Q: How does class loading work when the same class exists in different applications on the same server?

Ans:

A Java classloader typically works by looking for classes in one or more places in a fixed sequence. For instance, the classloader that loads your application when you run it from the command line looks first in the rt.jar file (and others on the bootclasspath), and then in the directories and JAR files specified by your classpath.

A webapp classloading is similar in principle, but a bit more complicated in practice. For a particular webapp, a webapp's classloader looks for classes in the following order. For example Tomcat 6 looks for classes in this order:

1. Bootstrap classes of your JVM
2. System class loader classes (described [here](http://tomcat.apache.org/tomcat-6.0-doc/class-loader-howto.html))
3. /WEB-INF/classes of the webapp
4. /WEB-INF/lib/\*.jar of the webapp
5. $CATALINA\_HOME/lib
6. $CATALINA\_HOME/lib/\*.jar

Of course, once the classloader has found the class it is looking for, it looks no further. So classes with the same name later in the order won't get loaded.

The complication is that the web container has one classloader for each webapp, and these classloaders delegate to other classloaders that manage the common classes. In practice, this means that some classes will only ever be loaded once for the entire container (e.g. 1. and 2.) and others may get loaded multiple times by different classloaders.

(When a class is loaded more than once, it results in distinct Class objects and distinct class statics. The versions of the class are different types as far as the JVM is concerned and you cannot typecast from one version to the other.)

Finally, Tomcat can be configure to allow individual webapps to be "hot loaded". This entails stopping a webapp, creating a new classloader for it, and restarting it.

**FOLLOWUP**

So ... synchronizing a static method will not protect access to a shared resource where the class has been loaded multiple times?

It depends on the details, but it probably won't. (Or to look at if another way, if a class has *actually*been loaded multiple times, then a static method of each "load" of the class will access a different set of static fields.)

If you really want a singleton application class instance to be shared by multiple webapps in the same container, it is simplest if you put the class into $CATALINA\_HOME/lib or the equivalent. But you also should ask yourself if this is good system design. Consider combining the webapps, or to using request forwarding etc instead of a shared data structure. The singleton pattern tends to be troublesome in webapps, and this flavor is even more so.

Q: What is a Java ClassLoader?

Taken from this nice [tutorial](http://www.oracle.com/technetwork/articles/javase/classloaders-140370.html) from Sun:

Motivation

Applications written in statically compiled programming languages, such as C and C++, are compiled into native, machine-specific instructions and saved as an executable file. The process of combining the code into an executable native code is called linking - the merging of separately compiled code with shared library code to create an executable application. This is different in dynamically compiled programming languages such as Java. In Java, the .class files generated by the Java compiler remain as-is until loaded into the Java Virtual Machine (JVM) -- in other words, the linking process is performed by the JVM at runtime. Classes are loaded into the JVM on an 'as needed' basis. And when a loaded class depends on another class, then that class is loaded as well.

When a Java application is launched, the first class to run (or the entry point into the application) is the one with public static void method called main(). This class usually has references to other classes, and all attempts to load the referenced classes are carried out by the class loader.

To get a feeling of this recursive class loading as well as the class loading idea in general, consider the following simple class:

public class HelloApp {

public static void main(String argv[]) {

System.out.println("Aloha! Hello and Bye");

}

}

If you run this class specifying the -verbose:class command-line option, so that it prints what classes are being loaded, you will get an output that looks as follows. Note that this is just a partial output since the list is too long to show here.

prmpt>java -verbose:class HelloApp

[Opened C:\Program Files\Java\jre1.5.0\lib\rt.jar]

[Opened C:\Program Files\Java\jre1.5.0\lib\jsse.jar]

[Opened C:\Program Files\Java\jre1.5.0\lib\jce.jar]

[Opened C:\Program Files\Java\jre1.5.0\lib\charsets.jar]

[Loaded java.lang.Object from shared objects file]

[Loaded java.io.Serializable from shared objects file]

[Loaded java.lang.Comparable from shared objects file]

[Loaded java.lang.CharSequence from shared objects file]

[Loaded java.lang.String from shared objects file]

[Loaded java.lang.reflect.GenericDeclaration from shared objects file]

[Loaded java.lang.reflect.Type from shared objects file]

[Loaded java.lang.reflect.AnnotatedElement from shared objects file]

[Loaded java.lang.Class from shared objects file]

[Loaded java.lang.Cloneable from shared objects file]

[Loaded java.lang.ClassLoader from shared objects file]

[Loaded java.lang.System from shared objects file]

[Loaded java.lang.Throwable from shared objects file]

.

.

.

[Loaded java.security.BasicPermissionCollection from shared objects file]

[Loaded java.security.Principal from shared objects file]

[Loaded java.security.cert.Certificate from shared objects file]

[Loaded HelloApp from file:/C:/classes/]

Aloha! Hello and Bye

[Loaded java.lang.Shutdown from shared objects file]

[Loaded java.lang.Shutdown$Lock from shared objects file]

As you can see, the Java runtime classes required by the application class (HelloApp) are loaded first.

Class Loaders in the Java 2 Platform

The Java programming language keeps evolving to make the life of applications developers easier everyday. This is done by providing APIs that simplify your life by allowing you to concentrate on business logic rather than implementation details of fundamental mechanisms. This is evident by the recent change of J2SE 1.5 to J2SE 5.0 in order to reflect the maturity of the Java platform.

As of JDK 1.2, a bootstrap class loader that is built into the JVM is responsible for loading the classes of the Java runtime. This class loader only loads classes that are found in the boot classpath, and since these are trusted classes, the validation process is not performed as for untrusted classes. In addition to the bootstrap class loader, the JVM has an extension class loader responsible for loading classes from standard extension APIs, and a system class loader that loads classes from a general class path as well as your application classes.

Since there is more than one class loader, they are represented in a tree whose root is the bootstrap class loader. Each class loader has a reference to its parent class loader. When a class loader is asked to load a class, it consults its parent class loader before attempting to load the item itself. The parent in turn consults its parent, and so on. So it is only after all the ancestor class loaders cannot find the class that the current class loader gets involved. In other words, a delegation model is used.

The java.lang.ClassLoader Class

The java.lang.ClassLoader is an abstract class that can be subclassed by applications that need to extend the manner in which the JVM dynamically loads classes. Constructors in java.lang.ClassLoader (and its subclasses) allow you to specify a parent when you instantiate a new class loader. If you don't explicitly specify a parent, the virtual machine's system class loader will be assigned as the default parent. In other words, the ClassLoader class uses a delegation model to search for classes and resources. Therefore, each instance of ClassLoader has an associated parent class loader, so that when requested to find a class or resources, the task is delegated to its parent class loader before attempting to find the class or resource itself. The loadClass() method of the ClassLoader performs the following tasks, in order, when called to load a class:

If a class has already been loaded, it returns it. Otherwise, it delegates the search for the new class to the parent class loader. If the parent class loader doesn't find the class, loadClass() calls the method findClass() to find and load the class. The finalClass() method searches for the class in the current class loader if the class wasn't found by the parent class loader.

There's more in the original article, which also shows you how to implement your own network class loaders, which answers your question as to why (and how). See also the [API docs](http://java.sun.com/javase/6/docs/api/java/lang/ClassLoader.html).

Java Classloader:

The Java Classloader is a part of the [Java Runtime Environment](https://en.wikipedia.org/wiki/Java_Runtime_Environment) that [dynamically loads](https://en.wikipedia.org/wiki/Dynamic_loading) [Java classes](https://en.wikipedia.org/wiki/Java_class) into the [Java Virtual Machine](https://en.wikipedia.org/wiki/Java_Virtual_Machine).

Each Java class must be loaded by a class loader.

When the JVM is started, three class loaders are used:[[3]](https://en.wikipedia.org/wiki/Java_Classloader" \l "cite_note-3)[[4]](https://en.wikipedia.org/wiki/Java_Classloader#cite_note-4)

1. Bootstrap class loader
2. Extensions class loader
3. System class loader

The bootstrap class loader loads the core Java libraries[[5]](https://en.wikipedia.org/wiki/Java_Classloader" \l "cite_note-5) located in the <JAVA\_HOME>/jre/lib directory. This class loader, which is part of the core JVM, is written in native code.

The extensions class loader loads the code in the extensions directories (<JAVA\_HOME>/jre/lib/ext,[[6]](https://en.wikipedia.org/wiki/Java_Classloader" \l "cite_note-6) or any other directory specified by the java.ext.dirs system property). It is implemented by the sun.misc.Launcher$ExtClassLoader class.

The system class loader loads code found on java.class.path, which maps to the [CLASSPATH](https://en.wikipedia.org/wiki/Classpath_(Java)) [environment variable](https://en.wikipedia.org/wiki/Environment_variable). This is implemented by the sun.misc.Launcher$AppClassLoaderclass.

# Java VisualVM - Monitoring Application Threads

Java VisualVM presents data for local and remote applications in a tab specific for that application. Application tabs are displayed in the main window to the right of the Applications window. You can have multiple application tabs open at one time. Each application tab contains sub-tabs that display different types of information about the application.

## Monitoring Thread Activity

Java VisualVM displays real-time, high-level data on thread activity in the Threads tab.  
**Note:** The information displayed in the Threads tab is based on Java Management Extensions (JMX). The Threads tab is visible if Java VisualVM can make a JMX technology-based connection (JMX connection) with the target application and retrieve JMX instrumentation from the Java Virtual Machine (JVM). If the target application is a local application running on Java Development Kit (JDK) version 1.6, the JMX connection is made automatically. If the target application is not running on JDK version 1.6, you may need to explicitly establish a JMX connection with the JVM software. For more on establishing a JMX connection, see the following document:

* [Connecting to JMX Agents Explicitly](https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/jmx_connections.html)

By default the Threads tab displays a timeline of current thread activity. You can click a thread in the timeline to view details about that thread in the Details tab.

