When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exception of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error.

9.4.1 Using Finally Statement

The idea behind finally is that the programmer needs to have a way to correctly dispose off system resources, such as open files, no matter which exceptions are thrown by the code in a try block. If try statement has a finally clause attached, the code block associated with finally is always executed regardless of how the try block exits.

- Normal termination, by falling through the end brace.
- Because of return or break statement.
- Because an exception was thrown.

The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

Example:

```
public int testx(String x)
{
   try
   {
      return someMethod(x);
   }
   catch (NullPointerException nex)
   {
      System.out.print("Null Pointer,");
      return -1;
   }
   catch (Run timeException rex)
   {
      System.our.print("Runtime");
      return -2;
   }
   finally
   {
      System.out.println("Finally");
   }
}
```

Output:

```
    No exception thrown in SomeMethod( )
     "Finally" is printed
```

```
If NullPointerException is thrown

"NullPointer, Finally" is printed

If ArithmeticException is thrown in someMethod( )

"Runtime, Finally" is printed.

An uncaught exception is thrown in someMethod "Finally" is printed.
```

9.5 THROWING OUR OWN EXCEPTIONS

It is possible for your program to throw an exception explicitly rather than let it be thrown by the Java run-time system. The exceptions that you throw can be standard system exceptions or you can create your own. The general syntax is

```
throw expression;
```

Here, expression is an object of type Throwable or subclass of Throwable. There are two ways of creating Throwable object:

- Using a parameter into catch clause.
- Creating one with the new operator.

```
throw new NullPointerException("demo");
throw new ArithmeticException( );
```

The flow of execution stops immediately after the throw statement.

The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the execution. If not, then the next enclosing try statement is inspected and so on. If no matching catch is found, then default exception handler halts the program and prints stack trace.

```
class ThrowDemo
{
   static void demoproc( )
   {
      try
      {
         throw new NullPointException("Demo");
      }
      catch (NullPointerException e)
      {
   System.out.println("Caught inside demoproc");
   throw e;
      }
   }
   public static void main(String args[ ])
   {
        try
        {
           demoproc( );
        }
        catch(NullPointerException e)
        {
           System.out.println("Recaught" + e);
        }
    }
}
```

Output:

```
Caught inside demoproc
Recaught : java.lang.NullPointerException : demo
```

main( ) method sets up an exception context and then calls demoproc( ). demoproc( ) method sets up another exception context and throws NULLPointerException. The exception is then rethrown.

9.6 THROWS

The alternative to using try and catch wherever a checked exception occurs, is simply to declare that the method throws the exception. If a method is capable of causing an exception that it does not handle, it must specify this behaviour so that the callers of the method can guard themselves against that exception. In other words, a method that throws an exception within it must catch that exception or have that exception declared in its "throws" clause, unless the exception is a subclass of either the Error clause or the RuntimeException class.

A throws clause lists the type of exception that a method might throw. General form of throws clause:

```
type method name(parameter List) throws exceptionlist
{
   // body of method
}
```

Example: int calc (String filename) throws IOException, InterruptedException

When you have a method in a subclass that has the same name, parameter list and return type as a method in the parent class, the subclass method is said to override the parent class. The Java compiler will allow the overriding subclass method to be defined as, throwing the same exception as its superclass method, throwing one or more subclasses of superclass exception or not throwing any exceptions. It will not allow the overriding subclass to be declared as throwing a more general exception or exception from other hierarchy.

Example:

```
class Throwdemo
{
static void throwOne( ) throws illegalAccessException
   {
      System.out.println ("Inside throwOne");
```

```
        throw new illegalAccessException ("demo");
    }
    public static void main(String arge[ ])
    {
        try
        {
            throwOne( );
        }
        catch (illegalAccessException e)
        {
            System.out.println ("Caught" + e);
        }
    }
}
```

Output:

```
inside throwOne
        Caught java.lang.IllegalAccessException : demo
```

The compiler relies on the declaration of throws clauses to determine if an exception may occur in an expression, a statement, or a method. The compiler issues an error message for any method that does not declare all exceptions in its throws clause.

9.7 JAVA'S BUILT-IN EXCEPTIONS

Inside the standard package java.lang, Java defines several exception classes. The most general exceptions are subclasses of type Runtime Exception. Since java.lang is implicitly added, most exceptions derived from Runtime Exception are automatically available (Table 9.1).

