

JAVA 8 - a revolutionary release of the development platform which brings some major tweaks and upgrades to the Java programming language -including enhanced JavaScript engine, new APIs for date and time manipulation, improved and faster JVM, and the lambda expressions, stream API, etc.



Many of you may be aware of these upgrades, but few who are wondering, what is it and how it works, we have a brief explanation with a simple example 😊 follows.

1. Lambda expressions – The tech giant's most anticipated upgrade for programming language.

What is lambda expression?

- To put in simple words, *expresses instances of functional interfaces, lambda expressions implement the only abstract function and therefore implement functional interfaces*

Syntax:

```
Lambda operator -> body
```

Now, we have another question, what is functional interface?

- *an interface that contains only one abstract method, they can have only one functionality to exhibit and it can have any number of default methods*

First we will see how to create functional interface.

Ex: *creating function interface*

```
@FunctionalInterface
public interface MyFuntionalInterface {

    void arithmeticOp(int operand);

    /**
     * Some default methods
     */
}
```

@FunctionalInterface annotation is used to ensure that the functional interface can't have more than one abstract method. In case more than one abstract methods are present, the compiler flags an 'Unexpected @FunctionalInterface annotation' message. However, it is not mandatory to use this annotation.

Now, will see how we can use this functional interface in our program.

```
.....
FunctionalInterface mFI_ADD_Func = (a) -> a + a;

System.out.println("Add the value : " + mFI_ADD_Func.arithmeticOp(4));

FunctionalInterface mFI_MUL_Func = (a) -> a * a;

System.out.println("Multiply the value : " + mFI_MUL_Func.arithmeticOp(4));

FunctionalInterface divideWithPrint_Func = (a) -> {
    System.out.println("Divide the value From lambda :");
    return a / a;
};
System.out.println(" --> " + divideWithPrint_Func.arithmeticOp(4));
.....
```

Ref : [GIT - lambda with on demand functions in java 8](#)

So, Lambda is nothing but function on demand which is not belongs to any class and can access variables declared outside its scope (not advisable).

Functionalities of Lambda

Exs : [GIT - lambda examples](#)

- ✚ Enable to treat functionality as a method argument, or code as data.
- ✚ A function that can be created without belonging to any class.
- ✚ A lambda expression can be passed around as if it was an object and executed on demand
- ✚ A lambda operator can have “**O**” - zero parameters, “**(a)**” - one parameter, “**(a, b, c)**” multiple parameters, Lambda expressions are just like functions and they accept parameters just like functions (parameters depends on abstract method in the functional interface designed)

Now, will see how the same interface in Java 7 works

```
.....
MyFunctionalInterface mFI_ADD_Func = new MyFunctionalInterface() {
    @Override
    public int arithmeticOp(int a) {
        return a + a;
    }
};

System.out.println("Java7 impl - Add the value : " + mFI_ADD_Func.arithmeticOp(4));

MyFunctionalInterface mFI_MUL_Func = new MyFunctionalInterface() {
    @Override
    public int arithmeticOp(int a) {
        return a * a;
    }
};

System.out.println("Java7 impl - Mul the value : " + mFI_MUL_Func.arithmeticOp(4));
.....
```

Ref : [GIT - On demand functions in java 7](#)

Yes, in Java7 too, we can have an on-demand functionality, but it cost us to create an Anonymous **InnerClass/ or new classes to implements**



interface on function, Java7, but stills does the works but on cost of bit ugly code design and it will compile with three more class files.

So, the Magic, **lambda expression**



Java8's cool Dude...! Allow us to write on demand functions with, beautiful code design.

Some of functional interfaces in Java - *Runnable, ActionListener, Comparable* and some java8 util.function package has *Predicate, BinaryOperator, and Function, etc.*

We can make functional interface as a generic type (above all the java's functional interface are generic type) and write our code with desired data types and various functions. (Ex: Function as name tells we can apply any functionality to it.)

Cool right?

