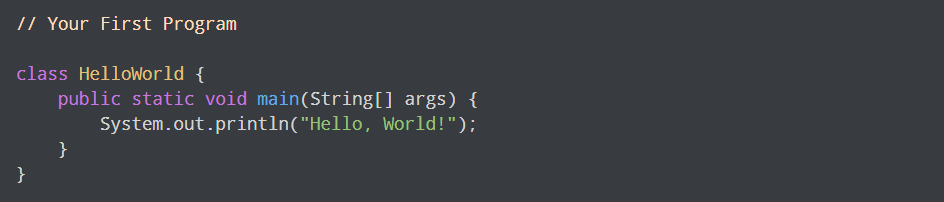
JAVA TURTORIAL

# 1. Introduction

## 1.1. Structure in Java

To explain all components contain in a java application, “Hello world” is one of good example to show:

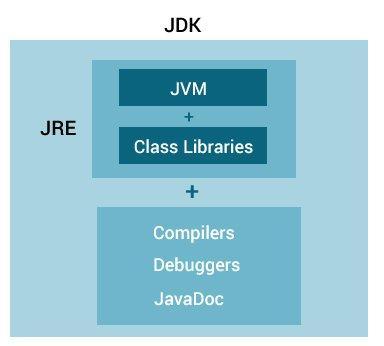
****

* **Comment (**//**):** Comments are intended for users reading the code to understand the intent and functionality of the program. It is completely ignored by the Java compiler**.**
* **Class (HelloWord):** In Java, every application begins with a class definition,the name of the class should match the filename in Java.
* **Main method:** Every application in Java must contain the main method. The Java compiler starts executing the code from the main method.
* **Print output:** print out the result to standard output (your screen)

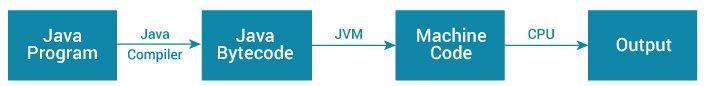
! Note:

* Every valid Java Application must have a class definition (that matches the filename).
* The main method must be inside the class definition.
* The compiler executes the codes starting from the main function.

## 1.2. Relationship between JVM, JRE & JDK



**Figure 1. Relationship between JVM, JRE & JDK**



**Figure 2. Working flow of Java Program**

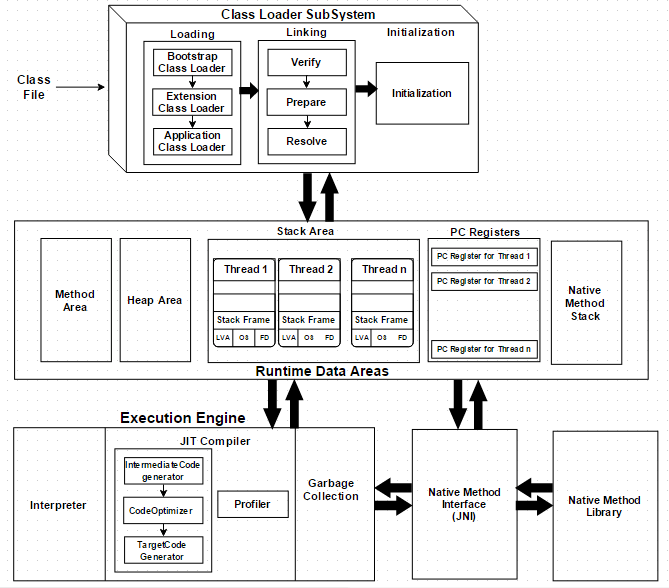
State of working:

1. Java compiler first compiles your Java code to bytecode.
2. The JVM translates bytecode into native machine code.

## 1.3. JVM

To understand those components, we need to know deeply about JVM which allows us to write code more efficiently.

* JVM (Java Virtual Machine) is an abstract machine that enables your computer to run a Java program.
* A Virtual Machine is a software implementation of a physical machine. Java was developed with the concept of WORA (Write Once Run Anywhere), which runs on a VM.
* The **compiler** compiles the Java file into a Java **.class file**, then that **.class file** is input into the JVM, which loads and executes the class file.



**Figure 3. JVM architecture**

From Figure 3, there are 3 main subsystems in JVM architecture:

