Java Programming – 18BCS43C

Dr. S. Chitra,
Associate Professor,
Post Graduate & Research Department of Computer Science,
Government Arts College(Autonomous), Coimbatore - 641018

Year	Subject Title	Sem.	Sub Code
2018 -19 Onwards	JAVA PROGRAMMING	IV	18BCS43C

UNIT-III

Interfaces, Packages and Thread: Defining Interface- Extending Interfaces Implementing Interfaces – Packages-Multithreaded Programming: Thread Life Cycle - Thread Exceptions – Thread Priority-Synchronization.

Interfaces: Multiple Inheritance

, classes in Java cannot have more than one superclass.

```
class A extends B extends C
{
   ..........
   ..........
}
```

is not permitted in Java. However, the designers of Java could not overlook the importance of multiple inheritance. A large number of real-life applications require the use of multiple inheritance whereby we inherit methods and properties from several, distinct classes. Since C++ like implementation of multiple inheritance proves difficult and adds complexity to the language, Java provides an alternate approach known as *interfaces* to support the concept of multiple inheritance. Although a Java class cannot be a subclass of more than one superclass, it can *implement* more than one interface, thereby enabling us to create classes that build upon other classes without the problems created by multiple inheritance.

Defining Interfaces

An interface is basically a kind of class. Like classes, interfaces contain methods and variables but with a major difference. The difference is that interfaces define only abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constants. Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods.

The syntax for defining an interface is very similar to that for defining a class. The general form of an interface definition is:

```
interface InterfaceName
{
    variables declaration;
    methods declaration;
}
```

Here, **interface** is the key word and *InterfaceName* is any valid Java variable (just like class names). Variables are declared as follows:

```
static final type VariableName = Value;
```

Note that all variables are declared as constants. Methods declaration will contain only a list of methods without any body statements. Example:

```
return-type methodName1 (parameter_list);
```

Here is an example of an interface definition that contains two variables and one method:

```
interface Item
{
    static final int code = 1001;
    static final String name = "Fan";
    void display ( ) ;
}
```

Note that the code for the method is not included in the interface and the method declaration simply ends with a semicolon. The class that implements this interface must define the code for the method.

Another example of an interface is:

```
interface Area
{
    final static float pi = 3.142F;
    float compute (float x, float y);
    void show ( );
}
```

Extending Interfaces

Like classes, interfaces can also be extended. That is, an interface can be subinterfaced from other interfaces. The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses. This is achieved using the keyword **extends** as shown below:

```
interface name2 extends name1
{
    body of name2
}
```

We can also combine several interfaces together into a single interface. Following declarations are valid:

```
interface ItemConstants
{
      int code = 1001;
      String name = "Fan";
}
interface ItemMethods
{
      void display( );
}
interface Item extends ItemConstants, ItemMethods
{
      ..........
      ..........
}
```

While interfaces are allowed to extend to other interfaces, subinterfaces cannot define the methods declared in the superinterfaces. After all, subinterfaces are still interfaces, not classes. Instead, it is the responsibility of any class that implements the derived interface to define all the methods. Note that when an interface extends two or more interfaces, they are separated by commas.

It is important two remember that an interface cannot extend classes. This would violate the rule that an interface can have only abstract methods and constants.

Implementing Interfaces

Interfaces are used as "superclasses" whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:

```
class classname implements interfacename
{
    body of classname
}
```

Here the class *classname* "implements" the interface *interfacename*. A more general form of implementation may look like this:

```
class classname extends superclass
      implements interface1, interface2, ......
{
      body of classname
}
```

This shows that a class can extend another class while implementing interfaces.

Implementing interfaces

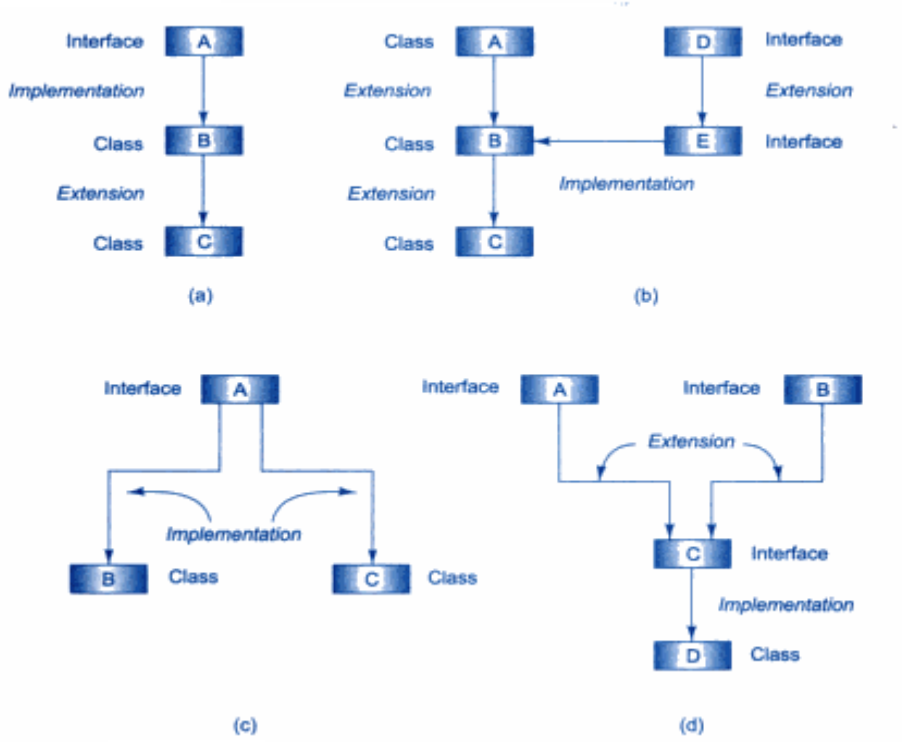
```
// InterfaceTest.java
interface Area // Interface defined
{
    final static float pi = 3.14F;
    float compute (float x, float y);
}
class Rectangle implements Area                    // Interface implemented
{
    public float compute (float x, float y)
    {
        return (x*y);
    }
}
class Circle implements Area                       // Another implementation
{
    public float compute (float x, float y)
    {
        return (pi*x*x);
    }
}
class InterfaceTest
{
    public static void main(String args[ ])
    {
        Rectangle rect = new Rectangle( );
        Circle cir = new Circle( );
        Area area;                                 // Interface object
        area = rect;                               // area refers to rect object
        System.out.println("Area of Rectangle = "
                                                   + area.compute(10, 20));
        area = cir;                                    // area refers to cir object
        System.out.println("Area of Circle = "
                                                   + area.compute(10, 0));
    }
```

ıe Output is as follows:

```
Area of Rectangle = 200
Area of Circle = 314
```

Any number of dissimilar classes can implement an interface. However, to implement the methods, we need to refer to the class objects as types of the interface rather than types of their respective classes. Note that if a class that implements an interface does not implement all the methods of the interface, then the class becomes an ***abstract*** class and cannot be instantiated.

