Multithreading & Concurrent Programming

Java supports single-thread as well as multi-thread operations. A single-thread program has a single entry point (the `main()` method) and a singlex exit point. A multi-thread program has an initial entry point (the `main()` method), followed by many entry and exit points, which are run concurrently with the `main().` The term "concurrency" refers to doing multiple tasks at the same time.

Java has built-in support for concurrent programming by running multiple threads concurrently within a single program.

Process based Multitasking

- Executing multiple processes at the same time. The processes under execution are not dependent on each other. Every process will have its own set of resources.
- The process based multitasking is an operating systems approach
- Example: writing a java program, downloading a software, installing an application, playing audio etc.
- A multi-processing Operating System can run several processes at the same time
- Each process has its own address/memory space
- The OS's scheduler decides when each process is executed
- Only one process is actually executing at any given time. However, the system appears to be running several programs simultaneously
- Separate processes do not have access to each other's memory space
- Many OSes have a shared memory system so that processes can share memory space

Thread based Multitasking

- Executing different parts of the same program simultaneously is called Thread based Multitasking. The different parts (threads) of the program may(or may not) be dependent on each other. All the different parts(threads) of the program may share same resources.
- Thread based multitasking is also called Multithreading
- Multithreading is a process of executing different parts of an application that has multiple controls, where every control can be considered as a “Thread”.

Thread

- A Thread is a group of Statements that are executed separately. Thread is used for dividing the task of an application into separate sub-processes which can run simultaneously. A program can have multiple threads
- A thread consumes some resources so we should not use more thread than our requirement
- A Java threads shares the common memory area so memory allocation for each thread is not required. So context switching requires less time than using the multi-processing
- Every Java program has at least one thread i.e the thread that executes the Java Program (main thread)
- Threads are faster and more efficient program which increases speed of execution
- In a multithreaded application, there are several points of execution within the same memory space.
  - Each point of execution is called a thread
  - Threads share access to memory
- Example : OS will start the process which would run MS Power Point
  - We can consider the MS Power Point as main process, in order to make interactive execution, MS Power Point process can create multiple sub processes i.e thread
  - When we type anything in the slide then spell checker thread can be automatically created. We can also consider the auto-correction as thread while typing

Multithreading Vs Multiprocessing

- OS can start multiple processes to achieve multitasking.
- In order to make execution faster, each process can be sub divided into the smaller chunk of lightweight sub-processes which is called as thread

Key difference:

- Multithreading refers to an application with multiple threads running within a process
- Multiprocessing refers to an application organized across multiple OS-level processes

Thread	Process
It is a programming concept in which a program or a process is divided into two or more subprograms or threads that are executed at the same time	It is an operating system approach in which multiple tasks are performed simultaneously
Threads does not requires separate address space	Process requires separate address space
Threads can be considered as sub process so they are lightweight	Processes are not lightweight.
Thread are not independent	Processes can be independent
Thread cannot be divided into multiple processes	Process can be divided into threads

Java Program has At least one Thread:

- Each and every java program have at least on thread
- First thread is created when you invoke the static main method of your Java class
- Many java programs can have more than one thread which are created automatically.
- Thread is mostly used for game development but non-game applications can also use multi-threading

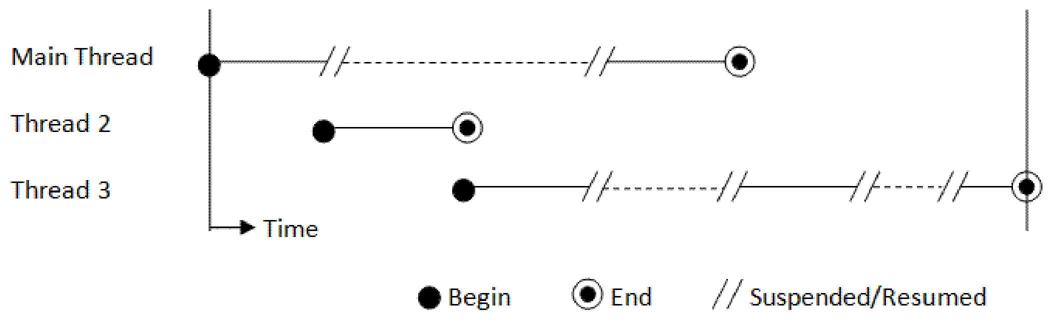
Program No. **Main Thread**

157

```
import java.lang.Thread;
public class ThreadDemo_157
{
	public static void main(String[] args)
	{
		Thread t = Thread.currentThread();
		System.out.println(t);
	}
}
```

Thread.currentThread() - it is a static method of java.lang.Thread Class which provides the information of current thread i.e. ThreadName, ThreadPriority, ThreadGroupName of the currently executing thread

A thread, also called a lightweight process, is a single sequential flow of programming operations, with a definite beginning and an end. During the lifetime of the thread, there is only a single point of execution. A thread by itself is not a program because it cannot run on its own. Instead, it runs within a program. The following figure shows a program with 3 threads running under a single CPU:



A typical Java program runs in a single process, and is not interested in multiple processes. However, within the process, it often uses multiple threads to to run multiple tasks concurrently. A standalone Java application starts with a single thread (called main thread) associated with the main () method. This main thread can then start new user threads

Creating a new Thread

There are two ways to create a new thread:

1. By extending Thread Class

Extend a subclass from the superclass `Thread` and override the `run()` method to specify the running behavior of the thread. Create an instance and invoke the `start()` method, which will call-back the `run()` on a new thread. For example:

```
public class ThreadDemo
{
	public static void main(String[] args)
	{
		Thread t1 = new Thread() {
		// Create an instance of an anonymous inner class that extends Thread
		   public void run() {
		// Override run() to specify the running behaviors
		      for (int i = 0; i < 20; ++i) {
		        System.out.println("Thread 1 :" + i);
		            // provide the necessary delay
```

```
                try {
                   sleep(1000); // milliseconds
                } catch (InterruptedException ex) {}
             }
          }
       };
       t1.start();  // Start the thread. Call back run() in a new thread


       Thread t2 = new Thread() {
                     public void run() {
                        for (int i = 0; i < 20; ++i) {
                          System.out.println("Thread 2 :" + i);
                           try {
                              sleep(1000);
                           } catch (InterruptedException ex) {}
                        }
                     }
                  };
       t2.start();

 }
}
```

Program No. 159

Thread Demo

```
import java.lang.Thread;

class ClassA extends Thread
{
	String name;
	public ClassA(String n)
	{
		name = n;
	}
	public void run()
	{
		for(int i=1; i<=5; i++)
		{
			System.out.println(name + " i = " + i);
		}
		System.out.println("Exit from " + name);
	}
}

public class ThreadDemo_159
{

	 public static void main(String[] args)
	 {
			ClassA obj1 = new ClassA("obj1");
			System.out.println("obj1 = " + obj1.getState());
			obj1.start();

			ClassA obj2 = new ClassA("obj2");
			obj2.start();

			try
```

```
                    {
                          obj1.join();
                          obj2.join();
                    }catch(InterruptedException e){}


                    System.out.println("Exit from main");
           }
}
```

- A newly created Thread will be in the born state or new state, then by calling the start() method thread moves from new state to runnable state
- start() method will call run() method which is overridden by the ThreadExample class.