Table 9.1

EXCEPTION	MEANING
Arithmetic Exception	Arithmetic error, such as divide by zero.
ArrayIndexOutOfBoundException	Array index is out of bound.
ArrayStoreException	Assignment to an array element of an incompatible type
ClassCast Exception	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
Illegal MonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
NegativeArraySizeException	Array created with a negative size.
Class NotFoundException	Class not found.
No SuchMethodException	A requested method does not exist.
InterruptedException	One thread has been interrupted by another thread.
StackOverflowException	Caused when the system runs out of stack space.You can define your own exception types to handle situations specific to your applications. It is done just by defining a subclass of Exception. All exceptions which you create have the methods defined by Throwable available to them. You can override one or more methods in exception class that you create. For e.g. String toString(), void print StackTrace( ), String getMessage( ).

Check Your Progress

Fill in the blanks:

1. A compile time error is shown by the ....
2. A .... error occurs due to memory-management problem.
3. An .... is a condition that is caused by a runtime error in the program.
4. When an exception is thrown, each .... statement is inspected in order, and the first one whose type matches that of the exception is executed.
5. The programmer can provide for handling of exceptions using the Java keywords .... and catch.

9.8 LET US SUM UP

A good program does not produce unexpected results. Some features should be incorporated in the program that could check for potential problem spots in programs and guard against program failures. Exceptions in Java must be handled carefully to avoid any program failure.

In this lesson we have discussed: Errors and their types including compile time errors and runtime errors; What exceptions are; How to throw system exceptions; How to define our own exceptions; How to catch and handle different types of exceptions; Where to use exception handling tools.

9.9 KEYWORDS

Error: The wrongs that can make a program go wrong.

Compile-time Errors: Errors such as syntax errors which will be detected and displayed by the Java compiler.

Run-time Errors: Errors produced due to wrong logic or due to stack overflow which will not be detected and displayed at compile time.

Exception: A condition that is caused by a run-time error in the program.

Exception Handler: Code that responds to and attempts to recover from an exception.

9.10 QUESTIONS FOR DISCUSSION

1. What is an error?
2. Give some common compile-time errors.
3. Give some common run-time errors.
4. What do you mean by exception handling?
5. How do we define a try block?
6. How do we define a catch block?
7. Define an exception called "No match Exception" that is thrown when a string is not equal to "India". Write a program that uses this exception.

Check Your Progress: Model Answers

1. Compiler
2. Run-time
3. Exception
4. Catch
5. Try

9.11 SUGGESTED READINGS

E. Balaguruswami, *Programming with Java*, Tata McGraw-Hill.

Davis, Stephen R., *Learn Java Now*, Microsoft Press.

Naughton, Patrick, *The Java Hand Book*, OSborne McGraw- Hill.

Sams.net, *Java unleased.*

Herbert Schildt, *The Complete Reference Java 2*, Tata McGraw-Hill.

LESSON

10

MULTIPLE THREAD PROGRAMMING

CONTENTS

10.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Define the multiple thread programming
- Describe java threads and creating several threads
- Explain deadlock-control on threads

10.1 INTRODUCTION

Java programs contain only single sequential flow of execution. A typical program is executed sequentially and is single threaded. If a single threaded program's execution is blocked while waiting for some I/O operation, no other portion of the program can proceed. All modern operating systems may execute multiple programs simultaneously, even if there is only a single CPU available to run all the applications. Multithreading allows multiple tasks to execute concurrently within a single program. The advantage of multiple threads in a program is that it utilizes system resources (like CPU) better because other threads can grab CPU time when one line of execution is blocked. By using multithreading, you can create programs showing animation, play music, display document and download files from the network simultaneously. One unique feature of Java is the ease with which a programmer can write multithreaded programs. In C or C++, implementation of multithreading involves platform specific toolkit. Java was designed from the start to accommodate multithreading. In this lesson you will learn how to create threads using thread class and runnable interface and how to allocate priority of execution to different threads in a program. You will be able to create a synchronized method to avoid collision in which more than one thread attempts to modify the same object at the same time.

10.2 OVERVIEW OF MULTITHREADING

Multithreading allows a running program to perform several tasks, apparently, at the same time. Java uses threads to enable the entire environment to be asynchronous. It makes maximum use of CPU because idle time can be kept to minimum. This helps reduce inefficiency by preventing the waste of CPU cycles. For example, it is useful for networked environment as data transfer over network is much slower than the rate at which the computer works. Multithreading lets you gain access to this idle time and put it to good use. Single-threaded system works on an event loop with polling approach. 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 a signal (e.g. 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 can happen in the system. This wastes CPU time. The benefit of Java multithreading is that the main loop/polling mechanism is eliminated.

The Java runtime system depends on threads for many things and all class libraries are designed with multithreading in mind. Threads are used extensively in Java enabled browsers such as Hot Java. A thread exists in several states (Figure 10.1).