### Timeline Tab

This tab displays a timeline with real-time thread states. Use the buttons in the Timeline toolbar to zoom in/out on the current view and to switch to the Scale to Fit mode. The drop-down list enables you to select which threads are displayed. You can choose to view all threads, live threads or finished threads. You can also select a single thread or multiple threads to display a subset of the threads. You can double-click on a thread timeline to open that thread in the Details tab.

A timeline for each thread provides a quick overview of the thread's activity.

## Taking a Thread Dump

You can use Java VisualVM to take a thread dump (stack trace) while a local application is running. Taking a thread dump does not stop the application. When you print the thread dump you get a printout of the thread stack that includes thread states for the Java threads.

When you print a thread dump in Java VisualVM, the tool prints a stack trace of the active threads of the application. Using Java VisualVM to take a thread dump can be very convenient in cases where you do not have a command-line console for the application. You can use a stack trace to help diagnose a number of issues such as deadlocks or when an application hangs.

Screenshot of thread dump (stack trace) in thread dump sub-tab.

Q: How to identify stuck thread in java? How to resolve issue?

1. By using Visual VM take Thread dump, and kill the threads which are stuck.
2. Use interrupt(), instead of thread.stop().

Q: prevent deadlocks in java?

Some quick tips out of my head

* don't use multiple threads (like Swing does, for example, by mandating that everything is done in the EDT)
* don't hold several locks at once. If you do, always acquire the locks in the same order
* don't execute foreign code while holding a lock
* use interruptible locks

Q: What is a covariant return type?

Covariant return, means that when one overrides a method, the return type of the overriding method is allowed to be a subtype of the overridden method's return type.

To clarify this with an example, a common case is Object.clone() - which is declared to return a type of Object. You could override this in your own class as follows:

public class MyFoo

{

...

// Note covariant return here, method does not just return Object

public MyFoo clone()

{

// Implementation

}

}

The benefit here is that any method which holds an explicit reference to a MyFoo object will be able to invoke clone() and know (without casting) that the return value is an instance of MyFoo. Without covariant return types, the overridden method in MyFoo would have to be declared to return Object- and so calling code would have to explicitly downcast the result of the method call (even thought both sides "know" it can only ever be an instance of MyFoo).

Note that there's nothing special about clone() and that any overridden method can have a covariant return - I used it as an example here as it's a standard method where this is often useful.

**Q: What is the difference between a process and a thread?**

Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.

I'm not sure what "hardware" vs "software" threads you might be referring to. Threads are an operating environment feature, rather than a CPU feature (though the CPU typically has operations that make threads efficient).

Erlang uses the term "process" because it does not expose a shared-memory multiprogramming model. Calling them "threads" would imply that they have shared memory

**Process**  
Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

**Thread**  
A thread is an entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients

Q: executorService.execute(**new** Task(i));

Executes the given task sometime in the future. The task may execute in a new thread or in an existing pooled thread. If the task cannot be submitted for execution, either because this executor has been shutdown or because its capacity has been reached, the task is handled by the current RejectedExecutionHandler.

Q: How does a single servlet handle multiple requests from client side?

Q: What do “branch”, “tag” and “trunk” mean in Subversion repositories?

Ans1:

Not sure I agree with Nick re tag being similar to a branch. A tag is just a marker

* [**Trunk**](http://svnbook.red-bean.com/en/1.8/svn.tour.importing.html#svn.tour.importing.layout) would be the main body of development, originating from the start of the project until the present.
* [**Branch**](http://svnbook.red-bean.com/en/1.8/svn.branchmerge.whatis.html) will be a copy of code derived from a certain point in the trunk that is used for applying major changes to the code while preserving the integrity of the code in the trunk. If the major changes work according to plan, they are usually merged back into the trunk.
* [**Tag**](http://svnbook.red-bean.com/en/1.8/svn.branchmerge.tags.html) will be a point in time on the trunk or a branch that you wish to preserve. The two main reasons for preservation would be that either this is a major release of the software, whether alpha, beta, RC or RTM, or this is the most stable point of the software before major revisions on the trunk were applied.

In open source projects, major branches that are not accepted into the trunk by the project stakeholders can become the bases for *forks* -- e.g., totally separate projects that share a common origin with other source code.

Ans2:

First of all, as @AndrewFinnell and @KenLiu point out, in SVN the directory names themselves mean nothing -- "trunk, branches and tags" are simply a common convention that is used by most repositories. Not all projects use all of the directories (it's reasonably common not to use "tags" at all), and in fact, nothing is stopping you from calling them anything you'd like, though breaking convention is often confusing.

I'll describe probably the most common usage scenario of branches and tags, and give an example scenario of how they are used.

* **Trunk**: The main development area. This is where your next major release of the code lives, and generally has all the newest features.
* **Branches**: Every time you release a major version, it gets a branch created. This allows you to do bug fixes and make a new release without having to release the newest - possibly unfinished or untested - features.
* **Tags**: Every time you release a version (final release, release candidates (RC), and betas) you make a tag for it. This gives you a point-in-time copy of the code as it was at that state, allowing you to go back and reproduce any bugs if necessary in a past version, or re-release a past version exactly as it was. Branches and tags in SVN are lightweight - on the server, it does not make a full copy of the files, just a marker saying "these files were copied at this revision" that only takes up a few bytes. With this in mind, you should never be concerned about creating a tag for any released code. As I said earlier, tags are often omitted and instead, a changelog or other document clarifies the revision number when a release is made.

For example, let's say you start a new project. You start working in "trunk", on what will eventually be released as version 1.0.

* **trunk/ - development version, soon to be 1.0**
* branches/ - empty

Once 1.0.0 is finished, you branch trunk into a new "1.0" branch, and create a "1.0.0" tag. Now work on what will eventually be 1.1 continues in trunk.

* trunk/ - development version, **soon to be 1.1**
* **branches/1.0 - 1.0.0 release version**
* **tags/1.0.0 - 1.0.0 release version**

You come across some bugs in the code, and fix them in trunk, and then merge the fixes over to the 1.0 branch. You can also do the opposite, and fix the bugs in the 1.0 branch and then merge them back to trunk, but commonly projects stick with merging one-way only to lessen the chance of missing something. Sometimes a bug can only be fixed in 1.0 because it is obsolete in 1.1. It doesn't really matter: you only want to make sure that you don't release 1.1 with the same bugs that have been fixed in 1.0.

* trunk/ - development version, soon to be 1.1
* branches/1.0 - **upcoming 1.0.1 release**
* tags/1.0.0 - 1.0.0 release version

Once you find enough bugs (or maybe one critical bug), you decide to do a 1.0.1 release. So you make a tag "1.0.1" from the 1.0 branch, and release the code. At this point, trunk will contain what will be 1.1, and the "1.0" branch contains 1.0.1 code. The next time you release an update to 1.0, it would be 1.0.2.

* trunk/ - development version, soon to be 1.1
* branches/1.0 - **upcoming 1.0.2 release**
* tags/1.0.0 - 1.0.0 release version
* **tags/1.0.1 - 1.0.1 release version**

Eventually you are almost ready to release 1.1, but you want to do a beta first. In this case, you likely do a "1.1" branch, and a "1.1beta1" tag. Now, work on what will be 1.2 (or 2.0 maybe) continues in trunk, but work on 1.1 continues in the "1.1" branch.

* trunk/ - development version, **soon to be 1.2**
* branches/1.0 - upcoming 1.0.2 release
* **branches/1.1 - upcoming 1.1.0 release**
* tags/1.0.0 - 1.0.0 release version
* tags/1.0.1 - 1.0.1 release version
* **tags/1.1beta1 - 1.1 beta 1 release version**

Once you release 1.1 final, you do a "1.1" tag from the "1.1" branch.

You can also continue to maintain 1.0 if you'd like, porting bug fixes between all three branches (1.0, 1.1, and trunk). The important takeaway is that for every main version of the software you are maintaining, you have a branch that contains the latest version of code for that version.

Another use of branches is for features. This is where you branch trunk (or one of your release branches) and work on a new feature in isolation. Once the feature is completed, you merge it back in and remove the branch.

* trunk/ - development version, soon to be 1.2
* branches/1.1 - upcoming 1.1.0 release
* **branches/ui-rewrite - experimental feature branch**

The idea of this is when you're working on something disruptive (that would hold up or interfere with other people from doing their work), something experimental (that may not even make it in), or possibly just something that takes a long time (and you're afraid if it holding up a 1.2 release when you're ready to branch 1.2 from trunk), you can do it in isolation in branch. Generally you keep it up to date with trunk by merging changes into it all the time, which makes it easier to re-integrate (merge back to trunk) when you're finished.

Also note, the versioning scheme I used here is just one of many. Some teams would do bug fix/maintenance releases as 1.1, 1.2, etc., and major changes as 1.x, 2.x, etc. The usage here is the same, but you may name the branch "1" or "1.x" instead of "1.0" or "1.0.x". (Aside, [semantic versioning](http://semver.org/) is a good guide on how to do version numbers).

# Q: [How to understand the “synchronous” and “asynchronouns” messaging in JMS?](https://stackoverflow.com/questions/22088873/how-to-understand-the-synchronous-and-asynchronouns-messaging-in-jms)

After reading some document of JMS, I totally puzzled by the phrase synchronous and asynchronouns.

See this page: <http://docs.oracle.com/cd/E19798-01/821-1841/bncdq/index.html>

**Synchronous**

You use the receive method to consume a message synchronously. You can use this method at any time after you call the start method:

connection.start();

Message m = consumer.receive();

connection.start();

Message m = consumer.receive(1000); // time out after a second

To consume a message asynchronously, you use a message listener, described in the next section.

