MULTITHREADED PROGRAMMING

Multithreaded Programming

- A multithreaded program contains two or more parts that can run concurrently.
- Each part of a multithreaded program is called a thread.
- Each thread defines a separate path of execution.
- Java provides built in support for multithreaded programming.
- Multithreading is a specialized form of multitasking.

- The two types of multitasking are

  1. Process-based
  2. Thread based

Process-based multitasking is the feature that allows our computer to run two or more programs concurrently.

For ex: It allows us to run the Java compiler at the same time that we are using a text editor.

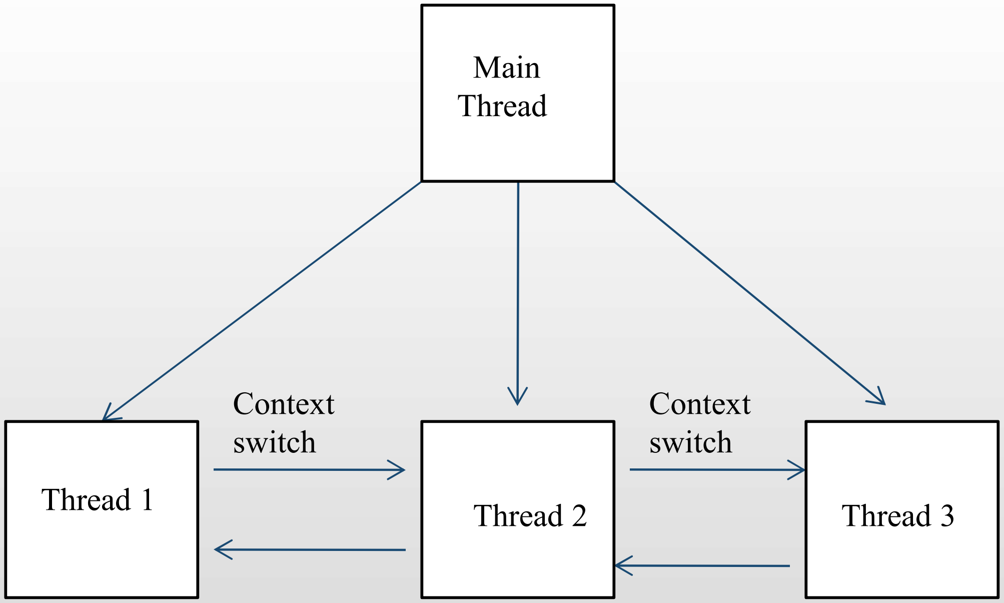
Thread based multitasking:

- Thread is the smallest unit of dispatchable code.
- A single program can perform two or more tasks simultaneously.
- **Ex:** a text editor can format text at the same time that is printing.
- Thus process-based multitasking deals with the "big picture," and thread-based multitasking handles the details.

Difference between process-based and thread-based multitasking processes:

- Multitasking threads require less overhead than multitasking processes.
- Processes are heavyweight tasks that require their own separate address spaces. But the threads are lightweight processes and they share the same address space.
- Context switching from one process to other process is costly. But, context switching from one thread to the next is low cost.
- Interprocess communication is expensive and limited in process-based multitasking. But, interthread communication is inexpensive.
- Process-based multitasking is not under the control of Java. But, the multithreaded multitasking is under the control of Java.

Multithreaded program



- Multithreading enables us to write very efficient programs that make the maximum use of the CPU, because the idle time can be kept to a minimum.

- In a single threaded environment, our program has to wait for each of these tasks to finish before it can proceed to the next one – even though the CPU is sitting idle most of the time.

- Multithreading allows us to gain access to the idle time and put it to good use.

The Thread class and the Runnable Interface

Multithreading in Java is facilitated using,

1. Thread class and its methods
2. The interface Runnable

- To create a new thread our program will have to extend either the Thread class or implement the Runnable interface.

The Main Thread

When a Java program is started the Main thread runs immediately. ie, it starts execution.

Importance of Main Thread:

1. It is the thread from which other “child” threads will be spawned.

2. Often, it must be the last thread to finish execution because it performs various shutdown actions.

The main thread can be controlled through a Thread object.

It is done by obtaining a reference to it by calling the method currentThread()

The currentThread() is a public static member of thread .

General form :

```
static  Thread  currentThread()
```

It returns a reference to the thread in which it is called.