Various forms of interface implementation



Implementing multiple inheritance

```
class Student
{
    int rollNumber;
    void getNumber(int n)
    {
        rollNumber = n;
    }
    void putNumber( )
    {
        System.out.println(" Roll No : " + rollNumber);
    }
}
class Test extends Student
{
    float part1, part2;
    void getMarks(float m1, float m2)
    {
        part1 = m1;
        part2 = m2;
    }
    void putMarks( )
    {
        System.out.println("Marks obtained ");
        System.out.println("part 1 = " + part1);
        System.out.println("Part2 = " + part2);
    }
}
interface Sports
{
    float sportWt = 6.0F;
    void putwt( );
}
class Results extends Test implements Sports
{
    float total;
    public void putWt( )
    {
        System.out.println("Sports Wt = " + sportWt);
    }
    void display( )
    {
        total = part1 + part2 + sportWt;
        putNumber( );
        putMarks( );
        putWt( );
        System.out.println("Total score = " + total);
    }
}
class Hybrid
{
    public static void main(String args[ ])
    {
        Results student1 = new Results( );
        student1.getNumber(1234);
        student1.getMarks(27.5F, 33.0F);
        student1.display( );
    }
}
```

Output of the Program

```
  Roll No : 1234
Marks obtained
Part1 = 27.5
Part2 = 33
Sports Wt = 6
Total score = 66.5
```

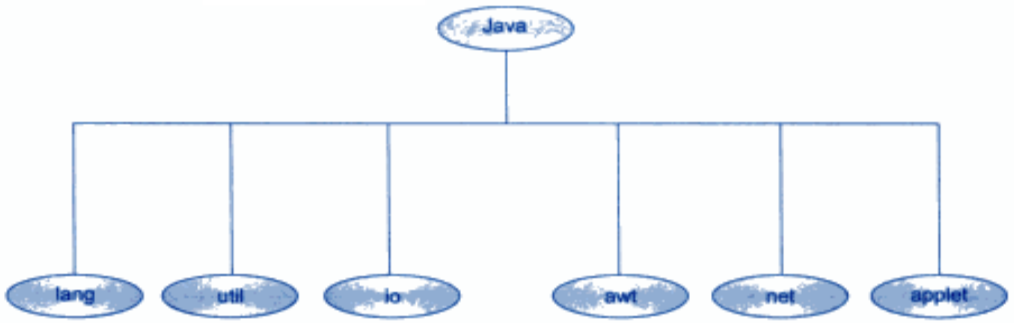
Packages: Putting Classes Together

Packages are Java's way of grouping a variety of classes and/or interfaces together. The grouping is usually done according to functionality. In fact, packages act as "containers" for classes. By organizing our classes into packages we achieve the following benefits:

1. The classes contained in the packages of other programs can be easily reused.
2. In packages, classes can be unique compared with classes in other packages. That is, two classes in two different packages can have the same name. They may be referred by their fully qualified name, comprising the package name and the class name.
3. Packages provide a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal use only.
4. Packages also provide a way for separating "design" from "coding". First we can design classes and decide their relationships, and then we can implement the Java code needed for the methods. It is possible to change the implementation of any method without affecting the rest of the design.

Java API Packages

Java API provides a large number of classes grouped into different packages according to functionality. Most of the time we use the packages available with the Java API. Figure shows the functional breakdown of packages that are frequently used in the programs. Table shows the classes that belong to each package



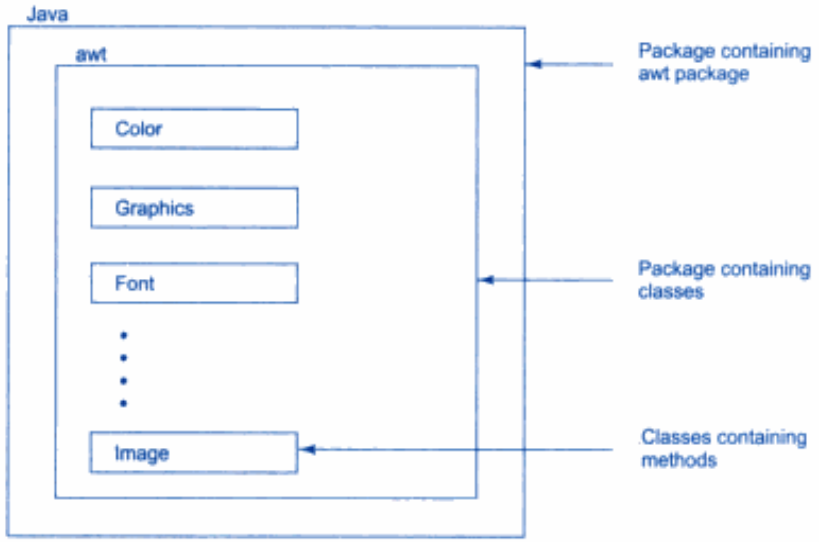
Frequently used API packages

Java System Packages and Their Classes

Package name	*Contents*
`java.lang`	Language support classes. These are classes that Java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions.
`java.util`	Language utility classes such as vectors, hash tables, random numbers, date, etc.
`java.io`	Input/output support classes. They provide facilities for the input and output of data.
`java.awt`	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.
`java.net`	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
`java.applet`	Classes for creating and implementing applets.

Using System Packages

The packages are organised in a hierarchical structure as illustrated in Fig. This shows that the package named **java** contains the package **awt**, which in turn contains various classes required for implementing graphical user interface.



