**5What Is the JVM?**





## How Does the JVM Work?

As shown in the above architecture diagram, the JVM is divided into three main subsystems:

1. ClassLoader Subsystem
2. Runtime Data Area
3. Execution Engine

### 1. ClassLoader Subsystem

Java's [dynamic class loading](http://www.javainterviewpoint.com/use-of-class-forname-in-java/) functionality is handled by the ClassLoader subsystem. It loads, links. and initializes the class file when it refers to a class for the first time at runtime, not compile time.

#### 1.1 Loading

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

1. **BootStrap [ClassLoader](http://www.javainterviewpoint.com/" \t "_blank)** – Responsible for loading classes from the bootstrap classpath, nothing but **rt.jar.**Highest priority will be given to this loader.
2. **Extension ClassLoader** – Responsible for loading classes which are inside the ext folder **(jre\lib).**
3. **Application ClassLoader** –Responsible for loading Application Level Classpath, path mentioned Environment Variable, etc.

The above ClassLoaders will follow Delegation Hierarchy Algorithm while loading the class files.

#### 1.2 Linking

1. **Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.
2. **Prepare** – For all static variables memory will be allocated and assigned with default values.
3. **Resolve** – All symbolic memory references are replaced with the original references from Method Area.

#### 1.3 Initialization

This is the final phase of ClassLoading; here, all [static variables](http://www.javainterviewpoint.com/use-of-static-keyword-in-java/) will be assigned with the original values, and the [static block](http://www.javainterviewpoint.com/java-static-import/) will be executed.

### 2. Runtime Data Area

The Runtime Data Area is divided into five major components:

1. **Method Area** – All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
2. **Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.
3. **Stack Area** – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory. The stack area is thread-safe since it is not a shared resource. The Stack Frame is divided into three subentities:
   1. **Local Variable Array** – Related to the method how many local variables are involved and the corresponding values will be stored here.
   2. **Operand stack** – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
   3. **Frame data** – All symbols corresponding to the method is stored here. In the case of any **exception**, the catch block information will be maintained in the frame data.
4. **PC Registers** – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.
5. **Native Method stacks** – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

### 3. Execution Engine

The bytecode, which is assigned to the **Runtime Data Area,** will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

1. **Interpreter** – The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
2. **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
   1. **Intermediate Code Generator** – Produces intermediate code
   2. **Code Optimizer** – Responsible for optimizing the intermediate code generated above
   3. **Target Code Generator** – Responsible for Generating Machine Code or Native Code
   4. **Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.
3. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

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

**Native Method Libraries**: This is a collection of the Native Libraries, which is required for the Execution Engine.

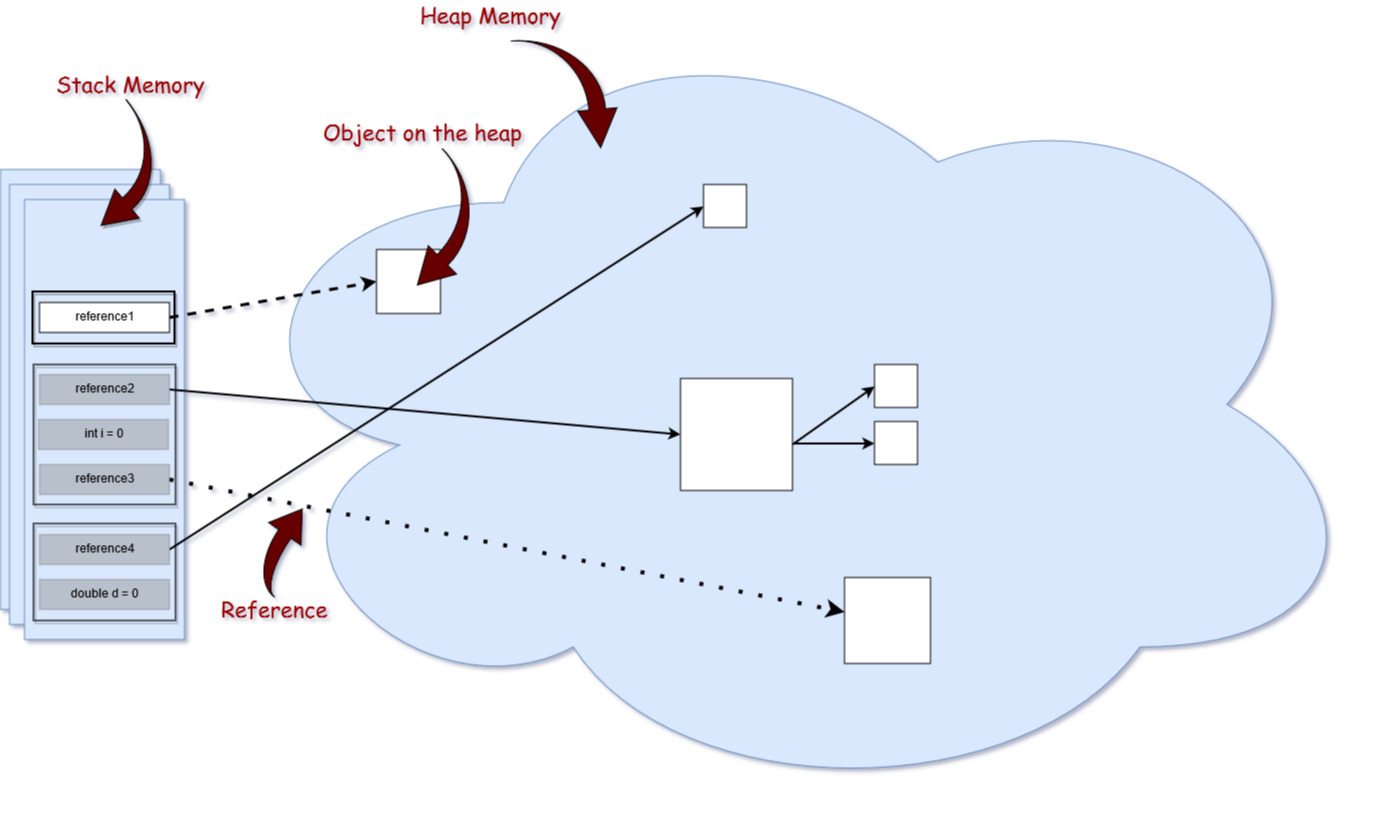
# Java Memory Management

Java has automatic memory management, a nice and quiet garbage collector that works in the background to clean up the unused objects and free up some memory.