By using a reference to the main thread, we can control it like any other thread.

General form:

1. static void sleep(long milliseconds) throws InterruptedException

2. static void sleep(long milliseconds, int nanoseconds) throws InterruptedException

The second form allows us to specify the period in terms of milliseconds and nanoseconds.

We can set the name of a thread by using setName().

General form:

```
final    void    setName(String    threadName)
```

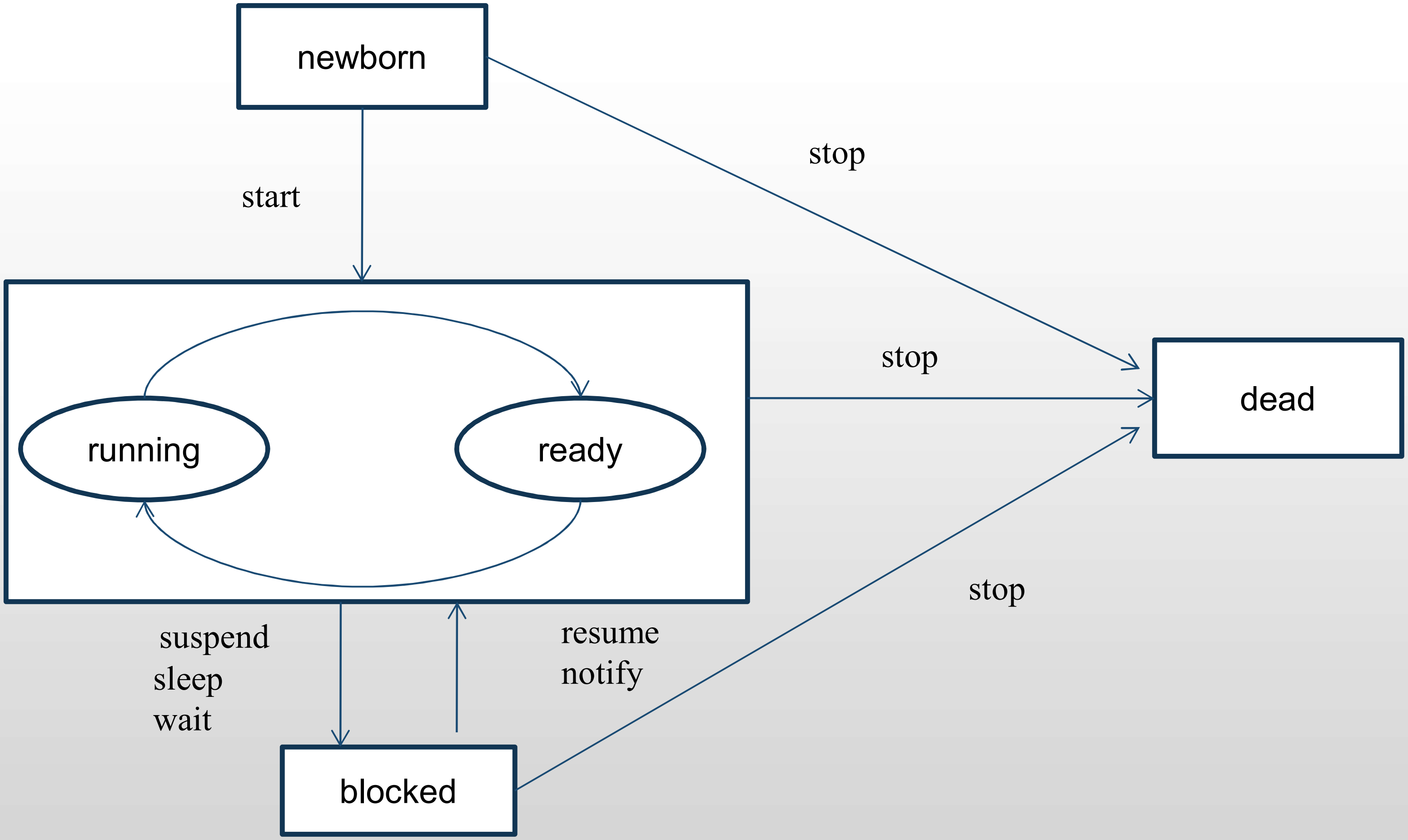
Here threadName specifies the name of the thread.

We can obtain the name of a thread by calling getName().