2. By Implementing Runnable Interface

A Thread can be created by extending Thread class also. But Java allows only one class to extend, it wont allow multiple inheritance. So it is always better to create a thread by implementing Runnable interface. Java allows you to implement multiple interfaces at a time

By implementing Runnable interface, we need to provide implementation for run() method. To run this implementation class, create a Thread object, pass Runnable implementation class object to its constructor. Call start() method on thread class to start executing run() method.

Implementing Runnable interface does not create a Thread object, it only defines an entry point for threads in your object

It allows you to pass the object to the Thread(Runnable implementation) constructor

Create a class that implements the `Runnable` interface and provide the implementation to the `abstract` method `run()` to specify the running behavior of the thread. Construct a new`Thread` instance using the constructor with a `Runnable` object and invoke the `start()` method, which will call back `run()` on a new thread.

```
// Create an anonymous instance of an anonymous inner class that implements Runnable
// and use the instance as the argument of Thread's constructor.
Thread t = new Thread(new Runnable() {
   // Provide implementation to abstract method run() to specify the running behavior
   @Override
   public void run() {
      for (int i = 0; i < 100000; ++i) {
         if (stop) break;
         tfCount.setText(count + "");
         ++count;
         // Suspend itself and yield control to other threads
    // Also provide the necessary delay
         try {
            Thread.sleep(10);   // milliseconds
         } catch (InterruptedException ex) {}
      }
   }
});
t.start();   // call back run() in new thread
```

The second method is needed as Java does not support multiple inheritance. If a class already extends from a certain superclass, it cannot extend from `Thread`, and have to implement the`Runnable` interface. The second method is also used to provide compatibility with JDK 1.1. It should be noted that the `Thread` class itself implements the `Runnable` interface.

The `run()` method specifies the running behavior of the thread and gives the thread something to do. You do not invoke the `run()` method directly from your program. Instead, you create a`Thread` instance and invoke the `start()` method.

The `start()` method, in turn, will call back the `run()` on a new thread.

Program No. 160

Implementing Runnable Interface

```
class MyThread implements Runnable
{
	public void run()
	{
		System.out.println(Thread.currentThread()+"starts");
		for(int i=1; i<=5; i++)
		{
			System.out.println(Thread.currentThread().getName() + "i = " + i);
		}
		System.out.println("\n\nExit from " + Thread.currentThread().getName());
	}
}

public class ThreadDemo_160
{
	public static void main(String[] args)
	{
		MyThread mythread1 = new MyThread();
		MyThread mythread2 = new MyThread();

		Thread t1 = new Thread(mythread1, "thread1");
		Thread t2 = new Thread(mythread2,  "thread2");

		t1.start();
		t2.start();
	}
}
```

Program No. 161

Implementing Runnable Interface And constructor

```
class MyThread implements Runnable
{
	Thread t;
	String tname;

	public MyThread(String n)
	{
		tname = n;
		t = new Thread(this, tname);
		t.start();
	}
	public void run()
	{
		for(int i=1; i<=5; i++)
		{
			System.out.println(tname + "i = " + i);
		}
		System.out.println("\n\nExit from " + tname );
	}
}

public class ThreadDemo_161
{
        public static void main(String[] args)
        {
                MyThread t1 = new MyThread("MyThread 1 ");
                MyThread t2 = new MyThread("MyThread 2 ");
        }
}
```

Interface Runnable

The interface `java.lang.Runnable` declares one `abstract` method `run()`, which is used to specify the running behavior of the thread:

```
public void run();
```

Class Thread

The class `java.lang.Thread` has the following constructors:

public **Thread**();

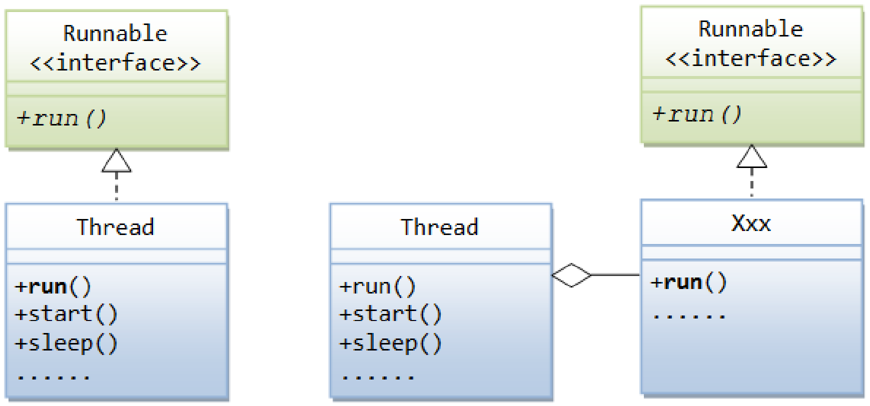
public **Thread**(String *threadName*);

public **Thread**(Runnable *target*);

public **Thread**(Runnable *target*, String *threadName*);

The first two constructors are used for creating a thread by sub-classing the `Thread` class. The next two constructors are used for creating a thread with an instance of class that implements `Runnable` interface.

The class `Thread` implements `Runnable` interface, as shown in the class diagram.



As mentioned, the method `run()` specifies the running behavior of the thread. You do not invoke the `run()` method explicitly. Instead, you call the `start()` method of the class `Thread`. If a thread is constructed by extending the `Thread` class, the method `start()` will call back the overridden `run()` method in the extended class. On the other hand, if a thread is constructed by providing a `Runnable` object to the `Thread`'s constructor, the `start()` method will call back the `run()` method of the `Runnable` object (and not the `Thread`'s version).

