Multi threading

Multithreading in java is a process of executing multiple threads simultaneously.

- A thread is a lightweight sub-process, the smallest unit of processing.
- Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area.
- They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.
2) You can perform many operations together, so it saves time.
3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

1. Process-based Multitasking (Multiprocessing)
2. Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

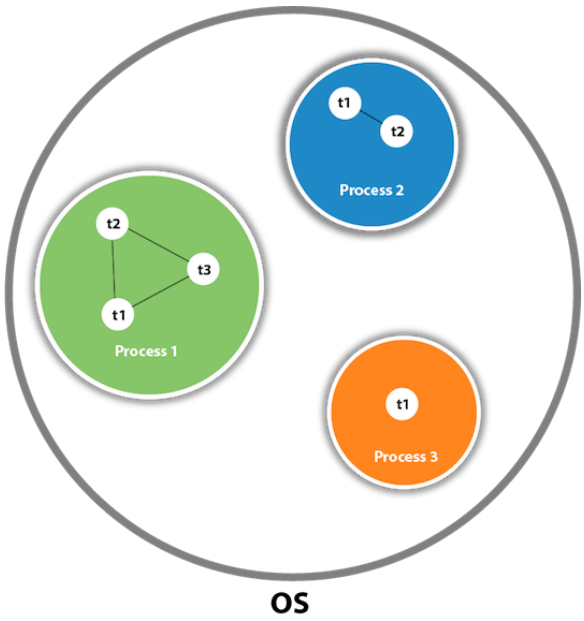
- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the processes is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.
  At least one process is required for each thread.

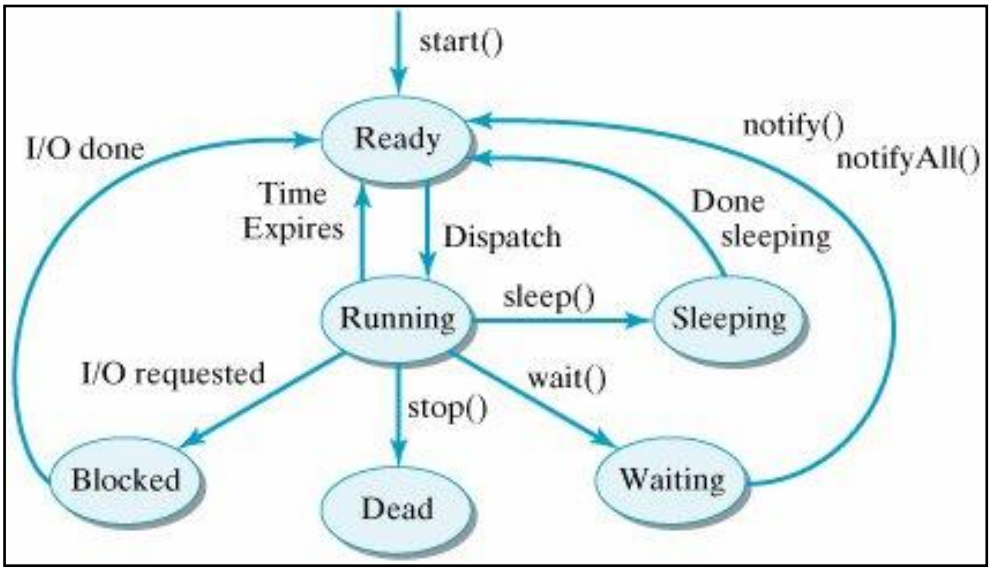
What is Thread in Java??

- A thread is a lightweight subprocess, the smallest unit of processing.
- It is a separate path of execution.
- Threads are independent. If occurs exception in one thread, it doesn't affect other threads.
- It uses a shared memory area.



- As shown in the figure, a thread is executed inside the process.
- There is **context-switching** between the threads.
- There can be multiple processes inside the OS, and one process can have multiple threads.
- **Note: At a time one thread is executed only.**

A thread can be in one of the 5 states. The life cycle of the thread is controlled by JVM.



1. NEW:
   The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
2. RUNNABLE(READY):
   The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
3. RUNNING:
   The thread is in running state if the thread scheduler has selected it.
4. NON-RUNNABLE (BLOCKED, SLEEPING or WAITING):
   This is the state when the thread is still alive, but is currently not eligible to run.
5. TERMINATED (DEAD):
   A thread is in terminated or dead state when its run() method exits.

