Java 8

With the Java 8 release, Java provided supports for functional programming, New JavaScript engine, new APIs for date time manipulation, new streaming API, etc.

List Sorting Java 8:

```
List<Integer> list = new ArrayList<>();
        list.add(90);
        list.add(80);
        list.add(70);
System.out.println("Before Sorting : "+ list);
Collections.sort(list,(s1,s2) -> s1.compareTo(s2));
System.out.println("Aftetr Sorting : "+ list);
```

Lambda Expressions

The Lambda expression is used provide the implementation of an interface which has Functional Interface. Java lambda expression is treated as function, so compiler does not create .class file. Use of lambda expression:

1. To provide the implementation of Functional interface.
2. Less coding.

Syntax: - (argument-list) -> {body}

Java lambda expression is consisted of three components.

1. Argument-list:
   It can be empty or non-empty as well.
2. Arrow-token:
   It is used to link arguments-list and body of expression.
3. Body:
   It contains expressions and statements for lambda expression.

Functional Interface (Overview)

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an annotation @FunctionalInterface, which is used to declare an interface as functional interface.

Scenarios of with and without lambda:

Without lambda:

```
@FunctionalInterface
interface Drawer{
        public void draw();
}
```

```
Main(){
Drawer d = new Drawer() {
                        @Override
                        public void draw() {
                                System.out.println("This is draw");
                                }
                };d.draw();}
```

Using Lambda no parameter:

```
Drawer d_lamda = () -> {System.out.println("This is draw by lambda");};
                d_lamda.draw();
```

1. **Lambda expression using one parameter :**

```
    @FunctionalInterface
    Interface SayHello{
    public void say(String name);
    }
    Main() {
SayHello sayHello = (name) -> {System.out.println("Hello "+name);};
sayHello.say("Vikash");
}
```

2. **Lambda using One parameter and return type:**

```
@FunctionalInterface
interface SayHello2{
    public String say(String name);
    }
    SayHello2 say = (name) -> {return "Hello "+name;};
                    System.out.println(say.say("Vikash"));
```

3. **Using multiple parameter:**

```
@FunctionalInterface
interface SayHello{
    public void say(String str,String name);
    }
SayHello say = (str,name) -> {System.out.println(str+name);};
                    say.say("Hello ", "Vikash");
```

4. **Creating Thread:**

```
    Thread t1 = new Thread(() -> {System.out.println("Hello from Java8 thread");});
                    t1.start();
    OR,
    Runnable r1 = () -> {System.out.println("Hello from Java8 thread");};
    Thread t2 = new Thread(r1); t2.start();
```

5. **Lambda Comparator:**

```
    List<Integer> list = new ArrayList<>();
            list.add(90);
                    list.add(80);
                    list.add(70);
    System.out.println("Before Sorting : "+ list);
                    Collections.sort(list,(s1,s2) -> s1.compareTo(s2));
            System.out.println("Aftetr Sorting : "+ list);
```

6. **Filter Collection Data:**

```
List<Product> list=new ArrayList<Product>();
    list.add(new Product(1,"Samsung A5",17000f));
    list.add(new Product(3,"Iphone 6S",65000f));
```

```
        list.add(new Product(2,"Sony Xperia",25000f));
Stream<Product> filtedProducts = list.stream().filter((product) -> product.getPrice() > 20000);
filtedProducts.forEach((p) -> {System.out.println(p.getId()+" "+p.getName()+" "+p.getPrice());});
```

Following are the important characteristics of a lambda expression

4. Optional type declaration
5. Optional parenthesis around parameter
6. Optional curly braces
7. Optional return keyword

Java Method References

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

Types of Method References

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

Reference to a Static Method

ContainingClass::staticMethodName:

```
interface SayHello{
      public void say();
}
class MethodReference{
      public static void sayHello(){
            System.out.println("Hello from Method reference.");
      }}
Use In main:
SayHello say = MethodReference :: sayHello;
            say.say();
```

Note:

We will use redefined functional interface Runnable to refer static method.

```
Thread t1 = new Thread(MethodReference::sayHello);
            t1.start();
```

We can also override static methods by referring methods. In the following example, we have defined and overloaded three add methods.

```
class Adder{
      public static int add(int a, int b){
            System.out.println("Int Calling");
            return a+b;
      }
      public static float add(int a, float b){
            System.out.println("Single float Calling");
            return a+b;
```

```
	}
	public static float add(float a, float b){
		System.out.println("float Calling");
		return a+b;
	}
}
public static void main(String[] args) {
		/* @param <T> the type of the first argument to the function
		 @param <U> the type of the second argument to the function
		 @param <R> the type of the result of the function*/
		//BiFunction<T, U, R>
		BiFunction<Integer, Integer, Integer> adder1 = Adder::add;
		int result1 = adder1.apply(10, 10);
		BiFunction<Integer, Float, Float> adder2 = Adder::add;
		float result2 = adder2.apply(10, 10.9f);
		BiFunction<Float, Float, Float> adder3 = Adder::add;
		float result3 = adder3.apply(10.0f, 10.9f);
		System.out.println(result1);
		System.out.println(result2);
		System.out.println(result3);
	}
```