Hierarchical representation of java.awt package

There are two ways of accessing the classes stored in a package. The first approach is to use the *fully qualified class name* of the class that we want to use. This is done by using the package name containing the class and then appending the class name to it using the dot operator. For example, if we want to refer to the class **Color** in the **awt** package, then we may do so as follows:

```
java.awt.Colour
```

But, in many situations, we might want to use a class in a number of places in the program or we may like to use many of the classes contained in a package. We may achieve this easily as follows:

```
import packagename.classname;
or
import packagename.*;
```

These are known as *import statements* and must appear at the top of the file, before any class declarations, **import** is a keyword.

Creating Packages

1. Declare the package at the beginning of a file using the form

```
package packagename;
```

2. Define the class that is to be put in the package and declare it **public**.
3. Create a subdirectory under the directory where the main source files are stored.
4. Store the listing as the classname.java file in the subdirectory created.
5. Compile the file. This creates **.class** file in the subdirectory.

```
package firstPackage;                    // package declaration
```

A java package file can have more than one class definitions. In such cases, only one of the classes may be declared **public** and that class name with **.java** extension is the source file name. When a source file with more than one class definition is compiled, Java creates independent **.class** files for those classes.

```
. . . . . .
}
```

Here the package name is **firstPackage**. The class **FirstClass** is now considered a part of this package. This listing would be saved as a file called **FirstClass.java**, and located in a directory named **firstPackage**. When the source file is compiled, Java will create a **.class** file and store it in the same directory.

Remember that the **.class** files must be located in a directory that has the same name as the package, and this directory should be a subdirectory of the directory where classes that will import the package are located.

A java package file can have more than one class definitions. In such cases, only one of the classes may be declared **public** and that class name with **.java** extension is the source file name. When a source file with more than one class definition is compiled, Java creates independent **.class** files for those classes.

Using a Package

Let us now consider some simple programs that will use classes from other packages. The listing below shows a package named **package1** containing a single class **ClassA**.

```
package package1;
public class ClassA
{
    public void displayA( )
    {
        System.out.println("Class A");
    }
}
```

This source file should be named **ClassA.java** and stored in the subdirectory **package1** as stated earlier. Now compile this java file. The resultant **ClassA.class** will be stored in the same subdirectory.

This source file should be named **ClassA.java** and stored in the subdirectory **package1** as stated earlier. Now compile this java file. The resultant **ClassA.class** will be stored in the same subdirectory.

```
import package1. ClassA;
class PackageTest1
{
    public static void main(String args[ ] )
    {
        ClassA objectA = new ClassA( ) ;
        objectA.displayA( );
    }
}
```

This listing shows a simple program that imports the class **ClassA** from the package **package1**. The source file should be saved as **PackageTest1.java** and then compiled. The source file and the compiled file would be saved in the directory of which **package1** was a subdirectory. Now we can run the program and obtain the results.

During the compilation of **PackageTest1.java** the compiler checks for the file **ClassA.class** in the **package1** directory for information it needs, but it does not actually include the code from **ClassA.class** in the file **PackageTest1.class**. When the **PackageTest1** program is run, Java looks for the file **PackageTest1.class** and loads it using something called *class loader*. Now the interpreter knows that it also needs the code in the file **ClassA.class** and loads it as well.

```
import package1. ClassA;
class PackageTest1
{
    public static void main(String args[ ] )
    {
        ClassA objectA = new ClassA( ) ;
        objectA.displayA( );
    }
}
```

This listing shows a simple program that imports the class **ClassA** from the package **package1**. The source file should be saved as **PackageTest1.java** and then compiled. The source file and the compiled file would be saved in the directory of which **package1** was a subdirectory. Now we can run the program and obtain the results.

During the compilation of **PackageTest1.java** the compiler checks for the file **ClassA.class** in the **package1** directory for information it needs, but it does not actually include the code from **ClassA.class** in the file **PackageTest1.class**. When the **PackageTest1** program is run, Java looks for the file **PackageTest1.class** and loads it using something called *class loader*. Now the interpreter knows that it also needs the code in the file **ClassA.class** and loads it as well.

Access Protection

Access location \ Access modifier →	*public*	*protected*	*friendly (default)*	*private protected*	*private*
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

Adding a Class to a Package

It is simple to add a class to an existing package. Consider the following package:

```
package p1;
public ClassA
{
    // body of A
}
```

The package **p1** contains one public class by name **A**. Suppose we want to add another class **B** to this package. This can be done as follows:

1. Define the class and make it public.
2. Place the package statement

   ```
   package p1;
   ```

   before the class definition as follows:

   ```
   package p1;
   public class B
   {
       // body of B
   }
   ```

3. Store this as **B.java** file under the directory **p1.**
4. Compile **B.java** file. This will create a **B.class** file and place it in the directory **p1.**

Static Import

Static import is another language feature introduced with the J2SE 5.0 release. This feature eliminates the need of qualifying a static member with the class name. The static import declaration is similar to that of import. We can use the import statement to import classes from packages and use them without qualifying the package. Similarly, we can use the static import statement to import static members from classes and use them without qualifying the class name. The syntax for using the static import feature is:

```
import static package-name.subpackage-name.class-
name.staticmember-name;
(or)
import static package-name.subpackage-name.class-name.*;
```

Before introducing the static import feature, we had to use the static member with the qualifying class name. For example, consider the following code that contains the static member PI:

```
double area_of_circle = Math.PI * radius * radius;
```

In the above code, PI is the static member of the class, **Math**. So the static member PI is used in the above program with the qualified class name called **Math**.

