Sri Sai Vidya Vikas Shikshana Samithi ®
SAI VIDYA INSTITUTE OF TECHNOLOGY
Approved by AICTE, New Delhi, Affiliated to VTU, Recognized by Govt. of Karnataka
Accredited by NBA
RAJANUKUNTE, BENGALURU 560 064, KARNATAKA
Phone: 080-28468191/96/97/98 ,Email: info@saividya.ac.in, URLwww.saividya.ac.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (CSE)

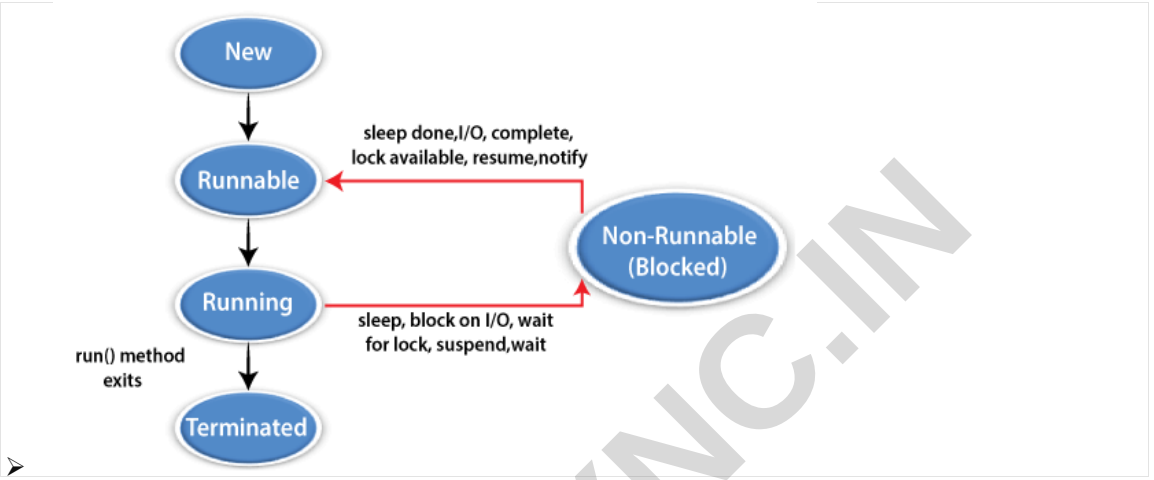
Module -5

Multithreaded Programming

- Multithreading in Java allows for concurrent execution of multiple parts of a program, known as threads.
- This contrasts with process-based multitasking, where each program is a separate unit, whereas threads share the same address space and belong to the same process.
- Multithreading is advantageous due to its lower overhead compared to process-based multitasking.
- It helps in maximizing processing power by minimizing idle time, especially in interactive and networked environments where tasks like data transmission and user input are slower compared to CPU processing speed.
- Multithreading enables more efficient use of available resources and smoother program execution by allowing other threads to run while one is waiting.

The Java Thread Model

- Java's runtime system heavily relies on threads to enable asynchronous behavior, which helps in utilizing CPU cycles more efficiently.
- Unlike single-threaded systems that use event loops with polling, Java's multithreading eliminates the need for such mechanisms. In single-threaded environments, blocking one thread can halt the entire program, leading to inefficiencies and potential domination of one part over others.
- With Java's multithreading, one thread can pause without affecting other parts of the program, allowing idle time to be utilized elsewhere.
- This is particularly beneficial for tasks like animation loops, where pauses between frames don't halt the entire system. Multithreading in Java works seamlessly on both single-core and multicore systems, with threads sharing CPU time on single-core systems and potentially executing simultaneously on multicore systems.
- Threads in Java can exist in various states, including running, ready to run, suspended, blocked, or terminated. Each state represents a different stage of thread execution, with the ability to suspend, resume, or terminate threads as needed.
- Overall, Java's multithreading capabilities contribute to more efficient and responsive software development.