1. Class Loader Subsystem
2. Runtime Data Areas
3. Execution Engine

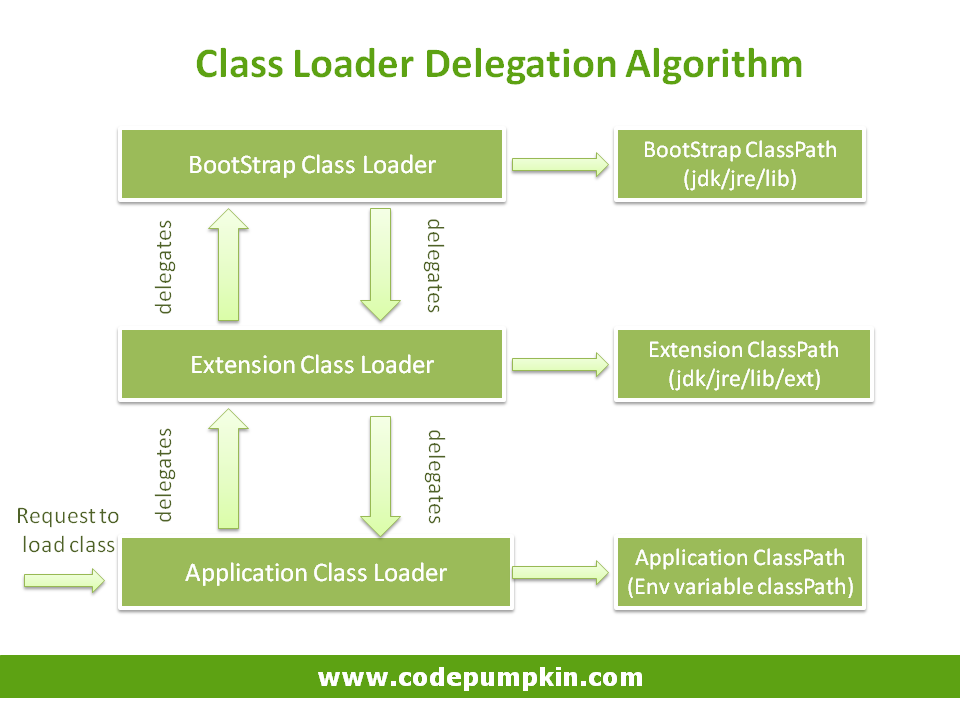
### 1.3.1. Class Loader Subsystem

Java's dynamic class loading functionality is handled by the ClassLoader subsystem. It **loads, links. and initializes the class file** when it refers to a class for the first time at runtime, not compile time.

- Loading: Classes will be loaded by this component. BootStrap ClassLoader, Extension ClassLoader, and Application ClassLoader are the three ClassLoaders that will help in achieving it.

1. **Boostrap Classloader**: Responsible for loading classes from the bootstrap classpath, nothing but runtime.jar(all the standard lib like java.lang.\*,…).Highest priority will be given to this loader.
2. **Extension ClassLoader**: Responsible for loading extension classes which are included inside the ext folder (jre\lib) like OJDBC, MQ lib,…
3. **Application ClassLoader**: Responsible for loading local application level classpath, path mentioned Environment Variable, etc.

🡺 The above ClassLoaders will follow Delegation Hierarchy Algorithm like in Figure 4 while loading the class files.



**Figure 4. Delegation Hierarchy Algorithm**

The order of its hierarchy is Application ClassLoader 🡪 Extension ClassLoader 🡪 Bootstrap ClassLoader. The Bootstrap ClassLoader is always given the higher priority, next is Extension ClassLoader and then Application ClassLoader.

- Linking:

1. **Verify**: Once the class files are loaded to the memory, there is a verify phase where the bytecode class file are verified if the class files conform to standards**.** If verification fails we will get the verification error.
2. **Prepare**: For all static variables memory will be allocated and assigned with default values**.**
3. **Resolve**: All symbolic memory references are replaced with the original references.

- Initialization: all static variables are assigned with the original values, and also executes the static block at this point.

### 1.3.2. Runtime Data Area

JVM need runtime data area to **store the class files and execute them**. The Runtime Data Area is divided into **five major components**:

- **Method Area:** All the class-level data (kind of global varibles in class)will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.

- **Heap Area:** All the Objects and their corresponding instance variables will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.

- **Stack Area:** For each thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory. The stack area is thread-safe since it is not a shared resource. The Stack Frame is divided into three subentities:

* **Local Variable Array**: Related to the method how many local variables are involved and the corresponding values will be stored here.
* **Operand stack**: If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
* **Frame data**: In the case of any exception, the catch block information will be maintained in the frame data.

- **PC Registers:** Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.

- **Native Method stacks:** Native Method Stack holds native method information related to native platforms. For every thread, a separate native method stack will be created. ⇨ For example, if we're running the JVM on Windows, it will contain Windows-related information. Likewise, if we're running on Linux, it will have all the Linux-related information we need.

### 1.3.3. Execution Engine

The bytecode, which is assigned to the Runtime Data Area, will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece. It seperated to 4 part:

- **Interpreter**: The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.

- **JIT(just in time) compiler**: The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.

* **Intermediate Code Generator**: Produces intermediate code.
* **Code Optimizer:** Responsible for optimizing the intermediate code generated above
* **Target Code Generator**: Responsible for Generating Native Machine Code.
* **Profiler**: A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.

**Working process of JIT:** Methods are not compiled when they are called the first time. For each and every method, the JVM maintains a call count, which is incremented every time the method is called. The methods are interpreted by the JVM until the call count exceeds the JIT compilation threshold (the JIT compilation threshold improves performance and helps the JVM to start quickly. The threshold has been selected carefully by Java developers for optimal performance. The balance between startup times and long-term performance is maintained).

Therefore, very frequently used methods are compiled as soon as the JVM has started, and less frequently used methods are compiled later.

After a method is compiled by JIT, its call count is reset to zero, and subsequent calls to the method increment its call count. When the call count of a method reaches a JIT recompilation threshold, the JIT compiler compiles method a second time, applying more optimizations as compared to optimizations applied in the previous compilation. This process is repeated until the maximum optimization level is reached. The most frequently used methods are always optimized to maximize the performance benefits of using the JIT compiler.

- **Garbage Collector**: Collects and removes unreferenced objects from heap to reclaim heap space. Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

- **Java Native Interface (JNI):** JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

If we are running the JVM (a Java application) on Windows, then the native method interface (Windows method interface) will connect the JVM with the Window method libraries (native method libraries) for executing Windows methods (native methods).

You may write your application purely in Java, but there are certain situations where Java code alone might not meet your requirements. Programmers use the JNI to write the Java native methods when an application cannot be written purely in Java.

⇨ For example: When classes in java library doesn’t support the platform-dependent features needed by the application. When you want your java code to access the library already written in any other language using JNI.

### 1.3.4. Conclusion of JVM

The most important JVM Components related to performance are:

* Heap
* JIT (Just in time) Compiler
* Garbage collection (compiles bytecode to machine code at runtime and improves the performance of Java applications.)