Here it is, can we have hands on to implement lambda, which will be the best practice and will also have experience on functionality?



We can skip if you guys want to move on to next topic and can do the hands on later. But



Skip, skip, skip

Use case – have an Employee class with firstName, lastName, age and salary.

Conditions:

- ✚ Sort by first name
- ✚ Sort by last name first letter
- ✚ Sort by last name last letter
- ✚ Sort by age and,
- ✚ Sort by salary

It is easy with lambda,

Hints - use `Collections.sort()` for sorting with `Comparable/ comparator` user interface.

Ref : [Git - lambda exercise](#)

Java7

vs

Java8 (With lambda & other upgrades)



That's how the transformation design happened from java7 -> java8, still we have more API's & upgrade to know about.

2. The New Date/ Time API - *is introduced to cover the drawbacks of old date-time API.*

- ✚ **Not Thread safe** – Developer must deal with concurrency (In Java8 it is immutable).
- ✚ **Poor design** - Default Date starts from 1900.
- ✚ **Difficult in time zone handling** – new API has domain-specific design in mind.

Java 8 introduces a new date-time API under the package `java.time`. Following are some of the important classes introduced in `java.time` package

- ✚ **Local** – Simplified date-time API with no complexity of timezone handling.
- ✚ **Zoned** – Specialized date-time API to deal with various timezones.



Simple example to get local time and date, UTC.

LocalDate/LocalTime and LocalDateTime classes simplify the development where timezones are not required

```
LocalDateTime currentTime = LocalDateTime.now();
System.out.println("Current DateTime: " + currentTime);
System.out.println("Current Date: " + currentTime.toLocalDate());
System.out.println("Current Time: " + currentTime.toLocalTime());

Instant dateTime = Instant.now();
System.out.println("UTC DateTime: " + dateTime);
```

Ref : [Git - Local Date Time](#)

Java8, have more fine-grained control over our date and time representation

Simple example to handle zoned time.

Ref: [Zone List](#) Java

```
Instant dateTime = Instant.now();

System.out.println("Current Europe/London time: "
    +dateTime.atZone(ZoneId.of("Europe/London")));

System.out.println("Current Atlantic/Azores time: "
    +dateTime.atZone(ZoneId.of("Atlantic/Azores")));

System.out.println("Current Africa/Lagos time: " +dateTime.atZone(ZoneId.of("Africa/Lagos")));

ZonedDateTime zonedDateTime = ZonedDateTime.parse("2007-12-03T10:15:30+05:30[Asia/Karachi]");

System.out.println("Zoned Date : " + zonedDateTime);
```

Ref: [Git - Zoned Date Time](#)

Using java8 handling time zone becomes easy.

Some Special Mentions (Date and Time API features)

Ref: [Git - Date Time Exercise](#)

✚ *Formatting - Using the DateTimeFormat*

✚ *Difference in Time: Duration() and Period*

🕒 *Period – It deals with date based amount of time. (Zoom In)*

🕒 *Duration – It deals with time based amount of time.*

✚ *Chrono Units Enum - enum is added in Java 8 to replace the integer values used in old API*

✚ *Temporal Adjusters - used to perform the date mathematics*

✚ *Backward Compatibility - A toInstant() method is added to the original Date and Calendar objects, which can be used to convert them to the new Date-Time API. Use an ofInstant(Instant,ZoneId) method to get a LocalDateTime or ZonedDateTime object*