Creating a new Thread by sub-classing Thread and overriding run()

To create and run a new thread by extending Thread class:

1. Define a subclass (named or anonymous) that extends from the superclass `Thread`.

2. In the subclass, override the `run()` method to specify the thread's operations, (and provide other implementations such as constructors, variables and methods).
3. A client class creates an instance of this new class. This instance is called a `Runnable` object (because `Thread` class itself implements `Runnable` interface).
4. The client class invokes the `start()` method of the `Runnable` object. The result is two thread running concurrently – the current thread continue after invoking the `start()`, and a new thread that executes `run()` method of the `Runnable` object.

For example,

```
class MyThread extends Thread {
   // override the run() method
   @Override
   public void run() {
      // Thread's running behavior
   }
   // constructors, other variables and methods
   ......
}
public class Client {
   public static void main(String[] args) {
      ......
      // Start a new thread
      MyThread t1 = new MyThread();
      t1.start();  // Called back run()
      ......
      // Start another thread
      new MyThread().start();
      ......
   }
}
```

Often, an inner class (named or anonymous) is used instead of a ordinary subclass. This is done for readability and for providing access to the private variables and methods of the outer class. For example,

```
public class Client {
   ......
   public Client() {
      Thread t = new Thread() { // Create an anonymous inner class extends Thread
         @Override
         public void run() {
            // Thread's running behavior
            // Can access the private variables and methods of the outer class
```

```
               }
            };
            t.start();
            ...
            // You can also used a named inner class defined below
            new MyThread().start();
         }
```

// Define a named inner class extends Thread

```
         class MyThread extends Thread {
            public void run() {
               // Thread's running behavior
               // Can access the private variables and methods of the outer class
            }
         }
      }
```

Example

```
      public class MyThread extends Thread {
         private String name;

         public MyThread(String name) {   // constructor
            this.name = name;
         }

         // Override the run() method to specify the thread's running behavior
         @Override
         public void run() {
            for (int i = 1; i <= 5; ++i) {
               System.out.println(name + ": " + i);
               yield();
            }
         }
      }
```

A class called `MyThead` is created by extending `Thread` class and overriding the `run()` method. A constructor is defined to takes a `String` as the name of the thread. The `run()` method prints 1 to 5, but invokes `yield()` to yield control to other threads voluntarily after printing each number.

```
            public class TestMyThread {
               public static void main(String[] args) {
                  Thread[] threads = {
                     new MyThread("Thread 1"),
                     new MyThread("Thread 2"),
                     new MyThread("Thread 3")
                  };
                  for (Thread t : threads) {
                     t.start();
                  }
               }
            }
```

Take note that the output is indeterminate (different run is likely to produce different output), as we do not have complete control on how the threads would be executed.

Creating a new Thread by implementing the Runnable Interface

To create and run a new thread by implementing `Runnable` interface:

1. Define a class that implements the `Runnable` interface.
2. In the class, provide implementation to the `abstract` method `run()` to specify the thread's operations, (and provide other implementations such as constructors, variables and methods).
3. A client class creates an instance of this new class. The instance is called a `Runnable` object.
4. The client class then constructs a new `Thread` object with the `Runnable` object as argument to the constructor, and invokes the `start()` method. The `start()` called back the `run()` in the `Runnable` object (instead of the `Thread` class).

```
class MyRunnable extends SomeClass implements Runnable {
   // provide implementation to abstract method run()
   @Override
   public void run() {
      // Thread's running behavior
   }
   ......
   // constructors, other variables and methods
}
public class Client {
   ......
   Thread t = new Thread(new MyRunnable());
   t.start();
   ...
}
```

Again, an inner class (named or anonymous) is often used for readability and to provide access to the private variables and methods of the outer class.

```
Thread t = new Thread(new Runnable() { // Create an anonymous inner class
that implements Runnable interface
   public void run() {
      // Thread's running behavior
```

```
              // Can access the private variables and methods of the outer class
           }
        });
        t.start();
```

Methods in the Thread Class

The methods available in Thread class include:

public void start(): Begin a new thread. JRE calls back the run() method of this class. The current thread continues.

public void run(): to specify the execution flow of the new thread. When run() completes, the thread terminates.

public static sleep(long millis)
public static sleep(long millis, int nanos)

public void interrupt()
Suspend the current thread and yield control to other threads for the given milliseconds (plus nanoseconds).

Method `sleep()` is thread-safe as it does not release its monitors. You can awaken a sleep thread before the specified timing via a call to the `interrupt ()` method. The awaken thread will throw an `InterruptedException` and execute its `InterruptedException` handler before resuming its operation. This is a `static` method (which does not require an instance) and commonly used to pause the current thread (via `Thread.sleep()`) so that the other threads can have a chance to execute. It also provides the necessary delay in many applications. For example:

```
try {
   // Suspend the current thread and give other threads a chance to run
   // Also provide the necessary delay
   Thread.sleep(100);  // milliseconds
} catch (InterruptedException ex) {}
```

public static void yield(): hint to the scheduler that the current thread is willing to yield its current use of a processor to allow other threads to run. The scheduler is, however, free to ignore this hint. Rarely-used.

public boolean isAlive(): Return false if the thread is new or dead. Returns true if the thread is "runnable" or "not runnable".

public void setPriority(int p): Set the priority-level of the thread, which is implementation dependent.

The `stop()`, `suspend()`, and `resume()` methods have been deprecated in JDK 1.4, because they are not thread-safe, due to the release of monitors. See JDK API documentation for more discussion.

Program No. 162

Thread Example Tortoise and hair story

```
class Racer implements Runnable
{
	private static String Winner;
	public Thread t;
	public String racername;

	public Racer(String rn)
	{
		racername = rn;
		t = new Thread(this, racername);
		t.start();
	}

	public void run()
	{
		startrace();
	}
	public void startrace()
	{
		for(int distance = 1; distance<=100; distance++)
		{
			System.out.println(racername + " covered : "
			+distance + " meters");
			boolean isRaceWon = isWon(distance);

			if(isRaceWon)
			{
				break;
			}
		}
	}
	public boolean isWon(int distance)
	{
		boolean result;
		if(Racer.Winner==null && distance==100)
		{
			//declare winner
			Racer.Winner = racername;
			System.out.println("\n\n\t Race Winner is " +
			Racer.Winner);
			result = true;
		}
		else if(Racer.Winner==null)
		{
			result =  false;
		}
		else if(Racer.Winner!=null)
		{
			result =  true;
		}
		else
		{
			result =  false;
		}
		return result;
	}
}
public class ThreadDemo_162
{
	public static void main(String[] args)
	{
		Racer tortoise = new Racer("Tortoise");
		Racer hair = new Racer("Hair");

	}
}
```

How to pause a thread:

You can make the currently running thread pauses its execution by invoking the static method `sleep(milliseconds)` of the `Thread` class. Then the current thread is put into sleeping state. Here's how to pause the current thread:

```
try {

      Thread.sleep(2000);

} catch (InterruptedException ex) {

      // code to resume or terminate...

}
```

This code pauses the current thread for about 2 seconds (or 2000 milliseconds). After that amount of time, the thread returns to continue running normally.

