




# Lesson 14

15.06.2023



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

```
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);  
}
```



## Optional<T>

При написанні коду розробник часто не може знати, чи буде потрібний об'єкт на момент виконання програми чи ні, і в таких випадках доводиться робити перевірки на null. Якщо такими перевітками знехтувати, то рано чи пізно (зазвичай рано) Ваша програма впаде з NullPointerException.

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

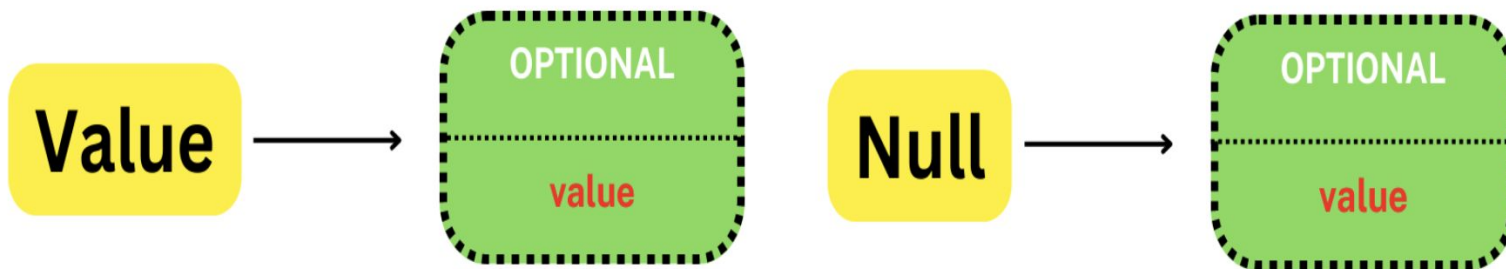
```
Optional<User> optionalUser = Optional.of(user);
```

Існує лише три категорії **Optional**:

**Optional.of** – повертає **Optional**-об'єкт.

**Optional.ofNullable** - повертає **Optional**-об'єкт, а якщо немає об'єкта, повертає порожній **Optional**-об'єкт.

**Optional.empty** – повертає порожній **Optional**-об'єкт






**.ifPresent()** - Метод дозволяє виконати якусь дію, якщо об'єкт не порожній

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

**.isPresent()** - Цей метод повертає відповідь, чи існує об'єкт, чи ні, у вигляді Boolean:

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





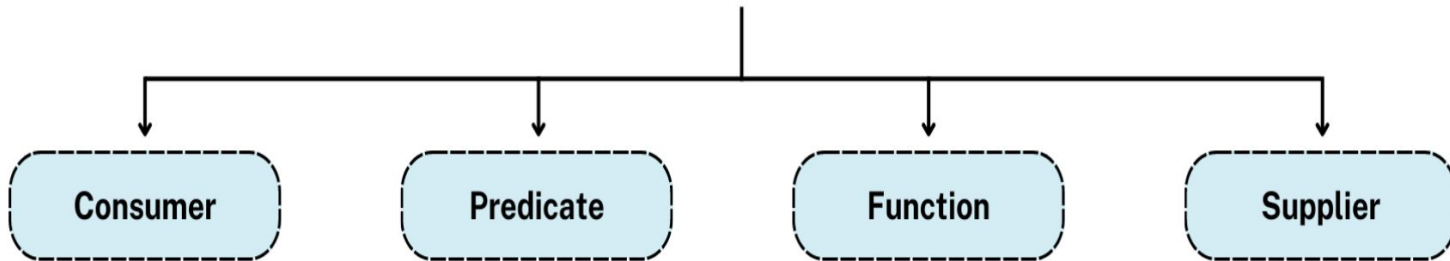
Існує три прямі методи подальшого отримання об'єкта сімейства `orElse()`; Як впливає з перекладу, ці методи спрацьовують у тому випадку, якщо об'єкта отриманого `Optional` не знайшлося.

- **`orElse()`** - Повертає об'єкт по дефолту.

