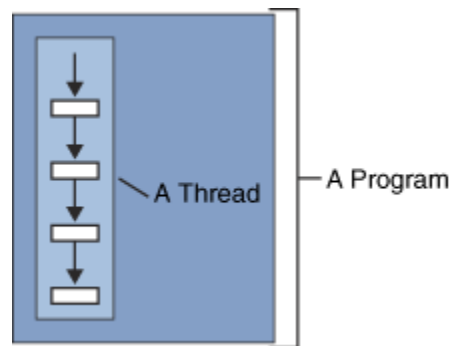


WHAT IS A THREAD?

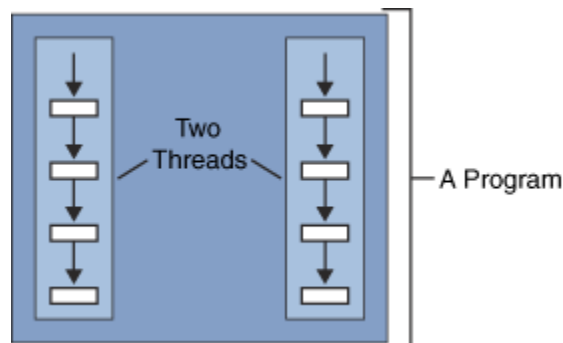
All programmers are familiar with writing sequential programs. You've probably written a program that displays "Hello World!" or sorts a list of names or computes a list of prime numbers. These are sequential programs. That is, each has a beginning, an execution sequence, and an end. At any given time during the runtime of the program, there is a single point of execution.

A thread is similar to the sequential programs described previously. A single thread also has a beginning, a sequence, and an end. At any given time during the runtime of the thread, there is a single point of execution. However, a thread itself is not a program; a thread cannot run on its own. Rather, it runs within a program. The following figure shows this relationship.



Definition: A *thread* is a single sequential flow of control within a program.

The real excitement surrounding threads is not about a single sequential thread. Rather, it's about the use of multiple threads running at the same time and performing different tasks in a single program. This use is illustrated in the next figure.



A Web browser is an example of a multithreaded application. Within a typical browser, you can scroll a page while it's downloading an applet or an image, play animation and sound concurrently, print a page in the background while you download a new page, or watch three sorting algorithms race to the finish.

Some texts call a thread a *lightweight process*. A thread is similar to a real process in that both have a single sequential flow of control. However, a thread is considered lightweight because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program's environment.

As a sequential flow of control, a thread must carve out some of its own resources within a running program. For example, a thread must have its own execution stack and program counter. The code running within the thread works only within that context. Some other texts use *execution context* as a synonym for thread.

Thread programming can be tricky, so if you think you might need to implement threads, consider using high-level thread APIs. For example, if your program must perform a task repeatedly, consider using the `java.util.Timer` class. The `Timer` class is also useful for performing a task after a delay. Examples of its use are in the section [Using the Timer and TimerTask Classes](#).

If you're writing a program with a graphical user interface (GUI), you might want to use the `javax.swing.Timer` class instead of `java.util.Timer`. Another utility class, `SwingWorker`, helps you with another common job: performing a task in a background thread, optionally updating the GUI when the task completes. You can find information about both the `Swing Timer` class and the `SwingWorker` class in [How to Use Threads](#).

Basic support for threads in the Java platform is in the class `java.lang.Thread`. It provides a thread API and provides all the generic behavior for threads. (The actual implementation of concurrent operations is system-specific. For most programming needs, the underlying implementation doesn't matter.) These behaviors include starting, sleeping, running, yielding, and having a priority.

To implement a thread using the `Thread` class, you need to provide it with a `run` method that performs the thread's task. The section [Implementing the Runnable Interface](#) tells you how to do this. The next section, [The Life Cycle of a Thread](#), discusses how to create, start, and stop a thread. The section [Thread Scheduling](#) describes how the Java platform schedules threads and how you can intervene in the scheduling.

Since threads share a program's resources, it is important to understand how to synchronize access to those resources so that the resources aren't corrupted. The section [Synchronizing Threads](#) describes how to maintain the integrity of shared resources and how to ensure that each thread has equal access to the resources.

The next section, [Thread Pools](#), discusses an approach to managing threads that relieves the programmer from worrying about the details of thread life cycles. The chapter concludes with a summary of thread support in the Java language and platform and pointers to sources of further information.

Java Threads Tutorial

• INTRODUCTION TO JAVA THREADS

Multithreading refers to two or more tasks executing concurrently within a single program. A thread is an independent path of execution within a program. Many threads can run concurrently within a program. Every thread in Java is created and controlled by the **java.lang.Thread class**. A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously.

Multithreading has several advantages over Multiprocessing such as;

- Threads are lightweight compared to processes
- Threads share the same address space and therefore can share both data and code
- Context switching between threads is usually less expensive than between processes
- Cost of thread intercommunication is relatively low that that of process intercommunication
- Threads allow different tasks to be performed concurrently.

The following figure shows the methods that are members of the Object and Thread Class.

Object	Thread
notify() notifyAll() wait()	sleep() yield()

THREAD CREATION

There are two ways to create thread in java;

- Implement the Runnable interface (java.lang.Runnable)
- By Extending the Thread class (java.lang.Thread)

IMPLEMENTING THE RUNNABLE INTERFACE

The Runnable Interface Signature

```
public interface Runnable {
```

```
void run();
```

One way to create a thread in java is to implement the Runnable Interface and then instantiate an object of the class. We need to override the run() method into our class which is the only method that needs to be implemented. The run() method contains the logic of the thread.

The procedure for creating threads based on the Runnable interface is as follows:

1. A class implements the Runnable interface, providing the run() method that will be executed by the thread. An object of this class is a Runnable object.
2. An object of Thread class is created by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run() method.
3. The start() method is invoked on the Thread object created in the previous step. The start() method returns immediately after a thread has been spawned.
4. The thread ends when the run() method ends, either by normal completion or by throwing an uncaught exception.

Below is a program that illustrates instantiation and running of threads using the runnable interface instead of extending the Thread class. To start the thread you need to invoke the **start()** method on your object.

```
class RunnableThread implements Runnable {

    Thread runner;

    public RunnableThread() {
    }

    public RunnableThread(String threadName) {
        runner = new Thread(this, threadName); // (1) Create
a new thread.
        System.out.println(runner.getName());
        runner.start(); // (2) Start the thread.
    }

    public void run() {
        //Display info about this particular thread
        System.out.println(Thread.currentThread());
    }
}

public class RunnableExample {
```

```

        public static void main(String[] args) {
            Thread thread1 = new Thread(new RunnableThread(),
"thread1");
            Thread thread2 = new Thread(new RunnableThread(),
"thread2");
            RunnableThread thread3 = new
RunnableThread("thread3");
            //Start the threads
            thread1.start();
            thread2.start();
            try {
                //delay for one second
                Thread.currentThread().sleep(1000);
            } catch (InterruptedException e) {
            }
            //Display info about the main thread
            System.out.println(Thread.currentThread());
        }
    }
}

```

Output

```

thread3
Thread[thread1,5,main]          Thread[thread2,5,main]          Thread[thread3,5,main]
Thread[main,5,main]private

```

Download Runnable Thread Program Example

This approach of creating a thread by implementing the Runnable Interface must be used whenever the class being used to instantiate the thread object is required to extend some other class.

EXTENDING THREAD CLASS

The procedure for creating threads based on extending the Thread is as follows:

1. A class extending the Thread class overrides the run() method from the Thread class to define the code executed by the thread.
2. This subclass may call a Thread constructor explicitly in its constructors to initialize the thread, using the super() call.

3. The start() method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running.

Below is a program that illustrates instantiation and running of threads by extending the Thread class instead of implementing the Runnable interface. To start the thread you need to invoke the **start()** method on your object.

```
class XThread extends Thread {

    XThread() {
    }

    XThread(String threadName) {
        super(threadName); // Initialize thread.
        System.out.println(this);
        start();
    }

    public void run() {
        //Display info about this particular thread

        System.out.println(Thread.currentThread().getName());
    }
}

public class ThreadExample {

    public static void main(String[] args) {
        Thread thread1 = new Thread(new XThread(),
"thread1");
        Thread thread2 = new Thread(new XThread(),
"thread2");
        //          The below 2 threads are assigned default
names

        Thread thread3 = new XThread();
        Thread thread4 = new XThread();
        Thread thread5 = new XThread("thread5");
        //Start the threads
        thread1.start();
        thread2.start();
```

```

        thread3.start();
        thread4.start();
        try {
            //The sleep() method is invoked on the main thread to
            cause a one second delay.
            Thread.currentThread().sleep(1000);
        } catch (InterruptedException e) {
        }
        //Display info about the main thread
        System.out.println(Thread.currentThread());
    }
}

```

Output

```

Thread[thread5,5,main] thread1
thread5
thread2
Thread-3
Thread-2
Thread[main,5,main]

```

Download Java Thread Program Example

When creating threads, there are two reasons why implementing the Runnable interface may be preferable to extending the Thread class:

- Extending the Thread class means that the subclass cannot extend any other class, whereas a class implementing the Runnable interface has this option.
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive.

An example of an anonymous class below shows how to create a thread and start it:

```

( new Thread() {

public void run() {

for(;;) System.out.println("Stop the world!");

}
}

```

```
}  
  
)start();
```

Java Threads Tutorial:

THREAD SYNCHRONIZATION

With respect to multithreading, Synchronization is a process of controlling the access of shared resources by the multiple threads in such a manner that only one thread can access a particular resource at a time.

In non synchronized multithreaded application, it is possible for one thread to modify a shared object while another thread is in the process of using or updating the object's value. Synchronization prevents such type of data corruption which may otherwise lead to dirty reads and significant errors. Generally critical sections of the code are usually marked with synchronized keyword.

Examples of using Thread Synchronization is in “The Producer/Consumer Model”.

Locks are used to synchronize access to a shared resource. A lock can be associated with a shared resource.

Threads gain access to a shared resource by first acquiring the lock associated with the object/block of code.

At any given time, at most only one thread can hold the lock and thereby have access to the shared resource.

A lock thus implements mutual exclusion.

The object lock mechanism enforces the following rules of synchronization:

- A thread must acquire the object lock associated with a shared resource, before it can enter the shared resource. The runtime system ensures that no other thread can enter a shared resource if another thread already holds the object lock associated with the shared resource. If a thread cannot immediately acquire the object lock, it is blocked, that is, it must wait for the lock to become available.

- When a thread exits a shared resource, the runtime system ensures that the object lock is also relinquished.
If another thread is waiting for this object lock, it can proceed to acquire the lock in order to gain access to the shared resource.

Classes also have a class-specific lock that is analogous to the object lock. Such a lock is actually a lock on the `java.lang.Class` object associated with the class. Given a class `A`, the reference `A.class` denotes this unique `Class` object. The class lock can be used in much the same way as an object lock to implement mutual exclusion.

There can be 2 ways through which synchronized can be implemented in Java:

- synchronized methods
- synchronized blocks

Synchronized statements are same as synchronized methods. A synchronized statement can only be executed after a thread has acquired the lock on the object/class referenced in the synchronized statement.

SYNCHRONIZED METHODS

Synchronized methods are methods that are used to control access to an object. A thread only executes a synchronized method after it has acquired the lock for the method's object or class. If the lock is already held by another thread, the calling thread waits. A thread relinquishes the lock simply by returning from the synchronized method, allowing the next thread waiting for this lock to proceed. Synchronized methods are useful in situations where methods can manipulate the state of an object in ways that can corrupt the state if executed concurrently. This is called a race condition. It occurs when two or more threads simultaneously update the same value, and as a consequence, leave the value in an undefined or inconsistent state. While a thread is inside a synchronized method of an object, all other threads that wish to execute this synchronized method or any other synchronized method of the object will have to wait until it gets the lock. This restriction does not apply to the thread that already has the lock and is executing a synchronized method of the object. Such a method can invoke other synchronized methods of the object without being blocked. The non-synchronized methods of the object can of course be called at any time by any thread.

Below is an example shows how synchronized methods and object locks are used to coordinate access to a common object by multiple threads. If the 'synchronized' keyword is removed, the message is displayed in random fashion.

```
public class SyncMethodsExample extends Thread {
```

```

        static String[] msg = { "Beginner", "java", "tutorial,",
        ".", "com",
            "is", "the", "best" };
        public SyncMethodsExample(String id) {
            super(id);
        }
        public static void main(String[] args) {
            SyncMethodsExample thread1 = new
SyncMethodsExample("thread1: ");
            SyncMethodsExample thread2 = new
SyncMethodsExample("thread2: ");
            thread1.start();
            thread2.start();
            boolean t1IsAlive = true;
            boolean t2IsAlive = true;
            do {
                if (t1IsAlive &&& !thread1.isAlive())
{
                    t1IsAlive = false;
                    System.out.println("t1 is dead.");
                }
                if (t2IsAlive &&& !thread2.isAlive())
{
                    t2IsAlive = false;
                    System.out.println("t2 is dead.");
                }
            } while (t1IsAlive || t2IsAlive);
        }
        void randomWait() {
            try {
                Thread.currentThread().sleep((long) (3000 *
Math.random()));
            } catch (InterruptedException e) {
                System.out.println("Interrupted!");
            }
        }
    }

```

```

        public synchronized void run() {
            SynchronizedOutput.displayList(getName(), msg);
        }
    }

    class SynchronizedOutput {

        // if the 'synchronized' keyword is removed, the message
        // is displayed in random fashion
        public static synchronized void displayList(String name,
String list[]) {
            for (int i = 0; i < list.length; i++) {
                SyncMethodsExample t = (SyncMethodsExample)
Thread
                    .currentThread();
                t.randomWait();
                System.out.println(name + list[i]);
            }
        }
    }
}

```

Output

```

thread1: Beginner
thread1: java
thread1: tutorial,
thread1: .,
thread1: com
thread1: is
thread1: the
thread1: best
t1 is dead.
thread2: Beginner
thread2: java
thread2: tutorial,
thread2: .,
thread2: com
thread2: is
thread2: the
thread2: best
t2 is dead.

```

[Download](#) Synchronized Methods Thread Program Example

Class Locks

SYNCHRONIZED BLOCKS

Static methods synchronize on the class lock. Acquiring and relinquishing a class lock by a thread in order to execute a static synchronized method, proceeds analogous to that of an object lock for a synchronized instance method. A thread acquires the class lock before it can proceed with the execution of any static synchronized method in the class, blocking other threads wishing to execute any such methods in the same class. This, of course, does not apply to static, non-synchronized methods, which can be invoked at any time. Synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class. A subclass decides whether the new definition of an inherited synchronized method will remain synchronized in the subclass. The synchronized block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object.

The general form of the synchronized block is as follows:

```
synchronized (<object reference expression>) {  
<code block>  
}
```

A compile-time error occurs if the expression produces a value of any primitive type. If execution of the block completes normally, then the lock is released. If execution of the block completes abruptly, then the lock is released. A thread can hold more than one lock at a time. Synchronized statements can be nested. Synchronized statements with identical expressions can be nested. The expression must evaluate to a non-null reference value, otherwise, a `NullPointerException` is thrown.

