Chapter 4- Exceptions and Multithreading

Marks Allotted: 16

Lectures: 08

Semester Pattern:

Exam:	Winter 2008	Summer 2009
Marks:	28	20

Yearly Pattern (JPR-IF 1526):

Exam:	S'04	W'04	S'05	W'05	S'06	W'06	S'07
Marks:	24	08	24	20	20	28	16

Syllabus Contents:

4.1 Multi Threading

Creating Thread, Extending a thread class, Stopping and Blocking a thread, Life cycle of thread, Using thread method, Thread exceptions, Thread priority, Synchronization, Implementing a 'Runnable' Interface.

4.2 Managing Errors and Exceptions

Types of errors, Exception, Multiple catch statement, using finally statement, Using Exception for Debugging.

Exceptions

Exception is a condition that is caused by run-time error in the program. In computer programming languages that do not support exception handling, errors must be checked and handled manually through the use of error codes, and so on. But this approach is cumbersome as well as troublesome. Java's exception handling avoids these problems and, in the process and brings run-time error management into the object-oriented world. C++ programming language supports exception handling but Java has enhanced some of its features.

As Java is strictly object oriented, an exception is also an object that describes an exceptional (that is, error) condition that has occurred in a piece of source code. When an exceptional condition arises in program, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Exceptions can be generated by the Java run-time system, or they can be manually generated by our code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

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

- Find the problem
- Inform about error has occurred
- Receive the error information
- Take the corrective action

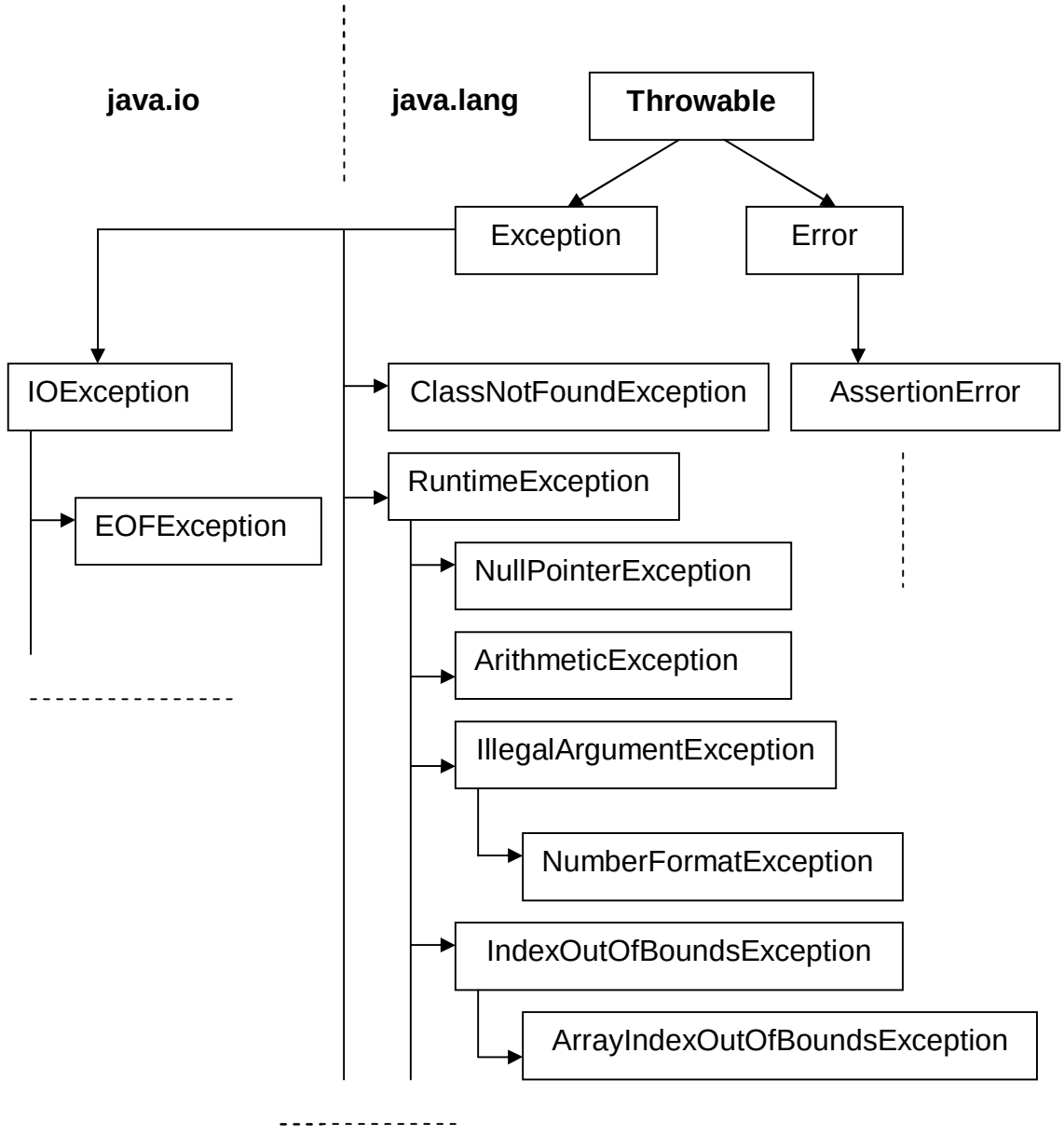
Java's exception handling mechanism provides every step to be implemented.

Types of Exceptions

All exception types in Java are subclasses of the built-in abstract class java.lang.Throwable. Thus, Throwable is at the top of the exception class hierarchy. There are two subclasses that partition exceptions into two distinct categories. That is Exception and Error. First category is Exception. This class is used for exceptional conditions that user programs should catch. This is also the class that we will subclass to create our own exception types.

There is an important subclass of Exception, called RuntimeException. Exceptions of this type are automatically defined for the programs that we write and include things such as division by zero and invalid array indexing. Second category of exceptions is Error, which defines exceptions that are not expected to be caught under normal circumstances by the program. Exceptions of type Error are used by the

Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.



Partial exception class hierarchy

Exception

The class Exception represents exceptions that a program would want to be made aware of during execution. Its subclass RuntimeException represents many common programming errors that manifest at runtime. Other subclasses of the Exception class define other categories of exceptions, for example, I/O-related exceptions (IOException, FileNotFoundException, EOFException) and GUI-related exceptions (AWTException). Remember here IOException is subclass of Exception which is defined in package java.io.

RuntimeException

Runtime exceptions, like out-of-bound array indices (ArrayIndexOutOfBounds Exception), uninitialized references (NullPointerException), illegal casting of references (ClassCastException), illegal parameters (IllegalArgumentException), division by zero (ArithmeticException) and number format problems (NumberFormatException) are all subclasses of the java.lang.RuntimeException class, which is a subclass of the Exception class. As these runtime exceptions are usually caused by program bugs that should not occur in the first place. It is more appropriate to treat them as faults in the program design, rather than merely catching them during program execution.

Error

The subclass AssertionError of the java.lang.Error class is used by the Java assertion facility. Other subclasses of the java.lang.Error class define exceptions that indicate class linkage (LinkageError), thread (ThreadDeath), and virtual machine (VirtualMachineError) related problems. These are invariably never explicitly caught and are usually irrecoverable.