Use of Static import

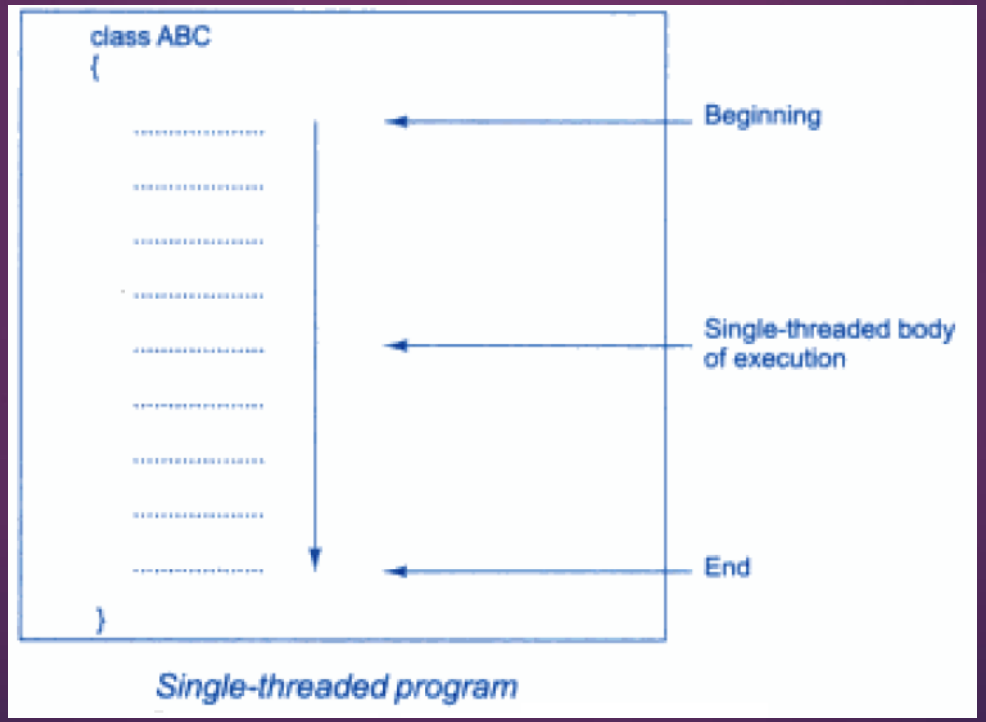
```
import static java.lang.Math.*;
public class mathop
{
      public void circle(double r)
           {
                double area=PI*r* r;
                System.out.println("The Area of Circle is :"+area);
           }
                public static void main(String args[])
                {
                  mathop obj=new mathop();
                  obj.circle(2.3);
                }
}
```

The output for the above programs is:

The Area of Circle is 16.619025137490002

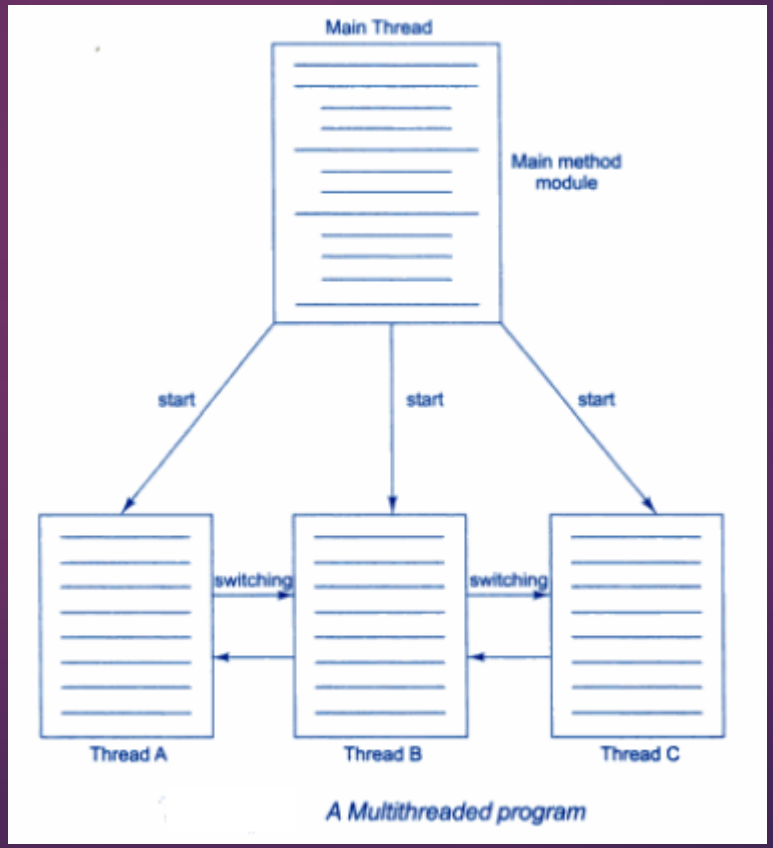
Multithreaded Programming

A *thread* is similar to a program that has a single flow of control. It has a beginning, a body, and an end, and executes commands sequentially. In fact, all main programs in our earlier examples can be called *single-threaded* programs.



Single-threaded program

A unique property of Java is its support for multithreading. That is, Java enables us to use multiple flows of control in developing programs. Each flow of control may be thought of as a separate tiny program (or module) known as a *thread* that runs in parallel to others as shown in Fig. A program that contains multiple flows of control is known as *multithreaded program*.



A Multithreaded program

Once initiated by the main thread, the threads A, B, and C run concurrently and share the resources jointly. It is like people living in joint families and sharing certain resources among all of them. The ability of a language to support multithreads is referred to as *concurrency*. Since threads in Java are subprograms of a main application program and share the same memory space, they are known as *lightweight threads* or *lightweight processes*.

It is important to remember that 'threads running in parallel' does not really mean that they actually run at the same time. Since all the threads are running on a single processor, the flow of execution is shared between the threads. The Java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently.

Creating Threads

Creating threads in Java is simple. Threads are implemented in the form of objects that contain a method called **run( )**. The **run( )** method is the heart and soul of any thread. It makes up the entire body of a thread and is the only method in which the thread's behaviour can be implemented. A typical **run( )** would appear as follows:

```
public void run( )
{
     .....
     ..... (statements for implementing thread)
     .....
}
```

The **run( )** method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called **start ( ).**

A new thread can be created in two ways.

1. ***By creating a thread class:*** Define a class that extends **Thread** class and override its **run( )** method with the code required by the thread.
2. ***By converting a class to a thread:*** Define a class that implements **Runnable** interface. The **Runnable** interface has only one method, **run( )**, that is to be defined in the method with the code to be executed by the thread.

Extending the Thread Class

1. Declare the class as extending the **Thread** class.
2. Implement the **run( )** method that is responsible for executing the sequence of code that the thread will execute.
3. Create a thread object and call the **start( )** method to initiate the thread execution.

Declaring the Class

The **Thread** class can be extended as follows:

```
class MyThread extends Thread
{
    . . . . . .
    . . . . . .
    . . . . . .
}
```

Now we have a new type of thread **MyThread.**

Implementing the *run()* Method

The **run( )** method has been inherited by the class **MyThread.** We have to override this method in order to implement the code to be executed by our thread. The basic implementation of **run( )** will look like this:

```
public void run( )
{
     . . . . . .
     . . . . . .    // Thread code here
     . . . . . .
}
```

When we start the new thread, Java calls the thread's **run( )** method, so it is the **run( )** where all the action takes place.