No hands On - 😞

// but no stopping 😊



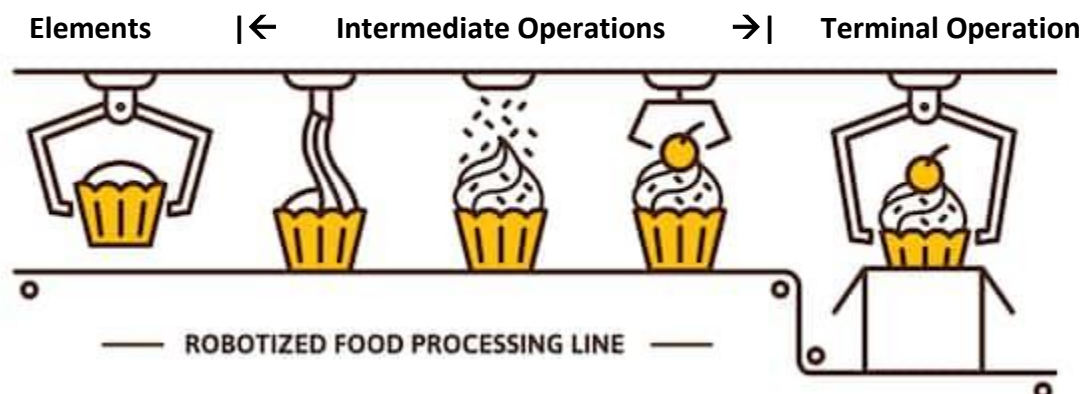
3. The Stream API –



- ✚ supports functional-style operations on streams of elements, such as map-reduce transformations on collections.
- ✚ stream operations are divided into intermediate and terminal operations.
- ✚ stream pipeline consists of a stream source, followed by zero or more intermediate operations, and a terminal operation

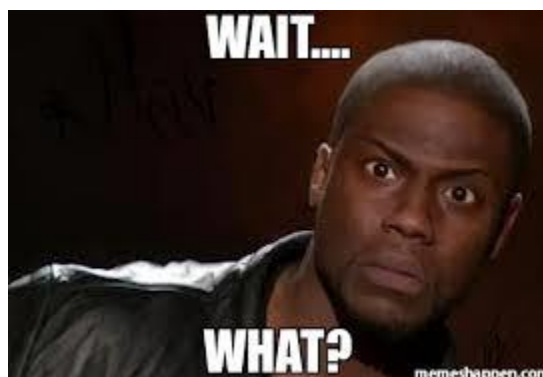
How stream works?

Take below example



As above image represents, we are going to do the similar kind of process in our stream, the data flow depends on the operations we choose. (stream the list of elements -> apply the operations on the elements -> pack it).

Yes, it is that simple, if we understood, what Operations (ingredients/ design) we must apply and how we expected it (End product).



How to create stream?

There are many ways to create stream on object, array or list, we see few in following example

```
Space[] arrayOfSpaceObject = { new Space(1, "comet", 170), // temp in degree
                                new Space(2, "Meteorite", 3000),
                                new Space(3, "dwarf planet", -223),
                                new Space(4, "kuiper belt", -223),
                                new Space(5, "oort cloud", 4) };

// Streaming on Arrays using Stream of
List<Space> sorted = Stream.of(arrayOfSpaceObject)
    .sorted((obj1, obj2) -> (int) (obj1.getTemperature() -obj2.getTemperature()))
    .collect(Collectors.toList());

System.out.println("Sorted List :" + sorted);

// Streaming on List
List<Space> sortedOnId = sorted.stream().sorted(Comparator.comparing(Space::getId))
    .collect(Collectors.toList());

System.out.println("Sorted List by Id :" + sortedOnId);

// Streaming on single object using stream of
Space ob = arrayOfSpaceObject[0];
System.out.println("Object modified :" + ob);

ob = Stream.of(ob).peek((object) -> object.setTemperature(object.getTemperature() * 10)
    .findFirst().orElse(null);

System.out.println("Object modified :" + ob);
```

Ref : [Git - few ways to create stream](#)

We can also use Stream builder to create stream.

We should do hands on for each functions/ topic, as stream is vast, we may want to experience hands on to understand clearly.



Stream Operations –

-> *forEach* *

✚ *forEach* - is simplest and most common operation; it loops over the stream elements, calling the supplied function on each element.

