

Contents

JVM architecture:.....	1
Class loader:	8
Bootstrap class loader:.....	9
Custom Class loader:.....	11
JVM Memory model.....	13
Heap Memory:	13
NON heap memory:	14
Meta Space:	14
Other memory	14
Metaspace.....	15
Code Cache	15
Method Area.....	15
Java Virtual Machine (JVM).....	19
Garbage collection:	19
Disadvantage of GC:.....	25
Stop the world:	25
Object Allocation.....	25

JVM architecture:

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a java code. JVM is a part of JRE(Java Run Environment).

Java applications are called WORA (Write Once Run Everywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

When we compile a *.java* file, a *.class* file(contains byte-code) with the same filename is generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.

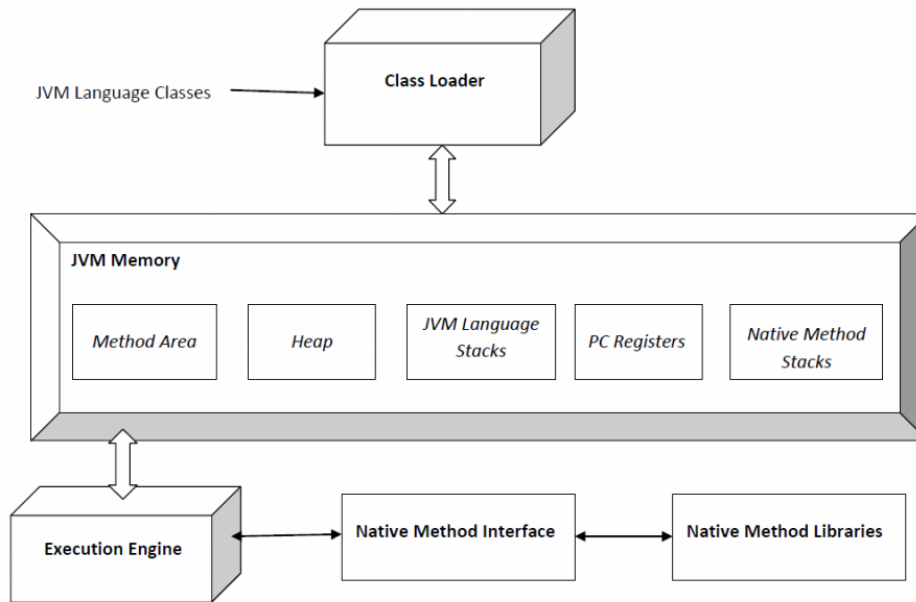


Image Source : https://en.wikipedia.org/wiki/Java_virtual_machine

Class Loader Subsystem

It is mainly responsible for three activities.

- Loading
- Linking
- Initialization

Loading : The Class loader reads the *.class* file, generate the corresponding binary data and save it in method area. For each *.class* file, JVM stores following information in method area.

- Fully qualified name of the loaded class and its immediate parent class.
- Whether *.class* file is related to Class or Interface or Enum
- Modifier, Variables and Method information etc.

After loading *.class* file, JVM creates an object of type *Class* to represent this file in the heap memory. Please note that this object is of type *Class* predefined in *java.lang* package. This *Class* object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc. To get this object reference we can use *getClass()* method of *Object* class.

```
// A Java program to demonstrate working of a Class type
// object created by JVM to represent .class file in
// memory.
```

```

import java.lang.reflect.Field;
import java.lang.reflect.Method;

// Java code to demonstrate use of Class object
// created by JVM
public class Test
{
    public static void main(String[] args)
    {
        Student s1 = new Student();

        // Getting hold of Class object created
        // by JVM.
        Class c1 = s1.getClass();

        // Printing type of object using c1.
        System.out.println(c1.getName());

        // getting all methods in an array
        Method m[] = c1.getDeclaredMethods();
        for (Method method : m)
            System.out.println(method.getName());

        // getting all fields in an array
        Field f[] = c1.getDeclaredFields();
        for (Field field : f)
            System.out.println(field.getName());
    }
}

// A sample class whose information is fetched above using
// its Class object.
class Student
{
    private String name;
    private int roll_No;

```

```
public String getName() { return name; }
public void setName(String name) { this.name = name; }
public int getRoll_no() { return roll_No; }
public void setRoll_no(int roll_no) {
    this.roll_No = roll_no;
}
}
```

Run on IDE

Output:

```
Student
getName
setName
getRoll_no
setRoll_no
name
roll_No
```

Note : For every loaded *.class* file, only **one** object of Class is created.

```
Student s2 = new Student();
// c2 will point to same object where
// c1 is pointing
Class c2 = s2.getClass();
System.out.println(c1==c2); // true
```

Linking : Performs verification, preparation, and (optionally) resolution.

- *Verification* : It ensures the correctness of *.class* file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError*.
- *Preparation* : JVM allocates memory for class variables and initializing the memory to default values.

- *Resolution* : It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

Initialization : In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in class hierarchy. In general there are three class loaders :

- *Bootstrap class loader* : Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in `JAVA_HOME/jre/lib` directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.
- *Extension class loader* : It is child of bootstrap class loader. It loads the classes present in the extensions directories `JAVA_HOME/jre/lib/ext`(Extension path) or any other directory specified by the `java.ext.dirs` system property. It is implemented in java by the `sun.misc.Launcher$ExtClassLoader` class.
- *System/Application class loader* : It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to `java.class.path`. It is also implemented in Java by the `sun.misc.Launcher$AppClassLoader` class.

// Java code to demonstrate Class Loader subsystem

```
public class Test
{
    public static void main(String[] args)
    {
        // String class is loaded by bootstrap loader, and
        // bootstrap loader is not Java object, hence null
        System.out.println(String.class.getClassLoader());

        // Test class is loaded by Application loader
        System.out.println(Test.class.getClassLoader());
    }
}
```

Run on IDE

Output:

null

sun.misc.Launcher\$AppClassLoader@73d16e93

Note : JVM follow Delegation-Hierarchy principle to load classes. System class loader delegate load request to extension class loader and extension class loader delegate request to boot-strap class loader. If class found in boot-strap path, class is loaded otherwise request again transfers to extension class loader and then to system class loader. At last if system class loader fails to load class, then we get run-time exception *java.lang.ClassNotFoundException*.