**Asynchronous**

JMS Message Listeners A message listener is an object that acts as an asynchronous event handler for messages. This object implements the MessageListener interface, which contains one method, onMessage. In the onMessage method, you define the actions to be taken when a message arrives.

You register the message listener with a specific MessageConsumer by using the setMessageListener method. For example, if you define a class named Listener that implements the MessageListener interface, you can register the message listener as follows:

Listener myListener = new Listener();

consumer.setMessageListener(myListener);

I have two questions:

1. As what I understood, the nature of JMS is asynchronous. Producer publishes messages to the queue/topic, it doesn't need to wait consumer. This is asynchronous behaviour. How can it be "synchronous"?
2. If the "mesageListener" is asynchronous, but in my test with spring-jms, I found it always running in a thread. That means, if I write Thread.sleep(2000) in onMessage, it have to be wait 2 seconds before processing next message. Is it "asynchronous"?

Ans:

If you understand it better like this, consumer.receive() uses a pull model: you read from a queue and are blocked waiting for this message until it comes, or some timeout has elapsed.

Using a listener uses a push model: you register a listener and, when a message comes in, the listener is called, in a separate thread.

Everything is done in a thread in Java, and the listener call is no exception. Whether the listener message handling prevents the processing of other messages in the queue depends on how many threads are dedicated to message processing. If you configure Spring to use a pool of 5 threads to process messages asynchronously, then 5 listeners will be able to process messages in parallel.

Q: What are the advantages of using an ExecutorService?

What is the advantage of using ExecutorService over running threads passing a Runnable into the Thread constructor?

ANs:

ExecutorService abstracts away many of the complexities associated with the lower-level abstractions like raw Thread. It provides mechanisms for safely starting, closing down, submitting, executing, and blocking on the successful or abrupt termination of tasks (expressed as Runnable or Callable).

From [JCiP](http://www.javaconcurrencyinpractice.com/), Section 6.2, straight from the horse's mouth:

Executor may be a simple interface, but it forms the basis for a flexible and powerful framework for asynchronous task execution that supports a wide variety of task execution policies. It provides a standard means of decoupling *task submission* from *task execution*, describing tasks as Runnable. The Executor implementations also provide lifecycle support and hooks for adding statistics gathering, application management, and monitoring. ... *Using an Executor is usually the easiest path to implementing a producer-consumer design in your application.*

Rather than spending your time implementing (often incorrectly, and with great effort) the underlying infrastructure for parallelism, the j.u.concurrent framework allows you to instead focus on structuring tasks, dependencies, potential parallelism. For a large swath of concurrent applications, it is straightforward to identify and exploit task boundaries and make use of j.u.c, allowing you to focus on the much smaller subset of true concurrency challenges which may require more specialized solutions.

Also, despite the boilerplate look and feel, the [Oracle API page summarizing the concurrency utilities](http://download.oracle.com/javase/1.5.0/docs/guide/concurrency/overview.html)includes some really solid arguments for using them, not least:

Developers are likely to already understand the standard library classes, so there is no need to learn the API and behavior of ad-hoc concurrent components. Additionally, concurrent applications are far simpler to debug when they are built on reliable, well-tested components.

This [question on SO](https://stackoverflow.com/questions/1237980/java-5-concurrency-book-recommendations) asks about a good book, to which the immediate answer is JCiP. If you haven't already, get yourself a copy. The comprehensive approach to concurrency presented there goes well beyond this question, and will save you a lot of heartache in the long run.

[ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) provides many advantages compared to plain threads

1. You can create/manage/control life cycle of Threads & optimize thread creation cost overheads
2. You can control processing of tasks ( Work Stealing, ForkJoinPool, invokeAll) etc.
3. You can schedule tasks in Future time
4. You can monitor the progress and health of threads

Even for a single Thread, I prefer to use Executors.newFixedThreadPool(1);

Have a look at related SE questions:

Q: The difference between Executors.newSingleThreadExecutor().execute(command) and new Thread(command).start();

One noticeable difference, is when you run new Thread(someRunnable).start(); when the runnable is finished the thread will die quietly.

The Executor though will persist until you shut it down. So running Executors.newSingleThreadExecutor().execute(command) When you think your application or the JVM may be finished the Executor may still be running in a background thread.

Ans2: Behaviourally, pretty much nothing.

However, once you have an Executor instance, you can submit multiple tasks to it, and have them executed one after another. You can't do that simply with a raw Thread.

Q:

We have three different multi threading techniques in java - **Fork/Join pool, Executor Service & CountDownLatch**

**Fork/Join pool** (<http://www.javacodegeeks.com/2011/02/java-forkjoin-parallel-programming.html>)

The Fork/Join framework is designed to make divide-and-conquer algorithms easy to parallelize. That type of algorithms is perfect for problems that can be divided into two or more sub-problems of the same type. They use recursion to break down the problem to simple tasks until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem

**ExecutorService** is an interface that extends Executor class and represents an asynchronous execution. It provides us mechanisms to manage the end and detect progress of the asynchronous tasks.

invokeAll() : Executes the given tasks, returning a list of Futures holding their status and results when all complete. Future.isDone() is true for each element of the returned list.

**CountDownLatch**:(<http://examples.javacodegeeks.com/core-java/util/concurrent/countdownlatch-concurrent/java-util-concurrent-countdownlatch-example/>)

CountDownLatch is used in synchronisation to allow one or more threads to wait until a set of operations being performed in other threads completes.

**My assumption:**

In both these alternatives, final result will be known only after completion of all tasks/threads.

**Are these three alternatives complimentary or supplementary to each other**?

Ans:

After research on various multi threading frameworks for past 3 months , I have found answer to question.

[ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html)

It is simple and easy to use with limited control. You can use it

1. To start parallel independent tasks with out Waiting
2. Wait for completion of all your tasks

I prefer this one when number of Callable/Runnable tasks are small in number and piling of tasks in unbounded queue does not cause pile-up in memory & degrade the performance of the system.

It hides low level details of ThreadPoolExecutor. It does not allow playing with other parameters ( Bounded Queue, Rejection Handler etc. to fine tune the performance) as in ThreadPoolExectuor.

[ThreadPoolExecutor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html)

ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,

TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,

RejectedExecutionHandler handler)

It provides more control to you. Apart from setting minimum and maximum threads, you can set queue size and make BlockingQueue is bounded.

You can come up with your own thread factory if you need below features

1. To set a more descriptive thread name
2. To set thread daemon status
3. To set thread priority

If your application is constrained by number of pending Runnable/Callable tasks, you will use bounded queue by setting the max capacity. Once the queue reaches maximum capacity, you can define RejectionHandler. Java provides four types of Rejection Handler [policies](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html).

1. In the default ThreadPoolExecutor.AbortPolicy, the handler throws a runtime RejectedExecutionException upon rejection.
2. In ThreadPoolExecutor.CallerRunsPolicy, the thread that invokes execute itself runs the task. This provides a simple feedback control mechanism that will slow down the rate that new tasks are submitted.
3. In ThreadPoolExecutor.DiscardPolicy, a task that cannot be executed is simply dropped.
4. In ThreadPoolExecutor.DiscardOldestPolicy, if the executor is not shut down, the task at the head of the work queue is dropped, and then execution is retried (which can fail again, causing this to be repeated.)

[CountDownLatch](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountDownLatch.html)

CountDownLatch : This framework allows a java thread to wait until other set of threads completes their tasks.

Use cases:

1. Achieving Maximum Parallelism: Sometimes we want to start a number of threads at the same time to achieve maximum parallelism
2. Wait for N threads to complete before start of executing other code block
3. Deadlock detection.

More details are listed in this [article](http://howtodoinjava.com/2013/07/18/when-to-use-countdownlatch-java-concurrency-example-tutorial/)

[ForkJoinPool](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html)

The ForkJoinPool is similar to the Java ExecutorService but with one difference. The ForkJoinPool makes it easy for tasks to split their work up into smaller tasks which are then submitted to the ForkJoinPool too. Task stealing happens in ForkJoinPool when free worker threads steal tasks from busy worker thread queue.

public ForkJoinPool(int parallelism,

ForkJoinPool.ForkJoinWorkerThreadFactory factory,

Thread.UncaughtExceptionHandler handler,

boolean asyncMode)

Creates a ForkJoinPool with the given parameters.

Parameters:

parallelism - the parallelism level. For default value, use Runtime.availableProcessors().

factory - the factory for creating new threads. For default value, use defaultForkJoinWorkerThreadFactory.

handler - the handler for internal worker threads that terminate due to unrecoverable errors

asyncMode - if true, establishes local first-in-first-out scheduling mode for forked tasks that are never joined.

Regarding main query:

You can use *ExecutorService.invokeAll()* or *CountDownLatch* framework or *ForkJoinPool* . All these frameworks are complimentary to each other varying of granularity to control the execution of tasks from high level to low level.

Q: How to wait for all threads to finish, using ExecutorService?

Ans1:

Basically on an [ExecutorService](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) you call [shutdown()](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#shutdown--) and then [awaitTermination()](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#awaitTermination-long-java.util.concurrent.TimeUnit-):

ExecutorService taskExecutor = Executors.newFixedThreadPool(4);

while(...) {

taskExecutor.execute(new MyTask());

}

taskExecutor.shutdown();

try {

taskExecutor.awaitTermination(Long.MAX\_VALUE, TimeUnit.NANOSECONDS);

} catch (InterruptedException e) {

...

}

Ans: 2

In Java8 you can do it with [CompletableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html):

ExecutorService es = Executors.newFixedThreadPool(4);

List<Runnable> tasks = getTasks();

CompletableFuture<?>[] futures = tasks.stream()

.map(task -> CompletableFuture.runAsync(task, es))

.toArray(CompletableFuture[]::new);

CompletableFuture.allOf(futures).join();

es.shutdown();

Ans3: [ExecutorService.invokeAll()](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#invokeAll-java.util.Collection-) does it for you.

