

Java 8

java.util.function

# Основные интерфейсы

- **Функции**
  - $Function<T, R> :: T \Rightarrow R$
  - $Predicate<T> :: T \Rightarrow \text{boolean}$
  - $Supplier<T> :: () \Rightarrow T$
  - $Consumer<T> :: T \Rightarrow \text{void}$
- **Операторы**
  - $UnaryOperator<T> \text{ — } T \Rightarrow T$
  - $BinaryOperator<T> \text{ — } T \Rightarrow T \Rightarrow T$
- **Удвоенные интерфейсы**
  - $BiFunction<T, U, R> :: T \Rightarrow U \Rightarrow R$
  - $BiConsumer<T, U> :: T \Rightarrow U \Rightarrow \text{void}$
  - $BiPredicate<T, U> :: T \Rightarrow U \Rightarrow \text{boolean}$

# Специализации для примитивов

- Типы
  - *Int*
  - *Long*
  - *Double*
- Примеры
  - *IntPredicate*
  - *IntBinaryOperator*
  - *IntToDoubleFunction*
  - *ObjIntConsumer*
  - *ToIntBiFunction*

java.util.collection

# Необязательные значения

- Класс *Optional<T>*
- Фабрики
  - *empty()* — без значения
  - *of(T)* / *ofNullable(T)* — из значения
- Операции
  - *get()* — получить значение или исключение (если null)
  - *isPresent()* — есть ли значение
- Комбинаторы
  - *filter(Predicate)* / *map(Function)* / *flatMap(Function)*
  - *orElse(T)* / *orElseGet(Supplier)* / *orElseThrow(Supplier)*

# Сравнения

- Методы интерфейса `Comparator`
  - `comparing(f, cmp?)` — сравнивать значения `f(o)`
  - `nullsFirst/nullsLast(cmp)` с порядок для `null`
  - `cmp.thenComparing(f, cmp?)` — лексикографический порядок

- Пример

```
students.sort(  
    Comparator.comparingInt(Student::getAge)  
        .thenComparing(  
            Student::getName,  
            String.CASE_INSENSITIVE_ORDER  
        )  
)
```

java.util.stream



# Потоки значений

- Классы *Stream*, *IntStream*/*LongStream*/*DoubleStream*
- Набор элементов, обрабатываемый оптом
  - Может не хранить элементы
  - Может быть ленивым
  - Может быть бесконечным
- Получение
  - Из коллекций и массивов
  - Генераторы
  - Строки из файлов и наборы файлов
  - Случайные потоки

# Примеры использования потоков

- `collection.stream()`  
    `.filter(s -> s.endsWith("s"))`  
    `.mapToInt(String::length)`  
    `.max();`
- `collection.parallelStream()`  
    `.filter(s -> s.contains("a"))`  
    `.sorted(String.CASE_INSENSITIVE_ORDER)`  
    `.limit(3)`  
    `.reduce((s1, s2) -> s1 + ", " + s2);`

# Операции над потоками

- Промежуточные (конвейерные)
  - Порождают поток
  - Ленивые
- Завершающие
  - Порождают значения
  - Жадные
- Как отличить?
  - Промежуточные возвращают Stream.
  - Терминальные возвращают что-то еще (или ничего).

# Типы операций

- Без состояния
  - Не зависят от других элементов
  - *map*
  - *filter*
- С состоянием
  - Зависят от других элементов
  - *sort*
  - *distinct*
- Обрывающие (short-circuit)
  - Могут прочесть часть потока
  - *limit*

# Способы создания стримов

1. Классический: Создание стрима из коллекции
  - `Collection<String> collection = Arrays.asList("a1", "a2", "a3");`  
`Stream<String> streamFromCollection = collection.stream();`
2. Создание стрима из значений
  - `Stream<String> streamFromValues = Stream.of("a1", "a2", "a3");`
3. Создание стрима из массива
  - `String[] array = {"a1","a2","a3"}; Stream<String>`  
`streamFromArrays = Arrays.stream(array);`
4. Создание стрима из файла (каждая строка в файле будет отдельным элементом в стриме)
  - `Stream<String> streamFromFile = Files.lines(Paths.get("file.txt"))`

# Способы создания стримов

## 5. Создание стрима из строки

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

## 6. С помощью `Stream.builder`

- `Stream.builder().add("a1").add("a2").add("a3").build()`

## 7. Создание параллельного стрима

- `Stream<String> stream = collection.parallelStream();`

## 8. Создание бесконечных стрима с помощью `Stream.iterate`

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

## 9. Создание бесконечных стрима с помощью `Stream.generate`

- `Stream<String> streamFromGenerate = Stream.generate(() -> "a1")`

# Методы работы со стримами

Java Stream API предлагает два вида методов:

1. Конвейерные (промежуточные) — возвращают другой stream, то есть работают как builder,
2. Терминальные — возвращают другой объект, такой как коллекция, примитивы, объекты, Optional и т.д.

