Java 8 Tutorial

In java 8, most talked about feature was lambda expressions. It has many other important features as well such as default methods, stream API and new date/time API.

Java 8 – forEach

The **Java forEach** is a utility method to iterate over a collection or stream and perform a certain action on each element of it.

Java program to iterate over all elements of a stream and perform an action. In this example, we are printing all even numbers.

```
ArrayList<Integer> numberList = new ArrayList<>(Arrays.asList(1,2,3,4,5));
Consumer<Integer> action = System.out::println;
numberList.forEach( action );
numberList.stream().forEach( action );
```

We can **create custom action** methods as well to perform our custom logic for each element in the collection.

```
HashMap<String, Integer> map = new HashMap<>();
map.put("A", 1);
map.put("B", 2);
map.put("C", 3);

Consumer<Map.Entry<String, Integer>> action = entry -> {
       System.out.println("Key is : " + entry.getKey());
       System.out.println("Value is : " + entry.getValue());
};
map.entrySet().forEach(action);
```

Java Stream

Another major change introduced **Java 8 Streams API**, which provides a mechanism for processing a set of data in various ways that can include filtering, transformation, or any other way that may be useful to an application.

```
List<String> filteredList = items.stream().filter(e -> (!e.startsWith
(“A”))).collect(Collectors.toList());
```

Here items.stream() indicates that we wish to have the data in the items collection processed using the Streams API.

A **Stream is a conceptually fixed data structure, in which elements are computed on demand**.

Stream **operations are either intermediate or terminal**. While **terminal operations return a result of a certain type**, **intermediate operations return the stream itself** so you can chain multiple method calls in a row. Streams are created on a source, e.g. a java.util.Collection like lists or sets (maps are not supported). Stream operations can either be executed sequential or parallel.

Different ways to create streams

```
Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);
stream.forEach(p -> System.out.println(p));

Stream<Integer> stream = list.stream();
stream.forEach(p -> System.out.println(p));

Stream<Date> stream = Stream.generate(() -> { return new Date(); });
stream.forEach(p -> System.out.println(p));
```

Convert streams to collections

```
Stream<Integer> stream = list.stream();
List<Integer> evenNumbersList = stream.filter(i -> i%2 ==0).collect
(Collectors.toList());
```

Convert Stream to array – Stream.toArray(EntryType[]::new)

```
Stream<Integer> stream = list.stream();
Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::
new);
System.out.print(evenNumbersArr);
```

Intermediate operations:

filter()
map()
flatMap()
distinct()
sorted()
peek()
limit()
skip()

Terminal operations:

forEach()
forEachOrdered()
toArray()
reduce()
collect()
min()
max()
count()
anyMatch()
allMatch()
noneMatch()
findFirst()
findAny()

Stream short-circuit operations

Though, stream operations are performed on all elements inside a collection satisfying a predicate, It is often desired to break the operation whenever a matching element is encountered during iteration. In external iteration, you will do with if-else block. In internal iteration, there are certain methods you can use for this purpose. Let's see example of two such methods:

Stream.anyMatch()
Stream.findFirst()

Parallelism in Java Steam:

To enable parallelism, all you have to do is to create a parallel stream, instead of sequential stream. And to surprise you, this is really very easy. In any of above listed stream examples, anytime you want to particular job using multiple threads in parallel cores, all you have to call method parallelStream()method instead of stream() method.

```
Stream<Integer> stream = list.parallelStream();
Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::
new);
System.out.print(evenNumbersArr);
```

Java Boxed Stream Example

In **Java 8**, if you want to convert stream of objects to collection, then you can use one of the static methods in the **Collectors** class.

```
//It works perfect !!
List<String> strings = Stream.of("how", "to", "do", "in", "java").collect
(Collectors.toList());
```

The same process doesn't work on streams of primitives, however.

```
//Compilation Error !!
IntStream.of(1,2,3,4,5).collect(Collectors.toList());
```

To convert a stream of primitives, you must first box the elements in their wrapper class and then collect them. This type of stream in called **boxe d stream**.

```
//Get the collection and later convert to stream to process elements
List<Integer> ints = IntStream.of(1,2,3,4,5)
                              .boxed()
                              .collect(Collectors.toList());
```

Lambda Expression

Lambda expression (or function) is just an *anonymous function*, i.e., a function with no name and without being bounded to an identifier. They are written exactly in the place where it's needed, typically *as a parameter to some other function*.

A typical lambda expression example will be like this:

(x, y) -> x + y //This function takes two parameters and return their sum.

