

**New java 8 features:**

Lambda Expressions: Functional programming with lambda expressions.

Stream API: Improved collection management, parallel processing.

New Date/Time API: Simpler management of dates and times.

Nashorn java script engine.

Concurrency enhancements.

Joining String values.

Aggregating numbers.

# **Lambda Expressions**

* Lambda expressions are anonymous functions. Maintainable and readable.
* It is used mainly where one time use of the code.
* Lambda expressions can only appear in places where they will be assigned to a variable whose type is a functional interface.
* A functional interface has single abstract method.

Ex: Runnable, Callable, Comparator, TimerTask

Collections.sort(names, (String a, String b) -> {

return b.compareTo(a);

});

For one line method bodies you can skip both the braces **{}** and the **return** keyword. But it gets even shorter:

Collections.sort(names, (String a, String b) -> b.compareTo(a));

List now has a sort method. Also the java compiler is aware of the parameter types so you can skip them as well.

names.sort((a, b) -> b.compareTo(a));

## **Functional Interfaces**

Each lambda corresponds to a given type, specified by an interface. A so called *functional interface* must contain **exactly one abstract method** declaration. Each lambda expression of that type will be matched to this abstract method. Since default methods are not abstract you're **free to add default methods** to your functional interface. To ensure that your interface meet the requirements, you should add the **@FunctionalInterface** annotation. The compiler is aware of this annotation and throws a compiler error as soon as you try to add a second abstract method declaration to the interface.

It should have only one abstract method and it can have one default method also. @FunctionalInterface is optional.

|  |
| --- |
| Runnable r1 = new Runnable() {  @Override  public void run() {  System.out.println("Running Thread 1");  }  };  Thread t1 = new Thread(r1).start(); |
| JAVA 8:  Runnable r1 = () -> {  System.out.println("Running Thread 1");  }// took less code  Thread t1 = new Thread(r1).start(); |

|  |
| --- |
| Thread t1 = new Thread(new Runnable(){  @Override  public void run() {  System.out.println("Running Thread 1");  }  });  t1.start(); |
| JAVA 8:  Thread t1 = new Thread(() -> {  System.out.println("Running Thread 1");  });  t1.start(); |

|  |
| --- |
| Collections.sort(list, new Comparator<String>(){  @Override  public int compare(String str1, String str2)  {  return -str1.compareTo(str2);  }  }); |
| JAVA 8:  Collections.sort(list, (str1,str2) -> {  return -str1.compareTo(str2);  });  If list is List<String> str1 and str2 acts as a String. Or use below code.  Comparator<String> comp = (str1, str2) -> {  return -str1.compareTo(str2);  };  Collections.sort(list, comp); |

## 

## **forEach:**

Iterable Interface has forEach() method, List is implemented Iterable Interface.

[**forEach**](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html#forEach-java.util.function.Consumer-)([**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html)> action)

Consumer is functional interface. All functional interface are in java.util.function.

public interface **Consumer<T>**

It has one method[**accept**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept-T-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) t)

|  |
| --- |
| for(String str : list)  {  System.out.println(str);  } |
| JAVA 8:  list.forEach((str) -> {  System.out.println(str);  }); |

map.forEach((k, v) -> System.***out***.println("Key " + k + " Value " + v));

public interface **BiConsumer<T,U>**

It has one method accept(T t, U u)

We can’t use stream() and filter() for map.

## **Lambda Scopes**

Accessing outer scope variables from lambda expressions is very similar to anonymous objects. You can access final variables from the local outer scope as well as instance fields and static variables.

### **Accessing local variables**

We can read final local variables from the outer scope of lambda expressions:

int num = 1;

Converter<Integer, String> stringConverter =

(from) -> String.valueOf(from + num);

### **Accessing fields and static variables**

In contrast to local variables, we have both read and write access to instance fields and static variables from within lambda expressions. This behavior is well known from anonymous objects.

class Lambda4 {

static int outerStaticNum;

int outerNum;

void testScopes() {

Converter<Integer, String> stringConverter1 = (from) -> {

outerNum = 23;

return String.valueOf(from);

};

Converter<Integer, String> stringConverter2 = (from) -> {

outerStaticNum = 72;

return String.valueOf(from);

};

}

}

### **Accessing Default Interface Methods**

Remember the formula example from the first section? Interface Formula defines a default method sqrt() which can be accessed from each formula instance including anonymous objects. This does not work with lambda expressions.

Default methods **cannot** be accessed from within lambda expressions. The following code does not compile:

Formula formula = (a) -> sqrt(a \* 100);

## **Predicate:**

Predicates are boolean-valued functions of one argument. The interface contains various default methods for composing predicates to complex logical terms (and, or, negate)

Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo"); // true

predicate.negate().test("foo"); // false

Predicate<Boolean> nonNull = Objects::nonNull;

Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;

Predicate<String> isNotEmpty = isEmpty.negate();

Predicate is functional interface and it has test(T t) method.

