Chapter-9
Multithreaded Programming

Multi threading is a conceptual programming paradigm where a program (process) is divided into two or more subprograms (processes), which can be implemented at the same time in parallel. For example, one subprogram can display an animation on the screen while another may build the next animation to be displayed. This is something similar to dividing a task into subtasks and assigning them to different people for execution independently and simultaneously.

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 java can be though of as *single-threaded* programs. 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

The ability of a language to support multi threads 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.

Multithreading is a powerful programming tool that makes Java distinctly different from its fellow programming languages. Multithreading is useful in a number of ways. It enables programmers to do multiple things at one time. They can divide a long program (containing operations that are conceptually concurrent) into threads and execute them in parallel. For example, we can send tasks such as printing into the background and continue to perform some other task in the foreground. This approach would considerably improve the speed of our programs. Threads are extensively used in Java-enabled browsers such as HotJava. These browsers can download a file to the local computer, display a Web page in the window, and output another Web page to a printer and so on.

9.1 Creatingthreads

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. 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 ().

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

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 thethread.
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 thethread.

The approach to be used depends on what the class we are creating requires. If it requires to extend another class, then we have no choice but to implement the Runnable interface, since Java classes cannot have two superclasses.

Extending the thread class

We can make our class runnable as a thread by extending the class java.lang. Thread. This gives us access to all the thread methods directly. It includes the following steps:
1. Declare the class as extending the Threadclass.
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 threadexecution.

9.2 Declaring theClass

The Thread class can be extended as follows:

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

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 ( ) is:

```
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 takesplace.

9.3 Starting NewThread

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 pausing 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

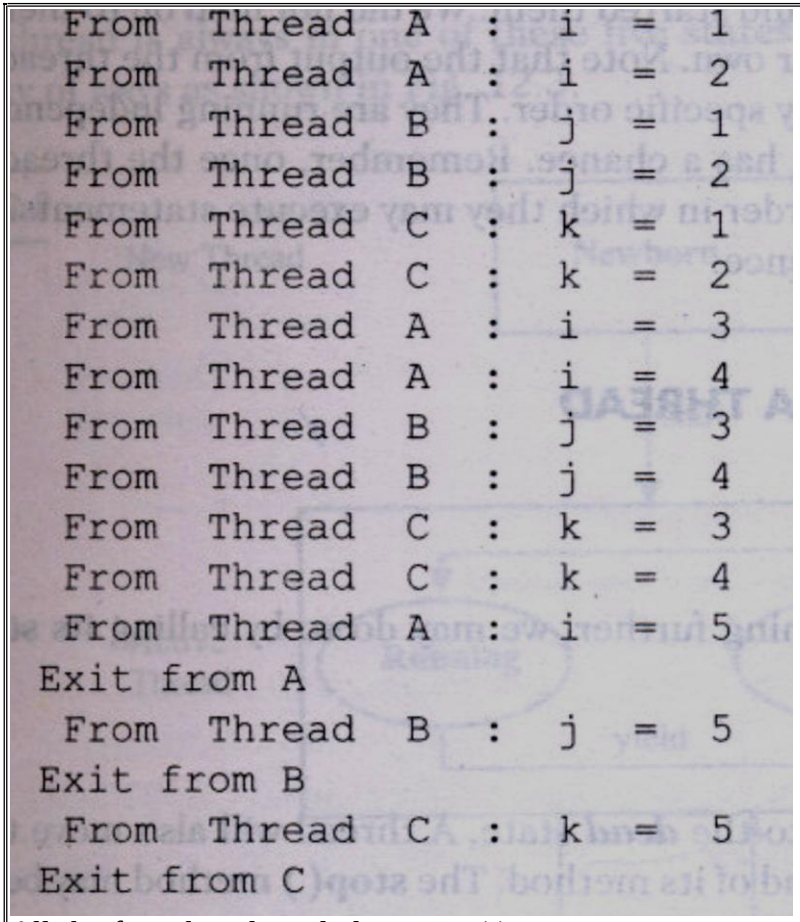
An Example of Using the Thread Class

The below program illustrates the use of Thread class for creating and running threads in an application. The program creates three threads A, B, and C for undertaking three different tasks. The main method in the ThreadTest class also constitutes another thread which we may call the "main thread".
The main thread dies at the end of its main method. However, before it dies, it creates and starts all the three threads A, B, and C. Note the statements like
new A ( ).start ( );
in the main thread. This is just a compact way of starting a thread. This is equivalent to:
A threadA = new A( );
threadA.start();
Immediately after the thread A is started, there will be two threads running in the program i.e. the main thread and the thread A. The start( ) method returns back to the main thread immediately after invoking the rune) method, thus allowing the main thread to start the thread B.

Creating threads using the thread class