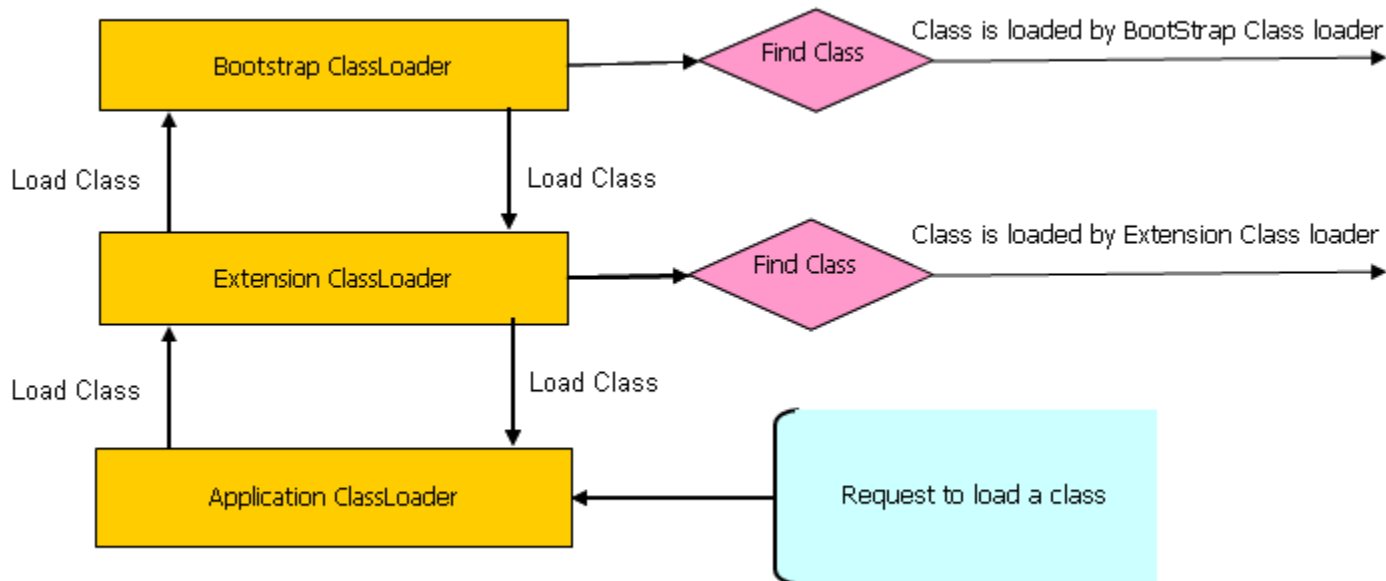


Image Source: <http://javarevisited.blogspot.in/2012/12/how-classloader-works-in-java.html>

JVM Memory

Method area : In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

Heap area : Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.

Stack area : For every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding

frame. After a thread terminate, it's run-time stack will be destroyed by JVM. It is not a shared resource.

PC Registers :Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.

Native method stacks :For every thread, separate native stack is created. It stores native method information.

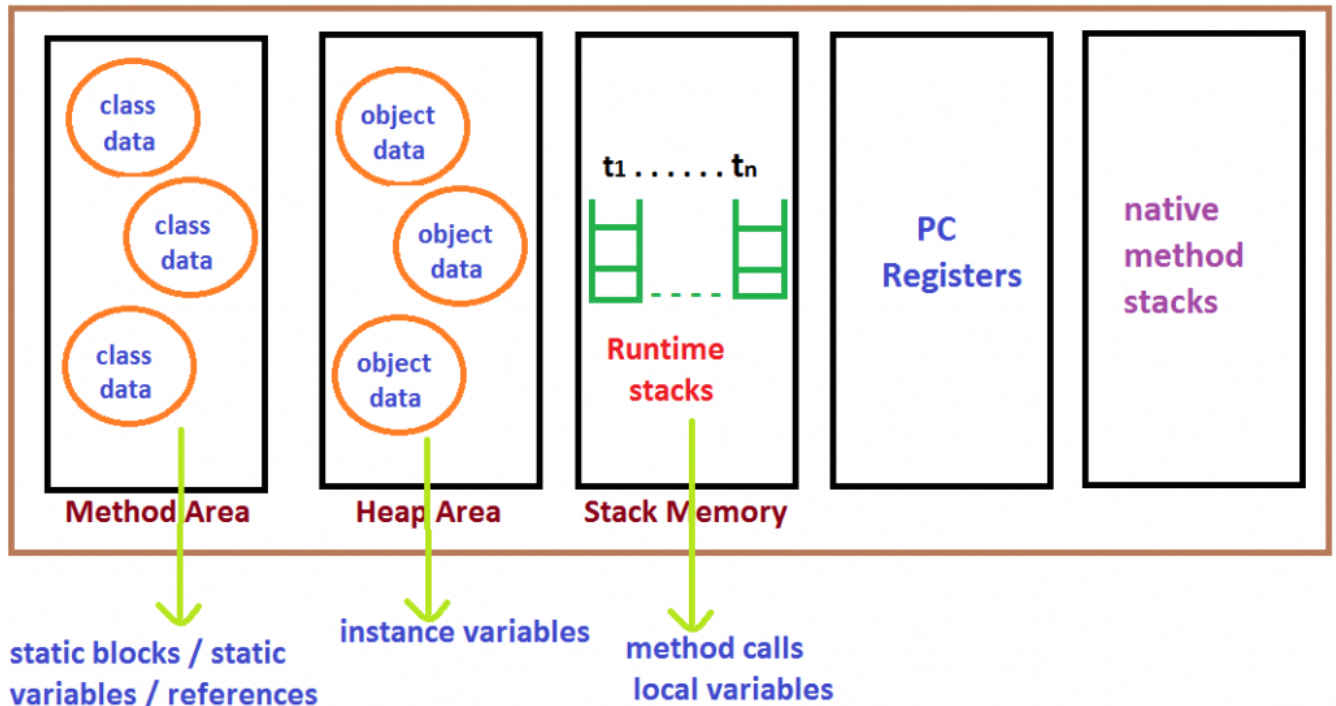


Image Source : <http://java.scjp.jobs4times.com/fund/fund2.png>

Execution Engine

Execution engine execute the *.class* (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts :-

- *Interpreter* : It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- *Just-In-Time Compiler(JIT)* : It is used to increase efficiency of interpreter.It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls,JIT provide direct native code for that part so re-interpretation is not required,thus efficiency is improved.
- *Garbage Collector* : It destroy un-referenced objects.For more on Garbage Collector,refer [Garbage Collector](#).