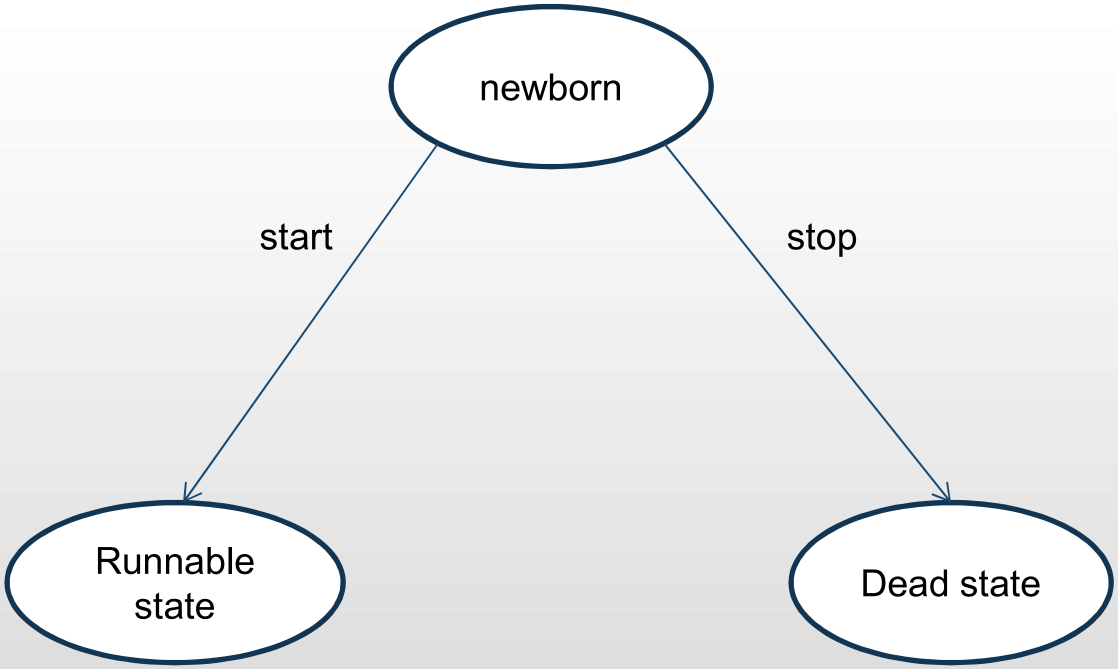
General form:

```
final    String    getName()
```