Starting New Thread

To actually create and run an instance of our thread class, we must write the following:

```
MyThread aThread = new MyThread( );
aThread.start( );                    // invokes run() method
```

The first line instantiates a new object of class **MyThread.** Note that this statement just creates the object. The thread that will run this object is not yet running. The thread is in a *newborn* state.

The second line calls the **start( )** method causing the thread to move into the *runnable* state. Then, the Java runtime will schedule the thread to run by invoking its **run( )** method. Now, the thread is said to be in the *running* state.

An Example of Using the Thread Class

```
class A extends Thread
{
     public void run( )
     {
          for (int i=1; i<=5; i++)
          {
                    System.out.println("\tFrom ThreadA : i = " + i);
          }
          System.out.println("Exit form A ");
     }
}
class B extends Thread
{
   public void run( )
   {
        for(int j=1; j<=5; j++)
        {
                 System.out.println("\tFrom Thread B :j = " + j);
        }
        System.out.println("Exit from B ");
   }
}
class B extends Thread
{
     public void run( )
     {
          for(int k=1; k<=5; k++)
          {
              System.out.println("\tFrom Thread C : k = " + k);
          }
          System.out.println("Exit from C ");
     }
}
class ThreadTest
{
     public static void main(String args[ ])
     {
          new A( ).start( );
          new B( ).start( );
          new C( ).start( );
     }
}
```

Output of Program would be:

First run

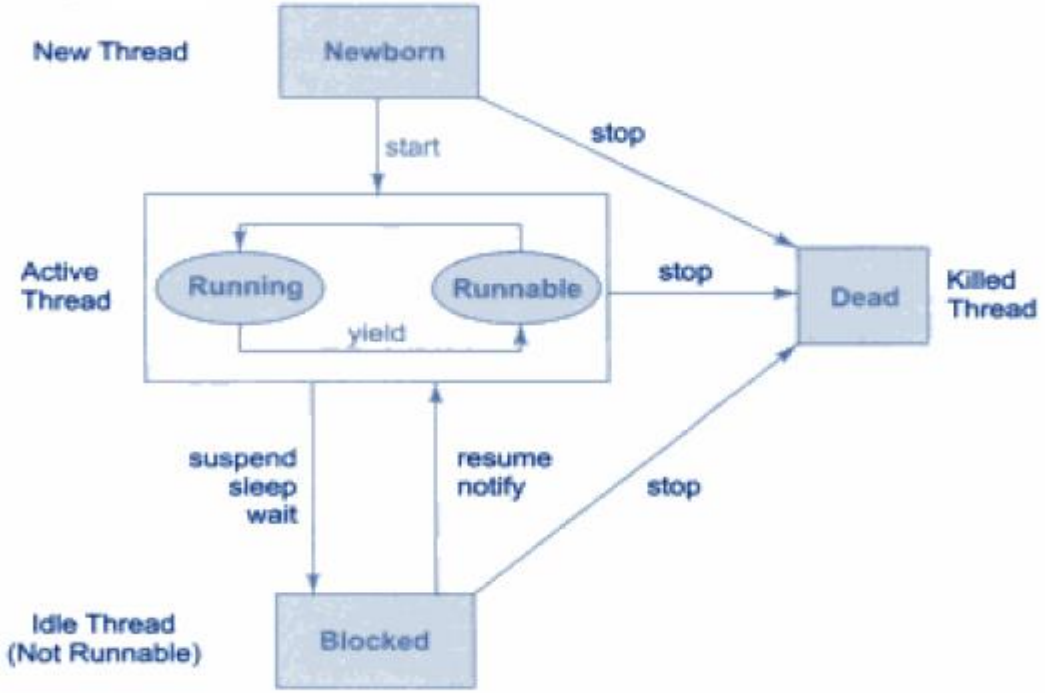
```
From Thread A : i = 1
From Thread A : i = 2
From Thread B : j = 1
From Thread B : j = 2
From Thread C : k = 1
From Thread C : k = 2
From Thread A : i = 3
From Thread A : i = 4
From Thread B : j = 3
From Thread B : j = 4
From Thread C : k = 3
From Thread C : k = 4
```

Life Cycle of a Thread

During the life time of a thread, there are many states it can enter. They include:

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in Fig.



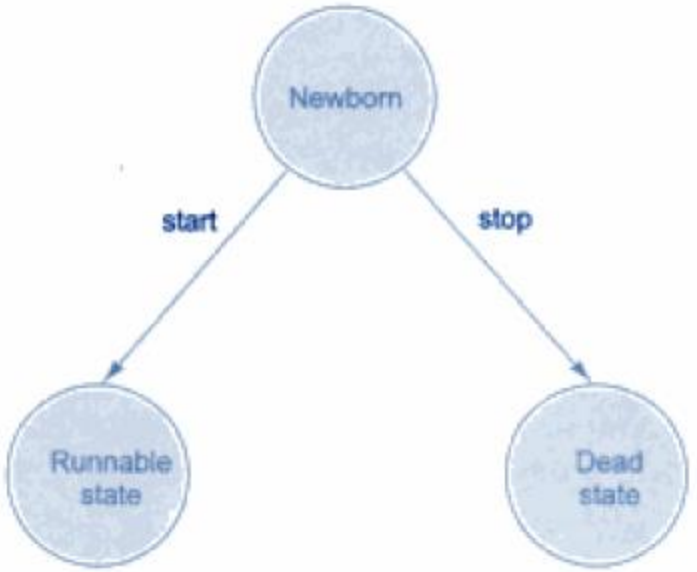
State transition diagram of a thread

Newborn State

When we create a thread object, the thread is born and is said to be in *newborn* state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it:

- Schedule it for running using **start( )** method.
- Kill it using **stop( )** method.

If scheduled, it moves to the runnable state If we attempt to use any other method at this stage, an exception will be thrown.

