JDK8 – Streams and Collectors

TECHNOLOGIES THAT GUIDE INDUSTRY

What is new in JDK8

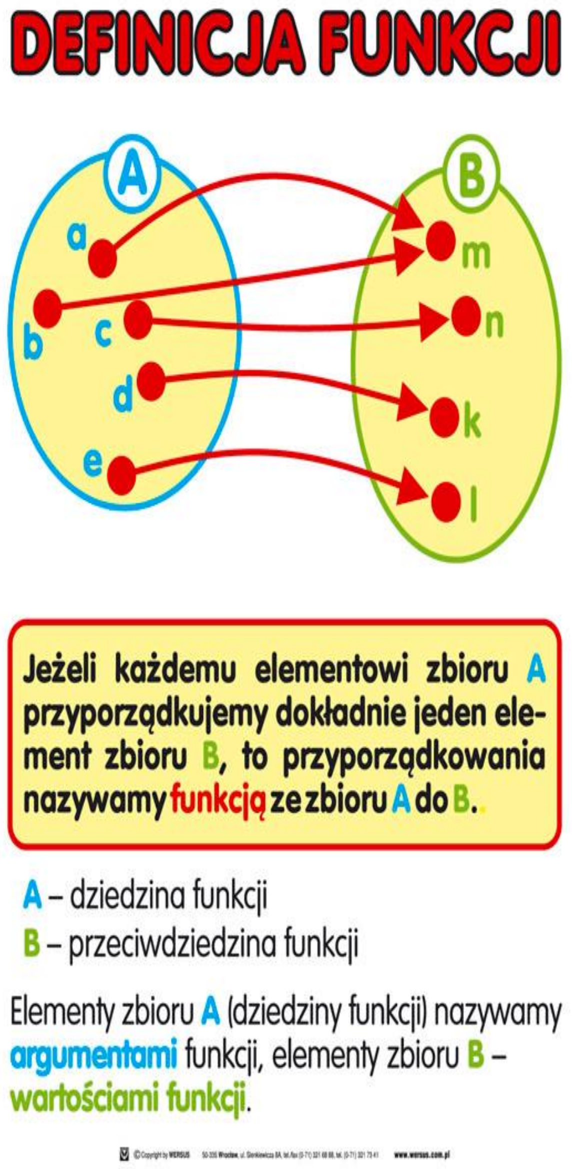
- Lambda expressions
  - enables to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly
- Method references
  - provide easy-to-read lambda expressions for methods that already have a name
- Default methods
  - enables new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces
- `java.util.stream` package
  - provide a Stream API to support functional-style operations on streams of elements. The Stream API is integrated into the Collections API, which enables bulk operations on collections, such as sequential or parallel map-reduce transformations
- Date-Time Package
  - a new set of packages that provide a comprehensive date-time model.
- And much more!

http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

Functional programming

- Programming paradigm
- Function is a king now!
- Coding style

- https://youtu.be/e-5obm1G_FY



DEFINICJA FUNKCJI

Jeżeli każdemu elementowi zbioru A przyporządkujemy dokładnie jeden element zbioru B, to przyporządkowania nazywamy funkcją ze zbioru A do B.

A – dziedzina funkcji
B – przeciwdziedzina funkcji

Elementy zbioru A (dziedziny funkcji) nazywamy argumentami funkcji, elementy zbioru B – wartościami funkcji.

Finite and Infinite stream

Dealing With The Indeterminate

- How to continue processing when we can't predict for how long?

