Note: It is assumed that the students have some basic knowledge of C and C++ so some common keywords and other concepts are skipped (like loop, array etc if required do refer to class room note).

What is Java?

- Java is a high-level, third generation programming language.
- Compared to other programming languages, Java is most similar to C. However although Java shares much of C's syntax, it is not C. Knowing how to program in C or, better yet, C++, will certainly help you to learn Java more quickly, but you don't need to know C to learn Java. \
- Unlike C++ Java is not a superset of C. A Java compiler won't compile C code, and most large C programs need to be changed substantially before they can become Java programs.
- What's most special about Java in relation to other programming languages is that it lets you write special programs called *applets* that can be downloaded from the Internet and played safely within a web browser.
- A Java applet cannot write to your hard disk without your permission. It cannot write to arbitrary addresses in memory and thereby introduce a virus into your computer. It should not crash your system.

The ByteCode and JVM (Java Virtual Machine)

- An imaginary machine that is implemented by emulating software on a real machine
- It provides the hardware platform specifications to which you compile all java technology code.
- **JVM** is an ***interpreter*** for bytecode.
- Bytecode is a highly **optimized set of instructions** designed to be executed by the **Java run-time system** i.e. ***Java Virtual Machine (JVM).***
- Independent of any particular computer hardware, so any computer with a java interpreter can execute the compiled java program, no matter what type of computer program was compiled on.
- ByteCode and JVM are the key that allows java to solve both the security and the portability problems.
- The reason is straightforward: only the JVM needs to be **implemented for each platform**.
- *The* ***details*** *of the* ***JVM*** *will* ***differ*** *from* ***platform*** *to* ***platform,*** but all interpret the same Java bytecode.
- *The fact that a Java program is interpreted also helps to make it* ***secure***. Because the execution of every Java program is under the control of the JVM, the JVM can contain the program and prevent it from generating side effects outside of the system.
- Safety is also enhanced by certain restrictions that exist in the Java language.
- When a program is **interpreted**, it generally runs substantially **slower** than it would run if **compiled** to executable code.
- With Java, differential between the two is not so great.
- Although Java was designed for interpretation, there is technically nothing about Java that prevents on-the-fly compilation of bytecode into native code.
- ***Just in Time (JIT)***
  - Sun supplies its ***Just In Time*** (JIT) **compiler** for bytecode, which is included in the **Java 2** release.

- When the **JIT compiler** is **part** of the **JVM**, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis.
- It is important to understand that it is **not possible** to **compile** an **entire Java program** into **executable code all at once**, because Java performs various run-time checks that can be done only at run time.
- **Just-in-time** approach still **yields** a **significant performance** boost.
- Even when **dynamic compilation** is applied to bytecode, the **portability** and **safety** features still apply, because the run time system which performs the compilation still is in charge of the execution environment.

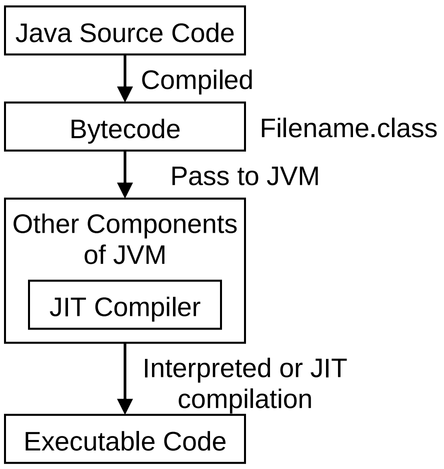

Fig.: Demonstration of JVM's Functionality (Java2)

Features of Java

- **Architecture-Neutral**
  - A central issue for the Java designer was that of code longevity and portability.
  - Software Industry is rapidly changing s/w tools.
  - JVM helps java to be run from one system to another.
  - If the architecture of the Hardware or OS changes, only the JVM will be changed and the current program will correctly.
- **Distributed**
  - Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols.
  - RMI feature of Java helps the programmer to design distributed software.
- **Dynamic**
  - Java programs carry with them substantial amount of run time type information that is used to verify and resolve accesses to objects at run time.
  - This makes it possible to dynamically link code in a safe and convenient manner.
- **Interpreted and High Performance**
  - Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode.
  - This code can be interpreted on any system that provides a JVM.
  - Relative to other interrelated languages like BASIC and Perl Java interpretation gives high performance.
- **Multithreaded**
  - Java was designed to meet the real world requirement of creating interactive, networked programs.