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

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

To start with, let’s have a look at how the memory is generally organized in Java:



Memory Structure

Generally, memory is divided into two big parts: the **stack** and the **heap.**Please keep in mind that the size of memory types in this picture are not proportional to the memory size in reality. The heap is a huge amount of memory compared to the stack.

## ****The Stack****

Stack memory is responsible for holding references to heap objects and for storing value types (also known in Java as primitive types), which hold the value itself rather than a reference to an object from the heap.

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

Maybe you noticed that in the picture above, there are multiple stack memories displayed. This due to the fact that the stack memory in Java is allocated per Thread. Therefore, each time a Thread is created and started, it has its own stack memory — and cannot access another thread’s stack memory.

## ****The Heap****

This part of memory stores the actual object in memory. Those are referenced by the variables from the stack. For example, let’s analyze what happens in the following line of code:

1

StringBuilder builder = new StringBuilder();

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

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

The maximum stack and the heap sizes are not predefined — this depends on the running machine. However, later in this article, we will look into some JVM configurations that will allow us to specify their size explicitly for a running application.

## ****Reference Types****

If you look closely at the Memory Structure picture, you will probably notice that the arrows representing the references to the objects from the heap are actually of different types. That is because, in the Java programming language, we have different types of references: **strong, weak, soft,**and **phantom**references. The difference between the types of references is that the objects on the heap they refer to are eligible for garbage collecting under the different criteria. Let’s have a closer look at each of them.

### 1. Strong Reference

These are the most popular reference types that we all are used to. In the example above with the StringBuilder, we actually hold a strong reference to an object from the heap. The object on the heap it is not garbage collected while there is a strong reference pointing to it, or if it is strongly reachable through a chain of strong references.

### **2. Weak Reference**

In simple terms, a weak reference to an object from the heap is most likely to not survive after the next garbage collection process. A weak reference is created as follows:

1

WeakReference<StringBuilder> reference = new WeakReference<>(new StringBuilder());

A nice use case for weak references are caching scenarios. Imagine that you retrieve some data, and you want it to be stored in memory as well — the same data could be requested again. On the other hand, you are not sure when, or if, this data will be requested again. So you can keep a weak reference to it, and in case the garbage collector runs, it could be that it destroys your object on the heap. Therefore, after a while, if you want to retrieve the object you refer to, you might suddenly get back a null value. A nice implementation for caching scenarios is the collection **WeakHashMap<K,V>**. If we open the WeakHashMap class in the Java API, we see that its entries actually extend the WeakReference class and uses its **ref**field as the map’s key:

1

/\*\*

2

\* The entries in this hash table extend WeakReference, using its main ref

3

\* field as the key.

4

\*/

5

​

6

private static class Entry<K,V> extends WeakReference<Object> implements Map.Entry<K,V> {

7

​

8

V value;

Once a key from the WeakHashMap is garbage collected, the entire entry is removed from the map.

### **3. Soft Reference**

These types of references are used for more memory-sensitive scenarios, since those are going to be garbage collected only when your application is running low on memory. Therefore, as long as there is no critical need to free up some space, the garbage collector will not touch softly reachable objects. Java guarantees that all soft referenced objects are cleaned up before it throws an OutOfMemoryError. The Javadocs state, “all soft references to softly-reachable objects are guaranteed to have been cleared before the virtual machine throws an OutOfMemoryError.”

Similar to weak references, a soft reference is created as follows:

1

SoftReference<StringBuilder> reference = new SoftReference<>(new StringBuilder());

### **4. Phantom Reference**

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

## ****How****Strings****Are Referenced****

TheString type in Java is a bit differently treated. Strings are immutable, meaning that each time you do something with a string, another object is actually created on the heap. For strings, Java manages a string pool in memory. This means that Java stores and reuse strings whenever possible. This is mostly true for string literals. For example:

1

String localPrefix = "297"; //1

2

String prefix = "297"; //2

3

​

4

if (prefix == localPrefix)

5

{

6

System.out.println("Strings are equal" );

7

}

8

else

9

{

10

System.out.println("Strings are different");

11

}

When running, this prints out the following:

 Strings are equal

Therefore, it turns out that after comparing the two references of the String type, those actually point to the same objects on the heap. However, this is not valid for Strings that are computed. Let’s assume that we have the following change in line //1 of the above code

1

String localPrefix = new Integer(297).toString(); //1

Output:

 Strings are different

In this case, we actually see that we have two different objects on the heap. If we consider that the computed String will be used quite often, we can force the JVM to add it to the string pool by adding the .intern() method at the end of computed string:

1

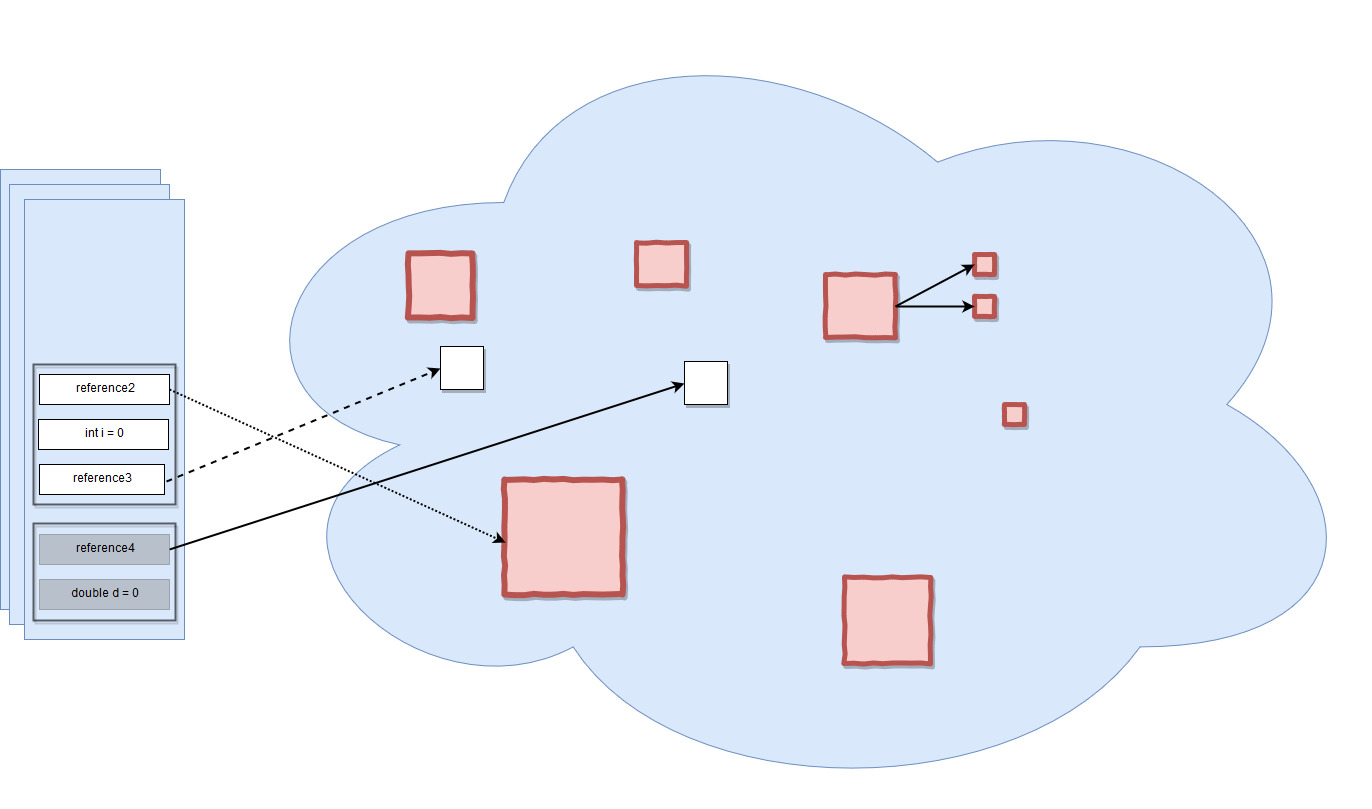
String localPrefix = new Integer(297).toString().intern(); //1

Adding the above change creates the following output:

 Strings are equal

## ****Garbage Collection Process****

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



Garbage-eligible objects

For example, all objects that are in red are eligible to be collected by the garbage collector. You might notice that there is an object on the heap, which has strong references to other objects that are also on the heap (e.g. could be a list that has references to its items, or an object that has two referenced type fields). However, since the reference from the stack is lost, it cannot be accessed anymore, so it is garbage as well.

To go a bit deeper into the details, let’s mention a few things first:

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

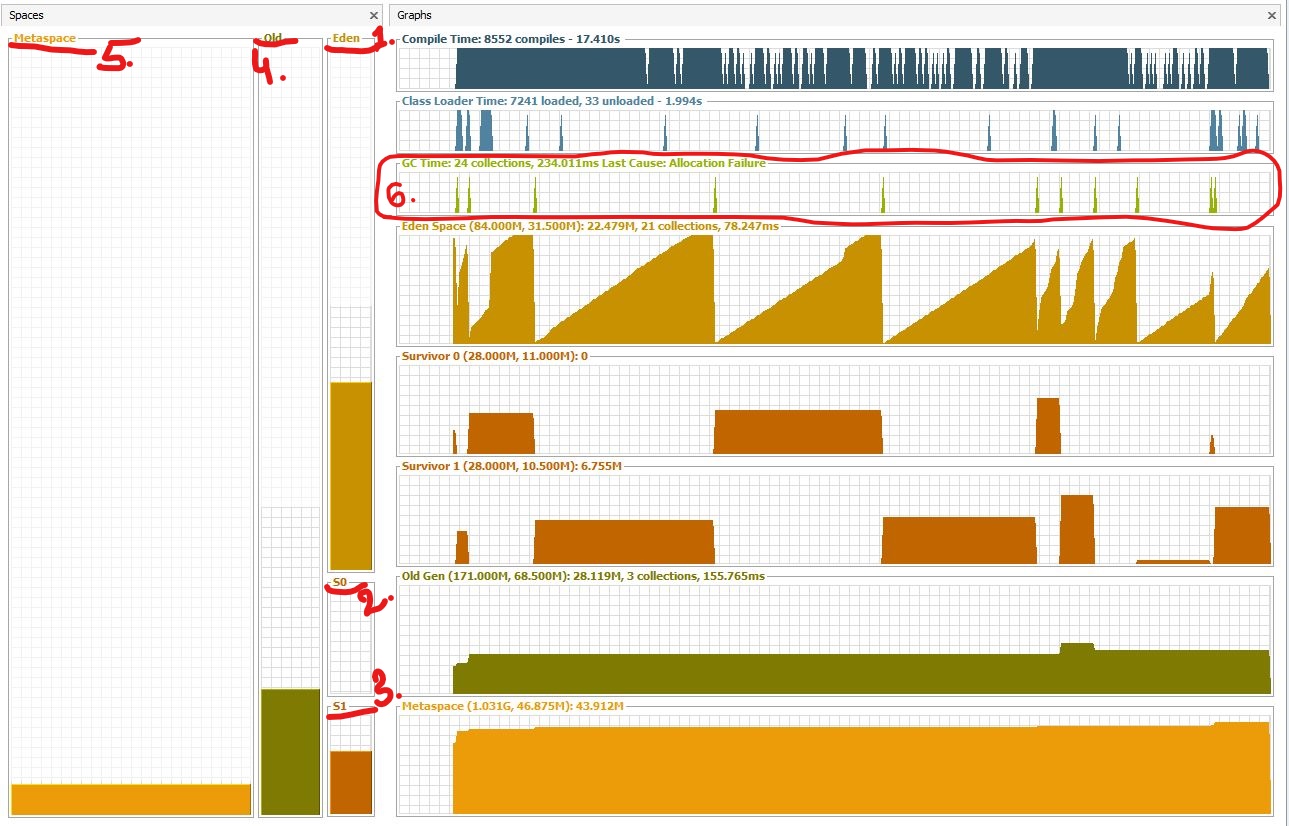
Even though Java decides when to run the garbage collector, you may explicitly call System.gc() and expect that the garbage collector will run when executing this line of code, right?