The code block is usually related to the object on which the synchronization is being done. This is the case with synchronized methods, where the execution of the method is synchronized on the lock of the current object:

```
public Object method() {  
    synchronized (this) { // Synchronized block on current object  
        // method block  
    }  
}
```

Once a thread has entered the code block after acquiring the lock on the specified object, no other thread will be able to execute the code block, or any other code requiring the same object lock, until the lock is relinquished. This happens when the execution of the code block completes normally or an uncaught exception is thrown.

Object specification in the synchronized statement is mandatory. A class can choose to synchronize the execution of a part of a method, by using the this reference and putting the relevant part of the method in the synchronized block. The braces of the block cannot be left out, even if the code block has just one statement.

```
class SmartClient {
    BankAccount account;
    // ...
    public void updateTransaction() {
        synchronized (account) { // (1) synchronized block
            account.update(); // (2)
        }
    }
}
```

In the previous example, the code at (2) in the synchronized block at (1) is synchronized on the BankAccount object. If several threads were to concurrently execute the method updateTransaction() on an object of SmartClient, the statement at (2) would be executed by one thread at a time, only after synchronizing on the BankAccount object associated with this particular instance of SmartClient.

Inner classes can access data in their enclosing context. An inner object might need to synchronize on its associated outer object, in order to ensure integrity of data in the latter. This is illustrated in the following code where the synchronized block at (5) uses the special form of the this reference to synchronize on the outer object associated with an object of the inner class. This setup ensures that a thread executing the method setPi() in an inner object can only access the private double field myPi at (2) in the synchronized block at (5), by first acquiring the lock on the associated outer object. If another thread has the lock of the associated outer object, the thread in the inner object has to wait for the lock to be relinquished before it can proceed with the execution of the synchronized block at (5). However, synchronizing on an inner object and on its associated outer object are independent of each other, unless enforced explicitly, as in the following code:

```
class Outer { // (1) Top-level Class
    private double myPi; // (2)
    protected class Inner { // (3) Non-static member Class
        public void setPi() { // (4)
            synchronized(Outer.this) { // (5) Synchronized block on outer object
                myPi = Math.PI; // (6)
            }
        }
    }
}
```

Below example shows how synchronized block and object locks are used to coordinate access to shared objects by multiple threads.

```

public class SyncBlockExample extends Thread {

    static String[] msg = { "Beginner", "java", "tutorial,",
        ".,", "com",
            "is", "the", "best" };

    public SyncBlockExample(String id) {
        super(id);
    }

    public static void main(String[] args) {
        SyncBlockExample thread1 = new
SyncBlockExample("thread1: ");
        SyncBlockExample thread2 = new
SyncBlockExample("thread2: ");
        thread1.start();
        thread2.start();
        boolean t1IsAlive = true;
        boolean t2IsAlive = true;
        do {
            if (t1IsAlive &&& !thread1.isAlive())
{
                t1IsAlive = false;
                System.out.println("t1 is dead.");
            }
            if (t2IsAlive &&& !thread2.isAlive())
{
                t2IsAlive = false;
                System.out.println("t2 is dead.");
            }
        } while (t1IsAlive || t2IsAlive);
    }

    void randomWait() {
        try {
            Thread.currentThread().sleep((long) (3000 *
Math.random()));
        } catch (InterruptedException e) {
            System.out.println("Interrupted!");
        }
    }
}

```

```

    }
    public void run() {
        synchronized (System.out) {
            for (int i = 0; i < msg.length; i++) {
                randomWait();
                System.out.println(getName() + msg[i]);
            }
        }
    }
}

```

Output

```

thread1: Beginner
thread1: java
thread1: tutorial,
thread1: .,
thread1: com
thread1: is
thread1: the
thread1: best
t1 is dead.
thread2: Beginner
thread2: java
thread2: tutorial,
thread2: .,
thread2: com
thread2: is
thread2: the
thread2: best
t2 is dead.

```

Synchronized blocks can also be specified on a class lock:

```
synchronized (<class name>.class) { <code block> }
```

The block synchronizes on the lock of the object denoted by the reference <class name>.class. A static synchronized method

classAction() in class A is equivalent to the following declaration:

```
static void classAction() {
```

```
    synchronized (A.class) { // Synchronized block on class A
```

```
// ...  
  
}  
  
}
```

In summary, a thread can hold a lock on an object

- by executing a synchronized instance method of the object
- by executing the body of a synchronized block that synchronizes on the object
- by executing a synchronized static method of a class

Java Threads Tutorial:

• THREAD STATES

A Java thread is always in one of several states which could be running, sleeping, dead, etc.

A thread can be in any of the following states:

- New Thread state (Ready-to-run state)
- Runnable state (Running state)
- Not Runnable state
- Dead state

NEW THREAD

A thread is in this state when the instantiation of a **Thread** object creates a new thread but does not start it running. A thread starts life in the Ready-to-run state. You can call only the **start()** or **stop()** methods when the thread is in this state. Calling any method besides **start()** or **stop()** causes an `IllegalThreadStateException`.

RUNNABLE

When the **start()** method is invoked on a `New Thread()` it gets to the runnable state or running state by calling the `run()` method. A Runnable thread may actually be running, or may be awaiting its turn to run.

NOT RUNNABLE

A thread becomes Not Runnable when one of the following four events occurs:

- When **sleep()** method is invoked and it sleeps for a specified amount of time
- When **suspend()** method is invoked
- When **the wait()** method is invoked and the thread waits for notification of a free resource or waits for the completion of another thread or waits to acquire a lock of an object.
- The thread is blocking on I/O and waits for its completion

Example: `Thread.currentThread().sleep(1000);`

Note: `Thread.currentThread()` may return an output like `Thread[threadA,5,main]`

The output shown in bold describes

- the name of the thread,
- the priority of the thread, and
- the name of the group to which it belongs.

Here, the **run()** method put itself to sleep for one second and becomes Not Runnable during that period.

A thread can be awakened abruptly by invoking the **interrupt()** method on the sleeping thread object or at the end of the period of time for sleep is over. Whether or not it will actually start running depends on its priority and the availability of the CPU.

Hence I hereby list the scenarios below to describe how a thread switches from a non runnable to a runnable state:

Java Threads Tutorial:

If a thread has been put to sleep, then the specified number of milliseconds must

elapse (or it must be interrupted).

- If a thread has been suspended, then its **resume()** method must be invoked
- If a thread is waiting on a condition variable, whatever object owns the variable must relinquish it by calling either **notify()** or **notifyAll()**.
- If a thread is blocked on I/O, then the I/O must complete.

DEAD STATE

A thread enters this state when the **run()** method has finished executing or when the **stop()** method is invoked. Once in this state, the thread cannot ever run again.

THREAD PRIORITY

In Java we can specify the priority of each thread relative to other threads. Those threads having higher priority get greater access to available resources than lower priority threads. A Java thread inherits its priority from the thread that created it. Heavy reliance on thread priorities for the behavior of a program can make the program non portable across platforms, as thread scheduling is host platform–dependent.

You can modify a thread's priority at any time after its creation using the **setPriority()** method and retrieve the thread priority value using **getPriority()** method.

The following static final integer constants are defined in the **Thread** class:

MIN_PRIORITY (0) Lowest Priority NORM_PRIORITY (5) Default Priority MAX_PRIORITY (10) Highest Priority

1. Another thread invokes the notify() method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened.
2. The waiting thread times out.
3. Another thread interrupts the waiting thread.

Notify

Invoking the notify() method on an object wakes up a single thread that is waiting on the lock of this object.

A call to the notify() method has no consequences if there are no threads in the wait set of the object.

The notifyAll() method wakes up all threads in the wait set of the shared object.

Below program shows three threads, manipulating the same stack. Two of them are pushing elements on the stack, while the third one is popping elements off the stack. This example illustrates how a thread waiting as a result of calling the wait() method on an object, is notified by another thread calling the notify() method on the same object

THREAD SCHEDULER

Schedulers in JVM implementations usually employ one of the two following strategies:

Preemptive scheduling

If a thread with a higher priority than all other Runnable threads becomes Runnable, the scheduler will

preempt the running thread (is moved to the runnable state) and choose the new higher priority thread for execution.

Time-Slicing or Round-Robin scheduling

A running thread is allowed to execute for a fixed length of time (a time slot it's assigned to), after which it moves to the Ready-to-run state (runnable) to await its turn to run again.

A thread scheduler is implementation and platform-dependent; therefore, how threads will be scheduled is unpredictable across different platforms.

YIELDING

A call to the static method `yield()`, defined in the Thread class, will cause the current thread in the Running state to move to the Runnable state, thus relinquishing the CPU. The thread is then at the mercy of the thread scheduler as to when it will run again. If there are no threads waiting in the Ready-to-run state, this thread continues execution. If there are other threads in the Ready-to-run state, their priorities determine which thread gets to execute. The `yield()` method gives other threads of the same priority a chance to run. If there are no equal priority threads in the "Runnable" state, then the yield is ignored.

SLEEPING AND WAKING UP

The thread class contains a static method named `sleep()` that causes the currently running thread to pause its execution and transit to the Sleeping state. The method does not relinquish any lock that the thread might have. The thread will sleep for at least the time specified in its argument, before entering the runnable state where it takes its turn to run again. If a thread is interrupted while sleeping, it will throw an `InterruptedException` when it awakes and gets to execute. The Thread class has several overloaded versions of the `sleep()` method.

WAITING AND NOTIFYING

Waiting and notifying provide means of thread inter-communication that synchronizes on the same object. The threads execute `wait()` and `notify()` (or `notifyAll()`) methods on the shared object for this purpose. The `notifyAll()`, `notify()` and `wait()` are methods of the Object class. These methods can be invoked only from within a synchronized context (synchronized method or synchronized block), otherwise, the call will result in an `IllegalMonitorStateException`. The `notifyAll()` method wakes up all the threads waiting on the resource. In this situation, the awakened threads compete for the resource. One thread gets the resource and the others go back to waiting.

`wait()` method signatures

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

The wait() call can specify the time the thread should wait before being timed out. Another thread can invoke an interrupt() method on a waiting thread resulting in an InterruptedException. This is a checked exception and hence the code with the wait() method must be enclosed within a try catch block.

notify() method signatures

final void notify()

final void notifyAll()

A thread usually calls the wait() method on the object whose lock it holds because a condition for its continued execution was not met. The thread leaves the Running state and transits to the Waiting-for-notification state. There it waits for this condition to occur. The thread relinquishes ownership of the object lock. The releasing of the lock of the shared object by the thread allows other threads to run and execute synchronized code on the same object after acquiring its lock. The wait() method causes the current thread to wait until another thread notifies it of a condition change.

A thread in the Waiting-for-notification state can be awakened by the occurrence of any one of these three incidents:

1. Another thread invokes the notify() method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened.
2. The waiting thread times out.
3. Another thread interrupts the waiting thread.

Notify

Invoking the notify() method on an object wakes up a single thread that is waiting on the lock of this object.

A call to the notify() method has no consequences if there are no threads in the wait set of the object.

The notifyAll() method wakes up all threads in the wait set of the shared object.

Below program shows three threads, manipulating the same stack. Two of them are pushing elements on the stack, while the third one is popping elements off the stack. This example illustrates how a thread waiting as a result of calling the wait() method on an object, is notified by another thread calling the notify() method on the same object.

```

class StackClass {

    private Object[] stackArray;
    private volatile int topOfStack;
    StackClass(int capacity) {
        stackArray = new Object[capacity];
        topOfStack = -1;
    }
    public synchronized Object pop() {
        System.out.println(Thread.currentThread() + ":
popping");
        while (isEmpty()) {
            try {

                System.out.println(Thread.currentThread()
                                   + ": waiting to pop");
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        Object obj = stackArray[topOfStack];
        stackArray[topOfStack--] = null;
        System.out.println(Thread.currentThread()
                           + ": notifying after pop");
        notify();
        return obj;
    }
    public synchronized void push(Object element) {
        System.out.println(Thread.currentThread() + ":
pushing");
        while (isFull()) {
            try {

                System.out.println(Thread.currentThread()

```

```

                                + ": waiting to push");
        wait();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
stackArray[++topOfStack] = element;
System.out.println(Thread.currentThread()
    + ": notifying after push");
notify();
}
public boolean isFull() {
    return topOfStack >= stackArray.length - 1;
}
public boolean isEmpty() {
    return topOfStack < 0;
}
}

abstract class StackUser extends Thread {

    protected StackClass stack;
    StackUser(String threadName, StackClass stack) {
        super(threadName);
        this.stack = stack;
        System.out.println(this);
        setDaemon(true);
        start();
    }
}

class StackPopper extends StackUser { // Stack Popper

    StackPopper(String threadName, StackClass stack) {
        super(threadName, stack);
    }
}

```

```

        public void run() {
            while (true) {
                stack.pop();
            }
        }
    }

class StackPusher extends StackUser { // Stack Pusher

    StackPusher(String threadName, StackClass stack) {
        super(threadName, stack);
    }

    public void run() {
        while (true) {
            stack.push(new Integer(1));
        }
    }
}

public class WaitAndNotifyExample {

    public static void main(String[] args) {
        StackClass stack = new StackClass(5);
        new StackPusher("One", stack);
        new StackPusher("Two", stack);
        new StackPopper("Three", stack);
        System.out.println("Main Thread sleeping.");
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Exit from Main Thread.");
    }
}

```

[Download](#) [Wait Notify methods Thread Program Example](#)

The field `topOfStack` in class `StackClass` is declared `volatile`, so that read and write operations on this variable will access the master value of this variable, and not any copies, during runtime. Since the threads manipulate the same stack object and the `push()` and `pop()` methods in the class `StackClass` are synchronized, it means that the threads synchronize on the same object.

How the program uses `wait()` and `notify()` for inter thread communication.

(1) The synchronized `pop()` method – When a thread executing this method on the `StackClass` object finds that the stack is empty, it invokes the `wait()` method in order to wait for some other thread to fill the stack by using the synchronized `push`. Once an other thread makes a `push`, it invokes the `notify` method.

(2) The synchronized `push()` method – When a thread executing this method on the `StackClass` object finds that the stack is full, it invokes the `wait()` method to await some other thread to remove an element to provide space for the newly to be pushed element. Once an other thread makes a `pop`, it invokes the `notify` method.

Java Threads Tutorial:

A thread invokes the `join()` method on another thread in order to wait for the other thread to complete its execution.

Consider a thread `t1` invokes the method `join()` on a thread `t2`. The `join()` call has no effect if thread `t2` has already completed. If thread `t2` is still alive, then thread `t1` transits to the `Blocked-for-join-completion` state.