- To accomplish this, Java supports multithreaded programming, which allows us to write programs that do many things simultaneously.

- **Object Oriented**
  - Java is purely object oriented language because any program created in java must have class.
- **Portable**
  - JVM feature in Java provides the portability facility.
  - Even though every operating system has different architecture JVM behaves as intermediate between Java bytecode and the OS.
- **Robust**
  - The multi-plat formed environment of the web places extraordinary demands on a program, because the program must execute reliably in a variety of systems.
  - Java can be used in multiple purpose software development
- **Simple**
  - Java inherits the features of C and C++ so most of the coding in Java is similar to C and C++
  - If any one has some Idea on C and C++ then learning java is simple.
  - Even though if you don’t know C and C++ you can learn Java easily
- **Security**
  - Java also provides security up to some extents. Since all the programs of Java runs under JVM, the program cannot do any side effect outside JVM.

First program in Java

```
//HelloWorld.java
class HelloWorld {

  public static void main (String args[]) {
    System.out.println("Hello World!");
  }

}
```

- The goal of this program is not to learn how to print words to the terminal. It's to learn how to type, save and compile a program.
- To write the code you need a text editor. You can use any text editor like Notepad, Brief, emacs or vi or notepad. Personally I use vi on Linux or kawa,Textpad,editplus on Windows.
- You should not use a word processor like Microsoft Word or WordPerfect since these save their files in a proprietary format and not in pure ASCII text. If you absolutely must use one of these, be sure to tell it to save your files as pure text. Generally this will require using Save As... rather than Save.
- When you've chosen your text editor, type or copy the above program into a new file. For now type it exactly as it appears here. Like C and unlike Fortran, Java is case sensitive so System.out.println is not the same as system.out.println. CLASS is not the same as class, and so on.
- However, white space is not significant except inside string literals. The exact number of spaces or tabs you use doesn't matter.
- Save this code in a file called HelloWorld.java.

Compiling and Running Hello World

- To make sure your Java environment is correctly configured, bring up a command-line prompt and type

```
javac null.java
```

- If your computer responds with

```
error: Can't read: null.java
```

- you're ready to begin. If, on the other hand, it responds

```
javac: Command not found
```

- or something similar, then you need make sure you have the Java environment properly installed and your PATH configured.
- Assuming that Java is properly installed on your system there are three steps to creating a Java program:

1. writing the code
2. compiling the code
3. running the code

- Under Unix, compiling and running the code looks like this:

```
$ javac HelloWorld.java
$ java HelloWorld
Hello World
$
```

- Under Windows, it's similar. You just do this in a DOS shell.

```
C:> javac HelloWorld.java
C:> java HelloWorld
Hello World
C:>
```

- Notice that you use the .java extension when compiling a file, but you do not use the .class extension when running a file.

CLASSPATH Problems

- If you get any message like this,

```
$ java HelloWorld
Can't find class HelloWorld
```

- it probably means your CLASSPATH environment variable isn't properly set up. Make sure it includes the current directory as well as the location where the classes.zip file was installed. On Unix it should look something like this:

```
CLASSPATH=.:/usr/local/java-1.6/lib
```

- Under Windows it will probably look something like this

```
C:\JDK\JAVA\CLASSES;c:\java\lib\classes.zip
```

- Under Unix you set CLASSPATH variables like this:

    csh: % setenv CLASSPATH *my_class_path*

    sh: % CLASSPATH=*my_class_path*

- You'll probably want to add one of these lines to your .login or .cshrc file so it will be automatically set every time.
- Under Windows you set the CLASSPATH environment variable with a DOS command like

```
C:\> SET CLASSPATH=C:\JDK\JAVA\CLASSES;c:\java\lib\classes.zip
```

- The CLASSPATH variable is also important when you run Java applets, not just when you compile them. It tells the web browser or applet viewer where it should look to find the referenced .class files. If the CLASSPATH is set improperly, you'll probably see messages like "Applet could not start."
- If the CLASSPATH environment variable has not been set, and you do not specify one on the command line, then Java sets the CLASSPATH to the default:

  - Unix: .:$JAVA/classes:$JAVA/lib/classes.zip
  - Windows: .:$JAVA\classes:$JAVA\lib\classes.zip
  - Mac: ./$JAVA:classes/$JAVA:lib:classes.zip

- Here . is the current directory and $JAVA is the main Java directory where the different tools like javac were installed.