The Main Thread

When a Java program starts up, one thread begins running immediately.
This is usually called the main thread of your program, because it is the one that is executed when your program begins.
The main thread is important for two reasons:

1. It is the thread from which other "child" threads will be spawned.
2. Often, it must be the last thread to finish execution because it performs various shutdown actions.

- Although the main thread is created automatically when your program is started, it
- can be controlled through a Thread object.
- To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread.
- Its general form is shown here: **static Thread currentThread( )**
- This method returns a reference to the thread in which it is called.
- Once you have a reference to the main thread, you can control it just like any other thread.

```
public class CurrentThreadDemo {
        public static void main(String args[]) {
                Thread t = Thread.currentThread();
                System.out.println("Current thread: " + t);
                //output of toString()i.e. "Thread[" + getName() + "," + getPriority()
                //+ "," +                    group.getName() + "]"

                // change the name of the thread
                t.setName("My Thread");
                System.out.println("After name change: " + t);
                for (int n = 5; n > 0; n--) {
                        System.out.println(n);
}   } }
```

OUTPUT:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Thread.sleep(long milli):

- Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the sleep( ) method.
- The argument to sleep( ) specifies the delay period in milliseconds.
- Notice the try/catch block around this loop. The sleep( ) method in Thread might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one.

```
public class CurrentThreadDemo {
        public static void main(String args[]) {
                Thread t = Thread.currentThread();
                System.out.println("Current thread: " + t);
                t.setName("My Thread");
                System.out.println("After name change: " + t);
                for (int n = 5; n > 0; n--) {
                        System.out.println(n);
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException ex){
}       }       }       }
```

Creating a Thread

You create a thread by instantiating an object of type Thread.
Java defines two ways in which this can be accomplished:

1. You can implement the Runnable interface.
2. You can extend the Thread class, itself.

Extending Thread

- The one way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.
- The extending class must **override the run( ) method**, which is the entry point for the new thread.
- It must also **call start( ) to begin execution** of the new thread.

- The child thread is created by instantiating an object of MyThread, which is derived from Thread.
- Notice the call to super( ) inside MyThread. This invokes the following form of the Thread constructor:**public Thread(String threadName)**
- Here, threadName specifies the name of the thread.

```
class MyThread extends Thread {
        MyThread() {
            super("Demo Thread");//calling super class constructor
            System.out.println("Child thread: " + this);
            this.start(); // Start the thread
        }
        public void run() {// This is the entry point for the second thread.
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                try {
                    Thread.sleep(500);
                } catch (InterruptedException ex) {
                }
            }
            System.out.println("Exiting child thread.");
}       }
class ThreadDemo {
        public static void main(String args[]) {
            new MyThread(); // create a new thread
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException ex) {
                }
            }
            System.out.println("Main thread exiting.");
}       }
```

OUTPUT:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Implementing Runnable

- The easiest way to create a thread is to create a class that implements the Runnable
- interface.
- Runnable abstracts a unit of executable code.
- You can construct a thread on any
- object that implements Runnable.
- To implement Runnable, a class need only implement a single method called run( ), which is declared like this: public void run( )

- Inside run( ), you will define the code that constitutes the new thread.
- It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can.
- The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within your program.
- This thread will end when run( ) returns.

- After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here: **Thread(Runnable threadOb, String threadName)**

- In this constructor, threadOb is an instance of a class that implements the Runnable interface.
- This defines where execution of the thread will begin. The name of the new thread is specified by threadName.
- After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread.
- In essence, start( ) executes a call to run( ). The start( ) method is shown here: void start( )

```
public class MyRunnableThread implements Runnable {
        Thread t;
        MyRunnableThread() {
                // Create a new, second thread:
                t = new Thread(this, "Demo Thread");
                System.out.println("Child thread: " + t);
                t.start(); // Start the thread
        }

        // This is the entry point for the second thread.
        public void run() {
                try {
                        for (int i = 5; i > 0; i--) {
                                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                }
                System.out.println("Exiting child thread.");
        }
}

public class MyRunnableThreadDemo {
        public static void main(String args[]) {
                new MyRunnableThread(); // create a new thread
                try {
                        for (int i = 5; i > 0; i--) {
                                System.out.println("Main Thread: " + i);
                                Thread.sleep(1000);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
                }
                System.out.println("Main thread exiting.");
        }
}
```