Thread Priorities

- **Thread Priorities**: Java assigns each thread a priority to determine its relative importance. Higher-priority threads are given preference during context switches, but priority doesn't affect the speed of execution.
- **Thread States**: Threads can be in various states like running, ready to run, suspended, blocked, or terminated. These states govern the behavior of threads in the system.
- **Synchronization**: Java provides mechanisms like monitors to enforce synchronicity between threads, ensuring that shared resources are accessed safely. Synchronization is achieved through the use of synchronized methods and blocks.
- **Messaging**: Java facilitates communication between threads through predefined methods that all objects have. This messaging system allows threads to wait until they are explicitly notified by another thread.
- **Thread Class and Runnable Interface**: Java's multithreading system is built around the **Thread** class and the **Runnable** interface. Threads can be created either by extending the **Thread** class or implementing the **Runnable** interface.
- **Main Thread**: Every Java program starts with a main thread, which is automatically created. The main thread is crucial for spawning other threads and often performs shutdown actions at the end of the program.
- **Thread Methods**: Java's **Thread** class provides various methods for managing threads, including **getName()**, **getPriority()**, **isAlive()**, **join()**, **run()**, **sleep()**, and **start()**.

Creating a Thread

Implementing Runnable Interface: To create a thread, you implement the Runnable interface in a class. This interface abstracts a unit of executable code and requires implementing a single method called run().

Runnable's run() Method: Inside the run() method, you define the code that constitutes the new thread. This method can call other methods, use other classes, and declare variables just like the main thread can.

Instantiating Thread Object: After implementing Runnable, you instantiate an object of type Thread within that class. The Thread constructor requires an instance of a class that implements Runnable and a name for the thread.

Starting the Thread: The new thread doesn't start running until you call its start() method. This method initiates a call to run(), effectively starting the execution of the new thread.

Example: An example code snippet demonstrates creating and starting a new thread:

```
class NewThread implements Runnable {
   Thread t;

   NewThread() {
      // Create a new, second thread
      t = new Thread(this, "Demo Thread");
      System.out.println("Child thread: " + t);
   }

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

class ThreadDemo {
   public static void main(String[] args) {
      NewThread nt = new NewThread(); // create a new thread
      nt.t.start(); // Start the thread
      // Main thread continues its execution
      // ...
   }
}
```


Extending Thread

Extending Thread Class: To create a thread, you create a new class that extends the Thread class. The extending class must override the run() method, which serves as the entry point for the new thread.

Constructor Invocation: Inside the constructor of the extending class, you can invoke the constructor of the Thread class using super() to specify the name of the thread.

Starting the Thread: After creating an instance of the extending class, you call the start() method to begin execution of the new thread.

Example:

```
class NewThread extends Thread {
NewThread() {
// Invoke Thread constructor to set thread name
super("Demo Thread");
System.out.println("Child thread: " + this);
}
```

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

class ExtendThread {
public static void main(String[] args) {
NewThread nt = new NewThread(); // create a new thread
nt.start(); // start the thread
// Main thread continues its execution
// ...
}
}
```

Creating Multiple Threads

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

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

   // This is the entry point for thread.
   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 MultiThreadDemo {
   public static void main(String[] args) { NewThread
      nt1 = new NewThread("One"); NewThread nt2 =
      new NewThread("Two"); NewThread nt3 = new
      NewThread("Three");

      // Start the threads. nt1.t.start();
      nt2.t.start();
      nt3.t.start();

      try {
         // wait for other threads to end
         Thread.sleep(10000);
      } catch (InterruptedException e) { System.out.println("Main
         thread Interrupted");
      }

      System.out.println("Main thread exiting.");
   }
}
```

