## Class Loaders

Class loaders are responsible for **loading Java classes during runtime dynamically to the JVM** (Java Virtual Machine). Also, they are part of the JRE (Java Runtime Environment). Hence, the JVM doesn't need to know about the underlying files or file systems in order to run Java programs thanks to class loaders.

Also, these Java classes aren't loaded into memory all at once, but when required by an application. This is where class loaders come into the picture. They are responsible for loading classes into memory.

In this tutorial, we're going to talk about different types of built-in class loaders, how they work and an introduction to our own custom implementation.

## 2. Types of Built-in Class Loaders

Let's start by learning how different classes are loaded using various class loaders using a simple example:

**public** **void** **printClassLoaders**() **throws** ClassNotFoundException {

System.out.println("Classloader of this class:"

+ PrintClassLoader.class.getClassLoader());

System.out.println("Classloader of Logging:"

+ Logging.class.getClassLoader());

System.out.println("Classloader of ArrayList:"

+ ArrayList.class.getClassLoader());

}

When executed the above method prints:

Class loader of this class:sun.misc.Launcher$AppClassLoader@18b4aac2

Class loader of Logging:sun.misc.Launcher$ExtClassLoader@3caeaf62

Class loader of ArrayList:null

As we can see, there are three different class loaders here; application, extension, and bootstrap (displayed as null).

The application class loader loads the class where the example method is contained. **An application or system class loader loads our own files in the classpath.**

Next, the extension one loads the Logging class. **Extension class loaders load classes that are an extension of the standard core Java classes.**

Finally, the bootstrap one loads the ArrayList class. **A bootstrap or primordial class loader is the parent of all the others.**

However, we can see that the last out, for the ArrayList it displays null in the output. **This is because the bootstrap class loader is written in native code, not Java – so it doesn't show up as a Java class.** Due to this reason, the behavior of the bootstrap class loader will differ across JVMs

Let's now discuss more in detail about each of these class loaders.

## 2.1. Bootstrap Class Loader

Java classes are loaded by an instance of *java.lang.ClassLoader*. However, class loaders are classes themselves. Hence, the question is, who loads the *java.lang.ClassLoader* itself*?*

This is where the bootstrap or primordial class loader comes into the picture.

It's mainly responsible for loading JDK internal classes, typically *rt.jar* and other core libraries located in *$JAVA\_HOME/jre/lib directory*. Additionally, **Bootstrap class loader serves as a parent of all the other *ClassLoader* instances**.

**This bootstrap class loader is part of the core JVM and is written in native code** as pointed out in the above example. Different platforms might have different implementations of this particular class loader.

## 2.2. Extension Class Loader

The **extension class loader is a child of the bootstrap class loader and takes care of loading the extensions of the standard core Java classes** so that it's available to all applications running on the platform.

Extension class loader loads from the JDK extensions directory, usually *$JAVA\_HOME/lib/ext* directory or any other directory mentioned in the *java.ext.dirs* system property.

## 2.3. System Class Loader

The system or application class loader, on the other hand, takes care of loading all the application level classes into the JVM. **It loads files found in the classpath environment variable, *-classpath* or *-cp* command line option**. Also, it's a child of Extensions classloader.

## 3. How Do Class Loaders Work?

Class loaders are part of the Java Runtime Environment. When the JVM requests a class, the class loader tries to locate the class and load the class definition into the runtime using the fully qualified class name.

The ***java.lang.ClassLoader.loadClass()* method is responsible for loading the class definition into runtime**. It tries to load the class based on a fully qualified name.

If the class isn't already loaded, it delegates the request to the parent class loader. This process happens recursively.

Eventually, if the parent class loader doesn’t find the class, then the child class will call *java.net.URLClassLoader.findClass()* method to look for classes in the file system itself.

If the last child class loader isn't able to load the class either, it throws [*java.lang.NoClassDefFoundError* or *java.lang.ClassNotFoundException.*](https://www.baeldung.com/java-classnotfoundexception-and-noclassdeffounderror)

Let's look at an example of output when ClassNotFoundException is thrown.

java.lang.ClassNotFoundException: com.baeldung.classloader.SampleClassLoader

at java.net.URLClassLoader.findClass(URLClassLoader.java:381)

at java.lang.ClassLoader.loadClass(ClassLoader.java:424)

at java.lang.ClassLoader.loadClass(ClassLoader.java:357)

at java.lang.Class.forName0(Native Method)

at java.lang.Class.forName(Class.java:348)

If we go through the sequence of events right from calling *java.lang.Class.forName()*, we can understand that it first tries to load the class through parent class loader and then *java.net.URLClassLoader.findClass()* to look for the class itself.

When it still doesn't find the class, it throws a *ClassNotFoundException.*

There are three important features of class loaders.

**3.1. Delegation Model**

Class loaders follow the delegation model where **on request to find a class or resource, a *ClassLoader* instance will delegate the search of the class or resource to the parent class loader**.

Let's say we have a request to load an application class into the JVM. The system class loader first delegates the loading of that class to its parent extension class loader which in turn delegates it to the bootstrap class loader.

Only if the bootstrap and then the extension class loader is unsuccessful in loading the class, the system class loader tries to load the class itself.

**3.2. Unique Classes**

As a consequence of the delegation model, it's easy to ensure **unique classes as we always try to delegate upwards**.

If the parent class loader isn't able to find the class, only then the current instance would attempt to do so itself.

**3.3. Visibility**

In addition, **children class loaders are visible to classes loaded by its parent class loaders**.

For instance, classes loaded by the system class loader have visibility into classes loaded by the extension and Bootstrap class loaders but not vice-versa.

To illustrate this, if Class A is loaded by an application class loader and class B is loaded by the extensions class loader, then both A and B classes are visible as far as other classes loaded by Application class loader are concerned.

Class B, nonetheless, is the only class visible as far as other classes loaded by the extension class loader are concerned.

## 4. Custom ClassLoader

The built-in class loader would suffice in most of the cases where the files are already in the file system.

However, in scenarios where we need to load classes out of the local hard drive or a network, we may need to make use of custom class loaders.

In this section, we'll cover some other uses cases for custom class loaders and we'll demonstrate how to create one.

**4.1. Custom Class Loaders Use-Cases**

Custom class loaders are helpful for more than just loading the class during runtime, a few use cases might include:

1. Helping in modifying the existing bytecode, e.g. weaving agents
2. Creating classes dynamically suited to the user's needs. e.g in JDBC, switching between different driver implementations is done through dynamic class loading.
3. Implementing a class versioning mechanism while loading different bytecodes for classes with same names and packages. This can be done either through URL class loader (load jars via URLs) or custom class loaders.

There are more concrete examples where custom class loaders might come in handy.

**Browsers, for instance, use a custom class loader to load executable content from a website.** A browser can load applets from different web pages using separate class loaders. The applet viewer which is used to run applets contains a *ClassLoader* that accesses a website on a remote server instead of looking in the local file system.

And then loads the raw bytecode files via HTTP, and turns them into classes inside the JVM. Even if these **applets have the same name, they are considered as different components if loaded by different class loaders**.

Now that we understand why custom class loaders are relevant, let's implement a subclass of *ClassLoader* to extend and summarise the functionality of how the JVM loads classes.

**4.2. Creating Our Custom Class Loader**

For illustration purposes, let's say we need to load classes from a file using a custom class loader.

**We need to extend the *ClassLoader* class and override the *findClass()* method:**

**public** **class** **CustomClassLoader** **extends** **ClassLoader** {

@Override

**public** Class **findClass**(String name) **throws** ClassNotFoundException {

**byte**[] b = loadClassFromFile(name);

**return** defineClass(name, b, 0, b.length);

}

**private** **byte**[] loadClassFromFile(String fileName) {

InputStream inputStream = getClass().getClassLoader().getResourceAsStream(

fileName.replace('.', File.separatorChar) + ".class");

**byte**[] buffer;

ByteArrayOutputStream byteStream = **new** ByteArrayOutputStream();

**int** nextValue = 0;

**try** {

**while** ( (nextValue = inputStream.read()) != -1 ) {

byteStream.write(nextValue);

}

} **catch** (IOException e) {

e.printStackTrace();

}

buffer = byteStream.toByteArray();

**return** buffer;

}

}

In the above example, we defined a custom class loader that extends the default class loader and loads a byte array from the specified file.

**5. Understanding *java.lang.ClassLoader***

Let's discuss a few essential methods from the *java.lang.ClassLoader* class to get a clearer picture of how it works.

**5.1. The *loadClass()* Method**

**public** Class<?> loadClass(String name, **boolean** resolve) **throws** ClassNotFoundException {

This method is responsible for loading the class given a name parameter. The name parameter refers to the fully qualified class name.

The Java Virtual Machine invokes *loadClass()* method to resolve class references setting resolve to *true*. However, it isn't always necessary to resolve a class. **If we only need to determine if the class exists or not, then resolve parameter is set to *false*.**

This method serves as an entry point for the class loader.