Java Native Interface (JNI) :

It is a interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

Native Method Libraries :

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

Class loader:

There are three types of built-in ClassLoader in Java:

1. **Bootstrap Class Loader** – It loads JDK internal classes, typically loads rt.jar and other core classes for example java.lang.* package classes
 2. **Extensions Class Loader** – It loads classes from the JDK extensions directory, usually \$JAVA_HOME/lib/ext directory.
 3. **System Class Loader** – It loads classes from the current classpath that can be set while invoking a program using -cp or -classpath command line options.
- ✓ Java ClassLoader are hierarchical and whenever a request is raised to load a class, it delegates it to its parent and in this way uniqueness is maintained in the runtime environment. If the parent class loader doesn't find the class then the class loader itself tries to load the class.
 - ✓ Note that the classloader hierarchy is not an inheritance hierarchy, but a delegation hierarchy.
 - ✓ It follows uniqueness and visibility principle. Uniqueness means class loaded by parent class loader should not be loaded again by child class loader however this can be violated by custom class loader.
 - ✓ One more important point to note is that Classes loaded by a child class loader have visibility into classes loaded by its parent class loaders. So

classes loaded by System ClassLoader have visibility into classes loaded by Extensions and Bootstrap ClassLoader.

- ✓ If there are sibling class loaders then they can't access classes loaded by each other.

Bootstrap class loader:

Java virtual machine implementations must be able to recognize and load classes and interfaces stored in binary files that conform to the Java class file format. An implementation is free to recognize other binary forms besides class files, but it must recognize class files.

Every Java virtual machine implementation has a bootstrap class loader, which knows how to load trusted classes, including the classes of the Java API. The Java virtual machine specification doesn't define how the bootstrap loader should locate classes. That is another decision the specification leaves to implementation designers.

Given a fully qualified type name, the bootstrap class loader must *in some way* attempt to produce the data that defines the type. One common approach is demonstrated by the Java virtual machine implementation in Sun's 1.1 JDK on Windows98. This implementation searches a user-defined directory path stored in an environment variable named CLASSPATH. The bootstrap loader looks in each directory, in the order the directories appear in the CLASSPATH, until it finds a file with the appropriate name: the type's simple name plus ".class". Unless the type is part of the unnamed package, the bootstrap loader expects the file to be in a subdirectory of one the directories in the CLASSPATH. The path name of the subdirectory is built from the package name of the type. For example, if the bootstrap class loader is searching for class java.lang.Object, it will look for Object.class in the java\lang subdirectory of each CLASSPATH directory.

In 1.2, the bootstrap class loader of Sun's Java 2 SDK only looks in the directory in which the system classes (the class files of the Java API) were installed. The bootstrap class loader of the implementation of the Java virtual machine from Sun's Java 2 SDK does not look on the CLASSPATH. In Sun's Java 2 SDK virtual

machine, searching the class path is the job of the *system class loader*, a user-defined class loader that is created automatically when

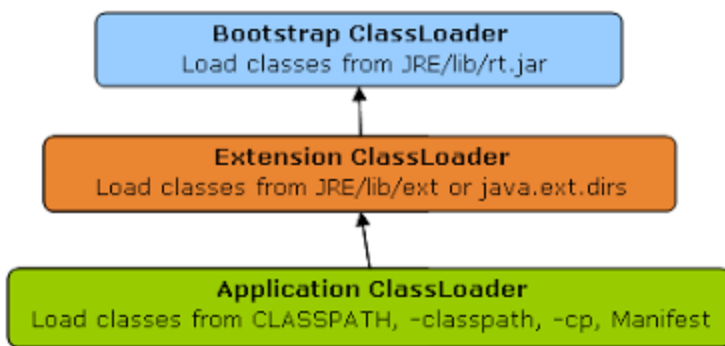
Every class loader has a predefined location, from where they loads class files. Bootstrap ClassLoader is responsible for loading standard JDK class files from rt.jar and it is parent of all class loaders in Java. Bootstrap class loader don't have any parents, if you call `String.class.getClassLoader()` it will return null and any code based on that may throw [NullPointerException in Java](#). Bootstrap class loader is also known as **Primordial ClassLoader** in Java.

Extension class loader:

Extension ClassLoader delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class from jre/lib/ext directory or any other directory pointed by java.ext.dirs system property. Extension ClassLoader in JVM is implemented by `sun.misc.Launcher$ExtClassLoader`.

System/Appclass loader:

Third default class loader used by JVM to load Java classes is called System or Application class loader and it is responsible for loading application specific classes from [CLASSPATH](#) environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR. Application class loader is a child of Extension ClassLoader and its implemented by `sun.misc.Launcher$AppClassLoader` class. Also, except Bootstrap class loader, which is implemented in native language mostly in C, all Java class loaders are implemented using `java.lang.ClassLoader`.



If getClassloader returns NULL means class is loaded by bootstrap class loader because there is non parent for bootstrap.

JVM creates CLASS object per loaded class only once.

Custom Class loader:

When JVM requests for a class, it invokes `loadClass` function of the ClassLoader by passing the fully classified name of the Class.

`loadClass` function calls for `findLoadedClass()` method to check that the class has been already loaded or not. It's required to avoid loading the class multiple times.

If the Class is not already loaded then it will delegate the request to parent ClassLoader to load the class.

If the parent ClassLoader is not finding the Class then it will invoke `findClass()` method to look for the classes in the file system.

1. `private byte[] loadClassFileData(String name):`

This method will read the class file from file system to byte array.

2. `private Class getClass(String name)`

This method will call the `loadClassFileData()` function and by invoking the parent `defineClass()` method, it will generate the Class and return it.

3. `public Class loadClass(String name):`

This method is responsible for loading the Class. If the class name starts with `com.journaldev` (Our sample classes) then it will load it using `getClass()` method or else it will invoke the parent `loadClass` function to load it.

4. `public CCLoader(ClassLoader parent):`

This is the constructor which is responsible for setting the parent `ClassLoader`.

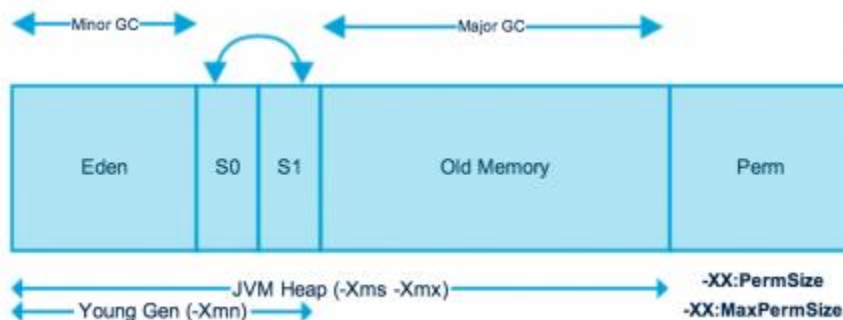
JVM Memory model

Once we launch the JVM, the operating system allocates memory for the process. Here, the JVM itself is a process, and the memory allocated to that process includes the Heap, Meta Space, JIT code cache, thread stacks, and shared libraries. We call this native memory. “**Native Memory**” is the memory provided to the process by the operating system. How much memory the operating system allocates to the Java process depends on the operating system, processor, and the JRE. Let me explain the different memory blocks available for the JVM.