This is a wrong assumption.

You only kind of ask Java to run the garbage collector, but it’s, again, up to it whether or not to do that. Anyway, explicitly calling System.gc() is not advised.

Since this is a quite complex process, and it might affect you performance, it is implemented in a smart way. A so-called “Mark and Sweep” process is used for that. Java analyzes the variables from the stack and “marks” all the objects that need to be kept alive. Then, all the unused objects are cleaned up.

So actually, Java does not collect any garbage. In fact, the more garbage there is, and the fewer that objects are marked alive, the faster the process is. To make this even more optimized, heap memory actually consists of multiple parts. We can visualize the memory usage and other useful things with **JVisualVM**, a tool that comes with the Java JDK. The only thing you have to do is install a plugin named **Visual GC**, which allows you to see how the memory is actually structured. Let’s zoom in a bit and break down the big picture:

Heap memory generations

When an object is created, it is allocated on the **Eden(1)** space. Because the Eden space is not that big, it gets full quite fast. The garbage collector runs on the Eden space and marks objects as alive.

Once an object survives a garbage collecting process, it gets moved into a so-called survivor space **S0(2)**. The second time the garbage collector runs on the Eden space, it moves all surviving objects into the **S1(3)** space. Also, everything that is currently on **S0(2)** is moved into the **S1(3)** space.

If an object survives for X rounds of garbage collection (X depends on the JVM implementation, in my case it’s 8), it is most likely that it will survive forever, and it gets moved into the **Old(4)** space.

Taking everything said so far, if you look at the **garbage collector graph(6)**, each time it has run, you can see that the objects switch to the survivor space and that the Eden space gained space. And so on and so forth. The old generation can be also garbage collected, but since it is a bigger part of the memory compared to Eden space, it does not happen that often. The **Metaspace(5)** is used to store the metadata about your loaded classes in the JVM.

The presented picture is actually a Java 8 application. Prior to Java 8, the structure of the memory was a bit different. The metaspace is called actually the PermGen. space. For example, in Java 6, this space also stored the memory for the string pool. Therefore, if you have too many strings in your Java 6 application, it might crash.

### **Garbage Collector Types**

Actually, the JVM has three types of garbage collectors, and the programmer can choose which one should be used. By default, Java chooses the garbage collector type to be used based on the underlying hardware.

**1. Serial GC** – A single thread collector. Mostly applies to small applications with small data usage. Can be enabled by specifying the command line option: -XX:+UseSerialGC

**2. Parallel GC** – Even from the naming, the difference between Serial and Parallel would be that Parallel GC uses multiple threads to perform the garbage collecting process. This GC type is also known as the throughput collector. It can be enabled by explicitly specifying the option: -XX:+UseParallelGC

**3. Mostly concurrent GC** – If you remember, earlier in this article, it was mentioned that the garbage collecting process is actually pretty expensive, and when it runs, all thread are paused. However, we have this mostly concurrent GC type, which states that it works concurrent to the application. However, there is a reason why it is “mostly” concurrent. It does not work 100% concurrently to the application. There is a period of time for which the threads are paused. Still, the pause is kept as short as possible to achieve the best GC performance. Actually, there are 2 types of mostly concurrent GCs:

**3.1 Garbage First** – high throughput with a reasonable application pause time. Enabled with the option: -XX:+UseG1GC

**3.2 Concurrent Mark Sweep** – The application pause time is kept to a minimum. It can be used by specifying the option: -XX:+UseConcMarkSweepGC. As of JDK 9, this GC type is deprecated.

## ****Tips and Tricks****

* To minimize the memory footprint, limit the scope of the variables as much as possible. Remember that each time the top scope from the stack is popped up, the references from that scope are lost, and this could make objects eligible for garbage collecting.
* Explicitly refer to null obsolete references. That will make objects those refer to eligible for garbage collecting.
* Avoid finalizers. They slow down the process and they do not guarantee anything. Prefer phantom references for cleanup work.
* Do not use strong references where weak or soft references apply. The most common memory pitfalls are caching scenarios,when data is held in memory even if it might not be needed.
* JVisualVM also has the functionality to make a heap dump at a certain point, so you could analyze, per class, how much memory it occupies.
* Configure your JVM based on your application requirements. Explicitly specify the heap size for the JVM when running the application. The memory allocation process is also expensive, so allocate a reasonable initial and maximum amount of memory for the heap. If you know it will not make sense to start with a small initial heap size from the beginning, the JVM will extend this memory space. Specifying the memory options with the following options:
  + Initial heap size -Xms512m – set the initial heap size to 512 megabytes.
  + Maximum heap size -Xmx1024m – set the maximum heap size to 1024 megabytes.
  + Thread stack size -Xss1m – set the thread stack size to 1 megabytes.
  + Young generation size -Xmn256m – set the young generation size to 256 megabytes.
* If a Java application crashes with an OutOfMemoryError and you need some extra info to detect the leak, run the process with the –XX:HeapDumpOnOutOfMemory parameter, which will create a heap dump file when this error happens next time.
* Use the -verbose:gc option to get the garbage collection output. Each time a garbage collection takes place, an output will be generated.

