



# Lesson 13

30.06.2022

```
public static void main(String[] args) {  
    int i = 10 ;  
    System.out.println(i > 3 != false );  
}
```

```
public static void main(String[] args) {  
    String javaworld = "JavaWorld";  
    String java = "Java";  
    String world = "World";  
    java += world;  
  
    System.out.println(java == javaworld);  
}
```

```
public static void main(String[] args) {  
    List<String> list = new ArrayList<>(); //line n1  
    list.add("A");  
    list.add("E");  
    list.add("I");  
    list.add("O");  
    list.add("U");  
    list.addAll(list.subList(0, 4)); //line n1  
    System.out.println(list);  
}
```

```
public static void main(String[] args) {  
    List<Integer> list = new ArrayList<>();  
    list.add( 15 );  
    list.add( 25 );  
    list.add( 15 );  
    list.add( 25 );  
    list.remove(Integer.valueOf ( 15 ));  
    System.out.println(list);  
}
```

## Optional<T>

При написании кода разработчик часто не может знать — будет ли существовать нужный объект на момент исполнения программы или нет, и в таких случаях приходится делать проверки на null. Если такими проверками пренебречь, то рано или поздно (обычно рано) Ваша программа рухнет с ***NullPointerException***.

```
User user = null;
if (Objects.nonNull(user)) {
    System.out.println(user.toString());
} else {
    System.out.println("null object");
}
```

```
Optional<User> optionalUser = Optional.of(user);
```

Существует всего три категории Optional:

**Optional.of** — возвращает Optional-объект.

**Optional.ofNullable** -возвращает Optional-объект, а если нет объекта, возвращает пустой Optional-объект.

**Optional.empty** — возвращает пустой Optional-объект.

## **.ifPresent()**

Метод позволяет выполнить какое-то действие, если объект не пустой.


```
Optional<User> optionalUserOf = Optional.of(user);  
  
optionalUserOf.ifPresent(optionalUserOf.get()::printUser);
```

## **.isPresent()**

Этот метод возвращает ответ, существует ли искомый объект или нет, в виде Boolean:

```
Boolean isUserPresent = optionalUserOf.isPresent();
```





Существует три прямых метода дальнейшего получения объекта семейства `orElse()`; Как следует из перевода, эти методы срабатывают в том случае, если объекта в полученном `Optional` не нашлось.

- `orElse()` — возвращает объект по дефолту.
- `orElseGet()` — вызывает указанный метод.
- `orElseThrow()` — выбрасывает исключение.



Работа с полученным объектом.

`get()` — возвращает объект, упакованный в `Optional`.


`map()` — преобразовывает объект в другой объект.

`filter()` — фильтрует содержащиеся объекты по предикату.

`flatMap()` — возвращает множество в виде стрима.

Стримы бывают последовательными (sequential) и параллельными (parallel). Последовательные выполняются только в текущем потоке, а вот параллельные используют общий пул [ForkJoinPool.commonPool\(\)](#). При этом элементы разбиваются (если это возможно) на несколько групп и обрабатываются в каждом потоке отдельно. Затем на нужном этапе группы объединяются в одну для предоставления конечного результата.

Чтобы получить параллельный стрим, нужно либо вызвать метод `parallelStream()` вместо `stream()`, либо превратить обычный стрим в параллельный, вызвав промежуточный оператор `parallel`.



Кроме объектных стримов `Stream<T>`, существуют специальные стримы для примитивных типов:

- `IntStream` для `int`,
- `LongStream` для `long`,
- `DoubleStream` для `double`.

## Предикаты

Предикаты — это функции, принимающие один аргумент, и возвращающие значение типа `boolean`. Интерфейс содержит различные методы по умолчанию, позволяющие строить сложные условия

```
Predicate<User> isAdult = (u) -> u.getAge() > 18;  
Predicate<User> isMale = (u) -> u.getSex().equals(Sex.MALE);
```