We can try to understand the internal working of the *loadClass()* method from the source code of *java.lang.ClassLoader:*

**protected** Class<?> loadClass(String name, **boolean** resolve)

**throws** ClassNotFoundException {

**synchronized** (getClassLoadingLock(name)) {

// First, check if the class has already been loaded

Class<?> c = findLoadedClass(name);

**if** (c == **null**) {

**long** t0 = System.nanoTime();

**try** {

**if** (parent != **null**) {

c = parent.loadClass(name, **false**);

} **else** {

c = findBootstrapClassOrNull(name);

}

} **catch** (ClassNotFoundException e) {

// ClassNotFoundException thrown if class not found

// from the non-null parent class loader

}

**if** (c == **null**) {

// If still not found, then invoke findClass in order

// to find the class.

c = findClass(name);

}

}

**if** (resolve) {

resolveClass(c);

}

**return** c;

}

}

The default implementation of the method searches for classes in the following order:

1. Invokes the *findLoadedClass(String)* method to see if the class is already loaded.
2. Invokes the *loadClass(String)* method on the parent class loader.
3. Invoke the *findClass(String)* method to find the class.

**5.2. The *defineClass()* Method**

**protected** **final** Class<?> defineClass(

String name, **byte**[] b, **int** off, **int** len) **throws** ClassFormatError

This method is responsible for the conversion of an array of bytes into an instance of a class. And before we use the class, we need to resolve it.

In case data didn't contain a valid class, it throws a *ClassFormatError.*

Also, we can't override this method since it's marked as final.

**5.3. The *findClass()* Method**

**protected** Class<?> findClass(

String name) **throws** ClassNotFoundException

This method finds the class with the fully qualified name as a parameter. We need to override this method in custom class loader implementations that follow the delegation model for loading classes.

Also, *loadClass()* invokes this method if the parent class loader couldn't find the requested class.

The default implementation throws a *ClassNotFoundException* if no parent of the class loader finds the class.

**5.4. The *getParent()* Method**

**public** **final** ClassLoader **getParent**()

This method returns the parent class loader for delegation.

Some implementations like the one seen before in Section 2. use *null* to represent the bootstrap class loader.

**5.5. The *getResource()* Method**

**public** URL **getResource**(String name)

This method tries to find a resource with the given name.

It will first delegate to the parent class loader for the resource. **If the parent is *null*, the path of the class loader built into the virtual machine is searched.**

If that fails, then the method will invoke *findResource(String)* to find the resource. The resource name specified as an input can be relative or absolute to the classpath.

It returns an URL object for reading the resource, or null if the resource could not be found or if the invoker doesn't have adequate privileges to return the resource.

It's important to note that Java loads resources from the classpath.

Finally, **resource loading in Java is considered location-independent** as it doesn't matter where the code is running as long as the environment is set to find the resources.

**6. Context Classloaders**

In general, context class loaders provide an alternative method to the class-loading delegation scheme introduced in J2SE.

Like we've learned before, **classloaders in a JVM follow a hierarchical model such that every class loader has a single parent with the exception of the bootstrap class loader.**

However, sometimes when JVM core classes need to dynamically load classes or resources provided by application developers, we might encounter a problem.

For example, in JNDI the core functionality is implemented by bootstrap classes in *rt.jar.* But these JNDI classes may load JNDI providers implemented by independent vendors (deployed in the application classpath). This scenario calls for the bootstrap class loader (parent class loader) to load a class visible to application loader (child class loader).

**J2SE delegation doesn't work here and to get around this problem, we need to find alternative ways of class loading. And it can be achieved using thread context loaders.**

The *java.lang.Thread* class has a method ***getContextClassLoader()* that returns the *ContextClassLoader* for the particular thread**. The *ContextClassLoader* is provided by the creator of the thread when loading resources and classes.

If the value isn't set, then it defaults to the class loader context of the parent thread.

**7. Conclusion**

Class loaders are essential to execute a Java program. We've provided a good introduction as part of this article.

We talked about different types of class loaders namely – Bootstrap, Extensions and System class loaders. Bootstrap serves as a parent for all of them and is responsible for loading the JDK internal classes. Extensions and system, on the other hand, loads classes from the Java extensions directory and classpath respectively.

points -:

**Delegation** - Each classloader first delegates loading of class to it's parent (goes all the way up the hierarchy). If parent is not able to load the class then class is tried to be loaded by it's child. If it cannot be loaded by any of the classloaders ClassNotFoundException exception is throws.

**Visibility** - Each classloader knows about the classes that it's parents have loaded. However it does not work the other way around. Parents will not know the classes loaded by their child. This brings us to the 3rd points.

**Uniqueness** - Each class is loaded exactly once. Since each child delegates class loading to it's parent and know the classes it's parents have loaded, it will try to load classes only when it is not loaded by its parent.Now these are ofcource default behavior of classloaders that already exist. However you can write your own class loaders and break it (not recommended though).

**Classloading in Java:**

Java has 3 main classloaders that are used to load classes at runtime -

--1.Bootstrap ClassLoader (Also called Primordial classLoader)

--2.Extension ClassLoader

--3.Application ClassLoader

In that order. So Bootstrap is parent of Extension and Extension is parent of Application classloader. Each of these classlaoders load classes from a predefined location.

Flow:

--1.Bootstrap ClassLoader- is the topmost level classloader. It does not have any parent. This classloader is a native implementation . This class loader is responsible of loading all standard JDK classes. It does this from path - <JRE>/lib/rt.jar. Since this is native implementation it does not refer to ClassLoader class.

---2.Extension ClassLoader -is direct child of Bootstrap classLoader. When this classloader tries to load a class it first delegates it to it's parent - Bootstrap ClassLoader. If parent is unsuccessful then Extension ClassLoader will try to load classes from path <JRE>/lib/ext or from path specified in java.ext.dirs system variable. In JVM this is implemented by - sun.misc.Launcher$ExtClassLoader

---3.Application classloader -is child of Extension classloader. Execution sequence remains same. When a class is loaded from this classloader it delegates to it's parent Extension which in turn delegates it to it's parent Bootstrap. If parents are unsuccessful in loading classes then Application classloaded will try to load class from the classpath - you can give it with arguments -classpath or -cp or specify it in manifest file of jar. In JVM this is implemented by sun.misc.Launcher$AppClassLoader

If application classloader is not able to load the class then it throws ClassNotFoundException. When JVM loads this is the order in which classloaders execute and load classes.

Principles of functionality of a Java ClassLoader: Principles of functionality are the set of rules or features on which a Java ClassLoader works. There are three principles of functionality, they are:

**A. Delegation Model:** The Java Virtual Machine and the Java ClassLoader use an algorithm called the Delegation Hierarchy Algorithm to Load the classes into the Java file.

The ClassLoader works based on a set of operations given by the delegation model. They are:

ClassLoader always follows the Delegation Hierarchy Principle.Whenever JVM comes across a class, it checks whether that class is already loaded or not.If the Class is already loaded in the method area then the JVM proceeds with execution.If the class is not present in the method area then the JVM asks the Java ClassLoader Sub-System to load that particular class, then ClassLoader sub-system hands over the control to Application ClassLoader.

Application ClassLoader then delegates the request to Extension ClassLoader and the Extension ClassLoader in turn delegates the request to Bootstrap ClassLoader.

Bootstrap ClassLoader will search in the Bootstrap classpath(JDK/JRE/LIB). If the class is available then it is loaded, if not the request is delegated to Extension ClassLoader.

Extension ClassLoader searches for the class in the Extension Classpath(JDK/JRE/LIB/EXT). If the class is available then it is loaded, if not the request is delegated to the Application ClassLoader.

Application ClassLoader searches for the class in the Application Classpath. If the class is available then it is loaded, if not then a ClassNotFoundException exception is generated.

**B.Visibility Principle:** The Visibility Principle states that a class loaded by a parent ClassLoader is visible to the child ClassLoaders but a class loaded by a child ClassLoader is not visible to the parent ClassLoaders. Suppose a class GEEKS.class has been loaded by the Extension ClassLoader, then that class is only visible to the Extension ClassLoader and Application ClassLoader but not to the Bootstrap ClassLoader. If that class is again tried to load using Bootstrap ClassLoader it gives an exception java.lang.ClassNotFoundException.

**C.Uniqueness Property:** The Uniquesness Property ensures that the classes are unique and there is no repetition of classes. This also ensures that the classes loaded by parent classloaders are not loaded by the child classloaders. If the parent class loader isn’t able to find the class, only then the current instance would attempt to do so itself.

## ClassNotFoundException vs. NoClassDefFoundError

## **ClassNotFoundException**

ClassNotFoundException is a runtime exception that is thrown when an application tries to load a class at runtime using the **Class.forName()** or **loadClass()** or **findSystemClass()** methods ,and the class with specified name are not found in the classpath. For example, you may have come across this exception when you try to connect to MySQL or Oracle databases and you have not updated the classpath with required JAR files. Most of the time, this exception occurs when you try to run an application without updating the classpath with required JAR files.

## ****NoClassDefFoundError****

NoClassDefFoundError is an error that is thrown when the Java Runtime System tries to load the definition of a class, and that class definition is no longer available. The required class definition was present at compile time, but it was missing at runtime.

|  |  |
| --- | --- |
| **ClassNotFoundException** | **NoClassDefFoundError** |
| It is an exception. It is of type java.lang.Exception. | It is an error. It is of type java.lang.Error. |
| It occurs when an application tries to load a class at run time which is not updated in the classpath. | It occurs when java runtime system doesn’t find a class definition, which is present at compile time, but missing at run time. |
| It is thrown by the application itself. It is thrown by the methods like Class.forName(), loadClass() and findSystemClass(). | It is thrown by the Java Runtime System. |
| It occurs when classpath is not updated with required JAR files. | It occurs when required class definition is missing at runtime. |

## Fail-Safe Iterator vs Fail-Fast Iterator

## ****1. Introduction****

In this article, we'll introduce the concept of Fail-Fast and Fail-Safe Iterators.