Uncaught Exceptions

Before learning how to handle exceptions in the program, it is interesting to see what happens when we don't handle them. Observe following small program 5.1 that includes an expression that intentionally causes a divide-by-zero error.

```
// Uncaught exception
  class Uncaught
  {
      public static void main(String args[])
      {
           int x = 23;
           int y = 0;
           System.out.println(x/y);
      }
  }
```

An exception is created intentionally

This program will be compiled successfully. But when we tried to run the program using Java interpreter, the Java run-time system detects an attempt to divide by zero. So, it constructs a new exception object and then throws this exception. This causes the execution of **Uncaught** to stop, because once an exception has been thrown, it must be caught by

an exception handler and take action immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java's run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates our program.

The following output will be generated:

```
Exception in thread "main" java.lang.ArithmeticException: / by
zero at Uncaught.main(Uncaught.java:8)
```

Note that this output gives the whole description of the exception. It has displayed class name Uncaught, method name main and source file name Uncaught,java with line number where the exception has occurred i.e. line number 8. It has also displayed the type of the exception thrown is a subclass of Exception called ArithmeticException, which more specifically describes what type of error happened. "/ by zero" is the description of the ArithmeticException occurred in the program. So it will be very easy to locate and remove the exception. The message displayed when exception occurred is called as stack trace. This stack trace always shows a sequence of method invocation to reach up the exception. Program will clear this concept.

```
// Uncaught exception in method calls
  class UncaughtEx
  {
      static void callme()
      {
           int x = 23;
           int y = 0;
           System.out.println(x/y);
      }
      public static void main(String args[])
      {
           UncaughtEx.callme();
      }
  }
```

Uncaught exceptions in method calls

When we execute this program the stack trace will be displayed as,

```
Exception in thread "main" java.lang.ArithmeticException: / by
zero    at UncaughtEx.callme(UncaughtEx.java:8)
        at UncaughtEx.main(UncaughtEx.java:12)
```

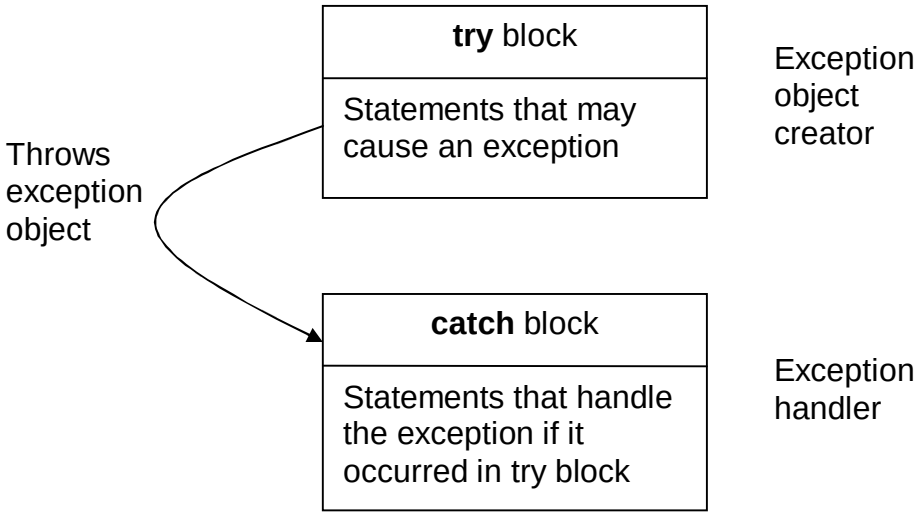
This is description of the exception that is actual exception is occurred at line number 8 in method callme( ) which has been called by main( ) at line number 12 in the program.

Exception handling using try and catch

The default exception handler provided by the Java run-time system is useful for debugging; we will usually want to handle an exception by ourselves. It provides two benefits. First, it allows us to fix the errors. Second, it prevents the program from automatically terminating. In order to take these two benefits Java has provided the mechanism of try-catch blocks. Its general form is as below:

```
try
{
     // block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
     // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
     // exception handler for ExceptionType2
}
//.......
```

Here, ExceptionType is the type of exception that has occurred and exOb. The keyword try is used to preface a block of code that is likely to cause an error condition and throw an exception. The catch block catches the exception thrown by the try block and handles it appropriately. The catch block is added immediately after the try block. Fig. 5.2 shows this exception handling mechanism.



Exception handling mechanism

```
// an example of simple try-catch
  class ArException
```

```
{
    public static void main(String args[])
    {
        int x = 56;
        int y = 0;
        try
        {
           int z = x/y;     //statement1
           System.out.println("Value: "+z);
        }
        catch(ArithmeticException e)
        {
           System.out.println("DIVISION BY ZERO");
        }
        System.out.println("End of program...");
    }
}
```

Simple try-catch statements

Output:

```
DIVISION BY ZERO
End of program...
```

Observe the program; here in the statement1 an attempt of divide by zero is made. So it creates an exception object of type ArithmeticException and throws out of it. That's why, the statement after statement1 will not be executed. The control will directly get transferred to catch statement. Thus line "DIVISION BY ZERO" is printed on the output. Program will not terminate here. Last statement of the program is also getting executed.

A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately after try statement. A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements, described next). The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.) You cannot use try on a single statement. Remember, the variables declared inside the try block are having local scope or block scope for the particular block. They can not be use outside of that block.

Displaying the description of exception

The class Throwable provides following common methods to query the exception. So they can be used along with the Exception class' object.

```
String getMessage()
```

It returns the detail message of the exception.

```
void printStackTrace()
```

It prints the stack trace on the standard error stream. The stack trace comprises the method invocation sequence on the runtime stack when the exception was thrown. It gives the same output as when exception is occurred but here the program is not terminated.

```
String toString()
```

It returns a short description of the exception, which typically comprises the class name of the exception together with the string returned by the getMessage() method.

Instead of these methods directly the object can also be referred to print the description of the exception. The program 5.4 illustrates the use of all these methods.

```
// Description of exception
  class ExMethods
  {
      public static void main(String args[])
      {
          int x = 56;
          int y = 0;
          try
          {
             int z = x/y;
             System.out.println("Value: "+z);
          }
          catch(ArithmeticException e)
          {
             System.out.println("getMessage:-");
             System.out.println(e.getMessage());

             System.out.println("\nprintStackTrace:-");
             e.printStackTrace();

             System.out.println("\ntoString:-");
             System.out.println(e.toString());

             System.out.println("\nReferring object:-");
             System.out.println(e);
          }
          System.out.println("\nEnd of program...");
      }
  }
```

Displaying description of exception

Output:

```
getMessage:-
/ by zero
```

```
printStackTrace:-
java.lang.ArithmeticException: / by zero
        at ExMethods.main(ExMethods.java:10)

toString:-
java.lang.ArithmeticException: / by zero

Referring object:-
java.lang.ArithmeticException: / by zero

End of program...
```

Multiple catch statements

In some cases, more than one exception may occur in a single program. In order to handle this type of situation, we can specify two or more catch statements in a program. Each 'catch' will catch a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types:

```
// Using multiple catch statements
  import java.util.Scanner;
  class MultiCatch
  {
      public static void main(String args[])
      {
          int x,z;
          Scanner in = new Scanner(System.in);
          System.out.print("Enter number : ");
          x = in.nextInt();
          try
          {
             z = 50 / x;      //statement1
             System.out.println("Division: "+z);
             short arr[] = {5,7,8};
             arr[10] = 120;   //statement2
             System.out.println("Try end...");
          }
          catch(ArithmeticException e)
          {
             System.out.println("Division by zero");
          }
          catch(ArrayIndexOutOfBoundsException e)
          {
             System.out.println("Array indexing wrong");
          }
```

```
            System.out.println("Program end...");
        }
    }
```

multiple catch statements

Two Outputs:

```
Enter number : 0
Division by zero
Program end...

Enter number : 5
Division: 10
Array indexing wrong
Program end...
```