- Running thread
- Ready to run as soon as it gets CPU time
- Suspended thread
- Suspended thread can be resumed
- Thread can be blocked
- Thread can be terminated.

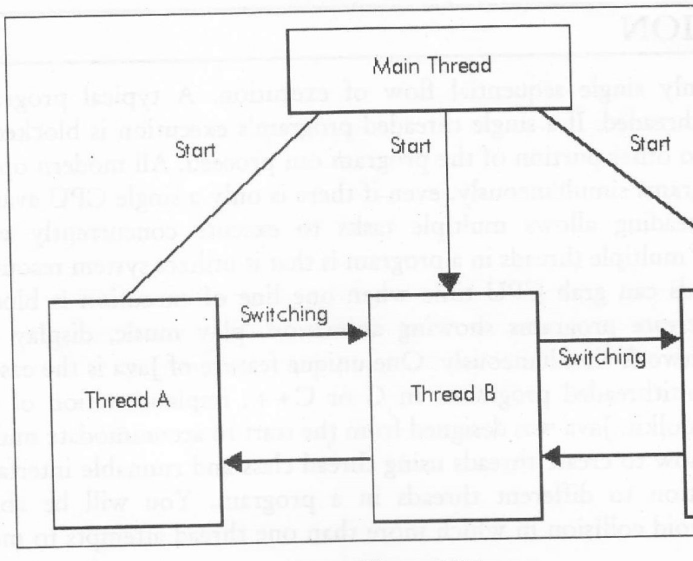

Figure 10.1

10.3 JAVA THREAD

Any single path of execution is called thread. The path is defined by elements such as stack (for handling calls to other methods with the code) and set of registers to store temporary memory. These elements are collectively known as context. Every program has at least one thread. Each thread has its own stack, priority and virtual set of registers. If there are two pieces of code that do not have the same context, then they are executed in two different threads.

Threads subdivide the runtime behavior of a program into separate, independently running subtasks. Switching between threads is carried out automatically by JVM. Even in a single processor system, thread provides an illusion of carrying out several tasks simultaneously. This is possible because switching between threads happens very fast.

10.3.1 The Thread Control Methods

The Thread class defines several methods that help manage threads. Some of them are as are as shown in Table 10.1:

Table 10.1

Method	Meaning
getName	Obtain a thread's name
getPriority	Obtain a thread's priority
isAlive	Determine if a thread is still running
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

10.3.2 Thread's Life Cycle

A thread is always in one of five states (Figure 10.2).

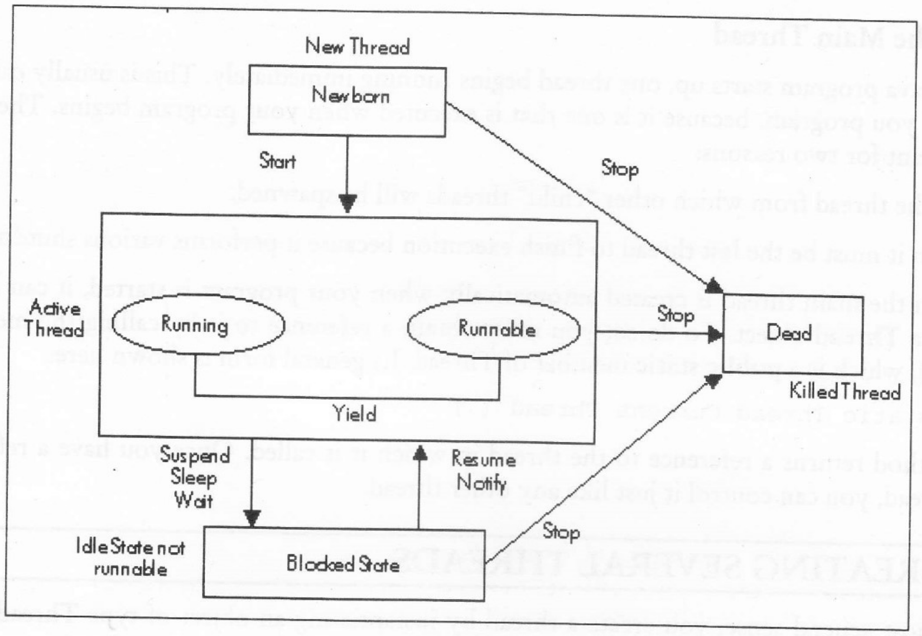

Figure 10.2

When we create a thread object, it is said to be in **newborn state**. At this stage we can start the thread using start( ) method or kill it using stop( ) method. The thread in **runnable state** means that the thread is ready for execution and is waiting for CPU time. **Running state** means CPU has given its time to the thread for execution.