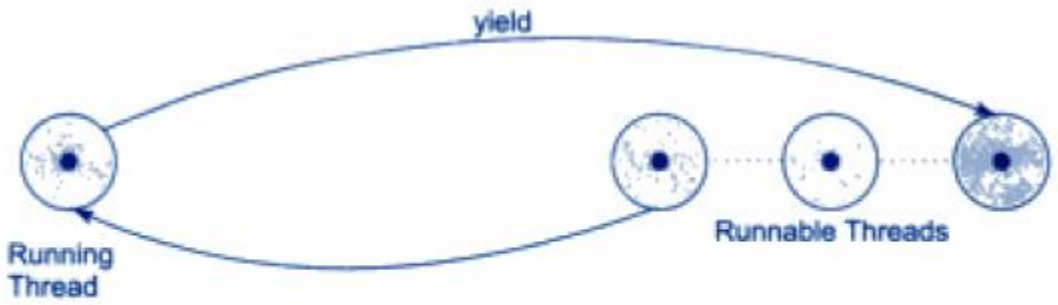


Scheduling a newborn thread

Runnable State

The *runnable* state means that the thread is ready for execution and is waiting for the availability of the processor. That is, the thread has joined the queue of threads that are waiting for execution. If all threads have equal priority, then they are given time slots for execution in round robin fashion, i.e., first-come, first-serve manner. The thread that relinquishes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as *time-slicing*.

However, if we want a thread to relinquish control to another thread to equal priority before its turn comes, we can do so by using the **yield( )** method

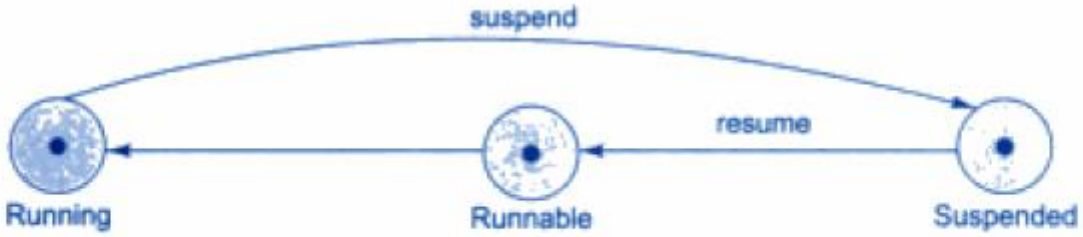


Relinquishing control using yield( ) method

Running State

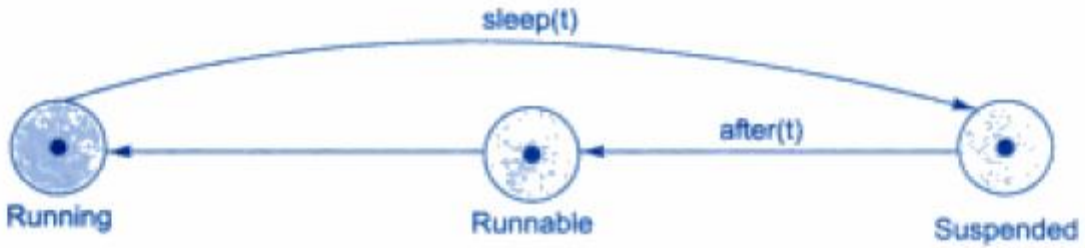
Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread. A running thread may relinquish its control in one of the following situations.

1. It has been suspended using **suspend( )** method. A suspended thread can be revived by using the **resume( )** method. This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.



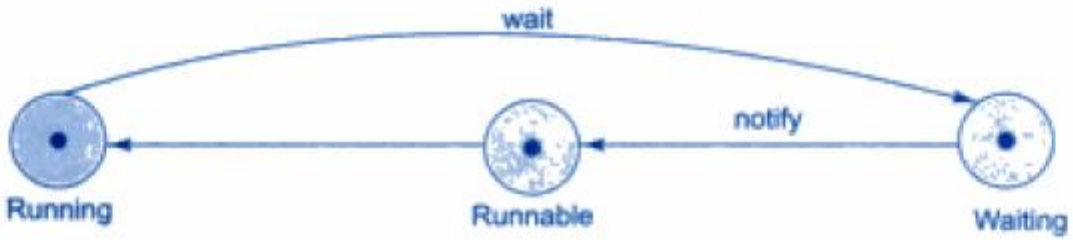
Relinquishing control using suspend( ) method

2. It has been made to sleep. We can put a thread to sleep for a specified time period using the method **sleep**(*time*) where *time* is in milliseconds. This means that the thread is out of the queue during this time period. The thread re-enters the runnable state as soon as this time period is elapsed.



Relinquishing control using sleep( ) method

3. It has been told to wait until some event occurs. This is done using the **wait( )** method. The thread can be scheduled to run again using the **notify( )** method.



Relinquishing control using wait( ) method

Blocked State

A thread is said to be *blocked* when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements. A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:

```
sleep( )          // blocked for a specified time
suspend( )        // blocked until further orders
wait( )           // blocked until certain condition occurs
```

These methods cause the thread to go into the ***blocked*** (or ***not-runnable***) state. The thread will return to the runnable state when the specified time is elapsed in the case of **sleep( )**, the **resume( )** method is invoked in the case of **suspend( )**, and the **notify( )** method is called in the case of **wait( )**.

Dead State

Every thread has a life cycle. A running thread ends its life when it has completed executing its **run( )** method. It is a natural death. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it. A thread can be killed as soon it is born, or while it is running, or even when it is in "not runnable" (blocked) condition.

Whenever we want to stop a thread from running further, we may do so by calling its **stop( )** method, like:

```
aThread.stop( );
```

This statement causes the thread to move to the ***dead*** state. A thread will also move to the dead state automatically when it reaches the end of its method. The **stop( )** method may be used when the *premature death* of a thread is desired.

Using Thread Methods

```
class A extends Thread
{
    public void run( )
    {
        for(int i = 1; i<=5; i++)
        {
            if(i==1) yield( );
            System.out.println("\tFrom Thread A : i = " +i);
        }
      System.out.println("exit from A ");
    }
}
class B extends Thread
{
    public void run( )
        {
            for(int i=1; i<=5; j++)
            {
                System.out.println("\tFrom Thread B : j = " + j);
                if(j==3) stop( );
        }
        System.out.println("Exit from B ");
    }
}
class C extends Thread
{
    public void run( )
    {
        for (int k=1; k<=5; k++)
        {
            System.out.println("\tFrom Thread C : k = " +k);
            if(k==1)
            try
                {
                    sleep(1000);
                }
            catch (Exception e)
            {
            }
        }
        System.out.println("Exit from C ");
    }
}

class ThreadMethods
{
    public static void main(String args[ ])
    {
        A threadA = new A( );
        B threadB = new B( );
        C threadC = new C( );

        System.out.println("Start thread A");
        threadA.start( );

        System.out.println("Start thread B");
        threadB.start( );

        System.out.println("Start thread C");
        threadC.start( );

        System.out.println("End of main thread");
    }
}
```

Output

```
Start thread A
Start thread B
Start thread C
    From Thread B : j = 1
    From Thread B : j = 2
    From Thread A : i = 1
    From Thread A : i = 2
End of main thread
    From Thread C : k = 1
    From Thread B : j = 3
    From Thread A : i = 3
    From Thread A : i = 4
    From Thread A : i = 5
Exit from A
    From Thread C : k = 2
    From Thread C : k = 3
    From Thread C : k = 4
    From Thread C : k = 5
Exit from C
```

Thread Exceptions

Note that the call to **sleep( )** method is enclosed in a **try** block and followed by a **catch** block. This is necessary because the **sleep( )** method throws an exception, which should be caught. If we fail to catch the exception, program will not compile.