ExecutorService taskExecutor = Executors.newFixedThreadPool(4);

List<Callable<?>> tasks; // your tasks

// invokeAll() returns when all tasks are complete

List<Future<?>> futures = taskExecutor.invokeAll(tasks);

Ans3:

Use a [CountDownLatch](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountDownLatch.html):

CountDownLatch latch = new CountDownLatch(totalNumberOfTasks);

ExecutorService taskExecutor = Executors.newFixedThreadPool(4);

while(...) {

taskExecutor.execute(new MyTask());

}

try {

latch.await();

} catch (InterruptedException E) {

// handle

}

and within your task (enclose in try / finally)

latch.countDown();

Q: What does this thread join code mean?

In this code, what does the two joins and break mean? t1.join() causes t2 to stop until t1terminates?

Thread t1 = new Thread(new EventThread("e1"));

t1.start();

Thread t2 = new Thread(new EventThread("e2"));

t2.start();

while (true) {

try {

t1.join();

t2.join();

break;

} catch (InterruptedException e) {

e.printStackTrace();

}

}

Ans:

To quote from the [Thread.join() method javadocs](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.html#join%28%29):

join() Waits for this thread to die.

There is a thread that is running your example code which is probably the [main thread](http://www.go4expert.com/articles/main-thread-java-t4178/).

1. The main thread creates and starts the t1 and t2 threads. The two threads start running in parallel.
2. The main thread calls t1.join() to wait for the t1 thread to finish.
3. The t1 thread completes and the t1.join() method returns in the main thread. Note that t1could already have finished before the join() call is made in which case the join() call will return immediately.
4. The main thread calls t2.join() to wait for the t2 thread to finish.
5. The t2 thread completes (or it might have completed before the t1 thread did) and the t2.join() method returns in the main thread.

It is important to understand that the t1 and t2 threads have been running **in parallel** but the main thread that started them needs to wait for them to finish before it can continue. That's a common pattern. Also, t1 and/or t2 could have finished before the main thread calls join() on them. If so then join() will not wait but will return immediately.

t1.join() means cause t2 to stop until t1 terminates?

No. The main thread that is calling t1.join() will stop running and wait for the t1 thread to finish. The t2 thread is running in parallel and is not affected by t1 or the t1.join() call at all.

In terms of the try/catch, the join() throws InterruptedException meaning that the main thread that is calling join() may itself be interrupted by another thread.

while (true) {

Having the joins in a while loop is a strange pattern. Typically you would do the first join and then the second join handling the InterruptedException appropriately in each case. No need to put them in a loop.

Ans2:

This is a **favorite Java interview** question.

Thread t1 = new Thread(new EventThread("e1"));

t1.start();

Thread e2 = new Thread(new EventThread("e2"));

t2.start();

while (true) {

try {

t1.join(); // 1

t2.join(); // 2 These lines (1,2) are in in public static void main

break;

}

}

t1.join() means, t1 says something like "**I want to finish first**". Same is the case with t2. No matter who started t1 or t2 thread (in this case the main method), main will wait until t1 and t2 finish their task.

However, an important point to note down, t1 and t2 themselves **can run in parallel irrespective of the join call sequence** on t1 and t2. It is the main/daemon thread that has to **wait**.

Q: what is difference between sleep method and yield method of multi threading?

Ans1:

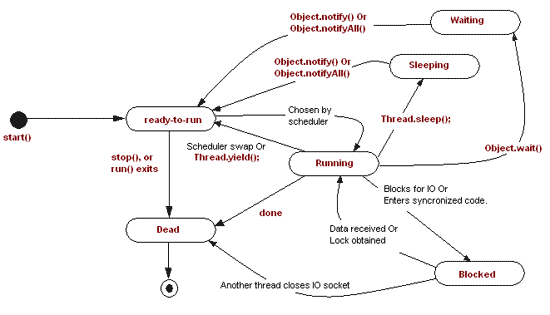
sleep() causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power saving mode).

yield() basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should. Otherwise, the current thread will continue to run.

Ans2:

[Sleep()](http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Thread.html#sleep%28long%29) causes the currently executing thread to sleep (temporarily cease execution).

[Yield()](http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Thread.html#yield%28%29) causes the currently executing thread object to temporarily pause and allow other threads to execute.



Read [this](http://www.bpurcell.org/blog/index.cfm?mode=entry&entry=934) for a good explanation of the topic.

Ans3:

We can prevent a thread from execution by using any of the 3 methods of Thread class:

1. yield()
2. join()
3. sleep()
4. yield() method pauses the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution. The yielded thread when it will get the chance for execution is decided by the thread scheduler whose behavior is vendor dependent.
5. join() If any executing thread t1 calls join() on t2 i.e; t2.join() immediately t1 will enter into waiting state until t2 completes its execution.
6. sleep() Based on our requirement we can make a thread to be in sleeping state for a specified period of time (hope not much explanation required for our favorite method).

# Q: JEP 180: Handle Frequent HashMap Collisions with Balanced Trees

Improve the performance of java.util.HashMap under high hash-collision conditions by using balanced trees rather than linked lists to store map entries. Implement the same improvement in the LinkedHashMap class.

This technique has already been implemented in the latest version of thejava.util.concurrent.ConcurrentHashMap class, which is also slated for inclusion in JDK 8 as part of [JEP 155](https://bugs.openjdk.java.net/browse/JDK-8046145). Portions of that code will be re-used to implement the same idea in the HashMap and LinkedHashMap classes. Only the implementations will be changed; no interfaces or specifications will be modified. Some user-visible behaviors, such as iteration order, will change within the bounds of their current specifications.

Q: Change to HashMap hash function in Java 8?

Question:

In java 8 java.util.Hashmap I noticed a change [from](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7-b147/java/util/HashMap.java#HashMap.hash%28int%29):

static int hash(int h) {

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

[to](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8-b132/java/util/HashMap.java#HashMap.hash%28java.lang.Object%29):

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

It appears from the code that the new function is a simpler XOR of the lower 16 bits with the upper 16 leaving the upper 16 bits unchanged, as opposed to several different shifts in the previous implementation, and from the comments that this is less effective at allocating the results of hash functions with a high number of collisions in lower bits to different buckets, but saves CPU cycles by having to do less operations.

The only thing I saw in the release notes was the [change](http://openjdk.java.net/jeps/180) from linked lists to balanced trees to store colliding keys (which I thought might have changed the amount of time it made sense to spend calculating a good hash), I was specifically interested in seeing if there was any expected performance impact from this change on large hash maps. Is there any information about this change, or does anyone with a better knowledge of hash functions have an idea of what the implications of this change might be (if any, perhaps I just misunderstood the code) and if there was any need to generate hash codes in a different way to maintain performance when moving to Java 8?

Ans:

As you noted: there is a significant performance improvement in HashMap in Java 8 as described in [JEP-180](http://openjdk.java.net/jeps/180). Basically, if a hash chain goes over a certain size, the HashMap will (where possible) replace it with a balanced binary tree. This makes the "worst case" behaviour of various operations O(log N) instead of O(N).

This doesn't directly explain the change to hash. However, I would hypothesize that the optimization in JEP-180 means that the performance hit due to a poorly distributed hash function is less important, and that the cost-benefit analysis for the hash method changes; i.e. the more complex version is less beneficial on average. (Bear in bind that when the key type's hashcode method generates high quality codes, then gymnastics in the complex version of the hash method are a waste of time.)

But this is only a theory. The real rationale for the hash change is most likely Oracle confidential.

Q: HashMap Java 8 implementation?

Question:

As per the following link document: [Java HashMap Implementation](https://dzone.com/articles/hashmap-performance)

I'm confused with the implementation of HashMap (or rather, an enhancement in HashMap). My queries are:

**Firstly**

static final int TREEIFY\_THRESHOLD = 8;

static final int UNTREEIFY\_THRESHOLD = 6;

static final int MIN\_TREEIFY\_CAPACITY = 64;

Why and how are these constants used? **I want some clear examples for this.** How they are achieving a performance gain with this?

**Secondly**

If you see the source code of HashMap in JDK, you will find the following static inner class:

static final class TreeNode<K, V> extends java.util.LinkedHashMap.Entry<K, V> {

HashMap.TreeNode<K, V> parent;

HashMap.TreeNode<K, V> left;

HashMap.TreeNode<K, V> right;

HashMap.TreeNode<K, V> prev;

boolean red;

TreeNode(int arg0, K arg1, V arg2, HashMap.Node<K, V> arg3) {

super(arg0, arg1, arg2, arg3);

}

final HashMap.TreeNode<K, V> root() {

HashMap.TreeNode arg0 = this;

while (true) {

HashMap.TreeNode arg1 = arg0.parent;

if (arg0.parent == null) {

return arg0;

}

arg0 = arg1;

}

}

//...

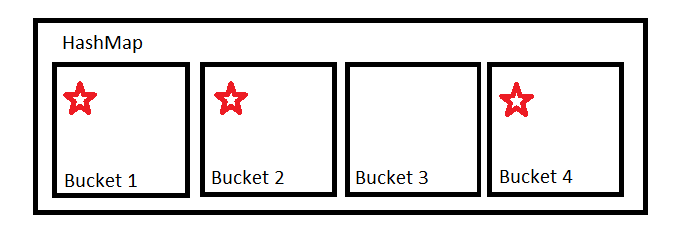
}

How is it used? **I just want an explanation of the algorithm**.