Program illustrates the use of multiple catch statements with single try. We have assume that the code written in the try block will cause either divide by zero error or array index will go out of bounds. If any one of these exceptions is occurred it will be caught appropriate catch statement and action will be taken. Depending upon the type of exception object thrown only one of the catch statements will be executed. If the exception thrown does not match with the exception object written in catch block, program will be terminated. In program above of this we have taken input from keyboard. If value of that input i.e. x is 0 then the divide by zero exception will occur at statement1 else array index out of bounds exception will occur at statement2. These exceptions will be caught by appropriate catch clause. We can write any number of catches for a try. But they must be unique and must follow the rule given below.

```
try
{
   z = 50 / x;      //statement1
   System.out.println("Division: "+z);
   short arr[] = {5,7,8};
   arr[10] = 120;   //statement2
   System.out.println("Try end...");
}
catch(Exception e)
{
   System.out.println("Division by zero");
}
catch(ArrayIndexOutOfBoundsException e) //statement3
{
   System.out.println("Array indexing wrong");
}
```

Observe the program code given above. It contains error. If we compile it we will get the output error on statement3 as,

exception java.lang.ArrayIndexOutOfBoundsException has already been caught

When we use multiple catch statements, it is important to remember that exception subclasses must come before any of their super classes. This is because a catch statement that uses a super class will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its super class. Further, in Java, unreachable code is an error. In above code ArrayIndexOutOfBoundsException is subclass of Exception. In order to fix the problem in above code, we just have to reverse the order of the catch statements.

Nesting of try statements

The try statement can be nested. That is, a try statement can be used inside the block of another try. Each time when a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match of exception. This continues until one of the catch statements succeeds, or until all the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

Program illustrates the use of nested try statements.

```
// Nested try blocks
  import java.util.Scanner;
  class NestTry
  {
      public static void main(String args[])
      {
          int x,z;
          Scanner in = new Scanner(System.in);
          System.out.print("Enter number : ");
          x = in.nextInt();
          try
          {
             z = 82 / x;      //statement1
             System.out.println("Division: "+z);
             try
             {
                int a = 100 / (x-1);  //statement2
                short arr[] = {15};
                arr[10] = 25;      //statement3
                System.out.println("Inner try end...");
             }
             catch(ArrayIndexOutOfBoundsException e) //1
             {
```

```
                System.out.println("Array indexing wrong");
            }
            System.out.println("Outer try end...");
        }
        catch(ArithmeticException e) //2
        {
            System.out.println("Division by zero");
        }
        System.out.println("Program end...");
    }
}
```

Nesting of try statements

Three outputs:

```
Enter number : 0
Division by zero
Program end...

Enter number : 1
Division: 82
Division by zero
Program end...

Enter number : 2
Division: 41
Array indexing wrong
Outer try end...
Program end...
```

Program has nested two try statements in each other. We have taken input as a number from keyboard and stored in into variable x. When value of x is inputted as 0, statement1 will cause an exception “Divide by zero”. So program control will directly transfer to catch statement no.2 to take the action. When value of x inputted is 1. The outer try will work properly. In inner try, the statement2 will cause the exception of “Divide by zero”. In this case, the program control will get directly transferred to inner try’s catch block i.e. catch clause no. 1. But the exception object will not match. So it will check for outer try’s catch clause. Now, the match is found. That’s why, we got the output no.2. In the third case, when the input is any number other than 0 and 1, statement3 will cause an exception “Array index out of bounds”. This will be caught by inner try’s catch clause.

That is, the outer try’s catch clause will catch all the exception occurred inside it. Remember, if the exception of “array index out of bounds” is occurred in outer try then this will not be caught by inner try’s catch clause.

```
// Nested try blocks with method
  import java.util.Scanner;
```

```
class NestTryMethod
{
    public static void myMeth()
    {
        try
        {
           int arr[] = new int[-2];   //statement1
           arr[4] = 10;
        }
        catch(ArithmeticException e)  //1
        {
           System.out.println("Exception:"+e);
        }
    }
    public static void main(String args[])
    {
        int a,b;
        Scanner in = new Scanner(System.in);
        System.out.print("Enter the number : ");
        a = in.nextInt();
        try
        {
           b = 40 / a;                //statement2
           System.out.println("Division: "+b);
           NestTryMethod.myMeth();    //statement3
        }
        catch(NegativeArraySizeException e) //2
        {
           System.out.println("Exception:"+e);
        }
        catch(ArithmeticException e)       //3
        {
           System.out.println("Exception:"+e);
        }
        System.out.println("Program end...");
    }
}
```

Nested try blocks with methods

Two outputs:

```
Enter the number : 0
Exception:java.lang.ArithmeticException: / by zero
Program end...

Enter the number : 1
Division: 40
Exception:java.lang.NegativeArraySizeException
Program end...
```

The program introduces a method in program that causes the exception. Observe the outputs. When the entered number from keyboard is 1, statement2 will be executed properly and the method is called at statement3. In statement1 the exception of “negative array size” is occurred. Here, the match will not found at catch clause 1. The program control will get returned and it will transfer to catch clause 2. Now the match will found. So the action will be taken.

The finally block

The finally is a block of code after try/catch block that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether an exception is thrown or not. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.

If any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method. The finally clause is optional. But, each try statement requires at least one catch or a finally clause.

The try-catch-finally of try-finally construct can be created as,

```
try
{
     .......
}
catch(....) //multiple catch are allowed
{
     .......
}
finally
{
     .......
}
```

or

```
try
{
     .......
}
finally
{
     .......
}
```

Program below illustrates the use of finally block.

```
//Use of finally statement
  class FinallyClause
  {
      public static void main(String args[])
      {
          try
          {
             //int val = 0;             //statement1
             //int m = 100 / val;       //statement2
             int []x = new int[-5];     //statement3
             System.out.println("No output");
          }
          catch(ArithmeticException e)
          {
             System.out.println("Exception: "+e);
          }
          finally
          {
             System.out.println("Program end");
             System.out.println("Bye bye...");
          }
      }
  }
```

Using the finally block

Output:

```
Program end
Bye bye...
Exception in thread "main"
java.lang.NegativeArraySizeException
        at FinallyClause.main(FinallyClause.java:10)
```

In program, the exception of “negative array size” has occurred at ststement3. But the catch statement will not catch it as only ArithmeticException is mentioned there. So the program will be terminated. But before its termination and printing the stack trace, the finally block is executed.

Now, if we removed the comments from statement1 and statement2, the output will be:

```
Exception: java.lang.ArithmeticException: / by zero
Program end
Bye bye...
```

Here the exception of “Divide by zero” will be occurred at statement2. This will be caught by catch clause. So catch block is executed first and then finally block is executed. That is, the try-catch-

finally are executed in sequence. If a finally block is associated with only a try, the finally block will be executed upon conclusion of the 'try'.

