble()

Функциональные интерфейсы

Пример 13:

```
public static void main(String[] args) {  
    DoubleConsumer dc = x -> System.out.println("Value=" + x);  
    dc.accept(6);  
    dc.accept(0.55);  
    dc.accept('7');  
}
```

Допускается только
расширяющее
приведение типов

Вывод в консоли:

Value=6.0
Value=0.55
Value=55.0

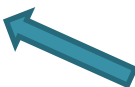


**Ссылка на метод,
оператор ::**

Ссылка на метод, оператор ::

- ❑ Если метод, который мы хотим использовать, уже имеет имя, можно не писать для этого лямбда-выражение, а просто использовать имя метода:

- `ClassName :: staticMethodName`
- `variable :: instanceMethodName`
- `ClassName :: new`

 Ссылка на конструктор

- ❑ Оба подхода в использовании метода класс **Math** эквивалентны:

```
applyMathematicalFunction(x -> Math.sin(x), 10, 100);  
applyMathematicalFunction(Math::sin, 10, 100);
```

<http://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>

Ссылка на метод, оператор ::

Например, отсортировать строки независимо от регистра букв:

```
Arrays.sort(strs, String::compareToIgnoreCase);
```



Тоже самое:

```
Arrays.sort(strs, (x, y) -> x.compareToIgnoreCase(y) );
```



- ❑ Ссылки на методы работают при условии, что параметры вызываемого метода и параметры в лямбда-выражении совпадают;
- ❑ Так же, как и лямбда-выражения, ссылки на методы всегда преобразуются в экземпляры функциональных интерфейсов;
- ❑ При наличии нескольких перегруженных методов с тем же именем компилятор попытается найти из контекста, какой имеется в виду.

Ссылка на метод, оператор ::

Пример 14:

```
public class MainClass {  
    public static void main(String[] args) {  
        Function<String, Integer> toInteger = MainClass::parse;  
        Integer integer = toInteger.apply("5");  
    }  
  
    private static Integer parse(String s) {  
        return Integer.parseInt(s);  
    }  
}
```

Статический метод – имя класса

Ссылка на метод, оператор ::

Пример 15, на статические методы:

```
public class ExpressionHelper {  
    static boolean isEven(int n) {  
        return n%2 == 0;  
    }  
    static boolean isPositive(int n) {  
        return n > 0;  
    }  
}
```

Методы проверки на четность
и положительность

```
public class LambdaApp {  
    private static int sum(int[] numbers, IntPredicate func) {  
        int result = 0;  
        for (int i : numbers) {  
            if (func.test(i))  
                result += i;  
        }  
        return result;  
    }  
}
```

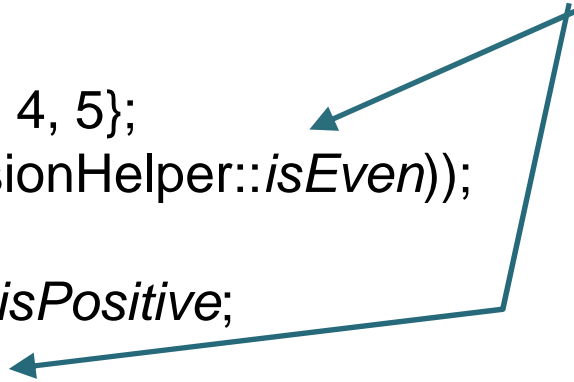
Функциональный интерфейс, у
которого совпадает метод *test()* с
методами класса ExpressionHelper

Ссылка на метод, оператор ::

Продолжение примера 15:

```
public static void main(String[] args) {  
    int[] nums = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};  
    System.out.println(sum(nums, ExpressionHelper::isEven));  
  
    IntPredicate expr = ExpressionHelper::isPositive;  
    System.out.println(sum(nums, expr));  
}  
}
```

Передача ссылки на
статический метод как
аргумента



Вывод в консоли:

0

15


Ссылка на метод, оператор ::

Пример 16, на методы экземпляра:

```
public class ExpressionHelper1 {  
    boolean isEven(int n) { return n%2 == 0; }  
}  
  
public class LambdaApp1 {  
    private static int sum(int[] numbers, IntPredicate func) {  
        int result = 0;  
        for (int i : numbers) {  
            if (func.test(i))  
                result += i;  
        }  
        return result;  
    }  
  
    public static void main(String[] args) {  
        int[] nums = {5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5};  
        ExpressionHelper1 exprHelper = new ExpressionHelper1();  
        System.out.println(sum(nums, exprHelper::isEven));  
    }  
}
```