Java run system will throw **IllegalThreadStateException** whenever we attempt to invoke a method that a thread cannot handle in the given state. For example, a sleeping thread cannot deal with the **resume( )** method because a sleeping thread cannot receive any instructions. The same is true with the **suspend( )** method when it is used on a blocked (Not Runnable) thread.

Whenever we call a thread method that is likely to throw an exception, we have to supply an appropriate exception handler to catch it. The **catch** statement may take one of the following forms:

```
catch (ThreadDeath e)
{
        . . . . . .
        . . . . . .                    // Killed thread
}
```

```
catch (InterruptedException e)
{
      . . . . . .
      . . . . . .                      // Cannot handle it in the current state
}
catch (IllegalArgumentException e)
{
      . . . . . .
      . . . . . .                      // Illegal method argument
}
catch (Exception e)
{
      . . . . . .
      . . . . . .                      // Any other
}
```

Thread Priority

In Java, each thread is assigned a priority, which affects the order in which it is scheduled for running. The threads that we have discussed so far are of the same priority. The threads of the same priority are given equal treatment by the Java scheduler and, therefore, they share the processor on a first-come, first-serve basis.

Java permits us to set the priority of a thread using the **setPriority( )** method as follows:

```
ThreadName.setPriority(intNumber);
```

The **intNumber** is an integer value to which the thread's priority is set. The **Thread** class defines several priority constants:

```
MIN_PRIORITY    =        1
NORM_PRIORITY   =        5
MAX_PRIORITY    =        10
```

The **intNumber** may assume one of these constants or any value between 1 and 10. Note that the default setting is NORM_PRIORITY.

```
class A extends Thread
{
    public void run( )
    {
        System.out.println("threadA started");
        for(int i=1; i<=4; i++)
        {
            System.out.println("\tFrom Thread A : i = " +i);
        }
        System.out.println("Exit from A ");
    }
}
class B extends Thread
{
    public void run( )
    {
        System.out.println("threadB started");
        for(int j=1; j<=4; j++)
        {
            System.out.println("\tFrom Thread B : j = " + j);
        }
        System.out.println("Exit from B ");
    }
}
class C extends Thread
{
    public void run( )
    {
        System.out.println("threadC started");
        for(int k=1; k<=4; k++)
        {
            System.out.println("\tFrom Thread C : k = " + k);
    }
```

```
        System.out.println("Exit from C ");
        }
}
class ThreadPriority
{
        public static void main(String args[ ])
        {
                A threadA = new A( );
                B threadB = new B( );
                C threadC = new C( );

                threadC.setPriority(Thread.MAX_PRIORITY);
                threadB.setPriority(threadA.getPriority( )+1);
                threadA.setPrioirty(Thread.MIN_PRIORITY);

                System.out.println("Start thread A");
                threadA.start( );

                System.out.println("Start thread B");
                threadB.start( );

                System.out.println("Start thread C");
                threadC.start( );

                System.out.println("End of main thread");
        }
}
```

```
Start thread A
Start thread B
Start thread C
threadB started
     From Thread B : j = 1
     From Thread B : j = 2
threadC started
     From Thread C : k = 1
     From Thread C : k = 2
     From Thread C : k = 3
     From Thread C : k = 4
Exit from C
End of main thread
     From Thread B : j = 3
     From Thread B : j = 4
Exit from B
threadA started
     From Thread A : i = 1
     From Thread A : i = 2
     From Thread A : i = 3
     From Thread A : i = 4
Exit from A
```

Synchronization

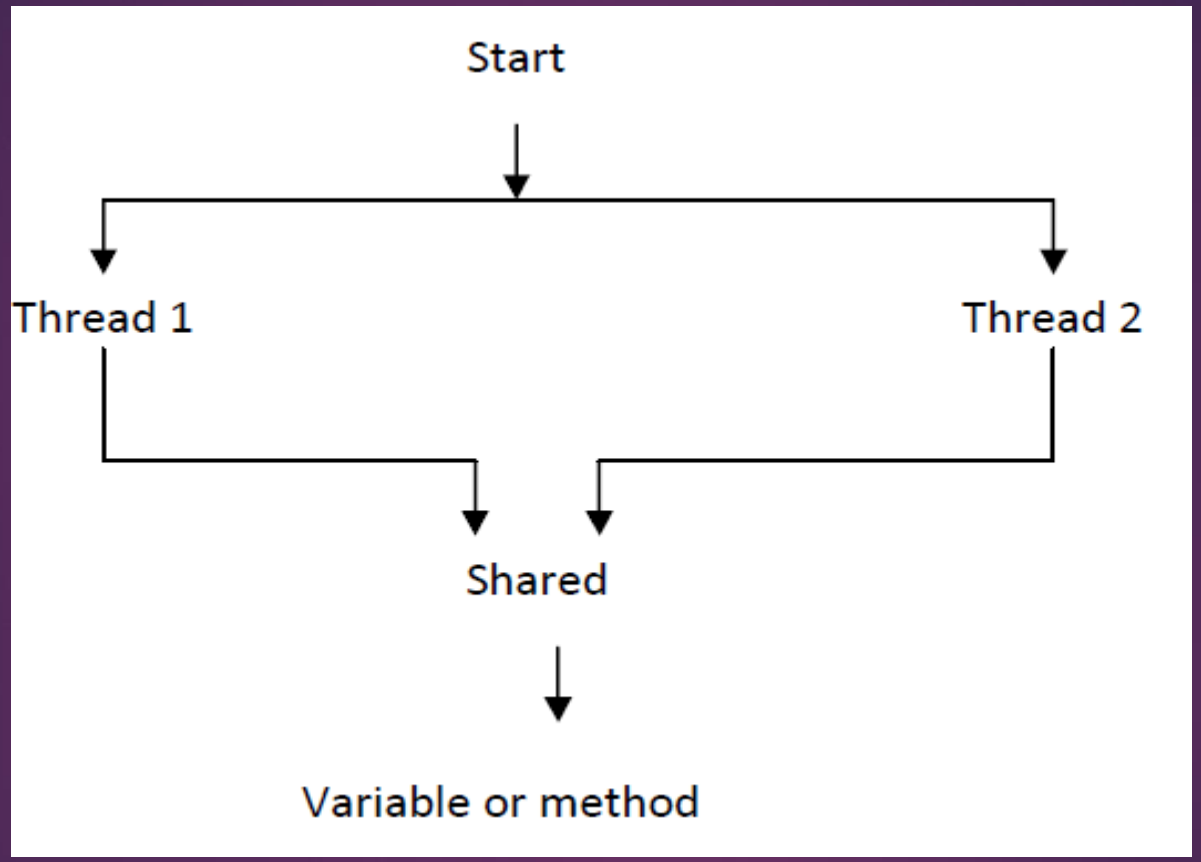
In Java, the threads are executed separately to each other. These types of threads are called as asynchronous threads. But two problems may occur with asynchronous threads.