Below is a program showing how threads invoke the overloaded thread `join` method.

```
public class ThreadJoinDemo {  
  
    public static void main(String[] args) {  
        Thread t1 = new Thread("T1");  
        Thread t2 = new Thread("T2");  
        try {  
            System.out.println("Wait for the child  
threads to finish.");  
            t1.join();  
        }  
    }  
}
```



```

        if (!t1.isAlive())
            System.out.println("Thread T1 is not
alive.");
        t2.join();
        if (!t2.isAlive())
            System.out.println("Thread T2 is not
alive.");
    } catch (InterruptedException e) {
        System.out.println("Main Thread
interrupted.");
    }
    System.out.println("Exit from Main Thread.");
}
}

```

[Download](#) Java Thread Join Method Program Example

Output

Wait for the child threads to finish.
 Thread T1 is not alive.
 Thread T2 is not alive.
 Exit from Main Thread.

DEADLOCK

There are situations when programs become deadlocked when each thread is waiting on a resource that cannot become available. The simplest form of deadlock is when two threads are each waiting on a resource that is locked by the other thread. Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever in the Blocked-for-lock-acquisition state. The threads are said to be deadlocked.

Thread t1 at tries to synchronize first on string o1 and then on string o2. The thread t2 does the opposite. It synchronizes first on string o2 then on string o1. Hence a deadlock can occur as explained above.

Below is a program that illustrates deadlocks in multithreading applications

```

public class DeadLockExample {

    String o1 = "Lock ";
    String o2 = "Step ";
}

```

```

Thread t1 = (new Thread("Printer1") {

    public void run() {
        while (true) {
            synchronized (o1) {
                synchronized (o2) {
                    System.out.println(o1 +
o2);

                }
            }
        }
    }
});

Thread t2 = (new Thread("Printer2") {

    public void run() {
        while (true) {
            synchronized (o2) {
                synchronized (o1) {
                    System.out.println(o2 +
o1);

                }
            }
        }
    }
});

public static void main(String[] args) {
    DeadLockExample dLock = new DeadLockExample();
    dLock.t1.start();
    dLock.t2.start();
}
}

```

[Download](#) Java Thread deadlock Program Example

Note: The following methods namely **join**, **sleep** and **wait** name the InterruptedException in its throws clause and can have a timeout argument as a parameter. The following methods

namely **wait**, **notify** and **notifyAll** should only be called by a thread that holds the lock of the instance on which the method is invoked. The Thread.start method causes a new thread to get ready to run at the discretion of the thread scheduler. The Runnable interface declares the run method. The Thread class implements the Runnable interface. Some implementations of the Thread.yield method will not yield to a thread of lower priority. A program will terminate only when all user threads stop running. A thread inherits its daemon status from the thread that created it

Multi-threading in java with examples

By Chaitanya Singh | Filed Under: [Multithreading](#)

Before we talk about **multithreading**, let's discuss threads. A thread is a light-weight smallest part of a process that can run concurrently with the other parts (other threads) of the same process. Threads are independent because they all have separate path of execution that's the reason if an exception occurs in one thread, it doesn't affect the execution of other threads. All threads of a process share the common memory. **The process of executing multiple threads simultaneously is known as multithreading.**

Let's summarize the discussion in points:

1. The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each such part of a program called thread.
2. Threads are lightweight sub-processes, they share the common memory space. In Multithreaded environment, programs that are benefited from multithreading, utilize the maximum CPU time so that the idle time can be kept to minimum.
3. A thread can be in one of the following states:
 - NEW – A thread that has not yet started is in this state.
 - RUNNABLE – A thread executing in the Java virtual machine is in this state.
 - BLOCKED – A thread that is blocked waiting for a monitor lock is in this state.
 - WAITING – A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
 - TIMED_WAITING – A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
 - TERMINATED – A thread that has exited is in this state.A thread can be in only one state at a given point in time.

Read more about thread states at this link: [Life cycle of threads](#)

[Multitasking vs Multithreading vs Multiprocessing vs parallel processing](#)

If you are new to java you may get confused among these terms as they are used quite frequently when we discuss multithreading. Let's talk about them in brief.

Multitasking: Ability to execute more than one task at the same time is known as multitasking.

Multithreading: We already discussed about it. It is a process of executing multiple threads simultaneously. Multithreading is also known as Thread-based Multitasking.

Multiprocessing: It is same as multitasking, however in multiprocessing more than one CPUs are involved. On the other hand one CPU is involved in multitasking.

Parallel Processing: It refers to the utilization of multiple CPUs in a single computer system.

Creating a thread in Java

There are two ways to create a thread in Java:

- 1) By extending Thread class.
- 2) By implementing Runnable interface.

Before we begin with the programs(code) of creating threads, let's have a look at these methods of Thread class. We have used few of these methods in the example below.

- `getName()`: It is used for Obtaining a thread's name
- `getPriority()`: Obtain a thread's priority
- `isAlive()`: Determine if a thread is still running
- `join()`: Wait for a thread to terminate
- `run()`: Entry point for the thread
- `sleep()`: suspend a thread for a period of time
- `start()`: start a thread by calling its `run()` method

Method 1: Thread creation by extending Thread class

Example 1:

```
class MultithreadingDemo extends Thread{
    public void run(){
        System.out.println("My thread is in running state.");
    }
    public static void main(String args[]){
        MultithreadingDemo obj=new MultithreadingDemo();
        obj.start();
    }
}
```

Output:

My thread is in running state.

Example 2:

```
class Count extends Thread
```

```

{
    Count()
    {
        super("my extending thread");
        System.out.println("my thread created" + this);
        start();
    }
    public void run()
    {
        try
        {
            for (int i=0 ;i<10;i++)
            {
                System.out.println("Printing the count " + i);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e)
        {
            System.out.println("my thread interrupted");
        }
        System.out.println("My thread run is over" );
    }
}
class ExtendingExample
{
    public static void main(String args[])
    {
        Count cnt = new Count();
        try
        {
            while(cnt.isAlive())
            {
                System.out.println("Main thread will be alive till the child
thread is live");
                Thread.sleep(1500);
            }
        }
        catch(InterruptedException e)
        {
            System.out.println("Main thread interrupted");
        }
        System.out.println("Main thread's run is over" );
    }
}

```

Output:

```

my thread createdThread[my runnable thread,5,main]
Main thread will be alive till the child thread is live
Printing the count 0
Printing the count 1
Main thread will be alive till the child thread is live
Printing the count 2
Main thread will be alive till the child thread is live
Printing the count 3

```

```
Printing the count 4
Main thread will be alive till the child thread is live
Printing the count 5
Main thread will be alive till the child thread is live
Printing the count 6
Printing the count 7
Main thread will be alive till the child thread is live
Printing the count 8
Main thread will be alive till the child thread is live
Printing the count 9
mythread run is over
Main thread run is over
```

Method 2: Thread creation by implementing Runnable Interface

A Simple Example

```
class MultithreadingDemo implements Runnable{
    public void run(){
        System.out.println("My thread is in running state.");
    }
    public static void main(String args[]){
        MultithreadingDemo obj=new MultithreadingDemo();
        Thread tobj =new Thread(obj);
        tobj.start();
    }
}
```

Output:

```
My thread is in running state.
```

Example Program 2:

Observe the output of this program and try to understand what is happening in this program. If you have understood the usage of each thread method then you should not face any issue, understanding this example.

```
class Count implements Runnable
{
    Thread mythread ;
    Count()
    {
        mythread = new Thread(this, "my runnable thread");
        System.out.println("my thread created" + mythread);
        mythread.start();
    }
    public void run()
    {
        try
        {
            for (int i=0 ;i<10;i++)
            {
                System.out.println("Printing the count " + i);
            }
        }
    }
}
```

```

        Thread.sleep(1000);
    }
}
catch (InterruptedException e)
{
    System.out.println("my thread interrupted");
}
System.out.println("mythread run is over" );
}
}
class RunnableExample
{
    public static void main(String args[])
    {
        Count cnt = new Count();
        try
        {
            while(cnt.mythread.isAlive())
            {
                System.out.println("Main thread will be alive till the child
thread is live");
                Thread.sleep(1500);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread interrupted");
        }
        System.out.println("Main thread run is over" );
    }
}

```

Output:

```

my thread createdThread[my runnable thread,5,main]
Main thread will be alive till the child thread is live
Printing the count 0
Printing the count 1
Main thread will be alive till the child thread is live
Printing the count 2
Main thread will be alive till the child thread is live
Printing the count 3
Printing the count 4
Main thread will be alive till the child thread is live
Printing the count 5
Main thread will be alive till the child thread is live
Printing the count 6
Printing the count 7
Main thread will be alive till the child thread is live
Printing the count 8
Main thread will be alive till the child thread is live
Printing the count 9
mythread run is over
Main thread run is over

```

Thread priorities

- Thread priorities are the integers which decide how one thread should be treated with respect to the others.
- Thread priority decides when to switch from one running thread to another, process is called context switching
- A thread can voluntarily release control and the highest priority thread that is ready to run is given the CPU.
- A thread can be preempted by a higher priority thread no matter what the lower priority thread is doing. Whenever a higher priority thread wants to run it does.
- To set the priority of the thread `setPriority()` method is used which is a method of the class `Thread` Class.
- In place of defining the priority in integers, we can use `MIN_PRIORITY`, `NORM_PRIORITY` or `MAX_PRIORITY`.

Methods: `isAlive()` and `join()`

- In all the practical situations main thread should finish last else other threads which have spawned from the main thread will also finish.
- To know whether the thread has finished we can call `isAlive()` on the thread which returns true if the thread is not finished.
- Another way to achieve this by using `join()` method, this method when called from the parent thread makes parent thread wait till child thread terminates.
- These methods are defined in the `Thread` class.
- We have used `isAlive()` method in the above examples too.

Synchronization

- Multithreading introduces asynchronous behavior to the programs. If a thread is writing some data another thread may be reading the same data at that time. This may bring inconsistency.
- When two or more threads need access to a shared resource there should be some way that the resource will be used only by one resource at a time. The process to achieve this is called synchronization.
- To implement the synchronous behavior java has `synchronized` method. Once a thread is inside a `synchronized` method, no other thread can call any other `synchronized` method on the same object. All the other threads then wait until the first thread come out of the `synchronized` block.
- When we want to synchronize access to objects of a class which was not designed for the multithreaded access and the code of the method which needs to be accessed synchronously is not available with us, in this case we cannot add the `synchronized` to the appropriate methods. In java we have the solution for this, put the calls to the methods (which needs to be synchronized) defined by this class inside a `synchronized` block in following manner.

```
Synchronized(object)
{
    // statement to be synchronized
}
```


Inter-thread Communication

We have few methods through which java threads can communicate with each other. These methods are `wait()`, `notify()`, `notifyAll()`. All these methods can only be called from within a synchronized method.

1) To understand synchronization java has a concept of monitor. Monitor can be thought of as a box which can hold only one thread. Once a thread enters the monitor all the other threads have to wait until that thread exits the monitor.

2) `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()`.

3) `notify()` wakes up the first thread that called `wait()` on the same object.

`notifyAll()` wakes up all the threads that called `wait()` on the same object. The highest priority thread will run first.

References

- The Complete Reference Java 2 by Herbert Schildt

Multithreading in Java Tutorial with Examples

Any application can have multiple processes (instances). Each of this process can be assigned either as a single thread or multiple threads.

We will see in this tutorial how to perform multiple tasks at the same time and also learn more about threads and synchronization between threads.

In this tutorial, we will learn-

- [What is Single Thread](#)
- [What is Multithreading](#)
- [Thread Life Cycle in Java](#)
- [Java Thread Synchronization](#)
- [Java Multithreading Example](#)

What is Single Thread?

A single thread is basically a lightweight and the smallest unit of processing. Java uses threads by using a "Thread Class".

There are two types of thread – **user thread and daemon thread** (daemon threads are used when we want to clean the application and are used in the background).

When an application first begins, user thread is created. Post that, we can create many user threads and daemon threads.

Single Thread Example:

```
package demotest;

public class GuruThread
{
    public static void main(String[] args) {
        System.out.println("Single Thread");
    }
}
```

Advantages of single thread:

- Reduces overhead in the application as single thread execute in the system
- Also, it reduces the maintenance cost of the application.

What is Multithreading?

Multithreading in java is a process of executing two or more threads simultaneously to maximum utilization of CPU.

Multithreaded applications are where two or more threads run concurrently; hence it is also known as **Concurrency** in Java. This multitasking is done, when multiple processes share common resources like CPU, memory, etc.

Each thread runs parallel to each other. Threads don't allocate separate memory area; hence it saves memory. Also, context switching between threads takes less time.

Example of Multi thread:

```
package demotest;

public class GuruThread1 implements Runnable
{
    public static void main(String[] args) {
        Thread guruThread1 = new Thread("Guru1");
        Thread guruThread2 = new Thread("Guru2");
        guruThread1.start();
        guruThread2.start();
        System.out.println("Thread names are following:");
        System.out.println(guruThread1.getName());
        System.out.println(guruThread2.getName());
    }
    @Override
    public void run() {
```

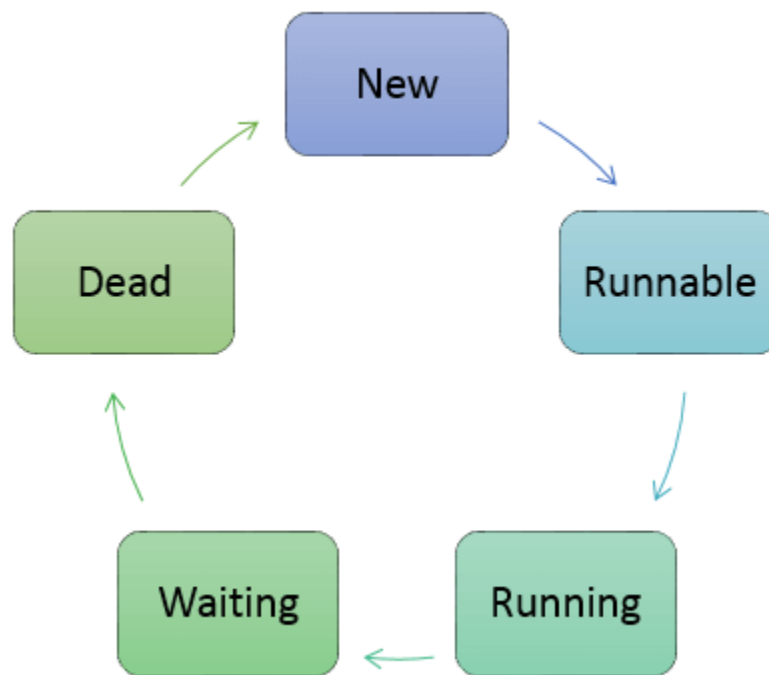
```
}  
  
}
```

Advantages of multithread:

- The users are not blocked because threads are independent, and we can perform multiple operations at times
- As such the threads are independent, the other threads won't get affected if one thread meets an exception.

Thread Life Cycle in Java

The Lifecycle of a thread:



There are various stages of life cycle of thread as shown in above diagram:

1. New
2. Runnable
3. Running
4. Waiting
5. Dead

1. **New:** In this phase, the thread is created using class "Thread class". It remains in this state till the program **starts** the thread. It is also known as born thread.