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

```
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.

Using isAlive( ) and join( )

Using the **isAlive()** and **join()** methods in Java threads:

- **isAlive() Method**:
  - Defined by the **Thread** class.
  - Returns **true** if the thread upon which it is called is still running.
  - Returns **false** otherwise.
  - Occasionally useful for checking the status of a thread.
- **join() Method**:
  - Also defined by the **Thread** class.
  - Waits until the thread on which it is called terminates.
  - The calling thread waits until the specified thread joins it.
  - Additional forms of **join()** allow specifying a maximum amount of time to wait for the specified thread to terminate.
- **Usage**:
  - **join()** is commonly used to ensure that one thread waits for another thread to finish its execution.
  - This is particularly useful when you want the main thread to finish last or when you need to synchronize the execution of multiple threads.
- **Example**:
  - An improved version of the example code can use **join()** to ensure that the main thread is the last to stop.
  - Additionally, **isAlive()** can be used to check if a thread is still running.

```
// Using join() to wait for threads to finish. class
NewThread implements Runnable {
   String name; // name of thread
   Thread t;

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

   // This is the entry point for thread. 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 DemoJoin {
        public static void main(String[] args) { NewThread
           nt1 = new NewThread("One"); NewThread nt2 =
           new NewThread("Two"); NewThread nt3 = new
           NewThread("Three");

           // Start the threads. nt1.t.start();
           nt2.t.start();
           nt3.t.start();

           System.out.println("Thread One is alive: "
                                + nt1.t.isAlive()); System.out.println("Thread
           Two is alive: "
                                + nt2.t.isAlive()); System.out.println("Thread
           Three is alive: "
                                + nt3.t.isAlive());
           // wait for threads to finish try {
              System.out.println("Waiting for threads to finish."); nt1.t.join();
              nt2.t.join();
              nt3.t.join();
           } catch (InterruptedException e) { System.out.println("Main
              thread Interrupted");
           }

           System.out.println("Thread One is alive: "
                                + nt1.t.isAlive()); System.out.println("Thread
           Two is alive: "
                                + nt2.t.isAlive()); System.out.println("Thread
           Three is alive: "
                                + nt3.t.isAlive());

           System.out.println("Main thread exiting.");
        }

     }
```


Thread Priorities

- **Thread Priorities**:
  - Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
  - In theory, higher-priority threads get more CPU time than lower-priority threads over a given period of time.
  - Higher-priority threads can preempt lower-priority ones, meaning they can interrupt the execution of lower-priority threads.

- **Equal Priority Threads**:
  - In theory, threads of equal priority should get equal access to the CPU.
  - However, Java is designed to work in various environments, and the actual behavior may differ depending on the operating system and multitasking implementation.
  - To ensure fairness, threads that share the same priority should yield control occasionally, especially in nonpreemptive environments.
- **Setting Thread Priority**:
  - Use the **setPriority()** method to set a thread's priority.
  - Syntax: **void setPriority(int level)**
  - The **level** parameter specifies the new priority setting for the thread, and it must be within the range of **MIN_PRIORITY** and **MAX_PRIORITY**, currently 1 and 10, respectively.
  - To return a thread to default priority, use **NORM_PRIORITY**, which is currently 5.
- **Getting Thread Priority**:
  - Use the **getPriority()** method to obtain the current priority setting of a thread.
  - Syntax: **int getPriority()**
- **Implementation Considerations**:
  - Implementations of Java may have different behaviors when it comes to scheduling and thread priorities.
  - To ensure predictable and cross-platform behavior, it's advisable to use threads that voluntarily give up CPU time.

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*.
- Key to synchronization is the concept of the monitor. A *monitor* is an object that is used as a mutually exclusive lock.
- Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.
- You can synchronize your code in either of two ways. Both involve the use of the
- **synchronized** keyword, and both are examined here.

Using Synchronized Methods

- **Implicit Monitors**:
  - All objects in Java have their own implicit monitor associated with them.
  - To enter an object's monitor, you can call a method that has been modified with the **synchronized** keyword.
  - While a thread is inside a **synchronized** method of an object, all other threads that try to call synchronized methods on the same instance have to wait.
- **Need for Synchronization**:
  - Synchronization is necessary to ensure thread safety, especially when multiple threads access shared resources concurrently.
  - Without synchronization, concurrent access to shared resources can lead to data corruption, race conditions, and other inconsistencies.

- **Example**:
  - The example program consists of three classes: **Callme**, **Caller**, and **Synch**.
  - The **Callme** class has a method **call()** which prints a message inside square brackets and then pauses the thread for one second using **Thread.sleep(1000)**.
  - The **Caller** class takes a reference to an instance of **Callme** and a message string. It creates a new thread that calls the **run()** method which in turn calls the **call()** method on the **Callme** instance.
  - The **Synch** class creates a single instance of **Callme** and three instances of **Caller**, each with a unique message string. All **Caller** instances share the same **Callme** instance.

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

class Caller implements Runnable { String
   msg;
   Callme
   target;
   Thread t;

   public Caller(Callme targ, String s) { target =
      targ;
      msg = s;
      t = new Thread(this);
   }

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

class Synch {
   public static void main(String[] args) { Callme
      target = new Callme();
      Caller ob1 = new Caller(target, "Hello");
      Caller ob2 = new Caller(target, "Synchronized"); Caller
      ob3 = new Caller(target, "World");

      // Start the threads. ob1.t.start();
      ob2.t.start();
      ob3.t.start();

      // wait for threads to end try {
```

```
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}
```