The throw statement

If we want to throw the exceptions by our own, the throw statement can be used. That is, it will force the Java run-time system to throw an exception. The general form of throw statement is:

```
throw Throwable-Instance;
```

Here, the **Throwable-Instance** must be an object of type Throwable or a subclass of Throwable. Primitive data types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.

There are two ways that we can obtain a Throwable object. First is using a parameter into a catch clause and second is creating it with a new operator. The flow of execution stops immediately after the throw statement. So any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler terminates the program and prints the stack trace.

```
// Demonstration of throw clause
  class ThrowDemo
  {
      public static void main(String args[])
      {
         int x = 10, y = 20;
         int z;
         z = x + y;
         try
         {
            throw new ArithmeticException();
         }
         catch(Exception e)
         {
            System.out.println("Exception caught");
            System.out.println("Addition: "+z);
         }
      }
  }
```

Demonstration of throw clause

Output:

```
Exception caught
```

```
Addition: 30
```

In program, an ArithmeticException has been thrown by throw statement and it is caught by the catch clause. It the exception thrown by throw and caught by catch does not matches the program will get terminated before printing any output. Just change the catch clause as,

```
catch(NegativeArraySizeException e)
```

We will get the output:

```
Exception in thread "main" java.lang.ArithmeticException
        at ThrowDemo.main(ThrowDemo.java:11)
```

We can give the description to the exception class also. Such as,

```
throw new ArithmeticException("Divide Error");
```

Just make the changes in the program as,

```
try
{
     throw new ArithmeticException("Divide Error");
}
catch(Exception e)
{
     System.out.println("Exception:"+e);
     System.out.println("Addition: "+z);
}
```

We will get the following output:

```
Exception:java.lang.ArithmeticException: Divide Error
Addition: 30
```

All of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to print( ) or println( ). It can also be obtained by a calling method getMessage( ), which is defined by Throwable.

The throws clause

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can protect themselves against that exception. We do this by including a

throws clause in front of the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will occur.

The general form of writing a throws clause is,

```
data-type method-name(parameter-list) throws exception-list
{
     // body of method
}
```

Here the exception-list is the list of exceptions that the method might throw separated by comma. For many classes and methods it is necessary to throw the exception by the method which is using it. For example, while using DataInputStream it is necessary to throw the IOException as well as for many methods of multithreading it is necessary to throw InterruptedException.

Just compile the following code.

```
class ThrowsDemo
{
    public static void main(String args[])
    {
        throw new ClassNotFoundException();
    }
}
```

This will output as,

```
ThrowsDemo.java:6: unreported exception
java.lang.ClassNotFoundException; must be caught or declared
to be thrown
          throw new ClassNotFoundException();
                ^
```

The solution for this is either to enclose the statement containing throw in try-catch block or use throws clause as given below:

```
class ThrowsDemo
  {
     public static void main(String args[])
         throws ClassNotFoundException
     {
         throw new ClassNotFoundException();
     }
  }
```

Here, the main( ) will throw the mentioned exception by itself. Because the ClassNotFoundException is neither of type Error nor RuntimeException. So it can be thrown using throws clause. Before the exception was occurred the other exceptions such as divide by zero, array index out of bounds or negative array size exception can not be written in throws clause as they are of type RuntimeException.

Built-in exception classes

In the library package java.lang, Java has defined several exception classes. A few have been used by the preceding examples. The most of these exceptions are subclasses of the standard class RuntimeException. Since java.lang is by default imported into all Java programs, most exceptions derived from RuntimeException are automatically available. They need not be included in any method's throws list. In Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in java.lang are listed in Table 1. Table 2 lists those exceptions defined by java.lang that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself. These are called checked exceptions. Java also defines several other types of exceptions that relate to its various class libraries.

Exception	**Meaning / When occurred**
ArithmeticException	Arithmetic error, such as divide by zero.
ArrayIndexOutOfBoundsException	Array index is out of bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid casting of classes
IllegalArgumentException	Illegal argument used to invoke a method
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out of bounds
NegativeArraySizeException	Array created with a negative size
NullPointerException	Invalid use of a null reference
NumberFormatException	Invalid conversion of a string to a numeric format
SecurityException	Attempt to violate security
StringIndexOutOfBounds Exception	Attempt to index outside the bounds of a string

UnsupportedOperationException	An unsupported operation was encountered

Table 1 checked exceptions

Exception	Meaning / When occurred
ClassNotFoundException	Class not found
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface
IllegalAccessException	Access to a class is denied
InstantiationException	Attempt to create an object of an abstract class or interface
InterruptedException	One thread has been interrupted by another thread
IOException	I/O error has occurred
NoSuchFieldException	A requested field does not exist
NoSuchMethodException	A requested method does not exist

Table 2 unchecked exceptions

Creating our own exception

Java has defined a lot of Exception and Error classes for different conditions. But many times it is required in some conditions that we want to create your own exception types to handle situations specific to our applications. This is quite easy to do. We just have to define a subclass of Exception (which is a subclass of Throwable). Our subclasses don't need to actually implement anything. It is their existence in the type system that allows us to use them as exceptions.

The Exception class does not define any methods of its own. It inherits all the methods provided by class Throwable. Thus, all exceptions, including those that we create, have the methods defined by Throwable available to them. In order to create our own exception we need to derive our class from Exception.

```
// Creating our own exception.
  class NegativeOutputException extends Exception
  {
     private int det;
     NegativeOutputException(int a)
     {
        det = a;
     }
     public String toString()
     {
        return "NegativeOutputException["+det+"]";
     }
  }
```

```
class OwnException
{
    public static void main(String args[])
    {
        int x = Integer.parseInt(args[0]);
        int y = Integer.parseInt(args[1]);;
        int z;
        try
        {
           z = x * y;
           if(z<0)     //statement1
             throw new NegativeOutputException(z);
           System.out.println("Output: "+z);
        }
        catch (NegativeOutputException e)
        {
             System.out.println("Caught: "+e);
        }
    }
}
```

Creating our own exception

Outputs:

```
java  OwnException 4 8
Output: 32

java  OwnException 4 -3
Caught: NegativeOutputException[-12]

java  OwnException -4 -3
Output: 12
```

Observe the program. In order to create our own exception we have inherited our exception class from java.lang.Exception. Name of the defined exception is NegativeOutputException. The method toString( ) has been overridden to print our own custom message with exception class. The condition for the exception defined is that when the result of the expression is negative then exception must occur. It is defined in statement1. The input has taken from the command line. First argument is passed to x and second to y. First time and third time when both the arguments are positive/negative the result is positive so we will get the output properly. But when any one of the arguments is negative the result is also negative. The statement1 will be true. Now exception is thrown using the throw clause. This exception is caught by catch clause. If we do not catch it using catch the compile time error will occur.

Exceptions for debugging

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It can also be used to hide the error from rest of the program. We can avoid the termination of program by handling exceptions. It is important to think of try, throw, and catch as clean ways to handle errors and unusual boundary conditions in our program's logic. It is possible that programmer may misuse the technique for hiding errors. Exception handling may effectively used to locate and report the run-time errors. If we are too familiar with C programming language, we will get the true power of exception handling mechanism of Java.

Java's exception-handling statements should not be considered a general mechanism for non-local branching. If we do so, it will only confuse our code and make it hard to maintain.

Multithreading

Multithreading is one of the features of Java. It provides built-in support for multithreaded programming. Basically, it is not supported by most of the programming languages. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. That is a thread is a light-weight process. We can call multithreading is a specialized form of multitasking. Multitasking is supported by virtually all modern operating systems such as Windows, Linux and Solaris.