Creating and working with multiple threads

```
public class NewThread implements Runnable {
		String name; // name of thread
		Thread t;
		NewThread(String threadname) {
			name = threadname;
			t = new Thread(this, name);
			System.out.println("New thread: " + t);
			t.start(); // Start the thread
		}
		@Override
		public void run() {
			try {
				for (int i = 5; i > 0; i--) {
					System.out.println(name + ": " + i);
					Thread.sleep(1000);
				 }
			} catch (InterruptedException e) {
				System.out.println(name + "Interrupted");
			}
			System.out.println(name + " exiting.");
}		}
```

OUTPUT:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Three: 4
Two: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Three exiting.
Two exiting.
Main thread exiting
```

- As you can see, once started, all three child threads share the CPU.
- Notice the call to **sleep(10000)** in **main( )**.
- This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using isAlive( ) and join( )

- For the main thread to finish last, calling **sleep( )** within **main( )**, with a long enough delay to ensure that all child threads terminate prior to the main thread, is hardly a satisfactory solution, and it also raises a larger question:
- How can one thread know when another thread has ended?

- First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here: **final boolean isAlive( )**
- The **isAlive( )** method returns **true** if the thread upon which it is called is still running.
- It returns **false** otherwise.

- The method that you will more commonly use to wait for a thread to finish is called **join( )**, shown here: **final void join( ) throws InterruptedException**
- This method waits until the thread on which it is called terminates.
- Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.
- Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate. join(long milliseconds)

```
public class DemoIsAlive {
        public static void main(String args[]) {
                NewThread ob1 = new NewThread("One");
                NewThread ob2 = new NewThread("Two");
                NewThread ob3 = new NewThread("Three");
              // wait for threads to finish
                try {
                        System.out.println("Waiting for threads to finish.");
                        while (ob1.t.isAlive() || ob2.t.isAlive() || ob3.t.isAlive()) {
                                Thread.sleep(100);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Main thread Interrupted");
                }
                System.out.println("Main thread exiting.");
}       }
```

OUTPUT: always **Main threading exiting** is last output

```
public class DemoJoin {
	public static void main(String args[]) {
		NewThread ob1 = new NewThread("One");
		NewThread ob2 = new NewThread("Two");
		NewThread ob3 = new NewThread("Three");

		System.out.println("Thread One is alive: " + ob1.t.isAlive());
		System.out.println("Thread Two is alive: " + ob2.t.isAlive());
		System.out.println("Thread Three is alive: " + ob3.t.isAlive());

		try {// wait for threads to finish
			System.out.println("Waiting for threads to finish.");
			ob1.t.join();
			ob2.t.join();
			ob3.t.join();
		}
		 catch (InterruptedException e) {
			System.out.println("Main thread Interrupted");
		}
		System.out.println("Thread One is alive: " + ob1.t.isAlive());
		System.out.println("Thread Two is alive: " + ob2.t.isAlive());
		System.out.println("Thread Three is alive: " + ob3.t.isAlive());
		System.out.println("Main thread exiting.");
}	}
```

OUTPUT:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Three: 4
Two: 4
One: 3
Two: 3
Three: 3
One: 2
Three: 2
Two: 2
Two: 1
Three: 1
One: 1
Three exiting.
Two exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting
```

Thread Priorities

Thread priorities are used by the **thread scheduler** to decide when each thread should be allowed to run.

- In theory, over a given period of time, higher-priority threads get **more CPU time** than lower-priority threads.
- In practice, the amount of CPU time that a thread gets **often depends on several factors** besides its priority.

A higher-priority thread can also preempt a lower-priority one.

For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

In theory,

- Threads of equal priority should get equal access to the CPU.
- But, Java is designed to work in a wide range of environments some of which implement multitasking fundamentally differently than others.
- For safety, threads that share the same priority should yield control once in a while.
- This ensures that all threads have a chance to run under a **non-preemptive operating.** system.

In practice,

- Even in non-preemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O.
- When this happens, the blocked thread is suspended and other threads can run.
- But, if you want smooth multithreaded execution, you are better off not relying on this.
- For the types of threads that dominate the CPU, you want to yield control occasionally so that other threads can run.

- To set a thread's priority, **final void setPriority(int level)** can be used.
- Here, level specifies the new priority setting for the calling thread.
- The value of level must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are **1 and 10**, respectively.
- To return a thread to default priority, specify **NORM_PRIORITY**, which is currently **5**.
- These **priorities** are defined as **static final variables within Thread**.

The current priority setting can be obtained by calling **final int getPriority( )** of Thread.

- Most of the inconsistencies arise when you have threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time.
- The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called ***synchronization***.