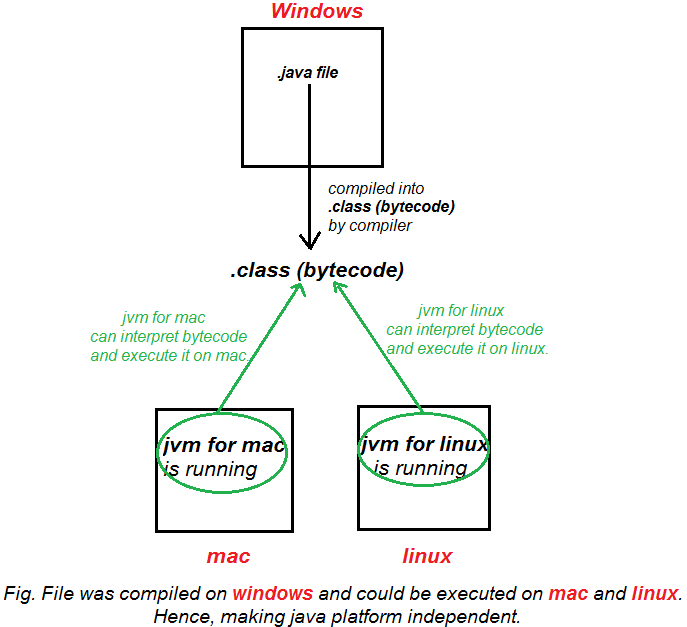
There are many JVM to which option we can config:

* Increasing and decreasing the heap size for managing object for best performance.
* Selecting different garbage collectors, depending on your requirement.

To know how to config those 3 JVM components, see at: <https://docs.oracle.com/cd/E12839_01/web.1111/e13814/jvm_tuning.htm#PERFM153>

*Question*: *Why Java is called “Wore” (write one run everywhere) ?*

Once source code (.java file) is compiled on one platform (bytecode is formed .class file), that bytecode can be executed (interpreted) on any other platform running a JVM like Figure 5.



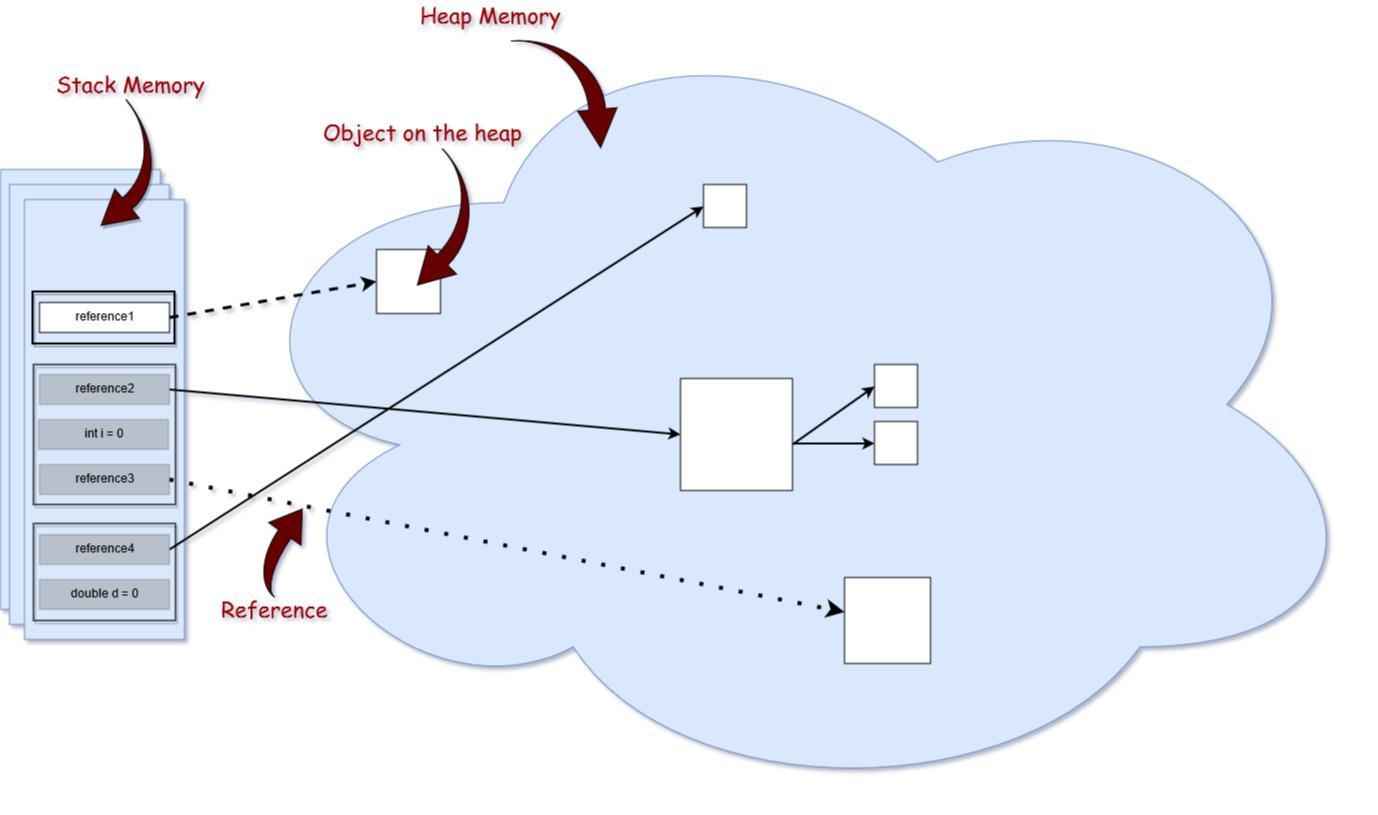
**Figure 5. How Java process a source code.**

## 1.4. Memory management in Java

Unlike C++, Java has automatic memory management, a nice and quiet garbage collector that works in the background to clean up the unused objects and free up some memory. Therefore, we do not need to bother ourself with problems like destroying objects, as they are not used anymore. However, even if this process is automatic in Java, it does not guarantee that all objects have been deleted by the Garbage collector.

