G.PULLAIAH COLLEGE OF ENGINEERING AND TECHNOLOGY

Kurnool – 518002

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Lecture notes of

OBJECT ORIENTED PROGRAMMING USING THROUGH JAVA

Prepared by: P. Rama Rao, Asst. Prof.

Dept of CSE

Unit – 4

Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

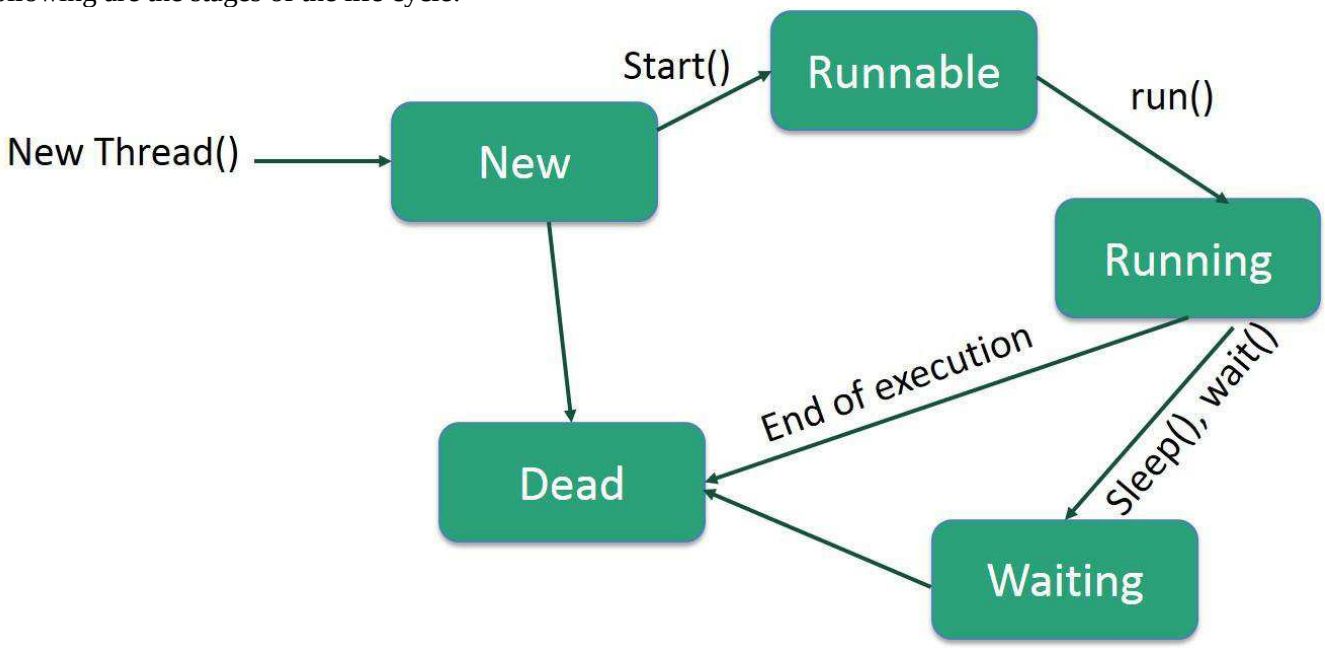
By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.

Following are the stages of the life cycle:



 New: A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.

 Runnable: After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

 Waiting: Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

 Timed Waiting: A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

 Terminated (Dead): A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Create a Thread by Implementing a Runnable Interface

If your class is intended to be executed as a thread then you can achieve this by implementing a **Runnable** interface. You will need to follow three basic steps:

Step 1

As a first step, you need to implement a run() method provided by a **Runnable** interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the run() method:

```
public void run( )
```

Step 2

As a second step, you will instantiate a **Thread** object using the following constructor:

```
Thread(Runnable threadObj, String threadName);
```

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

Step 3

Once a Thread object is created, you can start it by calling **start( )** method, which executes a call to run( ) method. Following is a simple syntax of start() method:

```
void start( );
```

Example

Here is an example that creates a new thread and starts running it:

```
class RunnableDemo implements Runnable {
private Thread t;
private String threadName;
RunnableDemo( String name){
threadName = name;
System.out.println("Creating " + threadName );
}
public void run() {
System.out.println("Running " + threadName );
try {
for(int i = 4; i > 0; i--) {
System.out.println("Thread: " + threadName + ", " + i);
// Let the thread sleep for a while.
Thread.sleep(50);
}
} 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 ();
}
}
}

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();
}
}
```

This will produce the following result:

```
Creating Thread-1
Starting Thread-1
```

```
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Create a Thread by Extending a Thread Class

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override **run( )** method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of run() method:

```
public void run( )
```

Step 2

Once Thread object is created, you can start it by calling **start( )** method, which executes a call to run( ) method. Following is a simple syntax of start() method:

```
void start( );
```

Example

Here is the preceding program rewritten to extend the Thread:

```
class ThreadDemo extends Thread {
private Thread t;
private String threadName;
ThreadDemo( String name){
threadName = name;
System.out.println("Creating " + threadName );
}
public void run() {
System.out.println("Running " + threadName );
try {
for(int i = 4; i > 0; i--) {
System.out.println("Thread: " + threadName + ", " + i);
// Let the thread sleep for a while.
Thread.sleep(50);
}
} 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 ();
}
}
}
```

