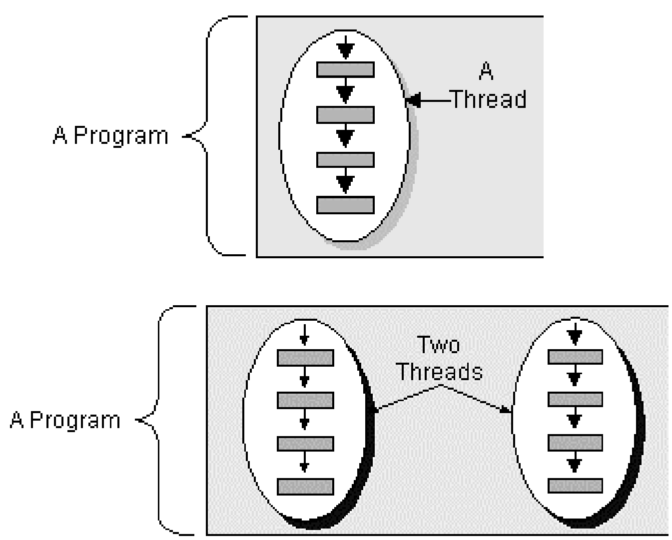
Threads - Simultaneous Activities

Human activities are intrinsically parallel. When you drive a car you steer, work the accelerator, listen to the radio, talk on your cell phone, etc. Computer programs often need this same form of simultaneous behavior. Threads are the Java mechanism for realizing simultaneous activities.

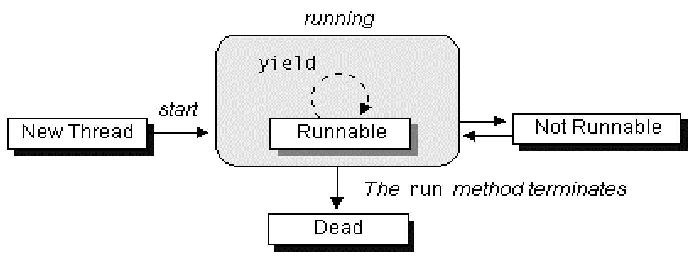
1. A Thread is a stream of activity.
2. A Java program can have many separate but interacting streams of activity.
3. There are mechanisms for creating threads of activity and mechanisms for assigning units of Java programs to these threads of activity.

The material which follows is taken from the Sun documentation on Threads.

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



Java and Threads



Java and Threads

public class Thread extends <u>Object</u> implements <u>Runnable</u>

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

Java and Threads

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:

```
class PrimeThread extends Thread {
   long minPrime;
   PrimeThread(long minPrime) {
      this.minPrime = minPrime;
   }
```


Java and Threads

```
   public void run() {
            // compute primes larger than minPrime
             . . .
      }
   }
```

The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

The other way to create a thread is to declare a class that implements the Runnable interface. That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started. The same example in this other style looks like the following:

```
class PrimeRun implements Runnable {
   long minPrime;
   PrimeRun(long minPrime) {
      this.minPrime = minPrime;
   }
```

Java and Threads

```
public void run() {
        // compute primes larger than minPrime
        . . .
    }
}
```

The following code would then create a thread and start it running:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

Java and Threads

Examples of Thread Programming

1. Unsynchronized threads
   Two threads running “at random.”
2. Cooperating Threads
   Two threads which attempt to cooperate by yielding to one another.
3. Sleeping Threads
   Note the “awakening” time.
4. An alternative mode of programming threads.
5. Blinking Balls - a simple animation.
6. Animation plus html supplied parameters.

Java and Threads

```
//  Readers/Writers with concurrent read or exclusive write
//
// Usage:
//        javac rw.real.java
//        java Main rounds

class RWbasic {        // basic read or write
  protected int data = 0;  // the "database"
  protected void read() {
    System.out.println("read:  " + data);
  }
  protected void write() {
    data++;
    System.out.println("wrote:  " + data);
  }
}
```


Java and Threads

```
class ReadersWriters extends RWbasic {  // Readers/Writers
  int nr = 0;
  private synchronized void startRead() {
    nr++;
  }
  private synchronized void endRead() {
    nr--;
    if (nr==0) notify();  // awaken waiting Writers
  }
  public void read() {
    startRead();
    System.out.println("read:  " + data);
    endRead();
  }
```

Java and Threads

```
public synchronized void write() {
    while (nr>0)
      try { wait(); }
        catch (InterruptedException ex) {return;}
    data++;
    System.out.println("wrote:  " + data);
    notify();    // awaken another waiting Writer
  }
}

class Reader extends Thread {
  int rounds;
  ReadersWriters RW;
  public Reader(int rounds, ReadersWriters RW) {
    this.rounds = rounds;
    this.RW = RW;
  }
```


Java and Threads

```
public void run() {
    for (int i = 0; i<rounds; i++) {
      RW.read();
    }
  }
}

class Writer extends Thread {
  int rounds;
  ReadersWriters RW;
  public Writer(int rounds, ReadersWriters RW) {
    this.rounds = rounds;
    this.RW = RW;
  }
  public void run() {
    for (int i = 0; i<rounds; i++) {
      RW.write();
    }
  }
}
```

Java and Threads

```
class Main {  // driver program -- two readers and one writer
  static ReadersWriters RW = new ReadersWriters();
  public static void main(String[] arg) {
    int rounds = Integer.parseInt(arg[0],10);
    new Reader(rounds, RW).start();
    new Reader(rounds, RW).start();
    new Writer(rounds, RW).start();
  }
}
```