Arrays in Java:

Arrays are manipulated by reference

Dynamically created with new

Automatically garbage collected when no longer in use.

Two ways of creating an array:

```
int integer_array[] = new int[1024];
byte octet_buffer[] = new byte[256];
```

or

```
int table[] = {1, 2, 4, 8, 16};
```

Multidimensional arrays:

```
matrix two_dim_array[][] = new int[256][256];
```


Arrays in Java

.. are implemented as arrays of arrays. So they do not have to be rectangular

```
short triangle[][] = new short[10][]; // single dimensional array
for (int i = 0; i < triangle.length; i++)
{
        triangle[i] = new short[i+1];
                for (int j = 0; j < i+1; j++)
                        triangle[i][j] = i + j;
}
```

you can also define and initialize triangular arrays at initialization:

```
short triangle[][] = {{1}, {1, 2}, {1, 2, 3}};
```

you can also simulate two dimensional arrays in one dimensional arrays much the same way as you would in C.

Accessing Array Elements

Array access is very similar to access in C.

```
int a[] = new int[256];

for (int i = 0; i < a.length; i++)
        a[i].length;
```

length gives the length of the array. It is a read-only field. Attempts to write it will lead to errors.

Passing Arrays to Functions:

Syntax similar to C

```
void reverse (char srtbuf[], int buf_size)
{
        char buffer[] = char[256];
        ...........
}
```

or even ..

```
void reverse (char[] srtbuf, nt buf_size)
{
        char[] buffer = new[256];
        ...........
}
```


More on Arrays:

```
char[] char_array = new char[256];

int[] vector = new int[256];

int[] matrix[]; // note that matrix is really an array of arrays;

// you can now do stuff like

matrix = new int[256][];
for (int i=0; i< matrix.length; i++)
        matrix[i] = new int[256];

// now you have defined a two dimensional matrix.
```

Strings in Java

These are not null terminates char strings. Instead, they are instances of the String class in java.lang.Strings support a variety of methods such as charAt(), equals(), indexOf(), substring(). In addition, it also supports operations like “+” (concatenation).

The Java development environment provides two classes that store and manipulate character data: String, for immutable strings, and StringBuffer, for mutable strings.

```
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);

        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```


What else can you do with Strings?

```
class Filename {
    String fullpath;
    char pathseparator;

    Filename(String str, char sep) {
        fullpath = str;
        pathseparator = sep;
    }

    String extension() {
        int dot = fullpath.lastIndexOf('.');
        return fullpath.substring(dot + 1);
    }

    String filename() {
        int dot = fullpath.lastIndexOf('.');
        int sep = fullpath.lastIndexOf(pathseparator);
        return fullpath.substring(sep + 1, dot);
    }

    String path() {
        int sep = fullpath.lastIndexOf(pathseparator);
        return fullpath.substring(0, sep);
    }
}
```


What else you can do with strings..

```
StringBuffer sb = new StringBuffer(“Drink Java!”);
sb.insert(6, “Hot “);
System.out.println(sb.toString());
```

This code snippet prints

```
Drink Hot Java!
```

Another useful StringBuffer modifier is setCharAt() which sets the character at a specific location in the StringBuffer. setCharAt() is useful when you want to reuse a StringBuffer.

The toString() Method

It’s often convenient or necessary to convert an object to a String because you need to pass it to a method that only accepts String values. For example, System.out.println() does not accept StringBuffers, so you need to convert a StringBuffer to a String before you can print it.

The valueOf() Method

As a convenience, the String class provides the static method valueOf() which you can use to convert variables of different types to Strings. For example, to print the value of pi

```
System.out.println(String.valueOf(Math.PI));
```


Abstract Classes:

Any class with an abstract method is an abstract class and must be declared as such

An abstract class cannot be instantiated

A subclass of an abstract class can be instantiated if it overrides each of the abstract methods

If a subclass does not override all of a class’ abstract methods, it is abstract itself.

```
public abstract class Shape {
    public abstract double area();
    public abstract double circumference();
}
public class Circle extends Shape {
    double x, y, r;
    public Circle() {
        x = 0.; y = 0.; r = 0.;
    }
    public area() {
        return 3.14 * r * r;
    }
    public circumference() {
        return 2 * 3.14 * r;
    }
}
```

Interfaces

Let us now try to extend our Shape class to include something called DrawableShapes.This can be implemented as a subclass of Shapes. We can further have subclasses of DrawableShapes like DrawableCircle and DrawableRectangle. However, since we want the area and circumference methods to be defined on them as well, we want them to be subclasses of Circle and Rectangle respectively. But Java only allows a single superclass.

Solution: use an interface:

```
public interface Drawable {
    public void setColor(Color c);
    public void setPosition(double x, double y);
    public void draw(DrawWindow dw);
}
```

Now we can say:

```
public class DrawableCircle extends Circle implements Drawable {
    // constructors here
    // here you have methods for setColor, setPosition, and draw
}
```


Programming Threads:

What Are Threads?

A thread--sometimes known as an execution context -- is a single sequential flow of control within a process.

Let's begin our exploration of the application with the SimpleThread class: a subclass of the Thread class that is provided by the java.lang package.