**Fail-Fast systems abort operation as-fast-as-possible exposing failures immediately and stopping the whole operation.**

Whereas, **Fail-Safe systems don't abort an operation in the case of a failure. Such systems try to avoid raising failures as much as possible.**

## ****2. Fail-Fast**** Iterators

Fail-fast iterators in Java don't play along when the underlying collection gets modified.

Collections maintain an internal counter called modCount. Each time an item is added or removed from the Collection, this counter gets incremented.

When iterating, on each next() call, the current value of modCount gets compared with the initial value. If there's a mismatch, it throws ConcurrentModificationException which aborts the entire operation.

**Default iterators for Collections from java.util package such as ArrayList, HashMap, etc. are Fail-Fast.**

ArrayList<Integer> numbers = // ...

Iterator<Integer> iterator = numbers.iterator();

**while** (iterator.hasNext()) {

Integer number = iterator.next();

numbers.add(50);

}

In the code snippet above, the ConcurrentModificationException gets thrown at the beginning of a next iteration cycle after the modification was performed.

The Fail-Fast behavior isn't guaranteed to happen in all scenarios as it's impossible to predict behavior in case of concurrent modifications. **These iterators throw ConcurrentModificationException on a best effort basis**.

If during iteration over a Collection, **an item is removed using Iterator‘s remove() method, that's entirely safe and doesn't throw an exception**.

However, if the Collection‘s remove() method is used for removing an element, it throws an exception:

ArrayList<Integer> numbers = // ...

Iterator<Integer> iterator = numbers.iterator();

**while** (iterator.hasNext()) {

**if** (iterator.next() == 30) {

iterator.remove(); // ok!

}

}

iterator = numbers.iterator();

**while** (iterator.hasNext()) {

**if** (iterator.next() == 40) {

numbers.remove(2); // exception

}

}

## ****3. Fail-Safe Iterators****

Fail-Safe iterators favor lack of failures over the inconvenience of exception handling.

Those iterators create a clone of the actual Collection and iterate over it. If any modification happens after the iterator is created, the copy still remains untouched. Hence, these Iterators continue looping over the Collection even if it's modified.

However, it's important to remember that there's no such thing as a truly Fail-Safe iterator. The correct term is Weakly Consistent.

That means, **if a** **Collection is modified while being iterated over, what the Iterator sees is weakly guaranteed**. This behavior may be different for different Collections and is documented in Javadocs of each such Collection.

The Fail-Safe Iterators have a few disadvantages, though. One disadvantage is that the **Iterator isn't guaranteed to return updated data from the Collection**, as it's working on the clone instead of the actual Collection.

Another disadvantage is the overhead of creating a copy of the Collection, both regarding time and memory.

**Iterators on Collections from java.util.concurrent package such as ConcurrentHashMap, CopyOnWriteArrayList, etc. are Fail-Safe in nature.**

ConcurrentHashMap<String, Integer> map = **new** ConcurrentHashMap<>();

map.put("First", 10);

map.put("Second", 20);

map.put("Third", 30);

map.put("Fourth", 40);

Iterator<String> iterator = map.keySet().iterator();

**while** (iterator.hasNext()) {

String key = iterator.next();

map.put("Fifth", 50);

}

In the code snippet above, we're using Fail-Safe Iterator. Hence, even though a new element is added to the Collection during the iteration, it doesn't throw an exception.

The default iteratorfor the ConcurrentHashMap is weakly consistent. This means that this Iterator can tolerate concurrent modification, traverses elements as they existed when Iterator was constructed and may (but isn't guaranteed to) reflect modifications to the Collection after the construction of the Iterator.

Hence, in the code snippet above, the iteration loops five times, which means **it does detect the newly added element to the Collection.**

## Experimental Garbage Collectors in the JVM

**1. Introduction**

In this tutorial, we'll cover the basic problems with [Java memory management](https://www.baeldung.com/jvm-garbage-collectors) and the need to constantly find better ways to achieve it. This will primarily cover the new experimental garbage collector introduced in Java called Shenandoah and how it compares against other garbage collectors.

**2. Understanding Challenges in Garbage Collection**

A garbage collector is a form of automatic memory management where a runtime like JVM manages allocation and reclamation of memory for the user programs running on it. There are several algorithms to implement a garbage collector. These include reference counting, mark-sweep, mark-compact, and copying.

**2.1. Considerations for a Garbage Collector**

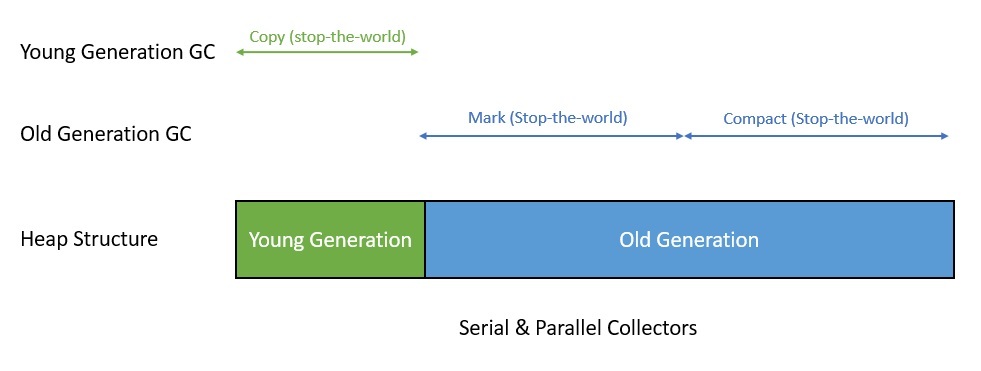
Depending upon the algorithm we use for garbage collection, it can **either run while the user program is suspended or run concurrently with the user program**. The former achieves higher throughput at the cost of high latency due to long pauses, also known as stop-the-world pauses. The latter aims for better latency but compromises on throughput.

In fact, most modern-day collectors use a hybrid strategy, where they apply both stop-the-world and concurrent approaches. It usually **works by dividing the heap space into young and old generations**. Generational collectors then use the stop-the-world collection in the young generation and concurrent collection in the old generation, possibly in increments to reduce pauses.

Nevertheless, the **sweet spot really is to find a garbage collector that runs with minimal pauses and provides high throughput** — all this with a predictable behavior on heap size that can vary from small to very large! This is a constant struggle that has kept the pace of innovation in the Java garbage collection alive since the early days.

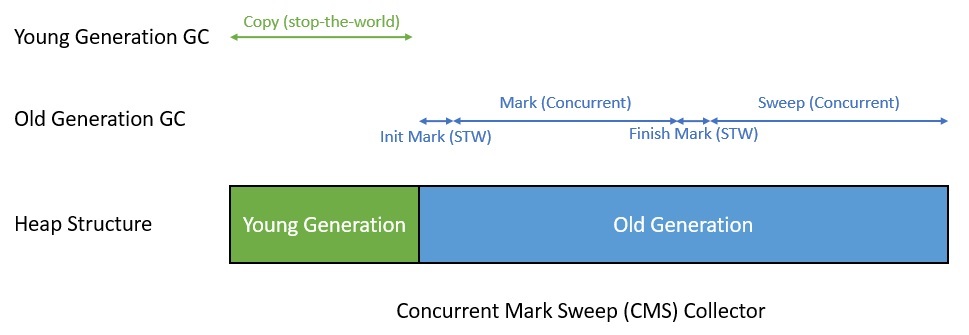
**2.2. Existing Garbage Collectors in Java**

Some of the **traditional** [**garbage collectors**](https://www.baeldung.com/jvm-garbage-collectors) **include serial and parallel collectors**. They are generational collectors and use copying in the young and mark-compact in the old generation:

[](https://www.baeldung.com/wp-content/uploads/2021/01/Garbage-Collector-Serial-Parallel-1.jpg)

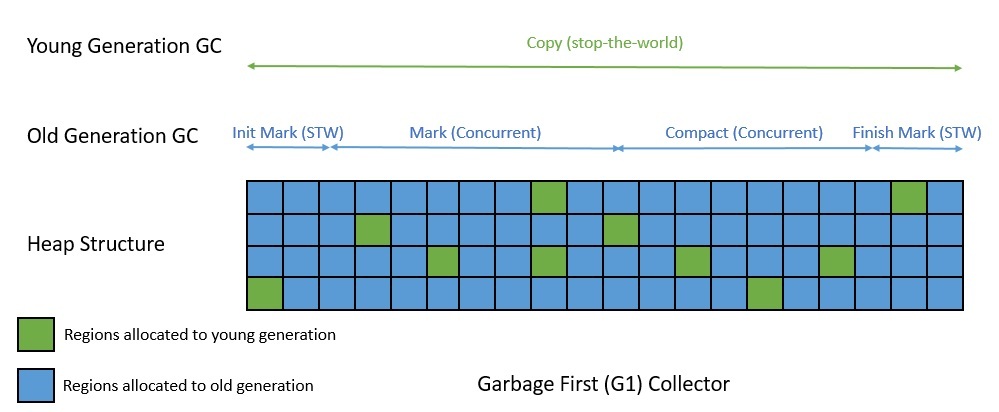
While providing good throughput, they **suffer from the problem of long stop-the-world pauses**.