#### [Java Programming Language](http://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html#javase8)

FEATURES

* Lambda Expressions, a new language feature, has been introduced in this release. They enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly.
* Method references provide easy-to-read lambda expressions for methods that already have a name.
* Default methods enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.
* Repeating Annotations provide the ability to apply the same annotation type more than once to the same declaration or type use.
* Type Annotations provide the ability to apply an annotation anywhere a type is used, not just on a declaration. Used with a pluggable type system, this feature enables improved type checking of your code.
* Improved type inference.
* Method parameter reflection.

#### [Collections](http://docs.oracle.com/javase/8/docs/technotes/guides/collections/changes8.html)

* Classes in the new java.util.stream package provide a Stream API to support functional-style operations on streams of elements. The Stream API is integrated into the Collections API, which enables bulk operations on collections, such as sequential or parallel map-reduce transformations.
* Performance Improvement for HashMaps with Key Collisions

[**Compact Profiles**](http://docs.oracle.com/javase/8/docs/technotes/guides/compactprofiles/) contain predefined subsets of the Java SE platform and enable applications that do not require the entire Platform to be deployed and run on small devices.

#### [Security](http://docs.oracle.com/javase/8/docs/technotes/guides/security/enhancements-8.html)

* Client-side TLS 1.2 enabled by default
* New variant of AccessController.doPrivileged that enables code to assert a subset of its privileges, without preventing the full traversal of the stack to check for other permissions
* Stronger algorithms for password-based encryption
* SSL/TLS Server Name Indication (SNI) Extension support in JSSE Server
* Support for AEAD algorithms: The SunJCE provider is enhanced to support AES/GCM/NoPadding cipher implementation as well as GCM algorithm parameters. And the SunJSSE provider is enhanced to support AEAD mode based cipher suites. See Oracle Providers Documentation, JEP 115.
* KeyStore enhancements, including the new Domain KeyStore type java.security.DomainLoadStoreParameter, and the new command option -importpassword for the keytool utility
* SHA-224 Message Digests
* Enhanced Support for NSA Suite B Cryptography
* Better Support for High Entropy Random Number Generation
* New java.security.cert.PKIXRevocationChecker class for configuring revocation checking of X.509 certificates
* 64-bit PKCS11 for Windows
* New rcache Types in Kerberos 5 Replay Caching
* Support for Kerberos 5 Protocol Transition and Constrained Delegation
* Kerberos 5 weak encryption types disabled by default
* Unbound SASL for the GSS-API/Kerberos 5 mechanism
* SASL service for multiple host names
* JNI bridge to native JGSS on Mac OS X
* Support for stronger strength ephemeral DH keys in the SunJSSE provider
* Support for server-side cipher suites preference customization in JSSE

#### [Tools](http://docs.oracle.com/javase/8/docs/technotes/tools/enhancements-8.html)

* The jjs command is provided to invoke the Nashorn engine.
* The java command launches JavaFX applications.
* The java man page has been reworked.
* The jdeps command-line tool is provided for analyzing class files.
* Java Management Extensions (JMX) provide remote access to diagnostic commands.
* The jarsigner tool has an option for requesting a signed time stamp from a Time Stamping Authority (TSA).

[**Javac tool**](http://docs.oracle.com/javase/8/docs/technotes/guides/javac/index.html)

* + The -parameters option of the javac command can be used to store formal parameter names and enable the Reflection API to retrieve formal parameter names.
  + The type rules for equality operators in the Java Language Specification (JLS) Section 15.21 are now correctly enforced by the javac command.
  + The javac tool now has support for checking the content of javadoc comments for issues that could lead to various problems, such as invalid HTML or accessibility issues, in the files that are generated when javadoc is run. The feature is enabled by the new -Xdoclint option. For more details, see the output from running "javac -X". This feature is also available in the javadoc tool, and is enabled there by default.
  + The javac tool now provides the ability to generate native headers, as needed. This removes the need to run the javah tool as a separate step in the build pipeline. The feature is enabled in javac by using the new -h option, which is used to specify a directory in which the header files should be written. Header files will be generated for any class which has either native methods, or constant fields annotated with a new annotation of type java.lang.annotation.Native.

[**Javadoc tool**](http://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/whatsnew-8.html)

* + The javadoc tool supports the new DocTree API that enables you to traverse Javadoc comments as abstract syntax trees.
  + The javadoc tool supports the new Javadoc Access API that enables you to invoke the Javadoc tool directly from a Java application, without executing a new process. See the [javadoc what's new](http://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/whatsnew-8.html) page for more information.
  + The javadoc tool now has support for checking the content of javadoc comments for issues that could lead to various problems, such as invalid HTML or accessibility issues, in the files that are generated when javadoc is run. The feature is enabled by default, and can also be controlled by the new -Xdoclint option. For more details, see the output from running "javadoc -X". This feature is also available in the javac tool, although it is not enabled by default there.