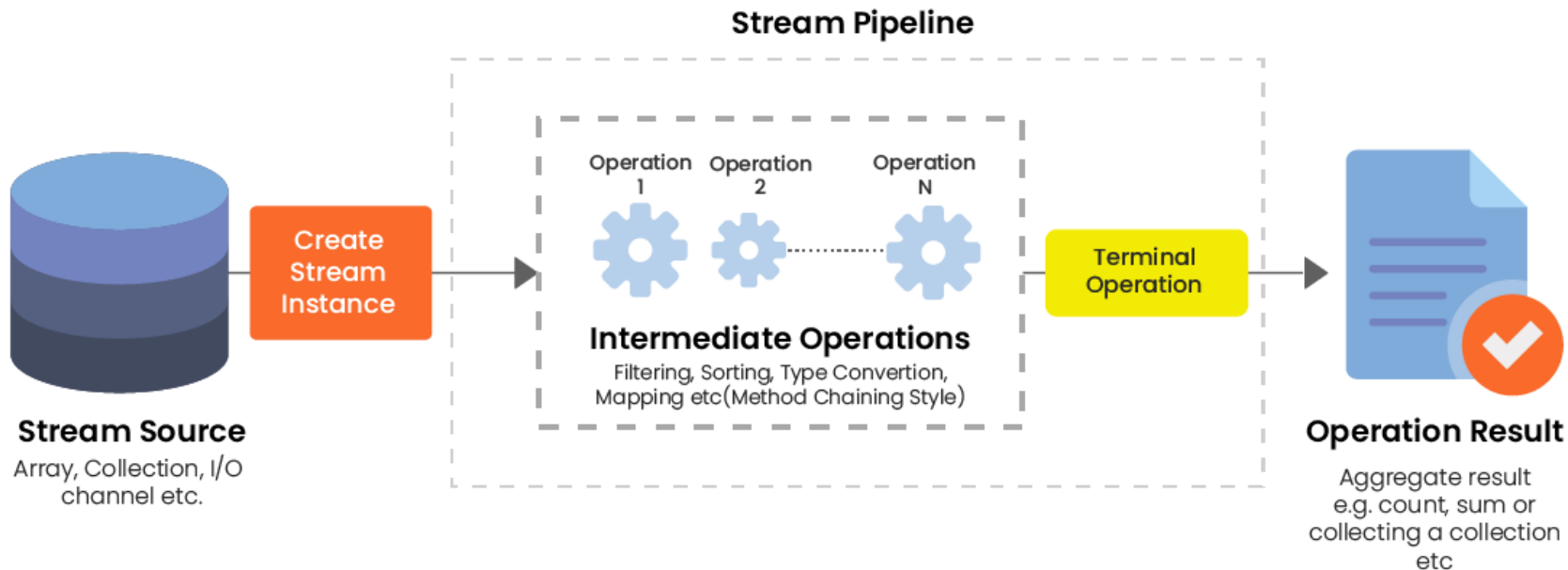
## Интерфейс Comparator

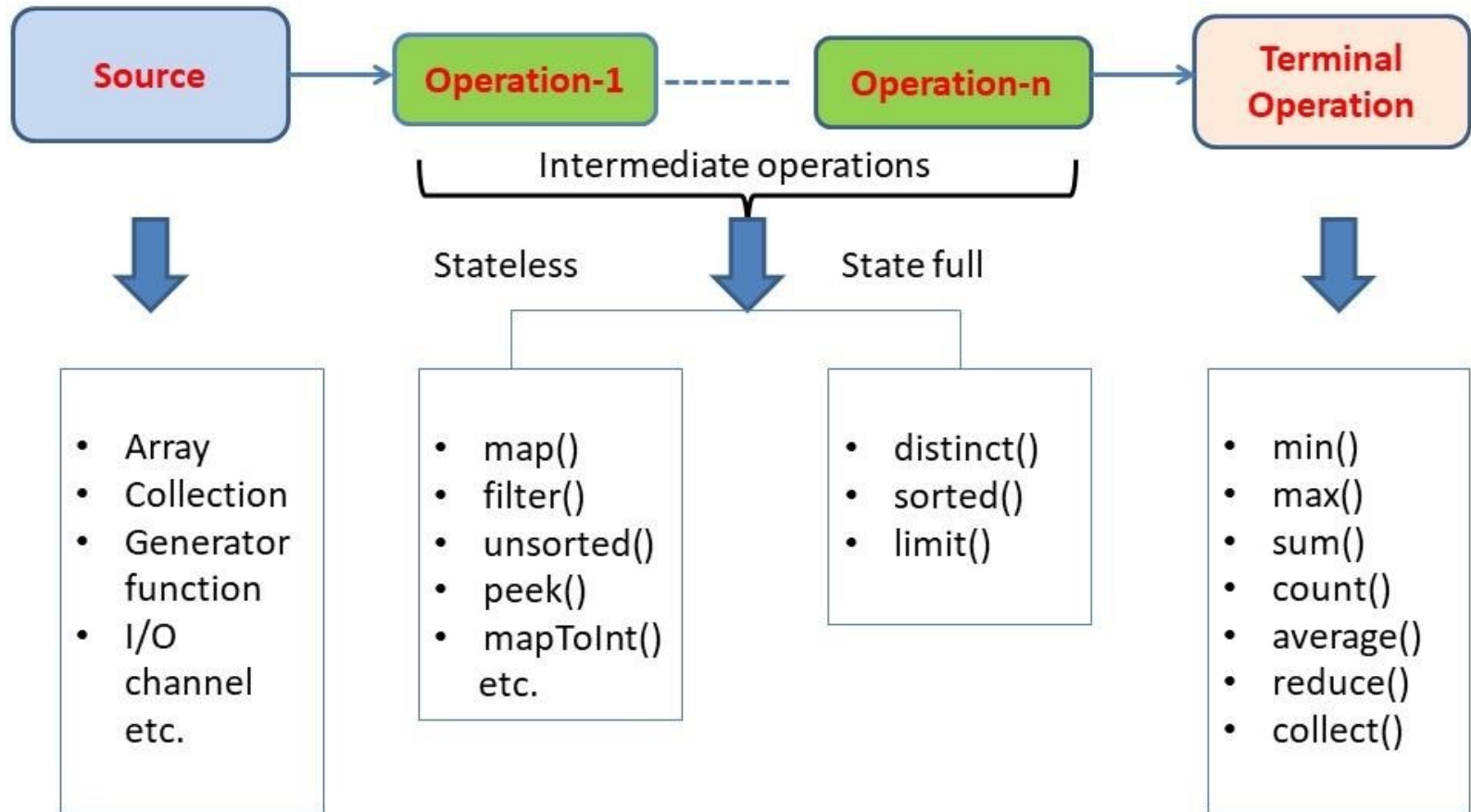
```
Comparator<User> byName = new Comparator<User>() {  
    @Override  
    public int compare(User o1, User o2) {  
        return o1.getFirstName().compareTo(o2.getFirstName());  
    }  
};
```

```
Comparator<User> bySurName = Comparator.comparing(User::getLastName);
```

```
Comparator<User> bySurName =  
    (User u1, User u2) -> u1.getLastName().compareTo(u2.getLastName());
```