By not knowing how the garbage collector and Java memory is designed, you could have objects that are not eligible for garbage collecting, even if you are no longer using them.

⇨ So knowing how memory actually works in Java is important, as it gives you the advantage of writing high-performance and optimized applications that will never ever crash with an **OutOfMemoryError**. On the other hand, when you find yourself in a bad situation, you will be able to quickly find the memory leak.



**Figure 6. Memory structure (Heap vs Stack)**

Firstly, we need to focus on 2 part of run-time memory: the stack and the heap (relationship between stack and heap when objects are create in Figure 6). Please keep in mind that the size of memory types in this picture are not proportional to the memory size in reality. The heap memory is larger than the stack memory.

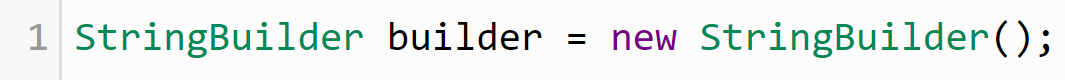
### 1.4.1. Stack memory

Stack memory is responsible for holding references to heap objects and for storing value types (also known in Java as primitive types).

In addition, variables on the stack have a certain visibility, also called scope. Only objects from the active scope are used. For example, assuming that we do not have any global scope variables (fields), and only local variables, if the compiler executes a method’s body, it can access only objects from the stack that are within the method’s body. It cannot access other local variables, as those are out of scope. Once the method completes and returns, the top of the stack pops out, and the active scope changes.

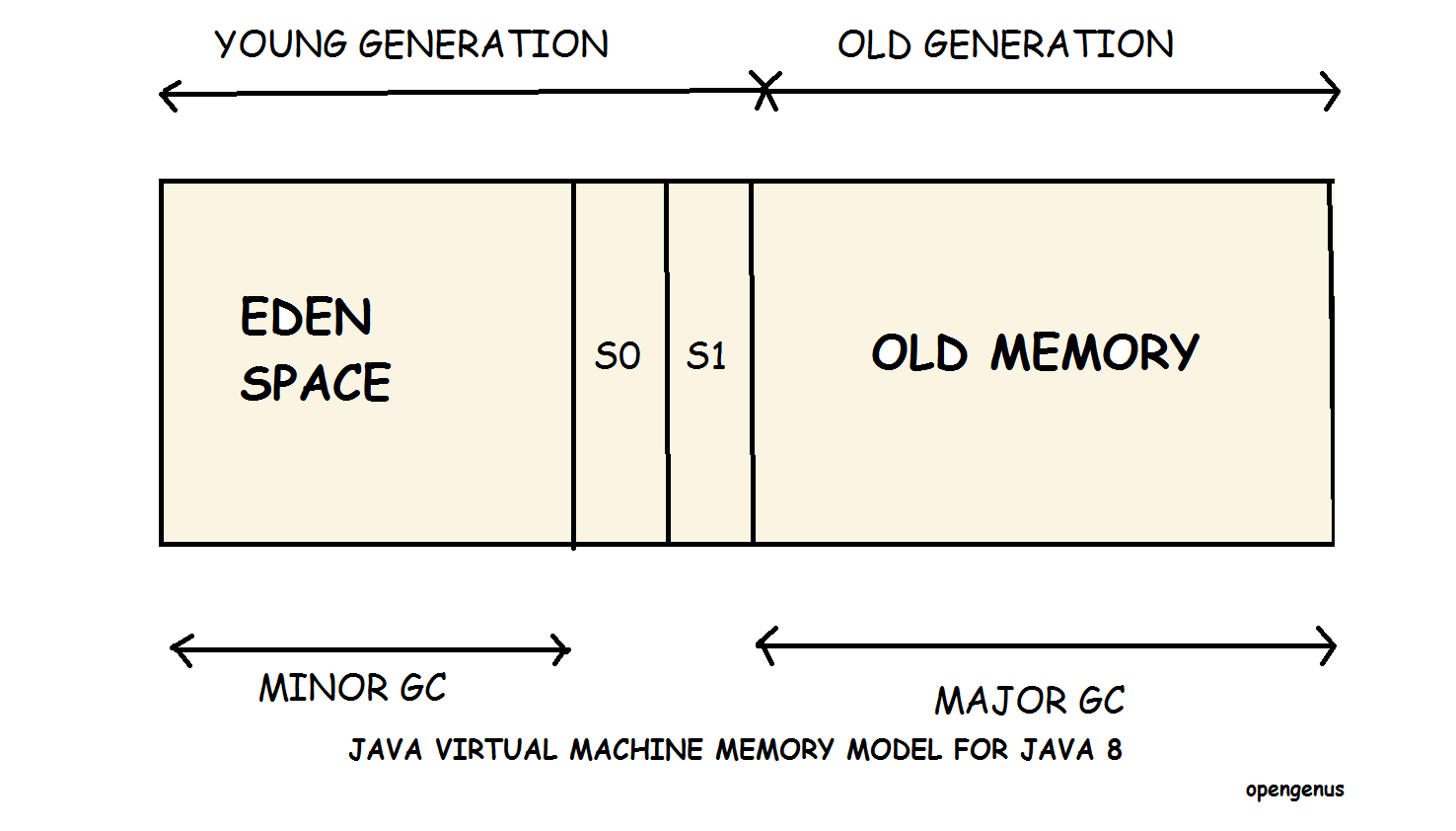
### 1.4.2. Heap memory

This part of memory stores the actual object in memory. Those are referenced by the variables from the stack.