`InterruptedException` is a checked exception so you must handle it. This exception is thrown when the thread is interrupted by another thread.

Let's see a full example. The following `NumberPrint` program is updated to print 5 numbers, each after every 2 seconds:

```
public class NumberPrint implements Runnable {

      public void run() {

            for (int i = 1; i <= 5; i++) {

                  System.out.println(i);

                  try {

                        Thread.sleep(2000);

                  } catch (InterruptedException ex) {

                        System.out.println("I'm interrupted");

                  }

            }

      }

      public static void main(String[] args) {

            Runnable task = new NumberPrint();

            Thread thread = new Thread(task);

            thread.start();

      }

}
```

Note that you can't pause a thread from another thread. Only the thread itself can pause its execution. And there's no guarantee that the thread always sleep exactly for the specified time because it can be interrupted by another thread, which is described in the next section.

* How to interrupt a thread:

Interrupting a thread can be used to stop or resume the execution of that thread from another thread. For example, the following statement interrupts the thread `t1` from the current thread:

```
t1.interrupt();
```

If `t1` is sleeping, then calling `interrupt()` on `t1` will cause the `InterruptedException` to be thrown. And whether the thread should stop or resume depending on the handling code in the catch block.

In the following code example, the thread `t1` prints a message after every 2 seconds, and the main thread interrupts `t1` after 5 seconds:

```
public class ThreadInterruptExample implements Runnable {

    public void run() {

        for (int i = 1; i <= 10; i++) {

            System.out.println("This is message #" + i);

            try {

                Thread.sleep(2000);

                continue;

            } catch (InterruptedException ex) {

                System.out.println("I'm resumed");

            }

        }

    }

    public static void main(String[] args) {

        Thread t1 = new Thread(new ThreadInterruptExample());

        t1.start();

        try {

            Thread.sleep(5000);

            t1.interrupt();
```

```
                } catch (InterruptedException ex) {

                      // do nothing

                }

          }

    }
```

As you can see in the catch block in the `run()` method, it continues the for loop when the thread is interrupted:

```
    try {

          Thread.sleep(2000);

    } catch (InterruptedException ex) {

          System.out.println("I'm resumed");

          continue;

    }
```

That means the thread resumes running while it is sleeping.

To stop the thread, just change the code in the catch block to return from the `run()` method like this:

```
    try {

          Thread.sleep(2000);

    } catch (InterruptedException ex) {

          System.out.println("I'm about to stop");

          return;

    }
```

You see, the return statement causes the `run()` method to return which means the thread terminates and goes to dead state.

What if a thread doesn't sleep (no handling for `InterruptedException`)?

In such case, you need to check the interrupt status of the current thread using either of the following methods of the `Thread` class:

- `interrupted()`: this static method returns `true` if the current thread has been interrupted, or `false` otherwise. Note that this method clears the interrupt status, meaning that if it returns `true`, then the interrupt status is set to `false`.

- `isInterrupted()`: this non-static method checks the interrupt status of the current thread and it doesn't clear the interrupt status.

The `ThreadInterruptExample` above can be modified to use the checking method as below:

```
public class ThreadInterruptExample implements Runnable {

    public void run() {

        for (int i = 1; i <= 10; i++) {

            System.out.println("This is message #" + i);

            if (Thread.interrupted()) {

                System.out.println("I'm about to stop");

                return;

            }

        }

    }

    public static void main(String[] args) {

        Thread t1 = new Thread(new ThreadInterruptExample());

        t1.start();

        try {

            Thread.sleep(5000);

            t1.interrupt();

        } catch (InterruptedException ex) {

            // do nothing

        }

    }

}
```

However this version doesn't behave the same as the previous one because the thread `t1` terminates very quickly as it doesn't sleep and the print statements are executed very fast. So this example is just to show you how it is used. In practice, this kind of checking on interrupt status should be applied for long-running operations such as IO, network, database, etc.

And remember that when the `InterruptedException` is thrown, the interrupt status is cleared.

If you look at the Thread class in Javadocs, you will see there are 4 methods:

```
destroy() - stop() - suspend() - resume()
```

However all these methods are deprecated, meaning that you shouldn't use them. Let use the interruption mechanism I have described so far.

* How to make a thread waits other threads?

This is called joining and is useful in case you want the current thread to wait for other threads to complete. After that the current thread continues running. For example:

```
t1.join();
```

This statement causes the current thread to wait for the thread `t1` to complete before it continues. In the following program, the current thread (main) waits for the thread `t1`to complete:

```
public class ThreadJoinExample implements Runnable {

      public void run() {

            for (int i = 1; i <= 10; i++) {

                  System.out.println("This is message #" + i);

                  try {

                        Thread.sleep(2000);

                  } catch (InterruptedException ex) {

                        System.out.println("I'm about to stop");

                        return;

                  }

            }

      }

      public static void main(String[] args) {

            Thread t1 = new Thread(new ThreadJoinExample());

            t1.start();

            try {

                  t1.join();

            } catch (InterruptedException ex) {

                  // do nothing

            }

            System.out.println("I'm " + Thread.currentThread().getName());

      }
```

```
        }
```

In this program, the current thread (main) always terminates after the thread t1 completes. Hence you see the message "`I'm main`" is always printed last:

```
        This is message #1

        This is message #2

        This is message #3

        This is message #4

        This is message #5

        This is message #6

        This is message #7

        This is message #8

        This is message #9

        This is message #10

        I'm main
```

Note that the `join()` method throws `InterruptedException` if the current thread is interrupted, so you need to catch it.

