Threads

Table of Contents

[Multithreading - Advanced Concepts 3](#_Toc516039865)

[Chapter 1: Introduction to Threads 4](#_Toc516039866)

[Chapter 2: What Is a Thread? 6](#_Toc516039867)

Multithreading - Advanced Concepts

Java Threads are a very powerful and most importantly a very complicated topic. Every experienced Java Programmer would have used them at some point of time in his/her career and if you are a novice Java Programmer then its a good idea to learn them...   
  
The Purpose of these topics explained below is to cover the advanced topics related to Multithreading and make you an expert in the same.

### Chapter 1: Introduction to Threads

Threads are an important and interesting aspect of the Java Programming Language. If you are an experienced Java Programmer, you are invariably expected to know the intricacies of multi-threading. Well, this series of chapters is aimed at helping you understand this interesting and complicated concept.   
  
So, lets get started!!!   
  
**Why Use Threads?**  
  
The notion of threading is so ingrained in Java that it's almost impossible to write even the simplest programs in Java without creating and using threads. And many of the classes in the Java API are already threaded, so often you are using multiple threads without realizing it. 

**The biggest question you might have is “Why use Threads?”**

Well, my friend, it is not so simple to explain the reason in a couple of lines. Maybe a 100 page thesis would sound more appropriate. But still, am gonna try explaining in a short and concise manner.   
  
First and foremost – threads increase performance. Given the processing power of the modern day processors, irrespective of how much code you write, the processor is going to finish running them in a jiffy and will remain underutilized almost 99% or more times. By creating multiple threads in your program, you are trying to utilize the processor effectively trying to keep it as little time as possible “idle”   
  
Some of the significant reasons for using Threads are explained in the subsequent sections of this article.   
  
**Nonblocking I/O**  
  
In Java, as in most programming languages, when you try to get input from the user, you execute a read() method specifying the user's terminal (System.in in Java). When the program executes the read() method, the program typically waits until the user types at least one character before it continues and executes the next statement. This type of I/O is called blocking I/O : the program blocks until some data is available to satisfy the read() method.  
  
This type of behavior is often undesirable. If you're reading data from a network socket, that data is often not available when you want to read it: the data may have been delayed in transit over the network, or you may be reading from a network server that sends data only periodically. If the program blocks when it tries to read from the socket, it's unable to do anything else until the data is actually available. If the program has a user interface that contains a button and the user presses the button while the program is executing the read() method, nothing happens: the program is unable to handle the mouse events and execute the event processing method associated with the button. This can be very frustrating for the user, who thinks the program has hung.   
  
This kind of situation is where Threads in Java look the most promising solution.   
  
**Independent Tasks**  
  
A Java program is often called on to perform independent tasks. In the simplest case, a single applet may perform two independent animations for a web page. A more complex program would be a calculation server that performs calculations on behalf of several clients simultaneously. In either case, while it is possible to write a single-threaded program to perform multiple tasks, it's easier and more elegant to place each task in its own thread.   
  
**Parallelizable Algorithms**  
  
With the advent of virtual machines that can use multiple CPUs simultaneously, Java has become a useful platform for developing programs that use algorithms that can be parallelized; that is, running one iteration of the loop on one CPU while another iteration of the loop is simultaneously running on another CPU. Dependencies between the data that each iteration of the loop needs may prohibit a particular loop from being parallelized, and there may be other reasons why a loop should not be parallelized. But for many programs with CPU-intensive loops, parallelizing the loop greatly speeds up the execution of the program when it is run on a machine with multiple processors.   
  
Now that we know why and how threads are useful for us as programmers, lets dig deep into the magical world of Threads!!!

Chapter 2: What Is a Thread?

A thread is an application task that is executed by a host computer. The notion of a task should be familiar to you even if the terminology is not.   
  