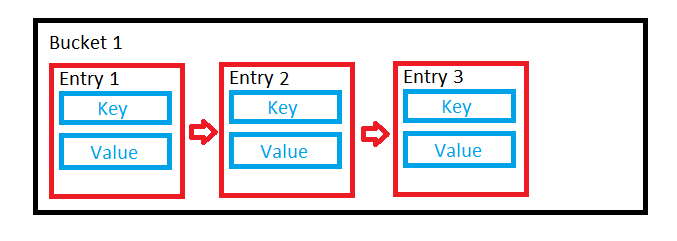
Ans:

HashMap contains a certain number of buckets. It uses hashCode to determine which bucket to put these into. For simplicity's sake imagine it as a modulus.

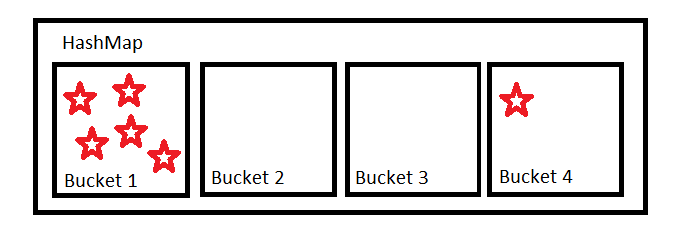
If our hashcode is 123456 and we have 4 buckets, 123456 % 4 = 0 so the item goes in the first bucket, Bucket 1.

[](https://i.stack.imgur.com/yg0zI.png)

If our hashcode function is good, it will provide an even distribution so all the buckets will be used somewhat equally. In this case, the bucket uses a linked list to store the values.

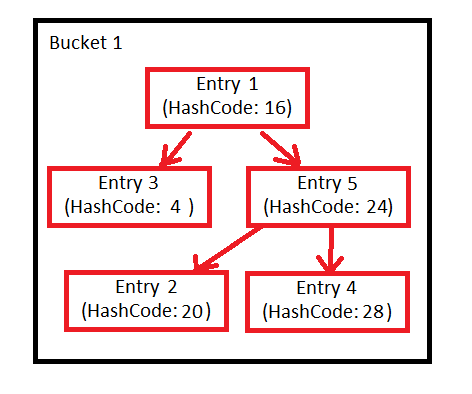
[](https://i.stack.imgur.com/4gDcD.png)

But you can't rely on people to implement good hash functions. People will often write poor hash functions which will result in a non-even distribution.

[](https://i.stack.imgur.com/laEaF.png)

The less even this distribution is, the further we're moving from O(1) operations and the closer we're moving towards O(n) operations.

The implementation of Hashmap tries to mitigate this by organising some buckets into trees rather than linked lists if the buckets becomes too large. This is what TREEIFY\_THRESHOLD = 8 is for. If a bucket contains more than eight items, it should become a tree.

[](https://i.stack.imgur.com/lMj8W.png)

The tree is first sorted by hash code. If the hash codes are the same, it uses the compareTo method of Comparable if the objects implement that interface, else the identity hash code.

If entries are removed from the map, the number of entries in the bucket might reduce such that this tree structure is no longer necessary. That's what the UNTREEIFY\_THRESHOLD = 6 is for. If the number of elements in a bucket drops below six, we might as well go back to using a linked list.

Finally, there is the MIN\_TREEIFY\_CAPACITY = 64.

When a hash map grows in size, it automatically resizes itself to have more buckets. If we have a small hash map, the likelihood of us getting very full buckets is quite high, because we don't that have many different buckets to put stuff into. It's much better to have a bigger hash map, with more buckets that are less full. This constant basically says not to start making buckets into trees if our hash map is very small - it should resize to be larger first instead.

To answer your question about the performance gain, these optimisations were added to improve the worst case. I'm only speculating but you would probably only see a noticeable performance improvement because of these optimisations if your hashCode function was not very good.

Q: Difference between <? super T> and <? extends T> in Java

### extends

The wildcard declaration of List<? extends Number> foo3 means that any of these are legal assignments:

List<? extends Number> foo3 = new ArrayList<Number>(); // Number "extends" Number (in this context)

List<? extends Number> foo3 = new ArrayList<Integer>(); // Integer extends Number

List<? extends Number> foo3 = new ArrayList<Double>(); // Double extends Number

1. **Reading** - Given the above possible assignments, what type of object are you guaranteed to read from List foo3:
   * You can read a Number because any of the lists that could be assigned to foo3 contain a Number or a subclass of Number.
   * You can't read an Integer because foo3 could be pointing at a List<Double>.
   * You can't read a Double because foo3 could be pointing at a List<Integer>.
2. **Writing** - Given the above possible assignments, what type of object could you add to List foo3 that would be legal for **all** the above possible ArrayList assignments:
   * You can't add an Integer because foo3 could be pointing at a List<Double>.
   * You can't add a Double because foo3 could be pointing at a List<Integer>.
   * You can't add a Number because foo3 could be pointing at a List<Integer>.

*You can't add any object to*List<? extends T>*because you can't guarantee what kind of*List*it is really pointing to, so you can't guarantee that the object is allowed in that*List*. The only "guarantee" is that you can only read from it and you'll get a*T*or subclass of*T*.*

### super

Now consider List <? super T>.

The wildcard declaration of List<? super Integer> foo3 means that any of these are legal assignments:

List<? super Integer> foo3 = new ArrayList<Integer>(); // Integer is a "superclass" of Integer (in this context)

List<? super Integer> foo3 = new ArrayList<Number>(); // Number is a superclass of Integer

List<? super Integer> foo3 = new ArrayList<Object>(); // Object is a superclass of Integer

1. **Reading** - Given the above possible assignments, what type of object are you guaranteed to receive when you read from List foo3:
   * You aren't guaranteed an Integer because foo3 could be pointing at a List<Number> or List<Object>.
   * You aren't guaranteed a Number because foo3 could be pointing at a List<Object>.
   * The **only** guarantee is that you will get an instance of an Object or subclass of Object(but you don't know what subclass).
2. **Writing** - Given the above possible assignments, what type of object could you add to List foo3 that would be legal for **all** the above possible ArrayList assignments:
   * You can add an Integer because an Integer is allowed in any of above lists.
   * You can add an instance of a subclass of Integer because an instance of a subclass of Integer is allowed in any of the above lists.
   * You can't add a Double because foo3 could be pointing at an ArrayList<Integer>.
   * You can't add a Number because foo3 could be pointing at an ArrayList<Integer>.
   * You can't add an Object because foo3 could be pointing at an ArrayList<Integer>.

### PECS

Remember *PECS*: **"Producer Extends, Consumer Super"**.

* **"Producer Extends"** - If you need a List to produce T values (you want to read Ts from the list), you need to declare it with ? extends T, e.g. List<? extends Integer>. But you cannot add to this list.
* **"Consumer Super"** - If you need a List to consume T values (you want to write Ts into the list), you need to declare it with ? super T, e.g. List<? super Integer>. But there are no guarantees what type of object you may read from this list.
* If you need to both read from and write to a list, you need to declare it exactly with no wildcards, e.g. List<Integer>.

### Example

Note [this example from the Java Generics FAQ](http://www.angelikalanger.com/GenericsFAQ/FAQSections/TypeArguments.html#FAQ103). Note how the source list src (the producing list) uses extends, and the destination list dest (the consuming list) uses super:

public class Collections {

public static <T> void copy(List<? super T> dest, List<? extends T> src) {

for (int i = 0; i < src.size(); i++)

dest.set(i, src.get(i));

}

}

Also see [How can I add to List<? extends Number> data structures?](https://stackoverflow.com/questions/2776975/how-can-i-add-to-list-extends-number-data-structures/2777297#2777297)

Ans2:

The wildcards introduce restrictions in how the collection can be used.

For example, with List<? extends Number>, I can't add new elements to the list. This is because all I know is that the list is some kind of subtype of Number, but I don't know what that actual subtype is (so how could I know what to add?). For example, take the following code:

public void doSomethingWith(List<? extends Number> numbers) {

numbers.add(Integer.valueOf(0)); // Won't compile

}

This won't compile because both of *these* method calls are legal:

doSomethingWith(new ArrayList<Integer>());

doSomethingWith(new ArrayList<Double>());

What you *can* do is *read* elements from the list:

// This will all compile

public void doSomethingWith(List<? extends Number> numbers) {

for (Number number : numbers) {

// Do something with number

}

// OR

Number number = numbers.get(0);

// OR

Number number = numbers.remove(0);

}

Calls to methods like get will return some kind of Number, we know that for a fact because of the ? extends Number, so we can treat it like that for reading purposes.

On the other hand, List<? super Integer> has exactly the opposite result. I can no longer read from the list, but I can write to it. I know that whatever ? is, it will definitely be a super-class of Integer, so concrete types of the list will definitely accept Integer values. For example:

public void doSomethingWith(List<? super Integer> integers) {

integers.add(Integer.valueOf(0));

}

That code is completely legal. However, if you want to read from the list, the only way to do this is to use Object since anything else requires casting (which requires knowing its concrete type):

for (Object obj : integers)

// OR

Object obj = integers.get(0);

// OR

Object obj = integers.remove(0);

**What's Really Happening**

Here's what's actually happening. When you specify ? extends Number, you're making any method that *takes* elements as a parameter unusable. In fact, if you try to auto-complete code in Eclipse using Ctrl+Space on a List<? extends Number>, it shows null as the parameters' types in the add methods and the like. Meanwhile, all the methods that *return* elements are guaranteed to return at least some kind of Number, though you won't know exactly which subclass of Number it might actually be.

When you specify ? super Integer, you're making any method that *takes* elements as a parameter guarantee that they'll accept Integer values (and sub-classes of Integer as well). This allows you to call methods like add since you know they'll accept Integer types. Meanwhile, all methods that *return* elements are only guaranteed to return *something*, but we don't know what, so all the methods that return elements are only guaranteed to return Object.

