JDK 8 Tutorial

Table of Contents

1. Lambda Expression
2. Functional Interface
3. Default Methods
4. Streams
5. Date/Time API Changes

**Lambda Expression**

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** |

Let’s see some examples as well:

|  |
| --- |
| (int a, int b) ->    a \* b  // takes two integers and returns their multiplication  (a, b)        ->   a - b  // takes two numbers and returns their difference  () -> 99  // takes no values and returns 99  (String a) -> System.out.println(a)  // takes a string, prints its value to the console, and returns nothing  a -> 2 \* a  // takes a number and returns the result of doubling it  c -> { //some complex statements }  // takes a collection and do some processing |

**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.

**What is a functional interface?**

Single Abstract Method interfaces (SAM Interfaces) is not a new concept. It means interfaces with only one single method. In java, we already have many examples of such SAM interfaces. From java 8, they will also be referred as functional interfaces as well. Java 8, enforces the rule of single responsibility by marking these interfaces with a new annotation i.e. @FunctionalInterface.

For example, new definition of Runnable interface is like this:

|  |
| --- |
| @FunctionalInterface  public interface Runnable {      public abstract void run();  } |

If you try to add a new method in any functional interface, compiler would not allow you to do this and will throw compile time error.

So far so good. But, how they are related to Lambda expressions? Let’s find out the answer.

We know that Lambda expressions are anonymous functions with no name and they are passed (mostly) to other functions as parameters. Well, in java method parameters always have a type and this type information is looked for to determine which method needs to be called in case of method overloading or even simple method calling. So, basically every lambda expression also must be convertible to some type to be accepted as method parameters. Well, that type in which lambda expressions are converted, are always of functional interface type.

|  |
| --- |
| new Thread(new Runnable() {      @Override      public void run() {          System.out.println("howtodoinjava");      }  }).start(); |

If we use the lambda expression for this task then code will be :

|  |
| --- |
| new Thread(              () ->   {                          System.out.println("My Runnable");                      }           ).start(); |

**Few examples of Lambda expressions**

I am listing out some code samples which you can read and analyze to how a lambda expression can be used in day to day programming.

1) Iterating over a List and perform some operations

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

2) Create a new runnable and pass it to thread

|  |
| --- |
| new Thread(      () -> System.out.println("My Runnable");  ).start(); |

3) Sorting employees objects by their name

|  |
| --- |
| public class LambdaIntroduction {      public static void main (String[] ar){            Employee[] employees  = {                new Employee("David"),                new Employee("Naveen"),                new Employee("Alex"),                new Employee("Richard")};              System.out.println("Before Sorting Names: "+Arrays.toString(employees));            Arrays.sort(employees, Employee::nameCompare);            System.out.println("After Sorting Names "+Arrays.toString(employees));        }  }    class Employee {    String name;      Employee(String name) {      this.name = name;    }      public static int nameCompare(Employee a1, Employee a2) {      return a1.name.compareTo(a2.name);    }      public String toString() {      return name;    }  }    Output:    Before Sorting Names: [David, Naveen, Alex, Richard]  After Sorting Names [Alex, David, Naveen, Richard] |

4) Adding an event listener to a GUI component

|  |
| --- |
| JButton button =  new JButton("Submit");  button.addActionListener((e) -> {      System.out.println("Click event triggered !!");  }); |

**package** jdk8demos;

**import** java.sql.SQLException;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.Comparator;

**import** java.util.List;

**import** java.util.function.Consumer;

**import** java.util.stream.Collector;

**import** java.util.stream.Collectors;

**import** javax.sql.RowSet;

**import** javax.sql.rowset.Predicate;

**class** Employee {

String name;

Employee(String name) {

**this**.name = name;

}

**public** **static** **int** nameCompare(Employee a1, Employee a2) {

**return** a1.name.compareTo(a2.name);

}

**public** String toString() {

**return** name;

}

}

**public** **class** LambdaExpression1 {

**public** **static** **void** main(String[] args) {

List<String> pointList = **new** ArrayList();

pointList.add("1");

pointList.add("2");

//Before JDK8 version

pointList.forEach(**new** Consumer<String>() {

@Override

**public** **void** accept(String arg0) {

System.***out***.println(arg0);

}

});

//JDK 8

pointList.forEach(p -> { System.***out***.println(p); } );

//Before JDK8 version

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

System.***out***.println("New Thread Start from My Runnable");

}

});

//JDK 8

**new** Thread(() -> System.***out***.println("New Thread Start from My Runnable")).start();

Employee[] employees = {**new** Employee("David"), **new** Employee("Naveen"), **new** Employee("Alex"), **new** Employee("Richard")};

System.***out***.println("Before Sorting Names: "+Arrays.*toString*(employees));

Arrays.*sort*(employees, Employee::*nameCompare*);

Arrays.*sort*(employees,**new** Comparator<Employee>() {

**public** **int** compare(Employee arg0, Employee arg1) {

**return** arg0.name.compareTo(arg1.name);

}

});