“The new keyword is responsible for ensuring that there is enough free space on heap, creating an object of the StringBuilder type in memory and referring to it via the “builder” reference, which goes on the stack.”

There exists only one heap memory for each running JVM process. Therefore, this is a shared part of memory regardless of how many threads are running. Actually, the heap structure is a bit different than it is shown in the picture above. The heap itself is divided into a few parts in Figure 7 (young generation, old generation), which facilitates the process of garbage collection.



**Figure 7. Heap memory in Java 8**

The maximum stack and the heap sizes are not predefined — this depends on the running machine.

#### 1.4.2.1. Young generation memory space

Young generation is the memory space where all the new objects are created. When young generation is filled, a garbage collection is performed. Garbage collection in the Young Generation memory space is called Minor Garbage Collection. Young Generation is divided into three parts namely Eden Memory space and two Survivor Memory spaces.

Keys points about Young Generation memory spaces:

* Most of the newly created objects are located in the Eden memory space.
* When Eden space is filled with objects, minor garbage collection is performed and all the survivor objects are moved to one of the survivor spaces.
* Minor garbage collection checks the survivor objects and move them to the other survivor space. Thus, one of the survivor space is always empty.
* Objects that are survived after many cycles of garbage collection, are moved to the Old generation memory space.

#### 1.4.2.2. Old generation memory space

Old Generation memory contains the objects that are long lived and survived after many rounds of Minor garbage collection. Usually garbage collection is performed in Old Generation memory when it is full.

Garbage Collection in the old generation memory space is called Major garbage collection and usually takes much longer than Minor garbage collection.

Keys points about Old Generation memory spaces:

* All long lived objects or objects that survived several minor garbage collections are located in the Old Generation memory space
* When Old generation memory space is filled with objects, major garbage collection is performed
* Number of Major garbage collections less than number of Minor garbage collections
* Time taken by Major Garbage collection more than time taken by Minor Garbage collection
* Major garbage collections take a long time and makes applications unresponsive

### 1.4.3. Reference types

In Figure 6, we can see that the arrows representing the references to the objects from the heap are actually of different types. That is because, in the Java programming language, we have 4 different types of references: strong, weak, soft, and phantom references.

The difference between the types of references is that the objects on the heap they refer to are eligible for garbage collecting under the different criteria.

* **Strong Reference:** These are the most popular reference types that we all are used to like the example StringBuilder above. The object on the heap which is not garbage collected while there is a strong reference pointing to it.
* **Weak Reference:** weak reference is most likely not survive after the next garbage collection process. A weak reference is created as follows:

****

* **Soft Reference:** These types of references are used for more memory-sensitive scenarios, since those are going to be garbage collected only when your application is running low on memory. Java guarantees that all soft referenced objects are cleaned up before it throws an **OutOfMemoryError.** A soft reference is created as follows:

****

* **Phantom reference:** Used to schedule post-mortem cleanup actions, since we know for sure that objects are no longer alive. Used only with a reference queue, since the .get() method of such references will always return null. These types of references are considered preferable to finalizers.

### 1.4.4. Garbage collection process

Depending on the type of reference that a variable from the stack holds to an object from the heap, at a certain point in time, that object becomes eligible for the garbage collector.

In Figure 6, all objects that are in red are eligible to be collected by the garbage collector which haven’t any reference or have WeakReference assigning to. And when it cannot be accessed anymore, it is garbage as well.

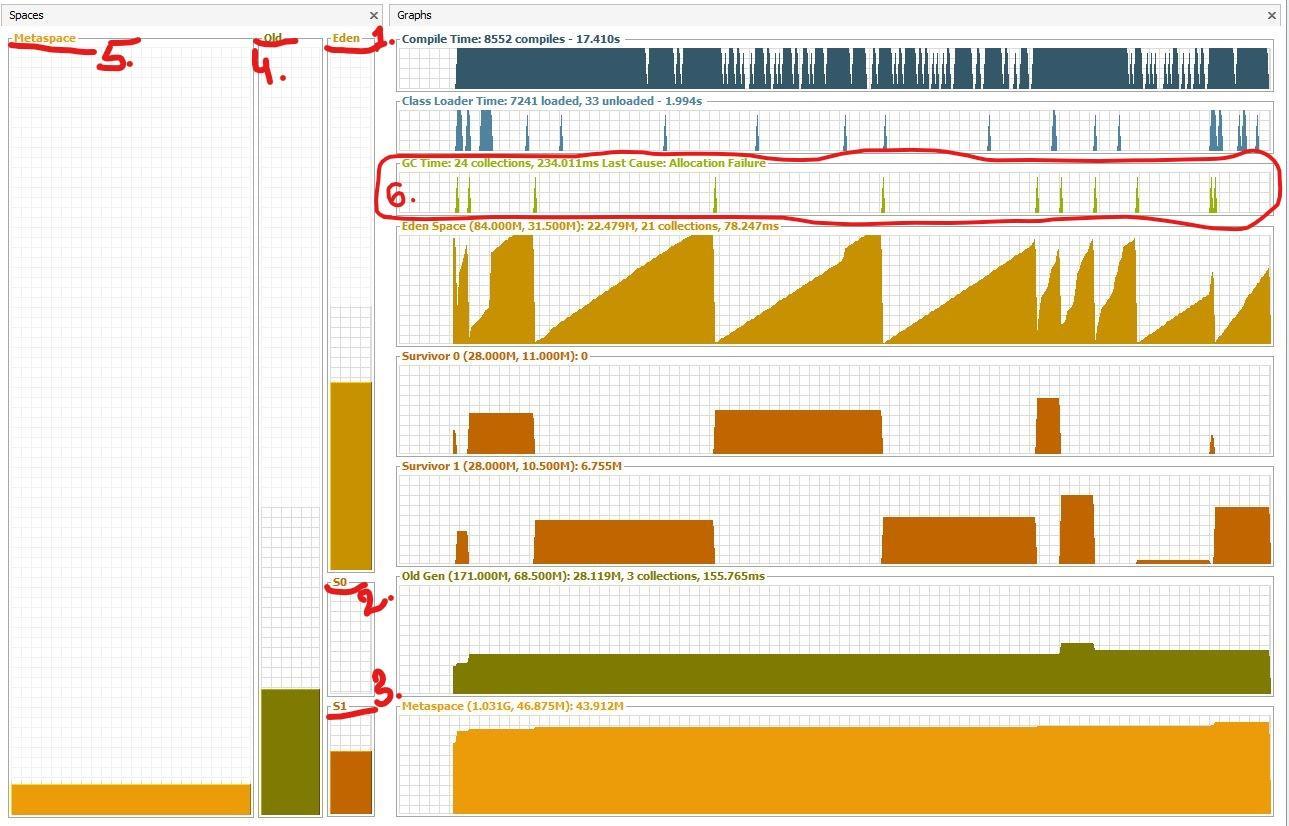
Some few things about the garbage collection:

* This process is triggered automatically by Java, and it is up to Java when and whether or not to start this process.
* It is actually an expensive process. When the garbage collector runs, all threads in your application are paused (depending on the GC type, which will be discussed later).
* This is actually a more complicated process than just garbage collecting and freeing up memory.

Even though Java decides when to run the garbage collector, we may explicitly call System.gc() but it is not advised.

In this part will show the working process of garbage collection and structure of heap memory:

1. When an object is created, it is allocated on the **Eden(1)** space in Figure 7. Because the Eden space is not that big, it gets full quite fast.
2. Once an object survives a garbage collecting process, it gets moved into a so-called survivor space **S0(2)**. The second time the minor garbage collector runs on the **Eden space**, it moves all surviving objects into the **S1(3)** space. Also, everything that is currently on **S0(2)** is moved into the **S1(3)** space.
3. If an object survives for X rounds of minor garbage collection (X depends on the JVM implementation) it is most likely that it will survive forever, and it gets moved into the **Old(4)** space.
4. The old generation can be also garbage collected by Major garbage collector, but since it is a bigger part of the memory compared to Eden space, it does not happen that often.

**Figure 8. Heap memory generations**

The **Metaspace(5)** which is changed from PERM since Java 8, is used to store the metadata about your loaded classes in the JVM.

🡺 From those Garbage Collection working process, the leak memory problem usually happeneded in old generation when major garbage collector can not reduce objects. And it would take the form of *java.lang.OutOfMemoryError: Permgen* *space*

### 1.4.5. How string stores in the memory?

String could be created either by using the new keyword (like any other object) or, by assigning value to the literal (like any other primitive datatype). Like the below example:

public class StringDemo {

   public static void main(String args[]) {

      String stringObject = new String("Hello how are you");

      System.out.println(stringObject);

      String stringLiteral = "Helloworld";

      System.out.println(stringLiteral);

   }

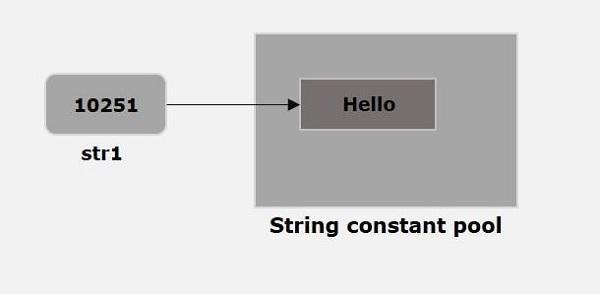
}

In this example, the object new String(“Hello how are you”) stores on heap memory normally like every object. But, object “Helloworld” (also call string literal) stores on another heap memory’s area, called String Constant pool.

! Note:

When you store a String directly as the code below, JVM creates a String object with the given value in a String constant pool and it show like Figure 9.

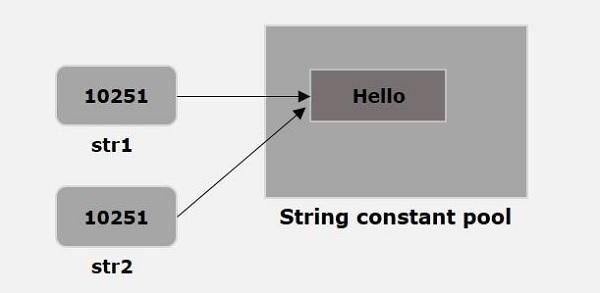
String str1 = "Hello";



**Figure 9. Create a string literal**

And whenever we try to create another String as the code below. JVM verifies weather any String object with the same value exists in the String constant pool, if so, instead of creating a new object JVM assigns the reference of existing object to the new variable like Figure 10.

String str2 = "Hello";



**Figure 10. Create another string literal with the same value**

## 1.5. JRE

## 1.6. JDK

# 2. Lambda expression and functional interface (optional to learn)

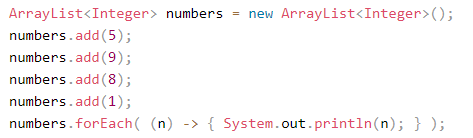
Added by Java 8, lambda expression enhance java because two reasons:

* Add new syntax element that increase the power of language
* New capabilities being incorporated into the API library

## 2.1. Lambda expression

It is an anonymous method (method no-name). However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface. => A lambda expression results in a form of anonymous class.