Вывод в консоли:
12

Вызов через экземпляр ExpressionHelper1



Ссылка на метод, оператор ::

Пример 17, использование ссылки на конструктор:

```
class User {  
    String name, surname;  
  
    User(String name, String surname) {  
        this.name = name;  
        this.surname = surname;  
    }  
}  
  
interface UserFactory {  
    User create(String name, String surname);  
}  
  
public static void main(String[] args) {  
    UserFactory userFactory = User::new;  
    User user = userFactory.create("John", "Snow");  
}
```

Методы функциональных интерфейсов должны принимать тот же список параметров, что и конструкторы класса, и должны возвращать объект данного класса




Области видимости лямбда выражений

Области видимости лямбда выражений

- ❑ Лямбда-выражения имеют доступ к переменным окружающего класса если они являются **final** или эффективной **final**.

Например,

```
public static void repeatText(String text, int count) {  
    Runnable r = () -> {  
        for (int i = 0; i < count; i++) {  
            System.out.println(text);  
            Thread.yield();  
        }  
    };  
    new Thread(r).start();  
}
```



Потоочная обработка данных: Stream

Поточная обработка данных: **Stream**

- ❑ **Stream API** – это способ работать со структурами данных в функциональном стиле.
 - чаще всего с помощью **Stream** в Java 8 работают с коллекциями, но на самом деле этот механизм может использоваться для самых различных данных.
- ❑ Все указанные операции выполняются над каждым значением при условии, что предшествующая операция вернула в поток значение. Это свойство "короткое замыкание".
- ❑ Возможность получить часть результата, когда еще не все данные из потока обработаны – это свойство "отложенность".

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>

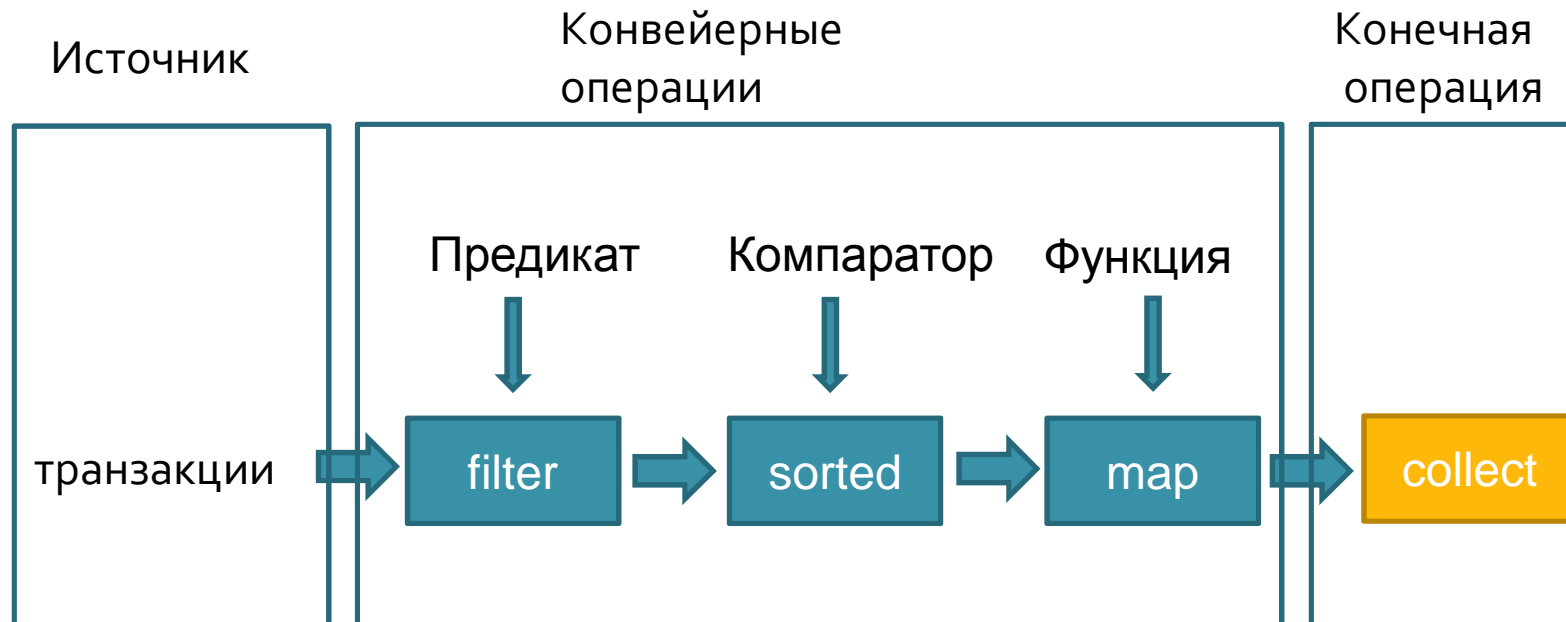
<https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html>