```
class SimpleThread extends Thread {
   public SimpleThread(String str) {
      super(str);
   }
   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.println(i + " " + getName());
         try {
            sleep((int)(Math.random() * 1000));
         } catch (InterruptedException e) {}
      }
      System.out.println("DONE! " + getName());
   }
}
```


Creating multiple threads:

The TwoThreadsTest class provides a main() method that creates two SimpleThread threads: one is named "study" and the other is named "party". (If you can't decide what to do you can use this program to help you decide--do the one whose thread prints "DONE!" first.)

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


Thread Attributes

Java threads are implemented by the Thread class which is part of the java.lang package.

Thread Body

All the action takes place in the thread body which is the thread's run() method. After a thread has been created and initialized, the runtime system calls its run() method. The code in the run() method implements the behaviour for which the thread was created. It's the thread's raison d'etre (reason to be).

The simplest way of creating a thread is to subclass the Thread class defined in the java.lang package and override the run() method.

Example: The SimpleThread class used in the the example described.

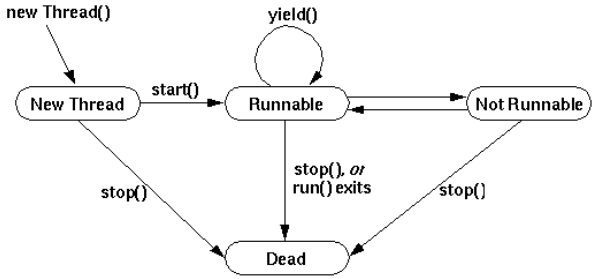
Alternately, you can create a class that implements the Runnable interface. This interface requires you to have a run() method. An instance of this class can then be passed to the Thread class. For instance:

```
class Clock extends Applet implements Runnable {
... // stuff here with a run() method in it
      public void start() {
            if (clockThread == null) {
                  clockThread = new Thread(this, "Clock");
            clockThread.start();
            }
      }
   }
```

```
// try this applet and see what it does on your web page.
import java.awt.Graphics;
import java.util.Date;

public class Clock extends java.applet.Applet implements Runnable {

   Thread clockThread;

   public void start() {
      if (clockThread == null) {
         clockThread = new Thread(this, "Clock");
         clockThread.start();
      }
   }
   public void run() {
      while (clockThread != null) {
         repaint();
         try {
            clockThread.sleep(1000);
         } catch (InterruptedException e){
         }
      }
   }
   public void paint(Graphics g) {
      Date now = new Date();
      g.drawString(now.getHours() + ":" + now.getMinutes() + ":" + now.getSeconds(), 5, 10);
   }
   public void stop() {
      clockThread.stop();
      clockThread = null;
   }
}
```


Thread State

The following diagram illustrates the various states that a Java thread can be in at any point during its life and which method calls cause a transition to another state.



New Thread

The following statement creates a new thread but does not start it thereby leaving the thread in the state labeled "New Thread" in the diagram.

```
Thread myThread = new MyThreadClass();
```

When a thread is in the "New Thread" state, it is merely an empty Thread object. No system resources have been allocated for it yet.

Runnable

Now consider these two lines of code:

```
Thread myThread = new MyThreadClass();
myThread.start();
```

The start() method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's run() method.

Not Runnable

A thread enters the "Not Runnable" state when one of these four events occur:

someone calls its suspend() method
someone calls its sleep() method
the thread uses its wait() method to wait on a condition variable
the thread is blocking on I/O.

For example, the bold line in this code snippet

```
Thread myThread = new MyThreadClass();
myThread.start();
try {
   myThread.sleep(10000);
} catch (InterruptedException e){
}
```

puts myThread to sleep for 10 seconds

Dead

A thread can die in two ways: either from natural causes, or by being killed (stopped). A thread dies naturally when its run() method exits normally. For example, the while loop in this method is a finite loop--it will iterate 100 times and then exit.

```
public void run() {
   int i = 0;
   while (i < 100) {
      i++;
      System.out.println("i = " + i);
   }
}
```

A thread with this run() method will die naturally after the loop and the run() method completes. You can also kill a thread at any time by calling its stop() method. This code snippet

```
Thread myThread = new MyThreadClass();
myThread.start();
try {
   Thread.currentThread().sleep(10000);
} catch (InterruptedException e){
}
myThread.stop();
```

creates and starts myThread then puts the current thread to sleep for 10 seconds. When the current thread wakes up, the bold line in the code segment kills myThread.

Thread Priority

The Java runtime supports a very simple, deterministic scheduling algorithm known as fixed priority scheduling. This algorithm schedules threads based on their priority relative to other "Runnable" threads. You can modify a thread's priority at any time after its creation using the setPriority() method. Thread priorities range between MIN_PRIORITY and MAX_PRIORITY (constants defined in class Thread).