```
public class TestThread {
public static void main(String args[]) {
ThreadDemo T1 = new ThreadDemo( "Thread-1");
T1.start();
ThreadDemo T2 = new ThreadDemo( "Thread-2");
T2.start();
}
}
```

This will produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2

Thread: Thread-2, 2

Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Thread Methods

Following is the list of important methods available in the Thread class.

Sr. No.	Methods with Description
1	**public void start()** Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	**public void run()** If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	**public final void setName(String name)** Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	**public final void setPriority(int priority)** Sets the priority of this Thread object. The possible values are between 1 and 10.
5	**public final void setDaemon(boolean on)** A parameter of true denotes this Thread as a daemon thread.
6	**public final void join(long millisec)** The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	**public void interrupt()** **Interrupts this thread, causing it to continue execution if it was blocked for any reason.**
8	**public final boolean isAlive()** **Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.**

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread

. **Sr. No.**	**Methods with Description**
1	**public static void yield()** Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	**public static void sleep(long millisec)** Causes the currently running thread to block for at least the specified number of milliseconds.
3	**public static boolean holdsLock(Object x)** Returns true if the current thread holds the lock on the given Object.
4	**public static Thread currentThread()** Returns a reference to the currently running thread, which is the thread that invokes this method.
5	**public static void dumpStack()** Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application

Major Java Multithreading Concepts

While doing Multithreading programming in Java, you would need to have the following concepts very handy:

- What is thread synchronization?
- Handling interthread communication
- Handling thread deadlock
- Major thread operations

Thread Synchronization

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement:

```
synchronized(objectidentifier) {
// Access shared variables and other shared resources
}
```

Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples, where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

Interthread Communication

If you are aware of interprocess communication then it will be easy for you to understand interthread communication. Interthread communication is important when you develop an application where two or more threads exchange some information.

There are three simple methods and a little trick which makes thread communication possible. All the three methods are listed below:

Sr. No.	Methods with Description
1	**public void wait()** Causes the current thread to wait until another thread invokes the notify().
2	**public void notify()** Wakes up a single thread that is waiting on this object's monitor.
3	**public void notifyAll()** Wakes up all the threads that called wait( ) on the same object.

Thread Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the **synchronized** keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object. Here is an example.

Example

```
public class TestThread {
public static Object Lock1 = new Object();
public static Object Lock2 = new Object();
public static void main(String args[]) {
ThreadDemo1 T1 = new ThreadDemo1();
ThreadDemo2 T2 = new ThreadDemo2();
T1.start();
T2.start();
}
private static class ThreadDemo1 extends Thread {
public void run() {
synchronized (Lock1) {
System.out.println("Thread 1: Holding lock 1...");
try { Thread.sleep(10); }
catch (InterruptedException e) {}
System.out.println("Thread 1: Waiting for lock 2...");

synchronized (Lock2) {
System.out.println("Thread 1: Holding lock 1 & 2...");
}
}
}
}
private static class ThreadDemo2 extends Thread {
public void run() {
synchronized (Lock2) {
System.out.println("Thread 2: Holding lock 2...");
try { Thread.sleep(10); }
catch (InterruptedException e) {}
System.out.println("Thread 2: Waiting for lock 1...");
synchronized (Lock1) {
System.out.println("Thread 2: Holding lock 1 & 2...");
}
}
}
}
}
```

When you compile and execute the above program, you find a deadlock situation and following is the output produced by the program:

Thread 1: Holding lock 1...
Thread 2: Holding lock 2...
Thread 1: Waiting for lock 2...
Thread 2: Waiting for lock 1...
The above program will hang forever because neither of the threads in position to proceed and waiting for each other to release the lock, so you can come out of the program by pressing CTRL+C.

Thread Control
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.	Methods with Description
1	**public void suspend()** This method puts a thread in the suspended state and can be resumed using resume() method.
2	**public void stop()** 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.

Applets

Hello World: The Applet

The reason people are excited about Java as more than just another OOP language is because it allows them to write interactive applets on the web. Hello World isn't a very interactive program, but let's look at a webbed version.

```
import java.applet.Applet;

import java.awt.Graphics;



public class HelloWorldApplet extends Applet {



  public void paint(Graphics g) {

    g.drawString("Hello world!", 50, 25);

  }



}
```

The applet version of HelloWorld is a little more complicated than the HelloWorld application, and it will take a little more effort to run it as well.

First type in the source code and save it into file called HelloWorldApplet.java. Compile this file in the usual way. If all is well a file called HelloWorldApplet.class will be created. Now you need to create an HTML file that will include your applet. The following simple HTML file will do.

```
<HTML>
<HEAD>
<TITLE> Hello World </TITLE>
</HEAD>

<BODY>
        This is the applet:<P>
        <applet code="HelloWorldApplet.class" width="150" height="50">
        </applet>
</BODY>
</HTML>
```

Save this file as HelloWorldApplet.html in the same directory as the HelloWorldApplet.class file. When you've done that, load the HTML file into a Java enabled browser like Internet Explorer 4.0 or Sun's applet viewer included with the JDK. You should see something like below, though of course the exact details depend on which browser you use.

If you're using the JDK 1.1 to compile your program, you should use the applet viewer, HotJava, Internet Explorer 4.0 or later, or Netscape 4.0.6 or later on Windows and Unix to view the applet. Netscape Navigator 4.0.5 and earlier and 3.x versions of Internet Explorer do not support Java 1.1. Furthermore, no Mac version of Navigator supports Java 1.1.

If the applet compiled without error and produced a HelloWorldApplet.class file, and yet you don't see the string "Hello World" in your browser chances are that the .class file is in the wrong place. Make sure HelloWorldApplet.class is in the same directory as HelloWorld.html. Also make sure that you're using a version of Netscape or Internet Explorer which supports Java. Not all versions do.

In any case Netscape's Java support is less than the perfect so if you have trouble with an applet, the first thing to try is loading it into Sun's Applet Viewer instead. If the Applet Viewer has a problem, then chances are pretty good the problem is with the applet and not with the browser.

What is an Applet?

According to Sun "An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application....The Applet class provides a standard interface between applets and their environment."

Four definitions of applet:

- A small application
- A secure program that runs inside a web browser
- A subclass of java.applet.Applet
- An instance of a subclass of java.applet.Applet

public class Applet extends Panel