✚ The method is so common that it has been introduced directly in *Iterable*, *Map* etc:

Simple example for *forEach* on stream and collection.

```
List<Space> elementList = Arrays.asList(new Space(1, "comet", 170), // temp in
degree
                                     new Space(2, "Meteorite", 3000),
                                     new Space(3, "dwarf planet", -223),
                                     new Space(4, "kuiper belt", -223),
                                     new Space(5, "oort cloud", 4));

elementList.stream().forEach((element) -> element.setId(element.getId() * 2));

elementList.forEach((element) -> element.setId(element.getId() * 2));
```

Ref: [Git - forEach](#)

forEach() is a **terminal operation**, which means that, after the operation is performed, the stream pipeline is considered consumed, and can no longer be used.

-> *map* *

produces a new stream after applying a function to each element of the original stream. The new stream could be of different type

```
List<Space> elementList = Arrays.asList(new Space(1, "comet", 170), // temp in
degree
                                     new Space(2, "Meteorite", 3000),
                                     new Space(3, "dwarf planet", -223),
                                     new Space(4, "kuiper belt", -223),
                                     new Space(5, "oort cloud", 4));

List<String> nameList =
elementList.stream().map(Space::getName).collect(Collectors.toList());
```

Ref: [Git - map](#)

As we can see in the above example, the **stream of Space** has been converted to **stream of String** after the *map* operation.

-> flatMap *

A stream can hold complex data structures like `Stream<List<String>>`. In cases like this, `flatMap()` helps us to flatten the data structure to simplify further operations:

```
List<List<Space>> listOfList = Arrays.asList(elementList, errorListTemp,
errorListName);

List<Space> flattenedList =
listOfList.stream().flatMap(Collection::stream).collect(Collectors.toList());
```

Ref: [Git - flatMap](#)

Notice how we were able to convert the `Stream<List<String>>` to a simpler `Stream<String>` – using the `flatMap()` API.



-> filter *

this produces a new stream that contains elements of the original stream that pass a given condition (specified by a Predicate – Functional interface).

```
List<Space> elementList = Arrays.asList(new Space(1, "comet", 170), // temp in
degree
                                     new Space(2, "Meteorite", 3000),
                                     new Space(3, "dwarf planet", -223),
                                     new Space(4, "kuiper belt", -223),
                                     new Space(5, "oort cloud", 4));

List<Space> filteredElementList = elementList.stream().filter((space) ->
space.getTemperature() > 1).collect(Collectors.toList());
```

Ref: [Git - filter](#)

Example shows that the list has been filtered by the space object, on condition which has greater than temperature one, and returns the new stream of list.

-> peek *

it performs the specified operation on each element of the stream and returns a new stream which can be used further. `peek()` is an intermediate operation:

```
List<Space> elementList = Arrays.asList(new Space(1, "comet", 170), // temp in
                                         degree
                                         new Space(2, "Meteorite", 3000),
                                         new Space(3, "dwarf planet", -223),
                                         new Space(4, "kuiper belt", -223),
                                         new Space(5, "oort cloud", 4));

List<Space> newList =
    elementList.stream().peek((space) -> space.setTemperature(space.getTemperature()
    * 21)).collect(Collectors.toList());
```

Ref: [Git - peek](#)

Here `peek` is used to increase the object temperature by double. Finally, `collect()` is used as the terminal operation.

-> collect *

We saw how `collect()` works in the previous example; its one of the common ways to get stuff out of the stream once we are done with all the processing

```
List<Space> elementList = Arrays.asList(new Space(1, "comet", 170), // temp in
                                         degree
                                         new Space(2, "Meteorite", 3000),
                                         new Space(3, "dwarf planet", -223),
                                         new Space(4, "kuiper belt", -223),
                                         new Space(5, "oort cloud", 4));

List<Space> filteredElementList = elementList.stream().filter((space) ->
    space.getTemperature() > 1).collect(Collectors.toList());
```

`collect()` performs mutable fold operations (repackaging elements to some data structures and applying some additional logic, concatenating them, etc.) on data elements held in the Stream instance.