There are 2 overloads of `join()` method:

```
        - join(milliseconds)

        - join(milliseconds,  nanoseconds)
```

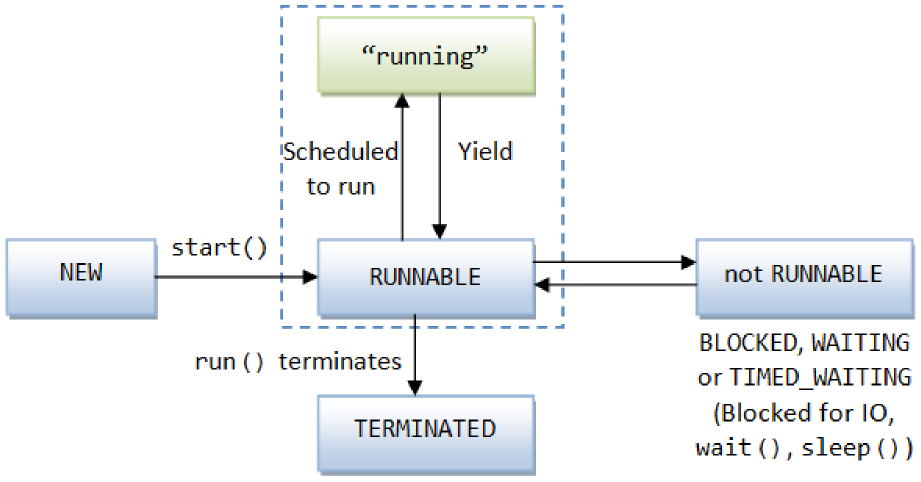
These methods cause the current thread to wait at most for the specified time. That means if the time expires and the joined thread has not completed, the current thread continues running normally.

You can also join multiple threads with the current thread, for example:

```
        t1.join();

        t2.join();

        t3.join();
```

In this case, the current thread has to wait for all three threads `t1`, `t2` and `t3` completes before it can resume running.

The Life Cycle of a Thread



The thread is in the "new" state, once it is constructed. In this state, it is merely an object in the heap, without any system resources allocated for execution. From the "new" state, the only thing you can do is to invoke the `start()` method, which puts the thread into the "runnable" state. Calling any method besides the `start()` will trigger an `IllegalThreadStateException`.

The `start()` method allocates the system resources necessary to execute the thread, schedules the thread to be run, and calls back the `run()` once it is scheduled. This put the thread into the "runnable" state. However, most computers have a single CPU and *time-slice* the CPU to support multithreading. Hence, in the "runnable" state, the thread may be running or waiting for its turn of the CPU time.

A thread cannot be started twice, which triggers a runtime `IllegalThreadStateException`.

The thread enters the "not-runnable" state when one of these events occurs:

1. The `sleep()` method is called to suspend the thread for a specified amount of time to yield control to the other threads. You can also invoke the `yield()` to hint to the scheduler that the current thread is willing to yield its current use of a processor. The scheduler is, however, free to ignore this hint.
2. The `wait()` method is called to wait for a specific condition to be satisfied.
3. The thread is *blocked* and waiting for an I/O operation to be completed.

For the "non-runnable" state, the thread becomes "runnable" again:

1. If the thread was put to sleep, the specified sleep-time expired or the sleep was interrupted via a call to the `interrupt()` method.
2. If the thread was put to wait via `wait()`, its `notify()` or `notifyAll()` method was invoked to inform the waiting thread that the specified condition had been fulfilled and the wait was over.
3. If the thread was blocked for an I/O operation, the I/O operation has been completed.

A thread is in a "terminated" state, only when the `run()` method terminates naturally and exits.

The method `isAlive()` can be used to test whether the thread is alive. The `isAlive()` returns `false` if the thread is "new" or "terminated". It returns `true` if the thread is "runnable" or "not-runnable".

Thread States (Thread Life Cycle) in Java

A thread can go through various states during its life. The `Thread`'s `getState()` method returns an enum constant that indicates current state of the thread, which falls in one of the following values:

- RUNNABLE
- BLOCKED
- WAITING
- TIMED_WAITING
- TERMINATED

These enum constants are defined in the `Thread.State` enum. Let me explain each state in details.

- **NEW**: when a thread is created but has not executed (the `start()` method has not been invoked), it is in the new state.

- **RUNNABLE**: when the `start()` method has been invoked, the thread enters the runnable state, and its `run()` method is executing. Note that the thread can come back to runnable state from another state (waiting, blocked), but it may not be picked immediately by the thread scheduler, hence the term "runnable", not running.

- **BLOCKED**: when a thread tries to acquire an intrinsic lock (not a lock in the `java.util.concurrent` package) that is currently held by another thread, it becomes blocked. When all other threads have relinquished the lock and the thread scheduler has allowed this thread to hold the lock, the thread becomes unblocked and enters the runnable state.

- **WAITING**: a thread enters this state if it waits to be notified by another thread, which is the result of calling `Object.wait()` or `Thread.join()`. The thread also enters waiting state if it waits for a `Lock` or `Condition` in the `java.util.concurrent` package. When another thread calls `Object`'s `notify()`/`notifyAll()` or `Condition`'s `signal()`/`signalAll()`, the thread comes back to the runnable state.

- **TIMED_WAITING**: a thread enters this state if a method with timeout parameter is called: `sleep()`, `wait()`, `join()`, `Lock.tryLock()` and `Condition.await()`. The thread exits this state if the timeout expires or the appropriate notification has been received.

- **TERMINATED**: a thread enters terminated state when it has completed execution. The thread terminates for one of two reasons:

  + the `run()` method exits normally.

  + the `run()` method exits abruptly due to a uncaught exception occurs.

The following diagram helps you visually understand the thread states and transitions between them:

And the following code example illustrates how to check state of a thread:

```
public class ThreadState {

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new Runnable() {
            public void run() {
                Thread self = Thread.currentThread();
                  System.out.println(self.getName() + "is " + self.getState());//
                  LINE 0
            }
        });
```

```
            System.out.println(t.getName() + "is " + t.getState()); // LINE 1

            t.start();

            t.join();

            if (t.getState() == Thread.State.TERMINATED) {
                System.out.println(t.getName() + " is terminated"); // LINE 2
            }
        }
    }
```

Run this program and you will see the following output:

```
Thread-0 is NEW
Thread-0 is RUNNABLE
Thread-0 is terminated
```

Line 1 prints `Thread-0 is NEW` because at this time the thread t has not been started.