2. **Runnable:** In this page, the instance of the thread is invoked with a start method. The thread control is given to scheduler to finish the execution. It depends on the scheduler, whether to run the thread.
3. **Running:** When the thread starts executing, then the state is changed to "running" state. The scheduler selects one thread from the thread pool, and it starts executing in the application.
4. **Waiting:** This is the state when a thread has to wait. As there multiple threads are running in the application, there is a need for synchronization between threads. Hence, one thread has to wait, till the other thread gets executed. Therefore, this state is referred as waiting state.
5. **Dead:** This is the state when the thread is terminated. The thread is in running state and as soon as it completed processing it is in "dead state".

Some of the commonly used methods for threads are:

Method	Description
start()	This method starts the execution of the thread and JVM calls the run() method on the thread.
Sleep(int milliseconds)	This method makes the thread sleep hence the thread's execution will pause for milliseconds provided and after that, again the thread starts executing. This help in synchronization of the threads.
getName()	It returns the name of the thread.
setPriority(int newpriority)	It changes the priority of the thread.
yield ()	It causes current thread on halt and other threads to execute.

Example: In this example we are going to create a thread and explore built-in methods available for threads.

```
package demotest;
public class thread_example1 implements Runnable {
    @Override
    public void run() {
    }
    public static void main(String[] args) {
        Thread guruthread1 = new Thread();
        guruthread1.start();
        try {
            guruthread1.sleep(1000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        guruthread1.setPriority(1);
        int gurupriority = guruthread1.getPriority();
        System.out.println(gurupriority);
        System.out.println("Thread Running");
    }
}
```

```
}  
}
```

Explanation of the code:

Code Line 2: We are creating a class "thread_Example1" which is implementing the Runnable interface (it should be implemented by any class whose instances are intended to be executed by the thread.)

Code Line 4: It overrides run method of the runnable interface as it is mandatory to override that method

Code Line 6: Here we have defined the main method in which we will start the execution of the thread.

Code Line 7: Here we are creating a new thread name as "guruthread1" by instantiating a new class of thread.

Code Line 8: we will use "start" method of the thread using "guruthread1" instance. Here the thread will start executing.

Code Line 10: Here we are using the "sleep" method of the thread using "guruthread1" instance. Hence, the thread will sleep for 1000 milliseconds.

Code 9-14: Here we have put sleep method in try catch block as there is checked exception which occurs i.e. InterruptedException.

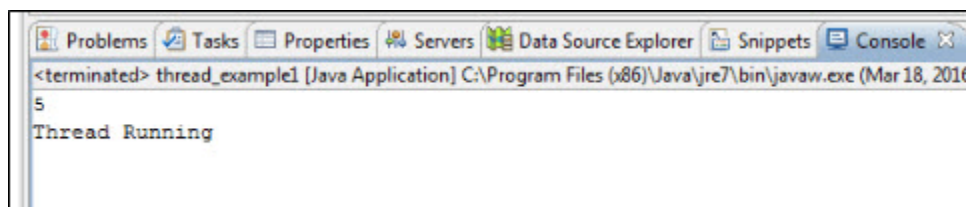
Code Line 15: Here we are setting the priority of the thread to 1 from whichever priority it was

Code Line 16: Here we are getting the priority of the thread using getPriority()

Code Line 17: Here we are printing the value fetched from getPriority

Code Line 18: Here we are writing a text that thread is running.

When you execute the above code, you get the following output:



Output:

5 is the Thread priority, and Thread Running is the text which is the output of our code.

Java Thread Synchronization

In multithreading, there is the asynchronous behavior of the programs. If one thread is writing some data and another thread which is reading data at the same time, might create inconsistency in the application.

When there is a need to access the shared resources by two or more threads, then synchronization approach is utilized.

Java has provided synchronized methods to implement synchronized behavior.

In this approach, once the thread reaches inside the synchronized block, then no other thread can call that method on the same object. All threads have to wait till that thread finishes the synchronized block and comes out of that.

In this way, the synchronization helps in a multithreaded application. One thread has to wait till other thread finishes its execution only then the other threads are allowed for execution.

It can be written in the following form:

```
Synchronized(object)
{
    //Block of statements to be synchronized
}
```

Java Multithreading Example

In this example, we will take two threads and fetch the names of the thread.

Example1:

```
GuruThread1.java
package demotest;

public class GuruThread1 implements Runnable{

    /**
     * @param args
     */
    public static void main(String[] args) {
        Thread guruThread1 = new Thread("Guru1");
        Thread guruThread2 = new Thread("Guru2");
        guruThread1.start();
        guruThread2.start();
        System.out.println("Thread names are following:");
        System.out.println(guruThread1.getName());
        System.out.println(guruThread2.getName());
    }
    @Override
```

```
        public void run() {  
        }  
    }  
}
```

Explanation of the code:

Code Line 3: We have taken a class "GuruThread1" which implements Runnable (it should be implemented by any class whose instances are intended to be executed by the thread.)

Code Line 8: This is the main method of the class

Code Line 9: Here we are instantiating the Thread class and creating an instance named as "guruThread1" and creating a thread.

Code Line 10: Here we are instantiating the Thread class and creating an instance named a "guruThread2" and creating a thread.

Code Line 11: We are starting the thread i.e. guruThread1.

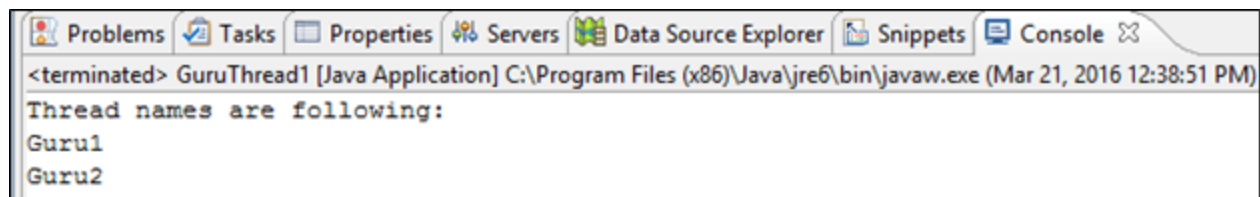
Code Line 12: We are starting the thread i.e. guruThread2.

Code Line 13: Outputting the text as "Thread names are following:"

Code Line 14: Getting the name of thread 1 using method getName() of the thread class.

Code Line 15: Getting the name of thread 2 using method getName() of the thread class.

When you execute the above code, you get the following output:



Output:

Thread names are being outputted here as

- Guru1
- Guru2

Example 2:

In this example, we will learn about overriding methods run() and start() method of a runnable interface and create two threads of that class and run them accordingly.

Also, we are taking two classes,

- One which will implement the runnable interface and
- Another one which will have the main method and execute accordingly.

```
package demotest;
public class GuruThread2 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        GuruThread3 threadguru1 = new GuruThread3("guru1");
        threadguru1.start();
        GuruThread3 threadguru2 = new GuruThread3("guru2");
        threadguru2.start();
    }
}
class GuruThread3 implements Runnable {
    Thread guruthread;
    private String guruname;
    GuruThread3(String name) {
        guruname = name;
    }
    @Override
    public void run() {
        System.out.println("Thread running" + guruname);
        for (int i = 0; i < 4; i++) {
            System.out.println(i);
            System.out.println(guruname);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Thread has been interrupted");
            }
        }
    }
    public void start() {
        System.out.println("Thread started");
        if (guruthread == null) {
            guruthread = new Thread(this, guruname);
            guruthread.start();
        }
    }
}
```

Explanation of the code:

Code Line 2: Here we are taking a class "GuruThread2" which will have the main method in it.

Code Line 4: Here we are taking a main method of the class.

Code Line 6-7: Here we are creating an instance of class `GuruThread3` (which is created in below lines of the code) as `"threadguru1"` and we are starting the thread.

Code Line 8-9: Here we are creating another instance of class `GuruThread3` (which is created in below lines of the code) as `"threadguru2"` and we are starting the thread.

Code Line 11: Here we are creating a class `"GuruThread3"` which is implementing the `Runnable` interface (it should be implemented by any class whose instances are intended to be executed by the thread.)

Code Line 13-14: we are taking two class variables from which one is of the type `Thread` class and other of the `String` class.

Code Line 15-18: we are overriding the `GuruThread3` constructor, which takes one argument as `String` type (which is thread's name) that gets assigned to class variable `guruName` and hence the name of the thread is stored.

Code Line 20: Here we are overriding the `run()` method of the `Runnable` interface.

Code Line 21: We are outputting the thread name using `println` statement.

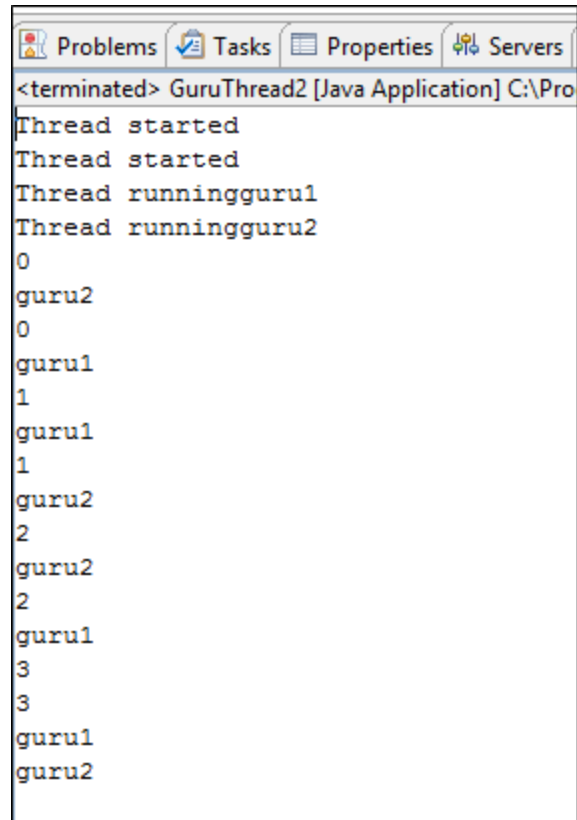
Code Line 22-31: Here we are using a `for` loop with counter initialized to 0, and it should not be less than 4 (we can take any number hence here loop will run 4 times) and incrementing the counter. We are printing the thread name and also making the thread sleep for 1000 milliseconds within a `try-catch` block as `sleep` method raised checked exception.

Code Line 33: Here we are overriding `start` method of the `Runnable` interface.

Code Line 35: We are outputting the text `"Thread started"`.

Code Line 36-40: Here we are taking an `if` condition to check whether class variable `guruThread` has value in it or no. If it's null then we are creating an instance using `Thread` class which takes the name as a parameter (value for which was assigned in the constructor). After which the thread is started using `start()` method.

When you execute the above code you get the following output:



```
<terminated> GuruThread2 [Java Application] C:\Pro
Thread started
Thread started
Thread runningguru1
Thread runningguru2
0
guru2
0
guru1
1
guru1
1
guru2
2
guru2
2
guru1
3
3
guru1
guru2
```

Output:

There are two threads hence, we get two times message "Thread started".

We get the names of the thread as we have outputted them.

It goes into for loop where we are printing the counter and thread name and counter starts with 0.

The loop executes three times and in between the thread is slept for 1000 milliseconds.

Hence, first, we get guru1 then guru2 then again guru2 because the thread sleeps here for 1000 milliseconds and then next guru1 and again guru1, thread sleeps for 1000 milliseconds, so we get guru2 and then guru1.

Summary:

In this tutorial, we saw multithreaded applications in Java and how to use single and multi threads.

- In multithreading, users are not blocked as threads are independent and can perform multiple operations at time
- Various stages of life cycle of the thread are,
 - New
 - Runnable

- Running
 - Waiting
 - Dead
- We also learned about synchronization between threads, which help the application to run smoothly.
- Multithreading makes many more application tasks easier.

Java & Threads

Threads are essentially **subprocesses**¹: informally, we can think of them as tasks that run "simultaneously" within a program. For example, a web server application may have a number of threads running at a given time, each responding to a different web page request. A graphics rendering application may have different threads running, each rendering a different portion of an image. A strategy game might have multiple threads running, each exploring different potential moves for an AI player to make.

When multiple threads are in process, the computer's operating system and processor(s) between them determine how to actually distribute the threads among the available CPU resources. If precisely two threads are running and there are two CPU cores available, then in principle, the threads can literally run simultaneously, one on each core. But the reality is usually much more complex:

- there will usually be a large number of threads competing for a small number of shared resources (CPU cores, memory, even specific components of a given CPU core...);
- at a given moment in time, a thread that is "running" may actually not be able to progress because it is waiting for some resource to become available (e.g. data from a particular source, a lock on a particular file etc).

This [complex process of allocating threads](#) to available resources is generally termed [thread scheduling](#).

Allocating threads to available CPU resources is complex in part because of the complexity of modern computer architectures, which are increasingly designed around allowing multiple processes to run in parallel. But the converse is therefore true: if your program *doesn't* make use of threads in some way, it is likely that you will not be able to make maximum use of the available computing resources.

Java threading APIs

Like modern programming languages generally, Java has a range of APIs available for managing threads within your application:

- The **Thread** class, along with associated interfaces and classes such as **Runnable**, provide the lowest level API that the programmer generally needs to deal with. This API provides simple [thread methods](#) allowing you to deal with threads in broad terms: starting them; defining the task they will run; [telling the current thread to "sleep"](#); [interrupting a thread](#); [setting a thread's priority](#) etc. This API does not define thread tasks in terms of logical concepts such as jobs, queues, time limits etc.
- The [Java executor framework](#) allows threads and their tasks to be defined in more "logical", higher-level terms (e.g. "tasks" are "scheduled" or added to [queues](#)).
- The Java Stream API provides a means to "process a collection of objects in parallel".

Getting started with the Java threading API

It's easier to illustrate what a thread is by diving straight in and seeing some code. We're going to write a program that "splits itself" into two simultaneous tasks. One task is to print `Hello, world!` every second. The other task is to print `Goodbye, cruel world!` every *two* seconds. OK, it's a silly example.

For them to run **simultaneously**, each of these two tasks will run in a **separate thread**. To define a "task", we create an instance of **Runnable**. Then we will wrap each of these `Runnable`s around a `Thread` object.

Runnable

A `Runnable` object defines an **actual task** that is to be executed. It doesn't define *how* it is to be executed (serial, two at a time, three at a time etc), but just *what*. We can define a `Runnable` as follows:

```
Runnable r = new Runnable() {
    public void run() {
        ... code to be executed ...
    }
};
```

`Runnable` is actually an interface, with the single `run()` method that we must provide. In our case, we want the `Runnable.run()` methods of our two tasks to print a message periodically. So here is what the code could look like:

```
Runnable r1 = new Runnable() {
    public void run() {
        try {
            while (true) {
                System.out.println("Hello, world!");
                Thread.sleep(1000L);
            }
        } catch (InterruptedException iex) {}
    }
};

Runnable r2 = new Runnable() {
    public void run() {
```

```

    try {
        while (true) {
            System.out.println("Goodbye, " +
                               "cruel world!");
            Thread.sleep(2000L);
        }
    } catch (InterruptedException iex) {}
}
};

```

For now, we'll gloss over a couple of issues, such as how the task ever stops. As you've probably gathered, the `Thread.sleep()` method essentially "pauses" for the given number of milliseconds, but could get "interrupted", hence the need to catch `InterruptedException`. We'll come back to this in more detail in the section on Thread interruption and [InterruptedException](#). The most important point for now is that with the `Runnable()` interface, we're just *defining* what the two tasks are. We haven't actually set them running yet. And that's where the `Thread` class comes in...

