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

|  |  |
| --- | --- |
| Runnable r1 = new Runnable() {  @Override  public void run() {  System.out.println("Running Thread 1");  }  };  Thread t1 = new Thread(r1).start(); | 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(); | 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);  }  }); | 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 (I) has forEach() method list is implemented Iterable (I)

[**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.

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

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);  } | list.forEach((str) -> {  System.out.println(str);  }); |

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

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

List<Integer> result = list.stream().filter(e -> e > 5).

collect(Collectors.*toList*());

List<Person> persons = Arrays.asList(

new Person("mkyong", 30),

new Person("jack", 20),

new Person("lawrence", 40)

);

Person result1 = persons.stream().filter(x -> "jack".equals(x.getName())) .findAny()// If 'findAny' then return found

.orElse(null); // If not found, return null

**Predicate:**

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;

**Method References: (::)**

**Default Method in Interface:**

**public** **interface** Interface1 {

**public** String m1();

**default** **public** String m2()

{

**return** "muni";// public is optional

}

}

Subclass can override m2() method and can provide his own implementation. Or he can use default Interface1 m2() method implementation.

**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().

# **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