When the thread started, its run method execute, hence line 0 prints `Thread-0 is RUNNABLE`.

The call `t.join()` causes the main thread to wait for the thread t to finish, hence line 2 is always executed after thread t completes, thus the output `Thread-0 is terminated`.

Note that the thread's state may change after the call to `getState()`. That means calling `getState()` may not reflect the actual state of the thread only a moment later.

Program No. 158

Thread Phases

```
import java.lang.Thread;

public class ThreadDemo extends Thread
{

      public void run()
        {
            System.out.println("Thread is running !!");
        }

        public static void main(String[] args){
            ThreadDemo t1 = new ThreadDemo();
            ThreadDemo t2 = new ThreadDemo();

            System.out.println("T1 ==> " + t1.getState());
            System.out.println("T2 ==> " + t2.getState());

            t1.start();
            System.out.println("T1 ==> " + t1.getState());
            System.out.println("T2 ==> " + t2.getState());

            t2.start();
            System.out.println("T1 ==> " + t1.getState());
            System.out.println("T2 ==> " + t2.getState());
```

```
          }

}
```

Thread Scheduling and Priority

JVM implements a fixed priority thread-scheduling scheme. Each thread is assigned a priority number (between the `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`). The higher the number, the higher is the priority for the thread. When a new thread is created, it inherits the priority number from the thread that created it. You can used the method `setPriority()` to change the priority number of a thread as follows:

```
public void setPriority(int priority);
```

The `int priority` is JVM dependent. It may take a value between 1 (lowest priority) to 10.

JVM chooses the highest-priority thread for execution. If there is more than one thread with the same highest-priority, JVM schedules them in a round-robin manner.

JVM also implements a preemptive scheduling scheme. In a preemptive environment, if at any time a higher priority thread becomes "runnable", the current lower priority thread will yield control to the higher priority thread immediately.

If there are more than one equal-priority runnable threads, one thread may run until the completion without yielding control to other equal-priority threads. This is known as *starvation*. Therefore, it is a good practice to yield control to other equal-priority thread via the `sleep()` or `yield()` method. However, you can never yield control to a lower-priority thread.

In some operating systems such as Windows, each of the running thread is given a specific amount of CPU time. It is known as time slicing to prevent a thread from starving the other equal-priority threads. However, do not rely on time slicing, as it is implementation dependent.

Hence, a running thread will continue running until:

- A higher priority thread becomes "runnable".
- The running thread yields control voluntarily by calling methods such as `sleep()`, `yield()`, and `wait()`.
- The running thread terminates, i.e., its `run()` method exits.
- On system that implements time slicing, the running thread consumes its CPU time quota.

An important point to note is the thread scheduling and priority is JVM dependent. This is natural as JVM is a virtual machine and requires the native operating system resources to support multithreading. Most JVM does not guarantee that the highest-priority thread is being run at all times. It may choose to dispatch a lower-priority thread for some reasons such as to prevent starvation. Therefore, you should not rely on the priority in your algorithm.

Monitor Lock & Synchronization

A `monitor` is an object that can be used to block and revive thread. It is supported in the `java.lang.Object` root class, via these mechanisms:

1. A lock for each object.
2. The keyword `synchronized` for accessing object's lock.
3. The `wait()`, `notify()` and `notifyAll()` methods in `java.lang.Object` for controlling threads.

Each Java object has a lock. At any time, the lock is controlled by, at most, a single thread. You could mark a method or a block of the codes with keyword `synchronized`. A thread that wants to execute an object's synchronized code must first attempt to acquire its lock. If the lock is under the control of another thread, then the attempting thread goes into the *Seeking Lock* state and becomes ready only when the lock becomes available. When a thread that owns a lock completes the synchronized code, it gives up the lock.

Keyword "synchronized"

For example,

```
public synchronized void methodA() { ...... }  // synchronized a method based on
this object

public void methodB() {
   synchronized(this) {      // synchronized a block of codes based on this object
      ......
   }

   synchronized(anObject) {  // synchronized a block of codes based on another
object
      ......
   }
   ......
}
```

Synchronization can be controlled at method level or block level. Variables cannot be synchronized. You need to synchronized the ALL the methods that access the variables.

```
private static int counter = 0;

public static synchronized void increment() {
   ++counter;
}

public static synchronized void decrement() {
   --counter;
}
```

You can also `synchronized` on `static` methods. In this case, the *class lock* (instead of the instance lock) needs to be acquired in order to execute the method.

Example

```
public class SynchronizedCounter {
   private static int count = 0;


   public synchronized static void increment() {
      ++count;
      System.out.println("Count is " + count + " @ " + System.nanoTime());
   }


   public synchronized static void decrement() {
      --count;
      System.out.println("Count is " + count + " @ " + System.nanoTime());
   }
}
public class TestSynchronizedCounter {
   public static void main(String[] args) {
      Thread threadIncrement = new Thread() {
         @Override
         public void run() {
            for (int i = 0; i < 10; ++i) {
               SynchronizedCounter.increment();
               try {
                  sleep(1);
               } catch (InterruptedException e) {}
            }
         }
      };

      Thread threadDecrement = new Thread() {
         @Override
         public void run() {
            for (int i = 0; i < 10; ++i) {
               SynchronizedCounter.decrement();
               try {
                  sleep(1);
               } catch (InterruptedException e) {}
            }
         }
      };

      threadIncrement.start();
      threadDecrement.start();
   }
}
```

It is important to note that while the object is locked, `synchronized` methods and codes are blocked. However, non-`synchronized` methods can proceed without acquiring the lock. Hence, it is necessary to synchronize all the methods involved the shared resources. For example, if `synchronized` access to a variable is desired, all the methods to that variable should be`synchronized`. Otherwise, a `non-synchronized` method can proceed without first obtaining the lock, which may corrupt the state of the variable.

wait(), notify() & notifyAll() for Inter-Thread Synchronization

These methods are defined in the `java.lang.Object` class (instead of `java.land.Thread` class). These methods can only be called in the *synchronous* codes.

The `wait()` and `notify()` methods provide a way for a shared object to pause a thread when it becomes unavailable to that thread and to allow the thread to continue when appropriate.