Here is the output produced by this program:

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

Interthread Communication

Interprocess communication using **wait()**, **notify()**, and **notifyAll()** methods in Java:

- **Purpose**:
  - These methods provide a means for threads to communicate and coordinate their activities without using polling, which can waste CPU cycles.
- **Method Definitions**:
  - **wait()**: Tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify()** or **notifyAll()**.
  - **notify()**: Wakes up a single thread that previously called **wait()** on the same object.
  - **notifyAll()**: Wakes up all threads that previously called **wait()** on the same object. One of the threads will be granted access.
  - All three methods are declared within the **Object** class and can only be called from within a synchronized context.

  Additional Forms of wait():
  - Additional forms of the **wait()** method exist that allow you to specify a period of time to wait.
- **Spurious Wakeups**:
  - In rare cases, a waiting thread could be awakened due to a spurious wakeup, where **wait()** resumes without **notify()** or **notifyAll()** being called. To handle this, calls to **wait()** are often placed within a loop that checks the condition on which the thread is waiting.
- **Best Practices**:
  - The Java API documentation recommends using a loop to check conditions when waiting, especially due to the possibility of spurious wakeups.

```
// An incorrect implementation of a producer and consumer.

 class Q {
    int n;

    synchronized int get() {
        System.out.println("Got: " + n); return
        n;
    }

    synchronized void put(int n) { this.n
        = n; System.out.println("Put: " +
        n);
    }
}

class Producer implements Runnable {
    Q q;
    Thread t;

    Producer(Q q)
        { this.q = q;
        t = new Thread(this, "Producer");
    }

    public void run() {
        int i = 0;

        while(true) {
            q.put(i++);



                }
            }
          }

          class Consumer implements Runnable {
              Q q;
              Thread t;

              Consumer(Q q) {
                  this.q = q;
                  t = new Thread(this, "Consumer");
              }

              public void run() {
```

```
        while(true) {
            q.get();
        }
    }
}

class PC {
    public static void main(String[] args) { Q q =
        new Q();
        Producer p = new Producer(q);
        Consumer c = new
        Consumer(q);

        // Start the threads. p.t.start();
        c.t.start();

        System.out.println("Press Control-C to stop.");
    }
}
```

Although the **put( )** and **get( )** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