String

- In Java a string is a sequence of characters as usual. But, unlike many other languages that implement strings as character arrays, Java implements string as objects of type String.
- Implementing string as built-in objects allows Java to provide a full complement of features that make string handling convenient.
- For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.
- Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed.
- When we create a String object, we are creating a string that cannot be changed.
- Even though the characters contained cannot be changed, we can still perform all types of string operations.
- The difference is that each time you need an altered version of an existing string, a new String approach is used because fixed, immutable strings can be implemented more efficiently that changeable ones.
- It is defined in java.lang. Thus, there are available to all class automatically.

Similarities between String and StringBuffer

- Both the String and String Buffer are the classes
- Both are defined in java.lang.
- Both are declared final, which means that neither of these classes may be sub-classed.

String Constructors

```
String s=new String( )
```

String s=new String(char_array[ ])
String(char char_array[ ], int startIndex, int numChars)
String(String strObj)
String(byte asciiChars[ ])
String(byte asciiChars[ ], int startIndex, int numChars)

String Methods

length( )
toString( )

- By overriding toString( ) for classes that we create, we allow them to be fully integrated into Java's programming environment.
- If you have created toString( ) method in any class, and you call the object without specifying any method of the class then toString( ) method is automatically invoked.
  Example
  Box b=new Box(10);
  System.out.println(b); //This line will print the string which will be returned by toString( ) method of class b automatically.

Character Extraction

charAt(int where)
getChars(int sourceStart, int sourceEnd, char targer[ ], int targetStart)
Example

```
Class getCharsDemo
{
        Public static void main(String args[ ])
        {
                String s="This is a demo of the getChars method.";
                Int start=10;
                Int end=14;
                Char buf[ ]=new char[end-start];

                s.getChars(start,end,buf,0);
                System.out.println(buf);
        }
}
```

getBytes( )

- There is an alternative to get Chars( ) that stores the characters in an array of bytes.
- It uses the default character to byte conversion provided by the platform.
  byte[ ] getBytes( )

toCharArray( )

String comparison

equals( )

- To compare two strings for equality.
  boolean equals(Object str)
  equalsIgnoreCase(String str)

Comparison between equals( ) and ==

- The equals( ) method compares the characters inside a String object.

- The == operator compares two object references to see whether they refer to the same instance.

  Example

```
Class EqualsNotEqualTo
{
    String s1=”Hello”;
    String s1=new String(s1);

    System.out.println(s1+”equals”+s2+”->”+s1.equals(s2));
    System.out.println(s1+”==”+s2+”->”+s1==s2);
}
```

Int indexOf(int ch)
Int lastIndexOf(int ch)
Int indexOf(String str)
Int lastIndexOf(String str)
String substring(int startIndex)
String substring(int startIndex, int endIndex)
String Concat(String str)
String replace(char original, char replacement)
String str.trim( )

StringBuffer

- It is a peer class of String that provides much of the functionality of strings.
- In contrast to String StringBuffer represents grow able and writeable character sequences.
- It may have characters and substrings inserted in the middle or appended to the end.
- It will automatically grow to make room for such additions and often has more characters per allocated than are actually needed, to allow room for growth.

StringBuffer Constructors

StringBuffer( )
StringBuffer(int size)
StringBuffer(String str)

Converting Strings to Numbers

When processing user input it is often necessary to convert a String that the user enters into an int. The syntax is straightforward. It requires using the static Integer.valueOf(String s) and intValue() methods from the java.lang.Integer class. To convert the String "22" into the int 22 you would write

```
int i = Integer.valueOf("22").intValue();
```

Doubles, floats and longs are converted similarly. To convert a String like "22" into the long value 22 you would write

```
long l = Long.valueOf("22").longValue();
```

To convert "22.5" into a float or a double you would write:

```
double x = Double.valueOf("22.5").doubleValue();
float y = Float.valueOf("22.5").floatValue();
```

The various valueOf() methods are relatively intelligent and can handle plus and minus signs, exponents, and most other common number formats. However if you pass one something completely non-numeric like "pretty in pink," it will throw a NumberFormatException. You haven't learned how to handle exceptions yet, so try to avoid passing theses methods non-numeric data.