Key to synchronization is the concept of the monitor.

- A *monitor* is an object that is used as a mutually exclusive lock.
- Only one thread can *own* a monitor at a given time.
- When a thread acquires a lock, it is said to have *entered* the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor.
- These other threads are said to be *waiting* for the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.

Code synchronization can be done in two ways.

1. **Using Synchronized Methods**

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

```
// This program is not synchronized.
class Callme {
        void call(String msg) {
                System.out.print("[" + msg);
                try {
                        Thread.sleep(1000);
                } catch(InterruptedException e) {
                        System.out.println("Interrupted");
                }
                System.out.println("]");
        }
}
```

```
class Caller implements Runnable {
	String msg;
	Callme target;
	Thread t;
	public Caller(Callme targ, String s) {
		target = targ;
		msg = s;
		t = new Thread(this);
		t.start();
	}
	public void run() {
		target.call(msg);
	}
}

class Synch {
	public static void main(String args[]) {
		Callme target = new Callme();
		Caller ob1 = new Caller(target, "Hello");
		Caller ob2 = new Caller(target, "Synchronized");
		Caller ob3 = new Caller(target, "World");
		// wait for threads to end
		try {
			ob1.t.join();
			ob2.t.join();
			ob3.t.join();
		} catch(InterruptedException e) {
			System.out.println("Interrupted");
		}
	}
}
```

OUTPUT:

```
[Synchronized[Hello[World]
]
]
```

- In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time.
- This is known as a **race condition**, because the three threads are racing each other to complete the method.

Solution is: class Callme {

synchronized void call(String msg) { }.

- This prevents other threads from entering **call( )** while another thread is using it.
- After **synchronized** has been added to **call( )**, the output of the program is as follows:

[Hello]
[Synchronized]
[World]

- Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the synchronized keyword to guard the state from race conditions.
- Once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance.
- However, non-synchronized methods on that instance will continue to be callable.

- Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, **the class does not use synchronized methods.**
- Further, this class was not created by you, but by a third party, and **you do not have access to the source code**. Thus, you **can't add synchronized** to the appropriate methods within the class.

How can access to an object of this class be synchronized?

You simply **put calls to the methods** defined by this class inside a **synchronized block**.

```
synchronized(objRef) {
        // statements to be synchronized
}
```

Here, objRef is a reference to the object being synchronized.

A synchronized block ensures that a call to a synchronized method that is a member of objRef's class occurs only after the current thread has successfully entered objRef's monitor.

```
public void run() {
                synchronized(target){
                        target.call(msg);
}               }
```

Inter Thread Communication

Unconditionally blocked other threads from asynchronous access to certain methods using of the implicit monitors in Java objects

Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time.

For example,

- Consider the classic queuing problem, where one thread is producing some data and another is consuming it.
- To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.
- In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce.
- Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on.
- Clearly, this situation is undesirable.

- To avoid polling, Java includes an elegant inter process communication mechanism via the **wait( ), notify( ), and notifyAll( ) methods**.
- These methods are implemented as final methods in Object, so all classes have them.
    - final void wait( ) throws InterruptedException
    - final void notify( )
    - final void notify All( )
    - Additional forms of wait( ) exist that allow you to specify a period of time to wait.
- **All three methods can be called only from within a synchronized context.**
- The rules for using these methods are actually quite simple:
    - **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll( ).
    - **notify( )** wakes up a thread that called wait( ) on the same object.
    - **notifyAll( )** wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

A simple form of the producer/ consumer problem consists of four classes:

1. Q, the queue that you're trying to synchronize;
2. Producer, the threaded object that is producing queue entries;
3. Consumer, the threaded object that is consuming queue entries;

4. PC, the tiny class that creates the single Q, Producer, and Consumer.

```
// An incorrect implementation of a producer and consumer.
class Q {
        int n;
        synchronized int get() {
                System.out.println("Got: " + n);
                return n;
        }
        synchronized void put(int n) {
                this.n = n;
                System.out.println("Put: " + n);
        }
}

class Producer implements Runnable {
        Q q;
        Producer(Q q) {
                this.q = q;
                new Thread(this, "Producer").start();
        }
        public void run() {
                int i = 0;
                while(true) {
                        q.put(i++);
                }
        }
}

class Consumer implements Runnable {
        Q q;
        Consumer(Q q) {
                this.q = q;
                new Thread(this, "Consumer").start();
        }
        public void run() {
                while(true) {
                        q.get();
                }
        }
}
```

```
class PC {
        public static void main(String args[]) {
                Q q = new Q();
                new Producer(q);
                new Consumer(q);
                System.out.println("Press Control-C to stop.");
        }
}
```