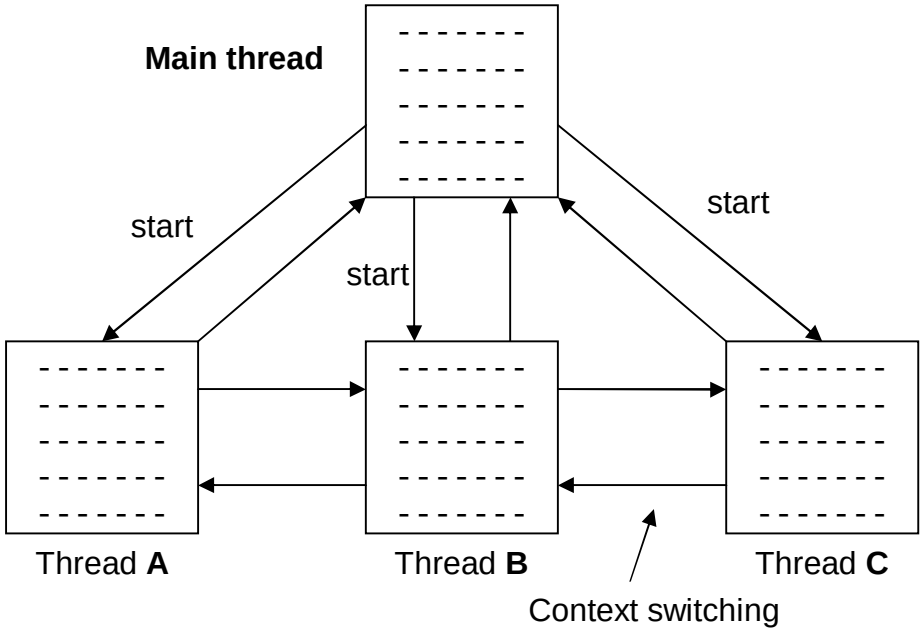
There are two kinds of multitasking i.e. process-based and thread-based multitasking. A process is a program that is executing. Thus, process-based multitasking is the feature that allows our computer to run two or more programs concurrently. For example, process-based multitasking enables us to run the Java compiler at the same time that we are using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. A thread is similar to a program that has single flow of control. It has beginning, the body and the end and executes the statements sequentially. For example, a text editor can format text at the same time that it is printing. These two actions are being performed by two separate threads. Thus, process-based multitasking deals with the "big picture," and thread-based multitasking handles the details.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Inter-process communication is expensive and limited. Context switching from one process to another is also costly. Threads are lightweight. They share the same address space and cooperatively share the same heavyweight process. Inter-thread communication is inexpensive, and context switching from one thread to

the next is low cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java. But, multithreaded multitasking is. Multithreading enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. We can divide a program into threads and execute them parallel.

If we have programmed for operating systems such as Windows 98 or Windows 2000, then it is very easy to learn multithreaded programming. When a program contains multiple flows of control it is known as multithreaded program. Figure illustrates a Java program containing four different threads, one main and three others.



Multithreaded program in Java

In a Java program a main thread is actually a main thread module which is the creating and starting point for all other threads A, B and C. Once initiated by main thread they can run concurrently and share the resources jointly.

Threads running in parallel do not actually mean that they are running at the same time. Since, all threads are running on the single processor, the flow of execution is shared among the threads. The Java interpreter handles the context switching of control among threads in such a way that they runs concurrently. In essence, Java makes the use of operating system's multithreading ability. That is, if a multithreaded program running under Windows platform, does not guarantee that it will give same output on Linux platform.

Threads are extensively used in Java-enabled browsers such as HotJava.

Java's thread model

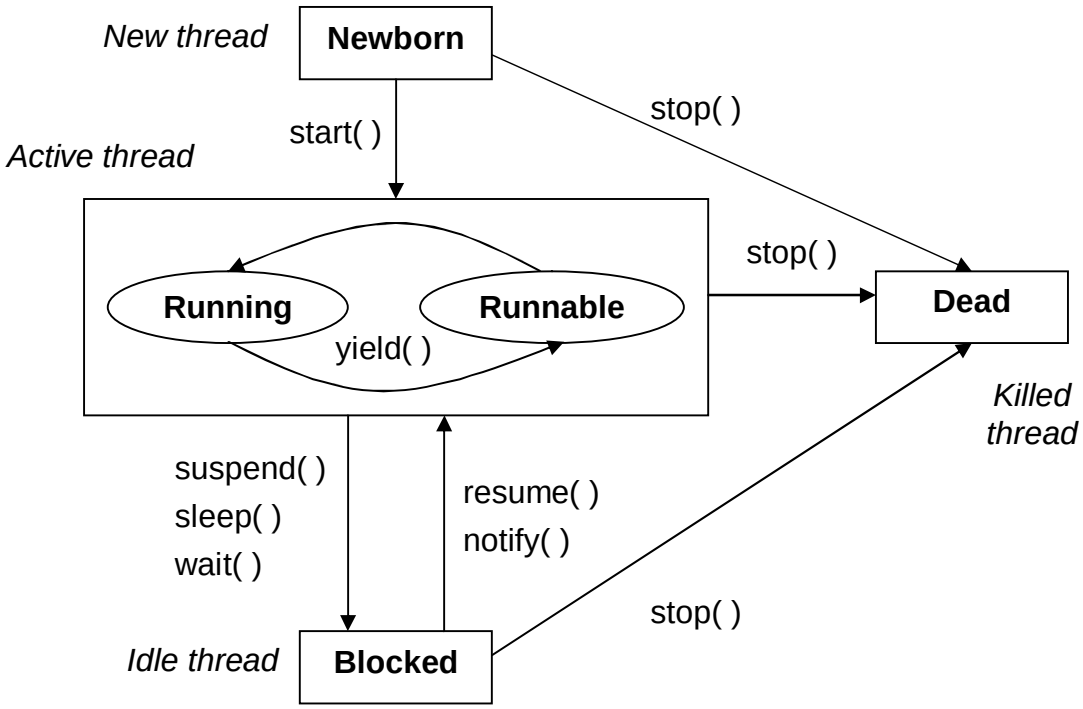
Basically, all Java's class library is designed with multithreading. That is, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

Single-threaded systems use an approach called an event loop with polling. In this system, 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 that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the system. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a singled-threaded environment, when a thread blocks (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Threads exist in several states. A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be suspended, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off. A thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Life cycle of thread



Thread life-cycle

There are many threads in which a thread can enter during its life-time. These are:

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 be shifted from one state to another via variety of ways as shown in figure.

Newborn state

After creation of the thread object, the thread is born which is called as newborn thread. This thread is not scheduled for running. We can either start this state to make it runnable using start( ) method or kill the thread using stop( ) method. Only these two operations can be performed on this state of the thread. If we attempt to perform any other operation, the exception will be thrown.

Runnable state

When the thread is ready for execution and is waiting for availability of the processor, it is said to be in runnable state. The thread has joined the queue of threads that are waiting for execution. If all the

threads have equal priority, then they are given time slots for execution in round robin fashion i.e. on first come first serve basis. The thread that relinquishes control joins the queue at the end and again waits for its execution. This process of assigning time to threads is known as time-slicing. If we want a thread to relinquish control to another thread of equal priority, a yield ( ) method can be used.