- **`orElseGet()`** - викликає вказаний метод.

- **`orElseThrow()`** - викидає виняток.

# Functional Interface



Предикати - це функції, що приймають один аргумент, і повертають значення типу `boolean`. Інтерфейс містить різні методи за промовчанням, що дозволяють будувати складні умови

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

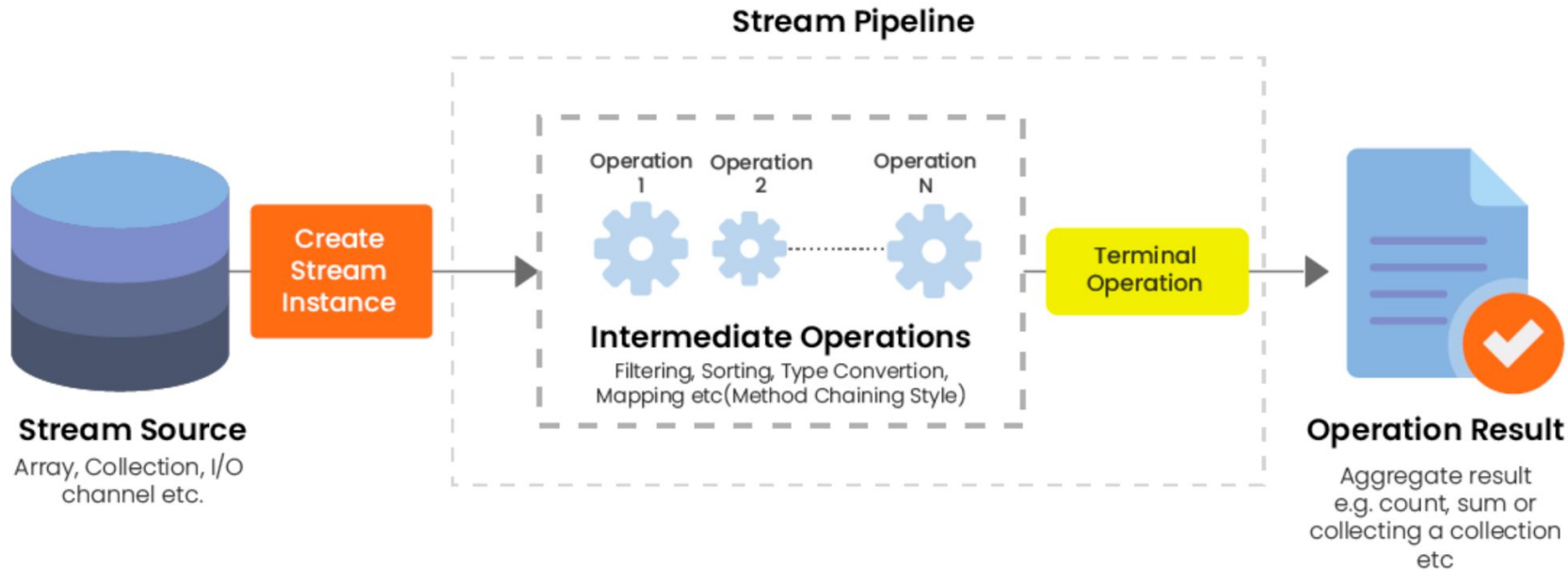
## Comparator

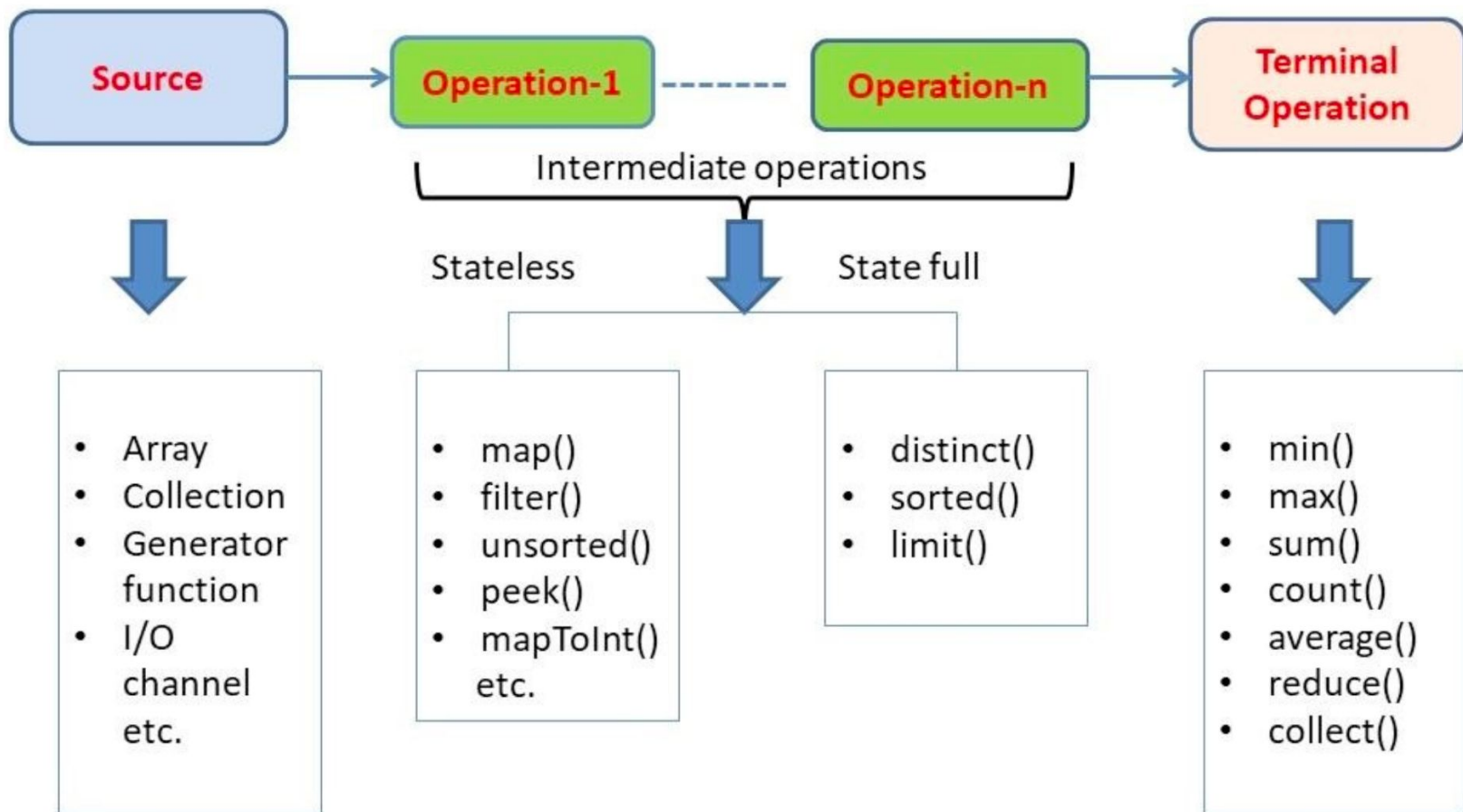
```
Comparator<User> byName = new Comparator<User>() {  
    @Override  
    public int compare(User o1, User o2) {  
        return o1.getFirstName().compareTo(o2.getFirstName());  
    }  
};
```

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

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

# Java Streams





List<String>

John
Martin
Mary
Steve

Stream<String>

John
Martin
Mary
Steve

Stream<String>

JOHN
MARTIN
MARY
STEVE

Output

JOHN
MARTIN
MARY
STEVE

`.stream()`

`.map((s)->s.toUpperCase())`

Intermediate Operation

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

Terminal Operation



## ***Проміжні оператори***

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

Фільтрує стрим, приймаючи лише ті елементи, які задовольняють задану умову.