You can now rewrite the E = mc2 program to accept the mass in kilograms as user input from the command line. Many of the exercises will be similar.

```
class Energy {
  public static void main (String args[]) {

    double c = 2.998E8;  // meters/second
    double mass = Double.valueOf(args[0]).doubleValue();
    double E = mass * c * c;
    System.out.println(E + " Joules");
  }
}
```

Here's the output:

```
$ javac Energy.java
$ java Energy 0.0456
4.09853e+15 Joules
```

Testing for Equality with equals()

That's not what you expected. To compare strings or any other kind of object you need to use the equals(Object o) method from java.lang.String. Below is a corrected version that works as expected. The reasons for this odd behavior go fairly deep into Java and the nature of object data types like strings.

```
class JackAndJill {

  public static void main(String args[]) {

    String s1 = new String("Jack went up the hill.");
    String s2 = new String("Jack went up the hill.");

    if ( s1.equals(s2) ) {
      System.out.println("The strings are the same.");
    }
    else {
      System.out.println("The strings are not the same.");
    }
  }
}
```

Vector

- An **array** is a collective type of variable which stores data in sequential memory blocks. Index is used to access any value in array
- ArrayList is class that extends AbstractList and implements the List interface which s
- **Vector** implements a dynamic array which is similar to ArrayList.
- **Vector constructors:**
  - **Vector( )**
  - **Vector(int size)**
  - **Vector(int size, int incr)**
  - **Vector(collection c)**
- **Vector ( ):** Default vector which has an initial size of 10.
- **Vector (int size):** Vector whose initial capacity is specified by size.
- **Vector (int size, int incr):** vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward.
- **Vector (Collection c):** Vector that contains the elements of collection c.
- When an object is to store on the vector and there is no space or have less space to store those objects then these no. of elements is automatically incremented.
- By default the vector size is incremented by doubled by each allocation cycle.
- **Vector's protected data members**
  - **int capicityIncrement;**
  - **int elementCount;**
  - **Object elementData[ ];**
- **capacityIncrement**: The increment value is stored in it.
- **elementCount**: The no. of elements currently in the vector is stored in it.
- **elementData**: the array that holds the vector stored in it
- **Methods**
  - **Void addElement(Object element)**
    - The object specified by the element is added to vector
  - **Int capacity( )**
    - The maximum number of elements that the vector can hold
  - **Boolean contains(Object element)**
    - Return true if element is contained by vector
  - **void copyInto(Object array[ ])**
    - Copy the content to an array
  - **Object elementAt(int index)**
  - **Object firstElement( )**
  - **int indexOf(Object element)**
  - **Boolean isEmpty( )**
  - **Object LastElement( )**
  - **int LastIndexOf(object element)**
  - **void removeAllEments( )**
  - **void removeElementAt(int index)**
  - **void setElementAt(Object element, int index)**
  - **int size( )**
  - **String toString( )**

Difference between vector and Array

Vector	**Array**
1. It is dynamic	1. It is fixed in size

2. It is a class and we can create its object instances	2. It is a collective data type.
3. It can store multiple type of data in single vector	3. It can store only on data type in single Array

Two differences between Vector class and ArrayList

1. Vector is synchronized and it contains many legacy methods that are not part of the collection framework.
2. In Java 2 Vector is fully compatible to collection as it extends AbstractList and implement the List interface.

Difference between class, Abstract Class and Interface

Concrete Class	**Abstract Class**	**Interface**
• Specify the full set of methods for an object	• Specify the full set of methods for an object	• Specifies a subset of methods for an object
• Implements all of its methods	• Implements none, some, or all of its methods	• Implements none of its methods
• Can have instances	• Cannot have instances	• Cannot have instances
• Can have subclasses	• Must have subclasses; otherwise useless	• Cannot have subclasses; must have classes that implement it; useless otherwise

Multithreading

Introduction to Multithreading

- A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a thread, and each thread defines a separate path of execution.
- Multithreading is a specialized form of multitasking

What are the differences between Processes based multitasking and thread based multitasking?

Process based Multitasking	**Thread based Multitasking**
• A process is, in essence, a program that is executed. Thus process based multitasking is the feature that allows our computer to run two or more programs concurrently.	• A single program can perform two or more tasks simultaneously.
• For example, process based multitasking enables us to run the Java compiler at the same time that we are using a text editor.	• For example, a text editor can format text at the same time that it is checking the spelling.
• A program is the smallest unit of code that can be dispatched by the scheduler	• The thread is the smallest unit of dispatch able code.
• It deals with the "big picture"	• It handles the details within the program

• It requires more overheads than thread based Multitasking	• It requires less overhead than process based Multitasking
• It is heavily weighted tasks that require their own separate address spaces	• It is light weight and shares the same address space and cooperatively share the same heavyweight process.
• Inter process communication is expensive and limited	• Inter thread communication is inexpensive, and context switching from one thread to the next is low cost.
• It is not under the control of Java	• It is under the control of Java
• It relatively west more CPU time.	• It enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
• It have to finish the current task to start another task with in a program	• It can handle multiple tasks concurrently without requiring finishing the current tasks.