Thread

A Java `Thread` object wraps around an actual thread of execution. It effectively defines *how* the task is to be executed—namely, at the same time as other threads². To run the above two tasks simultaneously, we create a `Thread` object for each `Runnable`, then call the `start()` method on each `Thread`:

```

Thread thr1 = new Thread(r1);
Thread thr2 = new Thread(r2);
thr1.start();
thr2.start();

```

When we call `start()`, a new thread is spawned, which will begin executing the task that was assigned to the `Thread` object at some time in the near future. Meanwhile, control returns to the caller of `start()`, and we can start the second thread. Once that starts, we'll actually have at least *three* threads now running in parallel: the two we've just started, plus the "main" thread from which we created and started the two others. (In reality, the JVM will tend to have a few extra threads running for "housekeeping" tasks such as garbage collection, although they're essentially outside of our program's control.)

Next: threading topics

On the next page, we look at the following Java threading topics:

- [Constructing a thread](#): different ways for constructing a task and/or thread, and which to use when;
- [thread control methods](#) provided by the `Thread` class;
- [thread interruption](#): a mechanism for "waking a thread up" early from a blocking call;
- [stopping a thread](#): looking at how to stop a thread in Java;
- [Threads with Swing](#): how to correctly uses threads in a GUI-enabled application, including a look at the key method [SwingUtilities.invokeLater\(\)](#);

- [How threads work](#): more details about what a thread actually is, how threads are implemented by the OS, and [how thread scheduling affects Java](#), looking at issues such as limits on thread control methods, how to avoid thread overhead and lock contention;
 - [Thread scheduling](#): a look at the component of the OS which manages "juggling" threads among the available CPUs;
 - coordinating threads with a [CountDownLatch](#);
 - [Thread safety](#): an overview of data synchronization and visibility issues that occur in multithreaded programming;
 - [Thread pools in Java](#): introducing the [ThreadPoolExecutor](#) class and associated utility classes and methods;
 - we describe the problem of [deadlock](#), where two threads cannot advance as they each are holding resources the other needs to proceed; and of course, we look at [how to prevent deadlock](#);
 - thread coordination with [CyclicBarrier](#), including the example of a [parallel sort](#) algorithm.
-

1. A **process** is essentially a "heavy" unit of multitasking: as an approximation, think of it as an "application" (though it can include 'background' or 'system' processes such as your battery monitor or file indexer). A **thread**, on the other hand, is a more "lightweight" unit. Different processes generally have certain resources allocated independently of one another (such as address space, file handle allocations), whereas a threads generally share the resources of their parent process.

2. We'll see later that the `Thread` object also encapsulates other details, such as a thread's priority.

Java Thread Example

[Pankaj - 21 Comments](#)

Filed Under: [Java](#)

- [Home](#) » [Java](#) » Java Thread Example

Welcome to the Java Thread Example. **Process** and **Thread** are two basic units of execution. [Concurrency](#) programming is more concerned with java threads.

Table of Contents [[hide](#)]

- [0.1 Process](#)
- [0.2 Thread](#)
- [1 Java Thread Example](#)
 - [1.1 Java Thread Benefits](#)
 - [1.2 Java Thread Example – implementing Runnable interface](#)
 - [1.3 Java Thread Example – extending Thread class](#)
 - [1.4 Runnable vs Thread](#)

Process

A process is a self contained execution environment and it can be seen as a program or application. However a program itself contains multiple processes inside it. Java runtime environment runs as a single process which contains different classes and programs as processes.

Thread

Thread can be called *lightweight process*. Thread requires less resources to create and exists in the process, thread shares the process resources.

Java Thread Example



Every java application has at least one thread – [main thread](#). Although there are so many other java threads running in background like memory management, system management, signal processing etc. But from application point of view – main is the first java thread and we can create multiple threads from it.

[Multithreading](#) refers to two or more threads executing concurrently in a single program. A computer single core processor can execute only one thread at a time and **time slicing** is the OS feature to share processor time between different processes and threads.

Java Thread Benefits

1. Java Threads are lightweight compared to processes, it takes less time and resource to create a thread.
2. Threads share their parent process data and code

3. Context switching between threads is usually less expensive than between processes.
4. Thread intercommunication is relatively easy than process communication.

Java provides two ways to create a thread programmatically.

1. Implementing the **java.lang.Runnable** interface.
2. Extending the **java.lang.Thread** class.

Java Thread Example – implementing Runnable interface

To make a class runnable, we can implement `java.lang.Runnable` interface and provide implementation in `public void run()` method. To use this class as Thread, we need to create a Thread object by passing object of this runnable class and then call `start()` method to execute the `run()` method in a separate thread.

Here is a java thread example by implementing Runnable interface.

```
package com.journaldev.threads;

public class HeavyWorkRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("Doing heavy processing - START\n"+Thread.currentThread().getName());
        try {
            Thread.sleep(1000);
            //Get database connection, delete unused data from DB
            doDBProcessing();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Doing heavy processing - END\n"+Thread.currentThread().getName());
    }

    private void doDBProcessing() throws InterruptedException {
        Thread.sleep(5000);
    }

}
```

Java Thread Example – extending Thread class

We can extend **java.lang.Thread** class to create our own java thread class and override `run()` method. Then we can create its object and call `start()` method to execute our custom java thread class run method.

Here is a simple java thread example showing how to extend Thread class.


```

package com.journaldev.threads;

public class MyThread extends Thread {

    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println("MyThread - START
"+Thread.currentThread().getName());
        try {
            Thread.sleep(1000);
            //Get database connection, delete unused data from DB
            doDBProcessing();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("MyThread - END
"+Thread.currentThread().getName());
    }

    private void doDBProcessing() throws InterruptedException {
        Thread.sleep(5000);
    }

}

```

Here is a test program showing how to create a java thread and execute it.

```

package com.journaldev.threads;

public class ThreadRunExample {

    public static void main(String[] args){
        Thread t1 = new Thread(new HeavyWorkRunnable(), "t1");
        Thread t2 = new Thread(new HeavyWorkRunnable(), "t2");
        System.out.println("Starting Runnable threads");
        t1.start();
        t2.start();
        System.out.println("Runnable Threads has been started");
        Thread t3 = new MyThread("t3");
        Thread t4 = new MyThread("t4");
        System.out.println("Starting MyThreads");
        t3.start();
        t4.start();
        System.out.println("MyThreads has been started");
    }

}

```

Output of the above java thread example program is:

```
Starting Runnable threads
Runnable Threads has been started
Doing heavy processing - START t1
Doing heavy processing - START t2
Starting MyThreads
MyThread - START Thread-0
MyThreads has been started
MyThread - START Thread-1
Doing heavy processing - END t2
MyThread - END Thread-1
MyThread - END Thread-0
Doing heavy processing - END t1
```

Once we start any thread, it's execution depends on the OS implementation of time slicing and we can't control their execution. However we can set threads priority but even then it doesn't guarantee that higher priority thread will be executed first.

Run the above program multiple times and you will see that there is no pattern of threads start and end.

Runnable vs Thread

If your class provides more functionality rather than just running as Thread, you should implement Runnable interface to provide a way to run it as Thread. If your class only goal is to run as Thread, you can extend Thread class.

Implementing Runnable is preferred because java supports implementing multiple interfaces. If you extend Thread class, you can't extend any other classes.

Tip: As you have noticed that thread doesn't return any value but what if we want our thread to do some processing and then return the result to our client program, check our [Java Callable Future](#).

Update: From [Java 8](#) onwards, Runnable is a functional interface and we can use lambda expressions to provide it's implementation rather than using anonymous class. For more details, check out [Java 8 Functional Interfaces](#).

A Simple Thread Example

The simple example shown in full on the first page of this lesson defines two classes: SimpleThread and TwoThreadsTest. Let's begin our exploration of the application with the SimpleThread class--a subclass of the Thread class, which is provided by the java.lang package:

```
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
        }
    }
}
```

```

        try {
            sleep((int)(Math.random() * 1000));
        } catch (InterruptedException e) {}
    }
    System.out.println("DONE! " + getName());
}
}

```

The first method in the `SimpleThread` class is a constructor that takes a `String` as its only argument. This constructor is implemented by calling a superclass constructor and is interesting to us only because it sets the `Thread`'s name, which is used later in the program.

The next method in the `SimpleThread` class is the `run` method. The `run` method is the heart of any `Thread` and where the action of the `Thread` takes place. The `run` method of the `SimpleThread` class contains a `for` loop that iterates ten times. In each iteration the method displays the iteration number and the name of the `Thread`, then sleeps for a random interval of up to 1 second. After the loop has finished, the `run` method prints `DONE!` along with the name of the thread. That's it for the `SimpleThread` class.

The `TwoThreadsTest` class provides a `main` method that creates two `SimpleThread` threads: one is named "Jamaica" and the other is named "Fiji". (If you can't decide on where to go for vacation you can use this program to help you decide--go to the island whose thread prints "DONE!" first.)

```

class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}

```

The `main` method also starts each thread immediately following its construction by calling the `start` method. To save you from typing in this program, [click here](#) for the source code to the `SimpleThread` class and [here](#) for the source code to the `TwoThreadsTest` program. Compile and run the program and watch your vacation fate unfold. You should see output similar to the following:

```

0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
3 Jamaica
4 Jamaica
4 Fiji
5 Jamaica
5 Fiji
6 Fiji
6 Jamaica
7 Jamaica
7 Fiji
8 Fiji
9 Fiji
8 Jamaica

```

```
DONE! Fiji
9 Jamaica
DONE! Jamaica
```

(Looks like I'm going to Fiji!!) Notice how the output from each thread is intermingled with the output from the other. This is because both `SimpleThread` threads are running concurrently. Thus, both `run` methods are running at the same time and each thread is displaying its output at the same time as the other.

Try This: Change the main program so that it creates a third thread with the name "Bora Bora". Compile and run the program again. Does this change the island of choice for your vacation? Here's the code for the new main program, which is now named `ThreeThreadsTest`.

Keep Going

This page glosses over many of the details of threads such as the `start` and `sleep` methods. Don't worry, the next several pages of this lesson explain these concepts and others in detail. The important thing to understand from this page is that a Java program can have many threads, and that those threads can run *concurrently*.

Java - Thread Control

Advertisements

[Previous Page](#)

[Next Page](#)

Core Java provides complete control over multithreaded program. You can develop a multithreaded program which can be suspended, resumed, or stopped completely based on your requirements. There are various static methods which you can use on thread objects to control their behavior. Following table lists down those methods –

Sr.No.	Method & Description
	public void suspend()
1	This method puts a thread in the suspended state and can be resumed using <code>resume()</code> method.
	public void stop()
2	This method stops a thread completely.

3 **public void resume()**

This method resumes a thread, which was suspended using suspend() method.

4 **public void wait()**

Causes the current thread to wait until another thread invokes the notify().

5 **public void notify()**

Wakes up a single thread that is waiting on this object's monitor.

Be aware that the latest versions of Java has deprecated the usage of suspend(), resume(), and stop() methods and so you need to use available alternatives.

[Example](#)

[Live Demo](#)

```
class RunnableDemo implements Runnable {
    public Thread t;
    private String threadName;
    boolean suspended = false;

    RunnableDemo( String name) {
        threadName = name;
        System.out.println("Creating " + threadName );
    }

    public void run() {
        System.out.println("Running " + threadName );
        try {
            for(int i = 10; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Let the thread sleep for a while.
                Thread.sleep(300);
                synchronized(this) {
                    while(suspended) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }

    public void start () {
        System.out.println("Starting " + threadName );
        if (t == null) {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}
```

```

    void suspend() {
        suspended = true;
    }

    synchronized void resume() {
        suspended = false;
        notify();
    }
}

public class TestThread {

    public static void main(String args[]) {

        RunnableDemo R1 = new RunnableDemo( "Thread-1");
        R1.start();

        RunnableDemo R2 = new RunnableDemo( "Thread-2");
        R2.start();

        try {
            Thread.sleep(1000);
            R1.suspend();
            System.out.println("Suspending First Thread");
            Thread.sleep(1000);
            R1.resume();
            System.out.println("Resuming First Thread");

            R2.suspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            R2.resume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        } try {
            System.out.println("Waiting for threads to finish.");
            R1.t.join();
            R2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}

```

The above program produces the following output –

Output

```

Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 10
Running Thread-2

```

```
Thread: Thread-2, 10
Thread: Thread-1, 9
Thread: Thread-2, 9
Thread: Thread-1, 8
Thread: Thread-2, 8
Thread: Thread-1, 7
Thread: Thread-2, 7
Suspending First Thread
Thread: Thread-2, 6
Thread: Thread-2, 5
Thread: Thread-2, 4
Resuming First Thread
Suspending thread Two
Thread: Thread-1, 6
Thread: Thread-1, 5
Thread: Thread-1, 4
Thread: Thread-1, 3
Resuming thread Two
Thread: Thread-2, 3
Waiting for threads to finish.
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
Main thread exiting.
```

Java Thread by Implementing Runnable Interface

- A Thread can be created by extending Thread class also. But Java allows only one class to extend, it won't 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, you 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.

Thread Sample Code

Code:

[?](#)

```

1
2 package com.myjava.threads;
3
4 class MyRunnableThread implements Runnable{
5
6     public static int myCount = 0;
7     public MyRunnableThread(){
8
9     }
9     public void run() {
10         while(MyRunnableThread.myCount <= 10){
11             try{
12                 System.out.println("Expl Thread:
13 "+(++MyRunnableThread.myCount));
14                 Thread.sleep(100);
15             } catch (InterruptedException iex) {
16                 System.out.println("Exception in thread:
17 "+iex.getMessage());
18             }
19         }
20     }
21 }
22 public class RunMyThread {
23     public static void main(String a[]){
24         System.out.println("Starting Main Thread...");
25         MyRunnableThread mrt = new MyRunnableThread();
26         Thread t = new Thread(mrt);
27         t.start();
28         while(MyRunnableThread.myCount <= 10){
29             try{
30                 System.out.println("Main Thread:
31 "+(++MyRunnableThread.myCount));
32                 Thread.sleep(100);
33             } catch (InterruptedException iex){
34                 System.out.println("Exception in main thread:
35 "+iex.getMessage());
36             }
37         }
38         System.out.println("End of Main Thread...");
39     }
40 }

```

Example Output

```

Starting Main Thread...
Main Thread: 1
Expl Thread: 2
Main Thread: 3
Expl Thread: 4
Main Thread: 5
Expl Thread: 6
Main Thread: 7
Expl Thread: 8

```



```
Main Thread: 9
Expl Thread: 10
Main Thread: 11
End of Main Thread...
```

Java Thread by Extending Thread Class

- A thread can be created in java by extending Thread class, where you must override run() method.
- Call start() method to start executing the thread object.