#### [Internationalization](http://docs.oracle.com/javase/8/docs/technotes/guides/intl/enhancements.8.html)

* Unicode Enhancements, including support for Unicode 6.2.0
* Adoption of Unicode CLDR Data and the java.locale.providers System Property
* New Calendar and Locale APIs
* Ability to Install a Custom Resource Bundle as an Extension

#### [Deployment](http://docs.oracle.com/javase/8/docs/technotes/guides/jweb/enhancements-8.html)

* For sandbox applets and Java Web Start applications, URLPermission is now used to allow connections back to the server from which they were started. SocketPermission is no longer granted.
* The Permissions attribute is required in the JAR file manifest of the main JAR file at all security levels.

[**Date-Time Package**](http://docs.oracle.com/javase/8/docs/technotes/guides/datetime/index.html)- a new set of packages that provide a comprehensive date-time model.

#### [Scripting](http://docs.oracle.com/javase/8/docs/technotes/guides/scripting/enhancements.html#jdk8)

* The Rhino javascript engine has been replaced with the [Nashorn](http://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/) Javascript Engine

#### [Pack200](http://docs.oracle.com/javase/8/docs/technotes/guides/pack200/enhancements.html)

* Pack200 Support for Constant Pool Entries and New Bytecodes Introduced by JSR 292
* JDK8 support for class files changes specified by JSR-292, JSR-308 and JSR-335

#### [IO and NIO](http://docs.oracle.com/javase/8/docs/technotes/guides/io/enhancements.html#jdk8)

* New SelectorProvider implementation for Solaris based on the Solaris event port mechanism. To use, run with the system property java.nio.channels.spi.Selector set to the value sun.nio.ch.EventPortSelectorProvider.
* Decrease in the size of the <JDK\_HOME>/jre/lib/charsets.jar file
* Performance improvement for the java.lang.String(byte[], \*) constructor and the java.lang.String.getBytes() method.

#### [java.lang and java.util Packages](http://docs.oracle.com/javase/8/docs/technotes/guides/lang/enhancements.html#jdk8)

* Parallel Array Sorting
* Standard Encoding and Decoding Base64
* Unsigned Arithmetic Support

#### [JDBC](http://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/)

* The JDBC-ODBC Bridge has been removed.
* JDBC 4.2 introduces new features.

#### Java DB

* JDK 8 includes Java DB 10.10.

#### [Networking](http://docs.oracle.com/javase/8/docs/technotes/guides/net/enhancements-8.0.html)

* The class java.net.URLPermission has been added.
* In the class java.net.HttpURLConnection, if a security manager is installed, calls that request to open a connection require permission.

#### [Concurrency](http://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/changes8.html)

* Classes and interfaces have been added to the java.util.concurrent package.
* Methods have been added to the java.util.concurrent.ConcurrentHashMap class to support aggregate operations based on the newly added streams facility and lambda expressions.
* Classes have been added to the java.util.concurrent.atomic package to support scalable updatable variables.
* Methods have been added to the java.util.concurrent.ForkJoinPool class to support a common pool.
* The java.util.concurrent.locks.StampedLock class has been added to provide a capability-based lock with three modes for controlling read/write access.

#### [Java XML](http://docs.oracle.com/javase/8/docs/technotes/guides/xml/enhancements.html) - [JAXP](http://docs.oracle.com/javase/8/docs/technotes/guides/xml/jaxp/enhancements-8.html)

#### [HotSpot](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/)

* Hardware intrinsics were added to use Advanced Encryption Standard (AES). The UseAES and UseAESIntrinsics flags are available to enable the hardware-based AES intrinsics for Intel hardware. The hardware must be 2010 or newer Westmere hardware. **Note:** AES intrinsics are only supported by the Server VM.

For example, to enable hardware AES, use the following flags: -XX:+UseAES -XX:+UseAESIntrinsics

To disable hardware AES use the following flags: -XX:-UseAES -XX:-UseAESIntrinsics

* Removal of PermGen.
* Default Methods in the Java Programming Language are supported by the byte code instructions for method invocation.

#### [Java Mission Control 5.3 Release Notes](https://www.oracle.com/java/technologies/javase/jmc53-release-notes.html)

* JDK 8 includes Java Mission Control 5.3.

## ****Using****try-with-resources

Support for *try-with-resources* — introduced in Java 7 — allows us to declare resources to be used in a *try* block with the assurance that the resources will be closed after the execution of that block.

The resources declared need to implement the *AutoCloseable* interface.

**try** (PrintWriter writer = **new** PrintWriter(**new** File("test.txt")))

{ writer.println("Hello World");

}

|  |  |
| --- | --- |
| Try catch | Try with resources |
| Scanner scanner = **null**;  **try**  {  scanner = **new** Scanner(**new** File("test.txt"));  **while** (scanner.hasNext()) { System.out.println(scanner.nextLine());  } }  **catch** (FileNotFoundException e) { e.printStackTrace();  } **finally** { **if** (scanner != **null**) { scanner.close(); } } | **try** (Scanner scanner = **new** Scanner(**new** File("test.txt")))  { **while** (scanner.hasNext())  { System.out.println(scanner.nextLine());  } } **catch** (FileNotFoundException fnfe)  { fnfe.printStackTrace(); } |

## try-with-resources****With Multiple Resources****

We can declare multiple resources just fine in a *try-with-resources* block by separating them with a semicolon:

**try** (Scanner scanner = **new** Scanner(**new** File("testRead.txt"));

PrintWriter writer = **new** PrintWriter(**new** File("testWrite.txt"))) {

**while** (scanner.hasNext()) {

writer.print(scanner.nextLine());

}

}

To construct a custom resource that will be correctly handled by a *try-with-resources* block, the class should implement the *Closeable* or *AutoCloseable* interfaces and override the *close* method:

**public** **class** **MyResource** **implements** **AutoCloseable**

{

@Override **public** **void** **close**() **throws** Exception { System.out.println("Closed MyResource");

}

}

Resources that were defined/acquired first will be closed last.