```
class SelfishRunner extends Thread {

   public int tick = 1;
   public int num;

   SelfishRunner(int num) {
      this.num = num;
   }

   public void run() {
      while (tick < 400000) {
         tick++;
         if ((tick % 50000) == 0) {
            System.out.println("Thread #" + num + ", tick = " + tick);
         }
      }
   }
}
```


The main program:

```
class RaceTest {

   final static int NUMRUNNERS = 2;

   public static void main(String args[]) {

      SelfishRunner runners[] = new SelfishRunner[NUMRUNNERS];

      for (int i = 0; i < NUMRUNNERS; i++) {
         runners[i] = new SelfishRunner(i);
         runners[i].setPriority(2);
      }
      for (int i = 0; i < NUMRUNNERS; i++) {
         runners[i].start();
      }
   }
}
```


Or try this thread:

```
class PoliteRunner extends Thread {

   public int tick = 1;
   public int num;

   PoliteRunner(int num) {
      this.num = num;
   }

   public void run() {
      while (tick < 400000) {
         tick++;
         if ((tick % 50000) == 0) {
            System.out.println("Thread #" + num + ", tick = " + tick);
            yield();
         }
      }
   }
}
```

Thread Group

In Java, all threads must be a member of a thread group. Thread groups provide a mechanism for collecting multiple threads together into a single object and manipulating those threads all at once through the group rather than individually through the threads themselves.

Creating a Thread Explicitly within a Group

The only time that you are allowed to set a thread's group is during creation.

The Thread class has three different constructors that let you set the new thread's group:

```
Thread(ThreadGroup, Runnable)
Thread(ThreadGroup, String)
Thread(ThreadGroup, Runnable, String)
```

Each of these constructors requires a ThreadGroup as its first parameter.

```
ThreadGroup myThreadGroup = new ThreadGroup("My Group of Threads");
Thread myThread = new Thread(myThreadGroup, "a thread for my group");
```

Getting a Thread's Group

To find out what group a thread is in, you can call its getThreadGroup() method.

```
theGroup = myThread.getThreadGroup();
```


The ThreadGroup Class

Methods that act on the group:

```
getMaxPriority(), setMaxPriority()
getDaemon(), setDaemon()
getName()
getParent(), parentOf()
toString()
```

Methods that Operate on All Threads within a Group

The ThreadGroup class supports several methods that allow you to modify the current state of all the threads within that group:

```
resume()
stop()
suspend()
```


Synchronization

There are many interesting situations where separate concurrently running threads share data and must consider the state and activities of those other threads. One such set of programming situations are known as Producer/Consumer scenarios where the Producer generates a stream of data which then is consumed by a Consumer.

Producer/Consumer Example

The Producer generates integers ranging from 0 to 9, stores it in a "CubbyHole" object, prints the generated number, and (just to make the synchronization problem more interesting) the Producer sleeps for a random amount of time between 0 and 100 milliseconds.

```
class Producer extends Thread {
private CubbyHole cubbyhole;
private int number;
public Producer(CubbyHole c, int number) {
    cubbyhole = c;
    this.number = number;
}
public void run() {
    for (int i = 0; i < 10; i++) {
        cubbyhole.put(i);
        System.out.println("Producer #" + this.number + " put: " + i);
        try {
            sleep((int)(Math.random() * 100));
        } catch (InterruptedException e) {
        }
    }
}
}
```


The Producer-Consumer Problem Continued.

The Consumer, being ravenous, consumes all integers from the CubbyHole (the exact same object into which the Producer put the integers in the first place) as quickly as they become available.

```
class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number + " got: " + value);
        }
    }
}
```

Problem: Producer may produce faster than consumer can consume, and vice versa.

Synchronizing producer and consumer:

Objects such as the CubbyHole, which are shared between two threads and whose accesses must be synchronized, are called condition variables.

Monitors

A monitor is associated with a specific data item (the condition variable) and functions as a lock on that data. When a thread holds the monitor for some data item, other threads are locked out and cannot inspect or modify the data.

The code segments within a program that make it possible for separate, concurrent threads to access the same data items are known as critical sections. In the Java language, you identify **critical sections** in your program with the synchronized keyword.

In the Java language, a unique monitor is associated with every object that has a synchronized method. The CubbyHole class for the Producer/Consumer example introduced above has two synchronized methods: the put() method used to change the value in the CubbyHole and the get() method used to retrieve the current value.