Thread Sample Code

Code:

```
?
1 package com.myjava.threads;
2
3 class MySmpThread extends Thread{
4     public static int myCount = 0;
5     public void run() {
6         while(MySmpThread.myCount <= 10){
7             try{
8                 System.out.println("Expl Thread:
9 "+(++MySmpThread.myCount));
10                Thread.sleep(100);
11            } catch (InterruptedException iex) {
12                System.out.println("Exception in thread:
13 "+iex.getMessage());
14            }
15        }
16    }
17    public class RunThread {
18        public static void main(String a[]){
19            System.out.println("Starting Main Thread...");
20            MySmpThread mst = new MySmpThread();
21            mst.start();
22            while(MySmpThread.myCount <= 10){
23                try{
24                    System.out.println("Main Thread:
25 "+(++MySmpThread.myCount));
26                    Thread.sleep(100);
27                } catch (InterruptedException iex){
28                    System.out.println("Exception in main thread:
29 "+iex.getMessage());
30                }
31            }
32            System.out.println("End of Main Thread...");
33        }
34    }
35 }
```

Example Output

```
Starting Main Thread...
Main Thread: 1
Expl Thread: 2
Expl Thread: 3
Main Thread: 4
Expl Thread: 5
Main Thread: 5
Expl Thread: 6
Main Thread: 7
Main Thread: 8
Expl Thread: 9
Expl Thread: 11
Main Thread: 10
End of Main Thread...
```

Java Thread Join Examples

- Imagine the following scenario. You are preparing for tomorrow's final examination and feel a little hungry. So, you give your younger brother ten bucks and ask him to buy a pizza for you. In this case, you are the main thread and your brother is a child thread. Once your order is given, both you and your brother are doing their job concurrently (i.e., studying and buying a pizza). Now, we have two cases to consider. First, your brother brings your pizza back and terminates while you are studying. In this case, you can stop studying and enjoy the pizza. Second, you finish your study early and sleep (i.e., your assigned job for today - study for tomorrow's final exam - is done) before the pizza is available. Of course, you cannot sleep; otherwise, you won't have a chance to eat the pizza. What you are going to do is to wait until your brother brings the pizza back.
- A thread can execute a thread join to wait until the other thread terminates
- A parent thread may join with many child threads created by the parent. Or, a parent only join with some of its child threads, and ignore other child threads. In this case, those child threads that are ignored by the parent will be terminated when the parent terminates.

Thread Join Sample Code

Code:

```
?
1 package com.myjava.threads;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class MyThreadJoin {
```

```

7   public static List<String> names = new ArrayList<String>();
8
9   public static void main(String a[]){
10      List<SampleThread> list = new ArrayList<SampleThread>();
11      for(int i=0;i<5;i++){
12          SampleThread s = new SampleThread();
13          list.add(s);
14          s.start();
15      }
16      for(SampleThread st:list){
17          try{
18              st.join();
19          } catch (Exception ex){}
20      }
21      System.out.println(names);
22  }
23
24  class SampleThread extends Thread{
25      public void run(){
26          for(int i=0; i<10; i++){
27              try{
28                  Thread.sleep(10);
29              } catch (Exception ex){}
30          }
31          MyThreadJoin.names.add(getName());
32      }
33  }
34
35
36

```

Example Output

```
[Thread-0, Thread-2, Thread-1, Thread-4, Thread-3]
```

Java Thread Sleep

- It makes current executing thread to sleep specified number of milli seconds. Below example shows sample code for thread sleep.

Thread Sleep Sample Code

Code:

```

1 package com.myjava.threads;

```

```

2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class MyThreadSuspend {
7     public static void main(String a[]){
8         List<ExmpThread> list = new ArrayList<ExmpThread>();
9         for(int i=0;i<4;i++){
10             ExmpThread et = new ExmpThread(i+10);
11             list.add(et);
12             et.start();
13         }
14         for(ExmpThread et:list){
15             try {
16                 et.join();
17             } catch (InterruptedException ex) { }
18         }
19     }
20 }
21
22 class ExmpThread extends Thread{
23     private int suspendCount;
24     public ExmpThread(int count){
25         this.suspendCount = count;
26     }
27     public void run(){
28         for(int i=0;i<50;i++){
29             if(i%suspendCount == 0){
30                 try {
31                     System.out.println("Thread Sleep: " + getName());
32                     Thread.sleep(500);
33                 } catch (InterruptedException ex) { }
34             }
35         }
36     }
37 }

```

Example Output

```

Thread Sleep: Thread-0
Thread Sleep: Thread-2
Thread Sleep: Thread-1
Thread Sleep: Thread-3
Thread Sleep: Thread-0
Thread Sleep: Thread-2
Thread Sleep: Thread-1
Thread Sleep: Thread-3
Thread Sleep: Thread-2

```

```
Thread Sleep: Thread-0
Thread Sleep: Thread-1
Thread Sleep: Thread-3
Thread Sleep: Thread-0
Thread Sleep: Thread-2
Thread Sleep: Thread-1
Thread Sleep: Thread-3
Thread Sleep: Thread-2
Thread Sleep: Thread-0
Thread Sleep: Thread-1
```

Java Thread Yield Examples

- When a thread executes a thread yield, the executing thread is suspended and the CPU is given to some other runnable thread. This thread will wait until the CPU becomes available again.
- Technically, in process scheduler's terminology, the executing thread is put back into the ready queue of the processor and waits for its next turn.

Thread Yield Sample Code

Code:

```
1 package com.myjava.threads;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class MyThreadYield {
7     public static void main(String a[]){
8         List<ExmpThread> list = new ArrayList<ExmpThread>();
9         for(int i=0;i<3;i++){
10             ExmpThread et = new ExmpThread(i+5);
11             list.add(et);
12             et.start();
13         }
14         for(ExmpThread et:list){
15             try {
16                 et.join();
17             } catch (InterruptedException ex) { }
18         }
19 }
20
21 class ExmpThread extends Thread{
22     private int stopCount;
23     public ExmpThread(int count){
24         this.stopCount = count;
25     }
26 }
```

```

24     public void run() {
25         for(int i=0;i<50;i++){
26             if(i%stopCount == 0){
27                 System.out.println("Stoping thread: "+getName());
28                 yield();
29             }
30         }
31     }
32
33
34
35
36

```

Example Output

```

Stoping thread: Thread-0
Stoping thread: Thread-1
Stoping thread: Thread-3
Stoping thread: Thread-1
Stoping thread: Thread-3
Stoping thread: Thread-1
Stoping thread: Thread-3
Stoping thread: Thread-1
Stoping thread: Thread-3
Stoping thread: Thread-1
Stoping thread: Thread-2
Stoping thread: Thread-2
Stoping thread: Thread-0
Stoping thread: Thread-2
Stoping thread: Thread-0
Stoping thread: Thread-2
Stoping thread: Thread-0
Stoping thread: Thread-2
Stoping thread: Thread-0

```

Java Thread Tutorial: Creating Threads and Multithreading in Java

A brief introduction to Java Thread concepts many people find tricky

Unlike many other computer languages, Java provides built-in support for multithreading. Multithreading in Java contains two or more parts that can run [concurrently](#). A Java thread is actually a lightweight process.

This article will introduce you to all the Java Thread concepts many people find tricky or difficult to understand.

I'll be covering the following topics:

1. What is a Java Thread?
2. The Java Thread Model
3. Multithreading in Java
4. Main Java Thread
5. How to Create a Java Thread?

Before we proceed with the first topic, consider this example:

Imagine a stockbroker application with lots of complex capabilities, like

- Downloading the last stock prices
- Checking prices for warnings
- Analyzing historical data for a particular company

These are time-consuming functions. In a single-threaded runtime environment, these actions execute one after another. The next action can happen only when the previous one has finished.

If a historical analysis takes half an hour, and the user selects to perform a download and check afterward, the warning may come too late to buy or sell stock. This is the sort of application that cries out for multithreading. Ideally, the download should happen in the background (that is, in another thread). That way, other processes could happen at the same time so that, for example, a warning could be communicated instantly. All the while, the user is interacting with other parts of the application. The analysis, too, could happen in a separate thread, so the user can work with the rest of the application while the results are being calculated.

This is where a Java thread helps.

What Is a Java Thread?

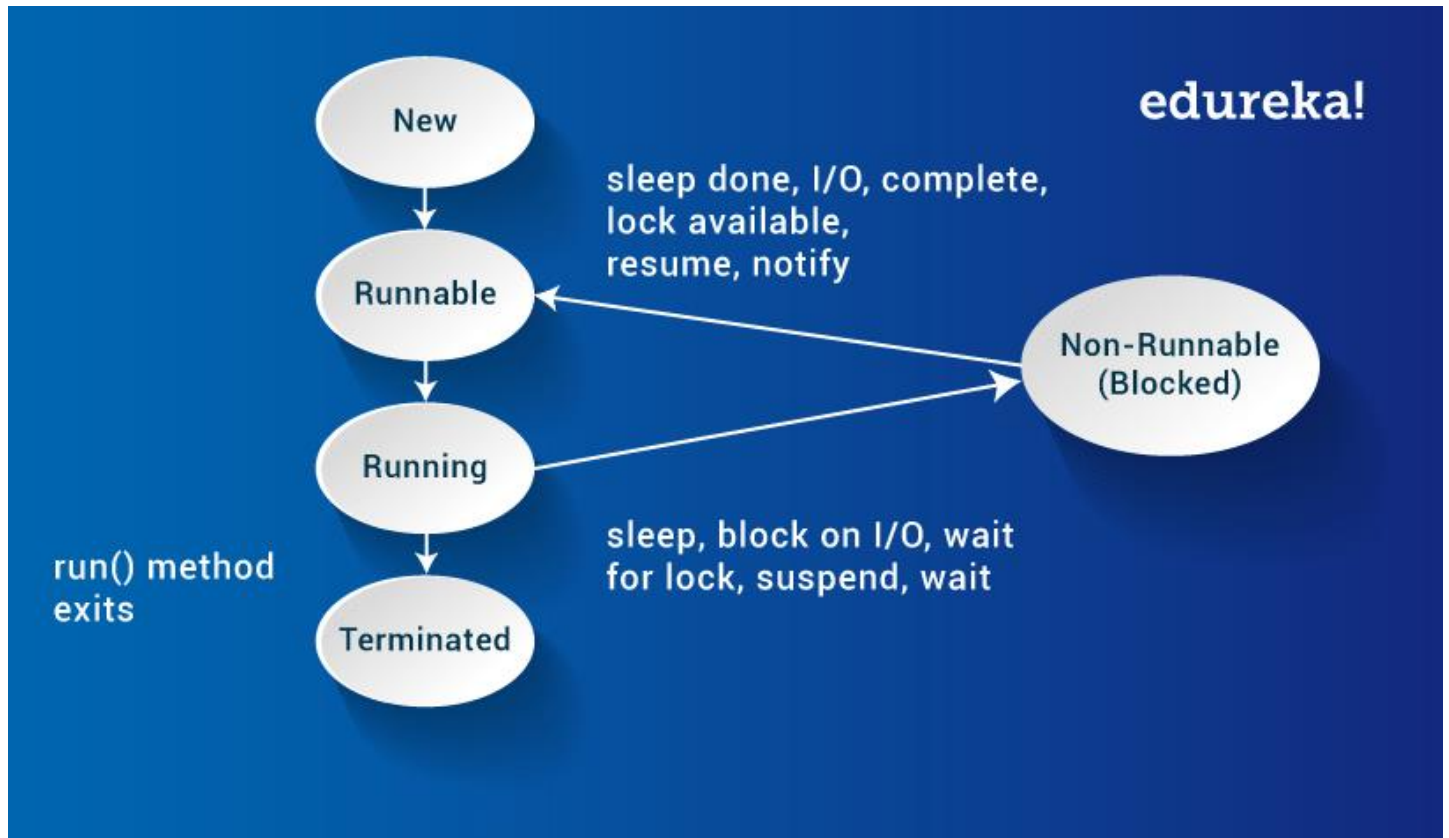
A thread is actually a lightweight process. Unlike many other computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run **concurrently**. Each part of such a program is called a thread and each thread defines a separate path of the execution. Thus, multithreading is a specialized form of multitasking.

The Java Thread Model

The Java run-time system depends on threads for many things. Threads reduce inefficiency by preventing the waste of CPU cycles.

Threads exist in several states:

- **New** - When we create an instance of Thread class, a thread is in a new state.
- **Running** - The Java thread is in running state.
- **Suspended** - A running thread can be **suspended**, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.
- **Blocked** - A Java thread can be blocked when waiting for a resource.
- **Terminated** - A thread can be terminated, which halts its execution immediately at any given time. Once a thread is terminated, it cannot be resumed.



Now let's jump to the most important topic of Java threads: thread class and runnable interface.

Multithreading in Java: Thread Class and Runnable Interface

Java's multithreading system is built upon the Thread class, its methods, and its companion interface, **Runnable**. To create a new thread, your program will either extend **Thread** or **implement** the **Runnable** interface.

The Thread class defines several methods that help manage threads:

Method Meaning

getName Obtain thread's name

getPriority	Obtain thread's priority
isAlive	Determine if a thread is still running
join	Wait for a thread to terminate
run	Entry point for the thread
sleep	Suspend a thread for a period of time
start	Start a thread by calling its run method

Now let's see how to use a Thread that begins with the **main java thread** that all Java programs have.

Main Java Thread

Here, I'll show you how to use Thread and Runnable interface to create and manage threads, beginning with the **main java thread**.

Why Is Main Thread So Important?

- Because it affects the other 'child' threads.
- Because it performs various shutdown actions.
- Because it's created automatically when your program is started.

How to Create a Java Thread

Java lets you create a thread one of two ways:

- By **implementing** the **Runnable** interface.
- By **extending** the **Thread**.

Let's look at how both ways help in implementing the Java thread.

Runnable Interface

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

To implement Runnable interface, a class need only implement a single method called run(), which is declared like this:

```
public void run( )
```

Inside run(), we will define the code that constitutes the new thread. Example:

```
public class MyClass implements Runnable {
    public void run(){
        System.out.println("MyClass running");
    }
}
```

To execute the run() method by a thread, pass an instance of MyClass to a Thread in its constructor (A **constructor in Java** is a block of code similar to a method that's called when an instance of an object is created). Here is how that is done:

```
Thread t1 = new Thread(new MyClass ());
t1.start();
```

When the thread is started it will call the run() method of the MyClass instance instead of executing its own run() method. The above example would print out the text "**MyClass running**".

Extending Java Thread

The second way to create a thread is to create a new class that extends Thread, then override the run() method and then to create an instance of that class. The run() method is what is executed by the thread after you call start(). Here is an example of creating a Java Thread subclass:

```
public class MyClass extends Thread {
    public void run(){
        System.out.println("MyClass running");
    }
}
```

To create and start the above thread:

```
MyClass t1 = new MyClass ();
t1.start();
```

When the run() method executes it will print out the text " **MyClass running** ".

So far, we have been using only two threads: the **main** thread and one **child** thread. However, our program can affect as many threads as it needs. Let's see how we can create multiple threads.