A thread is said to be in **blocked state** when it is suspended, sleeping or waiting in order to satisfy some condition. The suspended thread can be revived by using the resume( ) method. The Thread which is sleeping can enter into runnable state as soon as the specified time period is elapsed. The thread in the wait( ) state can be scheduled to run again using the notify( ) method. The thread dies when its run( ) method completes its execution. A thread can be killed using the stop( ) method.

10.3.3 Where are Threads Used?

Word processing program handles a print request in a separate thread. Many users find it convenient to continue working in word-processing application even after submitting print request. Downloading a file from Internet and browsing another site are implemented in different threads in the browser program. The Oracle database is another example of a program that uses threads. Database "connect" request to a common resource is handled by different threads. Operating Systems (specially networked OS) are written using threads. Operating systems usually run many applications such as a web browser and a word processing program at the same time. A multithreaded program can indirectly benefit the end user because it can efficiently use the CPU.

If a program is waiting for user response over the network, during this waiting time other useful tasks can be performed simultaneously. A thread program can directly benefit the user in applications that employ GUI components. When a button is pressed to execute one task, the program can make other buttons available to perform other independent programs.

10.3.4 The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of you program, because it is one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other "child" threads will be spawned.
- Often it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method current Thread (), which is a **public static** member of Thread. Its general form is shown here:

```
static Thread Current Thread ( )
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

10.4 CREATING SEVERAL THREADS

In the most general sense, you create a thread by instantiating an object of type Thread. Java defines two ways in which this can be accomplished:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

10.4.1 Creating Thread using the Thread Class

Creation of thread is a two step process:

1. Create the new class:

   (a) Define a subclass of thread.

   (b) Override its run( ) method.

2. Instantiate and run the thread:

   (a) Create an instance of class.

   (c) Call its start( ) method.

This causes the scheduler to launch the new thread by calling its run( ) method and putting the thread in the queue for execution. When the other running threads release CPU, it will run.

```
class A extends Thread
{
  public void run( )
  {
    for (int i = 1; i < = 5; i++)
    {
            System.out.println ("\t From 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 ("\t From 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 ("\t From Threadc : k = " +k);
            }
            System.out.println ("Exit from (");
            }
          }
          class ThreadTest
          {
            public static void main( )
            {
                new A( ).start( );
                new B( ).start( );
                new C( ).start( );
            }
          }
```

In the above program, main thread initiated three new threads and started them. The output of the program will not be sequential. They do not follow any specific order. They are running independently and each executes whenever it has a chance. Note that every time we run this program we get different output sequence.

10.4.2 Creating Threads using Runnable Interface

To create a thread, declare a class that implements the runnable interface. To implement runnable, a class needs to only implement a single method called run( ). After you create a class that implements runnable, you will instantiate an object of type thread using the constructor of thread class.

Thread (Object of Runnable Interface)

The new thread will not start running until you call its start( ) method. Start( ) executes a call to the run( ) method.

```
class X implements Runnable
          {
            public void run( )
            for (int i = 1; i < = 10; i++)
            {
                System.out.println ("\t Thread x: " +i);
            }
            System.out.println ("End of ThreadX");
          }
          class RunnableTest
          {
            public static void main( )
            {
                X runnable = new X( );
                Thread threadX = new Thread (runnable);
            threadX.start( );
            System.out.println ("End of main Thread");
            }
          }
```

In main method, we first create an instance of X and then pass this instance as the initial value of the object thread X. Whenever the new thread threadX starts up, its run( ) method calls the run() method of the target object supplied to it.

In multithreaded program, the main thread must be the last thread to finish running. If the main thread finishes before a child thread has completed, then the Java run-time system may "hang". One simple method to finish child thread before the main thread is to use sleep method in the main thread.

10.4.3 Stopping and Blocking Thread

Sometimes, suspending execution of thread is useful. For example, a separated thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whenever we want to stop a thread from running further, we may do so by calling its stop method.

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

Thread automatically stops when it reaches the end of method. The stop( ) method may be used when the premature death of a thread is desired. The thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using:

```
sleep( )
suspend( )
wait( )
```

The thread will return to the runnable state when the specified time is elapsed in case of sleep( ). The resume( ) method is invoked in the case of suspend( ), and the notify( ) method is called in the case of wait( ).

10.4.4 Creating Multiple Threads

Using IsAlive( ) and Join( )

You can have threads communicate with each other through shared data and by using thread control methods e.g. suspend( ), resume( ) etc. All threads in the same program share the same memory space. If the reference to an object is visible to different threads, or explicitly passed to different threads, these threads share access to the data member of that object.

In multithreaded application, the main thread must be the last thread to finish. This can be accomplished by calling sleep( ) within main( ) with long enough delay to ensure that all child threads terminate prior to the main thread.

The threads communicate by waiting for each other. For example, the join( ) method can be used for the caller thread to wait for the completion of the called thread. Also, a thread can suspend itself and wait at some point using suspend( ) method; another thread can wake it up through the waiting thread's resume( ) method, and both threads can run concurrently.

One thread can know when another thread has ended using isAlive( ) method which returns true if the thread upon which it is called is still running. If the thread is alive, it does not mean it is running – it is in runnable state. Another method used for this purpose is join( ). This method waits until the thread on which it is called terminates. join( ) also allows you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

```
class Nthread implements runnable
{
  string name;
  Thread t;
  NThread (String threadname)
  {
      name = threadname;
      t = new Thread (this, name);
      System.out.println ("New Thread" + t);
      t.start( );
  }
public void run( )
  {
      try
      {
        for (int i = 5; i > 0; i - - )
        {
        System.out.println (name +" : " + i);
        Thread.sleep (1000);
        }
      }
      catch (InterruptedException e)
```

```
                {
                System.out.println (name + "interrupted");
                }
                System.out.println (name + "exiting");
            }
        }

        class DemoJoin
        {
            public static void main( )
            {
                Nthread ob1 = new Nthread ("One");
                Nthread ob2 = new Nthread ("Two");
                Nthread ob3 = new Nthread ("Three");
                System.out.println ("Thread one is alive" +
                        ob1.t.isAlive( );
                System.out.println ("Thread Two is alive" +
                        ob2.t.isAlive( );
            System.out.println ("Thread Three is alive"
                    + ob3.t.isAlive( );
        try
                {
            System.out.println ("waiting for thread to finish");
                    ob1.t.join( );
                    ob2.t.join( );
                ob3.t.join( );
        }
        catch (InterruptedException e)
        {
            System.out.println ("Main thread Interrupted");
        }
            System.out.println ("Thread one is alive" + ob1.t. isAlive( );
            System.out.println ("Thread Two is alive" + ob2.t. isAlive( );
            System.out.println ("Thread Three is alive" +
                    ob3.t. isAlive( );
            System.out.println ("Main thread exiting");
        }
        }
```

Sample Output:

```
New thread : Thread [One, 5, main]
New thread : Thread [Two, 5, main]
New thread : Thread [Three, 5, main]
Thread Three is alive : true
waiting for thread to finish
One : 5
Two : 5
Three : 5
One : 4
```

```
Two : 4
Three : 4
One : 3
Two : 3
Three : 3
One : 2
Two : 2
Three : 2
One : 1
Two : 1
Three : 1
Two exiting
Three exiting
Thread One is alive : False
Thread Two is alive : False
Thread Three is alive : False
Main Thread exiting
```

Thread Priorities

Priorities are the way to make sure that important or time-critical threads are executed frequently or immediately. Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For e.g., how an OS implements multitasking). A higher priority thread can also preempt a lower priority one when higher priority thread resumes from sleep. Use setPriority( ) method to set a thread's priority.

```
final void setPriority (int level)
```

Here, level specifies the new priority setting for calling thread. The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. If the value to be set is outside the legal range an IllegalArgumentException will be thrown. To return a thread to default priority, specify NORM_PRIORITY. Using getPriority( ) method you can obtain the current priority settings.

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

```
    {
   System.out.println ("thread B started");
   for (int = 1; j < = 4; j ++)
   System.out.println ("\t From Thread B : j = " + j);
   }
     System.out.println ("Exit from B");
   }
   class C extends Thread
   {
     public void run( )
     {
     System.out.println ("thread c started");
     for (int k = 1; k < = 4; k++)
     {
     System.out.println ("\t from Thread C: k = "+k);                        }
          System.out.println ("Exit form C");
     }
   }
   class ThreadPriority
   {
     public static void main( )
 {
   A threadA = new A( );
     B threadB = new B( );
     C threadC = new C( );
     threadC. setPriority (Thread.MAX_PRIORITY);
     threadB. setPriority (Thread.get_Priority( )+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");
     }
   }
```

The above program illustrates the effect of assigning higher priority to a thread. Note that although the thread A started first, the higher priority thread B has preempted it and started printing the output first. Immediately, the thread C that has been assigned the highest priority takes control over the other two threads. The thread A is the last to complete.

10.5 DEADLOCK-CONTROL ON THREADS

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 a deadlock. For example, assume that the thread 'A' must access 'MethodA' before it can release 'MethodB', but the thread 'B' cannot release 'MethodA' until it gets hold of MethodB. Because these create mutually exclusive conditions, a deadlock occurs. Suppose, one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

10.5.1 Suspending, Resuming and Stopping Threads

The suspend( ) method of the thread class is deprecated in Java2 because suspend can sometimes cause serious system failure. Assume that a thread obtains locks on important resource and if that thread is suspended at that point, those locks are not relinquished. Other threads waiting for these resources can be deadlocked.