PECS is an excellent acronym to remember this, it means "Producer Extends, Consumer Supers". This means that if you want your list to give you something, it's a producer, and you should use extends. If you want your list to accept things from you, it's a consumer, so you use super. See [this answer](https://stackoverflow.com/questions/2723397/java-generics-what-is-pecs) for more.

**But what if I have a wildcard with no bounds?**

It does both! <?> restricts you from calling methods that take the generic type as an argument *and*causes all the methods that return the generic type to return Object. This is because we have no idea what the type is whatsoever. For example, all of these assignments into a List<?> are legal:

List<?> list;

list = new ArrayList<Integer>();

list = new ArrayList<String>();

list = new ArrayList<MyClass>();

And so on.

Q: ConcurrentHashMap read and write locks

Question:

I am trying to find answer to these, but not able to find it on Google or in Java docs.

Case 1: in ConcurrentHashMap, suppose a thread t1 is reading from segment n, and at same another thread t2 want to write on the same segment n:

Question 1: will these two operations will be one after another, or they will execute simultaneously?

Case 2: in ConcurrentHashMap, suppose a thread t1 is writing on segment n, and at same another thread t2 want to read from the same segment n,

Question 2: will these two operations will be one after another, or they will execute simultaneously?

Ans:

I think javadoc answers both your questions:

Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove). Retrievals reflect the results of the most recently completed update operations holding upon their onset. For aggregate operations such as putAll and clear, concurrent retrievals may reflect insertion or removal of only some entries.

Segments are for update operations:

The allowed concurrency among update operations is guided by the optional concurrencyLevel constructor argument (default 16), which is used as a hint for internal sizing.

So, in short, reads are not blocked (it is implemented as reading volatile variables). Writes could block each other if they write in the same segment.

Q: Need simple explanation how “lock striping” works with ConcurrentHashMap

According to Java Concurrency in Practice, chapter 11.4.3 says:

Lock splitting can sometimes be extended to partition locking on a variablesized set of independent objects, in which case it is called lock striping. For example, the implementation of ConcurrentHashMap uses an array of 16 locks, each of which guards 1/16 of the hash buckets; bucket N is guarded by lock N mod 16.

I still have problems to understand and visualize the lock striping and buckets mechanism. Can someone explain this with good understanding words :)

Ans: The hash map is built on an array, where the hash function maps an object to an element in the underlying array. Let's say the underlying array has 1024 elements - ConcurrentHashMap actually turns this into 16 different sub-arrays of 64 elements, e.g. {0, 63}, {64, 127}, etc. Each sub-array has its own lock, so modifying the {0, 63} sub-array doesn't impact the {64, 127} sub-array - one thread can write to the first sub-array while another thread writes to the second sub-array.

Q: Where are static methods and static variables stored in Java?

Ans:

Static methods (in fact all methods) as well as static variables are stored in the PermGen section of the heap, since they are part of the reflection data (class related data, not instance related).

*Update for clarification*:

Note that only the variables and their technical values (primitives or references) are stored in PermGen space.

If your static variable is a reference to an object that object itself is stored in the normal sections of the heap (young/old generation or survivor space). Those objects (unless they are interal objects like classes etc.) are *not* stored in PermGen space.

Example:

static int i = 1; //the value 1 is stored in the permgen section

static Object o = new SomeObject(); //the reference(pointer/memory address) is stored in the permgen section, the object itself is not.

A word on garbage collection:

Do *not* rely on finalize() as it's not guaranteed to run. It is totally up to the JVM to decide when to run the garbage collector and what to collect, even if an object is elligible for garbage collection.

Of course you can set a static variable to null and thus remove the reference to the object on the heap but that doesn't mean the garbage collector *will* collect it (even if there are no more references).

Additionally finalize() is run only once, so you have to make sure it doesn't throw exceptions or otherwise prevent the object to be collected. If you halt finalization through some exception, finalize() won't be invoked on the same object a second time.

*A final note*: how code, runtime data etc. are stored depends on the JVM which is used, i.e. HotSpot might do it differently than JRockit and this might even differ between versions of the same JVM. The above is based on HotSpot for Java 5 and 6 (those are basically the same) since at the time of answering I'd say that most people used those JVMs. Due to major changes in the memory model as of Java 8, the statements above might not be true for Java 8 HotSpot - and I didn't check the changes of Java 7 HotSpot, so I *guess* the above is still true for that version, but I'm not sure here.

Q: **SOLID** principle of Object oriented Programming language?

[**Single responsibility principle**](https://en.wikipedia.org/wiki/Single_responsibility_principle)[**:**](https://en.wikipedia.org/wiki/SOLID#cite_note-6)

A [class](https://en.wikipedia.org/wiki/Class_(computer_science)) should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class).

[**Open/closed principle**](https://en.wikipedia.org/wiki/Open/closed_principle)**:**

"Software entities … should be open for extension, but closed for modification."

[**Liskov substitution principle**](https://en.wikipedia.org/wiki/Liskov_substitution_principle)**:**

"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also [design by contract](https://en.wikipedia.org/wiki/Design_by_contract).

[**Interface segregation principle**](https://en.wikipedia.org/wiki/Interface_segregation_principle)**:**

"Many client-specific interfaces are better than one general-purpose interface."

[**Dependency inversion principle**](https://en.wikipedia.org/wiki/Dependency_inversion_principle)**:**

One should "depend upon abstractions, [not] concretions."

# S.O.L.I.D Principles (java examples)

* **Single responsibility principle** - A class should have only one reason to change.
* **Open/Closed principle - Software** entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
* **Liskov Substitution Principle** - Child classes should never break the parent class type definitions.
* **Interface Segregation Principle** - No client should be forced to depend on methods it does not use.
* **Dependency inversion principle** - High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.

Q: PermGen elimination in JDK 8?

Reasons of ignoring these argument is permanent generation has been removed in HotSpot for JDK8 because of following drawbacks

* Fixed size at startup – difficult to tune.
* Internal Hotspot types were Java objects : Could move with full GC, opaque, not strongly typed and hard to debug, needed meta-metadata.
* Simplify full collections : Special iterators for metadata for each collector
* Want to deallocate class data concurrently and not during GC pause
* Enable future improvements that were limited by PermGen.

The Permanent Generation (PermGen) space has completely been removed and is kind of replaced by a new space called Metaspace. The consequences of the PermGen removal is that obviously the **PermSize and MaxPermSize JVM arguments are ignored** and you will never get a java.lang.OutOfMemoryError: PermGen error.

**Advantages of MetaSpace**

* Take advantage of Java Language Specification property : Classes and associated metadata lifetimes match class loader’s
* Per loader storage area – Metaspace
* Linear allocation only
* No individual reclamation (except for RedefineClasses and class loading failure)
* No GC scan or compaction
* No relocation for metaspace objects

**Metaspace Tuning**

The maximum metaspace size can be set using the -XX:MaxMetaspaceSize flag, and the default is unlimited, which means that only your system memory is the limit. The -XX:MetaspaceSize tuning flag defines the initial size of metaspace If you don’t specify this flag, the Metaspace will dynamically re-size depending of the application demand at runtime.

Change enables other optimizations and features in the future

* Application class data sharing
* Young collection optimizations, G1 class unloading
* Metadata size reductions and internal JVM footprint projects

There is improved GC performace also. [More detail](http://java-latte.blogspot.in/2014/03/metaspace-in-java-8.html)

Q: How to identify whether the linked list has loop?

private static boolean isCircularLinkedList(LinkedList<Integer> linkedList) {

boolean isCircular = Boolean.FALSE;

Set<Integer> circular = new HashSet<>();

for (Integer integer : linkedList) {

if (circular.contains(integer)) {

System.out.println("Circular Linked List: " + integer);

isCircular = Boolean.TRUE;

break;

} else {

circular.add(integer);

}

}

return isCircular;

}

Q: Java 7 new Features?

1. Try-with-resources statement
2. Underscores in numeric literals
3. Strings in switch
4. Binary literals
5. Multiple exception catching
6. Improved Type Inference for Generic Instance Creation
7. SafeVarargs

## Java SE 7 [Features and Enhancements](http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html) from JDK 7 Release Notes

This is the Java 7 new features summary from the [OpenJDK 7 features page](http://openjdk.java.net/projects/jdk7/features/):

vm JSR 292: Support for dynamically-typed languages (InvokeDynamic)

Strict class-file checking

lang JSR 334: Small language enhancements (Project Coin)

core Upgrade class-loader architecture

Method to close a URLClassLoader

Concurrency and collections updates (jsr166y)

i18n Unicode 6.0

Locale enhancement

Separate user locale and user-interface locale

ionet JSR 203: More new I/O APIs for the Java platform (NIO.2)

NIO.2 filesystem provider for zip/jar archives

SCTP (Stream Control Transmission Protocol)

SDP (Sockets Direct Protocol)

Use the Windows Vista IPv6 stack

TLS 1.2

sec Elliptic-curve cryptography (ECC)

jdbc JDBC 4.1

client XRender pipeline for Java 2D

Create new platform APIs for 6u10 graphics features

Nimbus look-and-feel for Swing

Swing JLayer component

Gervill sound synthesizer [NEW]

web Update the XML stack

mgmt Enhanced MBeans [UPDATED]

## Code examples for new features in Java 1.7

### **Try-with-resources statement**

this:

BufferedReader br = new BufferedReader(new FileReader(path));

try {

return br.readLine();

} finally {

br.close();

}

becomes:

try (BufferedReader br = new BufferedReader(new FileReader(path)) {

return br.readLine();

}

You can declare more than one resource to close:

try (

InputStream in = new FileInputStream(src);

OutputStream out = new FileOutputStream(dest))

{

// code

}

### **Underscores in numeric literals**

int one\_million = 1\_000\_000;

### **Strings in switch**

String s = ...

switch(s) {

case "quux":

processQuux(s);