```
class A extends Thread
{
      public void run( )
     {
       for(int i=1; i<=5;i++)
          {
             System.out.println("\tFrom Thread A : 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 C 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



All the four threads including main( ) are running concurrently on their own. Note that the output from the threads are not specially sequential. They do not follow any specific order. They are running independently of one another and each executes whenever it has a chance. Remember, once the threads are started, we cannot decide with certainty the order in which they may execute statements. Note that a second run has a different output sequence.

Stopping and blocking a thread

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.

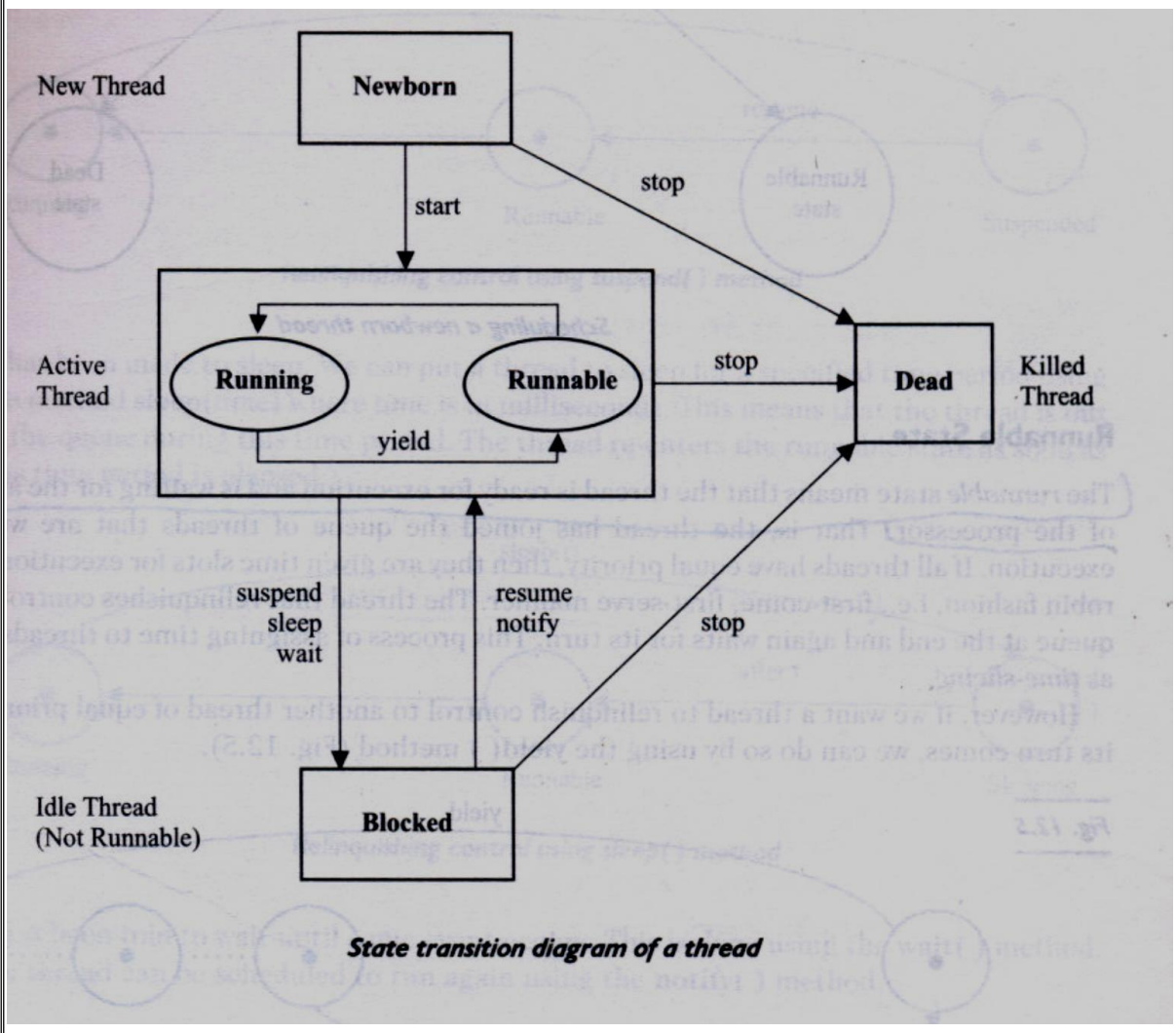
9.4 Blocking a Thread
A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:
sleep () / / blocked for a specified time
suspend( ) / / blocked until further orders
wait() / / blocked until certaincondition 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( ).

