**Java 8 forEach()**

**Example 1: Java program to iterate over a List using forEach()**

|  |
| --- |
| Using forEach() method |
| List<String> names = Arrays.asList("Alex", "Brian", "Charles");    names.forEach(System.out::println); |

Program Output:

|  |  |  |  |
| --- | --- | --- | --- |
| Alex  Brian  Charles   |  | | --- | | Creating consumer action | | List<String> names = Arrays.asList("Alex", "Brian", "Charles");    Consumer<String> makeUpperCase = new Consumer<String>()  {      @Override      public void accept(String t)      {          System.out.println(t.toUpperCase());      }  };    names.forEach(makeUpperCase); |   Program Output:   |  | | --- | | ALEX  BRIAN  CHARLES | |

**1. Lambda Expression**

Lambda expressions are not unknown to many of us who have worked on other popular programming languages like Scala. In Java programming language, a 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*.

The basic *syntax of a lambda expression* is:

|  |
| --- |
| either  (parameters) -> expression  Or  (parameters) -> { statements; }  Or    () -> expression |

A typical lambda expression example will be like this:

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

Please note that based on type of x and y, method may be used in multiple places. Parameters can match to int, or Integer or simply String also. Based on context, it will either add two integers or concat two strings.

**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.

**2. 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.

Another important point to remember is that if an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface’s abstract method count since any implementation of the interface will have an implementation from java.lang.Object or elsewhere. for example, below is perfectly valid functional interface.

|  |
| --- |
| @FunctionalInterface  public interface MyFirstFunctionalInterface  {      public void firstWork();        @Override      public String toString();                //Overridden from Object class        @Override      public boolean equals(Object obj);        //Overridden from Object class  } |

**3. 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");      }  } |

Moveable interface defines a method move() and provided a default implementation as well. If any class implements this interface then it need not to implement it’s own version of move() method. It can directly call instance.move(). e.g.

|  |
| --- |
| public class Animal implements Moveable{      public static void main(String[] args){          Animal tiger = new Animal();          tiger.move();      }  }    Output: 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.

**Example 1: Using lambda expression to iterate over a List and perform some action on list elements**

In the given example, we are iterating over the list and printing all the list elements in the standard output. We can perform any desired operation in place of printing them.

|  |
| --- |
| List<String> pointList = new ArrayList();    pointList.add("1");  pointList.add("2");    pointList.forEach( p ->  { System.out.println(p); } ); |

**4. Java 8 Streams**

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.

Streams API in Java 8 supports a different type of iteration where you simply define the set of items to be processed, the operation(s) to be performed on each item, and where the output of those operations is to be stored.

An example of stream API. In this example, items is collection of String values and you want to remove the entries that begin with some prefix text.

|  |
| --- |
| List<String> items;  String prefix;  List<String> filteredList = items.stream().filter(e -> (!e.startsWith(prefix))).collect(Collectors.toList()); |

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

processed using the Streams API.

**5. Java 8 Date/Time API Changes**

The new Date and Time APIs/classes (JSR-310), also called as *ThreeTen*, which have simply change the way you have been handling dates in java applications.

**Dates**

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();  OffsetDateTime offsetDateTime = OffsetDateTime.now();  ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId.of("Europe/Paris")); |

**Timestamp and Duration**

For representing the specific timestamp ant 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)); |