1. Two or more threads share the similar resource (variable or method) while only one of them can access the resource at one time.
2. If the producer and the consumer are sharing the same kind of data in a program then either producer may make the data faster or consumer may retrieve an order of data and process it without its existing.

Suppose, we have created two methods as increment( ) and decrement( ). which increases or decreases value of the variable "count" by 1 respectively shown as:

```
public void increment( )
{
count++;
}
```

When the two threads are executed to access these methods (one for increment( ),another for decrement( )) then both will distribute the variable "count". in that case, we can't be sure that what value will be returned of variable "count".


Start
Thread 1
Thread 2
Shared
Variable or method

To avoid this problem, Java uses monitor also known as“semaphore” to prevent data from being corrupted by multiple threads by a keyword synchronized to coordinate them and intercommunicate to each other. It is basically a mechanism which allows two or more threads to share all the available resources in a sequential manner.

Java's synchronized is used to ensure that only one thread is in a critical region. Critical region is a lock area where only one thread is run (or lock) at a time. Once the thread is in its critical section, no other thread can enter to that critical region. In that case, another thread will has to wait until the current thread leaves its critical section.General form of the synchronized statement is as:

```
synchronized(object)
{
// statements to be synchronized
}
```

Lock:

Lock term refers to the access approved to a particular thread that can access the shared resources. At any given time, only one thread can hold the lock and thereby have access to the shared resource. Every object in Java has build-in lock that only comes in action when the object has synchronized method code.

By associating a shared resource with a Java object and its lock, the object can act as a guard, ensuring synchronized access to the resource. Only one thread at a time can access the shared resource guarded by the object lock.

Since there is one lock per object, if one thread has acquired the lock, no other thread can acquire the lock until the lock is not released by first thread. Acquire the lock means the thread currently in synchronized method and released the lock means exits the synchronized method.

Remember the following points related to lock and synchronization:

1. Only methods (or blocks) can be synchronized, Classes and variable cannot be synchronized.
2. Each object has just one lock.
3. All methods in a class need not to be coordinated. A class can have both synchronized and non-synchronized methods.
4. If two threads wants to execute a synchronized method in a class, and both threads are using the similar instance of the class to invoke the method then only one thread can execute the method at a time.
5. If a class has both synchronized and non-synchronized methods, multiple threads can still access the class's nonsynchronized methods. If you have methods that don't access the data you're trying to protect, then you don't need to synchronize them. Synchronization can cause a hit in several cases (or even deadlock if used incorrectly), so you should be careful not to overuse it.
6. If a thread goes to sleep, it holds any locks it has—it doesn't let go them.
7. A thread can obtain more than one lock. For example, a thread can enter a synchronized method, thus acquiring a lock, and then directly invoke a synchronized method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are unrestricted again.
8. A block of code can also be synchronized.
9. Constructors cannot be synchronized

```
synchronized void update( )
{
    ..........
    ..........   // code here is synchronized
    ..........
}
```

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

It is also possible to mark a block of code as synchronized as shown below:

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

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

An interesting situation may occur when two or more threads are waiting to gain control of a resource. Due to some reasons, the condition on which the waiting threads rely on to gain control does not happen. This results in what is known as *deadlock*. For example, assume that the thread A must access Method1 before it can release Method2, but the thread B cannot release Method1 until it gets hold of Method2. Because these are mutually exclusive conditions, a deadlock occurs. The code below illustrates this:

Thread A

```
synchronized method2( )
{
    synchronized method1( )
    {
        . . . . . .
        . . . . . .
    }
}
```

Thread B

```
synchronized method1( )
{
    synchronized method2( )
    {
        . . . . . .
        . . . . . .
    }
}
```

Creating Threads using Runnable interface

1. Declare the class as implementing the **Runnable** interface.
2. Implement the **run( )** method.
3. Create a thread by defining an object that is instantiated from this "runnable" class as the target of the thread.
4. Call the thread's **start( )** method to run the thread.

```
class X implements Runnable                                    // Step 1
{
    public void run( )                                         // Step 2
    {
        for(int i = 1; i<=10; i++)
        {
            System.out.println("\tThreadX : " +i);
        }
        System.out.println("End of ThreadX");
    }
}
class RunnableTest
{
    public static void main(String args[ ])
    {
        X runnable = new X( );
        Thread threadX = new Thread(runnable);                 // Step 3
        threadX.start( );                                      // Step 4
        System.out.println("End of main Thread");
    }
}
```

Output

```
End of main Thread
        ThreadX : 1
        ThreadX : 2
        ThreadX : 3
        ThreadX : 4
        ThreadX : 5
        ThreadX : 6
        ThreadX : 7
        ThreadX : 8
        ThreadX : 9
        ThreadX : 10
End of ThreadX
```

References:

1. “Programming With JAVA – A Primer” , E. Balagurusamy, Fourth Edition, TMH Publications

2. “Java Programming” Internet Content – Anonomous Author

3. “The complete reference JAVA2”, Hervert schildt. TMH Publications.