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface **Predicate<T>**

boolean test([T](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html) t)

Predicate<Person> predicate = (p) -> {

return p.getAge() >= 18;

};

list.forEach(person -> {

if(predicate.test(person))

System.out.println(person);

});

Or

Predicate<Person> predicate = (p) -> p.getAge() >= 18;

### **Functions**

Functions accept one argument and produce a result. Default methods can be used to chain multiple functions together (compose, andThen).

Function<String, Integer> toInteger = Integer::valueOf;

Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123"); // "123"

### **Suppliers**

Suppliers produce a result of a given generic type. Unlike Functions, Suppliers don't accept arguments.

Supplier<Person> personSupplier = Person::new;

personSupplier.get(); // new Person

### **Consumers**

Consumers represent operations to be performed on a single input argument.

Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);

greeter.accept(new Person("Luke", "Skywalker"));

### **Comparators**

Comparators are well known from older versions of Java. Java 8 adds various default methods to the interface.

Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");

Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2); // > 0

comparator.reversed().compare(p1, p2); // < 0

### **Optional**

Optional are not functional interfaces, but nifty utilities to prevent NullPointerException. Instead of returning null you return an Optional in Java 8.

Optional<String> optional = Optional.of("bam");

optional.isPresent(); // true

optional.get(); // "bam"

optional.orElse("fallback"); // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0))); // "b"

**To create an empty Optional object:**

Optional<String> empty = Optional.empty();

empty.isPresent(); //false

The *isPresent()* API method is used to check if there is a value inside the *Optional* object. A value is present if and only if we have created *Optional* with a non-null value.

**We can also create an Optional object with the static of() method:**

String name = "baeldung";

Optional<String> opt = Optional.of(name);

if(opt.isPresent()) {

System.out.println(opt.get()); // baeldung

}

The argument passed to the *of()* method cannot be *null*, otherwise, we will get a *NullPointerException*:

String name = null;

Optional<String> opt = Optional.of(name); // *NullPointerException*

But, in case we expect some null values for the passed in argument, we can use the *ofNullable* API:

String name = "baeldung";

Optional<String> opt = Optional.ofNullable(name);

if(opt.isPresent()) {

System.out.println(opt.get()); // baeldung

}

In this way, if we pass in a null reference, it does not throw an exception but rather returns an empty *Optional* object as though we created it with the *Optional.empty* API:

String name = null;

Optional<String> opt = Optional.ofNullable(name);

opt.isPresent(); // false

**Conditional Action With ifPresent():**

Before *Optional*, we would do something like this:

if(name != null){

    System.out.println(name.length);

}

One could easily forget to perform the null checks in some part of the code. This can result in a *NullPointerException* at runtime.

Optional<String> opt = Optional.of("baeldung");

opt.ifPresent(name -> System.out.println(name.length()));

**Default Value With orElse:**

It takes one parameter which acts as a default value.

String nullName = null;

String name = Optional.ofNullable(nullName).orElse("john");

**Default Value With orElseGet:**

The *orElseGet* API is similar to *orElse*. However, instead of taking a value to return if the *Optional* value is not present, it takes a supplier functional interface which is invoked and returns the value of the invocation:

String nullName = null;

String name = Optional.ofNullable(nullName).orElseGet(() -> "john");

**Difference Between orElse and orElseGet:**

public String getMyDefault() {

    System.out.println("Getting Default Value");

    return "Default Value";

}

String text;

System.out.println("Using orElseGet:");

String defaultText = Optional.ofNullable(text).orElseGet(this::getMyDefault);

assertEquals("Default Value", defaultText); // true

System.out.println("Using orElse:");

defaultText = Optional.ofNullable(text).orElse(getMyDefault());

assertEquals("Default Value", defaultText); // true

When the wrapped value is not present, then both *orElse* and *orElseGet* APIs work exactly the same way.

String text = "Text present";

System.out.println("Using orElseGet:");

String defaultText = Optional.ofNullable(text).orElseGet(this::getMyDefault);

assertEquals("Text present", defaultText);//true getMyDefault() will not call

System.out.println("Using orElse:");

defaultText = Optional.ofNullable(text).orElse(getMyDefault());

assertEquals("Text present", defaultText); // ture getMyDefault will call

Notice that when using *orElseGet* to retrieve the wrapped value, the *getMyDefault* API is not even invoked since the contained value is present.

However, when using *orElse*, whether the wrapped value is present or not, the default object is created. So in this case, we have just created one redundant object that is never used.