Running state

When the processor has given time to the thread for its execution then the thread is said to be in running state. The thread runs until it relinquishes control to its own or it is preempted by a higher priority thread. A running thread may relinquish its control in any one of the following situations:

1. The thread has been suspended using suspend( ) method. This can be revived by using resume( ) method. This is useful when we want to suspend a thread for some time rather than killing it.
2. We can put a thread to sleep using sleep (time) method where 'time' is the time value given in milliseconds. Means, the thread is out of queue during this time period.
3. A thread can wait until some event occurs using wait( ) method. This thread can be scheduled to run again using notify( ) method.

Blocked state

When a thread is prevented from entering into the runnable state and subsequently in running state it is said to be blocked. 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.

Dead state

A running thread ends its life when it has completed its execution of 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 as it is born or while it is running or even when it is in blocked state.

The Thread class and Runnable interface

Java's multithreading system is built upon the Thread class, its methods, and its companion interface called Runnable. Thread class encapsulates a thread of execution. To create a new thread, program will either extend Thread or implement the Runnable interface.

The Thread class defines several methods that help manage threads. The methods that are used in this chapter are shown here:

Method	Meaning
`getName( )`	Obtains a thread's name.
`getPriority( )`	Obtains a thread's priority
`isAlive( )`	Determines whether a thread is still running
`join( )`	Waits for a thread to terminate
`run( )`	Creates entry point for the thread
`sleep( )`	Suspends a thread for some period of time
`start( )`	Starts a thread by calling its run method

Thread class methods

Until we have seen all the examples in this book with a single thread of execution. The remainder of this chapter explains how to use Thread and Runnable to create and manage threads, beginning with the one thread that all Java programs have: the main thread.

The main thread

All the Java programs are at least single threaded programs. When a Java program starts up, one thread begins running immediately. This is usually called the main thread of the program, because it is the one that is executed when our program begins. The main thread is important for two reasons:

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

Though the main thread is created automatically when the program is started, it can be controlled through a Thread object. For this, we must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread class. Its general form is as shown below:

```
static Thread currentThread( )
```

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

```
// Using the main Thread.
  class CurrentThread
  {
```

```
    public static void main(String args[])
                throws InterruptedException
    {
        Thread t = Thread.currentThread();

        System.out.println("Current thread: " + t);

        // changing the name of the thread
        t.setName("Prime Thread");
        System.out.println("After name change: " + t);

        Thread.sleep(3000);
        System.out.println("End of main thread & Program");
    }
}
```

Using and controlling the main thread

Observe the program, a reference to the current thread (the main thread, here) is obtained by calling currentThread( ), and this reference is stored in the local variable ‘t’. Next, the program displays information about the thread directly by printing the value of object. The program then calls setName( ) to change the internal name of the thread. Information about the thread is then redisplayed. Next, the current thread is made to sleep for 3000 milliseconds. The sleep( ) method in Thread might throw an InterruptedException. As it is an unchecked exception, it can be thrown using the throws clause. Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[Prime Thread,5,main]
End of main thread & Program
```

Observe the output of program when ‘t’ is used as an argument to println( ). This display, in order: the name of the thread, its priority, and the name of the thread group to which it belongs. By default, the name of the main thread is main. Its priority is 5, which is also the default value, and main is also the name of the group of threads to which this thread belongs. A thread group is a data structure that controls the state of a collection of threads as a whole. This process is managed by the particular run-time environment. After the name of the thread is changed, t is again displayed. This time, the new name of the thread i.e. “Prime Thread” is displayed.

The sleep( ) method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is:

```
static void sleep(long milliseconds)
       throws InterruptedException
```

The number of milliseconds to suspend is specified in milliseconds. This method may throw an InterruptedException. The sleep( ) method has another form, which allows us to specify the period in terms of milliseconds as well as nanoseconds:

```
static void sleep(long milliseconds, int nanoseconds)
       throws InterruptedException
```

This form is useful only in environments that allow timing periods as short as nanoseconds.

As the preceding program shows, we can set the name of a thread by using setName( ). We can obtain the name of a thread by calling getName( ). These methods are members of the Thread class and are declared as:

```
final void setName(String threadName)
final String getName( )
```

Here, threadName specifies the name of the thread.

5.16 Creating a thread by implementing the Runnable interface

The easiest way to create a thread is to create a class that implements the Runnable interface. We can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared in Runnable as:

```
public void run( )
```

Inside run( ), we will define the code that constitutes the new thread. The method run( ) can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within the program. This thread will end when run( ) returns.

After creating a class that implements Runnable, we have to instantiate an object of type Thread from within that class. Thread defines several constructors. The general form is:

```
Thread(Runnable threadOb, String threadName)
```

In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.

After the new thread is created, it will not start running until we call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run() method. General form of start( ) method is:

```
     void start()

// Creating another thread.
  class NewThread implements Runnable
  {
      Thread t;
      NewThread() //statement3
      {
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Statement4
      }
      public void run()
      {
        for(int i = 5; i > 0; i--)
        {
           System.out.println("Child Thread: " + i);
           try {
           Thread.sleep(500); //statement5
           } catch(Exception e){ }
        }
        System.out.println("Exiting child thread.");
      }
  }
  class ThreadDemo
  {
      public static void main(String args[])
           throws InterruptedException
      {
        new NewThread(); // statement1
        for(int i = 5; i > 0; i--)
        {
           System.out.println("Main Thread: " + i);
           Thread.sleep(1000); //statement2
        }
        System.out.println("Main thread exiting.");
      }
  }
```

Creating the thread using Runnable interface

Observe program. In order to create a thread of execution, we declared a class which implements the Runnable interface. The class name is NewThread. The statement1 shows the constructor which creates the anonymous object of this class. Passing ‘this’ as the first argument to constructor indicates that we want the new thread to call the run( ) method on this object. Next, start( ) is called, which starts the thread of execution beginning at the run( ) method. That is, job of the start method is to make a call to run method. This causes the child thread’s for loop to begin execution. After calling start( ), NewThread’s constructor returns to

main( ). When the main thread resumes, it enters it's for loop. Both threads continue running concurrently, sharing the CPU, until their loops finish. The output produced by this program is as follows:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

In a multithreaded program, often the main thread must be the last thread to finish running. For some older JVMs, if the main thread finishes before a child thread has completed then the Java run-time system "may hang." The example in program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to finish execution earlier than the main thread. Remember, in the above program, 'i' is the local variable for both the classes separately. So, there values will not affect each other.

Creating a thread by inheriting from Thread class

The second way to create a thread is to create a new class that is derived from Thread class, and then to create an instance of that class. The inheriting class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread. Observe the program below, which is just same as program above. But, here the class is created by extending Thread.