OUTPUT:

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

- As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row.
- Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use wait( ) and notify( ) to signal in both directions.

```
// A correct implementation of a producer and consumer.
class PCFixed {
        public static void main(String args[]) {
                Q q = new Q();
                new Producer(q);
                new Consumer(q);
                System.out.println("Press Control-C to stop.");
}       }
```

```
class Q {
        int n;
        boolean valueSet = false;
        synchronized int get() {
            while(!valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
            System.out.println("Got: " + n);
            valueSet = false;
            notify();
            return n;
        }
        synchronized void put(int n) {
            while(valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
            this.n = n;
            valueSet = true;
            System.out.println("Put: " + n);
            notify();
        }
}
class Producer implements Runnable {
        Q q;
        Producer(Q q) {
            this.q = q;
            new Thread(this, "Producer").start();
        }
        public void run() {
            int i = 0;
            while(true) {
                q.put(i++);
            }
}       }
```

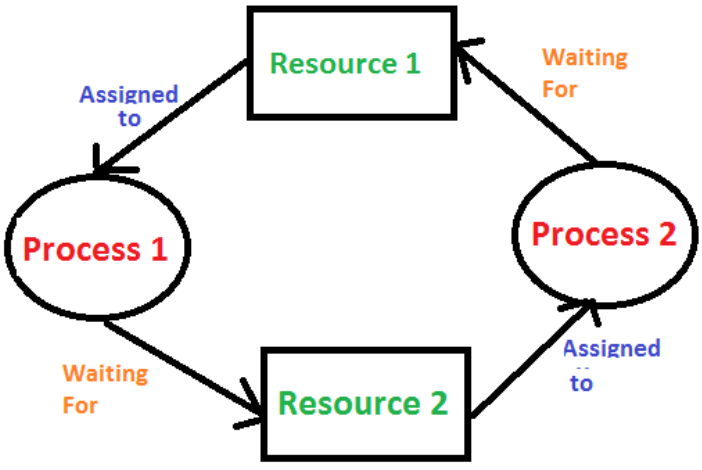
```
class Consumer implements Runnable {
        Q q;
        Consumer(Q q) {
                this.q = q;
                new Thread(this, "Consumer").start();
        }
        public void run() {
                while(true) {
                        q.get();
                }
        }
}
```

Here is some output from this program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.



Deadlock is a difficult error to debug **for two reasons**:

1. In general, it occurs only rarely, when the two threads time-slice in just the right way.
2. It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

- The next example creates two classes, A and B, with methods foo( ) and bar( ), respectively, which pause briefly before trying to call a method in the other class.
- The main class, named Deadlock, creates an A and a B instance, and then starts a second thread to set up the deadlock condition.
- The foo( ) and bar( ) methods use sleep( ) as a way to force the deadlock condition to occur.

```
class A {
        synchronized void foo(B b) {
                String name = Thread.currentThread().getName();
                System.out.println(name + " entered A.foo");
                try {
                        Thread.sleep(1000);
                } catch (Exception e) { System.out.println("A Interrupted"); }
                System.out.println(name + " trying to call B.last()");
                b.last();
        }
        synchronized void last() {
                System.out.println("Inside A.last");
}       }

class B {
        synchronized void bar(A a) {
                String name = Thread.currentThread().getName();
                System.out.println(name + " entered B.bar");
                try {
                        Thread.sleep(1000);
                } catch (Exception e) {System.out.println("B Interrupted"); }
                System.out.println(name + " trying to call A.last()");
                a.last();
        }
        synchronized void last() {
                System.out.println("Inside A.last");
}       }
```

```
public class Deadlock implements Runnable {
        A a = new A();
        B b = new B();

        Deadlock() {
                Thread.currentThread().setName("MainThread");
                Thread t = new Thread(this, "RacingThread");
                t.start();
                a.foo(b); // get lock on a in this thread.
                System.out.println("Back in main thread");
        }

        public void run() {
                b.bar(a); // get lock on b in other thread.
                System.out.println("Back in other thread");
        }

        public static void main(String args[]) {
                new Deadlock();
        }
}
```

When you run this program, you will see the output shown here:
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()

- Because the program has deadlocked, you need to press ctrl-c to end the program.
- You can see a full thread and monitor cache dump by pressing ctrl-break on a PC.
- You will see that RacingThread owns the monitor on b, while it is waiting for the monitor on a.
- At the same time, MainThread owns a and is waiting to get b.
- This program will never complete.