The [Concurrent Mark Sweep (CMS) collector](https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html) introduced in Java 1.4 is a generational, concurrent, low-pause collector. It works with copying in the young generation and mark-sweep in the old generation:

[](https://www.baeldung.com/wp-content/uploads/2021/01/Garbage-Collector-CMS-1.jpg)

It tries to minimize the pause time by doing most of the work concurrently with the user program. Nevertheless, it **still has problems leading to unpredictable pauses**, requires more CPU time, and is not suitable for a heap larger than 4 GB in size.

As a long-term replacement for CMS, the [Garbage First (G1) collector](https://docs.oracle.com/javase/9/gctuning/garbage-first-garbage-collector.htm) was introduced in Java 7. G1 is a generational, parallel, concurrent, and incrementally compacting low-pause collector. It works with copying in the young generation and mark-compact in the old generation:

[](https://www.baeldung.com/wp-content/uploads/2021/01/Garbage-Collector-G1-1.jpg)

G1, however, is also a regionalized collector and structures the heap area into smaller regions. This gives it **the benefit of more predictable pauses**. Targeted for multiprocessor machines with a large amount of memory, G1 **is also not free from the pauses**.

So, the race to find a better garbage collector continues, especially one that reduces the pause time further. There's a series of experimental collectors that JVM has introduced lately, like Z, Epsilon, and Shenandoah. Apart from that, G1 continues to get more improvements.

The objective is really to get as close as possible to a pauseless Java!

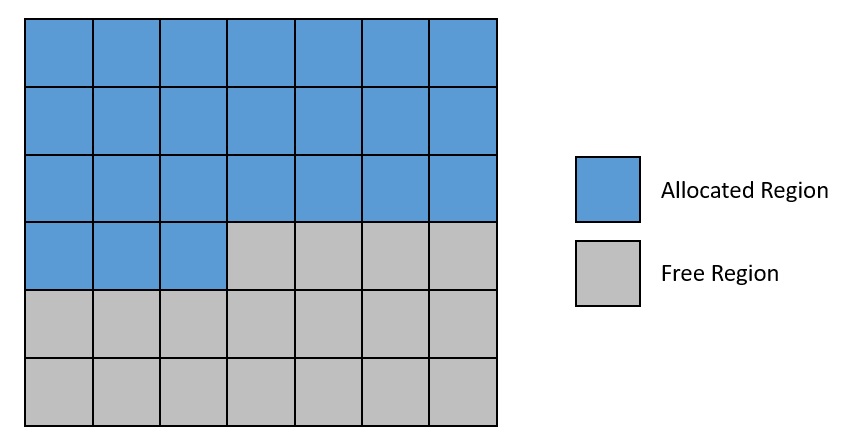
**3. Shenandoah Garbage Collector**

[Shenandoah](https://wiki.openjdk.java.net/display/shenandoah/Main) is **an experimental collector that has been introduced in Java 12 and is being positioned as a latency specialist**. It tries to reduce pause times by doing more of its garbage collection work concurrently with the user program.

For instance, Shenendoah attempts to perform object relocation and compaction concurrently. This essentially means that the pause time in Shenandoah is no longer directly proportional to the heap size. Hence, it can **provide consistent low-pause behavior, irrespective of the heap size**.

**3.1. Heap Structure**

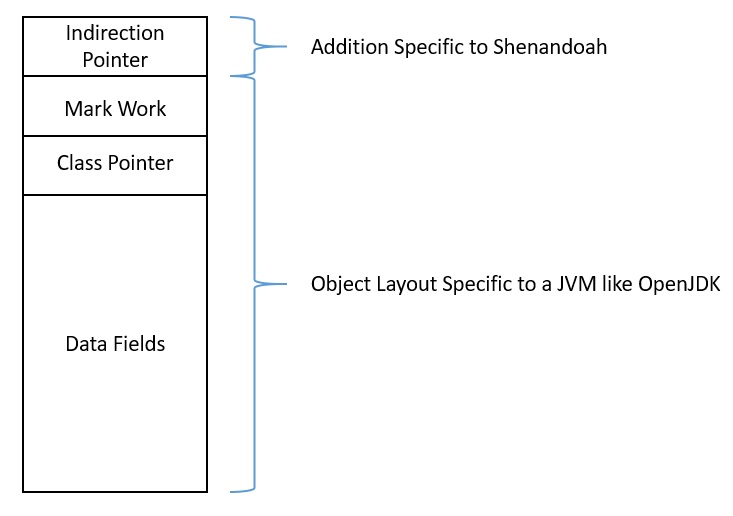
Shenandoah, like G1, is a regionalized collector. This means that it **divides the heap area into a collection of equal-sized regions**. A region is basically a unit of memory allocation or reclamation:

[](https://www.baeldung.com/wp-content/uploads/2021/01/Garbage-Collector-Shenandoah-Heap-Structure.jpg)

But, unlike G1 and other generational collectors, Shenandoah doesn't divide the heap area into generations. Therefore, it has to mark most of the live objects every cycle, which generational collectors can avoid.

**3.2. Object Layout**

In Java, objects in memory don't only include data fields — they carry some extra information as well. This extra information consists of the header, which contains a pointer to the object's class, and the mark word. There are several uses for the mark word, like forwarding pointers, age bits, locking, and hashing:

[](https://www.baeldung.com/wp-content/uploads/2021/01/Garbage-Collector-Shenandoah-Object-Layout.jpg)

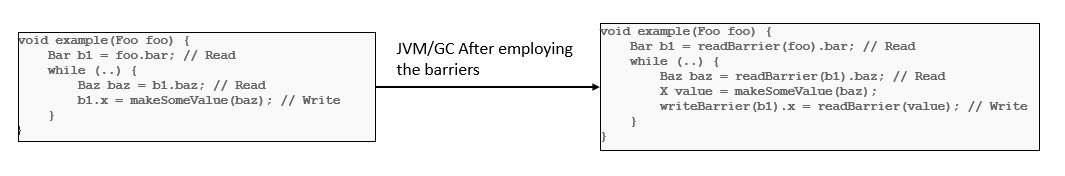
Shenandoah **adds an additional word to this object layout**. This serves as the indirection pointer and allows Shenandoah to move objects without updating all of the references to them. This is **also known as the Brooks pointer**.

**3.3. Barriers**

Performing a collection cycle in the stop-the-world mode is simpler, but the complexity just shoots up when we do that concurrently with the user program. It presents different challenges to the collection phases like concurrent marking and compaction.

The solution lies in **intercepting all heap accesses through what we call barriers**. Shenandoah and other concurrent collectors like G1 make use of barriers to ensure heap consistency. However, barriers are costly operations and generally tend to reduce the throughput of a collector.

For instance, the read and write operations to an object may be intercepted by the collector using barriers:

[](https://www.baeldung.com/wp-content/uploads/2021/01/Garbage-Collector-Barriers.jpg)

Shenandoah **makes use of multiple barriers in different phases, like the SATB barrier, read barrier, and write barrier**. We'll see where these are used in later sections.

**3.4. Modes, Heuristics, and Failure Modes**

Modes **define the way Shenandoah runs**, like which barriers it uses, and they also define its performance characteristics. There are three modes available: normal/SATB, iu, and passive. The normal/SATB mode is the default.

Heuristics **determine when a collection should start and which regions it should include**. These include adaptive, static, compact, and aggressive, with adaptive as the default heuristic. For instance, it may choose to select regions with 60 percent or more garbage and start a collection cycle when 75 percent of regions have been allocated.

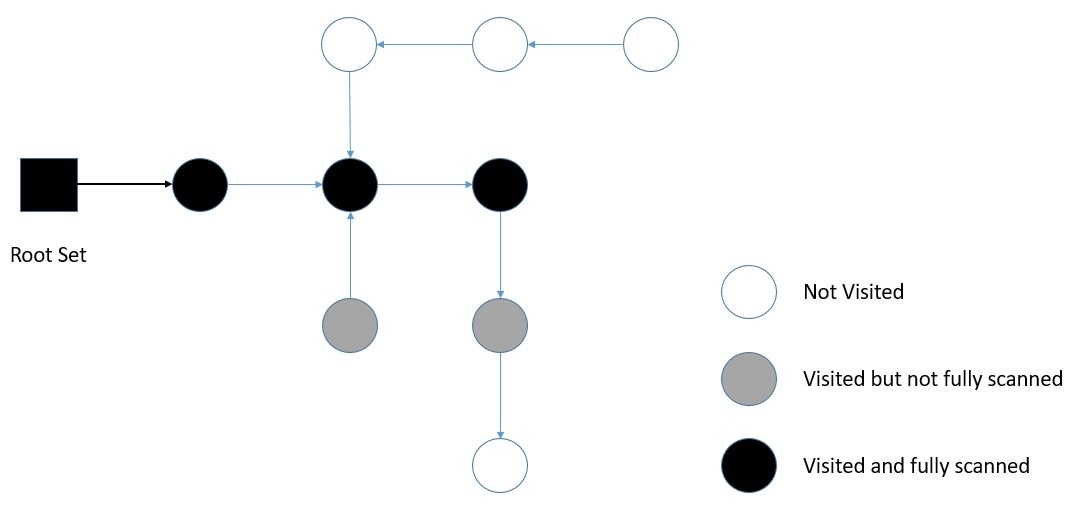
Shenandoah needs to collect heap faster than the user program that allocates it. But, **at times, it may fall behind, leading to one of the failure modes**. These failure modes include pacing, degenerated collection, and in the worst case, a full collection.