У stream может быть сколько угодно вызовов конвейерных вызовов и в конце один терминальный, при этом все конвейерные методы выполняются лениво и пока не будет вызван терминальный метод никаких действий на самом деле не происходит! Так же как создать объект Thread или Runnable, но не вызвать у него start.

# Конвейерные методы

1. **filter** - отфильтровывает записи, возвращает только записи, соответствующие условию
  - `collection.stream().filter("a1"::equals).count()`
2. **skip** - позволяет пропустить N первых элементов
  - `collection.stream().skip(collection.size()-1).findFirst().orElse("1")`
3. **distinct** - возвращает стрим без дубликатов (проверка на равенство методом equals)
  - `collection.stream().distinct().collect(Collectors.toList())`
4. **map** - преобразует каждый элемент стрима
  - `collection.stream().map(s-> s + "_1").collect(Collectors.toList())`



# Конвейерные методы

5. **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())`
6. **limit** - позволяет ограничить выборку определенным количеством первых элементов
  - `collection.stream().limit(2).collect(Collectors.toList())`
7. **sorted** - позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator
  - `collection.stream().sorted().collect(Collectors.toList())`

# Конвейерные методы

8. **mapToInt, mapToDouble, mapToLong** - аналоги `map`, но возвращают числовой стрим (то есть стрим из числовых примитивов)
  - `collection.stream().mapToInt((s) -> Integer.parseInt(s)).toArray()`
9. **flatMap, flatMapToInt, flatMapToDouble, flatMapToLong** – создает из каждого элемента потока новые потоки по некому правилу и объединяем получившиеся потоки в один
  - `col.stream().flatMap((p)->Arrays.asList(p.split(",")).stream()).toArray(String[]::new)`

# Терминальные методы

1. **findFirst** - возвращает первый элемент из стрима (возвращает Optional)
  - `collection.stream().findFirst().orElse(«1»)`
2. **findAny** – возвращает любой подходящий элемент из стрима (возвращает Optional)
  - `collection.stream().findAny().orElse(«1»)`
3. **collect** – представление результатов в виде коллекций и других структур данных
  - `collection.stream().filter((s)->s.contains(«1»)).collect(Collectors.toList())`
4. **count** – возвращает количество элементов в стриме
  - `collection.stream().filter(«a1»::equals).count()`

# Терминальные методы

5. **anyMatch** - возвращает true, если условие выполняется хотя бы для одного элемента
  - `collection.stream().anyMatch(«a1»::equals)`
6. **noneMatch** - возвращает true, если условие не выполняется ни для одного элемента
  - `collection.stream().noneMatch(«a8»::equals)`
7. **allMatch** - возвращает true, если условие выполняется для всех элементов
  - `collection.stream().allMatch((s) -> s.contains(«1»))`
8. **min** - возвращает минимальный элемент, в качестве условия использует компаратор
  - `collection.stream().min(String::compareTo).get()`
9. **max** - возвращает максимальный элемент, в качестве условия использует компаратор
  - `collection.stream().max(String::compareTo).get()`