Suspending, Resuming, and Stopping Threads

Sometimes, suspending execution of a thread is useful.

- For example, a separate thread can be used to display the time of day.
- If the user doesn't want a clock, then its thread can be suspended.
- Whatever the case, suspending a thread is a simple matter.
- Once suspended, restarting the thread is also a simple matter.

The mechanisms to suspend, stop, and resume threads differ between early versions of Java, such as Java 1.0, and modern versions, beginning with Java 2. Prior to Java 2, a program used suspend( ), resume( ), and stop( ), which are methods defined by Thread, to pause, restart, and stop the execution of a thread.
Although these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs.

The suspend( ) method of the Thread class was deprecated by Java2 several years ago.

- This was done because suspend( ) can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures.
- If that thread is suspended at that point, those locks are not relinquished.
- Other threads that may be waiting for those resources can be deadlocked.

- **The resume( ) method** is also deprecated.
- It does not cause problems, but cannot be used without the suspend( ) method as its counterpart.

- **The stop( ) method** of the Thread class, too, was deprecated by Java2.
- This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes.
- If that thread is stopped at that point, that data structure might be left in a corrupted state.
- The trouble is that stop( ) causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.

- Instead, a thread must be designed so that the run( ) method periodically checks to determine whether that thread should suspend, resume, or stop its own execution.

- Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread.
- As long as this flag is set to “running,” the run( ) method must continue to let the thread execute.
- If this variable is set to “suspend,” the thread must pause.
- If it is set to “stop,” the thread must terminate.

The following example illustrates how **the wait( ) and notify( ) methods** that are inherited from Object can be used to control the execution of a thread.

```
class NewThread implements Runnable {
        String name; // name of thread
        Thread t;
        boolean suspendFlag;

        NewThread(String threadname) {
            name = threadname;
            t = new Thread(this, name);
            System.out.println("New thread: " + t);
            suspendFlag = false;
            t.start(); // Start the thread
        }

        public void run() {
            try {
                for(int i = 15; i > 0; i--) {
                    System.out.println(name + ": " + i);
                    Thread.sleep(200);
                    synchronized(this) {
                        while(suspendFlag) {
                            wait();
                        }
                    }
                }
            } catch (InterruptedException e) {
                System.out.println(name + " interrupted.");
            }
            System.out.println(name + " exiting.");
        }
```

```
        synchronized void mysuspend() {
            suspendFlag = true;
        }
        synchronized void myresume() {
            suspendFlag = false;
            notify();
        }
}

class SuspendResume {
        public static void main(String args[]) {
            NewThread ob1 = new NewThread("One");
            NewThread ob2 = new NewThread("Two");

            try {
                Thread.sleep(1000);
                ob1.mysuspend();
                System.out.println("Suspending thread One");
                Thread.sleep(1000);
                ob1.myresume();
                System.out.println("Resuming thread One");
                ob2.mysuspend();
                System.out.println("Suspending thread Two");
                Thread.sleep(1000);
                ob2.myresume();
                System.out.println("Resuming thread Two");
            } catch (InterruptedException e) {
                System.out.println("Main thread Interrupted");
            }
            // wait for threads to finish
            try {
                System.out.println("Waiting for threads to finish.");
                ob1.t.join();
                ob2.t.join();
            } catch (InterruptedException e) {
                System.out.println("Main thread Interrupted");
            }
            System.out.println("Main thread exiting.");
        }
}
```

Obtaining A Thread's State

- You can obtain the current state of a thread by calling the getState( ) method defined by Thread.
- It is shown here: Thread.State getState( )
- It returns a value of type Thread.State that indicates the state of the thread at the time at which the call was made. State is an enumeration defined by Thread.
- (An enumeration is a list of named constants.)
- Here are the values that can be returned by getState( ):

Sr. No.	Value	State
1	BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
2	NEW	A thread that has not begun execution.
3	RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
4	TERMINATED	A thread that has completed execution.
5	TIMED_WAITING	− A thread that has suspended execution for a specified period of time, such as when it has called sleep(). − This state is also entered when a timeout version of wait() or join( ) is called.
6	WAITING	− A thread that has suspended execution because it is waiting for some action to occur. − For example, it is waiting because of a call to a non-timeout version of wait( ) or join( ).