The strategy for this operation is provided via the Collector interface implementation. In the example above, we used the `toList` collector to collect all Stream elements into a List instance.

-> **findFirst** *

findFirst() returns an Optional for the first entry in the stream; the Optional can, of course, be empty:

```
List<Space> elementList = Arrays.asList(new Space(1, "comet", 170), // temp in
degree
    new Space(2, "Meteorite", 3000),
    new Space(3, "dwarf planet", -223),
    new Space(4, "kuiper belt", -223),
    new Space(5, "oort cloud", 4));

Optional<Space> optionalSpace = elementList.stream().filter((space) ->
space.getTemperature() > 1000).peek((name) -> System.out.println("name : " +
name)).findFirst();

Space spaceObj = elementList.stream().filter((space) -> space.getTemperature() <
50).peek((name) -> System.out.println("name : " +
name)).findFirst().orElse(null);
```

Ref : [Git - findFirst](#)

We have two way to use findFirst(), i.e., we can get it as Optional<> object or just an object by orElse() in findFirst, it will not stream every object on the list, it will stop the stream once it finds the first element, which pass through the filter.



I think we can wind up on stream as we have seen important operations on stream, though we have more operations on stream like,

- ✚ toArray
- ✚ iterate, skip, limit, generate
- ✚ count, sorted
- ✚ min, max, distinct
- ✚ allMatch, anyMatch, noneMatch
- ✚ reduce
- ✚ Advanced collect
 - 🤖 Joining
 - 🤖 toList
 - 🤖 toSet
 - 🤖 toCollection
 - 🤖 partitioningBy
 - 🤖 groupingBy
 - 🤖 mapping
 - 🤖 reducing,



Yeah...!!!

-> parallel stream *

we can perform stream operations in parallel without having to write any boilerplate code, would get executed in parallel on multiple elements of the stream, by simply adding the parallel() syntax

This functionality can, of course, be tuned and configured further, if you need more control over the performance characteristics of the operation

As is the case with writing multi-threaded code, we need to be aware of few things while using parallel streams:

- ✚ We need to ensure that the code is thread-safe. Special care needs to be taken if the operations performed in parallel modifies shared data.
- ✚ We should not use parallel streams if the order in which operations are performed or the order returned in the output stream matters. For example operations like `findFirst()` may generate the different result in case of parallel streams.
- ✚ Also, we should ensure that it is worth making the code execute in parallel. Understanding the performance characteristics of the operation in particular

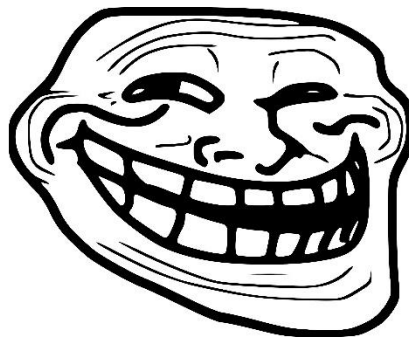
Ref : [Git -parallel stream](#)



Anything else?

NOPE

That's it from me on Stream API



JavaScript on Nashorn –

The Nashorn JavaScript Engine is part of Java SE 8 and competes with other standalone engines like Google V8 (the engine that powers Google Chrome and Node.js). Nashorn extends Java capabilities by running dynamic JavaScript code natively on the JVM.



*a much improved javascript engine is introduced, to replace the existing **Rhino in JAVA7**. Nashorn provides 2 to 10 times better performance, as it directly compiles the code in memory and passes the bytecode to JVM. Nashorn uses invoke dynamics feature, introduced in Java 7 to improve performance.*

jjs

For Nashorn engine, JAVA 8 introduces a new command line tool, jjs, to execute javascript codes at console.