What is the difference between multithreading and multitasking?

Multitasking	**Multithreading**
• Multitasking is a broader term than multithreading. Multitasking provides multiple tasks to be done concurrently.	• Specialized form of multitasking. Multithreading is program that contains two or more parts that can run concurrently. Each part of such a program is called thread.

The Java Thread Model

- Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.
- Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.
- ***Single-thread*** systems use an approach called an ***event loop*** with ***polling***.
- In this model a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.
- Once this polling mechanism returns with, say a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler.
- Until this event handler returns, nothing else can happen in the system.
- This wastes CPU time. It is also in one part of a program dominating the system and preventing any other events from being processed.
- In general, in a singled-threaded environment, when a thread blocks because it is waiting for some resource, the entire program stops running.
- The **benefit** of Java's **multithreading** is that the main loop/polling mechanism is eliminated.
- One thread can pause without stopping other parts of your program.
- For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere.
- Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.

- When a thread blocks in a Java program only the single thread that is blocked pauses.
- All other threads continue to run.
- Threads exist in several states.
    - A thread can be running.
    - A thread can be ready to run as soon as it gets CPU time.
    - A thread can be suspended, which temporarily suspends its activity.
    - A suspended thread can then be resumed, allowing it to pick up where it left off.
    - A thread can be blocked when waiting for a resource.
    - At any time, a thread can be terminated, which halts its execution immediately.

Characteristics of Thread

Thread in java have five Characteristics, they are:

- Thread body
- Thread state
- Thread priority
- Daemon Threads
- Threads group.

Thread Body

This is the sequence of instructions for thread to perform. This is defined in ( ) method. There are two ways to supply a run method to a thread.

1. Extending a thread class and overriding the run( ) method,
2. Creating a thread by implementing the Runable interface.

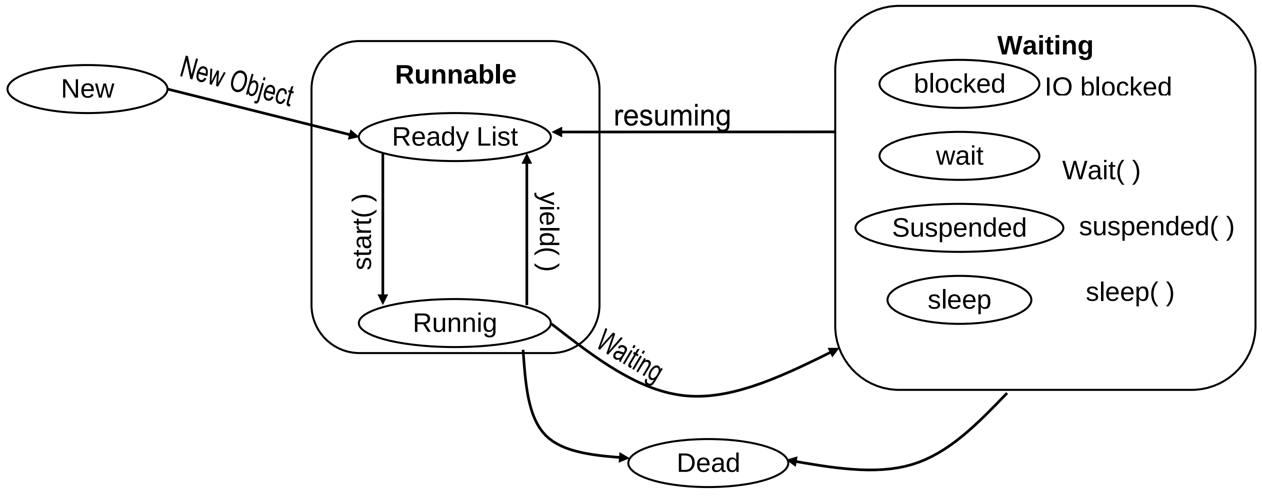

Fig: Thread states

Thread States

Every thread, after creation and before destruction, will always be in one of the six states. They are:

- New
- Runnable

    - Ready
    - Running
- Waiting
    - Blocked
    - Sleep
    - Suspended
    - Wait
- Dead

New

- A enters the newly created by using a new operator.
- It is in new state or born state immediately after creation; that is when a constructor is called the Thread is created but is not yet to **run** ( ) method will not begin until its **start** ( ) method is called.
- After the **start** ( ) method is called, the thread will go to the next state, Running state.