// fall-through

case "foo":

case "bar":

processFooOrBar(s);

break;

case "baz":

processBaz(s);

// fall-through

default:

processDefault(s);

break;

}

### **Binary literals**

int binary = 0b1001\_1001;

### **Improved Type Inference for Generic Instance Creation**

Map<String, List<String>> anagrams = new HashMap<String, List<String>>();

becomes:

Map<String, List<String>> anagrams = new HashMap<>();

### **Multiple exception catching**

this:

} catch (FirstException ex) {

logger.error(ex);

throw ex;

} catch (SecondException ex) {

logger.error(ex);

throw ex;

}

becomes:

} catch (FirstException | SecondException ex) {

logger.error(ex);

throw ex;

}

### **SafeVarargs**

this:

@SuppressWarnings({"unchecked", "varargs"})

public static void printAll(List<String>... lists){

for(List<String> list : lists){

System.out.println(list);

}

}

becomes:

@SafeVarargs

public static void printAll(List<String>... lists){

for(List<String> list : lists){

System.out.println(list);

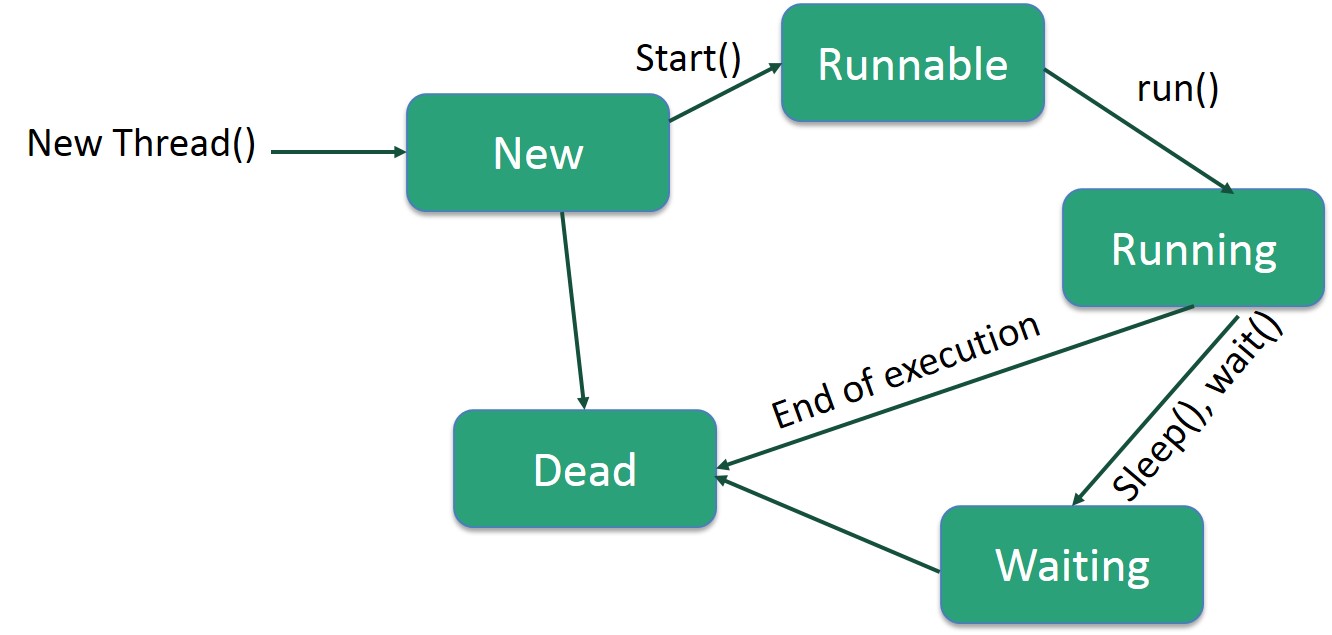
}

}

Q: Java 8 new features?

1. forEach() method in Iterable interface.
2. default and static methods in Interfaces.
3. Functional Interfaces and Lambda Expressions.
4. Java Stream API for Bulk Data Operations on Collections.
5. Java Time API.
6. Collection API improvements.
7. Concurrency API improvements.
8. Java IO improvements.

Thread Life Cycle:



Java Thread Above-mentioned stages are explained here:

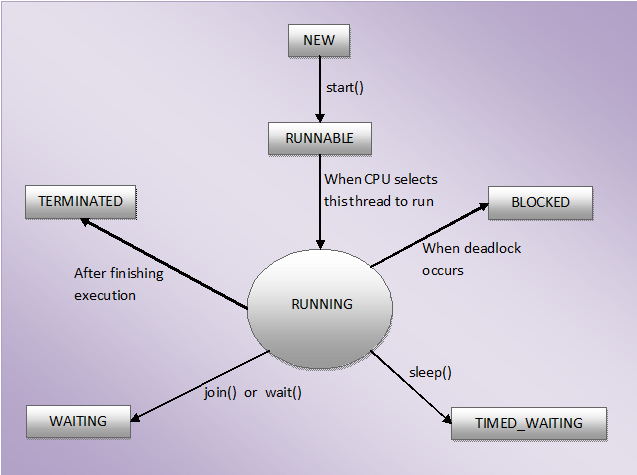
New: A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable: After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting: Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed waiting: A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated ( Dead ): A runnable thread enters the terminated state when it completes its task or otherwise terminates.



Q: Method Overloading:

If a class has more than one method with same name, with different types of arguments, is called Method Overloading. Return type will not considered

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw (String s) and draw (int i) are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Q: Method Overriding.

If a class has same method name, and same number of and same type of Arguments of its super class is called method overriding.

An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass overrides the superclass's method.

The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed.

Return type:

Parent return type: Number  
Child can be: Long/Integer/Double

If Parent return type: Long  
Child can be: Long

Q: What is a serialVersionUID and why should I use it?

The docs for [java.io.Serializable](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html) are probably about as good an explanation as you'll get:

The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in anInvalidClassException. A serializable class can declare its own serialVersionUID explicitly by declaring a field named "serialVersionUID" that must be static, final, and of type long:

ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;

If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is *strongly recommended* that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization. Therefore, to guarantee a consistent serialVersionUID value across different java compiler implementations, a serializable class must declare an explicit serialVersionUID value. It is also strongly advised that explicit serialVersionUID declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class--serialVersionUID fields are not useful as inherited members.

Q: How does a Java HashMap handle different objects with the same hash code?

down voteaccepted

A hashmap works like this (this is a little bit simplified, but it illustrates the basic mechanism):

It has a number of "buckets" which it uses to store key-value pairs in. Each bucket has a unique number - that's what identifies the bucket. When you put a key-value pair into the map, the hashmap will look at the hash code of the key, and store the pair in the bucket of which the identifier is the hash code of the key. For example: The hash code of the key is 235 -> the pair is stored in bucket number 235. (Note that one bucket can store more then one key-value pair).

When you lookup a value in the hashmap, by giving it a key, it will first look at the hash code of the key that you gave. The hashmap will then look into the corresponding bucket, and then it will compare the key that you gave with the keys of all pairs in the bucket, by comparing them with equals().

Now you can see how this is very efficient for looking up key-value pairs in a map: by the hash code of the key the hashmap immediately knows in which bucket to look, so that it only has to test against what's in that bucket.

Looking at the above mechanism, you can also see what requirements are necessary on the hashCode() and equals() methods of keys:

* If two keys are the same (equals() returns true when you compare them), their hashCode()method must return the same number. If keys violate this, then keys that are equal might be stored in different buckets, and the hashmap would not be able to find key-value pairs (because it's going to look in the same bucket).
* If two keys are different, then it doesn't matter if their hash codes are the same or not. They will be stored in the same bucket if their hash codes are the same, and in this case, the hashmap will use equals() to tell them apart.

Q: Why does Java have transient fields?

The transient keyword in Java is used to indicate that a field should not be serialized.

Variables may be marked transient to indicate that they are not part of the persistent state of an object.

Q: Do you ever use the volatile keyword in Java?

volatile has semantics for memory visibility. Basically, the value of a volatile field becomes visible to all readers (other threads in particular) after a write operation completes on it. Without volatile, readers could see some non-updated value.

To answer your question: Yes, I use a volatile variable to control whether some code continues a loop. The loop tests the volatile value and continues if it is true. The condition can be set to false by calling a "stop" method. The loop sees false and terminates when it tests the value after the stop method completes execution.

The book "[Java Concurrency in Practice](http://jcip.net/)," which I highly recommend, gives a good explanation of volatile. This book is written by the same person who wrote the IBM article that is referenced in the question (in fact, he cites his book at the bottom of that article). My use of volatile is what his article calls the "pattern 1 status flag."

If you want to learn more about how [volatile](http://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.3.1.4) works under the hood, read up on [the Java memory model](http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html). If you want to go beyond that level, check out a good computer architecture book like [Hennessy & Patterson](https://www.elsevier.com/books/computer-architecture/hennessy/978-0-12-383872-8) and read about cache coherence and cache consistency.

Q: Reverse a String in java?

**public** **class** ReverseString {

/\*\*

\* **@param** args

\*/

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

String string = "Dogs hates cats";

System.***out***.println("String Builder reverse..>" + **new** StringBuilder(string).reverse());

// char[] c=new char[string.length()];

**char**[] c = string.toCharArray();

**char** temp;

**for** (**int** i = 0; i < c.length / 2; i++) {

temp = c[i];

c[i] = c[c.length - 1 - i];

c[c.length - 1 - i] = temp;

}

System.***out***.println("Reverse String String...>" + **new** String(c));

}

}

Q: Encapsulation vs Abstraction?

Abstraction:

The process of abstraction in Java is used to hide certain details and only show the essential features of the object. In other words, it deals with the outside view of an object (interface). The only good example i see for this across different sites is interface.