### Resource 1:

**public** **class** **AutoCloseableResourcesFirst** **implements** **AutoCloseable** {

**public** **AutoCloseableResourcesFirst**() {

System.out.println("Constructor -> AutoCloseableResources\_First");

}

**public** **void** **doSomething**() {

System.out.println("Something -> AutoCloseableResources\_First");

}

@Override

**public** **void** **close**() **throws** Exception {

System.out.println("Closed AutoCloseableResources\_First");

}

}

### Resource 2:

**public** **class** **AutoCloseableResourcesSecond** **implements** **AutoCloseable** {

**public** **AutoCloseableResourcesSecond**() {

System.out.println("Constructor -> AutoCloseableResources\_Second");

}

**public** **void** **doSomething**() {

System.out.println("Something -> AutoCloseableResources\_Second");

}

@Override

**public** **void** **close**() **throws** Exception {

System.out.println("Closed AutoCloseableResources\_Second");

}

}

### Code:

**private** **void** **orderOfClosingResources**() **throws** Exception {

**try** (AutoCloseableResourcesFirst af = **new** AutoCloseableResourcesFirst();

AutoCloseableResourcesSecond as = **new** AutoCloseableResourcesSecond()) {

af.doSomething();

as.doSomething();

}

}

*Constructor -> AutoCloseableResources\_First*  
*Constructor -> AutoCloseableResources\_Second*  
*Something -> AutoCloseableResources\_First*  
*Something -> AutoCloseableResources\_Second*  
*Closed AutoCloseableResources\_Second*  
*Closed AutoCloseableResources\_First*

## Comparable Interface

Comparison logic can also be used to place objects in a specific order. **The**[**Comparable interface**](https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Comparable.html)**allows us to define an ordering between objects**, by determining if an object is greater, equal, or lesser than another.

The Comparable interface is generic and has only one method, compareTo(), which takes an argument of the generic type and returns an int. The returned value is negative if this is lower than the argument, 0 if they are equal, and positive otherwise.

Let's say, in our *Person* class, we want to compare *Person* objects by their last name:

**public** **class** **Person** **implements** **Comparable**<**Person**> {

//...

@Override

**public** **int** **compareTo**(Person o) {

**return** **this**.lastName.compareTo(o.lastName);

}

}

The *compareTo()* method will return a negative *int*if called with a *Person* having a greater last name than *this*, zero if the same last name, and positive otherwise.

## Comparator Interface

The [Comparator interface](https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Comparator.html) is generic and has a compare method that takes two arguments of that generic type and returns an integer. We already saw that pattern earlier with the Comparable interface.

Comparator is similar; however, it's separated from the definition of the class. Therefore, **we can define as many Comparators we want for a class, where we can only provide one Comparable implementation.**

Let's imagine we have a web page displaying people in a table view, and we want to offer the user the possibility to sort them by first names rather than last names. It isn't possible with Comparable if we also want to keep our current implementation, but we could implement our own Comparators.

Let's create a Person Comparator that will compare them only by their first names:

**Comparator<Person> compareByFirstNames = Comparator.comparing(Person::getFirstName);**

Person joe = **new** Person("Joe", "Portman");

Person allan = **new** Person("Allan", "Dale");

List<Person> people = **new** ArrayList<>();

people.add(joe);

people.add(allan);

people.sort(compareByFirstNames);

assertThat(people).containsExactly(allan, joe);

There are other methods on the *Comparator* interface we can use in our *compareTo()*implementation:

@Override

**public** **int** **compareTo**(Person o) {

**return** Comparator.comparing(Person::getLastName)

.thenComparing(Person::getFirstName)

.thenComparing(Person::getBirthDate, Comparator.nullsLast(Comparator.naturalOrder()))

.compare(**this**, o);

}

In this case, we are first comparing last names, then first names. Then, we compare birth dates but as they are nullable we must say [how to handle that](https://marcin-chwedczuk.github.io/comparing-with-nullsFirst-and-nullsLast) so we give a second argument telling they should be compared according to their natural order but with *null*values going last.

## ClassNotFoundException

ClassNotFoundException is a checked exception which occurs when an application tries to load a class through its fully-qualified name and can not find its definition on the classpath.

This occurs mainly when trying to load classes using Class.forName(), ClassLoader.loadClass() or ClassLoader.findSystemClass(). Therefore, we need to be extra careful of java.lang.ClassNotFoundException while working with reflection.

For example, let’s try to load the JDBC driver class without adding necessary dependencies which will get us ClassNotFoundException:

@Test(expected = ClassNotFoundException.class)

**public** **void** **givenNoDrivers\_whenLoadDriverClass\_thenClassNotFoundException**()

**throws** ClassNotFoundException {

Class.forName("oracle.jdbc.driver.OracleDriver");

}

**3. *NoClassDefFoundError***

*NoClassDefFoundError* is a fatal error.

It occurs when JVM can not find the definition of the class while trying to:

* Instantiate a class by using the *new* keyword
* Load a class with a method call

The error occurs when a compiler could successfully compile the class, but Java runtime could not locate the class file.

It usually happens when there is an exception while executing a static block or initializing static fields of the class, so class initialization fails.

Let's consider a scenario which is one simple way to reproduce the issue. *ClassWithInitErrors*initialization throws an exception. So, when we try to create an object of *ClassWithInitErrors,* it throws *ExceptionInInitializerError.*

If we try to load the same class again, we get the *NoClassDefFoundError:*

**public** **class** **ClassWithInitErrors** {

**static** **int** data = 1 / 0;

}

**public** **class** **NoClassDefFoundErrorExample** {

**public** ClassWithInitErrors **getClassWithInitErrors**() {

ClassWithInitErrors test;

**try** {

test = **new** ClassWithInitErrors();

} **catch** (Throwable t) {

System.out.println(t);

}

test = **new** ClassWithInitErrors();

**return** test;

}

}

## Unboxing Long Values

### Using the .longValue() Method

Next, let's use the “==” comparison operator, but in a safe way. The class Number has a method [.longValue()](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Number.html#longValue()) which unwraps the primitive long value:

Long l1 = 128L;

Long l2 = 128L;

assertThat(l1.longValue() == l2.longValue()).isTrue();

### Casting to Primitive Values

A different way to unbox a Long is by [casting](https://www.baeldung.com/java-type-casting) the objects to primitive types. Therefore, we'll extract the primitive value and then we can proceed to use the comparison operator:

Long l1 = 128L;

Long l2 = 128L;

assertThat((**long**) l1 == (**long**) l2).isTrue();

Note that, **for the .longValue() method or using casting, we should check if the object is null.** We could have a NullPointerException if the object is null.