**Example:**



## 2.2. Anonymous Classes:

Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. => use them if you need to use a local class only once.

Anonymous classes are expressions, which means that you define the class in another expression.

=> The following example, **HelloWorldAnonymousClasses**, uses anonymous classes in the initialization statements of the local variables **frenchGreeting**

Normally, you cannot create an instance from an interface.

**Example:**

public class HelloWorldAnonymousClasses {

interface HelloWorld {

public void greet();

public void greetSomeone(String someone);

}

public void sayHello() {

HelloWorld frenchGreeting = new HelloWorld() {

String name = "tout le monde";

public void greet() {

greetSomeone("tout le monde");

}

public void greetSomeone(String someone) {

name = someone;

System.out.println("Salut " + name);

}

};

}

public static void main(String... args) {

HelloWorldAnonymousClasses myApp =

new HelloWorldAnonymousClasses();

myApp.sayHello();

}

}

In example, because an anonymous class definition is an expression, it must be part of a statement => there is a semicolon after the closing brace.

## 2.3. Functional Interface

A functional interface is an interface that contains one and only one abstract method. Normally, this method specifies the intended purpose of the interface => typically represents a single action.

**For example:**

The standard interface **Runnable** is a functional interface because it defines only one method: run( ). Therefore, run( ) defines the action of Runnable.

A functional interface defines the target type of lambda expression.

**Key point:** A lambda expression can be used only in a context in which its target type is specified

**More details:**

**1. Lambda expression**

New syntax element is −>. It divides a lambda expression into two parts.

- On the left side specifies any parameters. (If no parameters are needed, an empty parameter list is used.)

- On the right side is the lambda body, which specifies the actions of the lambda expression. (can be a expression or block of code)

**Example:**

( ) −> 96.69

equals:

double myMethod( ) {

return 96.69;

}

**2. Functional interface**

**Example:**

interface MyNumber {

double getValue();

}

A lambda expression is not executed on its own. Rather, it forms the implementation of the abstract method defined by the functional interface that specifies its target type. => A lambda expression can be specified only in a context in which a target type is defined.

**Combine two of them:**

// create instance:

MyNumber myNum;

myNum = ( ) −> 96.69;

// Call getValue(), which is implemented by the previously assigned

// lambda expression.

System.out.println("myNum.getValue());

**For block of code:**

// A block lambda that computes the factorial of an int value.

interface NumericFunc {

int func(int n);

}

class BlockLambdaDemo {

public static void main(String args[]) {

// This block lambda computes the factorial of an int value.

NumericFunc factorial = (n) -> {

int result = 1;

for(int i=1; i <= n; i++)

result = i \* result;

return result;

};

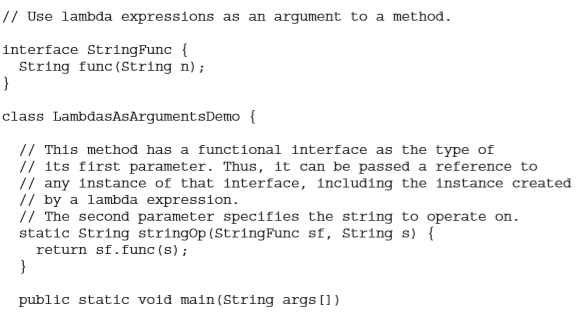
System.out.println("The factorial of 3 is " + factorial.func(3)); System.out.println("The factorial of 5 is " + factorial.func(5));

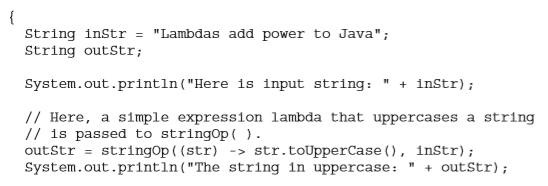
}

}

A lambda expression also can be parsed as an argument => very powerful to parse executable code as an argument to a method. This greatly enhances the expressive power of Java.

Let’s see Example:





# 3. Method references

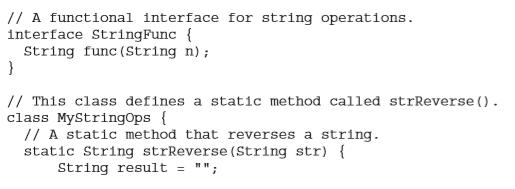
A method reference provides a way to refer to a method without executing it. It requires a target type context that consists of a compatible functional interface. When evaluated, a method reference also creates an instance of the functional interface.

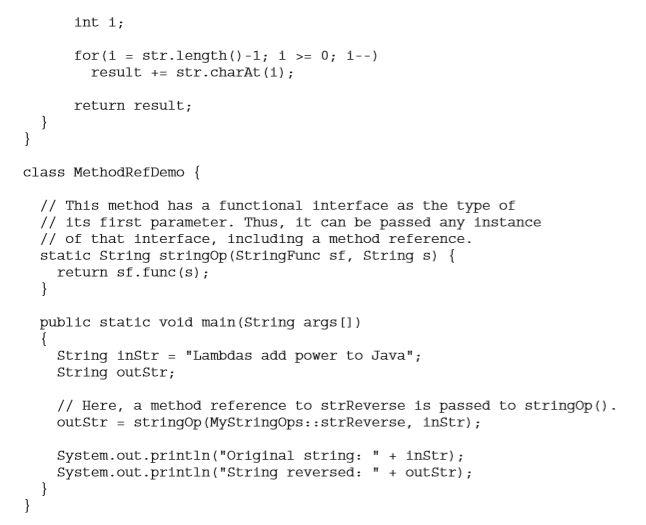
There are different types of method references

**Method reference to static method**

*ClassName::methodName*

**Example:**





*outStr = stringOp(MyStringOps::strReverse, inStr);*

A reference to the **static** method **strReverse( )**, declared inside **MyStringOps**, is passed as the first argument to **stringOp( )**. Because **strReverse** is compatible with the **StringFunc** functional interface.