Example: Consumer and Producer

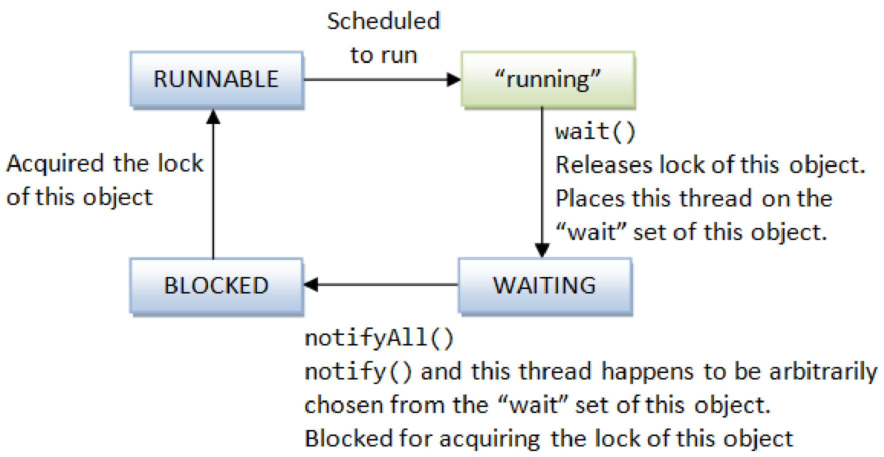
In this example, a producer produces a message (via `putMessage()` method) that is to be consumed by the consumer (via `getMessage()` method), before it can produce the next message. In a so-called producer-consumer pattern, one thread can suspend itself using `wait()` (and release the lock) until such time when another thread awaken it using `notify()` or `notifyAll()`.

```
// Testing wait() and notify()
public class MessageBox {
   private String message;
   private boolean hasMessage;

   // producer
   public synchronized void putMessage(String message) {
      while (hasMessage) {
         // no room for new message
         try {
            wait();  // release the lock of this object
         } catch (InterruptedException e) { }
      }
      // acquire the lock and continue
      hasMessage = true;
      this.message = message + " Put @ " + System.nanoTime();
      notify();
   }

   // consumer
   public synchronized String getMessage() {
      while (!hasMessage) {
         // no new message
         try {
            wait();  // release the lock of this object
         } catch (InterruptedException e) { }
      }
      // acquire the lock and continue
      hasMessage = false;
      notify();
      return message + " Get @ " + System.nanoTime();
   }
}
public class TestMessageBox {
   public static void main(String[] args) {
      final MessageBox box = new MessageBox();

      Thread producerThread = new Thread() {
         @Override
         public void run() {
            System.out.println("Producer thread started...");
            for (int i = 1; i <= 6; ++i) {
               box.putMessage("message " + i);
               System.out.println("Put message " + i);
            }
         }
      };

      Thread consumerThread1 = new Thread() {
         @Override
         public void run() {
            System.out.println("Consumer thread 1 started...");
            for (int i = 1; i <= 3; ++i) {
               System.out.println("Consumer thread 1 Get " + box.getMessage());
            }
         }
      };

      Thread consumerThread2 = new Thread() {
         @Override
```

```
            public void run() {
               System.out.println("Consumer thread 2 started...");
               for (int i = 1; i <= 3; ++i) {
                  System.out.println("Consumer thread 2 Get " + box.getMessage());
               }
            }
         };

         consumerThread1.start();
         consumerThread2.start();
         producerThread.start();
      }
   }
```

The output messages (on `System.out`) may appear out-of-order. But closer inspection on the put/get timestamp confirms the correct sequence of operations.

The `synchronized` producer method `putMessage()` acquires the lock of this object, check if the previous message has been cleared. Otherwise, it calls `wait()`, releases the lock of this object, goes into `WAITING` state and places this thread on this object's "wait" set. On the other hand, the `synchronized` consumer's method `getMessage()` acquires the lock of this object and checks for new message. If there is a new message, it clears the message and issues `notify()`, which arbitrarily picks a thread on this object's "wait" set (which happens to be the producer thread in this case) and place it on `BLOCKED` state. The consumer thread, in turn, goes into the `WAITING` state and placed itself in the "wait" set of this object (after the `wait()` method). The producer thread then acquires the thread and continue its operations.


Scheduled to run
RUNNABLE
"running"
wait()
Releases lock of this object.
Places this thread on the "wait" set of this object.
Acquired the lock of this object
BLOCKED
WAITING
notifyAll()
notify() and this thread happens to be arbitrarily chosen from the "wait" set of this object.
Blocked for acquiring the lock of this object

The difference between `notify()` and `notifyAll()` is `notify()` arbitrarily picks a thread from this object's waiting pool and places it on the Seeking-lock state; while `notifyAll()` awakens all the threads in this object's waiting pool. The awaken threads then compete for execution in the normal manner.

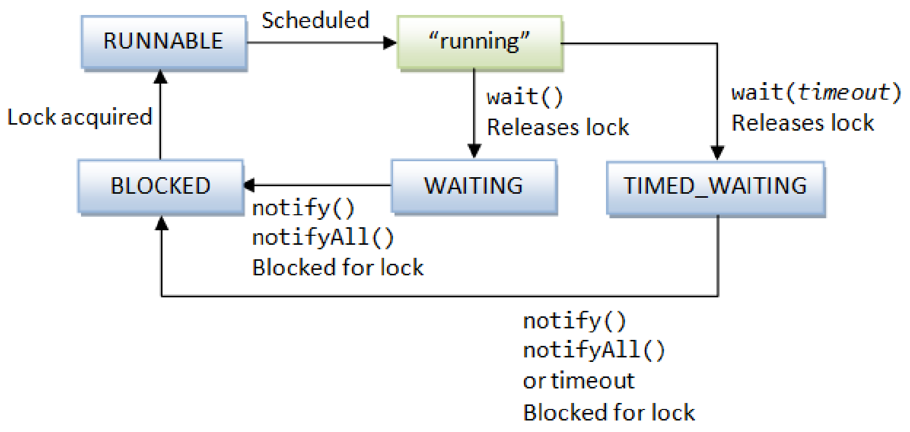
It is interesting to point out that multithreading is built into the Java language right at the root class `java.lang.Object`. The synchronization lock is kept in the `Object`. Methods `wait()`, `notify()`, `notifyAll()` used for coordinating threads are right in the class `Object`.

wait() with timeout

There are variations of `wait()` which takes in a timeout value:

```
public final void wait() throws InterruptedException
public final void wait(long timeout) throws InterruptedException
public final void wait(long timeout, int nanos) throws InterruptedException
```

The thread will ALSO go to `BLOCKED` state after the timeout expired.