# Java Streams









## Промежуточные операторы

### **filter(Predicate predicate)**

Фильтрует стрим, принимая только те элементы, которые удовлетворяют заданному условию.

### **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.

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



## **limit(long maxSize)**

Ограничивает стрим maxSize элементами.

## **skip(long n)**

Пропускает n элементов стрима.

## **sorted()**

## **sorted(Comparator comparator)**

Сортирует элементы стрима. 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.

## **distinct()**

Убирает повторяющиеся элементы и возвращаем стрим с уникальными элементами. Как и в случае с sorted, смотрит, состоит ли уже стрим из уникальных элементов и если это не так, отбирает уникальные и помечает стрим как содержащий уникальные элементы.



## **peek(Consumer action)**

Выполняет действие над каждым элементом стрима и при этом возвращает стрим с элементами исходного стрима. Служит для того, чтобы передать элемент куда-нибудь, не разрывая при этом цепочку операторов (вы же помните, что `forEach` — терминальный оператор и после него стрим завершается?), либо для отладки.

## **takeWhile(Predicate predicate)**

Появился в Java 9. Возвращает элементы до тех пор, пока они удовлетворяют условию, то есть функция-предикат возвращает `true`. Это как `limit`, только не с числом, а с условием.

## **dropWhile(Predicate predicate)**

Появился в Java 9. Пропускает элементы до тех пор, пока они удовлетворяют условию, затем возвращает оставшуюся часть стрима. Если предикат вернул для первого элемента `false`, то ни единого элемента не будет пропущено. Оператор подобен `skip`, только работает по условию.

## Терминальные операторы

### **void forEach(Consumer action)**

Выполняет указанное действие для каждого элемента стрима.

### **long count()**

Возвращает количество элементов стрима.

### **R collect(Collector collector)**

Один из самых мощных операторов Stream API. С его помощью можно собрать все элементы в список, множество или другую коллекцию, сгруппировать элементы по какому-нибудь критерию, объединить всё в строку и т.д

### **Object[] toArray()**

Возвращает нетипизированный массив с элементами стрима.



## **Optional min(Comparator comparator)**

## **Optional max(Comparator comparator)**

Поиск минимального/максимального элемента, основываясь на переданном компараторе.

## **Optional findAny()**

Возвращает первый попавшийся элемент стрима. В параллельных стримах это может быть действительно любой элемент, который лежал в разбитой части последовательности.

## **Optional findFirst()**

Гарантированно возвращает первый элемент стрима, даже если стрим параллельный.



## **boolean allMatch(Predicate predicate)**

Возвращает true, если все элементы стрима удовлетворяют условию predicate. Если встречается какой-либо элемент, для которого результат вызова функции-предиката будет false, то оператор перестаёт просматривать элементы и возвращает false.

## **boolean anyMatch(Predicate predicate)**

Возвращает true, если хотя бы один элемент стрима удовлетворяет условию predicate. Если такой элемент встретился, нет смысла продолжать перебор элементов, поэтому сразу возвращается результат.

## **boolean noneMatch(Predicate predicate)**

Возвращает true, если, пройдя все элементы стрима, ни один не удовлетворил условию predicate. Если встречается какой-либо элемент, для которого результат вызова функции-предиката будет true, то оператор перестаёт перебирать элементы и возвращает false.



## Методы Collectors

### **toList()**

Самый распространённый метод. Собирает элементы в List.

### **toSet()**

Собирает элементы в множество.

### **toMap(Function keyMapper, Function valueMapper)**

Собирает элементы в Map. Каждый элемент преобразовывается в ключ и в значение, основываясь на результате функций `keyMapper` и `valueMapper` соответственно. Если нужно вернуть тот же элемент, что и пришел, то можно передать `Function.identity()`.

### **counting()**

Подсчитывает количество элементов.