Heap Memory:

JVM uses this memory to store objects.

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.



This memory is in turn split into two different areas called the “**Young Generation Space**” and “**Tenured Space**”.

Young Generation: The Young Generation or the New Space is divided into two portions called “**Eden Space**” and “**Survivor Space**”.

Eden Space: When we create an object, the memory will be allocated from the Eden Space.

Survivor Space: This contains the objects that have survived from the Young garbage collection or Minor garbage collection. We have two equally divided survivor spaces called S0 and S1.

Tenured Space: The objects which reach to max tenured threshold during the minor GC or young GC, will be moved to “**Tenured Space**” or “**Old Generation Space**”.

When we discuss the garbage collection process we will come to know how the above memory locations will be used.

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.

Meta Space: This memory is out of heap memory and part of the native memory. As per the document by default the meta space doesn't have an upper limit. In earlier versions of Java we called this “**Perm Gen Space**”. This space is used to store the class definitions loaded by class loaders. This is designed to grow in order to avoid Out of memory errors. However, if it grows more than the available physical memory, then the operating system will use virtual memory. This will have an adverse effect on application performance, as swapping the data from virtual memory to physical memory and vice versa is a costly operation. We have JVM options to limit the Meta Space used by the JVM. In that case, we may get out of memory errors.

Other memory

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

Metaspace

With Java 8, there is no Perm Gen, that 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.

Code Cache

When a Java program is run, it executes the code in a tiered manner. In the first tier, it uses client compiler (C1 compiler) in order to compile the code with instrumentation. The profiling data is used in the second tier (C2 compiler) for the server compiler, to compile that code in an optimized manner. Tiered compilation is not enabled by default in Java 7, but is enabled in Java 8.

The Just-In-Time (JIT) compiler stores the compiled code in an area called code cache. It is a special heap that holds the compiled code. This area is flushed if its size exceeds a threshold and these objects are not relocated by the GC.

Some of the performance issues and the problem of the compiler not getting re-enabled has been addressed in Java 8 and one of the solution to avoid these issues in Java 7 is to increase the size of the code cache up to a point never being reached.

Method Area

Inside a Java virtual machine instance, information about loaded types is stored in a logical area of memory called the method area. When the Java virtual machine

loads a type, it uses a class loader to locate the appropriate class file. The class loader reads in the class file--a linear stream of binary data--and passes it to the virtual machine. The virtual machine extracts information about the type from the binary data and stores the information in the method area. Memory for class (static) variables declared in the class is also taken from the method area.

All threads share the same method area, so access to the method area's data structures must be designed to be thread-safe. If two threads are attempting to find a class named `Lava`, for example, and `Lava` has not yet been loaded, only one thread should be allowed to load it while the other one waits.

The size of the method area need not be fixed. As the Java application runs, the virtual machine can expand and contract the method area to fit the application's needs. Also, the memory of the method area need not be contiguous. It could be allocated on a heap--even on the virtual machine's own heap. Implementations may allow users or programmers to specify an initial size for the method area, as well as a maximum or minimum size.

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.

Method area stores per-class structures such as the runtime constant pool; field and method data; the code for methods and constructors, including the special methods used in class, instance, and interface initialization.

The method area is created on the virtual machine startup. Although it is logically a part of the heap but it can or cannot be garbage collected, whereas we already read that garbage collection in heap is not optional; it's mandatory. The method area may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger method area becomes unnecessary. The memory for the method area does not need to be contiguous.

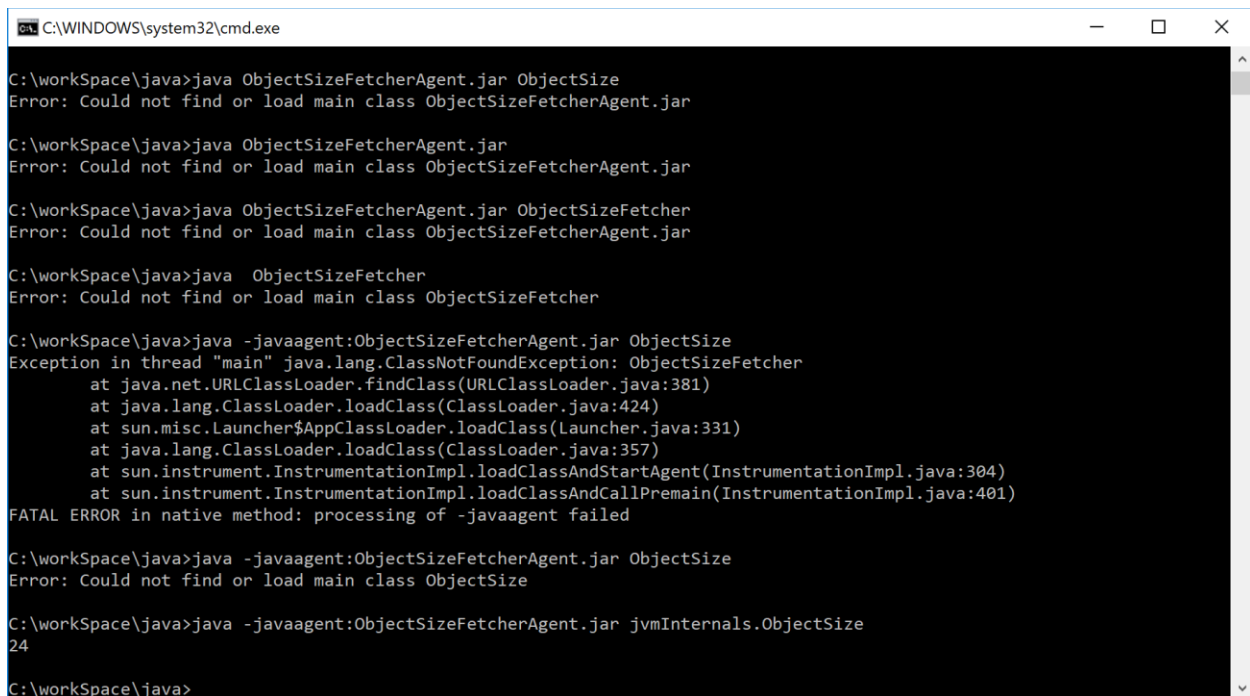
Memory Pool

Memory Pools are created by JVM memory managers to create pool of immutable objects. Memory Pool can belong to Heap or Perm Gen, depending on JVM memory manager implementation.