Arrays.*sort*(employees, (ee1,ee2)->ee1.name.compareTo(ee2.name));

System.***out***.println("After Sorting Names "+Arrays.*toString*(employees));

List<String> items=**null**;

String prefix=**null**;

List<String> filteredList = (List<String>) items.stream().filter(**new** java.util.function.Predicate<String>() {

@Override

**public** **boolean** test(String arg0) {

**return** !arg0.startsWith(prefix);

}

}).collect(Collectors.*toList*());

}

}

**Types of Method References – Quick Overview**

*Java 8 has four types of method references.*

|  |  |  |
| --- | --- | --- |
| **METHOD REFERENCE** | **DESCRIPTION** | **EXAMPLE** |
| Reference to **static method** | Used to refer static methods from a class | Math::max equivalent to Math.max(x,y) |
| Reference to **instance method from instance** | Refer to an instance method using a reference to the supplied object | System.out::println equivalent to System.out.println(x) |
| Reference to **instance method from class type** | Invoke the instance method on a reference to an object supplied by the context | String::length equivalent to str.length() |
| Reference to **constructor** | Reference to a constructor | ArrayList::new equivalent to new ArrayList() |

**Reference to static method – Class::staticMethodName**

|  |
| --- |
| List<Integer> integers = Arrays.asList(1,12,433,5);    Optional<Integer> max = integers.stream().reduce( Math::max );    max.ifPresent(value -> System.out.println(value)); |

**Reference to instance method from instance – ClassInstance::instanceMethodName**

|  |
| --- |
| List<Integer> integers = Arrays.asList(1,12,433,5);    Optional<Integer> max = integers.stream().reduce( Math::max );    max.ifPresent( System.out::println ); |

**Reference to instance method from class type – Class::instanceMethodName**

|  |
| --- |
| List<String> strings = Arrays.asList("how", "to", "do", "in", "java", "dot", "com");    List<String> sorted = strings.stream().  sorted((s1, s2) -> s1.compareTo(s2)).collect(Collectors.toList());    System.out.println(sorted);    List<String> sortedAlt = strings.stream()  .sorted(String::compareTo).collect(Collectors.toList());    System.out.println(sortedAlt); |

**Reference to constructor – Class::new**

|  |
| --- |
| List<Integer> integers = IntStream.range(1, 100)  .boxed().collect(Collectors.toCollection( ArrayList::new ));    Optional<Integer> max = integers.stream().reduce(Math::max);    max.ifPresent(System.out::println); |

**package** jdk8demos;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.Optional;

**import** java.util.function.BinaryOperator;

**import** java.util.function.Consumer;

**import** java.util.stream.Collectors;

**import** java.util.stream.IntStream;

**public** **class** MethodReference2 {

**public** **static** **void** main(String[] args) {

//Reference to static method

List<Integer> integers = Arrays.*asList*(1,12,433,5);

//JDK 8

Optional<Integer> max = integers.stream().reduce(Math::*max*);

max.ifPresent(value -> System.***out***.println(value));

/\*new BinaryOperator<Integer>() {

@Override

public Integer apply(Integer arg0, Integer arg1) {

// **TODO** Auto-generated method stub

return null;

}

};\*/

max.ifPresent( System.***out***::println );

//Reference to instance method from instance

/\*new Consumer<Integer>() {

@Override

public void accept(Integer arg0) {

}

};\*/

//Reference to instance method from class type

List<String> strings = Arrays.*asList*("how", "to", "do", "in", "java", "dot", "com");

List<String> sorted = strings.stream().sorted((s1, s2) -> s1.compareTo(s2)).collect(Collectors.*toList*());

List<String> sortedAlt = strings.stream().sorted(String::compareTo).collect(Collectors.*toList*());

System.***out***.println(sortedAlt);

//Reference to constructor

List<Integer> integers1 = IntStream.*range*(1, 100).boxed().collect(Collectors.*toCollection*( ArrayList::**new** ));

Optional<Integer> max1 = integers1.stream().reduce(Math::*max*);

max.ifPresent(System.***out***::println);

}

}

## Java Default Methods

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();

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

|  |
| --- |
| public class Animal implements Moveable{        public void move(){          System.out.println("I am running");      }        public static void main(String[] args){          Animal tiger = new Animal();          tiger.move();      }  } |

This is not all done here. Best part comes as following benefits:

1. Static default methods: You can define static default methods in interface which will be available to all instances of class which implement this interface. This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class. This enables you to define methods out of your class and yet share with all child classes.
2. They provide you a highly desired capability of adding a capability to number of classes without even touching their code. Simply add a default method in interface which they all implement.