Поточная обработка данных: Stream

- ❑ Потоки представляют собой последовательность элементов, которые поддерживают различные операции, выполняющие вычисления по этим элементам.
- ❑ Есть два типа операций:
 - *Промежуточные (конвейерные) операции* – возвращают другой поток, т.е. работают как строители (фильтрация, сортировка, преобразование и т.д.);
 - *Конечные (терминальные) операции* – возвращают другой объект такой как коллекция, примитивы, объекты, Optional и т.д.
(сборка, forEach, свертка и т.д.)

Поточная обработка данных: Stream

- ❑ Обработка последовательности состоит из источника, промежуточных операций и конечной операции.



- Потоки должны иметь источник!
- Потоки не имеют собственных данных

<http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html>
<http://www.oracle.com/technetwork/articles/java/architect-streams-pt2-2227132.html>

Поточная обработка данных: Stream

Пример 18, предположим, нужно найти все объекты, указанного типа, и вернуть список идентификаторов объектов, отсортированных в порядке убывания весомости объекта:

```
List<Transaction> groceryTransactions = new ArrayList<>();  
for (Transaction t: transactions) {  
    if (t.getType() == Transaction.GROCERY) {  
        groceryTransactions.add(t);  
    }  
}  
Collections.sort(groceryTransactions, new Comparator() {  
    public int compare(Transaction t1, Transaction t2) {  
        return t2.getValue().compareTo(t1.getValue());  
    }  
});  
// .....
```

Поточная обработка данных: Stream

Продолжение примера 18:

//

```
List<Integer> transactionIds = new ArrayList<>();  
for (Transaction t: groceryTransactions) {  
    transactionIds.add(t.getId());  
}
```

- ❑ Стиль написания с использованием поточной обработки будет выглядеть следующим образом:

```
List<Integer> transactionIds = transactions.stream()  
    .filter(t -> t.getType() == Transaction.GROCERY)  
    .sorted(comparing(Transaction::getValue).reversed())  
    .map(Transaction::getId)  
    .collect(Collectors.toList());
```

СОЗДАНИЕ ПОТОКА

➤ *Из коллекции*

```
List<T> myList = ...;  
myList.stream()...;  
myList.parallelStream() ...;
```

➤ *Из массива:*

```
T[] myArray = ...;  
Arrays.stream(myArray)...
```

➤ *Из набора значений:*

```
T t1;  
T t2;  
T t3;  
Stream.of(t1, t2, t3)...
```

Поточная обработка данных: Stream

Классы, методы которых создают потоки (1/2):

- ❑ `BufferedReader.lines()` — поток, элементы которого — строки, прочитанные этим классом;
- ❑ `Files.lines(Path file)` — поток, элементы которого — строки, прочитанные этим классом;
- ❑ `Files.list(Path dir)` — поток, элементы которого все файлы и поддиректории в указанной директории;
- ❑ `Random.ints()` — поток сгенерированных случайных чисел в указанном диапазоне;

<https://docs.oracle.com/javase/8/docs/api/java/nio/file/DirectoryStream.html>

Поточная обработка данных: Stream

Классы, методы которых создают потоки (2/2):

- ❑ `BitSet.stream()` — поток индексов из набора **BitSet** для указанного состояния бита;
- ❑ `Pattern.splitAsStream(java.lang.CharSequence)` — поток строк входной последовательности согласно сопоставлению с данным шаблоном разделителей;
- ❑ `JarFile.stream()` — упорядоченный поток записей из zip-файла.