Suppose you have a Java program to do some task:  
  
public class DoSomething{  
public static void main(String[] args) {  
Task 1;  
Task 2;  
Task 3;  
…  
Task N;  
}  
}  
  
When your computer runs this application, it executes a sequence of commands. At an abstract level, that list of commands looks like this:  
  
• Do Task 1  
• Do Task 2  
• Continue on until you reach Task N  
• Exit the Program  
  
Behind the scenes, what happens is somewhat more complicated since the instructions that are executed are actually machine-level assembly instructions; each of our logical steps requires many machine instructions to execute. But the principle is the same: an application is executed as a series of instructions. The execution path of these instructions is a thread.   
  
Consequently, every computer program has at least one thread: the thread that executes the body of the application. In a Java application, that thread is called the main thread, and it begins executing statements with the first statement of the main() method of your class. In other programming languages, the starting point may be different, and the terminology may be different, but the basic idea is the same.

**Trivia**:   
For Java applications, execution begins with the main() method of the class being run. What about other Java programs?  
In applets, servlets, and other J2EE programs, execution still begins with the main() method of the program, but in this case, the main() method belongs to the Java plug-in or J2EE container. Those containers then call your code through predetermined, well-known locations. An applet is called via its init() and start() methods; a servlet is called through its doGet() and doPost() methods, and so on.  
In any case, the procedure is the same: execution of your code begins with the first statements and proceeds by a single thread sequentially.

In a Java program, it so happens that every program has more than one thread. Many of these are threads that developers are unaware of, such as threads that perform garbage collection and compile Java bytecodes into machine-level instructions. In a graphical application, other threads handle input from the mouse and keyboard and play audio. Your Java application is highly threaded, whether you program & code additional threads into it or not.  
  
Returning to our example, let's suppose that we wrote a program that performed two tasks: one calculated the factorial of a number and one calculated the square root of that number. These are two separate tasks, and so you could choose to write them as two separate threads. Now how would your application run?  
  
The answer to that depends on the conditions under which the application is run. The Java virtual machine now has two distinct lists of instructions to execute. One list calculates the factorial of a number, and the other list calculates the square root of the number. The Java virtual machine executes both of these lists almost simultaneously.  
  
Although you may not have thought about it in these terms, this situation should also be familiar to you from the computer on which you normally do your work. The program you use to read your email is a list of instructions that the computer executes. So too is the program that you use to listen to music. You're able to read email and listen to music at the same time because the computer executes both lists of instructions at about the same time.  
  
In fact, what happens is that the computer executes a handful of instructions from the email application and then executes a handful of instructions from the music program. It continues this procedure, switching back and forth between lists of instructions, and it does that quickly enough so that both programs appear to be executing at the same time. Quickly enough, in fact, that there are no gaps in the music.  
  
If you happen to have more than one CPU on your computer, the lists of instructions can execute at exactly the same time: one list can execute on each CPU. But multiple CPUs aren't necessary to give the appearance of simultaneous execution or to exploit the power of threading. A single CPU can appear to execute both lists of instructions in parallel, letting you read your email and listen to music simultaneously.  
  
Threads behave exactly the same way. In our case, the Java virtual machine executes a handful of the instructions to calculate the factorial and then executes a handful of instructions to calculate the square root, and so on.  
  
So threads are simply tasks that you want to execute at roughly the same time. Why, then, write an application with multiple threads? Why not just write multiple applications? The answer lies in the fact that because threads are running in the same application, they share the same memory space in the computer. This allows them to share information seamlessly. Your email program and your music application don't communicate very well. At best, you can copy and paste some data (like the name of a file) between the two. That allows you to double-click on an MP3 attachment in your email and play it in your music application, but the only information that is shared between the two is the name of the MP3 file.   
  
In a multitasking environment, data in the programs is separated by default: each has its own stack for local variables, and each has its own area for objects and other data. All the programs can access various types of shared memory (including the name of the MP3 file that you clicked on in your email program). The shared memory is restricted to information put there by other programs, and the APIs to access it are usually quite different than the APIs used to access other data in the program.  
  