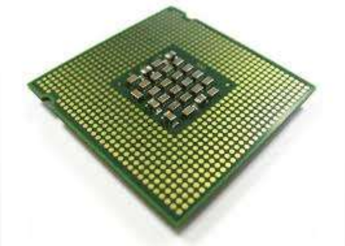
Life cycle of a thread

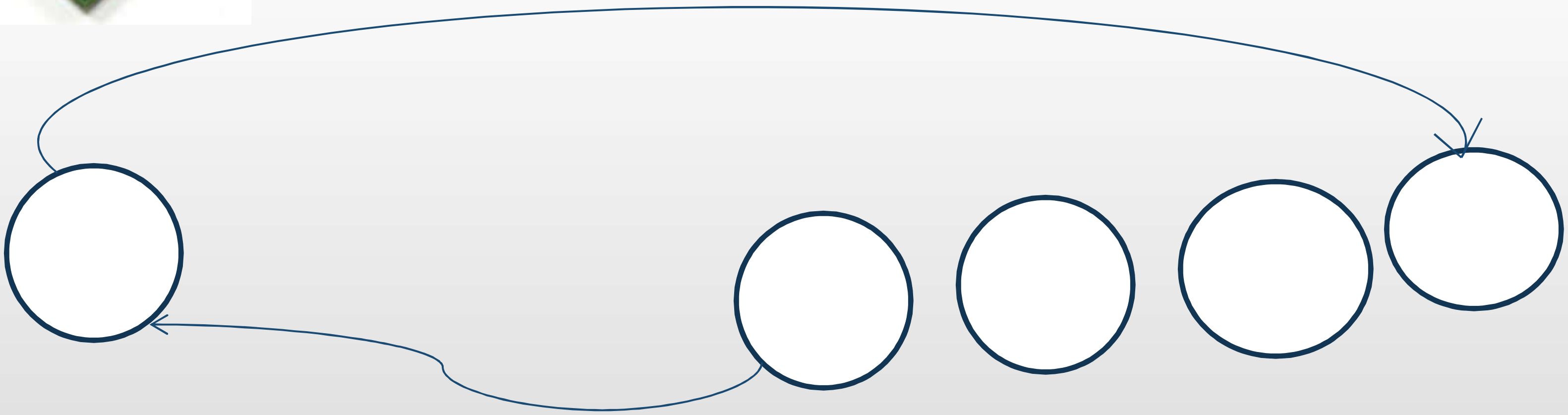


Newborn state



Runnable state

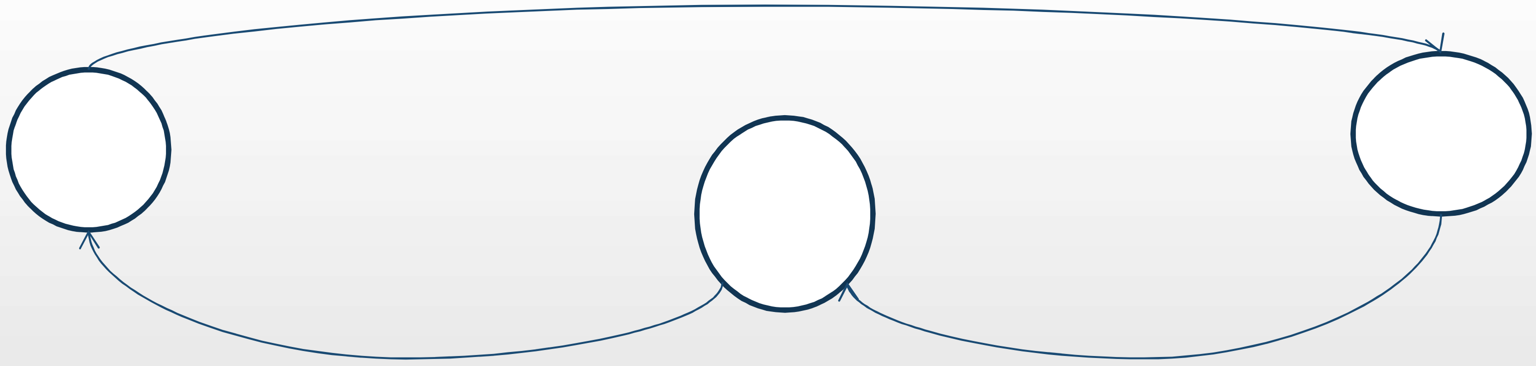




Running thread

Ready threads

Running state

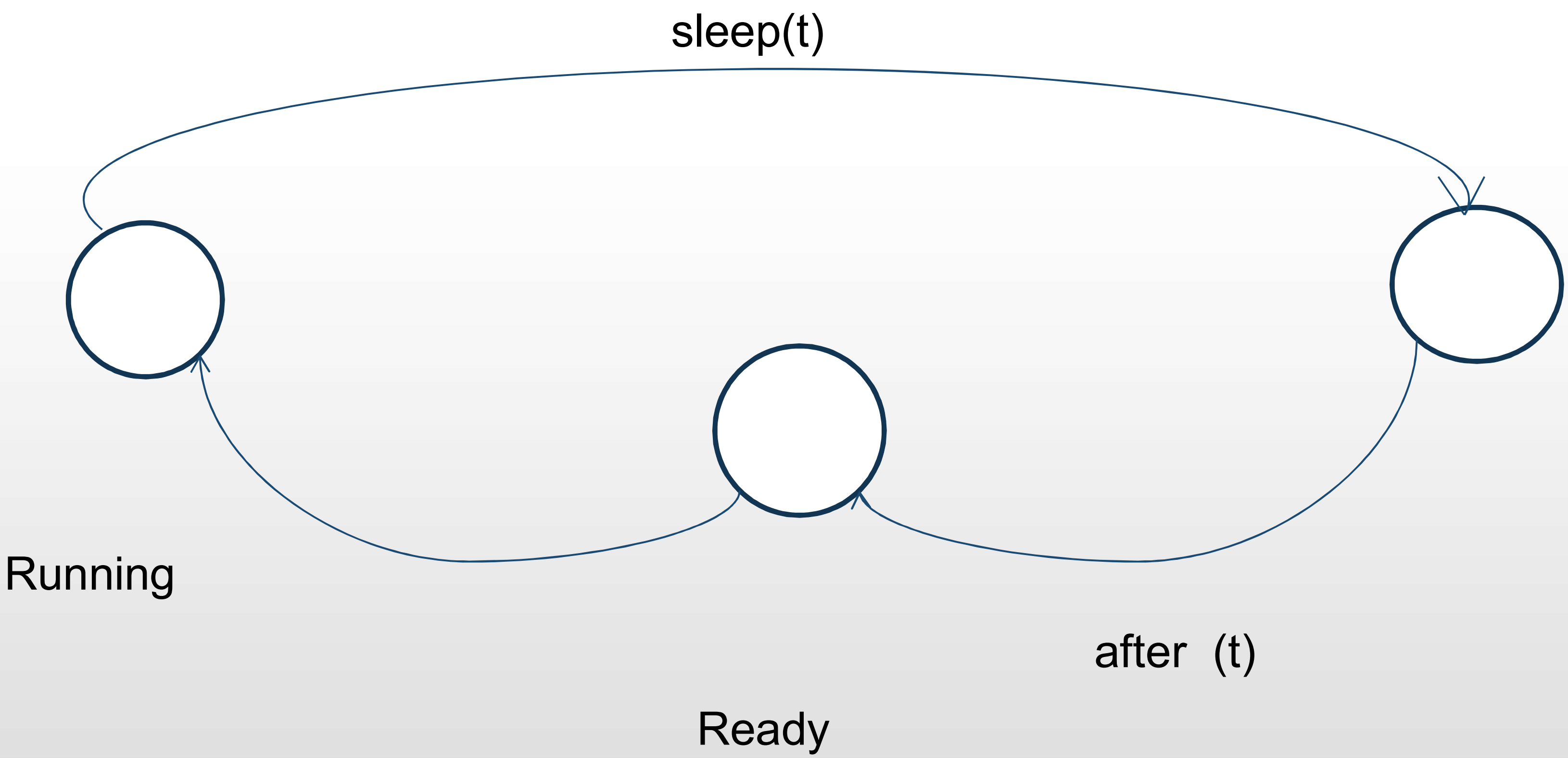


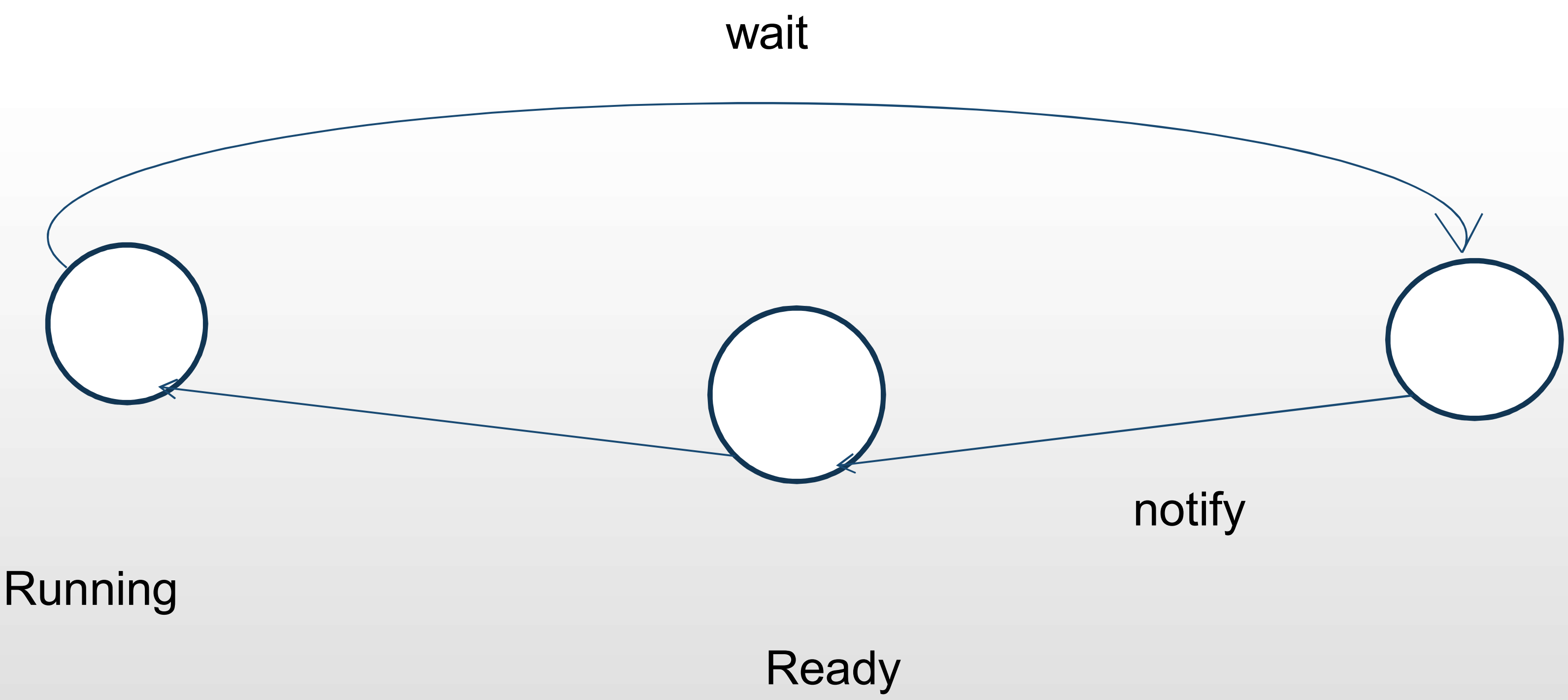
suspend

Running

resume

Ready


sleep(t)
Running
after (t)
Ready


wait
notify
Running
Ready

Creating a Thread

A thread can be created by instantiating an object of type Thread.

The 2 ways defined by Java for the creation of thread are:

1. By Implementing the Runnable interface.
2. By Extending the Thread class.

Implementing Runnable:

We can construct a thread on any object that implements **Runnable.**

To implement Runnable, a class need to implement a single method called **run().**

General form:

```
public  void  run()
```

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

- The run() can call other methods, use other classes and declare variables like the main thread.
- run() establishes an entry point for another concurrent thread of execution within our program.

After creating a class that implements Runnable ,we can instantiate an object of type Thread from within that class.

Constructors defined by Thread:

```
Thread()
Thread(Runnable   threadOb)
Thread(String   threadName)
Thread(Runnable   threadOb,  String   threadName)
```

Here,