<https://docs.oracle.com/javase/8/docs/api/java/util/BitSet.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html#splitAsStream-java.lang.CharSequence>

<https://docs.oracle.com/javase/8/docs/api/java/util/jar/JarFile.html>

Поточная обработка данных: **Stream**

➤ С помощью **Stream.Builder**

```
Stream.Builder<String> sb = Stream.<String>builder();
```

```
sb.accept(s1);
```

```
sb.accept(s2);
```

```
Stream<String> s = sb.build(). ...;
```

или

```
Stream<String> s =
```

```
Stream.<String>builder().add(s1).add(s2).build();
```

- ❑ Позволяет создать поток путем создания элементов по отдельности и добавить их в **Builder**.
- ❑ Затем вызывается метод *build()*, который создает поток из добавленных элементов в порядке их добавления.

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.Builder.html>

Поточная обработка данных: Stream

- Из строки:

```
IntStream streamFromString = "123".chars();
```

- ✓ возвращается поток символов указанной строки в виде кодов (любой символ, который отображает расширенный код пропускается необработанным).

- С помощью генератора функций (бесконечные потоки)

```
Stream.generate(Supplier<T> s)
```

```
Stream.iterate(T seed, UnaryOperator<T> f)
```

Например,

```
Stream<Integer> streamFromIterate = Stream.iterate(1, n -> n + 1)
```

```
Stream<String> streamFromGenerate =
```

```
Stream.generate(() -> "a1")
```

Поточная обработка данных: Stream

ПРОМЕЖУТОЧНЫЕ ОПЕРАЦИИ

- ❑ *map* – преобразование из одного типа в другой
 - ❑ *filter* – отбирает элементы потока по указанному критерию
 - ❑ *limit* – ограничивает выборку указанным количеством первых элементов потока
 - ❑ *skip* – позволяет пропустить указанное количество первых элементов потока
 - ❑ *sorted* – упорядочить значения в натуральном порядке или согласно компаратору
 - ❑ *mapToInt* – аналог операции *map*, но возвращает поток чисел указанного типа
 - ❑ *flatMap* – возможность создания из одного элемента нескольких
 - ❑ *parallel* – вернуть параллельный поток
 - ❑ *sequential* – вернуть последовательный поток
-
- Операции задокументированы в **java.util.stream.Stream**
 - Промежуточные операции будут выполнены только тогда, когда присутствуют конечные (терминальные) операции.

Поточная обработка данных: Stream

1) Операция **map** (mapToInt, mapToDouble, flatMap)

Stream<R> map(Function<? super T, ? extends R> mapper)

- функция преобразования применяется к каждому элементу потока и возвращает значение в поток, состоящий из результата функции.

Пример 19:

```
Stream<String> inputLines = Stream.of("a", "aB", "cd");  
List<String> outputLines =  
    inputLines.map(String::toUpperCase)  
                .collect(Collectors.toList());  
System.out.println(outputLines);
```

Вывод в консоли:
[A, AB, CD]

Поточная обработка данных: Stream

2) Операция **filter**

Stream<T> filter(Predicate<? super T> predicate)

- возвращает поток, состоящий из элементов текущего потока, которые удовлетворяют указанному Предикату.

Пример 20:

```
IntStream.range(1, 100)  
    .filter( e -> ((e % 2 == 0) && (e % 3 == 0)) )  
    .forEach(System.out.println);
```

Вывод в консоли:

```
6  
12  
18  
24  
30  
...
```

Поточная обработка данных: Stream

3) Операции **limit** и **skip**

- ✓ *limit(long maxSize)* — ограничивает поток указанным количеством элементов;
- ✓ *skip(long count)* — пропускает указанное количество элементов.

Пример 21:

```
Stream.iterate(2, n -> n + 2)  
    .limit(10)  
    .skip(4)  
    .forEach(System.out::println);
```

Вывод в консоли:

```
10  
12  
14  
16  
18  
20
```

КОНЕЧНЫЕ ОПЕРАЦИИ

- ❑ *forEach* – применить указанную функцию к каждому элементу потока
- ❑ *toArray, toList* – возвращает массив/список
- ❑ *reduce* – выполнить операцию над всей коллекцией и вернуть один результат
- ❑ *collect* – сборка результата в виде коллекции или другой структуры данных
- ❑ *min, max, count, sum* – возвращает минимальный, максимальный, количество и сумму
- ❑ *anyMatch, allMatch, noneMatch* – сопоставление и возврат **true**, если условие выполняется для одного, для всех и ни одного
- ❑ *findFirst, findAny* – возвращает первый и любой подходящий элемент
- ❑ *iterator, spliterator* – создание бесконечного потока

Поточная обработка данных: Stream

1) Операции **findFirst** и **findAny** - короткого замыкания, которые возвращают объект типа **Optional**.

