Introduction to Java 8

- What's the need for Java 8?
- Processing huge datsets
- Functional Programming Style
- New features: Lambdas, streams, default methods

What's the need for Java 8?

Java 8 was released in March 2014, which is a major release and quite different from previous releases.

Code Readability:

Sorting Sopra Steria employees based on their salaries in Java 7:

```
Collections.sort(employeesListJava7, new Comparator<Employee>() {
            public int compare(Employee emp1, Employee emp2) {
                  return emp2.getSalary() - emp1.getSalary();
            }

      });
```

Sorting Sopra Steria employees based on their salaries Java 8:

```
employeesListJava8.sort(comparing(Employee::getSalary));
```

The code above can be read like sorting employeesList comparing there salaries.

Multicore CPUs:

CPUs have become multicore but most of our java programs use only one CPU core, leaving other core. Java 8 provides a new API(Streams) which supports many parallel operations to manage parallelism in ways which is less error prone.

The readability of these API is very good. You express what you want in readable (High level) manner and the implementation chooses the best low level mechanism to solve your problem. You don't need to write **synchronized** on your code **which is more error prone** on multicore CPUs.

Functional Style:

Feature of passing code (which may also return some data) to methods provides access to the technique referred as functional programming. We are treating methods like objects and passing them around and multicore CPU architecture favors such style.

We have terabytes of data which needs to processed in parallel (To which java hasn't been so friendly) unlike functional programming competitors (Scala).

Stream Processing:

Streams API in java.util.stream is based on the idea of unix command (Linked with pipes) like streams. Reading one by one from a stream and writing to an output stream. This output stream is used by another program/method as input stream and so on.

Java 8 can run your pipeline of stream operations on multiple CPU cores on the disjoint parts of the input. This is parallelism without you working hard to manage the threads in your application.

What price do you have to pay?

You can pass a behavior (code) to the stream methods. But this piece of behavior should be safe to execute concurrently on different pieces of the input. That is, writing the code which doesn't access shared mutable data to do its job. Such behavior is known as pure function, side-effect-free functions or stateless functions.

It's still possible to write the code that uses synchronized but using synchronized with multiple cores is far more expensive. Synchronize forces code to execute sequentially, which works against the goal of parallelism.

No shared mutable data and ability to pass methods/functions to other methods are basics building blocks of functional programming.

Method References and Lambdas(Anonymous functions)

We can pass methods with names, which are defined as parameters and call them method references and we can pass methods (functions) as values (in style of anonymous class objects for an interface) and call them lambdas or anonymous functions.

Example of a lambda: (int x)->x+1;
Example of a Method Reference: MyMathOperations::add1

Filtering an Employee in java 7 & Java 8:
Source available from java-8 directory on https://github.com/bpjoshi/java-8
Check out the code of com.bpjoshi.java8.introduction package for this.

Predicate Interface: Below is what functional interface Predicate looks like:

```
public interface Predicate<T>{
boolean test(T t);
}
```

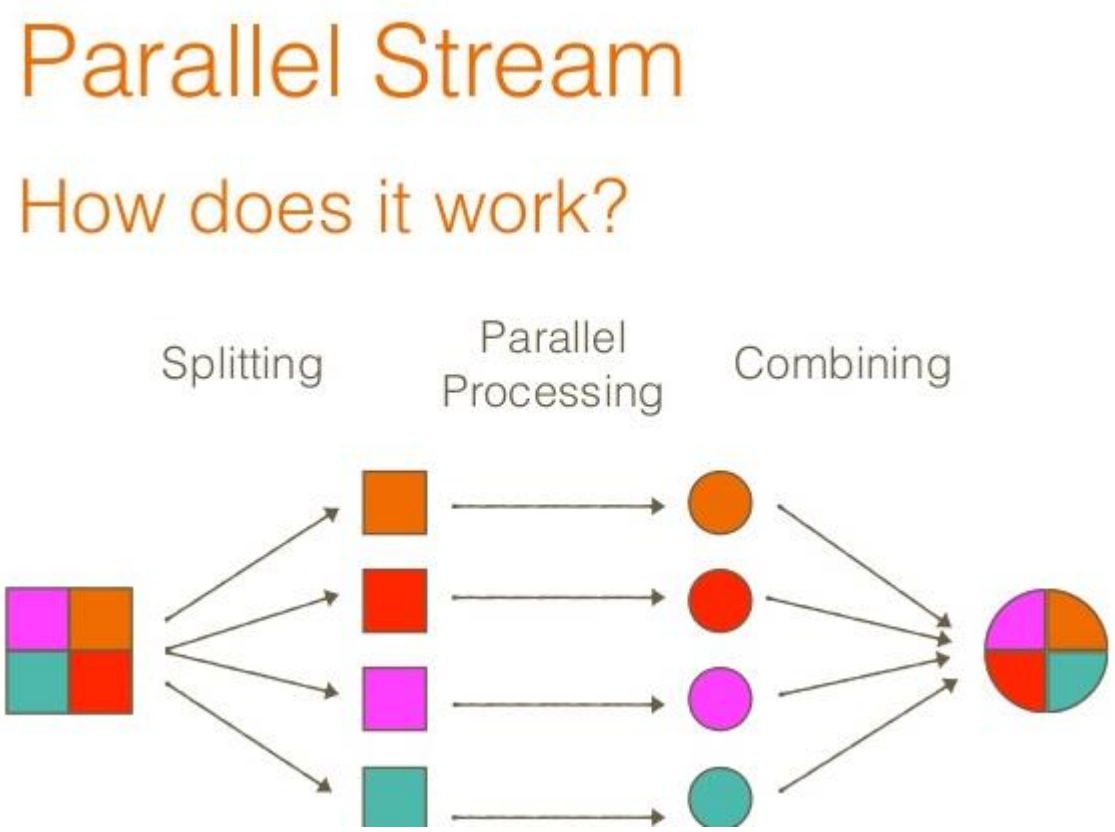
This interface is a part of java.util.function, a newly added API to java 8.