```
java.lang.Object
  |
  +----java.awt.Component
        |
        +----java.awt.Container
              |
              +----java.awt.Panel
                    |
                    +----java.applet.Applet
```

The APPLET HTML Tag

Applets are embedded in web pages using the <APPLET> and </APPLET> tags. The <APPLET> tag is similar to the <IMG> tag. Like <IMG> <APPLET> references a source file that is not part of the HTML page on which it is embedded. IMG elements do this with the SRC attribute. APPLET elements do this with the CODE attribute. The CODE attribute tells the browser where to look for the compiled .class file. It is relative to the location of the source document. Thus if you're browsing http://metalab.unc.edu/javafaq/index.html and that page references an applet with CODE="Animation.class", then the file Animation.class should be at the URL http://metalab.unc.edu/javafaq/animation.class.

For reasons that remain a mystery to HTML authors everywhere if the applet resides somewhere other than the same directory as the page it lives on, you don't just give a URL to its location. Rather you point at the CODEBASE. The CODEBASE attribute is a URL that points at the directory where the .class file is. The CODE attribute is the name of the .class file itself. For instance if on the HTML page of the previous section you had written

```
<APPLET
        CODE="HelloWorldApplet.class"
        CODEBASE="classes"
        WIDTH=200 HEIGHT=200>
</APPLET>
```

then the browser would have tried to find HelloWorldApplet.class in the classes directory in the same directory as the HTML page that included the applet. On the other hand if you had written

```
<APPLET
        CODE="HelloWorldApplet.class"
        CODEBASE="http://www.foo.bar.com/classes"
        WIDTH=200
        HEIGHT=200>
</APPLET>
```

then the browser would try to retrieve the applet from http://www.foo.bar.com/classes/HelloWorldApplet.class regardless of where the HTML page was.

In short the applet viewer will try to retrieve the applet from the URL given by the formula (CODEBASE + "/" + code). Once this URL is formed all the usual rules about relative and absolute URLs apply.

You can leave off the .class extension and just use the class name in the CODE attribute. For example,

```
<APPLET CODE="HelloWorldApplet"
        CODEBASE="http://www.foo.bar.com/classes"
        WIDTH=200
        HEIGHT=200>
</APPLET>
```

If the applet is in a non-default package, then the full package qualified name must be used. For example,

```
<APPLET
        CODE="com.macfaq.greeting.HelloWorldApplet"
        CODEBASE="http://www.foo.bar.com/classes"
        WIDTH=200
        HEIGHT=200>
</APPLET>
```

In this case the browser will look for

http://www.foo.bar.com/classes/com/macfaq/greeting/HelloWorldApplet.class so the directory structure on the server should also mirror the package hierarchy.

The HEIGHT and WIDTH attributes work exactly as they do with IMG, specifying how big a rectangle the browser should set aside for the applet. These numbers are specified in pixels and are required.

Spacing Preferences

The <APPLET> tag has several attributes to define how it is positioned on the page.

The ALIGN attribute defines how the applet's rectangle is placed on the page relative to other elements. Possible values include LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM and ABSBOTTOM. This attribute is optional.

You can specify an HSPACE and a VSPACE in pixels to set the amount of blank space between an applet and the surrounding text. The HSPACE and VSPACE attributes are optional.

```
<APPLET
        code="HelloWorldApplet.class"
        CODEBASE="http://www.foo.bar.com/classes"
        width=200
        height=200
        ALIGN=RIGHT
        HSPACE=5
        VSPACE=10>
</APPLET>
```

The ALIGN, HSPACE, and VSPACE attributes are identical to the attributes of the same name used by the <IMG> tag.

Alternate Text

The <APPLET> has an ALT attribute. An ALT attribute is used by a browser that understands the APPLET tag but for some reason cannot play the applet. For instance, if you've turned off Java in Netscape Navigator 3.0, then the browser should display the ALT text. Note that I said it **should**, not that it **does**. The ALT tag is optional.

```
<applet code="HelloWorldApplet.class"
        CODEBASE="http://www.foo.bar.com/classes" width=200 height=200
        ALIGN=RIGHT HSPACE=5 VSPACE=10
        ALT="Hello World!">
</APPLET>
```

ALT is not used by browsers that do not understand <APPLET> at all. For that purpose <APPLET> has been defined to require a closing tag, </APPLET>. All raw text between the opening and closing <APPLET> tags is ignored by a Java capable browser. However a non-Java capable browser will ignore the <APPLET> tags instead and read the text between them. For example the following HTML fragment says Hello to people both with and without Java capable browsers.

```
<APPLET code="HelloWorldApplet.class"
        CODEBASE="http://www.foo.bar.com/classes"
        width=200
        height=200
        ALIGN=RIGHT HSPACE=5 VSPACE=10
        ALT="Hello World!">
```

```
        Hello World!<P>
</APPLET>
```

Naming Applets

You can give an applet a name by using the NAME attribute of the APPLET tag. This allows communication between different applets on the same Web page.

```
<APPLET
        code="HelloWorldApplet.class"
        Name=Applet1
        CODEBASE="http://www.foo.bar.com/classes"
        width=200
        height=200
        ALIGN=RIGHT HSPACE=5 VSPACE=10
        ALT="Hello World!">
        Hello World!<P>
</APPLET>
```

JAR Archives

HTTP 1.0 uses a separate connection for each request. When you're downloading many small files, the time required to set up and tear down the connections can be a significant fraction of the total amount of time needed to load a page. It would be better if you could load all the HTML documents, images, applets, and sounds a page needed in one connection.

One way to do this without changing the HTTP protocol, is to pack all those different files into a single archive file, perhaps a zip archive, and just download that.

We aren't quite there yet. Browsers do not yet understand archive files, but in Java 1.1 applets do. You can pack all the images, sounds, and .class files an applet needs into one JAR archive and load that instead of the individual files. Applet classes do not have to be loaded directly. They can also be stored in JAR archives. To do this you use the ARCHIVES attribute of the APPLET tag