Reference to an Instance Method

```
MethodReference mr = new MethodReference();
		SayHello say1 = mr::add;
```

Reference to a Constructor

```
@FunctionalInterface
interface SayHello{
	public void say(String name);
}
class MethodReference{

	MethodReference(){
	}

	MethodReference(String name){
		System.out.println("Hello "+name);
	}}
SayHelloMR2 say2 = MethodReference :: new;
		say2.say("Vikash");
```

Java Functional Interface

An Interface that contains exactly one abstract method is known as functional Interface, that contains exactly one abstract method is known as functional Interface. **It can have any number of default, static methods** but can contains only abstract methods. It can also declare methods of object class.

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interface. It is a new feature in java, which helps to achieve functional programming approach.

A functional interface can have methods of object class. See in the following example:

```
@FunctionalInterface
interface sayable{
void say(String msg);  // abstract method
// It can contain any number of Object class methods.
```

```
int hashCode();
String toString();
boolean equals(Object obj);
}
```

Invalid functional Interface:

Functional Interface can extends another interface only when it does not have Invalid Functional Interface.

```
interface sayable{
void say(String msg);   // abstract method
}
@FunctionalInterface
interface Doable extends sayable{
   // Invalid '@FunctionalInterface' annotation; Doable is not a functional interface
   void doIt();
}
```

Note: - In this case we will get compile time error.

Following is the list of functional interfaces defined in java.util.Function package:

BiConsumer<T,U>
BiFunction<T,U,R>
BinaryOperator<T>
BiPredicate<T,U>
BooleanSupplier
Consumer<T>
DoubleBinaryOperator
DoubleConsumer
DoubleFunction<R>
DoublePredicate
DoubleSupplier
DoubleToIntFunction
DoubleToLongFunction
DoubleUnaryOperator
Function<T,R>
IntBinaryOperator
IntConsumer
IntFunction<R>
IntPredicate
IntSupplier
IntToDoubleFunction
IntToLongFunction
IntUnaryOperator
LongBinaryOperator
Predicate<T>
Supplier<T>
UnaryOperator<T>

Stream

Stream is a new abstract layer introduced in Java 8. Using stream, you can process data in a declarative way similar to SQL statements. For example, consider the following SQL statement.

Select max(salary),employee.id,employee.name from Employee employee;

The above SQL expression automatically returns the maximum salaried employee's details, without doing any computation on the developer's end. Using collections framework in Java, a developer has to use loops and make repeated checks. Another concern is efficiency; as multi-core processors are available at ease, a Java developer has to write parallel code processing that can be pretty error-prone. To resolve such issues, Java 8 introduced the concept of stream that lets the developer to process data declaratively and leverage multicore architecture without the need to write any specific code for it.

Stream represents a sequence of objects from a source, which supports aggregate operations. **Following are the characteristics of a Stream:**

Sequence of elements:

A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.

Source Stream takes Collections, Arrays, or I/O resources as input source.

Aggregate operations:

Stream supports aggregate operations like f**ilter, map, limit, reduce, find,match**, and so on.

Pipelining:

Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.

Automatic iterations:

Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

1. Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
2. Stream is functional in nature. Operations performed on a stream does not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
3. Stream is lazy and evaluates code only when required.

4. The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

Generating Streams:

With Java 8, Collection interface has two methods to generate a Stream.

1. **stream():** Returns a sequential stream considering collection as its source.
2. **parallelStream()** – Returns a parallel Stream considering collection as its source.

```
3. List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

4. List<String> filtered = strings.stream().filter(string ->
   !string.isEmpty()).collect(Collectors.toList());
```

Important Method of Stream API:

1. **ForEach():** void forEach(Consumer<? super T> action)

Internal understanding:

```
List<String> names = new ArrayList<>();
names.add("Larry");
names.add("Steve");
//Normal interface implementation overriding method
names.forEach(new Consumer<String>() {
   public void accept(String name) {
      System.out.println(name);
   }
});
```

OR,

We can use lambda expression:

```
Consumer<String> consumerNames = name -> {
   System.out.println(name);
};
names.forEach(consumerNames);
```

So, In terms of functional interface we can write above statement as:

```
name.forEach((name) -> System.out.println(name));
```

Or,

```
name.forEach((name) -> {System.out.println(name);});
```

Note: If we will use curly braces then we will have to terminate the line but for single line i.e without curly braces no need of termination.