Runtime Constant Pool

Runtime constant pool is a per-class runtime representation of constant pool in a class. It contains class runtime constants and static methods. Runtime constant pool is part of the method area.

Object size:



```
C:\WINDOWS\system32\cmd.exe

C:\workspace\java>java ObjectSizeFetcherAgent.jar ObjectSize
Error: Could not find or load main class ObjectSizeFetcherAgent.jar

C:\workspace\java>java ObjectSizeFetcherAgent.jar
Error: Could not find or load main class ObjectSizeFetcherAgent.jar

C:\workspace\java>java ObjectSizeFetcherAgent.jar ObjectSizeFetcher
Error: Could not find or load main class ObjectSizeFetcherAgent.jar

C:\workspace\java>java ObjectSizeFetcher
Error: Could not find or load main class ObjectSizeFetcher

C:\workspace\java>java -javaagent:ObjectSizeFetcherAgent.jar ObjectSize
Exception in thread "main" java.lang.ClassNotFoundException: ObjectSizeFetcher
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    at sun.instrument.InstrumentationImpl.loadClassAndStartAgent(InstrumentationImpl.java:304)
    at sun.instrument.InstrumentationImpl.loadClassAndCallPremain(InstrumentationImpl.java:401)
FATAL ERROR in native method: processing of -javaagent failed

C:\workspace\java>java -javaagent:ObjectSizeFetcherAgent.jar ObjectSize
Error: Could not find or load main class ObjectSize

C:\workspace\java>java -javaagent:ObjectSizeFetcherAgent.jar jvmInternals.ObjectSize
24

C:\workspace\java>
```

This pool is a subpart of the Method Area. Since it's an important part of the metadata, Oracle specifications describe the Runtime constant pool apart from the Method Areas. This constant pool is increased for each loaded class/interface. This pool is like a symbol table for a conventional programming language. In other words, when a class, method or field is referred to, the JVM searches the actual

address in the memory by using the runtime constant pool. It also contains constant values like string literals or constant primitives.

Java Stack Memory

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

Java Heap Memory Switches

Java provides a lot of memory switches that we can use to set the memory sizes and their ratios. Some of the commonly used memory switches are:

VM Switch	VM Switch Description
-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 is 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 allocated 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 1

Table 1.1

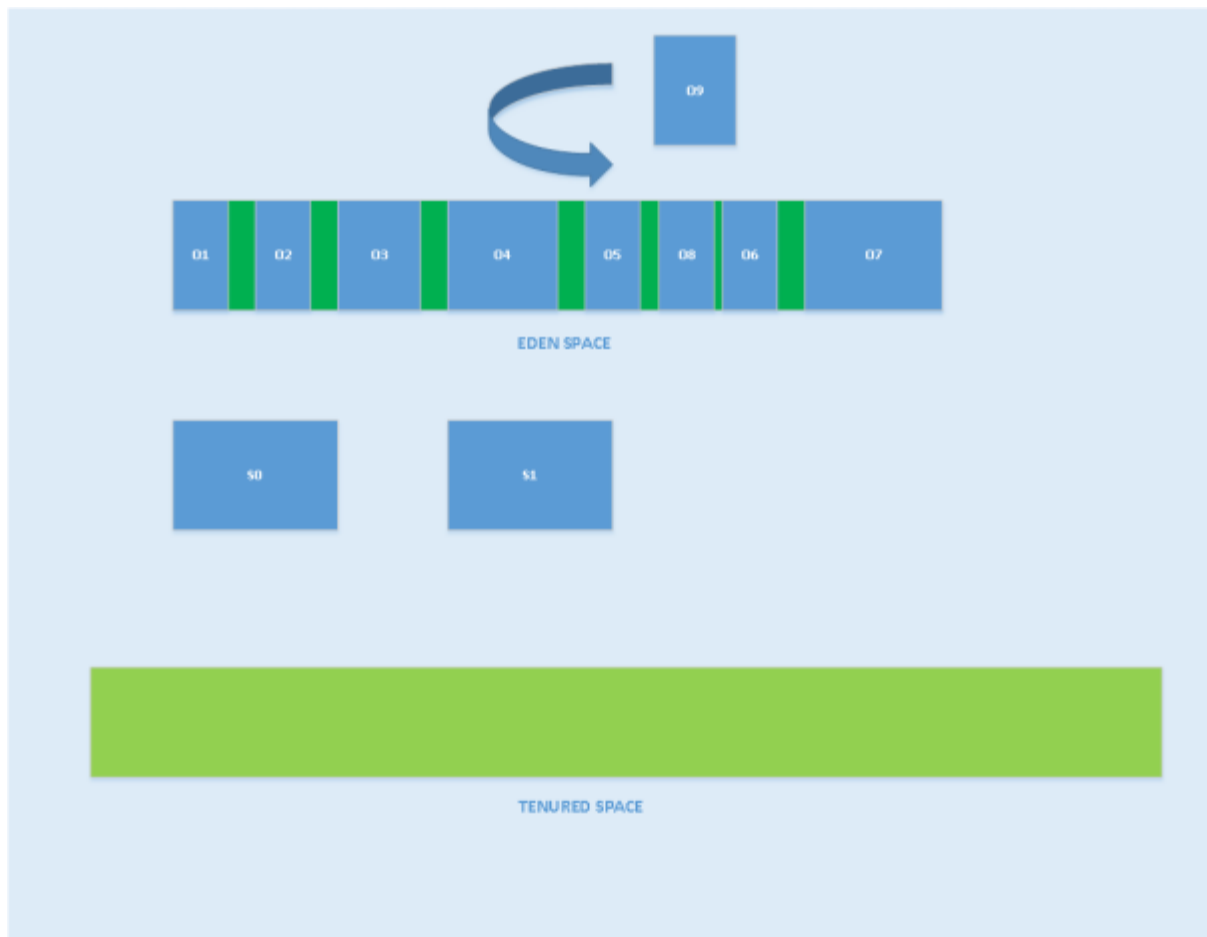
Java Virtual Machine (JVM)

The 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. But the implementation has been provided by Sun and other companies), **implementation** (known as (JRE) Java Runtime Environment) and **instance** (after writing Java command, to run Java class, an instance of JVM is created).