### **map(Function mapper)**

Застосовує функцію кожного елемента і потім повертає стрим, в якому елементами будуть результати функції. map можна використовувати для зміни типу елементів.

Спеціальні оператори для перетворення об'єктного стриму на примітивний, примітивного на об'єктний, або примітивного стриму одного типу на примітивний стрим іншого.

### **limit(long maxSize)**

Обмежує стриму maxSize елементами.

### **skip(long n)**

Пропускає n елементів стримування.

## **sorted(), sorted(Comparator comparator)**

Сортує елементи стримування. Причому працює цей оператор дуже хитро: якщо стрім вже позначений як відсортований, то сортування не проводитиметься, інакше збере всі елементи, відсортує їх і поверне новий стрім, позначений як відсортований.

## **distinct()**

Прибирає елементи, що повторюються, і повертаємо стрім з унікальними елементами. Як і у випадку з `sorted`, дивиться, чи складається вже стрім з унікальних елементів і якщо це не так, відбирає унікальні і позначає стрім як містить унікальні елементи.

## **peek(Consumer action)**

Виконує дію над кожним елементом стриму і при цьому повертає стриму з елементами вихідного стриму. Служить для того, щоб передати елемент кудись, не розриваючи при цьому ланцюжок операторів (ви ж пам'ятаєте, що `forEach` — термінальний оператор і після нього стримується?), або для налагодження.





## **takeWhile(Predicate predicate)**

З'явився в Java 9. Повертає елементи доти, доки вони задовольняють умові, тобто функція-предикат повертає true. Це як limit, тільки з числом, і з умовою.

## **dropWhile(Predicate predicate)**

З'явився в Java 9. Пропускає елементи до тих пір, поки вони задовольняють умові, потім повертає частину стриму, що залишилася. Якщо предикат повернув для першого елемента false, то жодного елемента не буде пропущено. Оператор подібний до skip, тільки працює за умовою



## Термінальні оператори

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

Виконує вказану дію для кожного елемента стримування.

### **long count()**

Повертає кількість елементів стримування.

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

Один із найпотужніших операторів Stream API. З його допомогою можна зібрати всі елементи в список, безліч або іншу колекцію, згрупувати елементи за якимось критерієм, об'єднати все в рядок і т.д.

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

Повертає нетипізований масив з елементами стор

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

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

Пошук мінімального/максимального елемента, ґрунтуючись на переданому компараторі.



## Optional findAny()

Повертає перший елемент стриму, що попався. У паралельних стримах це може бути справді будь-який елемент, який лежав у розбитій частині послідовності.

## Optional findFirst()

Гарантовано повертає перший елемент стриму, навіть якщо стрім паралельний

## boolean allMatch(Predicate predicate)

Повертає true, якщо всі елементи стриму задовольняють умову predicate. Якщо зустрічається якийсь елемент, для якого результат виклику функції предикату буде false, то оператор перестав переглядати елементи та повертає false.

## boolean anyMatch(Predicate predicate)

Повертає true, якщо хоча б один елемент стриму задовольняє умову predicate. Якщо такий елемент зустрівся, немає сенсу продовжувати перебір елементів, тому одразу повертається результат.

## boolean noneMatch(Predicate predicate)

Повертає true, якщо, пройшовши всі елементи стриму, жоден не задовольнив умову predicate. Якщо зустрічається якийсь елемент, для якого результат виклику функції-предикату буде true, оператор перестав перебирати елементи і повертає false.



## Методи Collectors

### **toList()**

Найпоширеніший метод. Збирає елементи у List.

### **toSet()**

Збирає елементи у безліч.

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

Збирає елементи в Map. Кожен елемент перетворюється на ключ і значення, ґрунтуючись на результаті функцій keyMapper і valueMapper відповідно. Якщо потрібно повернути той самий елемент, що й прийшов, можна передати Function.identity().

### **counting()**

Підраховує кількість елементів.