Conclusion:

As we can see forEach() resides in Iterable interface but ArrayList and other collections are just overriding the forEach() method accordingly and forEach() taking one argument as Consumer interface which is functional interface so according to functional interface rule we can just use lambda expression (Implementation) in place of writing Implementation code for overriding Interface method. For reference see above snaps.

Examples of ForEach():

1. How to print 10 random numbers using forEach.
   ```
   Random random = new Random();
   random.inti().limit(10).forEach(System.out::println);
   ```

2. Map()

The 'map' method is used to map each element to its corresponding result.
map() is used to **transform one object into other by applying a function.** It is also an intermediate Stream operation which means you can call other Stream methods, like a filter, or collect on this to create a chain of transformations.
For example, by using the map() function, you can convert a list of String into a List of Integer by applying the Integer.valueOf() method to each String on the input list.

```
<R>Stream<R>map(Function<? super T, ? extends R> mapper);
```

Example:
prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

        List<Integer> squaresList = numbers.stream().map(x ->
x*x).distinct().collect(Collectors.toList());

squaresList.forEach(System.out::println);
```

3. Filter()

The 'filter' method is used to eliminate elements based on a criteria.

For example, if your list contains numbers and you only want numbers, then you can use the filter method to only select a number that is fully divisible by two. The filter method essentially select elements based upon a condition you provide. That's the reason that the filter (Predicate condition) accepts a Predicate object, which provides a function that is applied to a condition. If the condition evaluates true, then the object is selected. Otherwise, it will be ignored.

Similar to map, the filter is also an intermediate operation which means you can call other Stream methods after calling the filter.

The filter() method is also **lazy** which means that it will not be evaluated until you call a reduction method, like collect, and it will stop as soon as it reaches the target.

```
List<String> str = Arrays.asList("3", "2", "2", "3","7","3","5","8","6");
        System.out.println("String to integer");
        List<Integer> integerList = str.stream().map(x ->
Integer.parseInt(x)).distinct().collect(Collectors.toList());
        integerList.forEach(System.out::println);
```

Or,

```
str.stream().map(x -> Integer.parseInt(x)).filter(x -> ( x%2 == 0
)).distinct().collect(Collectors.toList()).forEach(System.out::println);
```

==

Count of Empty String:

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl","");
        int count = (int) strings.stream().filter(x -> x.isEmpty()).count();
        System.out.println(count);
```

4. Limit()

The 'limit' method is used to reduce the size of the stream.

```
Random random = new Random();

random.ints().limit(10).forEach(System.out::println)
```

5. Sorted()

The 'sorted' method is used to sort the stream.

```
Random random = new Random();

random.ints().limit(10).sorted().forEach(System.out::println);
```

6. Parallel Processing

parallelStream is the alternative of stream for parallel processing.

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");



//get count of empty string

int count = strings.parallelStream().filter(string -> string.isEmpty()).count();
```

Note: when we should go with this (from stackoverflow)
A parallel stream has a much higher overhead compared to a sequential one. Coordinating the threads takes a significant amount of time. I would use sequential streams by default and only consider parallel ones if

- I have a massive amount of items to process (or the processing of each item takes time and is parallelizable)
- I have a performance problem in the first place
- I don't already run the process in a multi-thread environment (for example: in a web container, if I already have many requests to process in parallel, adding an additional layer of parallelism inside each request could have more negative than positive effects)

In your example, the performance will anyway be driven by the synchronized access to System.out.println(), and making this process parallel will have no effect, or even a negative one.
Moreover, remember that parallel streams don't magically solve all the synchronization problems. If a shared resource is used by the predicates and functions used in the process, you'll have to make sure that everything is thread-safe. In particular, side effects are things you really have to worry about if you go parallel.
In any case, measure, don't guess! Only a measurement will tell you if the parallelism is worth it or not.

7. Collectors

Collectors are used to combine the result of processing on the elements of a stream.

Collection list into map

```
List<Product> productsList = new ArrayList<Product>();

    //Adding Products
    productsList.add(new Product(1,"HP Laptop",25000f));
    productsList.add(new Product(2,"Dell Laptop",30000f));
    productsList.add(new Product(3,"Lenevo Laptop",28000f));
    productsList.add(new Product(4,"Sony Laptop",28000f));
    productsList.add(new Product(5,"Apple Laptop",90000f));
    //Map<Object,String> productMap =
     productsList.stream().filter(p -> p.getPrice() >
20000f).collect(Collectors.toMap(p -> p.getId(), p -> p.getName())).forEach((k,v) ->
{System.out.print(k+" -- ");System.out.print(v+"  ");});

 Map<Integer,String> productMap = productsList.stream().filter(p -> p.getPrice() >
20000f).collect(Collectors.toMap(p -> p.getId(), p -> p.getName()));
            productMap.forEach((k,v) -> {System.out.print(k+" --
        ");System.out.print(v+"  ");});
```

8. Statistics()

Statistics() collectors are introduced to calculate all statistics when stream processing is being done.