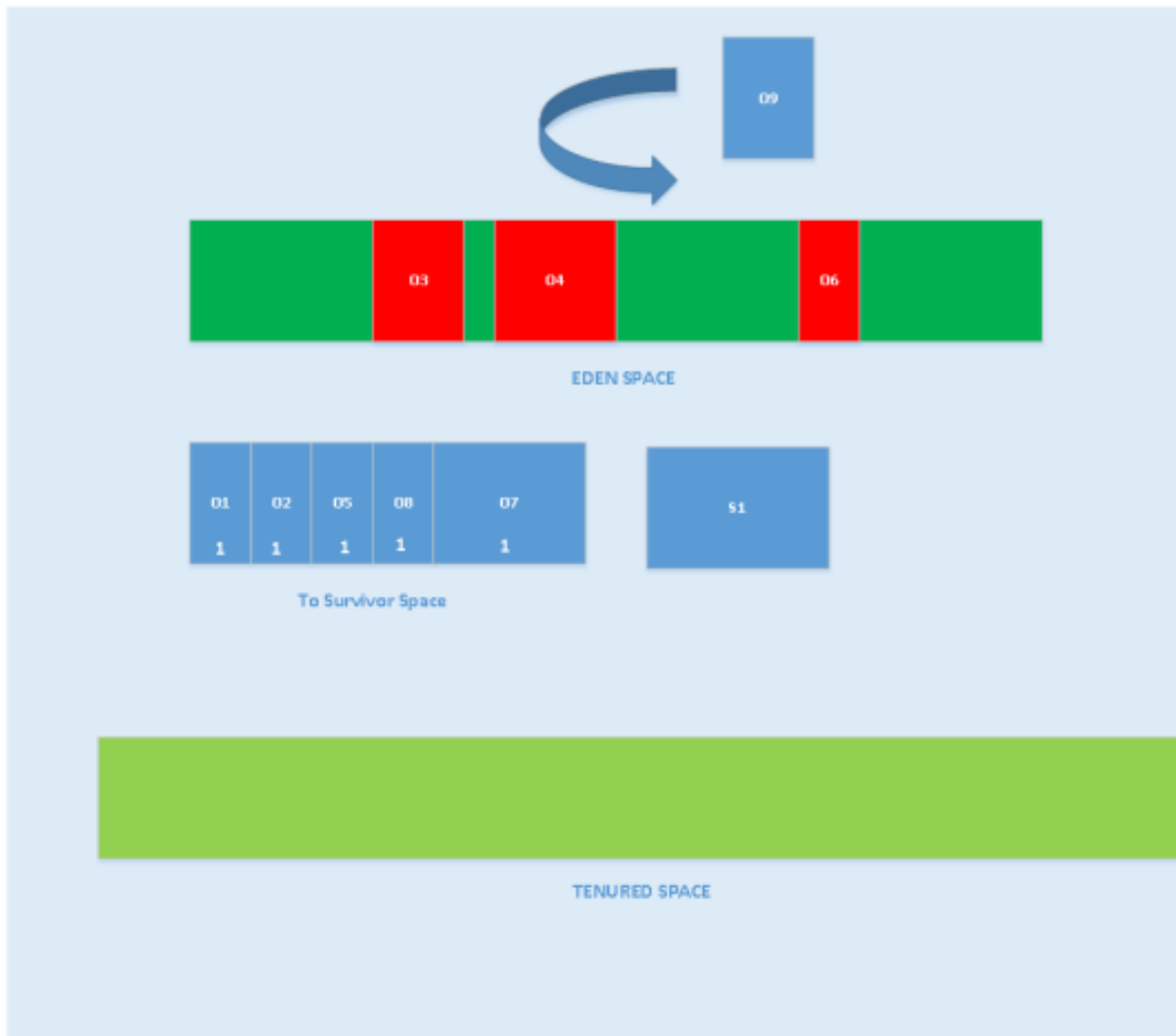
The JVM loads the code, verifies the code, executes the code, manages memory (this includes allocating memory from the Operating System (OS), managing Java allocation including heap compaction and removal of garbage objects) and finally provides the runtime environment.

Garbage collection:

The JVM uses a separate demon thread to do garbage collection. As we said above when the application creates the object the JVM try to get the required memory from the eden space. The JVM performs GC as minor GC and major GC. Let us understand the minor GC.