❑ **Optional** - это объект-контейнер, который может или не может содержать ненулевое значение (если значение присутствует, метод *isPresent()* возвращает *true* и метод *get()* вернет значение).

- **Optional** не является функциональным интерфейсом, однако является удобным средством предотвращения всеми известным **NullPointerException**.
- Обеспечивает дополнительные методы, которые зависят от наличия или отсутствия содержащегося значения (*например*, *orElse()* - возвращает значение по умолчанию, если оно отсутствует, и *ifPresent()* - выполнить блок кода, если значение присутствует).


<https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html>

Поточная обработка данных: Stream

2) Операция **collect**

- обычно аргумент для *collect()* является тип **Collector**, который воплощает в себе правило для вкладывания элементов в структуру данных или группирования.

Поставщик объекта
коллекции



Например: накапливание строк в **ArrayList**:

```
List asList = stringStream.collect(Collectors.toList())
```

- Вторая форма метода *collect()* тоже принимает **Collector**, который состоит из операций: поставщик, аккумулятор, объединитель и финишер.

Поточная обработка данных: Stream

Например,

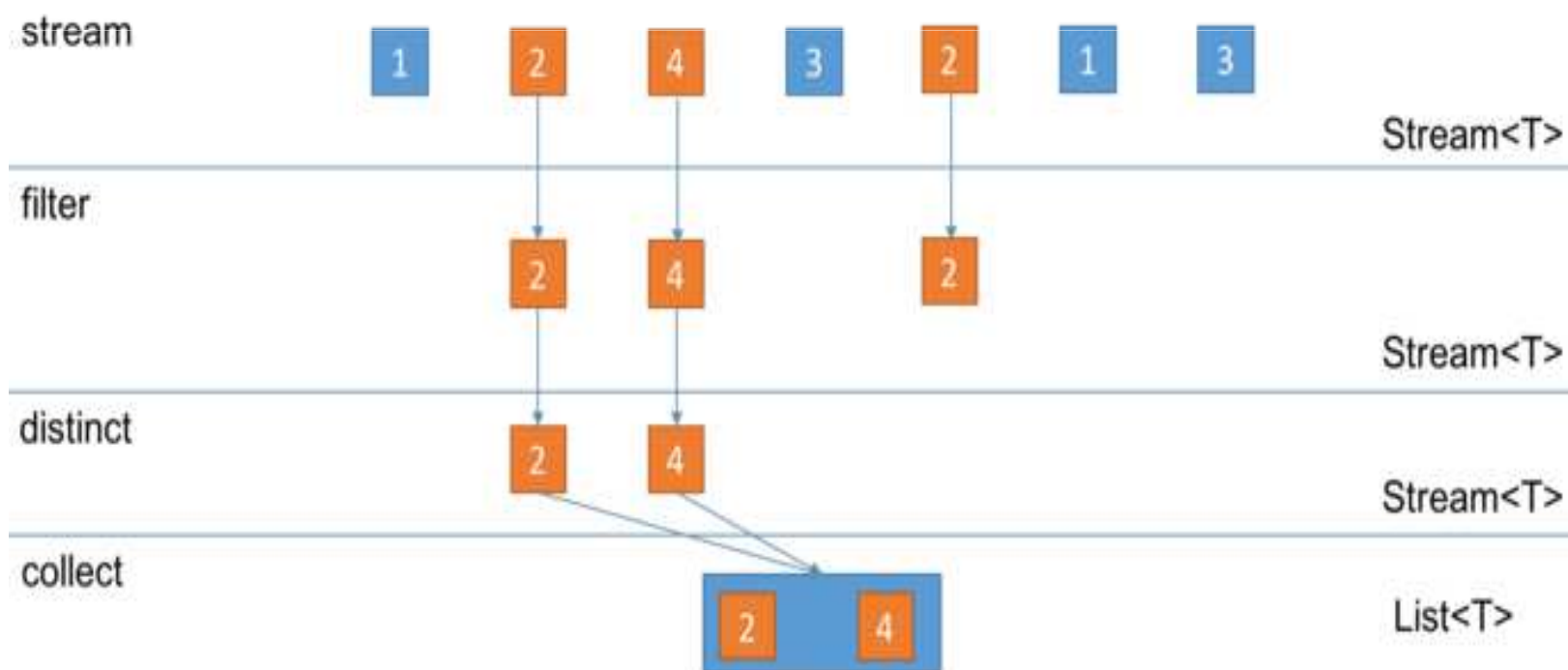
```
Stream<Integer> input = Stream.of(1,2,4,3,2,1,3);
```

```
List<Integer> output = input.filter(x->x%2==0)
```

```
.distinct()
```

```
.collect(Collectors.toList());
```