Streams Introduction

We use collection API heavily in our programs and we have to do lots of filtering from a collection of data to find the data that we need. In my project, we have a portal for maintenances of router and we have to email the customers to inform about the maintenances. We have to lots of filtering on the basis of date etc. It is quite a time taking process. An example to filter(technical certification(from a large list) to put in a to do list for an employee would look something like this.

```
1.  Map < Employee, List < Certificates >> certFicationTodo = new HashMap < > ();
2.          for (Certificate certi: certicates) {
3.              if (certi.isTechnical()) {
4.
5.              }
6.          }
```

Such filtering (when having lots of data) is done elegantly and efficiently(Parallel processing) using Java 8's streams. In streams, you don't even need to think about loops. The iteration is handled internally by this API and implementation provided by Java.



Divide a task into small tasks! 4 CPU cores are process the lists and one CPU is joining the list.

```
1.  //Use of stream
2.          List < Employee > richEmployees = employeesList.stream()
3.              .filter((Employee e) - > (e.getSalary() > 10)).collect(toList());
4.    //Use of parallel stream
5.          List < Employee > richEmployeesParallel = employeesList.parallelStream()
6.              .filter((Employee e) - > (e.getSalary() > 10)).collect(toList());
```

Default Methods

Write evolvable interfaces. We can now have methods inside interfaces. 1) Default methods and 2) Final methods

Other Changes in Java 8

Functional languages like Haskell avoid the null value. In java 8 there's an Optional<T> class, which can be used to avoid NullPointerException.

That ends this Basic Introduction to Java 8 and we will move on to specific details of java 8 features.(break);

Behavior Parameterization

- Behaviour Parameterization
- Anonymous Classes
- Lambda Introduction

Behavior Parameterization

It's a pattern that helps you to handle changing requirements easily. We make a method such that it's adaptable to changing requirements. Helps you create

Behavior parameterization is great because it allows you to separate logic of iterating over a collection to filter and the behavior to apply on each element of that collection.

Source available from java-8 directory on https://github.com/bpjoshi/java-8
Check out the code of com.bpjoshi.java8.parameterization package for this

Real World examples: 1) Comparator 2)Runnable

```
1.  //using Comparator
2.           List < Employee > empList = new ArrayList<>();
3.         //Add employees to this list and sort using anonymous comparator
4.         List<Employee> result= ListFilter.filterList(empList, anonymous comparator)
5.
```

Lambda Expressions

- What is Lambda
- Where and how to use Lambda
- Execute around pattern
- Functional Interfaces

What is a lambda?

Anonymous, Function(doesn't belong to class), Passed around and Concise

For example sort employees in decreasing order of their salaries:

```
1.  //using Comparator
2.          Comparator<Employee> bySalaryDesc
3.           =(Employee e1, Employee e2)->e2.getSalary()-e1.getSalary()
4.         //Add employees to this list and sort using anonymous comparator
```

Lambdas are different at bytecode generation, they are more efficient, used invokedynamic instruction that comes with JDK7.

Anonymous classes are processed by compiler as a new subtype for the given class or interface, so there will be generated a new class file for each.

For Lambdas this instruction is used to delay translate lambda expression in bytecode untill runtime. (instruction will be invoked for the first time only). They perform better in this way not loading needed at beginning.

Valid Lambdas

```
1.  //Valid Lambdas
2.          (String s)->s.length();
3.           (Employee e)->e.getSalary()>1000
4.           (int x, int y)->{
5.                Sysout(x+y);
6.               }
7.         ()->42
8.
```

Valid Lambdas Quiz

```
1.  //Valid Lambdas
2.          ()-{}
3.          ()->”Sopra Steia”;
4.          ()->{return “Sopra Steria”;}
            (String name)-> return “Hello ”+name;
5.          ()->{“Hello World”;}
6.          (List<Employee> list)->list.isEmpty();
7.          ()->new Employee();
8.          (int a, int b)->a*b;
```

Where to use lambda?

Lambda expressions are used in context of a functional interface. A functional interface is an interface with exactly one abstract method.

Function Interface

Java.util.function.Predicate<T>
Compator<T>
Runnable<T>

Default methods

```
default void print(){
      System.out.println("I am a vehicle!");
   }
```

@FunctionalInterface

Functional Descriptor

()->{} functional descriptor is ()->void
Predicate<T> T->void
Consumer<T> T->void
Function<T, R> T->R

Lambda Expression Type Checking

Valida Lambdas:

```
1. Public void execute(Runnable r){
        r.run();
   }
execute(()->{});

2. Predicate<Employee> p=(Employee e1)->a.getSalary();
```