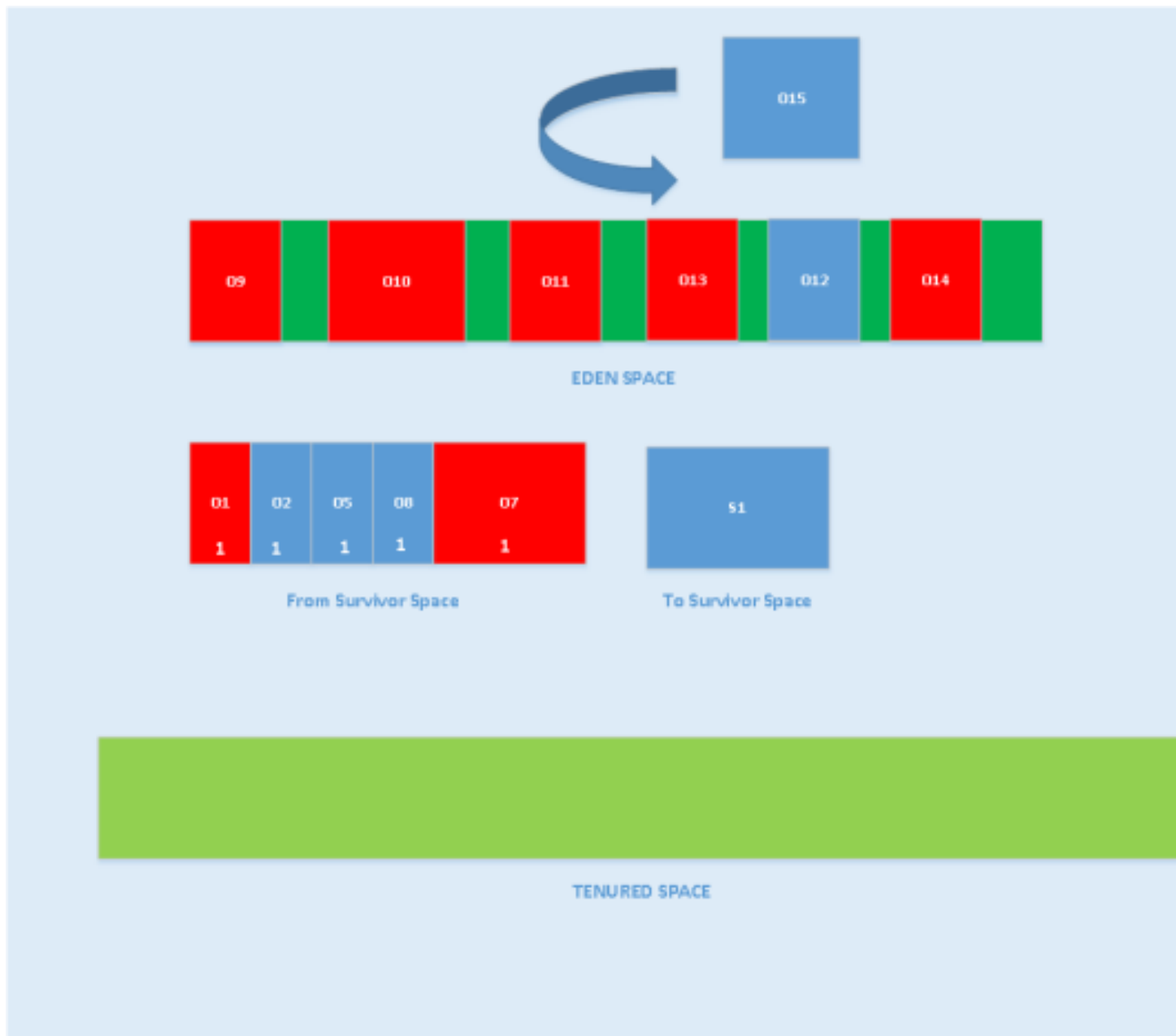


Initially the survivor space and the tenured space is empty. When the JVM is not able to get the memory from the eden space it initiates minor GC. During minor GC, the objects which are not reachable are marked to be collected. The JVM selects one of the survivor spaces as “To Space”. It might be S0/S1. Let us say the JVM selected S0 as the “To Space”. The JVM copies the reachable objects to “To Space”, S0, and increments the reachable object's age by 1. The objects which are not fit into the survivor space will be moved to the tenured space. This process is called “**premature promotion**”. *For this image's purpose I have made the “To Space” bigger than the allocated space. Remember the survivor space won't grow.*

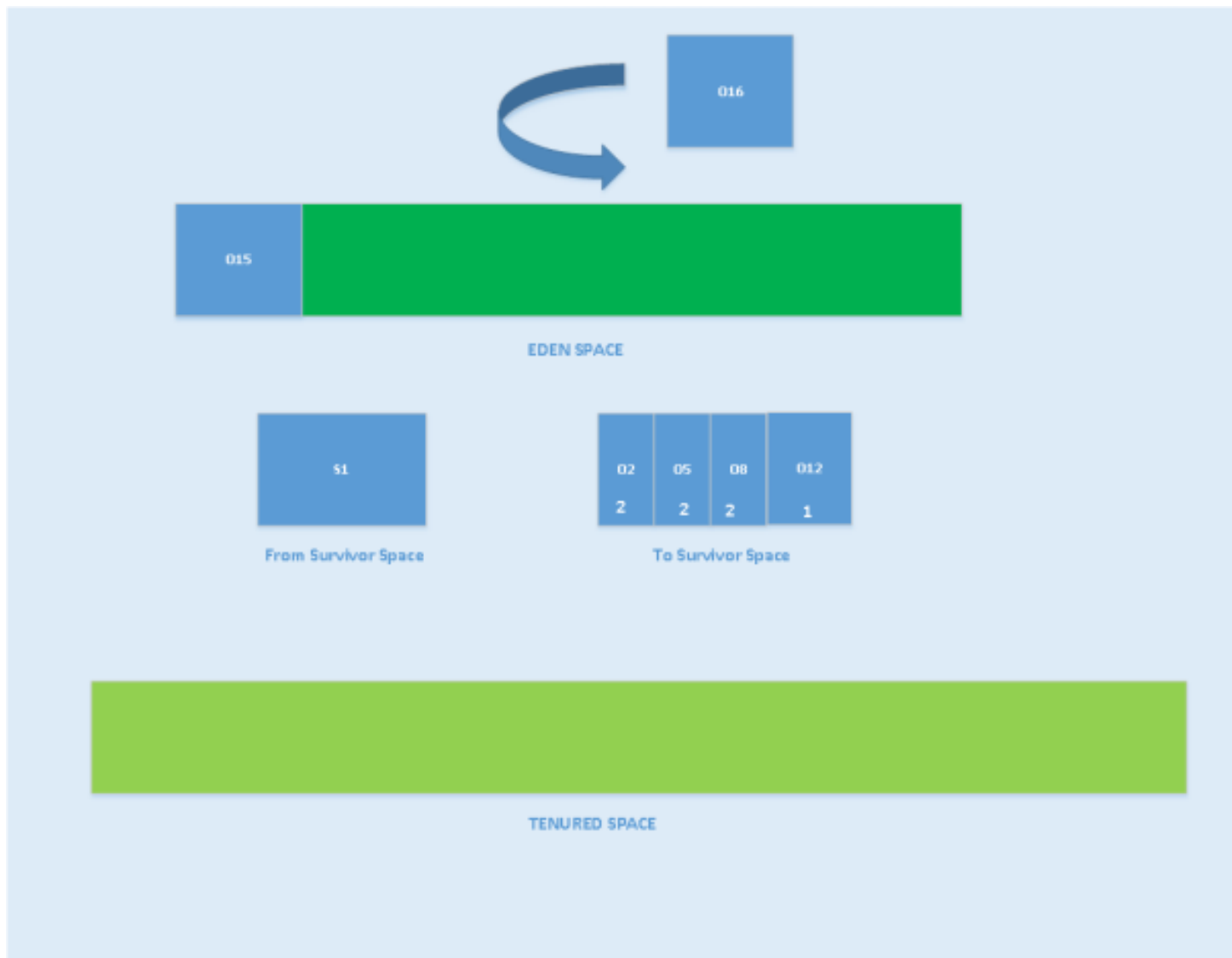


In the above diagram, the objects marked with a red color indicates that they are non-reachable. All the reachable objects are GC roots. The garbage collector won't remove the GC roots. The garbage collector removes the non-reachable objects and empties the eden space.