- threadOb is an instance of a class that implements Runnable interface. It defines where the execution of the thread will begin.
- threadName is the name of the new thread.

After the creation of a new thread it will not start running until the start() method declared inside the Thread is called.

start() executes a call to run().

General form:

```
void  start()
```

Extending Thread

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

```
// Create a second thread by extending Thread
class   NewThread   extends Thread
 {     NewThread()
       {        super("Demo Thread");
                System.out.println("Child thread: " + this);
                start(); // Start the 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[])
        {       new   NewThread(); // create a new thread
                try
                {       for(int i = 5; i > 0; i--)
                        {    System.out.println("Main Thread: " + i);
                             Thread.sleep(1000);
                        }
                  }
                  catch (InterruptedException   e)
                  {    System.out.println("Main thread interrupted.");
                  }
                  System.out.println("Main thread exiting.");
        }
}
```

Here the call to **super()** inside **NewThread** invokes the following form of the **Thread constructor**

```
public Thread(String  threadName)
```

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);
              t.start(); // Start the 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[])
    {
                   new   NewThread("One"); // start threads
                   new   NewThread("Two");
                   new   NewThread("Three");
                   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.");
    }
}
```

Constructors defined by Thread:

```
Thread()
Thread(String   threadName)
Thread(Runnable   threadOb)
Thread(Runnable   threadOb,  String   threadName)
```

Using isAlive() and join()

Two ways to determine whether a thread has finished or not are:

1. By calling **isAlive()** method defined by **Thread** on the thread.

   General form:

   ```
   final   boolean   isAlive()
   ```

   It **returns true** if the thread upon which it is called is **running,** else **returns false**.

2. By calling the method **join()**

   General form:

   ```
   final   void   join()   throws   InterruptedException
   ```

   This method waits until the thread on which it is called terminates.

Thread priorities

- Used by the thread scheduler to decide when each thread should be allowed to run.
- Higher priority threads get more CPU time than lower priority threads(Theoritically).
- To set a thread's priority the method **setPriority()** which is a member of **Thread** is used.

  General form:

```
final   void   setPriority(int    level)
```

value of level should be within the MIN_PRIORITY and MAX_PRIORITY. ie, between 1 to 10.

To return a thread to default priority, specify NORM_PRIORITY, which is currently 5.

Thread priorities

- To get the value of current priority setting we can call the **getPriority()** method defined by the **Thread**.

  General form:

```
final int getPriority()
```

Synchronization

- When two or more threads need concurrent access to a shared data resource, they need to take care to only access the data one at a time.

- For example, one thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we may get strange results.

- Java enables us to overcome this problem using a technique known as synchronization.

Synchronization

- Keyword synchronized helps to solve such problem by keeping a watch on such locations.
- For example the method that will read information from a file and the method that will update the same file may be declared as synchronized.

```
 synchronized void update()
{


}
```

- When we declare a method synchronized, java creates a “monitor” and hands it over to the thread that calls the method first time.
- As long as the thread holds the monitor, no other thread can enter the synchronized section of the code.
- A monitor is like a key and the thread that holds the key can only open the lock.

It is also possible to mark a block of code as synchronized as follows.

```
 synchronized(lock-object)
{
          // code here
 }
```

Whenever a thread has completed its work of using synchronized method(or block of code), it will handover the monitor to the next thread that is ready to use the same resources.

Marking block of code as synchronized

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

Interthread Communication

Methods must be called from inside of synchronized method.

All three are final methods.

1. 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( ).

```
final  void  wait( )  throws  InterruptedException
```

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

```
final  void  notify( )
```

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

```
final  void  notifyAll( )
```

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

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

```
class   Producer   implements   Runnable
{
    Q    q;
    Producer(Q    q)
     {
                this.q = q;
                new   Thread(this, "Producer").start();
    }

    public   void   run()
    {
                int   i = 0;
                while(true)
                 {
                          q.put(i++);
                }
    }
}
```

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

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

Output

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

Correct implementation using wait() and notify()

```
// A correct implementation .
class  Q
 {    int  n;
      boolean   valueSet = false;
      synchronized  int  get()
      {        if(!valueSet)
                  try
                  {    wait();    }
                 catch(InterruptedException   e)
                 {     System.out.println("Exception ");    }

              System.out.println("Got: " + n);
               valueSet = false;
               notify();
               return  n;
     }
```

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

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

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

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

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

Output showing synchronous behaviour

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

Deadlock

- **Deadlock** occurs when two threads have a circular dependency on a pair of synchronized objects.

- **For ex:**
- Suppose one thread enters the monitor on object X and another thread enters the monitor on object Y.

- If the thread in X tries to call any synchronized method on Y, it will block as expected.

- If the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

Example

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

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

                 System.out.println(name + " trying to call A.last()");
                 a.last();
    }
    synchronized   void   last()
    {            System.out.println("Inside B.last");
    }
}
```

```
class Deadlock implements Runnable
{       A   a = new   A();
        B   b = new   B();
        Deadlock()
        {       Thread.currentThread().setName("MainThread");
                Thread   t = new   Thread(this, "RacingThread");
                t.start();
                a.foo(b); // get lock on a in this thread.
                System.out.println("Back in main thread");
         }
         public   void   run()
         {      b.bar(a); // get lock on b in other thread.
                System.out.println("Back in other thread");
         }
         public  static  void  main(String  args[])
        {       new  Deadlock();
         }
}
```

Output

```
MainThread entered A.foo

RacingThread entered B.bar

MainThread trying to call B.last()

RacingThread trying to call A.last()
```

Suspending, Resuming and Stopping Threads

The following methods defined by Thread are used to suspend and resume threads in the Java 1.1 and other versions before Java 2.

```
final  void  suspend( )
final  void  resume( )
final   void  stop()
```

Once a thread has been stopped it cannot be restarted using **resume()**.

Suspend(), resume() and stop() methods in Java 2

- The suspend(), resume() and stop() methods of the Thread class are deprecated in Java 2 since they may result in serious system failures.

- To facilitate the above said operation in Java 2 a thread must be designed so that the run( ) method periodically checks to determine whether that thread should suspend, resume, or stop its own execution. This is accomplished by establishing flag variable that indicates the execution state of the thread. As long as this flag is to "running," the run( ) method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. If it is set to "stop," the thread must terminate.