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

```
// A correct implementation of a producer and consumer. class Q
{
    int n;
    boolean valueSet = false;

    synchronized int get() { while(!valueSet)
            try {
                wait();

            } catch(InterruptedException e) { System.out.println("InterruptedException
                caught");
            }

            System.out.println("Got: " + n);
            valueSet = false;
            notif
            y();
            retur
            n n;
    }

    synchronized void put(int n) {
        while(valueSet)
            try {
                wait();
            } catch(InterruptedException e) { System.out.println("InterruptedException
                caught");
            }

            this.n = n;
            valueSet = true;
            System.out.println("Put: " + n);
            notify();
    }
}

class Producer implements Runnable {
    Q q;
    Thread t;

    Producer(Q q)
        { this.q = q;
        t = new Thread(this, "Producer");
    }

    public void run() {
        int i = 0;
```

```
        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;
    Thread t;

    Consumer(Q q) {
        this.q = q;
        t = new Thread(this, "Consumer");
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PCFixed {
    public static void main(String[] args) { Q q =
        new Q();
        Producer p = new Producer(q);
        Consumer c = new
        Consumer(q);

        // Start the threads. p.t.start();
        c.t.start();

        System.out.println("Press Control-C to stop.");
    }
}
```

Inside **get( )**, **wait( )** is called. This causes its execution to suspend until **Producer** notifies you that some data is ready. When this happens, execution inside **get( )** resumes. After the data has been obtained, **get( )** calls **notify( )**. This tells **Producer** that it is okay to put more data in the queue. Inside **put( )**, **wait( )** suspends execution until **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and **notify( )** is called. This tells **Consumer** that it should now remove it.

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

```
Put:  1
Got:  1
Put:  2
Got:  2
```

```
Put:  3
Got:  3
Put:  4
Got:  4
Put:  5
Got:  5
```

Suspending, Resuming, and Stopping Threads

- **Deprecated Methods**:
  - In early versions of Java (prior to Java 2), thread suspension, resumption, and termination were managed using **suspend()**, **resume()**, and **stop()** methods defined by the **Thread** class.
  - However, these methods were deprecated due to potential issues and risks they posed, such as causing system failures and leaving critical data structures in corrupted states.
- **Reasons for Deprecation**:
  - **suspend()**: Can cause serious system failures, as it doesn't release locks on critical data structures, potentially leading to deadlock.
  - **resume()**: Deprecated as it requires **suspend()** to work properly.
  - **stop()**: Can cause system failures by leaving critical data structures in corrupted states.
- **Alternative Approach**:
  - Instead of using deprecated methods, threads should be designed to periodically check a flag variable to determine whether to suspend, resume, or stop their own execution.
  - Typically, a boolean flag variable is used to indicate the execution state of the thread.
  - If the flag is set to "running," the thread continues to execute. If it's set to "suspend," the thread pauses. If it's set to "stop," the thread terminates.
- **Example Using wait() and notify()**:
  - The **wait()** and **notify()** methods inherited from **Object** can be used to control the execution of a thread.
  - An example provided demonstrates how to use these methods to control thread execution.
  - It involves a boolean flag (**suspendFlag**) to control the execution of the thread.
  - The **run()** method periodically checks **suspendFlag**, and if it's **true**, the thread waits. Methods **mysuspend()** and **myresume()** are used to set and unset the flag and notify the thread to wake up.

```
// Suspending and resuming a thread the modern way. class
NewThread implements Runnable {
   String name; // name of thread
   Thread t;

            boolean suspendFlag;

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

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

    synchronized void mysuspend() {
       suspendFlag = true;
    }

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

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

       ob1.t.start(); // Start the thread ob2.t.start(); // Start
       the thread

       try {
          Thread.sleep(1000);
          ob1.mysuspend();
          System.out.println("Suspending thread One"); Thread.sleep(1000);
          ob1.myresume();
          System.out.println("Resuming thread One");
          ob2.mysuspend();
          System.out.println("Suspending thread Two"); Thread.sleep(1000);
```

```
            ob2.myresume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        // wait for threads to finish try {
            System.out.println("Waiting for threads to finish."); ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Main thread exiting.");
    }
}
```

When you run the program, you will see the threads suspend and resume. Later in this book, you will see more examples that use the modern mechanism of thread control. Although this mechanism may not appear as simple to use as the old way, nevertheless, it is the way required to ensure that run-time errors don't occur. It is the approach that *must* be used for all new code.