```
<APPLET
        CODE=HelloWorldApplet
        WIDTH=200 HEIGHT=100
        ARCHIVES="HelloWorld.jar">
        <hr>
                Hello World!
        <hr>
</APPLET>
```

In this example, the applet class is still HelloWorldApplet. However, there is no HelloWorldApplet.class file to be downloaded. Instead the class is stored inside the archive file HelloWorld.jar.

Sun provides a tool for creating JAR archives with its JDK 1.1. For example,

```
% jar cf HelloWorld.jar *.class
```

This puts all the .class files in the current directory in the file named "HelloWorld.jar". The syntax of the jar command is deliberately similar to the Unix tar command.

The OBJECT Tag

HTML 4.0 deprecates the <APPLET> tag. Instead you are supposed to use the <OBJECT> tag. For the purposes of ewbedding applets, the <OBJECT> tag is used almost exactly like the <APPLET> tag except that the class attribute becomes the classid attribute. For example,

```
<OBJECT classid="MyApplet.class"
        CODEBASE="http://www.foo.bar.com/classes" width=200 height=200
        ALIGN=RIGHT HSPACE=5 VSPACE=10>
</OBJECT>
```

The <OBJECT> tag is also used to embed ActiveX controls and other kinds of active content, and it has a few additional attributes to allow it to do that. However, for the purposes of Java you don't need to know about these.

The <OBJECT> tag is supported by Netscape 4.0 and later and Internet Explorer 4.0 and later. It is not supported by earlier versions of those browsers so <APPLET> is unlikely to disappear anytime soon.

You can support both by placing an <APPLET> element inside an <OBJECT> element like this:

```
<OBJECT classid="MyApplet.class" width=200 height=200>
        <APPLET code="MyApplet.class" width=200 height=200>
        </APPLET>
</OBJECT>
```

Browsers that understand <OBJECT> will ignore its content while browsers that don't will display its content.

PARAM elements are the same for <OBJECT> as for <APPLET>.

Finding an Applet's Size

When running inside a web browser the size of an applet is set by the height and width attributes and cannot be changed by the applet. Many applets need to know their own size. After all you don't want to draw outside the lines. :-)

Retrieving the applet size is straightforward with the getSize() method. java.applet.Applet inherits this method from java.awt.Component. getSize() returns a java.awt.Dimension object. A Dimension object has two public int fields, height and width. Below is a simple applet that prints its own dimensions.

```
import java.applet.*;

import java.awt.*;


public class SizeApplet extends Applet {


  public void paint(Graphics g) {

    Dimension appletSize = this.getSize();

    int appletHeight = appletSize.height;

    int appletWidth = appletSize.width;


    g.drawString("This applet is " + appletHeight +

      " pixels high by " + appletWidth + " pixels wide.",
```

```
    15, appletHeight/2);
  }
}
```

Note how the applet's height is used to decide where to draw the text. You'll often want to use the applet's dimensions to determine how to place objects on the page. The applet's width wasn't used because it made more sense to left justify the text rather than center it. In other programs you'll have occasion to use the applet width too.

Passing Parameters to Applets

Parameters are passed to applets in NAME=VALUE pairs in <PARAM> tags between the opening and closing APPLET tags. Inside the applet, you read the values passed through the PARAM tags with the getParameter() method of the java.applet.Applet class.

The program below demonstrates this with a generic string drawing applet. The applet parameter "Message" is the string to be drawn.

```
import java.applet.*;

import java.awt.*;


public class DrawStringApplet extends Applet {

  private String defaultMessage = "Hello!";

   public void paint(Graphics g) {

    String inputFromPage = this.getParameter("Message");

    if (inputFromPage == null) inputFromPage = defaultMessage;

    g.drawString(inputFromPage, 50, 25);

  }

  }
```

You also need an HTML file that references your applet. The following simple HTML file will do:

```
<HTML>
<HEAD>
<TITLE> Draw String </TITLE>
</HEAD>

<BODY>
        This is the applet:<P>
        <APPLET code="DrawStringApplet.class" width="300" height="50">
                <PARAM name="Message" value="Howdy, there!">
                This page will be very boring if your
                browser doesn't understand Java.
        </APPLET>
</BODY>
</HTML>
```

Of course you are free to change "Howdy, there!" to a "message" of your choice. You only need to change the HTML, not the Java source code. PARAMs let you customize applets without changing or recompiling the code.

This applet is very similar to the HelloWorldApplet. However rather than hardcoding the message to be printed it's read into the variable inputFromPage from a PARAM in the HTML.

You pass getParameter() a string that names the parameter you want. This string should match the name of a <PARAM> tag in the HTML page. getParameter() returns the value of the parameter. All values are passed as strings. If you want to get another type like an integer, then you'll need to pass it as a string and convert it to the type you really want.

The <PARAM> HTML tag is also straightforward. It occurs between <APPLET> and </APPLET>. It has two attributes of its own, NAME and VALUE. NAME identifies which PARAM this is. VALUE is the value of the PARAM as a String. Both should be enclosed in double quote marks if they contain white space.

An applet is not limited to one PARAM. You can pass as many named PARAMs to an applet as you like. An applet does not necessarily need to use all the PARAMs that are in the HTML. Additional PARAMs can be safely ignored.

Processing An Unknown Number Of Parameters

Most of the time you have a fairly good idea of what parameters will and won't be passed to your applet. However some of the time there will be an undetermined number of parameters. For instance Sun's imagemap applet passes each "hot button" as a parameter. Different imagemaps have different numbers of hot buttons. Another applet might want to pass a series of URL's to different sounds to be played in sequence. Each URL could be passed as a separate parameter.

Or perhaps you want to write an applet that displays several lines of text. While it would be possible to cram all this information into one long string, that's not too friendly to authors who want to use your applet on their pages. It's much more sensible to give each line its own <PARAM> tag. If this is the case, you should name the tags via some predictable and numeric scheme. For instance in the text example the following set of <PARAM> tags would be sensible:

```
<PARAM name="Line1" value="There once was a man from Japan">
<PARAM name="Line2" value="Whose poetry never would scan">
<PARAM name="Line3" value="When asked reasons why,">
<PARAM name="Line4" value="He replied, with a sigh:">
<PARAM name="Line5" value="I always try to get as many
        syllables into the last line as I can.">
```

The program below displays this limerick. Lines are accumulated into an array of strings called poem. A for loop fills the array with the different lines of poetry. There are 101 spaces in the array, but since you won't normally need that many, an if clause tests to see whether the attempt to get a parameter was successful by checking to see if the line is null. As soon as one fails, the loop is broken. Once the loop is finished num_lines is decremented by one because the last line the loop tried to read wasn't there.

The paint() method loops through the poem array and prints each String on the screen, incrementing the y position by fifteen pixels each step so you don't draw one line on top of the other.

Processing An Unknown Number Of Parameters

```
import java.applet.*;

import java.awt.*;


public class PoetryApplet extends Applet
```

```
{
  private String[] poem = new String[101];
  private int numLines;
  public void init() {
    String nextline;
    for (numLines = 1; numLines < poem.length; numLines++) {
      nextline = this.getParameter("Line" + numLines);
      if (nextline == null) break;
      poem[numLines] = nextline;
    }
    numLines--;
  }
    public void paint(Graphics g) {
    int y = 15;
    for (int i=1; i <= numLines; i++) {
      g.drawString(poem[i], 5, y);
      y += 15;
    }
  }
}
```

You might think it would be useful to be able to process an arbitrary list of parameters without knowing their names in advance, if nothing else so you could return an error message to the page designer. Unfortunately there's no way to do it in Java 1.0 or 1.1. It may appear in future versions.

Applet Security

The possibility of surfing the Net, wandering across a random page, playing an applet and catching a virus is a fear that has scared many uninformed people away from Java. This fear has also driven a lot of the development of Java in the direction it's gone. Earlier I discussed various security features of Java including automatic garbage collection, the elimination of pointer arithmetic and the Java interpreter. These serve the dual purpose of making the language simple for programmers and secure for users. You can surf the web without worrying that a Java applet will format your hard disk or introduce a virus into your system.

In fact both Java applets and applications are much safer in practice than code written in traditional languages. This is because even code from trusted sources is likely to have bugs. However Java programs are much less susceptible to common bugs involving memory access than are programs written in traditional languages like C. Furthermore the Java runtime environment provides a fairly robust means of trapping bugs before they bring down your system. Most users have many more problems with bugs than they do with deliberately malicious code. Although users of Java applications aren't protected from out and out malicious code, they are largely protected from programmer errors.

Applets implement additional security restrictions that protect users from malicious code too. This is accomplished through the java.lang.SecurityManager class. This class is subclassed to provide different security environments in different virtual machines. Regrettably implementing this additional level of protection does somewhat restrict the actions an applet can perform. Let's explore exactly what an applet can and cannot do.

What Can an Applet Do?

An applet can:

- Draw pictures on a web page
- Create a new window and draw in it.
- Play sounds.
- Receive input from the user through the keyboard or the mouse.
- Make a network connection to the server from which it came and can send to and receive arbitrary data from that server.

Anything you can do with these abilities you can do in an applet. An applet cannot:

- Write data on any of the host's disks.
- Read any data from the host's disks without the user's permission. In some environments, notably Netscape, an applet cannot read data from the user's disks even with permission.
- Delete files
- Read from or write to arbitrary blocks of memory, even on a non-memory-protected operating system like the MacOS. All memory access is strictly controlled.
- Make a network connection to a host on the Internet other than the one from which it was downloaded.
- Call the native API directly (though Java API calls may eventually lead back to native API calls).
- Introduce a virus or trojan horse into the host system.
- An applet is not supposed to be able to crash the host system. However in practice Java isn't quite stable enough to make this claim yet.

Who Can an Applet Talk To?

By default an applet can only open network connections to the system from which the applet was downloaded. This system is called the *codebase*. An applet cannot talk to an arbitrary system on the Internet. Any communication between different client systems must be mediated through the server.

The concern is that if connections to arbitrary hosts were allowed, then a malicious applet might be able to make connections to other systems and launch network based attacks on other machines in an organization's internal network. This would be an especially large problem because the machine's inside a firewall may be configured to trust each other more than they would trust any random machine from the Internet. If the internal network is properly protected by a firewall, this might be the only way an external machine could even talk to an internal machine. Furthermore arbitrary network connections would allow crackers to more easily hide their true location by passing their attacks through several applet intermediaries.

HotJava, Sun's applet viewer, and Internet Explorer (but not Netscape) let you grant applets permission to open connections to any system on the Internet, though this is not enabled by default.

How much CPU time does an applet get?

One of the few legitimate concerns about hostile applets is excessive use of CPU time. It is possible on a non-preemptively multitasking system (specifically the Mac) to write an applet that uses so much CPU time in a tight loop that it effectively locks up the host system. This is not a problem on preemptively multitasking systems like Solaris and Windows NT. Even on those platforms, though, it is possible for an applet to force the user to kill their web browser, possibly losing accumulated bookmarks, email and other work.

It's also possible for an applet to use CPU time for purposes other than the apparent intent of the applet. For instance, a popular applet could launch a Chinese lottery attack on a Unix password file. A popular game applet could launch a thread in the background which tried a random assortment of keys to break a DES encrypted file. If the key was found, then a network connection could be opened to the applet server to send the decrypted key back. The more popular the applet was the faster the key would be found. The ease with which Java applets are

decompiled would probably mean that any such applet would be discovered, but there really isn't a way to prevent it from running in the first place.

User Security Issues and Social Engineering

Contrary to popular belief most computer break-ins by external hackers don't happen because of great knowledge of operating system internals and network protocols. They happen because a hacker went digging through a company's garbage and found a piece of paper with a password written on it, or perhaps because they talked to a low-level bureaucrat on the phone, convinced this person they were from the local data processing department and that they needed him or her to change their password to "DEBUG."

This is sort of attack is called social engineering. Java applets introduce a new path for social engineering. For instance imagine an applet that pops up a dialog box that says, "You have lost your connection to the network. Please enter your username and password to reconnect." How many people would blindly enter their username and password without thinking? Now what if the box didn't really come from a lost network connection but from a hacker's applet? And instead of reconnecting to the network (a connection that was never lost in the first place) the username and password was sent over the Internet to the cracker? See the problem?

Preventing Applet Based Social Engineering Attacks

To help prevent this, Java applet windows are specifically labeled as such with an ugly bar that says: "Warning: Applet Window" or "Unsigned Java Applet Window." The exact warning message varies from browser to browser but in any case should be enough to prevent the more obvious attacks on clueless users. It still assumes the user understands what "Unsigned Java Applet Window" means and that they shouldn't type their password or any sensitive information in such a window. User education is the first part of any real security policy.

Content Issues

Some people claim that Java is insecure because it can show the user erotic pictures and play flatulent noises. By this standard the entire web is insecure. Java makes no determination of the content of an applet. Any such determination would require artificial intelligence and computers far more powerful than what we have today.

The Basic Applet Life Cycle

1. The browser reads the HTML page and finds any <APPLET> tags.
2. The browser parses the <APPLET> tag to find the CODE and possibly CODEBASE attribute.
3. The browser downloads the .class file for the applet from the URL found in the last step.
4. The browser converts the raw bytes downloaded into a Java class, that is a java.lang.Class object.
5. The browser instantiates the applet class to form an applet object. This requires the applet to have a noargs constructor.
6. The browser calls the applet's init() method.
7. The browser calls the applet's start() method.
8. While the applet is running, the browser passes any events intended for the applet, e.g. mouse clicks, key presses, etc., to the applet's handleEvent() method. Update events are used to tell the applet that it needs to repaint itself.
9. The browser calls the applet's stop() method.
10. The browser calls the applet's destroy() method.

The Basic Applet Life Cycle

All applets have the following four methods:

```
public void init();
public void start();
public void stop();
public void destroy();
```

They have these methods because their superclass, java.applet.Applet, has these methods. (It has others too, but right now I just want to talk about these four.)

In the superclass, these are simply do-nothing methods. For example,

```
public void init() {}
```

Subclasses may override these methods to accomplish certain tasks at certain times. For instance, the init() method is a good place to read parameters that were passed to the applet via <PARAM> tags because it's called exactly once when the applet starts up. However, they do not have to override them. Since they're declared in the superclass, the Web browser can invoke them when it needs to without knowing in advance whether the method is implemented in the superclass or the subclass. This is a good example of polymorphism.

init(), start(), stop(), and destroy()

The init() method is called exactly once in an applet's life, when the applet is first loaded. It's normally used to read PARAM tags, start downloading any other images or media files you need, and set up the user interface. Most applets have init() methods.

The start() method is called at least once in an applet's life, when the applet is started or restarted. In some cases it may be called more than once. Many applets you write will not have explicit start()methods and will merely inherit one from their superclass. A start() method is often used to start any threads the applet will need while it runs.

The stop() method is called at least once in an applet's life, when the browser leaves the page in which the applet is embedded. The applet's start() method will be called if at some later point the browser returns to the page containing the applet. In some cases the stop() method may be called multiple times in an applet's life. Many applets you write will not have explicit stop()methods and will merely inherit one from their superclass. Your applet should use the stop() method to pause any running threads. When your applet is stopped, it *should* not use any CPU cycles.

The destroy() method is called exactly once in an applet's life, just before the browser unloads the applet. This method is generally used to perform any final clean-up. For example, an applet that stores state on the server might send some data back to the server before it's terminated. many applets will not have explicit destroy() methods and just inherit one from their superclass.

For example, in a video applet, the init() method might draw the controls and start loading the video file. The start() method would wait until the file was loaded, and then start playing it. The stop() method would pause the video, but not rewind it. If the start() method were called again, the video would pick up where it left off; it would not start over from the beginning. However, if destroy() were called and then init(), the video would start over from the beginning.

In the JDK's appletviewer, selecting the Restart menu item calls stop() and then start(). Selecting the Reload menu item calls stop(), destroy(), and init(), in that order. (Normally the byte codes will also be reloaded and the HTML file reread though Netscape has a problem with this.)

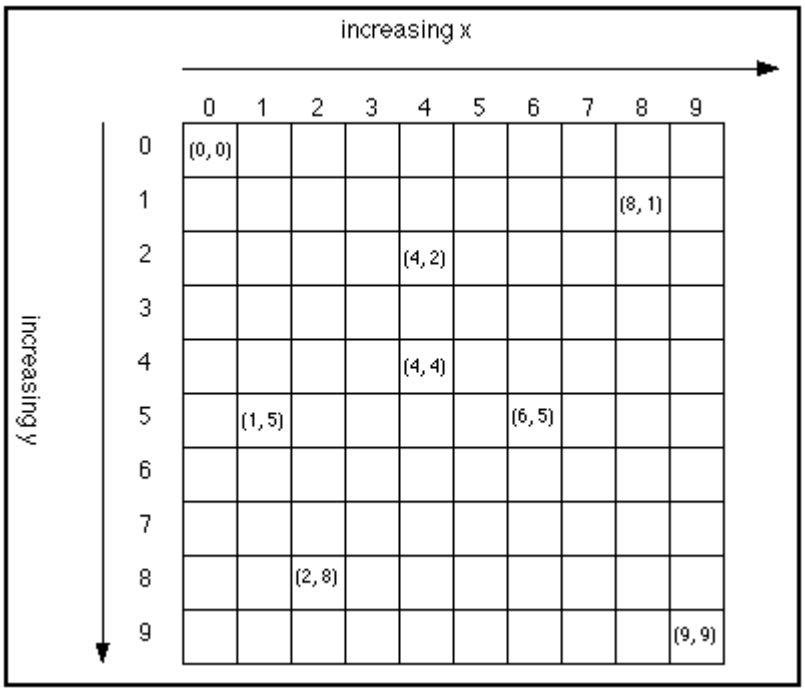
The applet start() and stop() methods are not related to the similarly named methods in the java.lang.Thread class.

Your own code may occasionally invoke start() and stop(). For example, it's customary to stop playing an animation when the user clicks the mouse in the applet and restart it when they click the mouse again.

Your own code can also invoke init() and destroy(), but this is normally a bad idea. Only the environment should call init() and destroy().

The Coordinate System

Java uses the standard, two-dimensional, computer graphics coordinate system. The first visible pixel in the upper left-hand corner of the applet canvas is (0, 0). Coordinates increase to the right and down.



Graphics Objects

In Java all drawing takes place via a Graphics object. This is an instance of the class java.awt.Graphics.

Initially the Graphics object you use will be the one passed as an argument to an applet's paint() method. Later you'll see other Graphics objects too. Everything you learn today about drawing in an applet transfers directly to drawing in other objects like Panels, Frames, Buttons, Canvases and more.

Each Graphics object has its own coordinate system, and all the methods of Graphics including those for drawing Strings, lines, rectangles, circles, polygons and more. Drawing in Java starts with particular Graphics object. You get access to the Graphics object through the paint(Graphics g) method of your applet. Each draw method call will look like g.drawString("Hello World", 0, 50) where g is the particular Graphics object with which you're drawing.

For convenience's sake in this lecture the variable g will always refer to a preexisting object of the Graphics class. As with any other method you are free to use some other name for the particular Graphics context, myGraphics or appletGraphics perhaps.

Drawing Lines

Drawing straight lines with Java is easy. Just call