```
// Creating another thread by extending Thread.
  class NewThread extends Thread
  {
      NewThread() //statement3
      {
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Statement4
      }
      public void run()
      {
```

```
        for(int i = 5; i > 0; i--)
        {
           System.out.println("Child Thread: " + i);
           try {
           Thread.sleep(500); //statement5
           } catch(Exception e){ }
        }
        System.out.println("Exiting child thread.");
     }
}
class ExtendThread
{
     public static void main(String args[])
          throws InterruptedException
     {
        new NewThread(); // statement1
        for(int i = 5; i > 0; i--)
        {
           System.out.println("Main Thread: " + i);
           Thread.sleep(1000); //statement2
        }
        System.out.println("Main thread exiting.");
     }
}
```

Creating thread by inheriting the Thread class

The program generates the same output as the preceding version. As we can see, the child thread is created by instantiating an object of NewThread, which is derived from Thread. We can call the start( ) method directly as it is the part of Thread class. Notice the call to super( ) inside NewThread. This invokes the following form of the Thread constructor, which gives the name to the thread:

```
public Thread(String threadName)
```

Here, threadName specifies the name given to the thread.

It is wonderful to see that Java has provided two different approaches to create a multithreaded program. So the question is that which approach should we prefer? The Thread class defines several methods that can be overridden by a derived class. Of these methods, the only one that must be overridden is run( ). This is the same method required when we implement Runnable. Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if we will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable. This is up to you, of course. Many times it is required to extend a class from another library class. If we want to create a thread for such classes we can not extend is from thread again. As, it will create the multiple

inheritance it is useful to implement the class from Runnable to create the thread.

Creating multiple threads

Till we have seen only double threaded program. However, a program can create any number of threads as per its need. This concept is illustrated in program.

```
// Creating multiple Threads.
  class NewThread extends Thread
  {
      Thread t;
      String n;
      NewThread(String name)
      {
        n = name;
        t = new Thread(this, name);
        System.out.println("Thread: " + t);
        t.start();
      }
      public void run()
      {
        for(int i = 3; i > 0; i--)
        {
           System.out.println("Thread: " + n);
           try {
           Thread.sleep(500);
           } catch(Exception e){ }
        }
        System.out.println("Exiting thread "+n);
      }
  }
  class MultiThread
  {
      public static void main(String args[])
           throws InterruptedException
      {
        new NewThread("IND"); // statement1
        new NewThread("PAK"); // statement2
        new NewThread("SRI"); // statement3
        for(int i = 3; i > 0; i--)
        {
           System.out.println("Main Thread: " + i);
           Thread.sleep(1000); //statement4
        }
        System.out.println("Main thread exiting.");
      }
  }
```

Creating the multiple threads

Output:

```
Thread: Thread[IND,5,main]
Thread: Thread[PAK,5,main]
Thread: Thread[SRI,5,main]
Thread: PAK
Thread: IND
Main Thread: 3
Thread: SRI
Thread: PAK
Thread: IND
Thread: SRI
Thread: PAK
Thread: IND
Main Thread: 2
Thread: SRI
Exiting thread PAK
Exiting thread IND
Exiting thread SRI
Main Thread: 1
Main thread exiting.
```

Observe the output, once started; all three child threads created in statement1, statement2 and statement3 share the CPU. The call to sleep(1000) in main( ). This causes the main thread to sleep for 1 second each time in the for loop and ensures that it will finish last.

The isAlive( ) and join( ) methods

The isAlive( ) method of Thread class is used to determine whether the thread has finished its execution or not? The general form of this method is:

```
final boolean isAlive()
```

This method returns true if the thread upon which it is called is still running else returns false.

The join( ) method is used to wait for the thread to finish. Its general form is:

```
final void join() throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of join( ) allow us to specify a maximum amount of time that we want to wait for the specified thread to terminate.

```
// Using join() and isAlive()
```

```
class NewThread implements Runnable
{
    String name;
    Thread t;
    NewThread(String tname)
    {
      name = tname;
      t = new Thread(this, name);
      System.out.println("Thread: " + t);
      t.start();
    }
    public void run()
    {
      try {
        for(int i = 3; i > 0; i--)
        {
          System.out.println(name + ": " + i);
          Thread.sleep(1000);
        }
       } catch (InterruptedException e) { }
       System.out.println(name + " exited");
    }
}
class JoinDemo
{
    public static void main(String args[])
       throws InterruptedException
    {
      NewThread ob1 = new NewThread("First");
      NewThread ob2 = new NewThread("Second");
      NewThread ob3 = new NewThread("Third");
      System.out.println("First Thread is alive: "
                        + ob1.t.isAlive());
      System.out.println("Second Thread is alive: "
                        + ob2.t.isAlive());
      System.out.println("Third Thread is alive: "
                        + ob3.t.isAlive());
      // waiting for threads to finish
      System.out.println("Waiting for threads to finish.");
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
      System.out.println("First Thread is alive: "
                        + ob1.t.isAlive());
      System.out.println("Second Thread is alive: "
                        + ob2.t.isAlive());
      System.out.println("Third Thread is alive: "
                        + ob3.t.isAlive());
      System.out.println("Main thread exited"); //statement1
    }
}
```

Using the isAlive( ) and join( ) methods

Output:

```
Thread: Thread[First,5,main]
Thread: Thread[Second,5,main]
First: 3
Thread: Thread[Third,5,main]
First Thread is alive: true
Third: 3
Second: 3
Second Thread is alive: true
Third Thread is alive: true
Waiting for threads to finish.
First: 2
Third: 2
Second: 2
First: 1
Third: 1
Second: 1
First exited
Third exited
Second exited
First Thread is alive: false
Second Thread is alive: false
Third Thread is alive: false
Main thread exited
```

After the calls to join( ) return, the threads have stopped their execution. If we change the sleeping period given in the thread, then also the statement will be executed at last always. Because of previous three statements, it made to wait the termination of threads first, second and third respectively.

Thread priorities

Every initiated thread in a program has its priority. Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. So, higher-priority threads get more CPU time than lower priority threads. Actually, the amount of CPU time that a thread gets often depends on several factors apart from its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O), it will preempt the lower-priority thread.

In order to set a thread's priority, use the setPriority( ) method of thread class. It is having the following general form:

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

Here, 'level' specifies the new priority setting for the calling thread. The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively. I we want to set priority of the thread to default priority; it can be specified by NORM_PRIORITY, which is currently 5. These priorities are defined as final Variables within Thread.

Thread class has defined another method to obtain the priority given to the Thread named getPriority( ). It has following general form:

final int getPriority()

```
// Assigning Thread priorities.
  class NewThread implements Runnable
  {
      Thread t;
      String n;
      public NewThread(int pri, String name)
      {
        n = name;
        t = new Thread(this, name);
        t.setPriority(pri);  //statement7
        System.out.println("Thread: " + t);
      }
      public void run()
      {
        for(int i = 100;; i--)
        {
           try {
           System.out.println("Thread: " + n);
           Thread.sleep(10); //statement8
           }catch(InterruptedException e){ }
        }
      }
      public void start()
      {
        t.start();
      }
  }
  class ThreadPrio
  {
      public static void main(String args[])
           throws InterruptedException
      {
        Thread.currentThread().setPriority(10); // statement1
        NewThread abc = new NewThread(7,"ABC"); // statement2
        NewThread xyz = new NewThread(3,"XYZ"); // statement3
        abc.start();     // statement4
        xyz.start();     // statement5
           Thread.sleep(1000); //statement6
```

```
        System.out.println("Main thread exiting.");
      }
  }
```

assigning and using thread priorities

Snapshot of the output:

```
Thread: ABC
Thread: XYZ
Thread: ABC
Thread: XYZ
Thread: ABC
Thread: XYZ
Thread: ABC
Thread: XYZ
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
Thread: ABC
```

In program, three different threads are created including main thread. The statement1 assigns highest priority to the main thread. The statement2 and statement3 creates the thread named “ABC” and “XYZ” with priorities 7 and 3 respectively. So the thread “ABC” is having higher priority than “XYZ”. That is reflected in the output also. Actually, this program will run into infinity. Observe the snapshot of the output, the highest priority thread is executed more number of times than the lower priority one. Instead of directly giving the numbers we can use the constants, NORM_PRIORITY, MAX_PRIORITY and MIN_PRIORITY etc.