=> The expression **MyStringOps::strReverse** evaluates to a reference to an object in which **strReverse** provides the implementation of **func( )** in **StringFunc.**

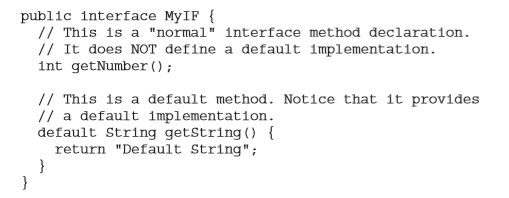
# 4. Default and static method in interface

## 4.1. Default method in interface

The methods specified by an interface were abstract, containing nobody. The release of JDK 8 has changed this by adding a new capability to interface called the default method. A default method lets you define a default implementation for an interface method

=> It is now possible for an interface method to provide a body.

**Example:**



Because **getString( )** includes a default implementation => it is not necessary for an implementing class to override it. In other words, if an implementing class does not provide its own implementation, the default is used.

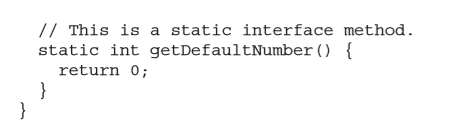
## 4.2. Static method in interface

JDK 8 added another new capability to interface: the ability to define one or more static methods. Like static methods in a class, a static method defined by an interface can be called independently of any object.

=> No implementation of the interface is necessary, and no instance of the interface is required, in order to call a static method. we use:

*InterfaceName.staticMethodName*

**Example:**



This method can be called using:



=> As mentioned, no implementation or instance of MyIF is required to call getDefaultNumber( ) because it is static.

**One last point:** static interface methods are not inherited by either an implementing class or a sub-interface.

# 5. Java Stream API

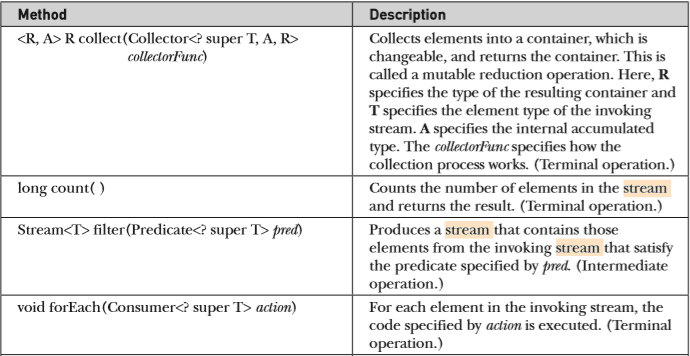
Of the many new features added by JDK 8, the two most important are lambda expressions and the stream API.

**Stream**: a stream is a conduit for data. Thus, a stream represents a sequence of objects. => operates on a data source, such as an array or a collection.

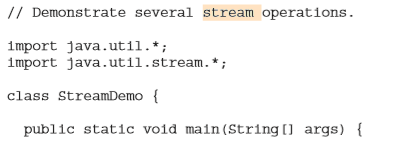
**Keypoint:** never storage for data. It moves data, filter, sort, or otherwise operator (not modify the data source).

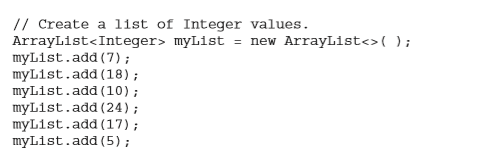
**Stream interface:**

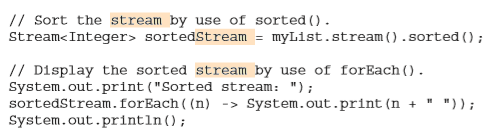
**Stream API** defines several stream interfaces, let’s see some example below:



**A Simple Stream Example**







ForEach combine with functional interface

**Using parallel stream:**

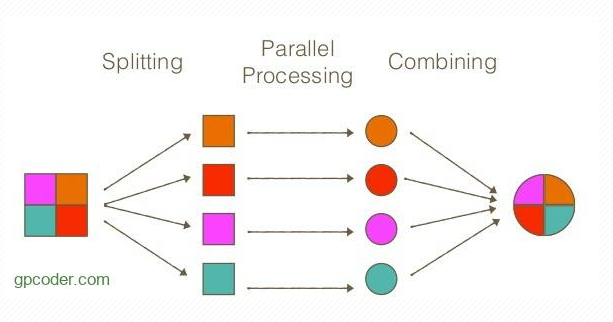
The parallel ( ) method:

*S parallel( )*

It returns a parallel stream based on the sequential stream that invokes it. (If it is called on a stream that is already parallel, then the invoking stream is returned.)

=> Once a parallel stream has been obtained, operations on the stream can occur in parallel.

**How it works**

****

**Example:**

Operation in the preceding program can be parallelized by substituting parallelStream( ) for the call to stream( ):



The results will be the same, but the multiplications can occur in different threads.

# Reference

1. Introduction (Kiên lớn)

<https://www.programiz.com/java-programming/>

<https://dzone.com/articles/jvm-architecture-explained>

<https://www.youtube.com/watch?v=QHIWkwxs0AI&ab_channel=BigDataElearningBigDataElearning>

2-4. Lamda expression, method reference,… (Chương)

<https://drive.google.com/file/d/1QIBdmvCA9eRfGKrUzZ0bOlRwtAPr5eln/view?usp=sharing>