9.5 Life cycle of athread

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

1. Newbornstate
2. Runnablestate
3. Runningstate
4. Blockedstate
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 figure.



State transition diagram of a thread

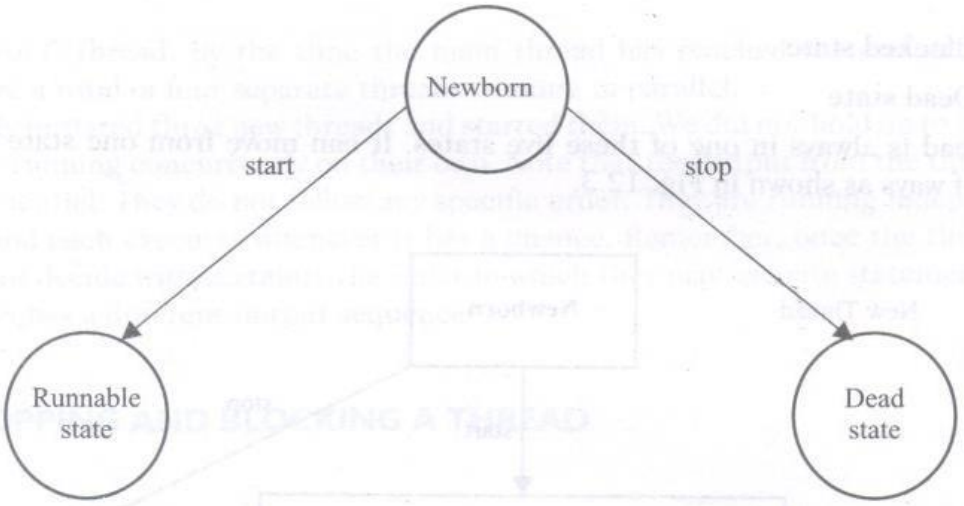
Newborn State

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.