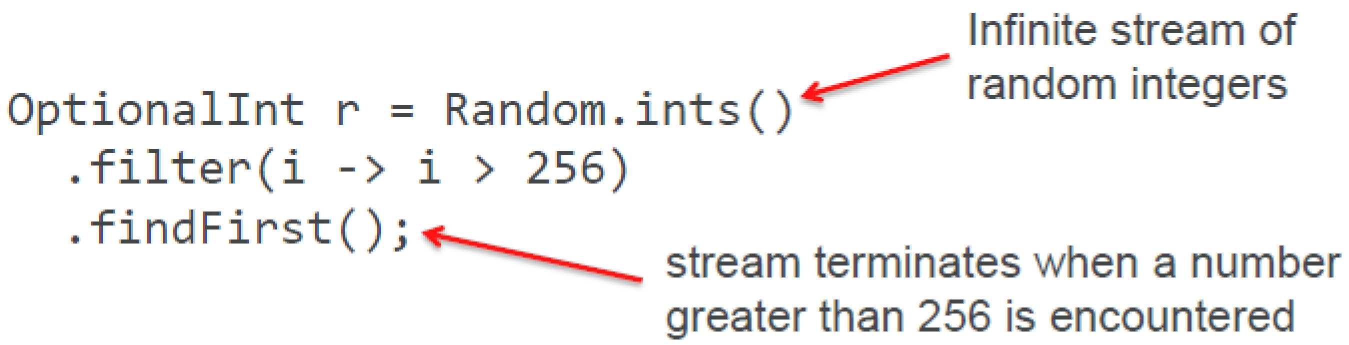
```
while (true) {
  doSomeProcessing();

  if (someCriteriaIsTrue())
    break;

  // Loop repeats indefinitely
}
```

Using Infinite Streams

- Terminate the stream when an element is read from the input stream
  - `findFirst()`
  - `findAny()`



```
OptionalInt r = Random.ints()
  .filter(i -> i > 256)
  .findFirst();
```

Infinite stream of random integers

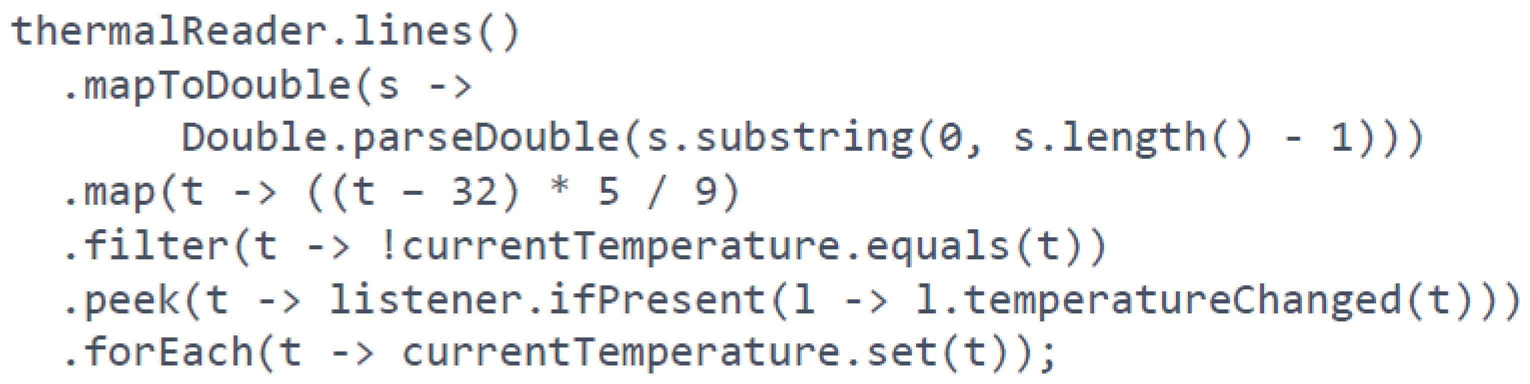
stream terminates when a number greater than 256 is encountered

Using Infinite Streams (cont'd)

- Sometimes we need to continue to use a stream indefinitely
- What terminal operation should we use for this?
  - Use `forEach()`
  - This consumes the element from the stream
  - But does not terminate it

Using Infinite Streams (cont’d)