This type of data sharing is fine for dissimilar programs, but it is inadequate for other programs that need to communicate with one another. Consider a network server that sends stock quotes to multiple clients. Sending a quote to a client is a discrete task and may be done in a separate thread. In fact, if the client must acknowledge the quote, then sending the data in separate threads is highly recommended: you don't want all clients to wait for a particularly slow client to respond. Here the data to be sent to the clients is the same; you don't want each client to require a separate server process which must then replicate all the data held by every other server process. Instead, you want multiple threads in one program so that they may share data and each perform discrete tasks on that data.   
  
Conceptually, the threads seem to be the same as programs. The key difference here is that the global memory is the entire Java heap: threads can transparently share access between any object in the heap. Each thread still has its own space for local variables (variables specific to the method the thread is executing). But objects are shared automatically and transparently.

**A thread, then, is a discrete task that operates on data shared with other threads.**

Chapter 3: Creating a Thread

Creating a Thread is fairly simple. The complexities get introduced once you have your threads running and then you see that the output isn’t what you exactly expected. Well, I don’t want to jump ahead too much. We will learn all those complexities one by one. For now lets keep our focus on how to create threads.   
  
You can create threads in two ways:   
  
1. By Extending the Thread Class   
2. By Implementing the Runnable Interface   
  
Common sense related to Java Programming tells us that using the interface may be a better option because, your class may need to inherit features from its parent class or such a feature may be required in future. Either ways, it would be a better idea to implement the Runnable Interface so that our classes can do, what they were designed/created to do.

Chapter 4: The Lifecycle of a Thread

Now that we know how to create a thread, the next step in the learning process it to understand the lifecycle of a thread.   
  
**Before we get started:**  
  
The class Thread contains many methods that might affect the life of a thread. They are:   
public void start( );  
public void run( );  
public void stop( ); // Deprecated, do not use  
public void resume( ); // Deprecated, do not use  
public void suspend( ); // Deprecated, do not use  
public static void sleep(long millis);  
public static void sleep(long millis, int nanos);  
public boolean isAlive( );  
public void interrupt( );  
public boolean isInterrupted( );  
public static boolean interrupted( );  
public void join( ) throws InterruptedException;  
  
Each of these methods has a purpose and an effect on the thread when used.   
  
**Creating a Thread**  
  
The first phase in this lifecycle is thread creation. Threads are represented by instances of the Thread class, so creating a thread is done by calling a constructor of that class. In our example, we use the simplest constructor available to us. Additional constructors of the Thread class allow you to specify the thread's name or a Runnable object to serve as the thread's target.  
  
All threads have names that serve to identify them in the virtual machine. By default, that name consists of information about the thread: its priority, its thread group, and other thread information. If you like, you can give a thread a different name, perhaps one that will have meaning to you if you print it out.   
  
**Starting a Thread**  
  
A thread exists once it has been constructed, but at that point it is not executing any code. The thread is in a waiting state.  
  
In this waiting state, other threads can interact with the existing thread object. Various attributes of the waiting thread can be set: its priority, its name, its daemon status, and so on. Therefore, even though the thread is waiting, its state may be changed by other threads.  
When you're ready for the thread to begin executing code, you call its start() method. This method performs some internal housekeeping and calls the thread's run() method. When the start() method returns, two threads are now executing in parallel: the original thread (which has returned from calling the start() method) and the newly started thread (which is now executing its run() method).  
  
After its start() method has been called, the new thread is said to be alive. In fact, the Thread class has an isAlive() method that tells you the state of the thread: if the isAlive() method returns true, the thread has been started and is executing its run() method. If the isAlive( ) method returns false, however, the thread may not be started yet or may be terminated.  
  
**Terminating a Thread**  
  
Once started, a thread executes only one method: the run() method. The run() method may be very complicated, it may execute forever, and it may call millions of other methods. Regardless of all this, once the run() method finishes executing, the thread has completed its execution. Like all Java methods, the run() method finishes when it executes a return statement, when it executes the last statement in its method body, or when it throws an exception or fails to catch an exception thrown to it.  
  