Rules for writing lambda expressions:

1. A lambda expression can have zero, one or more parameters.

2. The type of the parameters can be explicitly declared or it can be inferred from the context.

3. Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.

4. When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a*a.

5. The body of the lambda expressions can contain zero, one or more statements.

6. If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in body than these must be enclosed in curly brackets.

Functional Interface

Functional interfaces are also called Single Abstract Method interfaces (SAM Interfaces). As name suggest, they permit exactly one abstract method inside them. Java 8 introduces an annotation i.e. @FunctionalInterface which can be used for compiler level errors when the interface you have annotated violates the contracts of Functional Interface.

A typical functional interface example:

```
@FunctionalInterface
public interface MyFirstFunctionalInterface {
        public void firstWork();
}
```

Please note that a functional interface is valid even if the **@FunctionalInterface** annotation would be omitted. It is only for informing the compiler to enforce single abstract method inside interface.

Also, since default methods are not abstract you're free to add default methods to your functional interface as many as you like.

Default Methods

Java 8 allows you to add non-abstract methods in interfaces. These methods must be declared default methods. Default methods were introduces in java 8 to enable the functionality of lambda expression.

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

Let's understand with an example:

```
public interface Moveable {
       default void move(){
                System.out.println("I am moving");
        }
}
```

If class willingly wants to customize the behavior of move() method then it can provide it's own custom implementation and override the method.

Optionals

All of us must have encountered **NullPointerException** in our applications. This exception happen when you try to utilize an object reference which has not been initialized, initialized with null or simply does not point to any instance. ***NULL simply means 'absence of a value'.***

Optional is a way of replacing a nullable T reference with a non-null value. An Optional may either contain a non-null T reference (in which case we say the reference is "present"), or it may contain nothing (in which case we say the reference is "absent").

```
Optional<Integer> canBeEmpty1 = Optional.of(5);
canBeEmpty1.isPresent();     // returns true
canBeEmpty1.get();              // returns 5

Optional<Integer> canBeEmpty2 = Optional.empty();
canBeEmpty2.isPresent();     // returns false
```

Creating Optional objects

There are 3 major ways to create an Optional.

1 Use Optional.empty() to create empty optional.
   Optional<Integer> possible = Optional.empty();
2 Use Optional.of() to create optional with default non-null value. If you pass null in of(), then a NullPointerException is thrown immediately.
   Optional<Integer> possible = Optional.of(5);
3 Use Optional.ofNullable() to create an Optional object that may hold a null value. If parameter is null, the resulting Optional object would be empty (remember that value is absent; don't read it null).
   Optional<Integer> possible = Optional.ofNullable(null); ***//or***
   Optional<Integer> possible = Optional.ofNullable(5);

Do something If Optional value is present

```
Optional<Integer> possible = Optional.of(5);
possible.ifPresent(System.out::println);
```

Default/absent values and actions

```
//Assume this value has returned from a method
Optional<Company> companyOptional = Optional.empty();
```

```
//Now check optional; if value is present then return it,
//else create a new Company object and retur it
Company company = companyOptional.orElse(new Company());

//OR you can throw an exception as well
Company company = companyOptional.orElseThrow(IllegalStateException::new);
```

Rejecting certain values using the filter method

```
Optional<Company> companyOptional = Optional.empty();
companyOptional.filter(department -> "Finance".equals(department.getName())
.ifPresent(() -> System.out.println("Finance is present"));
```

Please note that Optional is not meant to be used in these below contexts, as possibly it won't buy us anything:
• in the domain model layer (it's not serializable)
• in DTOs (it's not serializable)
• in input parameters of methods
• in constructor parameters

Predicates

In Java 8, **Predicate** is a **functional interface** and can therefore be used as the assignment target for a **lambda expression** or method reference. So, where you think, we can use these true/false returning functions in day to day programming? I will say you can use predicates anywhere where you need to evaluate a condition on group/collection of similar objects such that evaluation can result either in true or false.

For example, you can use prdicates in these realtime usecases
- Find all children borned after a particular date
- Pizzas ordered a specific time
- Employees greater than certain age and so on..

We can pass **lambda** expressions wherever **predicate** is expected. For example one such method is **filter()** method from Stream interface.

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

So, essentially we can use stream and predicate to:

- first filter certain elements from a group, and
- then perform some operation on filtered elements.