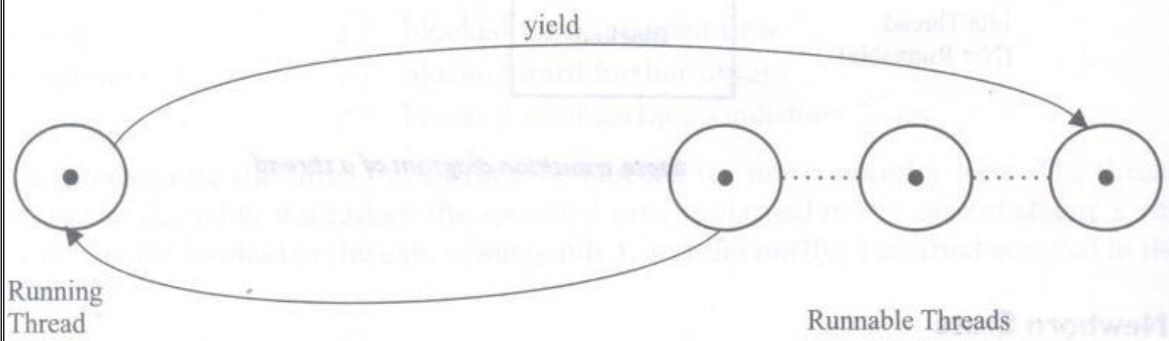
If scheduled, it moves to the runnable state. If we attempt to use any other method at this stage, an exception will be thrown.



Scheduling a newborn thread

Runnable State

The *runnable* state means that the thread is ready for execution and is waiting forthe 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 of equal priority before its turn comes, we can do so by using the yield( )method.

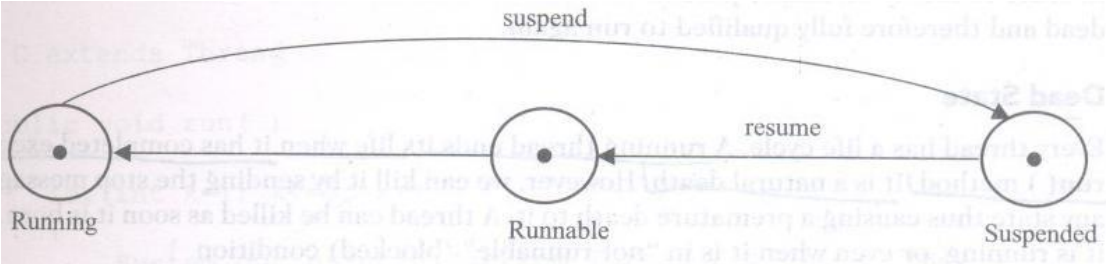


Relinquishing control using yield( ) method

Running State

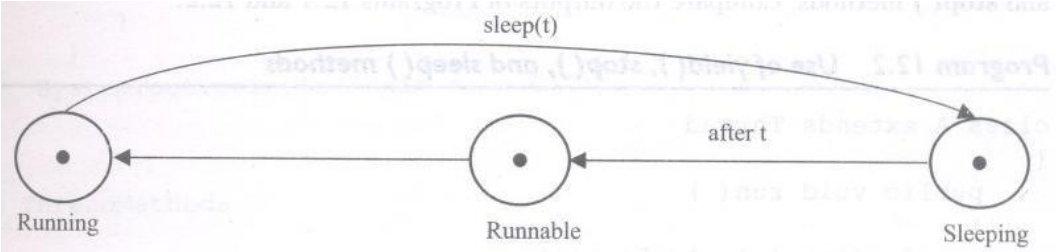
Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread. A running thread may relinquish its control in one of the following situations.

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 killit.



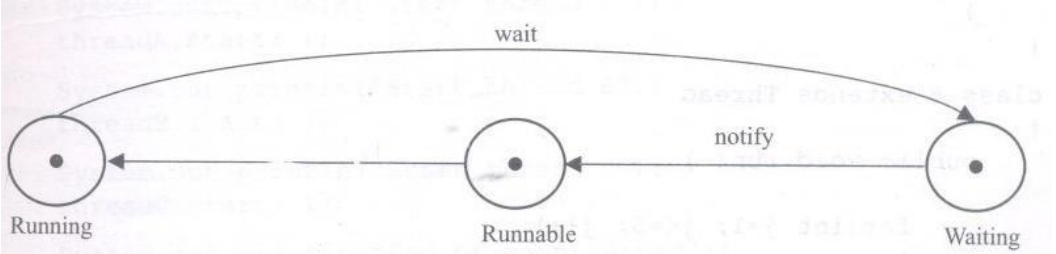
Relinquishing control using suspend( ) method

2. It has been made to sleep. We can put a thread to sleet 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 iselapsed.