```
g.drawLine(x1, y1, x2, y2)
```

where (x1, y1) and (x2, y2) are the endpoints of your lines and g is the Graphics object you're drawing with.

This program draws a line diagonally across the applet.

```
import java.applet.*;

import java.awt.*;

public class SimpleLine extends Applet {

 public void paint(Graphics g) {

    g.drawLine(0, 0, this.getSize().width, this.getSize().height);

 }
```

```
}
```

Drawing Rectangles

Drawing rectangles is simple. Start with a Graphics object g and call its drawRect() method:

```
public void drawRect(int x, int y, int width, int height)
```

As the variable names suggest, the first int is the left hand side of the rectangle, the second is the top of the rectangle, the third is the width and the fourth is the height. This is in contrast to some APIs where the four sides of the rectangle are given.

This uses drawRect() to draw a rectangle around the sides of an applet.

```
import java.applet.*;

import java.awt.*;

public class RectangleApplet extends Applet {

  public void paint(Graphics g) {

    g.drawRect(0, 0, this.getSize().width - 1, this.getSize().height - 1);

  }

}
```

Remember that getSize().width is the width of the applet and getSize().height is its height.

Why was the rectangle drawn only to getSize().height-1 and getSize().width-1?

Remember that the upper left hand corner of the applet starts at (0, 0), not at (1, 1). This means that a 100 by 200 pixel applet includes the points with x coordinates between 0 and 99, not between 0 and 100. Similarly the y coordinates are between 0 and 199 inclusive, not 0 and 200.

There is no separate drawSquare() method. A square is just a rectangle with equal length sides, so to draw a square call drawRect() and pass the same number for both the height and width arguments.

Filling Rectangles

The drawRect() method draws an open rectangle, a box if you prefer. If you want to draw a filled rectangle, use the fillRect() method. Otherwise the syntax is identical.

This program draws a filled square in the center of the applet. This requires you to separate the applet width and height from the rectangle width and height. Here's the code:

```
import java.applet.*;

import java.awt.*;

public class FillAndCenter extends Applet {

  public void paint(Graphics g) {

    int appletHeight = this.getSize().height;

    int appletWidth  = this.getSize().width;
```

```
    int rectHeight   = appletHeight/3;
    int rectWidth    = appletWidth/3;
    int rectTop      = (appletHeight - rectHeight)/2;
    int rectLeft     = (appletWidth - rectWidth)/2;

    g.fillRect(rectLeft, rectTop, rectWidth-1, rectHeight-1);
  }
}
```

Clearing Rectangles

It is also possible to clear a rectangle that you've drawn. The syntax is exactly what you'd expect:

```
public abstract void clearRect(int x, int y, int width, int height)
```

This program uses clearRect() to blink a rectangle on the screen.

```
import java.applet.*;
import java.awt.*;

public class Blink extends Applet {
  public void paint(Graphics g) {
    int appletHeight = this.getSize().height;
    int appletWidth  = this.getSize().width;
    int rectHeight   = appletHeight/3;
    int rectWidth    = appletWidth/3;
    int rectTop      = (appletHeight - rectHeight)/2;
    int rectLeft     = (appletWidth - rectWidth)/2;

    for (int i=0; i < 1000; i++) {
      g.fillRect(rectLeft, rectTop, rectWidth-1, rectHeight-1);
      g.clearRect(rectLeft, rectTop, rectWidth-1, rectHeight-1);
    }
  }
}
```

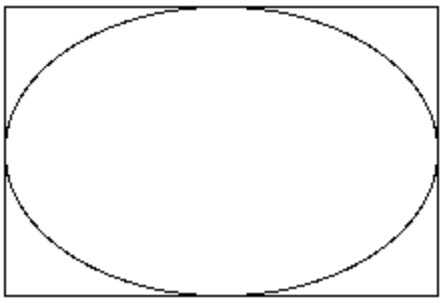
This is not how you should do animation in practice, but this is the best we can do until we introduce threads.

Ovals and Circles

Java has methods to draw outlined and filled ovals. As you'd probably guess these methods are called drawOval() and fillOval() respectively. As you might not guess they take identical arguments to drawRect() and fillRect(), i.e.