**4. Shenandoah Collection Phases**

Shenandoah's collection cycle consists primarily of three phases: mark, evacuate, and update references. Although most of the work in these phases happens concurrently with the user program, there are still small parts that must happen in a stop-the-world mode.

**4.1. Marking**

Marking is **the process of identifying all objects in the heap or parts of it that are unreachable**. We can do this by starting from the root objects and traversing the object graph to find reachable objects. While traversing, we also assign each object one of three colors: white, grey, or black:

[](https://www.baeldung.com/wp-content/uploads/2021/01/Garbage-Collector-Shenandoah-Marking-1.jpg)

Marking in the stop-the-world mode is simpler, but it gets complicated in concurrent mode. This is because the user program concurrently mutates the object graph while marking is in progress. Shenandoah solves this by **using the Snapshot At the Beginning (SATB) algorithm**.

This means that any object that was alive at the beginning of the marking or that has been allocated since the beginning of marking is considered live. Shenandoah **makes use of the SATB barrier** to maintain the SATB view of the heap.

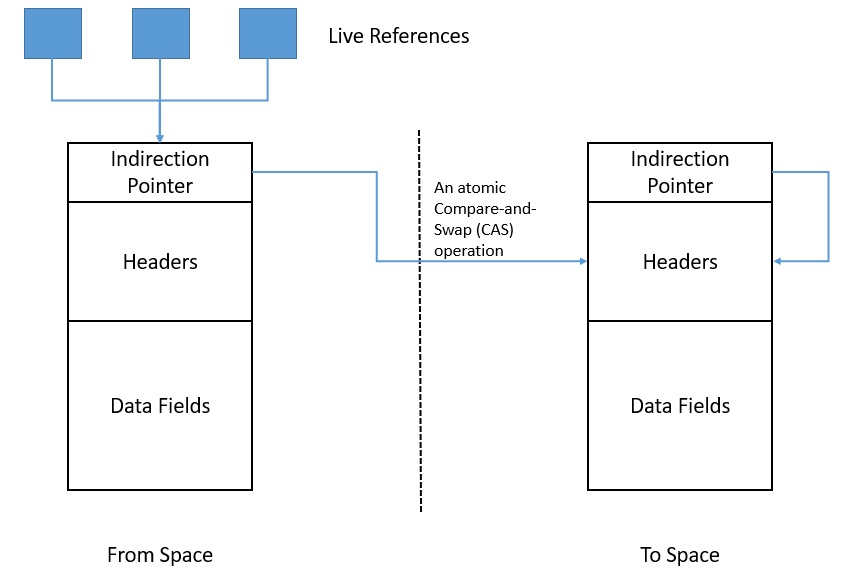
While **most of the marking is done concurrently**, there are still some parts that are done in stop-the-world mode. The parts that happen in the stop-the-world mode are the init-mark to scan the root set and the final-mark to drain all pending queues and re-scan the root set. The final-mark also prepares the collection set that indicates the regions to be evacuated.

**4.2. Cleanup and Evacuation**

Once the marking is complete, the garbage regions are ready to be reclaimed. The **garbage regions are the regions where no live objects are present**. The cleanup happens concurrently.

Now, the next step is to move the live objects in the collection set to other regions. This is done to reduce the fragmentation in memory allocation and, hence, is also known as compact. Evacuation or compacting happens entirely concurrently.

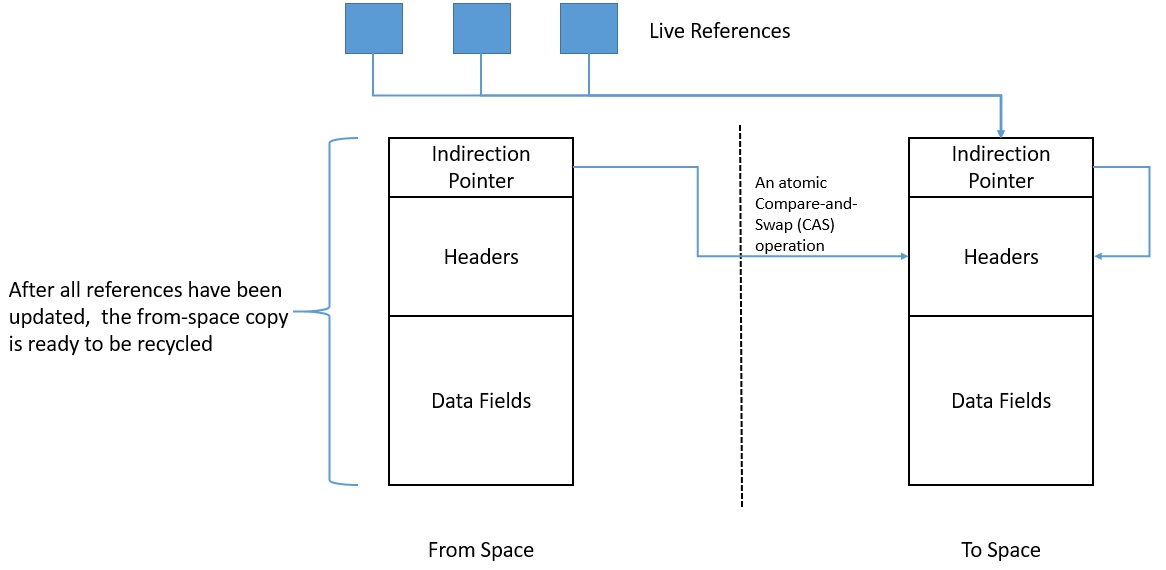
Now, this is where Shenandoah is different from other collectors. A concurrent relocation of objects is tricky as the user program continues to read and write them. Shenandoah manages to achieve this by performing **a compare-and-swap operation on the Brooks pointer** of an object to point to its to-space version:

[](https://www.baeldung.com/wp-content/uploads/2021/01/Garbage-Collector-Shenandoah-Barriers.jpg)

Further, Shenandoah **uses the read and write barriers to ensure that a strict “to-space” invariant is maintained** during the concurrent evacuation. What this means is that the read and write must happen from the to-space that is guaranteed to survive the evacuation.

**4.3. Reference Update**

This phase in the collection cycle is **to traverse through the heap and update the references to objects that were moved during the evacuation**:

[](https://www.baeldung.com/wp-content/uploads/2021/01/Garbage-Collector-Shenandoah-Update-Refs.jpg)

The update reference phase is, again, **mostly done concurrently**. There are brief periods of init-update-refs that initialize the update reference phase and final-update-refs that re-update the root set and recycle the regions from the collection set. Only these require the stop-the-world mode.

**5. Comparision With Other Experimental Collectors**

Shenandoah is not the only experimental garbage collector that has been introduced recently in Java. Others include Z and Epsilon. Let's understand how they compare against Shenandoah.

**5.1. Z Collector**

Introduced in Java 11, the [Z collector](https://www.baeldung.com/jvm-zgc-garbage-collector) is **a single-generation, low-latency collector designed for very large heap sizes** — we're talking multi-terabyte territory. The Z collector does most of its work concurrently with the user program and leverages the load barrier for heap references.

Further, the Z collector takes advantage of 64-bit pointers with a technique called pointer coloring. Here, the colored pointers store extra information about objects on the heap. The Z collector remaps objects using the extra information stored in the pointer to reduce memory fragmentation.

Broadly speaking, **the Z collector's goals are similar to those of Shenandoah**. They both aim to achieve low pause times that are not directly proportional to the heap size. However, **there are more tuning options available with Shenandoah than with the Z collector**.

**5.2. Epsilon Collector**

[Epsilon](https://www.baeldung.com/jvm-epsilon-gc-garbage-collector), also introduced in Java 11, has a very different approach to garbage collection. It's basically **a passive or “no-op” collector,** which means that it handles memory allocation but doesn't recycle it! So, when the heap runs out of memory, the JVM simply shuts down.

But why would we ever want to use a collector like that? Basically, any garbage collector has an indirect impact on the performance of the user program. It's very difficult to benchmark an application and understand the impact of garbage collection on it.

Epsilon serves exactly that purpose. It simply **removes the impact of a garbage collector** and lets us run the application in isolation. But, this expects us to have a very clear understanding of the memory requirements of our application. Consequently, we can achieve better performance from the application.

Clearly, **Epsilon has a very different goal from that of Shenandoah**.

**6. Conclusion**

In this article, we went through the basics of garbage collection in Java and the need to constantly improve it. We discussed in detail the most recent experimental collector introduced in Java — Shenandoah. We also went through how it fares against the other experimental collectors available in Java.

The pursuit of a universal garbage collector isn't going to be realized anytime soon! So, while G1 remains the default collector, these new additions provide us with options to use Java in low-latency situations. However, we shouldn't consider them as a drop-ship replacement of other high-throughput collectors.