***volatile* and Thread Synchronization**

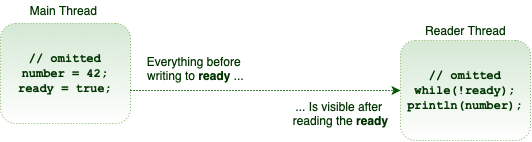
For multithreaded applications, we need to ensure a couple of rules for consistent behavior:

* Mutual Exclusion – only one thread executes a critical section at a time
* Visibility – changes made by one thread to the shared data are visible to other threads to maintain data consistency

*synchronized* methods and blocks provide both of the above properties, at the cost of application performance.

*volatile* is quite a useful keyword because it **can help ensure the visibility aspect of the data change without, of course, providing mutual exclusion**. Thus, it's useful in the places where we're ok with multiple threads executing a block of code in parallel, but we need to ensure the visibility property.

let's suppose thread A writes to a *volatile* variable, and then thread B reads the same *volatile* variable. In such cases, **the values that were visible to A before writing the *volatile* variable will be visible to B after reading the *volatile* variable:**

[](https://www.baeldung.com/wp-content/uploads/2017/08/happens-before.png)

**Technically speaking, any write to a *volatile* field happens before every subsequent read of the same field**. This is the *volatile* variable rule of the Java Memory Model ([JMM](https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html)).

**Because of the strength of the happens-before memory ordering, sometimes we can piggyback on the visibility properties of another *volatile* variable**. For instance, in our particular example, we just need to mark the *ready*variable as *volatile*:

**public** **class** **TaskRunner** {

**private** **static** **int** number; // not volatile

**private** **volatile** **static** **boolean** ready;

// same as before

}

Anything prior to writing *true*to the *ready* variable is visible to anything after reading the *ready*variable. Therefore, the *number*variable piggybacks on the memory visibility enforced by the *ready* variable. **Put simply*,*even though it's not a *volatile*variable, it is exhibiting a *volatile* behavior.**

By making use of these semantics, we can define only a few of the variables in our class as *volatile*and optimize the visibility guarantee.

## ThreadLocal****API****

The TheadLocal construct allows us to store data that will be **accessible only**by**a specific thread**.

Let's say that we want to have an Integer value that will be bundled with the specific thread:

Next, when we want to use this value from a thread we only need to call a get() or set() method. Simply put, we can think that ThreadLocal stores data inside of a map – with the thread as the key.

Due to that fact, when we call a get() method on the threadLocalValue, we will get an Integer value for the requesting thread:

ThreadLocal<Integer> threadLocalValue = **new** ThreadLocal<>();

threadLocalValue.set(1);

Integer result = threadLocalValue.get();

We can construct an instance of the *ThreadLocal* by using the *withInitial()* static method and passing a supplier to it:

ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 1);

To remove the value from the *ThreadLocal*, we can call the *remove()* method:

threadLocal.remove();

## ****Difference Between Daemon and User Threads****

Java offers two types of threads: user threads and daemon threads.

User threads are high-priority threads. **The JVM will wait for any user thread to complete its task before terminating it.**

On the other hand, **daemon threads are low-priority threads whose only role is to provide services to user threads.**

Since daemon threads are meant to serve user threads and are only needed while user threads are running, they won't prevent the JVM from exiting once all user threads have finished their execution.

That's why infinite loops, which typically exist in daemon threads, will not cause problems, because any code, including the *finally* blocks, won't be executed once all user threads have finished their execution. For this reason, **daemon threads are not recommended for I/O tasks.**

However, there're exceptions to this rule. Poorly designed code in daemon threads can prevent the JVM from exiting. For example, calling *Thread.join()* on a running daemon thread can block the shutdown of the application.

o set a thread to be a daemon thread, all we need to do is to call *Thread.setDaemon().*In this example, we'll use the *NewThread* class which extends the *Thread* class:

NewThread daemonThread = **new** NewThread();

daemonThread.setDaemon(**true**);

daemonThread.start();

**Any thread inherits the daemon status of the thread that created it.** Since the main thread is a user thread, any thread that is created inside the main method is by default a user thread.

The method *setDaemon()* can only be called after the *Thread* object has been created and the thread has not been started. An attempt to call *setDaemon()* while a thread is running will throw an *IllegalThreadStateException*:

@Test(expected = IllegalThreadStateException.class)

**public** **void** **whenSetDaemonWhileRunning\_thenIllegalThreadStateException**() {

NewThread daemonThread = **new** NewThread();

daemonThread.start();

daemonThread.setDaemon(**true**);

}

# **Fork/Join**

The fork/join framework was presented in Java 7. It provides tools to help speed up parallel processing by attempting to use all available processor cores – which is accomplished **through a divide and conquer approach**.

In practice, this means that **the framework first “forks”**, recursively breaking the task into smaller independent subtasks until they are simple enough to be executed asynchronously.

After that, **the “join” part begins**, in which results of all subtasks are recursively joined into a single result, or in the case of a task which returns void, the program simply waits until every subtask is executed.

To provide effective parallel execution, the fork/join framework uses a pool of threads called the *ForkJoinPool*, which manages worker threads of type *ForkJoinWorkerThread*.