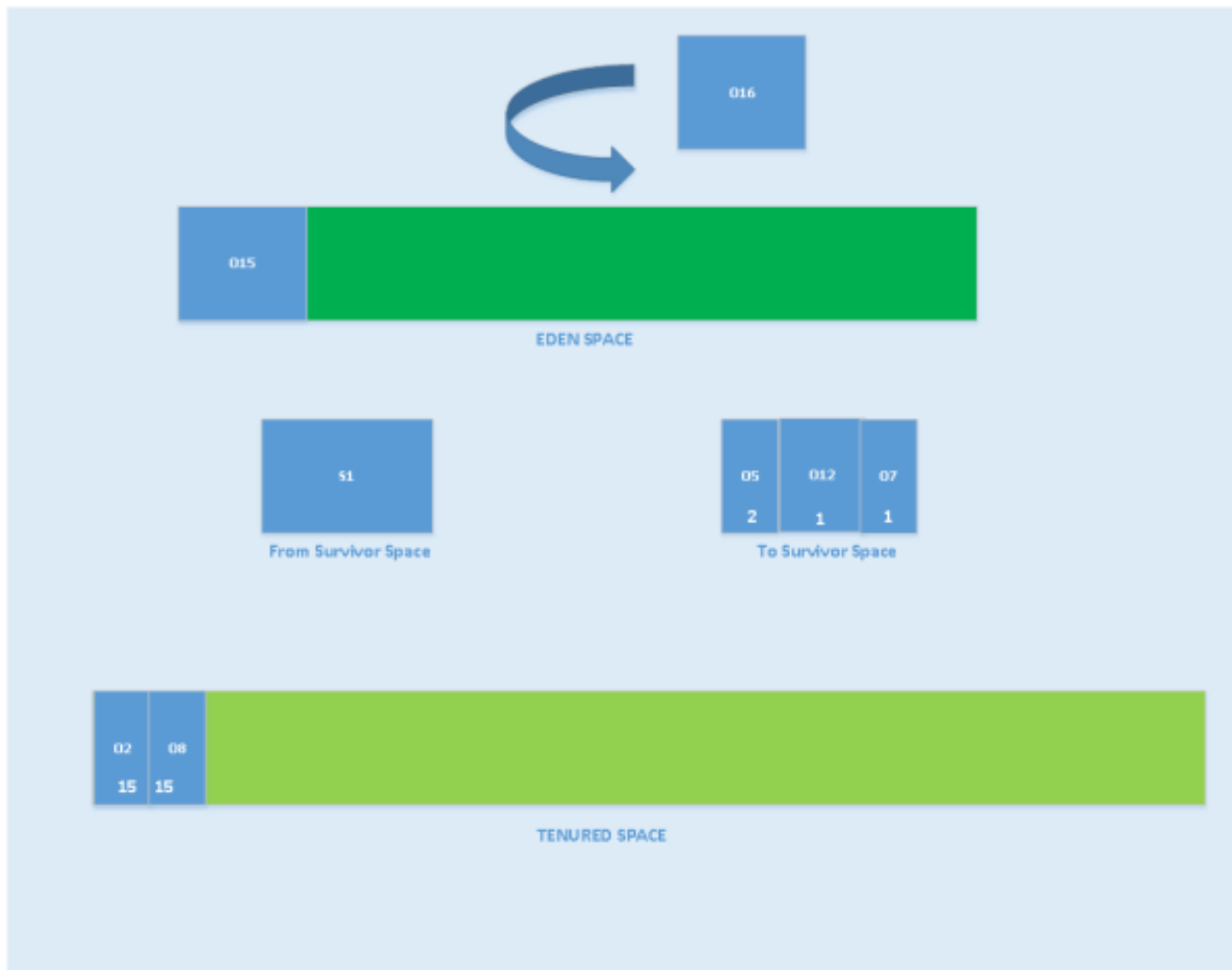
For the second minor GC, the garbage collector marks the non-reachable objects from "eden space" and the "To survivor space (S0)", copies the GC roots to the other survivor space S1, and the reachable object's age will be incremented.



In the above diagram, the objects marked with red are eligible for GC, and the other objects will be copied from the eden and survivor spaces to the other survivor space, S1, and the objects age incrementally.



The above process repeats for each minor GC. When objects reach the max age threshold, then those objects are copied to the tenured space.



There is a JVM level option called “**MaxTenuringThreshold**” to specify the object age threshold to promote the object to tenured space. By default the value is 15.

So it is clear that minor GC reclaims the memory from the “**Young Generation Space**”. Minor GC is a “**stop the world**” process. Some times the application pause is negligible. The minor GC will be performed with single thread or multi-thread, based on the GC collector applied.

If minor GC triggers several times, eventually the “**Tenured Space**” will be filled up and will require more garbage collection. During this time the JVM triggers a “**major GC**” event. Some times we call this full GC. But, as part of full GC, the JVM

reclaims the memory from “Meta Space”. If there are no objects in the heap, then the loaded classes will be removed from the meta space.

Now let us see what possibilities make the JVM trigger major GC.

- If the developer calls `System.gc()`, or `Runtime.getRuntime().gc()` suggests the JVM to initiate GC.
- If the JVM decides there is not enough tenured space.
- During minor GC, if the JVM is not able to reclaim enough memory from the eden or survivor spaces, then a major GC may be triggered.
- If we set a “MaxMetaspaceSize” option for the JVM and there is not enough space to load new classes, then the JVM triggers a major GC.

Disadvantage of GC:

Yes. Whenever the garbage collector runs, it has an effect on the application’s performance. This is because all other threads in the application have to be stopped to allow the garbage collector thread to effectively do its work.

Depending on the requirements of the application, this can be a real problem that is unacceptable by the client. However, this problem can be greatly reduced or even eliminated through skillful optimization and garbage collector tuning and using different GC algorithms.

Stop the world:

When the garbage collector thread is running, other threads are stopped, meaning the application is stopped momentarily. This is analogous to house cleaning or fumigation where occupants are denied access until the process is complete.

Depending on the needs of an application, “stop the world” garbage collection can cause an unacceptable freeze. This is why it is important to do garbage collector tuning and JVM optimization so that the freeze encountered is at least acceptable.

Object Allocation

During object allocation, the JRockit JVM distinguishes between *small* and *large* objects. The limit for when an object is considered

large depends on the JVM version, the heap size, the garbage collection strategy and the platform used, but is usually somewhere between 2 and 128 kB. Please see the documentation for `-XXtlaSize` and `-XXlargeObjectLimit` for more information.

Small objects are allocated in *thread local areas (TLAs)*. The thread local areas are free chunks reserved from the heap and given to a Java thread for exclusive use. The thread can then allocate objects in its TLA without synchronizing with other threads. When the TLA becomes full, the thread simply requests a new TLA. The TLAs are reserved from the nursery if such exists, otherwise they are reserved anywhere in the heap.

Large objects that don't fit inside a TLA are allocated directly on the heap. When a nursery is used, the large objects are allocated directly in old space. Allocation of large objects requires more synchronization between the Java threads, although the JRockit JVM uses a system of caches of free chunks of different sizes to reduce the need for synchronization and improve the allocation speed.