**Stack Memory and Heap Space in Java**

## ****1. Introduction****

To run an application in an optimal way, JVM divides memory into stack and heap memory. **Whenever we declare new variables and objects, call new method, declare a String or perform similar operations, JVM designates memory to these operations from either Stack Memory or Heap Space.**

In this tutorial, we'll discuss these memory models. We'll enlist some key differences between them, how they are stored in RAM, the features they offer and where to use them.

## ****2. Stack Memory in Java****

**Stack Memory in Java is used for static memory allocation and the execution of a thread.** It contains primitive values that are specific to a method and references to objects that are in a heap, referred from the method.

Access to this memory is in Last-In-First-Out (LIFO) order. Whenever a new method is called, a new block on top of the stack is created which contains values specific to that method, like primitive variables and references to objects.

When the method finishes execution, it’s corresponding stack frame is flushed, the flow goes back to the calling method and space becomes available for the next method.

### ****2.1. Key Features of Stack Memory****

Apart from what we have discussed so far, following are some other features of stack memory:

* It grows and shrinks as new methods are called and returned respectively
* Variables inside stack exist only as long as the method that created them is running
* It's automatically allocated and deallocated when method finishes execution
* If this memory is full, Java throws java.lang.StackOverFlowError
* Access to this memory is fast when compared to heap memory
* This memory is threadsafe as each thread operates in its own stack

## ****3. Heap Space in Java****

**Heap space in Java is used for dynamic memory allocation for Java objects and JRE classes at the runtime**. New objects are always created in heap space and the references to this objects are stored in stack memory.

These objects have global access and can be accessed from anywhere in the application.

This memory model is further broken into smaller parts called generations, these are:

1. **Young Generation –** this is where all new objects are allocated and aged. A minor Garbage collection occurs when this fills up
2. **Old or Tenured Generation –** this is where long surviving objects are stored. When objects are stored in the Young Generation, a threshold for the object's age is set and when that threshold is reached, the object is moved to the old generation
3. **Permanent Generation –** this consists of JVM metadata for the runtime classes and application methods

These different portions are also discussed in this article – [Difference Between JVM, JRE, and JDK.](https://www.baeldung.com/jvm-vs-jre-vs-jdk)

We can always manipulate the size of heap memory as per our requirement. For more information, visit this [linked Baeldung article](https://www.baeldung.com/jvm-parameters).

### ****3.1. Key Features of Java Heap Memory****

Apart from what we have discussed so far, following are some other features of heap space:

* It's accessed via complex memory management techniques that include Young Generation, Old or Tenured Generation, and Permanent Generation
* If heap space is full, Java throws java.lang.OutOfMemoryError
* Access to this memory is relatively slower than stack memory
* This memory, in contrast to stack, isn't automatically deallocated. It needs Garbage Collector to free up unused objects so as to keep the efficiency of the memory usage
* Unlike stack, a heap isn't threadsafe and needs to be guarded by properly synchronizing the code

## ****4. Example****

Based on what we've learned so far, let's analyze a simple Java code and let's assess how memory is managed here:

**class** **Person** {

**int** id;

String name;

**public** **Person**(**int** id, String name) {

**this**.id = id;

**this**.name = name;

}

}

**public** **class** **PersonBuilder** {

**private** **static** Person **buildPerson**(**int** id, String name) {

**return** **new** Person(id, name);

}

**public** **static** **void** **main**(String[] args) {

**int** id = 23;

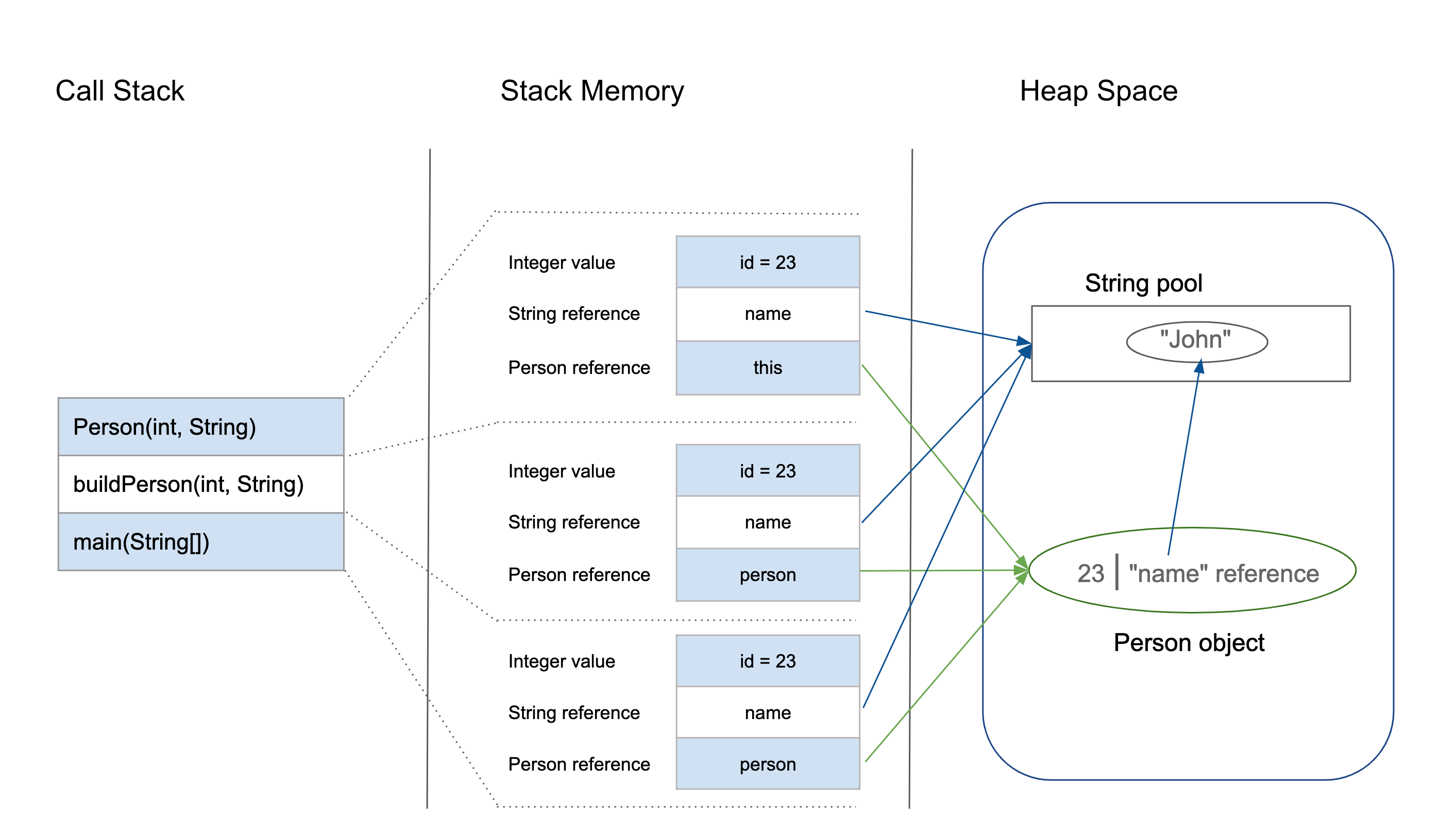
String name = "John";

Person person = **null**;

person = buildPerson(id, name);

}

}



Let's analyze this step-by-step:

1. Upon entering the main() method, a space in stack memory would be created to store primitives and references of this method
   * The primitive value of integer id will be stored directly in stack memory
   * The reference variable person of type Person will also be created in stack memory which will point to the actual object in the heap
2. The call to the parameterized constructor Person(int, String) from main() will allocate further memory on top of the previous stack. This will store:
   * The this object reference of the calling object in stack memory
   * The primitive value id in the stack memory
   * The reference variable of String argument name which will point to the actual string from string pool in heap memory
3. The main method is further calling the buildPerson() static method, for which further allocation will take place in stack memory on top of the previous one. This will again store variables in the manner described above.
4. However, for the newly created object person of type Person, all instance variables will be stored in heap memory.

This allocation is explained in this diagram:

## ****5. Summary****

Before we conclude this article, let's quickly summarize the differences between the Stack Memory and the Heap Space:

| **Parameter** | **Stack Memory** | **Heap Space** |
| --- | --- | --- |
| Application | Stack is used in parts, one at a time during execution of a thread | The entire application uses Heap space during runtime |
| Size | Stack has size limits depending upon OS and is usually smaller then Heap | There is no size limit on Heap |
| Storage | Stores only primitive variables and references to objects that are created in Heap Space | All the newly created objects are stored here |
| Order | It is accessed using Last-in First-out (LIFO) memory allocation system | This memory is accessed via complex memory management techniques that include Young Generation, Old or Tenured Generation, and Permanent Generation. |
| Life | Stack memory only exists as long as the current method is running | Heap space exists as long as the application runs |
| Efficiency | Comparatively much faster to allocate when compared to heap | Slower to allocate when compared to stack |
| Allocation/Deallocation | This Memory is automatically allocated and deallocated when a method is called and returned respectively | Heap space is allocated when new objects are created and deallocated by Gargabe Collector when they are no longer referenced |

## ****6. Conclusion****

Stack and heap are two ways in which Java allocates memory. In this article, we understood how they work and when to use them for developing better Java programs.