Удалить дубликаты



Поточная обработка данных: Stream

Пример 22:

1. `List<Integer> numbers =
Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);`
2. `List<Integer> result =
numbers.stream()
.filter(n -> (n % 2 == 0))
.map(n -> n * n)
.collect(Collectors.toList());`
3. `System.out.println(result);`

Вывод в консоли:
[4, 16, 36, 64]

Поточная обработка данных: Stream

Пример 23:

1. `List<Integer> numbers =
 Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);`
2. `List<Integer> result =
 numbers.stream()
 .peek(e -> System.out.println("1. Input value: " + e))
 .filter(n -> (n % 2 == 0))
 .peek(e -> System.out.println("2. Filtered value: " + e))
 .map(n -> n * n)
 .peek(e -> System.out.println("3. Mapped value: " + e))
 .collect(Collectors.toList());`
3. `System.out.println(result);`

Вывод в консоли:

```
1. Input value: 1  
1. Input value: 2  
2. Filtered value: 2  
3. Mapped value: 4  
1. Input value: 3  
.....
```

Поточная обработка данных: Stream

Как определить поток?

Последовательность элементов: Поток обеспечивает интерфейс для последовательного множества значений определенного типа элемента. Тем не менее, потоки фактически не хранят элементы; они вычисляют их по запросу.

Источник: Потоки потребляют данные из источника, такого как коллекция, массив или ресурс ввода/вывода.

Агрегирование операций: Потоки поддерживают SQL-подобные операции и общие операции от функциональных языков программирования, такие как *filter*, *map*, *reduce*, *find*, *match*, *sorted* и так далее.

Вывод в консоли:

```
1. Input value: 1
1. Input value: 2
2. Filtered value: 2
3. Mapped value: 4
1. Input value: 3
1. Input value: 4
2. Filtered value: 4
3. Mapped value: 16
1. Input value: 5
1. Input value: 6
2. Filtered value: 6
3. Mapped value: 36
1. Input value: 7
1. Input value: 8
2. Filtered value: 8
3. Mapped value: 64
[4, 16, 36, 64]
```