Encapsulation:

Its basically about hiding the state of object with the help of modifiers like private,public,protected etc. we expose the state thru public methods only if require.

**Encapsulation** is a strategy used as part of abstraction. Encapsulation refers to the state of objects - objects encapsulate their state and hide it from the outside; outside users of the class interact with it through its methods, but cannot access the classes state directly. So the class *abstracts* away the implementation details related to its state.

**Abstraction** is a more generic term, it can also be achieved by (amongst others) subclassing. For example, the interface List in the standard library is an abstraction for a sequence of items, indexed by their position, concrete examples of a List are an ArrayList or a LinkedList. Code that interacts with a List abstracts over the detail of which kind of a list it is using.

Abstraction is often not possible without hiding underlying state by encapsulation - if a class exposes its internal state, it can't change its inner workings, and thus cannot be abstracted.

**Abstraction** is the concept of describing something in simpler terms, i.e abstracting away the details, in order to focus on what is important (This is also seen in abstract art, for example, where the artist focuses on the building blocks of images, such as colour or shapes). The same idea translates to OOP by using an inheritance hierarchy, where more abstract concepts are at the top and more concrete ideas, at the bottom, build upon their abstractions. At its most abstract level there is no implementation details at all and perhaps very few commonalities, which are added as the abstraction decreases.

As an example, at the top might be an interface with a single method, then the next level, provides several abstract classes, which may or may not fill in some of the details about the top level, but branches by adding their own abstract methods, then for each of these abstract classes are concrete classes providing implementations of all the remaining methods.

**Encapsulation** is a *technique*. It may or may not be for aiding in abstraction, but it is certainly about information hiding and/or organisation. It demands data and functions be grouped in some way - of course good OOP practice demands that they should be grouped by abstraction. However, there are other uses which just aid in maintainability etc.

**Encapsulation** is a way to achieve [*"information hiding"*](http://en.wikipedia.org/wiki/Information_hiding) so, following your example, you don't *"need to know the internal working of the mobile phone to operate"* with it. You have an *interface* to use the device behaviour without knowing implementation details.

**Abstraction** on the other side, can be explained as the capability to use the same *interface* for different objects. Different implementations of the same interface can exist. Details are hidden by *encapsulation*.

**Abstraction** : you'll never buy a "device", but always buy something more specific : iPhone, GSII, Nokia 3310... Here, iPhone, GSII and N3310 are concrete things, device is abstract.

**Encapsulation** : you've got several devices, all of them have got an USB port. You don't know what kind of printed circuit there's back, you just have to know you'll be able to plug an USB cable onto.

Abstraction is a concept, which is allowed by encapsulation. My exemple wasn't the best one (there's no real link between the two blocks).

You can do encapsulation without using abstraction, but if you wanna use some abstraction in your projets, you'll need encapsulation.

Q: Abstract class vs Interface?

I will give you an example first:

public interface LoginAuth{

public String encryptPassword(String pass);

public void checkDBforUser();

}

Now suppose you have 3 databases in your application. Then each and every implementation for that database needs to define the above 2 methods:

public class DBMySQL implements LoginAuth{

// Needs to implement both methods

}

public class DBOracle implements LoginAuth{

// Needs to implement both methods

}

public class DBAbc implements LoginAuth{

// Needs to implement both methods

}

But what if encryptPassword() is not database dependent, and it's the same for each class? Then the above would not be a good approach.

Instead, consider this approach:

public abstract class LoginAuth{

public String encryptPassword(String pass){

// Implement the same default behavior here

// that is shared by all subclasses.

}

// Each subclass needs to provide their own implementation of this only:

public abstract void checkDBforUser();

}

Now in each child class, we only need to implement one method - the method that is database dependent.

I tried my best and Hope this will clear your doubts.

Q: Can An Interface have implementation?

Ans: till java 7, is not possible, from JDK 8, yes it is possible.

default and public static methods may have implementations  
and non-static public methods of interface do not have, implementation

Q: Abstract class vs Interface?

I was asked a question, I wanted to get my answer reviewed here.

**Q:** In which scenario it is more appropriate to extend an abstract class rather than implementing the interface(s)?

**A:** If we are using template method design pattern.

Am I correct ?

I am sorry if I was not able to state the question clearly.  
I know the basic difference between abstract class and interface.

1) use abstract class when the requirement is such that we need to implement the same functionality in every subclass for a specific operation (implement the method) and different functionality for some other operations (only method signatures)

2) use interface if you need to put the signature to be same (and implementation different) so that you can comply with interface implementation

3) we can extend max of one abstract class, but can implement more than one interface

Choosing between these two really depends on what you want to do, but luckily for us, Erich Gamma can help us a bit.

As always there is a trade-off, an interface gives you freedom with regard to the base class, an abstract class gives you the **freedom to add new methods later**. – Erich Gamma

You **can’t go and change an Interface without having to change a lot of other things** in your code, so the only way to avoid this would be to create a whole new Interface, which might not always be a good thing.

**When To Use Interfaces**

An interface allows somebody to start from scratch to implement your interface or implement your interface in some other code whose original or primary purpose was quite different from your interface. To them, your interface is only incidental, something that have to add on to the their code to be able to use your package. The disadvantage is every method in the interface must be public. You might not want to expose everything.

**When To Use Abstract classes**

An abstract class, in contrast, provides more structure. It usually defines some default implementations and provides some tools useful for a full implementation. The catch is, code using it must use your class as the base. That may be highly inconvenient if the other programmers wanting to use your package have already developed their own class hierarchy independently. In Java, a class can inherit from only one base class.

**When to Use Both**

You can offer the best of both worlds, an interface and an abstract class. Implementors can ignore your abstract class if they choose. The only drawback of doing that is calling methods via their interface name is slightly slower than calling them via their abstract class name.

Q: What is the difference between a .war and .ear file?

Ans: A **WAR (Web Archive)** is a module that gets loaded into a [Web container](https://en.wikipedia.org/wiki/Web_container) of a [Java Application Server](https://en.wikipedia.org/wiki/Application_server#Java_application_servers). A Java Application Server has two containers (runtime environments) - one is a Web container and the other is a EJB container.

The **Web container** hosts Web applications based on JSP or the Servlets API - designed specifically for web request handling - so more of a request/response style of distributed computing. A Web container requires the Web module to be packaged as a **WAR file** - that is a special JAR file with a web.xml file in the WEB-INF folder.

An **EJB container** hosts Enterprise java beans based on the EJB API designed to provide extended business functionality such as declarative transactions, declarative method level security and multiprotocol support - so more of a RPC style of distributed computing. EJB containers require EJB modules to be packaged as **JAR files** - these have a ejb-jar.xml file in the META-INF folder.

**Enterprise applications** may consist of one or more modules that can either be Web modules (packaged as a WAR file) or EJB modules (packaged as a JAR file) or both of them. Enterprise applications are packaged as **EAR files** - these are special JAR files containing an application.xml file in the META-INF folder.

Basically **EAR files** are a superset containing *WAR files* and *JAR files*. Java Application Servers allow deployment of standalone web modules in a WAR file, though internally they create EAR files as a wrapper around WAR files. Standalone web containers such as Tomcat and Jetty do not support EAR files - these are not full fledged Application servers. Web applications in these containers are to be deployed as WAR files only.

In application servers - EAR files contain configurations such as application security role mapping, EJB reference mapping and context root url mapping of web modules.

Apart from Web modules and EJB modules EAR files can also contain connector modules packaged as RAR files and Client modules packaged as JAR files.

Q: How are “mvn clean package” and “mvn clean install” different?

|  |  |
| --- | --- |
| test | run tests using a suitable unit testing framework.  These tests should not require the code be packaged or deployed. |

|  |  |
| --- | --- |
| package | take the compiled code and package it in its distributable format, such as a JAR. |
| install | install the package into the local repository, for use as a dependency in other projects locally. |
| deploy | done in an integration or release environment,  copies the final package to the remote repository for sharing with other developers and projects. |

Well, both will clean. That means they'll remove the target folder. The real question is what's the difference between package and install?

package will compile your code and also package it. For example, if your pom says the project is a jar, it will create a jar for you when you package it and put it somewhere in the target directory (by default).

install will compile and package, but it will also put the package in your local repository. This will make it so other projects can refer to it and grab it from your local repository.

#### A Build Lifecycle is Made Up of Phases

Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

For example, the default lifecycle comprises of the following phases (for a complete list of the lifecycle phases, refer to the [Lifecycle Reference](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference)):

* validate - validate the project is correct and all necessary information is available
* compile - compile the source code of the project
* test - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
* package - take the compiled code and package it in its distributable format, such as a JAR.
* verify - run any checks on results of integration tests to ensure quality criteria are met
* install - install the package into the local repository, for use as a dependency in other projects locally
* deploy - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the default lifecycle. Given the lifecycle phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

JQuery:

Q: What is the difference between window.onload and document.ready()

What are the differences between JavaScript's [window.onload](https://developer.mozilla.org/en/docs/Web/API/GlobalEventHandlers/onload) and jQuery's [$(document).ready()](https://api.jquery.com/ready/)method?

The ready event occurs after the HTML document has been loaded, while the onload event occurs later, when all content (e.g. images) also has been loaded.

The onload event is a standard event in the DOM, while the ready event is specific to jQuery. The purpose of the ready event is that it should occur as early as possible after the document has loaded, so that code that adds functionality to the elements in the page doesn't have to wait for all content to load.

$(document).ready(function() {

// Executes when the HTML document is loaded and the DOM is ready

alert("Document is ready");

});

// .load() method deprecated from jQuery 1.8 onward

$(window).on("load", function() {

// Executes when complete page is fully loaded, including

// all frames, objects and images

alert("Window is loaded");

});

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>