-> Interpreting js File

Create and save the file **hello_world.js**

```
print('Hello World!');
```

Now open console and use the following command.

```
C:\java_script_project>jjs hello_world.js
```

It will produce the following output:

```
Hello World!
```



-> *jjs in Interactive Mode*

Open the console and use the following command.

```
C:\ java_script_project >jjs
jjs> print("Hello, World!")
Hello, World!
jjs> quit()
>>
```

-> *Calling JavaScript from Java*

Using `ScriptEngineManager`, JavaScript code can be called and interpreted in Java.

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class RunJavaScriptFromJAVAMain {

    public static void main(String args[]) {

        ScriptEngineManager scriptEngineManager = new ScriptEngineManager();
        ScriptEngine nashorn = scriptEngineManager.getEngineByName("nashorn");

        String name = "Nashorn awesome :)";
        Integer result = null;

        try {
            nashorn.eval("print('\" + name + \"')");
            result = (Integer) nashorn.eval("10 * 10");
            nashorn.eval("print('\" + result + \" %')");
        } catch (ScriptException e) {
            System.out.println("Error executing script: " + e.getMessage());
        }
    }
}
```

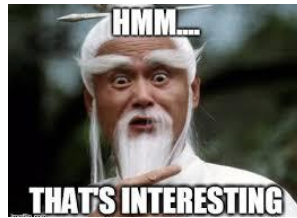
Ref: [Git -run js from java](#)

-> Calling Java from JavaScript

Following example explains how to import and use Java classes in java script

[illegible]

Ref: [Git -use java code in js](#)



Now relax yourself find the word by connecting the below pictures:

Hint - festival



MA030N3DN393DN :rowm4

Memory Management for JVM –

Java memory management is an ongoing challenge and a skill that must be mastered to have properly tuned applications that function in a scalable manner. Fundamentally, it is the process of allocating new objects and properly removing unused objects.

-> JVM is an abstract computing machine that enables a computer to run a Java program.

There are three notions of JVM:

- ✚ **Specification** - where working of JVM is specified
- ✚ **Implementation** - known as (JRE) Java Runtime Environment
- ✚ **Instance** - an instance of JVM is created

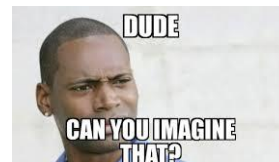
-> **Java (JVM) Memory Structure**

JVM memory is divided into multiple parts: Heap Memory, Non-Heap Memory, and Other.



-> **Heap Memory**

Heap memory is the run time data area from which the memory for all java class instances and arrays is allocated. The heap is created when the JVM starts up and may increase or decrease in size while the application runs. The size of the heap can be specified using `-Xms` VM option. The heap can be of fixed size or variable size depending on the garbage collection strategy. Maximum heap size can be set using `-Xmx` option. By default, the maximum heap size is set to 64 MB.



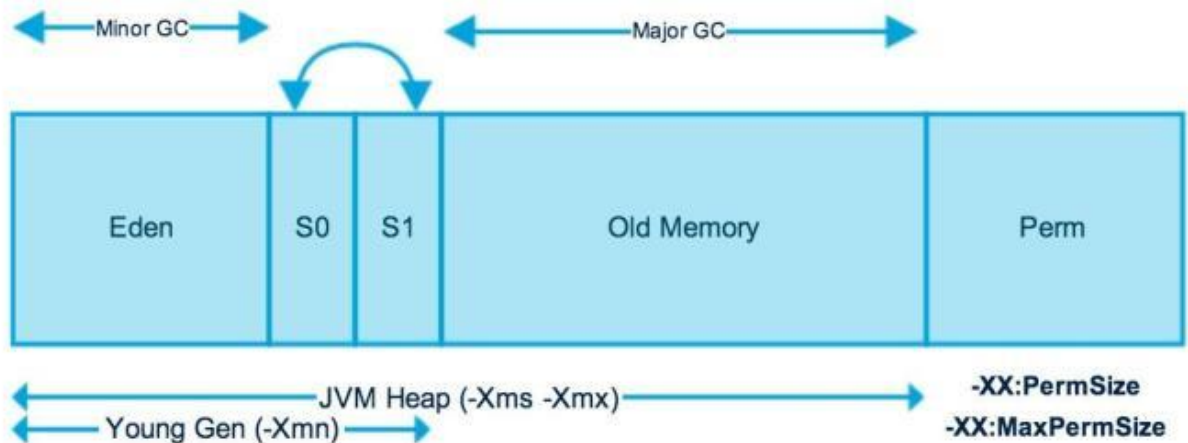
-> **Non-Heap Memory**

The JVM has memory other than the heap, referred to as Non-Heap Memory. It is created at the JVM startup and stores per-class structures such as runtime constant pool, field and method data, and the code for methods and constructors, as well as interned Strings. The default maximum size of non-heap memory is 64 MB. This can be changed using `-XX:MaxPermSize` VM option.