**Types of polymorphism in java- Runtime and Compile time polymorphism**

There are two types of polymorphism in java:  
1) **Static Polymorphism** also known as compile time polymorphism  
2) **Dynamic Polymorphism** also known as runtime polymorphism

## Compile time Polymorphism (or Static polymorphism)

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading is an example of compile time polymorphism.  
**Method Overloading**: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters.

## Runtime Polymorphism (or Dynamic polymorphism)

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, thats why it is called runtime polymorphism.

there are other characteristics in the Java programming language that exhibit polymorphism.

### ****Coercion****

Polymorphic coercion deals with implicit type conversion done by the compiler to prevent type errors. A typical example is seen in an integer and string concatenation:

String str = “string” + 2;

### ****Operator Overloading****

Operator or method overloading refers to a polymorphic characteristic of same symbol or operator having different meanings (forms) depending on the context.

For example, the plus symbol (+) can be used for mathematical addition as well as String concatenation. In either case, only context (i.e. argument types) determines the interpretation of the symbol:

String str = "2" + 2; //22

**int** sum = 2 + 2; //4

System.out.printf(" str = %s\n sum = %d\n", str, sum);

### ****Polymorphic Parameters****

Parametric polymorphism allows a name of a parameter or method in a class to be associated with different types. We have a typical example below where we define content as a String and later as an Integer:

**public** **class** **TextFile** **extends** **GenericFile** {

**private** String content;

**public** String **setContentDelimiter**() {

**int** content = 100;

**this**.content = **this**.content + content;

}

}

It's also important to note that **declaration of polymorphic parameters can lead to a problem known as** **variable hiding** where a local declaration of a parameter always overrides the global declaration of another parameter with the same name.

To solve this problem, it is often advisable to use global references such as this keyword to point to global variables within a local context.

**Covariant Return**

Before examining covariance from the return type's point of view, let's see what that means.

## 2. Covariance

**Covariance can be considered as a contract for how a subtype is accepted when only the supertype is defined.**

Let's consider a couple of basic examples of covariance:

List<? extends Number> integerList = **new** ArrayList<Integer>();

List<? extends Number> doubleList = **new** ArrayList<Double>();

**So** **covariance means, we can access specific elements defined via their supertype**. However, **we aren't allowed to put elements into a covariant system**, since the compiler would fail to determine the actual type of the generic structure.

## 3. The Covariant Return Type

The **covariant return type is – when we override a method – what allows the return type to be the subtype of the type of the overridden method**.

To put this into practice, let's take a simple Producer class with a produce() method. By default, it returns a String as an Object to provide flexibility for the child classes:

**public** **class** **Producer** {

**public** Object **produce**(String input) {

Object result = input.toLowerCase();

**return** result;

}

}

As a result of the Object as a return type, we can have a more concrete return type in the child class. That will be the covariant return type and will produce numbers from character sequences:

**public** **class** **IntegerProducer** **extends** **Producer** {

@Override

**public** Integer **produce**(String input) {

**return** Integer.parseInt(input);

}

}

## 4. The Usage of the Structure

The main idea behind the covariant return types is to support the [Liskov substitution](https://www.baeldung.com/solid-principles#l).

**A well-known scenario is the Object#clone method, which returns an Object by default. Whenever we override the clone() method, the facility of covariant return types allows us to have a more concrete return object than the Object itself.**

**Advantages:**

* It helps to avoid confusing type casts present in the class hierarchy and thus making the code readable, usable and maintainable.
* We get a liberty to have more specific return types when overriding methods.
* Help in preventing run-time ClassCastExceptions on returns

**Rules for Covariant Return Type in Java**

There are mainly three rules for covariant return types that should be kept in mind. They are as follows:

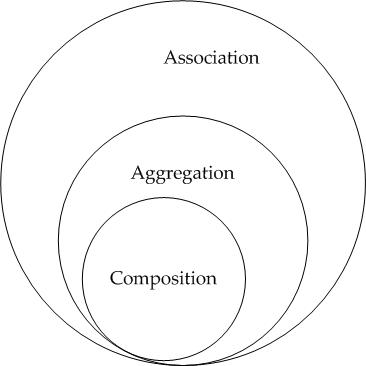
1. The return type of overriding method in the subclass should be either the same as the return type of [superclass or subclass](https://www.scientecheasy.com/2020/07/java-superclass-subclass.html/).

2. The return type of overriding method in the subclass should not be a parent of the parent method return type.

3. The covariant return type is applicable only for object types not for primitive types

# [**What is the difference between association, aggregation and composition?**](https://stackoverflow.com/questions/885937/what-is-the-difference-between-association-aggregation-and-composition)

**Aggregation and Composition are subsets of association** meaning they are specific cases of association.



**Composition Example** : Consider the example of a Car and an engine that is very specific to that car (meaning it cannot be used in any other car). This type of relationship between **Car** and **SpecificEngine** class is called Composition. An object of the Car class cannot exist without an object of SpecificEngine class and object of SpecificEngine has no significance without Car class. To put in simple words Car class solely "owns" the SpecificEngine class.

**Aggregation Example** : Now consider class **Car** and class **Wheel**. Car needs a Wheel object to function. Meaning the Car object owns the Wheel object but we cannot say the Wheel object has no significance without the Car Object. It can very well be used in a Bike, Truck or different Cars Object.

**Summing it up -**

To sum it up association is a very generic term used to represent when a class uses the functionalities provided by another class. We say it's composition if one parent class object owns another child class object and that child class object cannot meaningfully exist without the parent class object. If it can then it is called Aggregation.

1. Composition:

Since Engine is-part-of Car, the relationship between them is Composition. Here is how they are implemented between Java classes.

public class Car {

//final will make sure engine is initialized

private final Engine engine;

public Car(){

engine = new Engine();

}

}

class Engine {

private String type;

}

2. Aggregation:

Since Organization has Person as employees, the relationship between them is Aggregation. Here is how they look like in terms of Java classes

public class Organization {

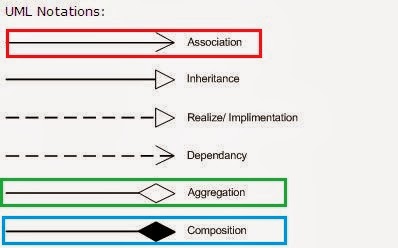
private List employees;

}

public class Person {

private String name;

}

[](https://1.bp.blogspot.com/-VL_9cjhwEE4/UvJN__IvaBI/AAAAAAAABCc/IkDmShgM-Yc/s1600/Association,+Composition+UML.JPG)

# **Difference between Inheritance and Composition in Java**

[Inheritance](https://www.geeksforgeeks.org/inheritance-in-c/)**:**  
When we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. In doing this, we can reuse the fields and methods of the existing class without having to write them ourself.

A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

Types of Inheritance are:

1. Single inheritance
2. Multi-level inheritance
3. Multiple inheritance
4. Hybrid inheritance
5. Hierarchical inheritance

**Example of Inheritance:**

[Composition](https://www.geeksforgeeks.org/association-composition-aggregation-java/)**:**  
The composition also provides code reusability but the difference here is we do not extend the class for this.

| S.NO | Inheritance | Composition |
| --- | --- | --- |
| 1. | In inheritance, we define the class which we are inheriting(super class) and most importantly it cannot be changed at runtime | Whereas in composition we only define a type which we want to use and which can hold its different implementation also it can change at runtime. Hence, Composition is much more flexible than Inheritance. |
| 2. | Here we can only extend one class, in other words more than one class can’t be extended as java do not support multiple inheritance. | Whereas composition allows to use functionality from different class. |
| 3. | In inheritance we need parent class in order to test child class. | Composition allows to test the implementation of the classes we are using independent of parent or child class. |
| 4. | Inheritance cannot extend final class. | Whereas composition allows code reuse even from final classes. |
| 5. | It is an **is-a** relationship. | While it is a **has-a** relationship. |

# **What would it be more secure to store sensitive data (such as a password, social security number, etc.**

Why Char[] array is more secure (store sensitive data) than String in Java?

there is one other good reason. With plain String you have much higher chances of **accidentally printing the password to logs**, monitors or some other insecure place. char[] is less vulnerable. (In contrast, if you use a mutable object like a character array, for example, to store the value, you can set it to blank once you are done with it with confidence that it will no longer be retained in memory.)

Consider this:

public static void main(String[] args) {

Object pw = "Password";

System.out.println("String: " + pw);

pw = "Password".toCharArray();

System.out.println("Array: " + pw);

}

String: Password

Array: [C@5829428e

These are all the reasons, one should choose a **char[]** array instead of **String** for a password.

**1.** Since Strings are immutable in Java, if you store the password as plain text it will be available in memory until the Garbage collector clears it, and since String is used in the String pool for reusability there is a pretty high chance that it will remain in memory for a long duration, which poses a security threat.

Since anyone who has access to the memory dump can find the password in clear text, that's another reason you should always use an encrypted password rather than plain text. Since Strings are immutable there is no way the contents of Strings can be changed because any change will produce a new String, while if you use a char[] you can still set all the elements as blank or zero. So storing a password in a character array clearly mitigates the security risk of stealing a password.

**2.** Java itself recommends using the getPassword() method of JPasswordField which returns a char[], instead of the deprecated getText() method which returns passwords in clear text stating security reasons. It's good to follow advice from the Java team and adhere to standards rather than going against them.