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 runagain.

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 natur l 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.

9.6 Using thread methods

Thread class methods can be used to control the behaviour of a thread. There are also methods that can move a thread from one state to another. Program illustrates the use of yield(), sleep() and stop()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 j=1; j<=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");
    }
}
```

Out Put

```
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
```

Program uses the yield( ) method in thread A at the iteration i = 1. Therefore, the thread A, although started first, has relinquished its control to the thread B. The stop() method in thread B has killed it after implementing the for loop only three times. Note that it has not reached the end of run( ) method. The thread C started sleeping after executing the for loop only once. When it woke up (after 1000 milliseconds), the other two threads have already completed their runs and therefore was running alone. The main thread died much earlier than the other three threads.

Thread exceptions

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. 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 stament 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 asfollows:

```
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.

Most user-level processes should use NORM_PRIORITY, plus or minus 1. Back-ground tasks such as network I/O and screen repainting should use a value very near to the lower limit. We should be very cautious when trying to use very high priority values. This may defeat the very purpose of using multithreads. By assigning priorities to threads, we can ensure that they are given the attention (or lack of it) they deserve. For example, we may needtoansweraninputasquicklyaspossible.Whenevermultiplethreadsarereadyfor

execution, the Java system chooses the highest priority thread and executes it. For a thread of lower.priority to gain control, one of the following things should happen:
It stops running at the end of run( ).
2. It is made to sleep using sleep().
3. It is told to wait using wait().

However, if another thread of a higher priority comes along, the currently running thread will be *preempted* by the incoming thread thus forcing the current thread to move to the runnable state. Remember that the highest priority thread always preempts any lower priority threads.

```
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.setPriority(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");
    }
}
```

Output

```
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
```

9.7 Synchronization

If a thread try to use data and methods outside their run( ) method then they may compete for the same resources and may lead to serious problems. For example, one thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we may get strange results. Java enables us to overcome this problem using a technique known as *synchronization.* In case of Java, the keyword synchronised helps to solve such problems by keeping a watch on such locations. For example, the method that will read information from a file and the method that will update the same file may be declared as synchronized.Example:

```
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 shownbelow:

```
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 reason, 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.

```
Thread A
    synchronized method2( )
    {
        synchronized method1( )
        {
            ..........
            ..........
        }
    }
Thread B
    synchronized method1( )
    {
        synchronized method2( )
        {
            ..........
            ..........
        }
    }
```

9.8 Implementing the runnable interface

We stated earlier that we can create threads in two ways: one by using the extended Thread class and another by implementing the Runnable interface, In this section, we shall see how to make use of the Runnable interface to implement threads
The Runnable interface declares the run( ) method that is required for implementing threads in our programs. To do this, we must perform the steps listed below:

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

```
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");
    }
}
```

Chapter-10
Managing Errors and Exceptions

10.1 Introduction

Rarely does a program run successfully at its very first attempt. It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing the program to produce unexpected results. *Errors* are mistakes that can make a program go wrong.

An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

10.2 Types of errors

Errors may broadly be classified into two categories:
Compile-time errors
Run-time errors

Compile-Time Errors

All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the .class file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.
The Java compiler does a nice job of telling us where the errors are in the program. For example, if we have missed the semicolon at the end of print statement the following message will be displayed in thescreen:

```
Error1.java :7: ';' expected
System.out.println ("Hello Java!")
^
1 error
```

Most of the compile-time errors are due to typing mistakes. Typographical errors are hard to find. We may have to check the code word by word, or even character by character. The most common problems are:

- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments / initialization
- Bad references to objects
- Use of = in place of == operator
- And so on

Other, errors we may encounter are related to directory paths. An error such as javac : command not found means that we have not set the path correctly. We must ensure that the path includes the directory where the Java executables are stored.