Creating Multiple Threads

```
class MyThread implements Runnable {
    String name;
    Thread t;
    MyThread(String thread){
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start();
    }
    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.");
}
}
class MultiThread {
public static void main(String args[]) {
    new MyThread("One");
    new MyThread("Two");
    new NewThread("Three");
try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
}
    System.out.println("Main thread exiting.");
}
}

```

The output from this program is shown here:

```

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
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.

```

Java Thread Methods and Thread States

Last update on September 19 2019 10:37:18 (UTC/GMT +8 hours)

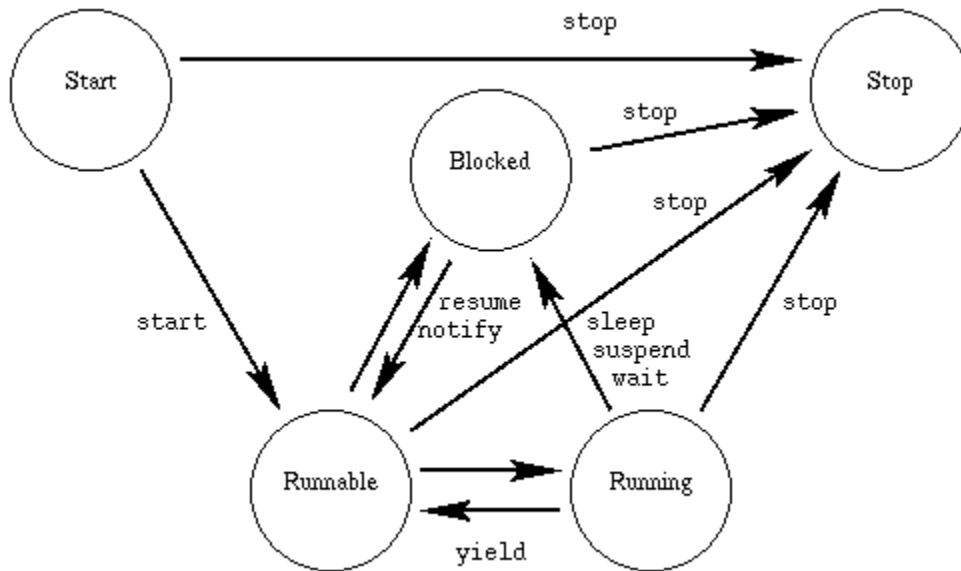
Introduction

We have various methods which can be called on Thread class object. These methods are very useful when writing a multithreaded application. Thread class has following important methods. We will understand various thread states as well later in this tutorial.

Method Signature	Description
String getName()	Retrieves the name of running thread in the current context in String format
void start()	This method will start a new thread of execution by calling run() method of Thread/runnable object.
void run()	This method is the entry point of the thread. Execution of thread starts from this method.
void sleep(int sleeptime)	This method suspend the thread for mentioned time duration in argument (sleeptime in ms)
void yield()	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
void join()	This method used to queue up a thread in execution. Once called on thread, current thread will wait till calling thread completes its execution
boolean isAlive()	This method will check if thread is alive or dead

Thread States

The thread scheduler's job is to move threads in and out of the the running state. While the thread scheduler can move a thread from the running state back to runnable, other factors can cause a thread to move out of running, but not back to runnable. One of these is when the thread's run() method completes, in which case the thread moves from the running state directly to the dead state.



New/Start:

This is the state the thread is in after the Thread instance has been created, but the start() method has not been invoked on the thread. It is a live Thread object, but not yet a thread of execution. At this point, the thread is considered not alive.

Runnable:

This means that a thread can be run when the time-slicing mechanism has CPU cycles available for the thread. Thus, the thread might or might not be running at any moment, but there's nothing to prevent it from being run if the scheduler can arrange it. That is, it's not dead or blocked.

Running:

This state is important state where the action is. This is the state a thread is in when the thread scheduler selects it (from the runnable pool) to be the currently executing process. A thread can transition out of a running state for several reasons, including because "the thread scheduler felt like it". There are several ways to get to the runnable state, but only one way to get to the running state: the scheduler chooses a thread from the runnable pool of thread.

Blocked:

The thread can be run, but something prevents it. While a thread is in the blocked state, the scheduler will simply skip it and not give it any CPU time. Until a thread reenters the runnable state, it won't perform any operations. Blocked state has some sub-states as below,

- **Blocked on I/O:** The thread waits for completion of blocking operation. A thread can enter this state because of waiting I/O resource. In that case, the thread sends back to runnable state after the availability of resources.

- **Blocked for join completion:** The thread can come in this state because of waiting for the completion of another thread.
- **Blocked for lock acquisition:** The thread can come in this state because of waiting for acquire the lock of an object.

Dead:

A thread in the dead or terminated state is no longer schedulable and will not receive any CPU time. Its task is completed, and it is no longer runnable. One way for a task to die is by returning from its `run()` method, but a task's thread can also be interrupted, as you'll see shortly.

Let's take an example of Java program to demonstrate various thread state and methods of thread class.

Java Code (AnimalRunnable.java)

```
package threadstates;
public class AnimalRunnable implements Runnable {
    @Override
    public void run() {
        for (int x = 1; x < 4; x++) {
            System.out.println("Run by " +
Thread.currentThread().getName());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
        System.out.println("Thread State of: "+
Thread.currentThread().getName()+ " - "+Thread.currentThread().getState());
        System.out.println("Exit of Thread: "
            + Thread.currentThread().getName());
    }
}
```

Copy

Java Code (AnimalMultiThreadDemo.java): [Go to the editor](#)

```
public class AnimalMultiThreadDemo {
    public static void main(String[] args) throws Exception{
        // Make object of Runnable
        AnimalRunnable anr = new AnimalRunnable();
        Thread cat = new Thread(anr);
        cat.setName("Cat");
        Thread dog = new Thread(anr);
        dog.setName("Dog");
        Thread cow = new Thread(anr);
        cow.setName("Cow");
        System.out.println("Thread State of Cat before calling start:
"+cat.getState());
```

```

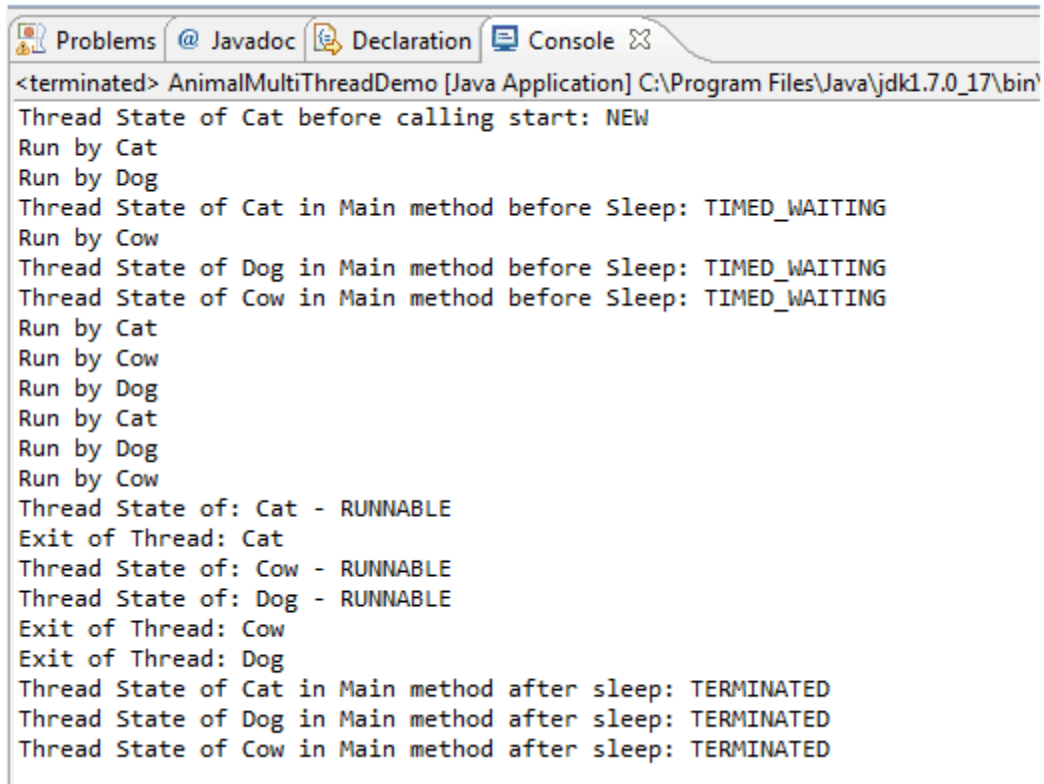
        cat.start();
        dog.start();
        cow.start();
        System.out.println("Thread State of Cat in Main method before
Sleep: " + cat.getState());
        System.out.println("Thread State of Dog in Main method before
Sleep: " + dog.getState());
        System.out.println("Thread State of Cow in Main method before
Sleep: " + cow.getState());
        Thread.sleep(10000);
        System.out.println("Thread State of Cat in Main method after
sleep: " + cat.getState());
        System.out.println("Thread State of Dog in Main method after
sleep: " + dog.getState());
        System.out.println("Thread State of Cow in Main method after
sleep: " + cow.getState());
    }
}

class AnimalRunnable implements Runnable {
    @Override
    public void run() {
        for (int x = 1; x < 4; x++) {
            System.out.println("Run by " +
Thread.currentThread().getName());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
        System.out.println("Thread State of: "+
Thread.currentThread().getName()+ " - "+Thread.currentThread().getState());
        System.out.println("Exit of Thread: "
+ Thread.currentThread().getName());
    }
}

```

Copy

Output:



```
<terminated> AnimalMultiThreadDemo [Java Application] C:\Program Files\Java\jdk1.7.0_17\bin'
Thread State of Cat before calling start: NEW
Run by Cat
Run by Dog
Thread State of Cat in Main method before Sleep: TIMED_WAITING
Run by Cow
Thread State of Dog in Main method before Sleep: TIMED_WAITING
Thread State of Cow in Main method before Sleep: TIMED_WAITING
Run by Cat
Run by Cow
Run by Dog
Run by Cat
Run by Dog
Run by Cow
Thread State of: Cat - RUNNABLE
Exit of Thread: Cat
Thread State of: Cow - RUNNABLE
Thread State of: Dog - RUNNABLE
Exit of Thread: Cow
Exit of Thread: Dog
Thread State of Cat in Main method after sleep: TERMINATED
Thread State of Dog in Main method after sleep: TERMINATED
Thread State of Cow in Main method after sleep: TERMINATED
```

Summary:

- Once a new thread is started, it will always enter the runnable state.
- The thread scheduler can move a thread back and forth between the runnable state and the running state.
- For a typical single-processor machine, only one thread can be running at a time, although many threads may be in the runnable state.
- There is no guarantee that the order in which threads were started determines the order in which they'll run. There's no guarantee that threads will take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation.
- A running thread may enter a blocked/waiting state by a `wait()`, `sleep()`, or `join()` call.

Java Code Editor:

Previous: [Java Defining, Instantiating and Starting Thread](#)

Next: [Java Thread Interaction](#)

- **New Content published on w3resource :**
- [Python Numpy exercises](#)
- [Python GeoPy Package exercises](#)
- [Python Pandas exercises](#)

- [Python nltk exercises](#)
 - [Python BeautifulSoup exercises](#)
 - [Form Template](#)
 - [Composer - PHP Package Manager](#)
 - [PHPUnit - PHP Testing](#)
 - [Laravel - PHP Framework](#)
 - [Angular - JavaScript Framework](#)
 - [React - JavaScript Library](#)
 - [Vue - JavaScript Framework](#)
 - [Jest - JavaScript Testing Framework](#)
-

Java Defining, Instantiating, and Starting Threads

Last update on September 19 2019 10:37:18 (UTC/GMT +8 hours)

Introduction

One of the most appealing features in Java is the support for easy thread programming. Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus we can say that multithreading is a specialized form of multitasking.

The formal definition of a thread is, A thread is a basic processing unit to which an operating system allocates processor time, and more than one thread can be executing code inside a process. A thread is sometimes called a lightweight process or an execution context

Imagine an online ticket reservation application with a lot of complex capabilities. One of its functions is "search for train/flight tickets from source and destination" another is "check for prices and availability," and a third time-consuming operation is "ticket booking for multiple clients at a time".

In a single-threaded runtime environment, these actions execute one after another. The next action can happen only when the previous one is finished. If a ticket booking takes 10 mins, then other users have to wait for their search operation or book operation. This kind of application will result into waste of time and clients. To avoid this kind of problems java provides multithreading features where multiple operations can take place simultaneously and faster response can be achieved for better user experience. Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Defining a Thread

In the most general sense, you create a thread by instantiating an object of type Thread.

Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class

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 implements 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.

```
class MyRunnableThread implements Runnable {  
public void run() {  
System.out.println("Important job running in MyRunnableThread");  
}  
}
```

Extending java.lang.Thread

The simplest way to define code to run in a separate thread is to

- Extend the java.lang.Thread class.
- Override the run() method.

It looks like this:

```
class MyThread extends Thread {  
public void run() {  
System.out.println("Important job running in MyThread");  
}  
}
```

The limitation with this approach (besides being a poor design choice in most cases) is that if you extend Thread, you can't extend anything else. And it's not as if you really need that inherited Thread class behavior because in order to use a thread you'll need to instantiate one anyway.

Instantiating a Thread

Remember, every thread of execution begins as an instance of class Thread. Regardless of whether your run() method is in a Thread subclass or a Runnable implementation class, you still need a Thread object to do the work.

If you have approach two (extending Thread class): Instantiation would be simple

```
MyThread thread = new MyThread();
```

If you implement Runnable, instantiation is only slightly less simple.

To instantiate your Runnable class:

```
MyRunnableThreadmyRunnable = new MyRunnableThread ();  
Thread thread = new Thread(myRunnable); // Pass your Runnable to the Thread
```

Giving the same target to multiple threads means that several threads of execution will be running the very same job (and that the same job will be done multiple times).

Thread Class Constructors

- Thread() :

Default constructor – To create thread with default name and priority

- Thread(Runnable target)

This constructor will create a thread from the runnable object.

- Thread(Runnable target, String name)

This constructor will create thread from runnable object with name as passed in the second argument

- Thread(String name)

This constructor will create a thread with the name as per argument passed.

So now we've made a Thread instance, and it knows which run() method to call. But nothing is happening yet. At this point, all we've got is a plain old Java object of type Thread. It is not yet a thread of execution. To get an actual thread—a new call stack—we still have to start the thread.

Starting a Thread

You've created a Thread object and it knows its target (either the passed-in Runnable or itself if you extended class Thread). Now it's time to get the whole thread thing happening—to launch a new call stack. It's so simple it hardly deserves its own subheading:

```
t.start();
```

Prior to calling start() on a Thread instance, the thread is said to be in the new state. There are various thread states which we will cover in next tutorial.

When we call t.start() method following things happens:

- A new thread of execution starts (with a new call stack).
- The thread moves from the new state to the runnable state.
- When the thread gets a chance to execute, its target run() method will run.