**Rules for this conflict resolution are as follows:**

1. Most preferred are the overridden methods in classes. They will be matched and called if found before matching anything.
2. The method with the same signature in the “most specific default-providing interface” is selected. This means if class Animal implements two interfaces i.e. Moveable and Walkable such that Walkable extends Moveable. Then Walkable is here most specific interface and default method will be chosen from here if method signature is matched.
3. If Moveable and Walkable are independent interfaces then a serious conflict condition happens, and compiler will complain then it is unable to decide. The you have to help compiler by providing extra info that from which interface the default method should be called. e.g.

|  |
| --- |
| Walkable.super.move();  //or  Moveable.super.move(); |

## What is the Type of Null?

in Java null is actually a type, a special one. It has no name so we cannot declare variables of its type or cast any variables to it; in fact there is only a single value that can be associated with it.

## What is wrong with just returning null?

the API designers put the descriptive java docs in APIs and mention there that API can return a null value, and in which case(s). Now, the problem is that the caller of the API might have missed reading the javadoc for any reason, and forget about handling the null case. This is going to be a bug in future for sure.

## How Java 8 Optionals provide the solution?

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 |

You can also **view Optional as a single-value container that either contains a value or doesn’t**.

It is important to note that the intention of the Optional class is not to replace every single null reference. Instead, its purpose is to help design more-comprehensible APIs so that by just reading the signature of a method, you can tell whether you can expect an optional value. This forces you to fetch the value from Optional and work on it, and at the same time handle the case where optional will be empty. Well, this is exactly the solution of null references/return values which ultimately result into NullPointerException.

## Creating Optional objects

There are 3 major ways to create an Optional.

**i)** Use **Optional.empty()** to create empty optional.

|  |
| --- |
| Optional<Integer> possible = Optional.empty(); |

**ii)** 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); |

**iii)** 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);

if(possible.isPresent()){

    System.out.println(possible.get());

}

## 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")); |

## What is inside Optional make it work?

|  |  |
| --- | --- |
| /\*\*   \* If non-null, the value; if null, indicates no value is present   \*/  private final T value; | |
| /\*\*   \* Common instance for {@code empty()}.   \*/  private static final Optional<?> EMPTY = new Optional<>(); | |
| this.value = Objects.requireNonNull(value); |
| public T get() {      if (value == null) {          throw new NoSuchElementException("No value present");      }      return value;  } |

## What is Optional trying to solve?

By using Optional, user is forced to think about the exceptional case. Besides the increase in readability that comes from giving null a name, the biggest advantage of Optional is its idiot-proof-ness. It forces you to actively think about the absent case if you want your program to compile at all, since you have to actively unwrap the Optional and address that failure cases.

## What is Optional not trying to solve?

Optional is not meant to be a mechanism to avoid all types of null pointers. e.g. The mandatory input parameters of methods and constructors will still have to be tested.

Like when using null, Optional does not help with conveying the meaning of an absent value. So the caller of the method will still have to check the javadoc of the API for understanding the meaning of the absent Optional, in order to deal with it properly.

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

## How should Optional be used?

**Optional** should be used almost all the time**as the return type of functions** that might not return a value.

## Java Predicate

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

1) Find all children borned after a particular date

2) Pizzas ordered a specific time

3) Employees greater than certain age and so on..

**How to use Predicate on a collection**

To demonstrate, we have an Employee class as below:

|  |
| --- |
| package predicateExample;    public class Employee {     public Employee(Integer id, Integer age, String gender, String fName, String lName){         this.id = id;         this.age = age;         this.gender = gender;         this.firstName = fName;         this.lastName = lName;     }     private Integer id;   private Integer age;   private String gender;     private String firstName;   private String lastName;       //Please generate Getter and Setters        @Override      public String toString() {    return this.id.toString()+" - "+this.age.toString(); //To change body of generated methods, choose Tools | Templates.      }  } |

You can build more of them as an when needed. So far so good. Far using above methods I have included above 3 methods in **EmployeePredicates.java** :

|  |
| --- |
| package predicateExample;    import java.util.List;  import java.util.function.Predicate;  import java.util.stream.Collectors;    public class EmployeePredicates  {      public static Predicate<Employee> isAdultMale() {          return p -> p.getAge() > 21 && p.getGender().equalsIgnoreCase("M");      }        public static Predicate<Employee> isAdultFemale() {          return p -> p.getAge() > 18 && p.getGender().equalsIgnoreCase("F");      }        public static Predicate<Employee> isAgeMoreThan(Integer age) {          return p -> p.getAge() > age;      }        public static List<Employee> filterEmployees (List<Employee> employees, Predicate<Employee> predicate) {          return employees.stream().filter( predicate ).collect(Collectors.<Employee>toList());      }  } |

## 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();  } |

*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  } |

## 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.

*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 |

## 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()); |
| List<String> items=null;  String prefix=null;  List<String> filteredList = (List<String>) items.stream().filter(new java.util.function.Predicate<String>() {  @Override  public boolean test(String arg0) {  return !arg0.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.*

## 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.*

|  |
| --- |
| 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 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)); |

# Method References with Example