Run-Time Errors

Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overt1ow{Most common run-time errors are:

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible class or type
- Trying to cast an instance of a class to one of its subclasses
- Passing a parameter that is not in a valid range or value for a method
- Trying to illegally change the state of a thread
- Attempting to use a negative size for an array
- Using a null object reference as a legitimate object reference to access a method or a variable
- Converting invalid string to a number
- Accessing a character that is out of bounds of a string
- And many more

10.3 Exceptions

An *exception* is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it(i.e., informs us that an error has occurred). If the exception object is not caught and handled properly, the interpreter will display an error message and will terminate the program.

If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as *exception handling*

The purpose of exception handling mechanism is to provide a means to detect and report an exceptional circumstance so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the followingtasks:

1. Find the problem (***Hit*** the exception).
2. Inform that an error has occurred (***Throw*** the exception)
3. Receive the error information (***Catch*** the exception)
4. Take corrective actions (***Handle*** the exception)

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions. When writing programs, we must always be on the lookout for places in the program where an exception could be generated.

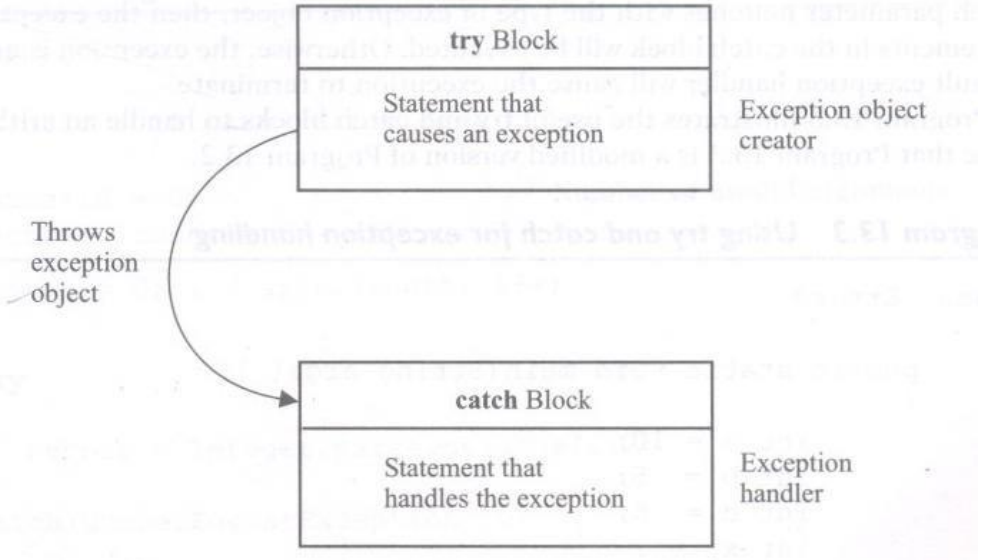
Common Java Exceptions

Exception Type	Cause of Exception
ArithmeticException	Caused by math errors such as division by zero
ArrayIndexOutOfBoundsException	Caused by bad array indexes
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
FileNotFoundException	Caused by an attempt to access a nonexistent file

Exception Type	Cause of Exception
IOException	Caused by general I/O failures, such as inability to read from a file
NullPointerException	Caused by referencing a null object
NumberFormatException	Caused when a conversion between strings and number fails
OutOfMemoryException	Caused when there's not enough memory to allocate a new object
SecurityException	Caused when an applet tries to perform an action not allowed by the browser's security setting
StackOverflowException	Caused when the system runs out of stack space
StringIndexOutOfBoundsException	Caused when a program attempts to access a nonexistent character position in a string

Syntax of exception handling code

The basic concepts of exception handling are *throwing* an exception and catching it.



Java uses a keyword try to preface a block of code that is likely to cause an error condition and "throw" an exception. A catch block defined by the keyword catch "catches" the exception "thrown" by the try block and handles it appropriately. The catch block is added immediately after the try block. The following example illustrates the use of simple try and catch statements:

```
.....
.....
try
{
    statement;      // generates an exception
}
catch (Exception-type e)
{
    statement;      // processes the exception
}
.....
.....
```

The try block can have one or more statements that could generate an exception. If anyone statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block) that is placed next to the try block. The catch block too can have one or more statements that are necessary to process the exception. Remember that (every try statement should be followed by *at least one* catch statement; otherwise compilation error will occur.). catch statement works like a method definition. The catch statement is passed
a single parameter, which is reference to the exception object thrown (by the try block). If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught and the default exception handler will cause the execution to terminate.