Obtaining a Thread's State

We can obtain the current state of a thread by calling the **getState( )** method defined by **Thread**. It is shown here:

```
Thread.State getState( )
```

It returns a value of type **Thread.State** that indicates the state of the thread at the time at which the call was made. **State** is an enumeration defined by **Thread**. (An enumeration is a list of named constants. It is discussed in detail in Chapter 12.) Here are the values that can be returned by **getState( )**:

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called **sleep( )**. This state is also entered when a timeout version of **wait( )** or **join( )** is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non- timeout version of **wait( )** or **join( )**.

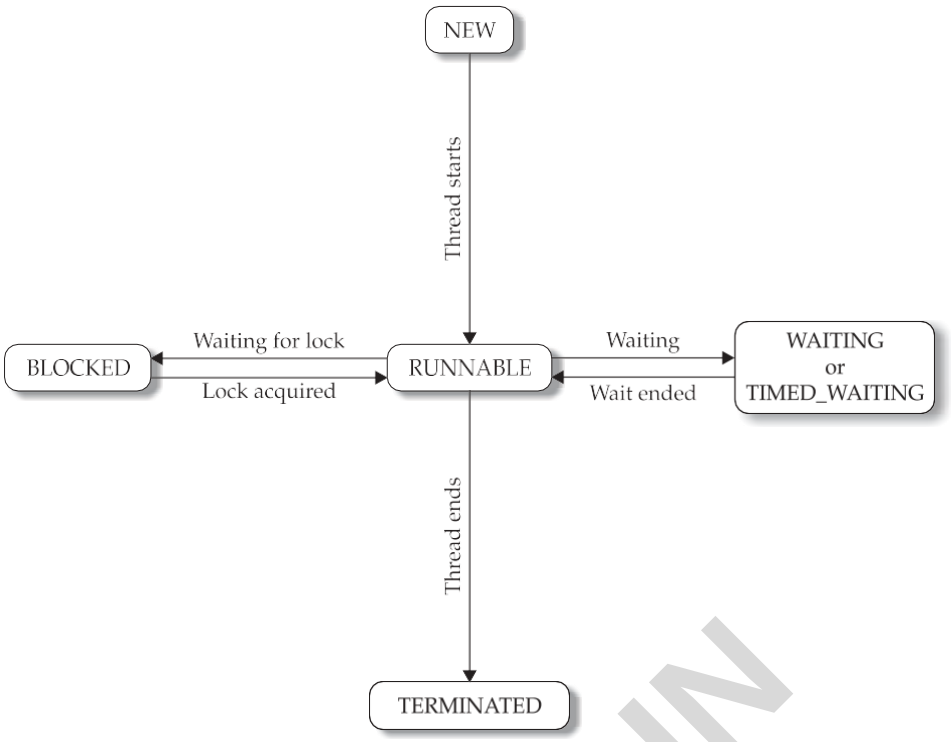


Figure 11-1 Thread states

Figure 11-1 diagrams how the various thread states relate.

Given a **Thread** instance, you can use **getState( )** to obtain the state of a thread. For example, the following sequence determines if a thread called **thrd** is in the **RUNNABLE** state at the time **getState( )** is called:

```
Thread.State ts = thrd.getState(); if(ts ==

Thread.State.RUNNABLE) // ...
```

It is important to understand that a thread's state may change after the call to **getState( )**. Thus, depending on the circumstances, the state obtained by calling **getState( )** may not reflect the actual state of the thread only a moment later. For this (and other) reasons, **getState( )** is not intended to provide a means of synchronizing threads. It's primarily used for debugging or for profiling a thread's run-time characteristics.