**3.** With String there is always a risk of printing plain text in a log file or console but if you use an Array you won't print contents of an array, but instead its memory location gets printed. Though not a real reason, it still makes sense.

# **Is Java “pass-by-reference” or “pass-by-value”?**

Java is always pass-by-value. Unfortunately, when we pass the value of an object, we are passing the reference to it.

There is no such thing as "pass-by-reference" in Java.

This is confusing to beginners.

The Java Spec says that everything in Java is pass-by-value. There is no such thing as "pass-by-reference" in Java.

The key to understanding this is that something like

Dog myDog;

is not a Dog; it's actually a pointer to a Dog. The use of the term "reference" in Java is very misleading and is what causes most of the confusion here. What they call "references" act/feel more like what we'd call "pointers" in most other languages.

What that means, is when you have

Dog myDog = new Dog("Rover");

foo(myDog);

you're essentially passing the address of the created Dog object to the foo method.

(I say essentially because Java pointers/references aren't direct addresses, but it's easiest to think of them that way.)

# **Provide some examples when a finally block won't be executed in Java?**

Yes, finally will be called after the execution of the try or catch code blocks.The only times finally won't be called are:

* If you invoke System.exit()
* If you invoke Runtime.getRuntime().halt(exitStatus)
* If the JVM crashes first
* If the JVM reaches an infinite loop (or some other non-interruptable, non-terminating statement) in the try or catch block
* If the OS forcibly terminates the JVM process; e.g., kill -9 <pid> on UNIX
* If the host system dies; e.g., power failure, hardware error, OS panic, et cetera
* If the finally block is going to be executed by a daemon thread and all other non-daemon threads exit before finally is called

public static void main(String[] args) { System.out.println(Test.test());}

public static int test() {

try { return 0; }

finally { System.out.println("finally trumps return."); }}

finally trumps return.

0

from the Java Language Specification.[14.20.2. Execution of try-finally and try-catch-finally](https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.20.2) **A try statement with a finally block is executed by first executing the try block. Then there is a choice:**

1. If execution of the try block completes normally, [...]
2. If execution of the try block completes abruptly because of a throw of a value *V*, [...]
3. **If execution of the try block completes abruptly for any other reason *R*, then the finally block is executed. Then there is a choice:**
   1. If the finally block completes normally, then the try statement completes abruptly for reason *R*.
   2. If the finally block completes abruptly for reason *S*, then the try statement completes abruptly for reason *S* (**and reason *R* is discarded**).

The specification for return actually makes this explicit:[JLS 14.17 The return Statement](https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.17)

1. ReturnStatement:
2. return Expression(opt) ;
3. A return statement with no Expression **attempts** to transfer control to the invoker of the method or constructor that contains it.
4. A return statement with an Expression **attempts** to transfer control to the invoker of the method that contains it; the value of the Expression becomes the value of the method invocation.
5. The preceding descriptions say "**attempts** *to transfer control*" rather than just "*transfers control*" because if there are any try statements within the method or constructor whose try blocks contain the return statement, then any finally clauses of those try statements will be executed, in order, innermost to outermost, before control is transferred to the invoker of the method or constructor. Abrupt completion of a finally clause can disrupt the transfer of control initiated by a return statement.

# [**What is the difference between a deep copy and a shallow copy?**](https://stackoverflow.com/questions/184710/what-is-the-difference-between-a-deep-copy-and-a-shallow-copy)

## ****Shallow Copy****

First let’s talk about the shallow copy. A shallow copy of an object copies the ‘main’ object, but doesn’t copy the inner objects. The ‘inner objects’ are shared between the original object and its copy.

**public class Person {**

**private Name name;**

**private Address address;**

**public Person(Person originalPerson) {**

**this.name = originalPerson.name;**

**this.address = originalPerson.address;**

**}**

**}**

## ****Deep Copy****

Unlike the shallow copy, a deep copy is a **fully independent copy of an object**. If we copied our Person object, we would copy the entire object structure.

**public class Person {**

**private Name name;**

**private Address address;**

**public Person(Person otherPerson) {**

**this.name = new Name(otherPerson.name);**

**this.address = new Address(otherPerson.address);**

**}**

**}**

# **How to Clone Objects in Java?**

So we can define Cloning as “create a copy of object “ .I think now we are somehow clear about cloning but there is more to it depending on how we are doing this copy, we can divide cloning into two types.

Shallow Copy

Deep Copy

Before going into the deep of shallow and deep copy we need to understand how we achieve cloning in java.

In Java, everything is achieved through class, object, and interface.By default, no Java class support cloning but Java provide one interface called Cloneable, which is a [marker interface](http://javarevisited.blogspot.com/2012/01/what-is-marker-interfaces-in-java-and.html) and by implementing this interface we can make the duplicate copy of our object by calling clone() method of java.lang.Object class.

This Method is protected inside the object class and Cloneable interface is a marker interface and this method also throw **CloneNotSupportedException** if we have not implemented thisinterface and try to call clone() method of Object class. By default any clone() method gives the **shallow copy** of the object i.e. if we invoke **super.clone()** then it’s a shallow copy but if we want to **deep copy** we have to override the clone() method and make it public and give own definition of making copy of object. Now we let’s see what is shallow and deep copy of object in Java programming language.

### 1. Shallow Copy in Java

Whenever we use default implementation of clone method we get shallow copy of object means it create new instance and copy all the field of object to that new instance and return it as object type we need to explicitly cast it back to our original object. This is shallow copy of the object. clone() method of the object class support shallow copy of the object. If the object contains primitive as well as non primitive or reference type variable In shallow copy, the cloned object also refers to the same object to which the original object refers as only the object references gets copied and not the referred objects themselves. That's why the name shallow copy or shallow cloning in Java. If only primitive type fields or [Immutable objects](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html) are there then there is no difference between shallow and deep copy in Java.

### 2. Deep Copy in Java

Whenever we need own meaning of copy not to use default implementation we call it as deep copy, whenever we need deep copy of the object we need to implement according to our need. So for deep copy we need to ensure all the member class also implement the Cloneable interface and override the clone() method of the object class.

After that we override the clone() method in all those classes even in the classes where we have only primitive type members otherwise we would not be able to call the protected clone() method of Object class on the instances of those classes inside some other class. It’s typical restriction of the protected access.

# **Difference between Shallow and Deep Copy in Java**

I think now we know what is deep and shallow copy of object in Java, let see some difference between them so that we can get some more clarity on them.

* When we call Object.clone(), this method performs a shallow copy of object, by copying data field by field, and if we override this method and by convention first call super.clone(), and then modify some fields to "deep" copy, then we get deep copy of object. This modification is done to ensure that original and cloned object are independent to each other.
* In shallow copy main or parent object is copied, but they share same fields or children if fields are modified in one parent object other parent fields have automatic same changes occur,but in deep copy this is not the case.
* If our parent object contains only primitive value then shallow copy is good for making clone of any object because in new object value is copied but if parent object contains any other object then only reference value is copied in new parent object and both will point to same object so in that case according to our need we can go for deep copy.
* Deep copy is expensive as compare to shallow copy in terms of object creation, because it involves recursive copying of data from other mutable objects, which is part of original object.

This is all about deep copy and shallow copy of objects in Java. Now the question comes when we use shallow copy and when go for deep copy, so the answer would be simple that if the object has only primitive fields or Immutable objects, then obviously we will go for shallow copy, but if the object has references to other mutable objects, then based on the requirement, shallow copy or deep copy can be chosen.

This means if the references are not modified anytime, then there is no point in going for deep copy, We can go for shallow copy. But if the references are modified often, then you need to go for deep copy. Again there is no hard and fast rule, it all depends on the requirement.

Hope this article will help to make clear about deep and shallow copy of cloning process.

# **What is the difference between StringBuilder and StringBuffer and String ?**

**Both StringBuilder and StringBuffer** are mutable. That means you can change the content of them, with in the same location.

Simply use StringBuilder unless you really are trying to share a buffer between threads. StringBuilder is

the unsynchronized (less overhead = more efficient) younger brother of the original synchronized StringBuffer class.

StringBuffer came first. Sun was concerned with correctness under all conditions, so they made it synchronized to make it thread-safe just in case.

StringBuilder came later. Most of the uses of StringBuffer were single-thread and unnecessarily paying the cost of the synchronization.

Since StringBuilder is a drop-in replacement for StringBuffer without the synchronization, there would not be differences between any examples.

If you are trying to share between threads, you can use StringBuffer, but consider whether higher-level synchronization is necessary, e.g. perhaps instead of using StringBuffer, should you synchronize the methods that use the StringBuilder.

**String**:

String is immutable, if you try to alter their values, another object gets created,

whereas StringBuffer and StringBuilder are mutable so they can change their values.

The String class represents character strings. All string literals in Java program, such as "abc" are implemented as instances of this class.

String objects are immutable once they are created we can't change. (Strings are constants).

If a String is created using constructor or method then those strings will be stored in Heap Memory as well as SringConstantPool. But before saving in pool it invokes intern() method to check object availability with same content in pool using equals method. If String-copy is available in the Pool then returns the reference.

Otherwise, String object is added to the pool and returns the reference.

Which one to use when?

StringBuilder : When you need a string, which can be modifiable, and only one thread is accessing

and modifying it.

StringBuffer : When you need a string, which can be modifiable, and multiple threads are accessing and modifying it.