-> Other Memory

JVM uses this space to store the JVM code itself, JVM internal structures, loaded profiler agent code, and data, etc.

Java (JVM) Heap Memory Structure



Permanent Generation (Replaced by Metaspace since Java 8)

Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application. Perm Gen is populated by JVM at runtime based on the classes used by the application. Perm Gen also contains Java SE library classes and methods. Perm Gen objects are garbage collected in a full garbage collection.


Metaspace


With Java 8, there is no Perm Gen, which means there is no more “java.lang.OutOfMemoryError: PermGen” space problems. Unlike Perm Gen which resides in the Java heap, Metaspace is not part of the heap. Most allocations of the class metadata are now allocated out of native memory. Metaspace by default auto increases its size (up to what the underlying OS provides), while Perm Gen always has fixed maximum size. Two new flags can be used to set the size of the metaspace, they are: “-XX:MetaspaceSize” and “-XX:MaxMetaspaceSize”. The theme behind the Metaspace is that the lifetime of classes and their metadata matches the lifetime of the classloaders. That is, as long as the classloader is alive, the metadata remains alive in the Metaspace and can’t be freed.




Other memory models

Code Cache

 **Method Area** - is part of space in the Perm Gen and used to store class structure (runtime constants and static variables) and code for methods and constructors.

 **Memory Pool** - Memory Pools are created by JVM memory managers to create a pool of immutable objects.

 **Runtime Constant Pool** - is a per-class runtime representation of a constant pool in a class..

 **Java Stack Memory** - is used for the execution of a thread. They contain method-specific values that are short-lived and references to other objects in the heap that is getting referred from the method.

Java Heap Memory Switches-

-Xms

- For setting the initial heap size when JVM starts

-Xmx

- For setting the maximum heap size

-Xmn

- For setting the size of young generation, rest of the space goes for old generation

-XX:PermGen

- For setting the initial size of the Permanent Generation Memory

-XX:MaxPermGen

- For setting the maximum size of Perm Gen

-XX:SurvivorRatio

- For providing ratio of Eden space, for example, if young generation size is 10m and VM switch is `-XX:SurvivorRatio=2` then 5m will be reserved for Eden space and 2.5m each for both the Survivor spaces. The default value is 8

-XX:NewRatio

- For providing ratio of old/new generation sizes. The default value is 2

Garbage Collection

Mark and Sweep Model of Garbage Collection

🚩 **Java Garbage Collection Types**

- ☹️ *Serial GC (-XX:+UseSerialGC)*
- ☹️ *Parallel GC (-XX:+UseParallelGC)*
- ☹️ *Parallel Old GC (-XX:+UseParallelOldGC)*
- ☹️ *Concurrent Mark Sweep (CMS) Collector (-XX:+UseConcMarkSweepGC)*
- ☹️ *G1 Garbage Collector (-XX:+UseG1GC)*



References :

<https://www.google.com/> → 🙄

<https://stackify.com/streams-guide-java-8/> - for stream

<https://betsol.com/2017/06/java-memory-management-for-java-virtual-machine-jvm> - for JVM

I think every one has been mastered in java 8 😏 by NOW