Using* try *and* catch *for* exception *handling

```
class  Error3
{
    public static void main(String args[ ])
    {
        int a = 10;
        int b =  5;
        int c =  5;
        int x, y ;

        try
        {
            x  =  a / (b-c);     // Exception here
        }
        catch (ArithmeticException e)
        {
            System.out.println("Division by zero");
        }
        y  =  a / (b+c);
        System.out.println("y  =  " + y);
    }
}
```

Program displays the following output:

```
Division by zero
y = 1
```

The program did not stop at the point of exceptional condition. It catches the error condition, prints the error message, and then continues the execution, as if nothing has happened.

Multiple catch statements

It is possible to have more than one catch statement in the catch block as illustrated below:

```
.....
.....
try
{
   statement ;                    // generates an exception
}
catch (Exception-Type-1  e)

{
   statement;                  // processes exception type 1
}
catch (Exception-Type-2  e)
{
   statement;                  // processes exception type 2
}
.
.
.
catch (Exception-Type-N  e)
{
   statement ;                 // processes exception type N
 }
.....
.....
```

When an exception in a try block is generated, the Java treats the multiple catch statements like cases in a switch statement. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will skipped. Note that Java does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion.
Example: catch (Exception e) ;
The catch statement simply ends with a semicolon, which does nothing. This statement will catch an exception and then ignore it.

Using *multiple catch blocks*

```
class Error4
{
    public static void main(String args[ ])
    {
        int a[ ]  =  {5,10};
        int b  =  5;

        try
        {
            int x = a[2] / b - a[1];
        }

        catch(ArithmeticException  e)
        {
            System.out.println("Division by zero");
        }

        catch(ArrayIndexOutOfBoundsException  e)
        {
            System.out.println("Array index error");
        }

        catch(ArrayStoreException e)
        {
            System.out.println("Wrong data type");
        }

        int y  =  a[1] / a[0];
        System.out.println("y =  " + y);
    }
}
```