As a result, the only way to terminate a thread is to arrange for its run() method to complete. If you look at the documentation of the Thread class, you notice that the class contains a stop() method which seems like it might be used to terminate a thread. It turns out that the stop() method has an inherent problem (an internal race condition, we will see what a race condition is very soon). As a result, the stop() method is deprecated and should not be used. Some Java implementations prohibit its use directly, and the security manager can also be used to prohibit programs from calling it.  
  
There are many threads that you don't need to stop. Often, threads are performing a fixed task, and you always want the task to run to completion. In other cases, the thread can run until the application exits (e.g., when we call the System.exit() method in response to the user pressing the Quit button).  
  
Often, however, you want a thread to continue to execute until some other condition is met.   
The run() method cannot throw a checked exception, but like all Java methods, it can throw an unchecked exception. Throwing an unchecked exception (an exception that extends the RuntimeException class)—or failing to catch a runtime exception thrown by something the run( ) method has called—also causes a thread to stop.   
  
**Pausing, Suspending, and Resuming Threads**  
  
Once a thread begins executing its run() method, it continues execution until the run() method completes. If you're familiar with other thread models, you may know of a concept called thread suspension, where a thread is told to pause its execution. Later, the thread is resumed, which is to say that it is told to continue its execution. The Thread class contains suspend() and resume() methods, but they suffer from the same race condition problem as the stop() method, and they, too, are deprecated.  
  
It is possible for a thread to suspend its own execution for a specific period of time by calling the sleep() method. When a thread executes the sleep() method, it pauses for a given number of milliseconds, during which it is said to be asleep. When the pause time has elapsed, the thread wakes up and continues execution with the statements immediately following the sleep() method.

**Trivia**:   
The Thread class provides a version of the sleep() method that allows the developer to specify the time in nanoseconds. Most Java virtual machines do not support this sort of precise timing. When the sleep() method executes, it rounds the nanosecond argument to the nearest millisecond. In fact, most operating systems then further adjust the millisecond argument so that it is a multiple of some number: e.g., 20 or 50 milliseconds. Consequently, the least amount of time that you can sleep on most Java implementations is 20 or 50 milliseconds.  
Note that this is true even in J2SE 5.0, which introduces other nanosecond time functionality (e.g., the System.nanoTime() method). The resolution of the sleep( ) method is still only good to a few milliseconds.

Strictly speaking, sleeping is not the same thing as thread suspension. One important difference is that with true thread suspension, one thread would suspend (and later resume) another thread. Conversely, the sleep() method affects only the thread that executes it; it's not possible to tell another thread to go to sleep.  
  
Threads can use the wait and notify mechanism to achieve the functionality of thread suspension and resumption. The difference is that the threads must be coded to use that technique (rather than a generic suspend/resume mechanism that could be imposed from other threads). Don't worry much about the wait and notify just yet. We will cover it in detail very soon.   
  
**Thread Cleanup**  
  
A thread that has completed its run() method has terminated. It is no longer active (the isAlive() method returns false). However, the thread object itself may be holding interesting information. As long as some other active object holds a reference to the terminated thread object, other threads can execute methods on the terminated thread and retrieve that information. If the thread object representing the terminated thread goes out of scope, the thread object is garbage collected. On some platforms, this also has the effect of cleaning up system resources associated with the thread.  
  
In general, then, you should not hold onto thread references so that they may be collected when the thread terminates.  
  
One reason to hold onto a thread reference is to determine when it has completed its work. That can be accomplished with the join() method. The join() method is often used when you have started threads to perform discrete tasks and want to know when the tasks have completed.   
The join() method blocks until the thread has completed its run() method. If the thread has already completed its run() method, the join() method returns immediately. This means that you may call the join() method any number of times to see whether a thread has terminated. Be aware, though, that the first time you call the join() method, it blocks until the thread has actually completed. You cannot use the join() method to poll a thread to see if it's running (instead, use the isAlive() method just discussed).