Program No. 158 | Interthread communication

Program No. 158 | Interthread communication chat program

```
class Chat {
    boolean flag = false;

    public synchronized void Question(String msg) {
        if (flag) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(msg);
        flag = true;
        notify();
    }

    public synchronized void Answer(String msg) {
        if (!flag) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println(msg);
        flag = false;
        notify();
    }
}

class T1 implements Runnable {
    Chat m;
    String[] s1 = { "Hi", "How are you ?", "I am also doing fine!" };

    public T1(Chat m1) {
        this.m = m1;
        new Thread(this, "Question").start();
    }

    public void run() {
        for (int i = 0; i < s1.length; i++) {
            m.Question(s1[i]);
        }
    }
}

class T2 implements Runnable {
    Chat m;
    String[] s2 = { "Hi", "I am good, what about you?", "Great!" };

    public T2(Chat m2) {
        this.m = m2;
        new Thread(this, "Answer").start();
    }

    public void run() {
        for (int i = 0; i < s2.length; i++) {
            m.Answer(s2[i]);
        }
    }
}
public class TestThread {
    public static void main(String[] args) {
        Chat m = new Chat();
        new T1(m);
        new T2(m);
    }
}
```

Deadlock

Deadlock describes a situation where two more threads are blocked because of waiting for each other forever. When deadlock occurs, the program hangs forever and the only thing you can do is to kill the program

The program encounters a deadlock and cannot continue

```
public class Business {
      private Object lock1 = new Object();
      private Object lock2 = new Object();
      public void foo() {
            synchronized (lock1) {
                synchronized (lock2) {
                    System.out.println("foo");
                }
            }
        }
        public void bar() {
            synchronized (lock2) {
                synchronized (lock1) {
                    System.out.println("bar");
                }
            }
        }
    }
```

As you can see, both the methods `foo()` and `bar()` try to acquire two lock objects `lock1` and `lock2` but in different order.

And consider the following test program:

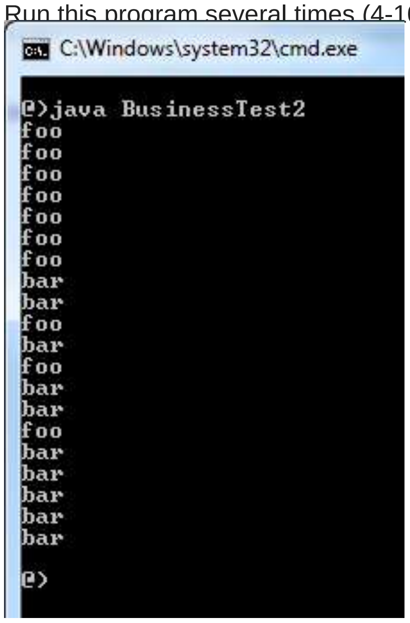
```
    public class BusinessTest1 {
        public static void main(String[] args) {
            Business business = new Business();
            Thread t1 = new Thread(new Runnable() {
                public void run() {
                    business.foo();
                }
            });
            t1.start();
            Thread t2 = new Thread(new Runnable() {
                public void run() {
                    business.bar();
                }
            });
            t2.start();
        }
    }
```

This program creates two threads, one executes the `foo()` method and another executes the `bar()` method on a shared instance of the `Business` class. But deadlock is likely never to occur because one thread can execute and exit a method very quickly so the other thread have chance to acquire the locks.

Let's modify this test program in order to create 10 threads for executing `foo()` and other 10 threads for executing `bar()` as follows:

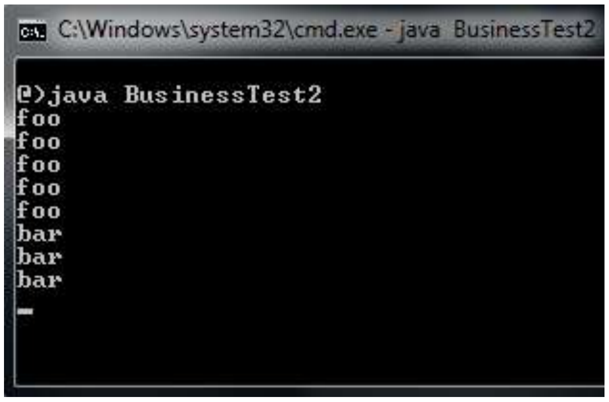
```
    public class BusinessTest2 {
        public static void main(String[] args) {
            Business business = new Business();
            for (int i = 0; i < 10; i++) {
                new Thread(new Runnable() {
                    public void run() {
                        business.foo();
                    }
                }).start();
            }
            for (int i = 0; i < 10; i++) {
                new Thread(new Runnable() {
                    public void run() {
                        business.bar();
                    }
                }).start();
            }
        }
    }
```

Run this program several times (4-10 times), you will see that sometimes the program runs fine:



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

@>java BusinessTest2
foo
foo
foo
foo
foo
foo
foo
bar
bar
foo
bar
foo
bar
bar
foo
bar
bar
bar
bar
bar

@>
```

But sometimes it hangs like this:



```
C:\Windows\system32\cmd.exe - java  BusinessTest2

@>java BusinessTest2
foo
foo
foo
foo
foo
bar
bar
bar
_
```

Why? It's because deadlock happens. Let me explain how:

- Thread 1 enters `foo()` method and it acquires `lock1`. At the same time, thread 2 enters `bar()` method and it acquires `lock2`.
- Thread 1 tries to acquire `lock2` which is currently held by thread 2, hence thread 1 blocks.
- Thread 2 tries to acquire `lock1` which is currently held by thread 1, hence thread 2 blocks.

Both threads block each other forever, deadlock occurs and the program hangs.

So how to avoid deadlock?

Java doesn't have anything to escape deadlock state when it occurs, so you have to design your program to avoid deadlock situation. Avoid acquiring more than one lock at a time. If not, make sure that you acquire multiple locks in consistent order. In the above example, you can avoid deadlock by synchronize two locks in the same order in both methods:

```
public void foo() {
      synchronized (lock1) {
            synchronized (lock2) {
                  System.out.println("foo");
            }
```

```
		}
	}
	public void bar() {
		synchronized (lock1) {
			synchronized (lock2) {
				System.out.println("bar");
			}
		}
	}
```

Also try to shrink the synchronized blocks as small as possible to avoid unnecessary locking on code that doesn't need to be synchronized.