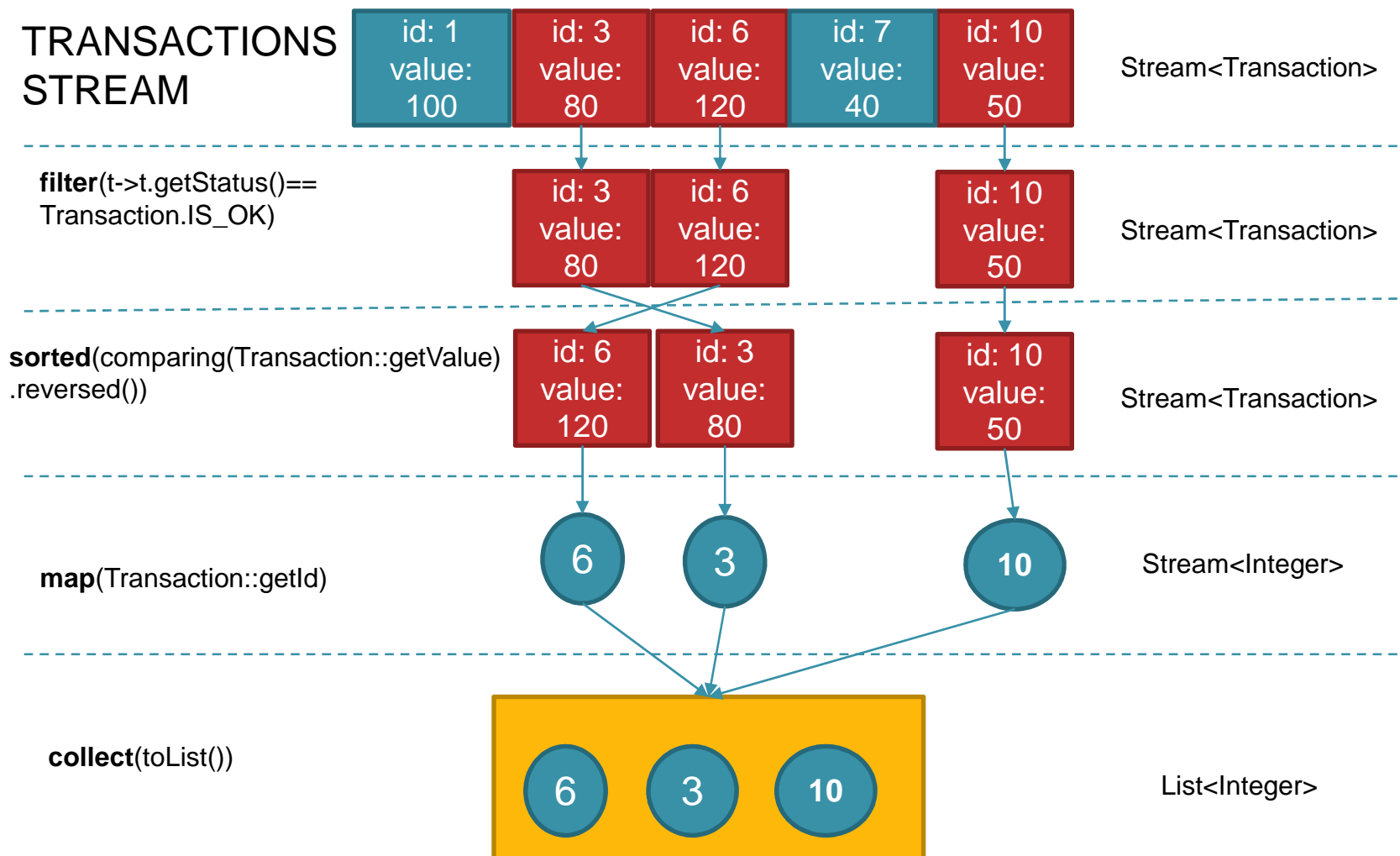
Поточная обработка данных: Stream

Пример 24:

```
List<Integer> transactionsIds =  
    transactions.stream()  
        .filter(t -> (t.getStatus() == Transaction.IS_OK))  
        .sorted(comparing(Transaction::getValue).reversed())  
        .map(Transaction::getId)  
        .collect(Collectors.toList());
```

Поточная обработка данных: Stream

TRANSACTIONS
STREAM



Поточная обработка данных: Stream

3) Операция **reduce()** выполняет терминальные операции свертки, возвращая некоторое значение - результат операции. Он имеет следующие формы:

`Optional<T> reduce(BinaryOperator<T> accumulator);`

`T reduce(T identity, BinaryOperator<T> accumulator);`

`U reduce(U identity, BiFunction<U, ? super T, U>
accumulator, BinaryOperator<U> combiner)`

- Объект **BinaryOperator<T>** представляет функцию, которая принимает два элемента и выполняет над ними указанную операцию, возвращая результат.
- При этом метод **reduce** сохраняет результат и применяет его как первый аргумент бинарной операции, а следующий элемент в потоке, как второй аргумент.

Поточная обработка данных: Stream

Пример 25:

Вывод в консоли:

45

```
public static void main(String[] args) {  
    Stream<Integer> numStream = Stream.of(1,2,3,4,5,6,7,8,9);  
    Optional<Integer> res = numbersStream.reduce((x, y)->x+y);  
    System.out.println(result.get());  
}
```

Вывод в консоли:

Результат: мама мыла раму

Пример 26:

```
Stream<String> wordsStream =  
    Stream.of("мама", "мыла", "раму");  
String sentence = wordsStream.reduce("Результат:",  
    (x,y)->x + " " + y);  
System.out.println(sentence);
```

Начальное значение, а также значение по умолчанию