Runnable

Ready

- Once the **start** ( ) method is invoked, the new thread is appended to Queue (Ready List).
- When it is in queued state, it is waiting in the queue and competing for its turn to spend CPU cycles.
- It is controlled by Virtual Machine Controller.

Running

- When the thread is running state, it is assigned by CPU cycles and is actually running.
- When we use the **yield** ( ) method it makes sure other threads of the same priority have chance to run.
- This method causes voluntary move itself to the queued state from the running state.

Waiting states

- The waiting state is entered when one of the following events occurs:
    - The thread itself or another thread calls the **wait** ( ) method.
    - The thread itself calls the **sleep**( ) method
    - The thread is waiting for some IO operation to complete.
    - The thread will **join** ( ) another thread.
    - The thread itself or another thread calls the **suspend** ( ) method.
- The thread in a block state will not be scheduled for running.
- It will go back to the Ready Queue when its cause of block is completed.
    - If the thread is blocked by calling **wait** ( ) method, the object's **notify** ( ) and **noifyAll** ( ) method is called.
    - If a thread has been put to sleep, the specified number of milliseconds must expire.
    - If the thread is blocked on I/O, the specified I/O operation is completed.
    - If the thread is suspended, another thread calls its **resume** ( ) method.

Dead

- A thread is dead for any one of the two reasons:
    - It dies a natural death because the run method exits normally.
    - It dies an abnormally because uncaught exception terminates the run method.
- In particular the **stop** ( ) method is used to kill the thread.
- We can use interruption for terminating the thread.

- To find whether a thread is alive that is currently in runnable or Blocked state, use the threads **isAlive** ( ) method, if it returns true the thread is alive.

Thread Priority

- Every thread in java is assigned a priority value, when more than one thread is competing for CPU time; the thread with highest priority value is given preference.
- You can also use the Thread class constants.
-

Constants	**Integer value**
Thread.MIN_PRIORITY	1
Thread.MAX_PRIORITY	10
Thread_NORM_PRIORITY	5

- The **setPriority** ( ) method is used to set the priority value of a thread
- The **getPriority** ( ) method is used to get the priority of the thread.
- The default priority is Thread.NORM_PRIORITY.

Daemon Thread

- This denotes that a thread is a "server" thread.
- A server thread is a thread that services client request.
- They normally enter an endless loop waiting for clients requesting services.
- To create a daemon Thread, call the **setDaemon** ( ) method after the thread creation and before the execution is started.
- The syntax is

  setDaemon (boolean b)
    - If b is true, the thread is marked as daemon thread. Otherwise, it is a non-daemon thread.
- **isDaemon** ( ), this method returns true if this thread is a daemon else return false.

Thread Group

- For large programs that generate many things, java allows us to group similar threads and manage them as a group.
- Every thread instance is a member of exactly one thread group.
- When a program is started, the java virtual machine creates the main thread group as a member of the system thread group.
- A new thread group is created by instantiating the ThreadGroup class.
- The **getThreadGroup** ( ) returns the parent thread group of the thread.
    - **getParent**( ) returns the parent Thread group of the thread group
    - **getName** ( ) returns the name of the thread group.
- The thread class defines several methods that help us to manage threads.

Method	Description
Getname( )	Obtain the threads name
setName( )	Set the name of a thread
isAlive( )	Determine if the thread is still running
join( )	Wait for thread to terminate
sleep( )	Suspend a thread for specified time
Start	Start a thread by calling its run Method
wait( )	This is used to make a thread wait.

notify( )	This is used to wake a thread that is waiting
notifyall( )	This is used to wake all the threads that are waiting

Why to use Runnable interface.

Using the Runnable interface is a bit more complicated but it has two advantages:

- The thread's run( ) method has access to the private variables of the class in which it is located
- Since Java only has single inheritance, you may not be able to subclass Thread if you also want your class to subclass something else. For example, you cannot create an applet which is a subclass of Thread because your applet is already a subclass of the Applet class.