- Reading temperature from a serial sensor
  - Converting from Farenheit to Celcius, removing F
  - Notifying a listener of changes if registered



```
thermalReader.lines()
   .mapToDouble(s ->
          Double.parseDouble(s.substring(0, s.length() - 1)))
   .map(t -> ((t – 32) * 5 / 9)
   .filter(t -> !currentTemperature.equals(t))
   .peek(t -> listener.ifPresent(l -> l.temperatureChanged(t)))
   .forEach(t -> currentTemperature.set(t));
```

Using Collectors

Collector Basics

- A `Collector` performs a mutable reduction on a stream
  - Accumulates input elements into a mutable result container
  - Results container can be a `List`, `Map`, `String`, etc
- Use the `collect()` method to terminate the stream
- `Collectors` utility class has many methods that can create a `Collector`

Composing Collectors

- Several `Collectors` methods have versions with a downstream collector
- Allows a second collector to be used
  - `collectingAndThen()`
  - `groupingBy()/groupingByConcurrent()`
  - `mapping()`
  - `partitioningBy()`

Collecting Into A Collection

- `toCollection(Supplier factory)`
  - Adds the elements of the stream to a `Collection` (created using factory)
  - Uses encounter order
- `toList()`
  - Adds the elements of the stream to a `List`
- `toSet()`
  - Adds the elements of the stream to a `Set`
  - Eliminates duplicates

Collecting To A Map

- `toMap(Function keyMapper, Function valueMapper)`
  - Creates a `Map` from the elements of the stream
  - *key* and *value* produced using provided functions

```
Map<String, String> myPeople = personList.stream()
            .collect(Collectors.toMap(Person::getSurname,
                        item -> concat(item)));
```

Collecting To A Map – Handling Duplicate Keys

- `toMap(Function keyMapper, Function valueMapper, BinaryOperator merge)`
- The same proces as first `toMap()` method
    - But uses the `BinaryOperator` to merge values for duplicate keys

```
Map<String, String> myPeople = personList.stream()
        .collect(Collectors.toMap(Person::getName,
                Person::getSurname,
                (x, y) -> x + " " + y));
```

Grouping Results

- `groupingBy(Function)`
  - Groups stream elements using the `Function` into a `Map`
  - Result is `Map<K, List<V>>`
  - `Map m = words.stream().collect(Collectors.groupingBy(String::length));`
- `groupingBy(Function, Collector)`
  - Groups stream elements using the `Function`
  - A reduction is performed on each group using the downstream `Collector`
  - `Map m = words.stream() .collect(Collectors.groupingBy(String::length, counting()));`

Parallel Streams

(And When Not To Use Them)

Serial And Parallel Streams

- Collection stream sources
  - `stream()`
  - `parallelStream()`
- Stream can be made parallel or sequential at any point
  - `parallel()`
  - `sequential()`
- The last call wins
  - Whole stream is either sequential or parallel

Parallel Streams

- Implemented internally using the fork-join framework
- Will default to as many threads for the pool as the OS reports processors
  - Which may not be what you want

```
System.setProperty(
"java.util.concurrent.ForkJoinPool.common.parallelism", "32767");
```

- Remember, parallel streams always need more work to process
  - But they might finish it more quickly

Parallel Stream Considerations

- `findFirst()` and `findAny()`
  - `findAny()` is non-deterministic, so better for parallel stream performance
  - Use `findFirst()` if a deterministic result is required
- `forEach()` and `forEachOrdered()`
  - `forEach()` is non-deterministic for a parallel stream and ordered data
  - Use `forEachOrdered()` if a deterministic result is required

No Simple Answer When To Use Parallel Streams

- Data set size is important, as is the type of data structure
  - `ArrayList`: GOOD
  - `HashSet`, `TreeSet`: OK
  - `LinkedList`: BAD
- Operations are also important
  - Certain operations decompose to parallel tasks better than others
  - `filter()` and `map()` are excellent
  - `sorted()` and `distinct()` do not decompose well