```
    resume( ) method cannot be used without suspend( ).
```

The stop( ) method is also deprecated in Java2 because arbitrarily stopping a thread can leave an object in a damaged state that might then cause unpredictable results.

In Java2 we can suspend, resume or stop the thread by using wait( ), notify( ) methods.

```
        class NThread implements Runnable
        {
          String name;
          Thread t;
          boolean suspendFlag;

        NThread (String threadname)
{

        name = threadname;
          t = new Thread (this, name);
          System.out.println ("New thread" + t);
          suspendFlag = false;
          t.start( );
        }
        public void run( ) {
        try
        {
          for (int i = 15; i > 0; i - -)
          {
              System.our.println (name + ": " +i);
              Thread.sleep (200);
              synchronized (this)
                {
                while (suspendFlag)
                {
                      wait( );
                }
```

```
         }
        }
        catch (InterruptedException e)
          {
          System.out.println (name + "interrupted");
          }
          System.out.println (name + "exiting");
        }
        void mysuspend( )
        {
          suspendFlag = true;
        }
        synchronized void myresume( )
        {
          suspendFlag = false;
          notify( );
        }
        }
        class SuspendResume
        {
          public static void main( )
          {
              NThread ob1 = new NThread ("One");
              NThread ob2 = new NThread ("Two");
              try
              {
              Thread.sleep (1000);
              ob1.mysuspend( );
          System.out.println ("Suspending thread One");
          Thread.sleep (1000);
          ob1.myresume( );
          System.out.println ("Resuming thread One");
        ob2.mysuspend( );
          System.out.println ("Suspending thread Two");
              Thread.sleep (1000);
              ob2.myresume( );
              System.out.println ("Resuming thread Two");
              }
              catch (InterruptedException e)
              {
          System.our.println ("Main thread Interrupted");
              }
          try
        {
          System.out.println ("Waiting for thread to finish");
```

```
  ob1.t.join( );
  ob2.t.join( );
}
catch (InterruptedException e)
{
  System.out.println ("Main thread Interrupted");
}
  System.out.println ("Main thread exiting");
}
}
```

```
New thread : Thread [One, 5, main]
One : 15
New thread : Thread [Two 5, main]
Two : 15
One : 14
Two : 14
One : 13
Two : 13
One : 12
Two : 12
One : 11
Two : 11
Suspending thread One
Two : 10
Two : 9
Two : 8
Two : 7
Two : 6
Resuming thread One
Suspending thread Two
One : 10
One : 9
One : 8
One : 7
One : 6
Resuming thread Two
Waiting for threads to finish
Two : 5
One : 5
Two : 4
One : 4
Two : 3
One : 3
Two : 2
One : 2
Two : 1
```

```
One : 1
Two exiting
One exiting
Main thread exiting
```

Check Your Progress

State whether the following statements are true or false:

1. Switching between threads is not carried out automatically by JVM.
2. The Java runtime system does not depends on threads for many things and all class libraries are designed with multithreading in mind.
3. Operating systems usually run many applications such as a web browser and a word processing program at the same time.
4. To implement runnable, a class needs to only implement a single method called run( ).
5. The suspend( ) method of the thread class is not deprecated in Java2 because suspend can sometimes cause serious system failure.

10.6 LET US SUM UP

Multithreading allows a running program to perform several tasks, apparently, at the same time. Java uses threads to enable the entire environment to be asynchronous. It makes maximum use of CPU because idle time can be kept to minimum. Any single path of execution is called thread. The path is defined by elements such as stack (for handling calls to other methods with the code) and set of registers to store temporary memory. These elements are collectively known as context. Word processing program handles a print request in a separate thread. Many users find it convenient to continue working in word-processing application even after submitting print request. Sometimes, suspending execution of thread is useful. For example, a separated thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher priority threads get more CPU time than lower-priority threads. 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.

10.7 KEYWORDS

Multithreading: It allows a running program to perform several tasks, apparently, at the same time.

Threads: They subdivide the runtime behavior of a program into separate, independently running subtasks.

Priorities: It is the way to make sure that important or time-critical threads are executed frequently or immediately.

10.8 QUESTIONS FOR DISCUSSION

1. What is thread? How do we start it? What are the two methods by which we may stop threads?
2. Describe the complete life cycle of a thread.
3. How do we set priorities for threads?
4. When we implement the runnable interface, we must define which of the following methods?
   (a) start()
   (b) init()
   (c) run()
   (f) main()
   (g) runnable()
5. What will be the output of the following code?