```
class CubbyHole {
   private int seq;
   private boolean available = false;
   public synchronized int get() {
      while (available == false) {
         try {
            wait();
         } catch (InterruptedException e) {
         }
      }
      available = false;
      notify();
      return seq;
   }
   public synchronized void put(int value) {
      while (available == true) {
         try {
            wait();
         } catch (InterruptedException e) {
         }
      }
      seq = value;
      available = true;
      notify();
   }
}
```


Putting it all together .. the producer-consumer problem.

Here's a small stand-alone Java application that creates a CubbyHole object, one Producer, one Consumer, and then starts both the Producer and the Consumer.

```
class ProducerConsumerTest {
   public static void main(String args[]) {
      CubbyHole c = new CubbyHole();
      Producer p1 = new Producer(c, 1);
      Consumer c1 = new Consumer(c, 1);

      p1.start();
      c1.start();
   }
}
```


Deadlocks:

Two threads may wait on monitors and lock each other out. In this situation, no thread makes any progress. A very simple case can be constructed in which two monitors are requested one after the other without releasing the first one. If the same operation was being performed (except the monitors were being requested in the reverse order), there would be an infinite wait. This is called a deadlock.

Importing Classes from Packages

Consider the following application that displays the current date and time.

```
import java.util.Date;
class DateApp {
   public static void main (String args[]) {
      Date today = new Date();
      System.out.println(today);
   }
}
```

The first line in the program imports the Date class from the java.util package. Importing a class from a package makes that class available to the file into which it was imported. The java.util package is a collection of classes that provide miscellaneous functionality. The Date class from the java.util package lets you manage and manipulate calendar dates in a system independent way.

Let us look at other packages and their classes.

Java Packages

Java interfaces and classes are grouped into packages. The following are the java packages, from which you can access interfaces and classes, and then fields, constructors, and methods.

java.lang: Package that contains essential Java classes, including numerics, strings, objects, compiler, runtime, security, and threads. This is the only package that is automatically imported into every Java program.
java.io: Package that provides classes to manage input and output streams to read data from and write data to files, strings, and other sources.
java.util: Package that contains miscellaneous utility classes, including generic data structures, bit sets, time, date, string manipulation, random number generation, system properties, notification, and enumeration of data structures.
java.net: Package that provides classes for network support, including URLs, TCP sockets, UDP sockets, IP addresses, and a binary-to-text converter.
java.awt: Package that provides an integrated set of classes to manage user interface components such as windows, dialog boxes, buttons, checkboxes, lists, menus, scrollbars, and text fields. (AWT = Abstract Window Toolkit)
java.awt.image: Package that provides classes for managing image data, including color models, cropping, color filtering, setting pixel values, and grabbing snapshots.
java.awt.peer: Package that connects AWT components to their platform-specific implementations (such as Motif widgets or Microsoft Windows controls).
java.applet: Package that enables the creation of applets through the Applet class. It also provides several interfaces that connect an applet to its document and to resources for playing audio.

The Java Language Package

The Java language package, a.k.a. java.lang, provides classes that are core to the Java language. If you are writing stand-alone Java applications, you are likely to encounter and use the classes and interfaces in this package first. The classes in this package are grouped in the following manner:

Object
The grand-daddy of all classes--the class from which all others derive.

Data Type Wrappers
A collection of classes used to wrap variables of a simple data type: Boolean, Character, Double, Float, Integer and Long. Each of these classes are subclasses of the abstract class Number.

Strings
Two classes that implement mutable and immutable character data.

System and Runtime
These two classes provide let your programs use system resources. System provides a system-independent programming interface to system resources and Runtime gives you direct system-specific access to the runtime environment. Using System Resources describes both the System and Runtime classes and their methods.

Threads
The Thread, ThreadDeath and ThreadGroup classes implement the multi-threading capabilities so important to the Java language.

Classes
The Class class provides a runtime description of a class and the ClassLoader class allows you to load classes into your program during runtime.

Math
A library of math routines and values such as pi.

Exceptions, Errors and Throwable
When an error occurs in a Java program, the program throws an object which indicates what the problem was and the state of the interpreter when the error occurred. Only objects that derive from the Throwable class can be thrown. There are two main subclasses of Throwable: Exception and Error. Exceptions are a form of Throwable that “normal” programs may try to catch. Errors are used for more catastophic errors--normal programs should not catch errors. The java.lang package contains the Throwable, Exception and Error classes, and numerous subclasses of Exception and Error that represent specific problems.

Processes
Process objects represent the system process that is created when you use Runtime to execute system commands. The java.lang packages defines and implements the generic Process class and two of its subclasses that represent processes on specific platforms: UNIXProcess and Win32Process.