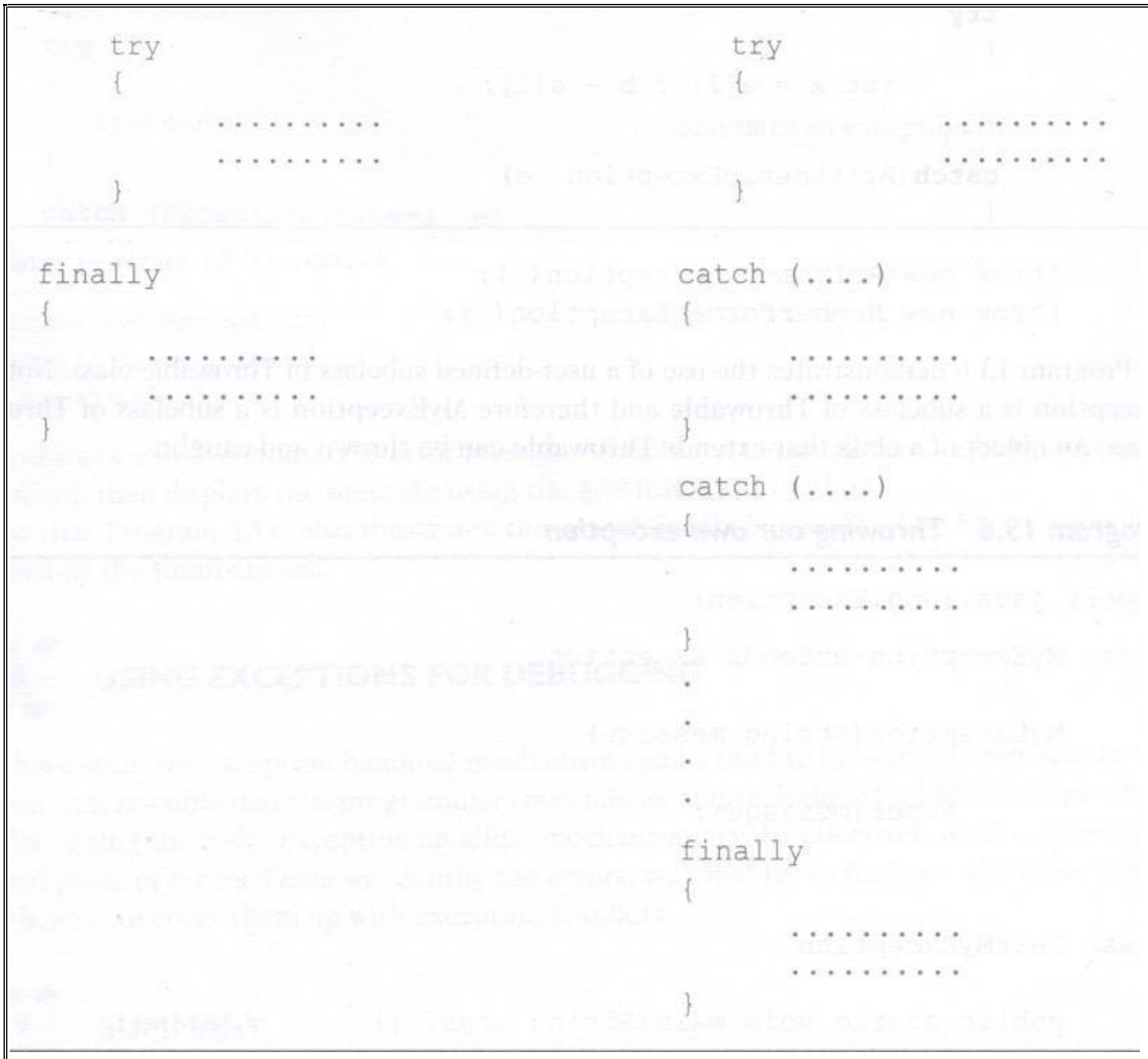
uses a chain of catch blocks and, when run, produces the following output:

```
Array index error
y  =  2
```

Note that the array element a[2] does not exist because array a is defined to have only two elements, a(O] and all]. Therefore, the index 2 is outside the array boundary thus causing the block. Catch(ArraylndexOutOfBoundsException e) to catch and handle the error. Remaining catch blocks areskipped.

Using finallyStatement

Java supports another statement known as finally statement that can be used to handle an Exception that is not caught by any of the previous catch statements. finally block can be used to handle any exception generated with in a try block. It may be added immediately after the try block or after the last catch block shown asfollows:



When a finally block is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown. As a result, we can use it to perform certain house-keeping operations such as closing files and releasing system resources.

Throwing our own exceptions

There may be times when we would like to throw our own exceptions. We can do this by using the keyword throw as follows:

```
throw new Throwable_subclass;
```

Examples:
throw new ArithmeticException( );
throw new NumberFormatException( );
Exception is a subclass of Throwable and therefore MyException is a subclass of Throwable class. An object of a class that extends Throwable can be thrown and caught.

***Throwing our own* exception**

```
import java.lang.Exception;
class MyException extends Exception
{
      MyException(String message)
      {
            super(message);
      }
}
class TestMyException
{
      public static void main(String args[ ])
      {
            int x = 5, y = 1000;
            try
            {
               float z = (float) x / (float) y ;
               if(z < 0.01)
               {
                  throw new MyException("Number is too smal
               }
            }
            catch (MyException   e)
            {
               System.out.println("Caught my exception");
               System.out.println(e.getMessage( ) );
            }
            finally
            {
               System.out.println("I am always here");
            }
       }
 }
```