```
class t1 extends Thread
{
            public void run( )
            {
            System.out.prinln ("hello");
            suspend( );
            resume( );
            System.out.println ("Bye");
            }
}
class test
{
            public static void main (String args[ ])
            {
            t1 T = new t1( );
T.start( );
            }
}
```

6. If you create a class that implements runnable, you have already written its run() method. Which of these start() methods should you use to call run method?

```
          public void start( )
  {
  new Thread (this).start( );
  }
        public void start( )
```

```
{
Thread myT = new Thread( );
myT.start( );
}
      public void start( )
{
Thread myT = new Thread (this);
myT.run( );
}
```

7. Which of the following statement is correct for the line given below?

   Thread myT = new Thread();

   (a) The myT is in runnable state.

   (b) The thread myT has the NORM_PRIORITY.

   (c) If myT.start() is called, the run method in the calling class will be executed.

   (d) If myT.start() is called, the run method in the thread class will be executed.

8. The object class has a wait method that is used to coordinate access to an object by multiple threads. Which of the following statements about the wait method are true?

   (a) The wait method is an instance method of the object class.

   (b) The wait() method is a static method of the object class.

   (c) To call wait, a thread must have a lock on the object involved.

   (d) An object can have only one thread in waiting state at a time.

Check Your Progress: Model Answers

1. False
2. False
3. True
4. True
5. False

10.9 SUGGESTED READINGS

E. Balaguruswami, *Programming with Java*, Tata McGraw-Hill.

Davis, Stephen R., *Learn Java Now*, Microsoft Press.

Naughton, Patrick, *The Java Hand Book*, OSborne McGraw- Hill.

Sams.net, *Java unleased*.

Herbert Schildt, *The Complete Reference Java 2*, Tata McGraw-Hill.

UNIT V

LESSON 11

INPUT-OUTPUT OPERATIONS

CONTENTS

11.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Define reading characters
- Explain sentences and writing console
- Describe file processing in java
- Explain copying of files

11.1 INTRODUCTION

Java programs can be of both types – text based console programs and also applets that rely upon Java's Abstract Windows Toolkit (AWT) for interaction with the user. Input can be from the console or can also be from the AWT such as textfield, button etc. Output can be on the screen, file or printer.

11.2 STREAMS (BYTE AND CHARACTER)

Java program performs Input/Output through streams. A stream is an abstraction that either produces or consumes information. It is linked to a physical device by the Java Input/Output systems. Java implements streams within class hierarchies defined in java.io package.

Java defines two types of streams - byte and character. Byte streams provide a convenient means for handling input and output of bytes. They are used, for example, when reading or writing binary data. Character streams provide a convenient means for handling input and output of characters.

11.2.1 Reading Character

In Java, the only way to perform console input is to use a byte stream and an older code that uses this approach persists. The preferred method of reading console input for Java2 is to use a character oriented stream.

In Java, console input is accomplished by reading from system.in. To obtain a character based stream that is attached to the console, you wrap System.in in a Buffered Reader object to create a character stream.

Input Reader is the stream that is linked to the instance of BufferedReader that is being created. Reader is an abstract class, one of its concrete subclasses is InputStream-Reader, which converts bytes to characters. To obtain an InputStreamReader that is linked to System.in refers to an object of type InputStream it can be used for inputStream. The following line of code creates a BufferedReader that is connected to the keyboard:

```
BufferedReader br = new BufferedReader

(new InputStreamReader(System.in));
```

To read a character we use BufferedReader read( ) ® int read( ) throws IOException. Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns -1 when the end of the stream is encountered.

The following program demonstrates read( ):

```
import java.io.*;
class BRead
{
  public static void main string args[ ])
  throws IOException
  {
  char C;
  BufferedReader  br  =  new  BufferedReader  (new  InputStreamReader
System.in));
      System.out.println ("Enter Characters, 'q' to quit");
      do
      {
        c = (char) br read( );
System.out.println (c);
      }
```