# Терминальные методы

**10. forEach** - применяет функцию к каждому объекту стрима, порядок при параллельном выполнении не гарантируется

- `set.stream().forEach((p) -> p.append("_1"));`

**11. forEachOrdered** - применяет функцию к каждому объекту стрима, сохранение порядка элементов гарантирует

- `list.stream().forEachOrdered((p) -> p.append("_new"));`

**12. toArray** - возвращает массив значений стрима

- `collection.stream().map(String::toUpperCase).toArray(String[]::new);`

# Параллельное исполнение

- Невмешательство (non-interference)
  - Нельзя изменять обрабатываемые данные
- Отсутствие состояния
  - Нельзя сохранять данные между вызовами
- Без побочных эффектов
  - Нельзя изменять контекст
- Упорядоченность
  - Результат — упорядоченные
  - Вызовы — нет

```
int sum = IntStream.iterate (1, n -> n + 1)
    .filter (n -> n % 5 == 0 && n % 2 != 0)
    .limit (10)
    .map (n -> n * n)
    .sum ();
```

```
Set <String> vocabulary = ...;  
Stream <String> stream1 = vocabulary.stream ();
```

```
BufferedReader reader = ...;  
Stream <String > stream2 = reader.lines ();
```

```
Path path = ...;  
Stream <Path> stream3 = Files.list ( path );  
Stream <Path> stream4 = Files.walk ( path );  
IntStream chars = "hello". chars ();
```



```
DoubleStream randomNumbers =  
    DoubleStream.generate(Math :: random);
```

```
IntStream integers =  
    IntStream.iterate (0, n -> n + 1);
```

```
IntStream smallIntegers =  
    IntStream.range (0, 100);
```

```
IntStream smallIntegers2 =  
    IntStream.rangeClosed (0, 100);
```

```
IntStream combinedStream =  
    IntStream.concat ( stream1 , stream2 );  
IntStream empty = IntStream.empty ();  
  
double [] array = ...;  
DoubleStream streamFromArray =  
    Arrays.stream(array);  
  
IntStream streamOfElements =  
    IntStream.of (2, 4, 6, 8, 10);
```

```
IntStream stream = ...;  
stream.filter (n -> n > 100)  
    .mapToObj ( Integer :: toString )  
    .flatMapToInt (s -> s. chars ())  
    .distinct ()  
    .sorted ()  
    .skip (3)  
    .limit (2);
```

```
IntStream stream1 = ...;  
stream1.forEach ( System.out :: println );
```

```
IntStream stream2 = ...;  
OptionalInt result = stream2.findFirst ();
```

```
Stream <String> stream3 = ...;  
boolean allStringsAreAtLeast10Chars =  
    stream3.allMatch (s -> s.length () > 10);
```

```
Stream <String > stream1 = ...;  
Optional <String > minString = stream1.min(  
    Comparator.comparing (  
        String::length, Integer::compare ));
```

```
IntStream stream2 = ...;  
int count = stream2.count ();
```

```
IntStream stream3 = ...;  
int sum = stream3.sum ();
```

```
Stream <String> stream1 = ...;  
List <String> list =  
    stream1.collect (Collectors.toList ());
```

```
Stream <BigInteger> bigInts = ...;  
BigInteger sum = bigInts.reduce (  
    BigInteger.ZERO, BigInteger::add);
```

```
public static BigInteger factorial ( int n) {  
    return IntStream.rangeClosed (1, n)  
        .mapToObj (i -> BigInteger.valueOf (i))  
        .reduce (BigInteger.ONE,  
                BigInteger::multiply);  
}
```

```
public static boolean isPalindrome (String s) {  
    StringBuilder leftToRight = new StringBuilder ();  
    s.chars().filter (Character :: isLetterOrDigit)  
        .map (Character::toLowerCase)  
        .forEach (leftToRight::appendCodePoint);  
    StringBuilder rightToLeft =  
        new StringBuilder (leftToRight.reverse());  
  
    return leftToRight.toString ()  
        .equals(rightToLeft.toString ());  
}
```



```
List<String> myList =  
    Arrays.asList("a1", "a2", "b1", "c2", "c1");  
  
myList  
    .stream()  
    .filter(s -> s.startsWith("c"))  
    .map(String::toUpperCase)  
    .sorted()  
    .forEach(System.out::println);
```

```
Arrays.asList("a1", "a2", "a3")  
    .stream()  
    .findFirst()  
    .ifPresent(System.out::println); // a1
```

```
Arrays.stream(new int[] {1, 2, 3})  
    .map(n -> 2 * n + 1)  
    .average()  
    .ifPresent(System.out::println); // 5.0
```

```
Stream.of("a1", "a2", "a3")  
    .map(s -> s.substring(1))  
    .mapToInt(Integer::parseInt)  
    .max()  
    .ifPresent(System.out::println); // 3
```

```
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);
```

```
// a1
```

```
// a2
```

```
// a3
```

```
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);
```

```
// a1
```

```
// a2
```

```
// a3
```

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

```
filter: d2
forEach: d2
filter: a2
forEach: a2
filter: b1
forEach: b1
filter: b3
forEach: b3
filter: c
forEach: c
```



```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("A");
    });
```

```
// map:      d2
// anyMatch: D2
// map:      a2
// anyMatch: A2
```

```

Stream.of("d2", "a2", "b1", "b3", "c")
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));

```

```

sort: a2; d2
sort: b1; a2
sort: b1; d2
sort: b1; a2
sort: b3; b1
sort: b3; d2
sort: c; b3
sort: c; d2
filter: a2
map: a2
forEach: A2
filter: b1
filter: b3
filter: c
filter: d2