**Exceptions with orElseThrow:**

Instead of returning a default value when the wrapped value is not present, it throws an exception:

String nullName = null;

String name = Optional.ofNullable(nullName).orElseThrow(IllegalArgumentException::new);

Method references in Java 8 come in handy here, to pass in the exception constructor.

**Returning Value with get():**

Optional<String> opt = Optional.of("baeldung");

String name = opt.get(); // baeldung

The get() API method can only return a value if the wrapped object is not null, otherwise, it throws a no such element exception.

**Conditional Return with filter():**

The *filter* API is used to run an inline test on the wrapped value. It takes a predicate as an argument and returns an *Optional* object.

Optional<String> opt = Optional.of("baeldung");

String name = opt.get(); // baeldung

String name = "muni";

if(name != null && name.equals("muni")) {

System.out.println(name);

}

Optional<String> optional = Optional.ofNullable(name);

if(optional.filter(n -> n.equals("muni")).isPresent()) {

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

}

Exp:

public boolean priceIsInRange2(Modem modem2) {

     return Optional.ofNullable(modem2)

       .map(Modem::getPrice)

       .filter(p -> p >= 10)

       .filter(p -> p <= 15)

       .isPresent();

 }

The *map* call is simply used to transform a value to some other value. Keep in mind that this operation does not modify the original value.

**Transforming Value with map():**

The *map* API returns the result of the computation wrapped inside *Optional*. We then have to call an appropriate API on the returned *Optional* to retrieve its value.

String name = "baeldung";

Optional<String> nameOptional = Optional.of(name);

int len = nameOptional

     .map(String::length())

     .orElse(0);

We can chain *map* and *filter* together to do something more powerful.

String password = " password ";

Optional<String> passOpt = Optional.of(password);

boolean correctPassword = passOpt

      .map(String::trim)

      .filter(pass -> pass.equals("password"))

      .isPresent();

**Transforming Value with flatMap():**

## **Default Method in Interfaces**

Java 8 enables us to add **non-abstract method implementation** to interfaces by utilizing the **default** keyword.

interface Formula {

double calculate(int a);

default double sqrt(int a) {

return Math.sqrt(a);

}

}

Concrete classes have to implement the only abstract method calculate(). The default method sqrt() can be used out of the box.

Formula formula = (a) -> {

**return** a + 2;

};

formula.calculate(2); // 4.0

formula.sqrt(16); // 4.0

## **Static Method in Interface:**

public interface Interface1 {

public String m1();

default public String m2(){

return "default method";

}

static public String m3(){

return "static method";

}

}

Subclass can’t override m3() method, any class can use m3() method by Interface1.m3().

## **Method References: (::)**

@FunctionalInterface

interface Converter<F, T> {

T convert(F from);

}

Converter<String, Integer> converter = (from) -> Integer.valueOf(from);

Integer converted = converter.convert("123");

System.out.println(converted); // 123

The above example code can be further simplified by utilizing static method references:

Converter<String, Integer> converter = Integer::valueOf;

Integer converted = converter.convert("123");

System.out.println(converted); // 123

# **Stream API**

A collection is an in-memory data structure to hold values and before we start using collection, all the values should have been populated. Whereas a java Stream is a data structure that is computed on-demand.

Print number greater than 3.

1. list.stream().filter(p -> p > 4).forEach(p -> System.***out***.print(p));
2. list.stream().filter(p -> p > 3).forEach(System.***out***::println);
3. Stream<String> stream = Stream.*of*(**new** String[]{"muni","swamy","palla"});

stream.forEach(str -> System.***out***.println(str));

1. Stream<String> stream = Arrays.*stream*(**new** String[]{"muni","swamy","palla"});

stream.forEach(str -> System.***out***.println(str));

1. **long** size = list.stream().count();
2. **long** sum = list.stream().mapToInt(p -> p).sum();

Parallel stream useful when you are doing aggregate operations. Parallel will works divide concur algorithm it will split the date into different units and finally collects the data shows the result.

1. **long** size = list.stream().parallel().count();
2. **long** sum = list.stream().parallel().mapToInt(p -> p.getAge()).sum();

# **New Date API**

New date API is immutable and thread safe.

LocalDate localDate = LocalDate.*now*();

LocalTime localTime = LocalTime.*now*();

LocalDateTime localDateTime = LocalDateTime.*now*();

String string = String.*join*(",", "muni","swamy","palla");//muni,swamy,palla

String string = String.*join*(" and ", "muni","swamy","palla");// u can use String[] also. Or you can use StringJoiner class.

StringJoiner joiner = **new** StringJoiner(",");

joiner.add("muni").add("swamy").add("palla");// muni,swamy,palla