```
            while (c ! = 'q')
        }
    }
    READING STRINGS
    import java.io.*;
    class BRreadlines
    {
      public static void main (String args[ ])
      throws IOException
      {
            BufferedReader br = new BufferedReader (new InputStreamReader
            (System.in));
            String str;
            System.out.println ("Enter lines of text.");
            System.out.println ("Enter stop" to Quit.");
            do
            {
              str = br.readLine( );
              System.out.println (str)
            }
            while (! str.equals ("stop"));
      }
    }
```

11.2.2 Writing Console Output

Console output is most easily accomplished using print( ) and println( ). These methods are defined by the class PrintStream. Even though System.out is a byte stream, using it for simple program output is still acceptable.

Because PrintStream is an output stream derived from OutputStream, it also implements the low level method write( ). Thus, write( ) can be used to write to the console. The simplest form of write( ) defined by PrintStream is:

```
        void write (inbyteval) throws IOException.
        // Demonstrate System.out.write( )
        class WriteDemo
        {
    public static void main (String args[ ])
        {
              int b;
              b = 'A';
              System.out.write (b);
              System.out.write ('\n');
          }
        }
```

You will not often use write( ) to perform write because print( ) and println( ) are substantially easier to use.

11.2.3 Reading and Writing Files

Java provides a number of classes and methods that allow you to read and write files. In Java all files are byte oriented and Java provides methods to read and write bytes from and to a file. However, Java allows you to wrap a byte oriented file stream within a character based object.

Two of the most often used stream classes are FileInputStream and FileOutputStream, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. While both classes support additional overridden constructors, the following are the forms:

- FileInputStream (String filename) throws FileNotFoundException
- FileOutputStream (String filename) throws FileNotFoundException

When you are done with a file you should close it by calling close( ). It is defined both in FileOutputStream and FileInputStream.

```
void close( ) throws IOException
```

To read from a file you can use a version of read( ) that is defined in FileInputStream.

```
int read( ) throws IOException.
```

Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. read( ) returns -1 when the end of the file is encountered. It can throw an IOException.

```
/ * Display a text file
import java.io.*;
class showfile
{
 public static void main (String s[ ]) throws IOException
 {
     int i;
     FileInputStream fis;
     try
     {
       fis = new FileInputStream (s[ ]);
     }
     catch (FileNotFoundException e)
     {
     System.out.println ("File not found" + e);
       return;
     }
     catch (ArrayIndexOutOfBoundsException e)
     { System.out.println ("Usage : Showfile");
     return;

 }
 // read characters until EOF is encountered
 do
 {
     i = fis.read( );
```

```
            if (i ! = -1) System.out.println ((char);););
        }
        while (i ! = -1);
        fis.close( );
        }
        }
```

11.3 FILE PROCESSING

File class in Java does not specify how information is retrieved from or stored in files. The File class is used to represent the name of a file or a set of files (a directory). A File object is used to obtain or manipulate the information associated with a disk file such as the permission, time, date and directory path. You can also get some more attributes for the file such as whether the file exists, whether the file is writeable or not and so on. The constructors used to create File objects are:

- File (String path of the Directory)

  ```
  File f = new File ("/") ; // f refers to root directory
  ```

- File (String dirpath, String filename)

  ```
  File  f  =  new  File  ("/",   "etc/passwd");  //  f  refers  to  file
  "/etc/passwd"
  File f = new File ("config.sys"); // f refers to config.sys in the
  current directory
  ```

- File (File dirobj, String filename)

  ```
  File f1 = new File (f, "a.dat"); // f refers to a.dat in the root
  directory
  ```

Some common methods used in File are (Table 11.1):

Table 11.1

Method	Description
boolean isFile( )	Returns true if the invoking object is a file. This method returns false for some special files such as device drivers and pipes.
boolean isAbsolute( )	Returns true if the file has an absolute path and false if path is relative.
boolean rename To (File newname)	It returns true if filename specified by "newname" becomes the new name of invoking File object and false if file cannot be renamed.
boolean delete( )	Returns true if it deletes the file and false if the file cannot be removed. It can also be used to delete directory provided the directory is empty.
String getName( )	Returns name of the file without path.
String getpath( )	Returns full path including filename.
String getParent( )	Returns full path of parent directory.
boolean exists( )	Returns true if file exists and false if file does not exist on the disk.
boolean canWrite( )	Returns true if file is writable and false if it is not writable.
boolean canRead( )	Returns true if file is readable and false if it is not readable.
boolean isDirectory( )	Returns true if the invoking object is a directory and false if it is a file.
int length( )	Returns file size in bytes.
boolean isHidden	Returns true if file is hidden and false if it is not hidden.

Contd....