The following example demonstrates what we've covered so far—defining, instantiating, and starting a thread: In below Java program we are not implementing thread communication or synchronization, because of that output may depend on operating system's scheduling mechanism and JDK version.

We are creating two threads t1 and t2 of MyRunnable class object. Starting both threads, each thread is printing thread name in the loop.

Java Code (MyRunnable.java)

```
package mythreading;
public class MyRunnable implements Runnable{
    @Override
    public void run() {
        for(int x =1; x < 10; x++) {
            System.out.println("MyRunnable running for Thread Name:
" + Thread.currentThread().getName());
        }
    }
}
```

Copy

Java Code (TestMyRunnable.java)

```
package mythreading;
public class TestMyRunnable {
    public static void main (String [] args) {
        MyRunnable myrunnable = new MyRunnable();
        //Passing myrunnable object to Thread class constructor
        Thread t1 = new Thread(myrunnable);
        t1.setName("Amit-1 Thread");
        //Starting Thread t1
    }
}
```

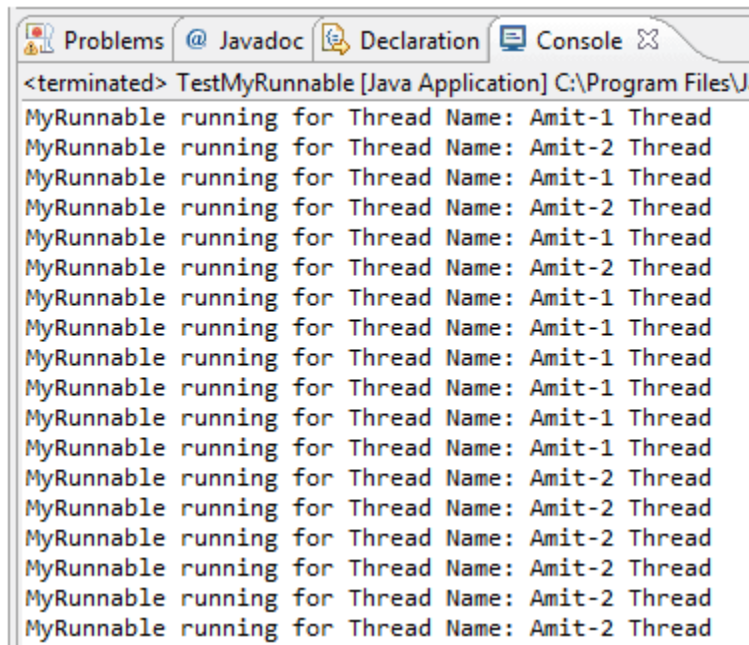
```

        t1.start();
        Thread t2 = new Thread(myrunnable);
        t2.setName("Amit-2 Thread");
        t2.start();
    }
}

```

Copy

Output:



```

<terminated> TestMyRunnable [Java Application] C:\Program Files\J
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread

```

Summary:

- Threads can be created by extending Thread and overriding the public void run() method
- Thread objects can also be created by calling the Thread constructor that takes a Runnable argument. The Runnable object is said to be the target of the thread.
- You can call start() on a Thread object only once. If start() is called more than once on a Thread object, it will throw a Runtime Exception.
- Each thread has its own call stack which is storing state of thread execution
- When a Thread object is created, it does not become a thread of execution until its start() method is invoked. When a Thread object exists but hasn't been started, it is in the new state and is not considered alive.

Java Code Editor:

Previous: [Java Utility Class](#)

Next: [Java Thread States and Transitions](#)

Java Defining, Instantiating, and Starting Threads

Introduction

One of the most appealing features in Java is the support for easy thread programming. Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus we can say that multithreading is a specialized form of multitasking.

The formal definition of a thread is, A thread is a basic processing unit to which an operating system allocates processor time, and more than one thread can be executing code inside a process. A thread is sometimes called a lightweight process or an execution context

Imagine an online ticket reservation application with a lot of complex capabilities. One of its functions is "search for train/flight tickets from source and destination" another is "check for prices and availability," and a third time-consuming operation is "ticket booking for multiple clients at a time".

In a single-threaded runtime environment, these actions execute one after another. The next action can happen only when the previous one is finished. If a ticket booking takes 10 mins, then other users have to wait for their search operation or book operation. This kind of application will result into waste of time and clients. To avoid this kind of problems java provides multithreading features where multiple operations can take place simultaneously and faster response can be achieved for better user experience. Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Defining a Thread

In the most general sense, you create a thread by instantiating an object of type Thread.

Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class

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 implements 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.

```
class MyRunnableThread implements Runnable {
public void run() {
System.out.println("Important job running in MyRunnableThread");
}
}
```

Extending `java.lang.Thread`

The simplest way to define code to run in a separate thread is to

- Extend the `java.lang.Thread` class.
- Override the `run()` method.

It looks like this:

```
class MyThread extends Thread {
public void run() {
System.out.println("Important job running in MyThread");
}
}
```

The limitation with this approach (besides being a poor design choice in most cases) is that if you extend `Thread`, you can't extend anything else. And it's not as if you really need that inherited `Thread` class behavior because in order to use a thread you'll need to instantiate one anyway.

Instantiating a Thread

Remember, every thread of execution begins as an instance of class `Thread`. Regardless of whether your `run()` method is in a `Thread` subclass or a `Runnable` implementation class, you still need a `Thread` object to do the work.

If you have approach two (extending `Thread` class): Instantiation would be simple

```
MyThread thread = new MyThread();
```

If you implement `Runnable`, instantiation is only slightly less simple.

To instantiate your `Runnable` class:

```
MyRunnableThread myRunnable = new MyRunnableThread ();
Thread thread = new Thread(myRunnable); // Pass your Runnable to the Thread
```

Giving the same target to multiple threads means that several threads of execution will be running the very same job (and that the same job will be done multiple times).

Thread Class Constructors

- `Thread()` :

Default constructor – To create thread with default name and priority

- `Thread(Runnable target)`

This constructor will create a thread from the runnable object.

- `Thread(Runnable target, String name)`

This constructor will create thread from runnable object with name as passed in the second argument

- `Thread(String name)`

This constructor will create a thread with the name as per argument passed.

So now we've made a Thread instance, and it knows which `run()` method to call. But nothing is happening yet. At this point, all we've got is a plain old Java object of type Thread. It is not yet a thread of execution. To get an actual thread—a new call stack—we still have to start the thread.

Starting a Thread

You've created a Thread object and it knows its target (either the passed-in Runnable or itself if you extended class Thread). Now it's time to get the whole thread thing happening—to launch a new call stack. It's so simple it hardly deserves its own subheading:

```
t.start();
```

Prior to calling `start()` on a Thread instance, the thread is said to be in the new state. There are various thread states which we will cover in next tutorial.

When we call `t.start()` method following things happens:

- A new thread of execution starts (with a new call stack).
- The thread moves from the new state to the runnable state.
- When the thread gets a chance to execute, its target `run()` method will run.

The following example demonstrates what we've covered so far—defining, instantiating, and starting a thread: In below Java program we are not implementing thread communication or synchronization, because of that output may depend on operating system's scheduling mechanism and JDK version.

We are creating two threads t1 and t2 of MyRunnable class object. Starting both threads, each thread is printing thread name in the loop.

Java Code (MyRunnable.java)

```
package mythreading;
public class MyRunnable implements Runnable{
    @Override
    public void run() {
        for(int x =1; x < 10; x++) {
            System.out.println("MyRunnable running for Thread Name:
" + Thread.currentThread().getName());
        }
    }
}
```

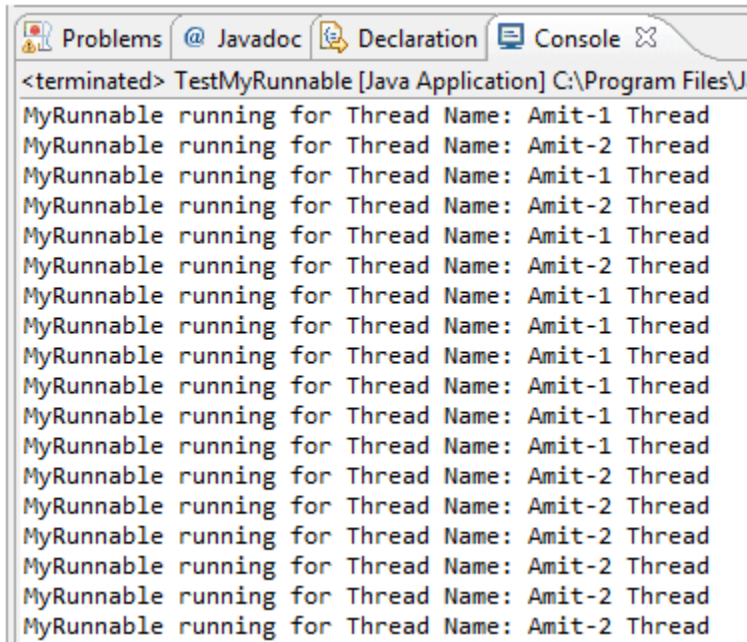
Copy

Java Code (TestMyRunnable.java)

```
package mythreading;
public class TestMyRunnable {
    public static void main (String [] args) {
        MyRunnable myrunnable = new MyRunnable();
        //Passing myrunnable object to Thread class constructor
        Thread t1 = new Thread(myrunnable);
        t1.setName("Amit-1 Thread");
        //Starting Thread t1
        t1.start();
        Thread t2 = new Thread(myrunnable);
        t2.setName("Amit-2 Thread");
        t2.start();
    }
}
```

Copy

Output:



```
<terminated> TestMyRunnable [Java Application] C:\Program Files\J
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
```

Summary:

- Threads can be created by extending Thread and overriding the public void run() method
- Thread objects can also be created by calling the Thread constructor that takes a Runnable argument. The Runnable object is said to be the target of the thread.
- You can call start() on a Thread object only once. If start() is called more than once on a Thread object, it will throw a Runtime Exception.
- Each thread has its own call stack which is storing state of thread execution
- When a Thread object is created, it does not become a thread of execution until its start() method is invoked. When a Thread object exists but hasn't been started, it is in the new state and is not considered alive.

Java Code Editor:

Main.java

```
public class Main {

    public static void main(String[] args) {

        // Write your code here

    }

}
```

Java Thread Interaction

Introduction

The Object class has three methods, wait(), notify(), and notifyAll() that help threads communicate about the status of an event that the threads care about. In last tutorial, we have seen about synchronization on resources where one thread can acquire a lock which forces other threads to wait for lock availability. Object class has these three methods for inter-thread communication.

For example, we have two threads named car_owner and car_mechanic, if thread car_mechanic is busy, so car_owner thread has to wait and keep checking for the car_mechanic thread to get free. Using the wait and notify mechanism, the car_owner for service thread could check for car_mechanic, and if it doesn't find any it can say, "Hey, I'm not going to waste my time checking every few minutes. I'm going to go hang out, and when a car_mechanic gets free, have him notify me so I can go back to runnable and do some work." In other words, using wait() and notify() lets one thread put itself into a "waiting room" until some another thread notifies it that there's a reason to come back out.

Method Signature	Description
final void wait() throws InterruptedException	Calling this method will make calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify() method.
final void notify()	This method is similar to notify(), but it wakes up all the threads which are on waiting for state for object lock but only one thread will get access to object lock.

One key point to remember about wait/notify is this:

wait(), notify(), and notifyAll() must be called from within a synchronized context. A thread can't invoke a wait or notify method on an object unless it owns that object's lock.

Below program explains the concept of car service queue where car_owner and car_mechanic thread interact with each other in the loop.

Java Code:

```
package threadcommunication;
public class CarOwner implements Runnable {
    CarQueueClass q;
    CarOwner(CarQueueClass queue) {
        this.q=queue;
        new Thread(this, "CarOwner").start();
    }
    @Override
```

```

    public void run() {
        int count =0;
        try {
            while(count< 5){
                Thread.sleep(2000);
                q.put(count++);
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

Copy

```

package threadcommunication;
public class CarMechanic implements Runnable {
    CarQueueClass q;
    CarMechanic(CarQueueClass queue){
        this.q=queue;
        new Thread(this, "CarMechanic").start();
    }
    @Override
    public void run() {
        for(int i=0;i< 5;i++)
            q.get();
    }
}

```

Copy

```

package threadcommunication;
public class CarQueueClass {
    int n;
    boolean mechanic_available = false;
    synchronized int get() {
        if(!mechanic_available)
            try {
                wait(5000);
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        System.out.println("Got Request for Car Service: " + n);
        mechanic_available = false;
        notify();
        return n;
    }
    synchronized void put(int n) {
        if(mechanic_available)
            try {
                wait(5000);
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        this.n = n;
        mechanic_available = true;
    }
}

```

```

        System.out.println("Put Request for Car Service: " + n);
        notify();
    }
}

```

Copy

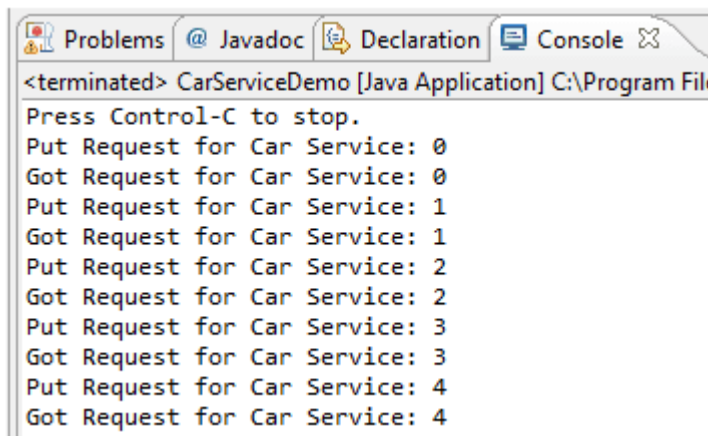
```

package threadcommunication;
public class CarServiceDemo {
    /**
     * @param args
     */
    public static void main(String[] args) {
        CarQueueClass q = new CarQueueClass();
        new CarOwner(q);
        new CarMechanic(q);
        System.out.println("Press Control-C to stop.");
    }
}

```

Copy

Output:



```

<terminated> CarServiceDemo [Java Application] C:\Program File
Press Control-C to stop.
Put Request for Car Service: 0
Got Request for Car Service: 0
Put Request for Car Service: 1
Got Request for Car Service: 1
Put Request for Car Service: 2
Got Request for Car Service: 2
Put Request for Car Service: 3
Got Request for Car Service: 3
Put Request for Car Service: 4
Got Request for Car Service: 4

```

Summary:

- The wait() method puts a thread in waiting for pool from running state.
- The notify() method is used to send a signal to one and only one of the threads that are waiting in that same object's waiting pool.
- The method notifyAll() works in the same way as for notify(), only it sends the signal to all of the threads waiting on the object.
- All three methods—wait(), notify(), and notifyAll()—must be called from within a synchronized context. A thread invokes wait() or notify() on a particular object, and the thread must currently hold the lock on that object.

Java Code Editor:

Main.java

```
public class Main {  
    public static void main(String[] args) {  
        // Write your code here  
    }  
}
```