```
public void drawOval(int left, int top, int width, int height)
public void fillOval(int left, int top, int width, int height)
```

Instead of the dimensions of the oval itself, the dimensions of the smallest rectangle which can enclose the oval are specified. The oval is drawn as large as it can be to touch the rectangle's edges at their centers. This picture may help:



The arguments to drawOval() are the same as the arguments to drawRect(). The first int is the left hand side of the enclosing rectangle, the second is the top of the enclosing rectangle, the third is the width and the fourth is the height.

There is no special method to draw a circle. Just draw an oval inside a square.

Java also has methods to draw outlined and filled arcs. They're similar to drawOval() and fillOval() but you must also specify a starting and ending angle for the arc. Angles are given in degrees. The signatures are:

```
public void drawArc(int left, int top, int width, int height,
  int startangle, int stopangle)
public void fillArc(int left, int top, int width, int height,
  int startangle, int stopangle)
```

The rectangle is filled with an arc of the largest circle that could be enclosed within it. The location of 0 degrees and whether the arc is drawn clockwise or counter-clockwise are currently platform dependent.

Bullseye

This is a simple applet which draws a series of filled, concentric circles alternating red and white, in other words a bullseye.

```
import java.applet.*;

import java.awt.*;


public class Bullseye extends Applet {


  public void paint(Graphics g) {

    int appletHeight = this.getSize().height;

    int appletWidth = this.getSize().width;
```

```
   for (int i=8; i >= 0; i--) {
     if ((i % 2) == 0) g.setColor(Color.red);
     else g.setColor(Color.white);

     // Center the rectangle
     int rectHeight = appletHeight*i/8;
     int rectWidth  = appletWidth*i/8;
     int rectLeft   = appletWidth/2  - i*appletWidth/16;
     int rectTop    = appletHeight/2 - i*appletHeight/16;
     g.fillOval(rectLeft, rectTop, rectWidth, rectHeight);
    }
  }

}
```

The .class file that draws this image is only 684 bytes. The equivalent GIF image is 1,850 bytes, almost three times larger.

Almost all the work in this applet consists of centering the enclosing rectangles inside the applet. The lines in bold do that. The first two lines just set the height and the width of the rectangle to the appropriate fraction of the applet's height and width. The next two lines set the position of the upper left hand corner. Once the rectangle is positioned, drawing the oval is easy.

Polygons

In Java rectangles are defined by the position of their upper left hand corner, their height, and their width. However it is implicitly assumed that there is in fact an upper left hand corner. Not all rectangles have an upper left hand corner. For instance consider the rectangle below.

Where is its upper left hand corner? What's been assumed so far is that the sides of the rectangle are parallel to the coordinate axes. You can't yet handle a rectangle that's been rotated at an arbitrary angle.

There are some other things you can't handle either, triangles, stars, rhombuses, kites, octagons and more. To take care of this broad class of shapes Java has a Polygon class.

Polygons are defined by their corners. No assumptions are made about them except that they lie in a 2-D plane. The basic constructor for the Polygon class is

```
public Polygon(int[] xpoints, int[] ypoints, int npoints)
```

xpoints is an array that contains the x coordinates of the polygon. ypoints is an array that contains the y coordinates. Both should have the length npoints. Thus to construct a 3-4-5 right triangle with the right angle on the origin you would type

```
int[] xpoints = {0, 3, 0};
int[] ypoints = {0, 0, 4};
Polygon myTriangle = new Polygon(xpoints, ypoints, 3);
```

To actually draw the polygon you use java.awt.Graphics's drawPolygon(Polygon p) method within your paint() method like this:

```
g.drawPolygon(myTriangle);
```

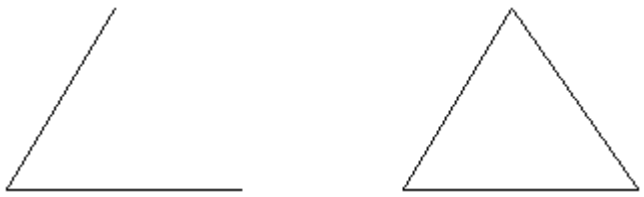
You can pass the arrays and number of points directly to the drawPolygon() method if you prefer:

```
g.drawPolygon(xpoints, ypoints, xpoints.length);
```

There's also an overloaded fillPolygon() method. The syntax is exactly as you expect:

```
g.fillPolygon(myTriangle);
g.fillPolygon(xpoints, ypoints, xpoints.length());
```

Java automatically closes the polygons it draws. That is it draws polygons that look like the one on the right rather than the one on the left.



If you don't want your polygons to be closed, you can draw a polyline instead with the Graphics class's drawPolyline() method

```
public abstract void drawPolyline(int xPoints[], int yPoints[], int nPoints)
```

Loading Images

Polygons, ovals, lines and text cover a lot of ground. The remaining graphic object you need is an image. Images in Java are bitmapped GIF or JPEG files that can contain pictures of just about anything. You can use any program at all to create them as long as that program can save in GIF or JPEG format.

Images displayed by Java applets are retrieved from the web via a URL that points to the image file. An applet that displays a picture must have a URL to the image its going to display. Images can be stored on a web server, a local hard drive or anywhere else the applet can point to via a URL. Make sure you put your images somewhere the person viewing the applet can access them. A file URL that points to your local hard drive may work while you're developing an applet, but it won't be of much use to someone who comes in over the web.

Typically you'll put images in the same directory as either the applet or the HTML file. Though it doesn't absolutely have to be in one of these two locations, storing it there will probably be more convenient. Put the image with the applet .class file if the image will be used for all instances of the applet. Put the applet with the HTML file if different instances of the applet will use different images. A third alternative is to put all the images in a common location and use PARAMs in the HTML file to tell Java where the images are.

If you know the exact URL for the image you wish to load, you can load it with the getImage() method:

```
URL imageURL = new URL("http://www.prenhall.com/logo.gif");
java.awt.Image img = this.getImage(imageURL);
```

You can compress this into one line as follows

```
Image img = this.getImage(new URL("http://www.prenhall.com/logo.gif"));
```

The getImage() method is provided by java.applet.Applet. The URL class is provided by java.net.URL. Be sure to import it.