Example:

```
public static Predicate<Employee> isAdultMale() {
      return p -> p.getAge() > 21 && p.getGender().equalsIgnoreCase("M");
}

public static List<Employee> filterEmployees (List<Employee> employees,
Predicate<Employee> predicate){
      return employees.stream()
             .filter( predicate )
             .collect(Collectors.<Employee>toList());
```

```
}

System.out.println( filterEmployees(employees, isAdultMale()) );
```

Benefits of Predicate:
- They move your conditions (sometimes business logic) to a central place. This helps in unit-testing them separately.
- Any change need not be duplicated into multiple places. Java predicate improves code maintenance.
- The code e.g. “filterEmployees(employees, isAdultFemale())” is very much readable than writing a if-else block.

Regex as Predicate

Use **Pattern.compile().asPredicate()** method to get a predicate from compiled regular expression. This predicate can be used with lambda streams to apply it on each token into stream.

```
// Compile regex as predicate
Predicate<String> emailFilter = Pattern .compile("^(.+)@example.com$") .
asPredicate();

// Input list
List<String> emails = Arrays.asList("alex@example.com", "bob@yahoo.com",
"cat@google.com", "david@example.com");

// Apply predicate filter
List<String> desiredEmails = emails.stream() .filter(emailFilter) .collect
(Collectors.<String&gttoList());

// Now perform desired operation
desiredEmails.forEach(System.out::println);
```

Date Time API

Date class has even become obsolete. The new classes intended to replace Date class are LocalDate, LocalTime and LocalDateTime.

1. The LocalDate class represents a date. There is no representation of a time or time-zone.
2. The LocalTime class represents a time. There is no representation of a date or time-zone.
3. The LocalDateTime class represents a date-time. There is no representation of a time-zone.

If you want to use the date functionality with zone information, then Lambda provide you extra 3 classes similar to above one i.
e. OffsetDate, OffsetTime and OffsetDateTime. Timezone offset can be represented in “+05:30” or “Europe/Paris” formats. This is done via using another class i.e. ZoneId.

```
LocalDate localDate = LocalDate.now();
LocalTime localTime = LocalTime.of(12, 20);
LocalDateTime localDateTime = LocalDateTime.now();
```

Timestamp and Duration:

For representing the specific timestamp at any moment, the class needs to be used is `Instant`. The `Instant` class represents an instant in time to an accuracy of nanoseconds. Operations on an Instant include comparison to another `Instant` and adding or subtracting a duration.

```
Instant instant = Instant.now();
Instant instant1 = instant.plus(Duration.ofMillis(5000));
Instant instant2 = instant.minus(Duration.ofMillis(5000));
Instant instant3 = instant.minusSeconds(10);
```

`Duration` class is a whole new concept brought first time in java language. It represents the time difference between two time stamps.

```
Duration duration = Duration.ofMillis(5000);
duration = Duration.ofSeconds(60);
duration = Duration.ofMinutes(10);
```

`Duration` deals with small unit of time such as milliseconds, seconds, minutes and hour. They are more suitable for interacting with application code. To interact with human, you need to get **bigger durations** which are presented with `Period` class.

```
Period period = Period.ofDays(6);
period = Period.ofMonths(6);
period = Period.between(LocalDate.now(), LocalDate.now().plusDays(60));
```

WatchService API

Will learn to watch a directory along with all sub-directories and files inside it, using java 8 WatchService API.

To Register **WatchService**, get the directory path and use **path.register()** method.

```
Path path = Paths.get(".");
WatchService watchService = path.getFileSystem().newWatchService();
path.register(watchService, StandardWatchEventKinds.ENTRY_MODIFY);
```

Watching for change events

To get the changes occurred on directory and files inside it, use **watchKey.pollEvents()** method which return the collection of all change events in form of stream.

```
WatchKey watchKey = null;
while (true) {
       watchKey = watchService.poll(10, TimeUnit.MINUTES);
       if(watchKey != null) {
           watchKey.pollEvents().stream().forEach(event -> System.out.
println(event.context()));
       }
       watchKey.reset();
}
```

The key remains valid until:

- It is cancelled, explicitly, by invoking its cancel method, or
- Cancelled implicitly, because the object is no longer accessible, or
- By closing the watch service.

If you are reusing the same key to get change events multiple times inside a loop, then don't forget to call **watchKey.reset()** method which set the key in ready state again.

Join String Array

Method Syntax:

String join(CharSequence delimiter, CharSequence... elements)

Example:

```
String joinedString = String.join(", ", "How", "To", "Do", "In", "Java");
System.out.println(joinedString);

Output:
How, To, Do, In, Java
```

Java 8 – Join String

Example:

```
String joined = String.join("/","usr","local","bin");
System.out.println(joined);

Output:
usr/local/bin
```