```

```
Stream.of("d2", "a2", "b1", "b3", "c")  
    .filter(s -> {  
        System.out.println("filter: " + s);  
        return s.startsWith("a");  
    })  
    .sorted((s1, s2) -> {  
        System.out.printf("sort: %s; %s\n", s1, s2);  
        return s1.compareTo(s2);  
    })  
    .map(s -> {  
        System.out.println("map: " + s);  
        return s.toUpperCase();  
    })  
    .forEach(s -> System.out.println("forEach: " + s));
```

// filter: d2  
// filter: a2  
// filter: b1  
// filter: b3  
// filter: c  
// map: a2  
// forEach: A2

collect

```
class Person {  
    String name; int age;  
    Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public String toString() {  
        return name;  
    }  
}
```

```
List<Person> persons = Arrays.asList(  
    new Person("Max", 18),  
    new Person("Peter", 23),  
    new Person("Pamela", 23),  
    new Person("David", 12));
```

```
List<Person> filtered =  
    persons  
        .stream()  
        .filter(p -> p.name.startsWith("P"))  
        .collect(Collectors.toList());  
  
System.out.println(filtered); //[Peter, Pamela]
```

```
Map<Integer, List<Person>> personsByAge = persons
    .stream()
    .collect(Collectors.groupingBy(p -> p.age));

personsByAge.forEach(
    (age, p) -> System.out.format("age %s: %s\n",
        age, p));

// age 18: [Max]
// age 23: [Peter, Pamela]
// age 12: [David]
```

```
Double averageAge = persons
    .stream()
    .collect(Collectors.averagingInt(
        p -> p.age));

System.out.println(averageAge);    // 19.0
```



```
IntSummaryStatistics ageSummary =  
    persons  
        .stream()  
        .collect(Collectors.summarizingInt(  
            p -> p.age));
```

```
System.out.println(ageSummary);  
// IntSummaryStatistics{count=4, sum=76,  
min=12, average=19.000000, max=23}
```

```
String phrase = persons
    .stream()
    .filter(p -> p.age >= 18)
    .map(p -> p.name)
    .collect(
        Collectors.joining(
            " and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
//In Germany Max and Peter and Pamela are of legal age
```

```
Map<Integer, String> map = persons
    .stream()
    .collect(Collectors.toMap(
        p -> p.age,
        p -> p.name,
        (name1, name2) -> name1 + ";" +
name2));
```

```
System.out.println(map);
// {18=Max, 23=Peter;Pamela, 12=David}
```

```
Collector<Person, StringJoiner, String> personNameCollector =  
    Collector.of(  
        () -> new StringJoiner(" | "),           // supplier  
        (j, p) -> j.add(p.name.toUpperCase()),    // accumulator  
        (j1, j2) -> j1.merge(j2),                // combiner  
        StringJoiner::toString);                  // finisher
```

```
String names = persons  
    .stream()  
    .collect(personNameCollector);
```

```
System.out.println(names); // MAX | PETER | PAMELA | DAVID
```

FlatMap

```
class Foo {  
    String name;  
    List<Bar> bars = new ArrayList<>();  
  
    Foo(String name) {  
        this.name = name;  
    }  
}  
  
class Bar {  
    String name;  
  
    Bar(String name) {  
        this.name = name;  
    }  
}
```

```
List<Foo> foos = new ArrayList<>();
```

```
// create foos
```

```
IntStream
```

```
    .range(1, 4)
```

```
    .forEach(i -> foos.add(new Foo("Foo" + i)));
```

```
// create bars
```

```
foos.forEach(f ->
```

```
    IntStream
```

```
        .range(1, 4)
```

```
        .forEach(i -> f.bars.add(new Bar("Bar" + i  
+ " <- " + f.name))));
```

```
foos.stream()  
    .flatMap(f -> f.bars.stream())  
    .forEach(b -> System.out.println(b.name));
```

```
// Bar1 <- Foo1  
// Bar2 <- Foo1  
// Bar3 <- Foo1  
// Bar1 <- Foo2  
// Bar2 <- Foo2  
// Bar3 <- Foo2  
// Bar1 <- Foo3  
// Bar2 <- Foo3  
// Bar3 <- Foo3
```



```
IntStream.range(1, 4)
    .mapToObj(i -> new Foo("Foo" + i))
    .peek(f -> IntStream.range(1, 4)
        .mapToObj(i -> new Bar("Bar" + i +
            " <- " + f.name)))
    .forEach(f.bars::add))
    .flatMap(f -> f.bars.stream())
    .forEach(b -> System.out.println(b.name));
```