This concept is thoroughly useful in developing the application Java program for operating system. Remember we will get different output of the same program on Linux or any other operating system. That is, it depends upon the algorithm that the operating system uses for thread scheduling.

Thread synchronization

Threads can share the same memory space, that is, they can share resources. However, there are critical situations where it is required that only one thread at a time has access to a shared resource. For example, crediting and debiting a shared bank account concurrently amongst several users without proper discipline, will put a risk the integrity of the account data. Java provides high-level concepts for synchronization in order to control access to shared resources.

A lock (also called as monitor or semaphore) is used to synchronize access to a shared resource. A lock can be associated with any shared resource. Threads acquire access to a shared resource by first gaining the lock associated with that resource. At any given time, maximum one thread can hold the lock (i.e., own the monitor) and thereby it can have access to that shared resource. Thus, a lock implements mutual exclusion (or mutex).

In Java, all objects have a lock including arrays. This means that the lock from any Java object can be used to implement mutual exclusion. 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. At a time, only one thread can access the shared resource guarded by the object lock.

The object lock mechanism forces the following rules of synchronization:

- A thread must acquire the object lock associated with a shared resource, before it can enter the shared resource. The runtime system ensures that no other thread can enter a shared resource if another thread already holds the object lock associated with the shared resource. If a thread cannot immediately acquire the object lock, it is blocked, that is, it must wait for the lock to become available.
- When a thread exits a shared resource, the runtime system ensures that the object lock is also handed over. If another thread is waiting for this object lock, it can proceed to acquire the lock in order to access to the shared resource.

The keyword ‘synchronized’ and the lock forms the basis for implementing synchronized execution of code. There are two different ways in which execution of code can be synchronized:

- synchronized methods
- synchronized blocks

Synchronized methods

As all objects have their own implicit monitor associated with them, synchronization is easy in Java. In order to enter an object’s lock, we just need to call a method that has been modified with the keyword

'synchronized'. When a thread is running inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance they have to wait. To exit the lock and give up control of the object to the next waiting thread, the owner of that lock simply returns from the synchronized method.

In order to understand the concept of synchronization, just observe the program.

```
// The program does not have any synchronization.
  class JavaProg
  {
    void display(String name,int num)
    {
      System.out.print(name+": <" + num);
      try {
        Thread.sleep(1000);
      } catch(InterruptedException e) { }
      System.out.println(">");
    }
  }
  class NewThread implements Runnable
  {
    String msg;
    JavaProg tar;
    Thread t;
    public NewThread(JavaProg targ, String s)
    {
       tar = targ;
       msg = s;
       t = new Thread(this);
       t.start();
    }
    public void run()
    {
       for(int i=0;i<5;i++)
         tar.display(msg,i);
    }
  }
  class Synchronization
  {
      public static void main(String args[])
                throws InterruptedException
      {
        JavaProg obj = new JavaProg();
        NewThread ob1 = new NewThread(obj, "First");
        NewThread ob2 = new NewThread(obj, "Second");
        NewThread ob3 = new NewThread(obj, "Third");
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
```

```
      }
  }
```

Program without synchronization

Output:

```
First: <0Second: <0Third: <0>
>
Third: <1First: <1>
Second: <1>
Third: <2>
Second: <2>
First: <2>
Second: <3>
First: <3>
Third: <3>
First: <4>
Second: <4>
Third: <4>
>
>
```

Observe the output, by calling sleep( ), the call( ) method allows execution to switch to another thread. So, it resulted in the mixed-up output of the three message strings. In above program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a race condition, because the three threads are racing each other to complete the method display( ). This example used sleep( ) to make the effects repeatable and obvious. Many times, a race condition is more subtle and less predictable, because we can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next time. So, we must serialize access to display( ) method. That is, we must have to restrict its access to only one thread at a time. For this, we simply need to precede display( ) method's definition with the keyword 'synchronized', such as:

```
class JavaProg
{
     synchronized void display(String name,int num)
     {
          ……….
          ……….
     }
}
```

After this, when a thread is executing the display( ) method none of the threads will be allowed to enter in it. So, only one thread can access the method at a time. After adding 'synchronized', we will get the output:

```
First: <0>
```

```
Third: <0>
Third: <1>
Third: <2>
Third: <3>
Second: <0>
Third: <4>
First: <1>
First: <2>
First: <3>
First: <4>
Second: <1>
Second: <2>
Second: <3>
Second: <4>
```

Now the output is serialized.

Synchronized blocks

Using synchronized methods is very easy and effective means of synchronization. But it will not work in all cases. For example, if we want to make a synchronized access to particular objects, that is all its member variables and methods then synchronized blocks can be used. The form of using the synchronized blocks is as follows:

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

Here, 'object' is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's lock. If we want to synchronize the current object and its methods then the 'this' keyword can be used instead of 'object'.

Now we will modify the previous program with block synchronization. It is shown in program below.

```
// The program with synchronized block.
  class JavaProg
  {
    void display(String name,int num)
    {
      System.out.print(name+": <" + num);
      try {
        Thread.sleep(1000);
      } catch(InterruptedException e) { }
      System.out.println(">");
    }
```

```
}
class NewThread implements Runnable
{
  String msg;
  JavaProg tar;
  Thread t;
  public NewThread(JavaProg targ, String s)
  {
     tar = targ;
     msg = s;
     t = new Thread(this);
     t.start();
  }
  public void run()
  {
     synchronized(tar)      //statement1
     {
       for(int i=0;i<5;i++)
         tar.display(msg,i);
     }
  }
}
class SynchBlock
{
    public static void main(String args[])
              throws InterruptedException
    {
      JavaProg obj = new JavaProg();
      NewThread ob1 = new NewThread(obj, "First");
      NewThread ob2 = new NewThread(obj, "Second");
      NewThread ob3 = new NewThread(obj, "Third");
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    }
}
```

Using synchronized block

Output:

```
First: <0>
First: <1>
First: <2>
First: <3>
First: <4>
Third: <0>
Third: <1>
Third: <2>
Third: <3>
Third: <4>
Second: <0>
Second: <1>
```

```
Second: <2>
Second: <3>
Second: <4>
```

Observe the output of program. Here we got the best synchronization than that of pervious one. In statement1, we have synchronized the object 'tar'. So, each thread has completed its work independently. If we removed the statement1, we might get the output as:

```
First: <0Second: <0Third: <0>
First: <1>
Second: <1>
Third: <1>
First: <2>
Second: <2>
Third: <2>
First: <3>
Second: <3>
Third: <3>
First: <4>
>
Third: <4Second: <4>
>
>
```

This output is too random and disturbed.

Synchronized blocks can also be specified on a class lock:

```
synchronized (classname.class)
{
     //code block to be synchronized
}
```

The block synchronizes on the lock of the object denoted by the reference classname.class. For example, a static synchronized method display( ) in class A is equivalent to the following declaration:

```
static void display( )
{
     synchronized (A.class)
     {
          // synchronized block on class A
          // ......
     }
}
```

In short, a thread can hold a lock on the object,

- by executing a synchronized instance method of the object

- by executing the body of a synchronized block that synchronizes on the object
- by executing a synchronized static method of a class

Using wait( ) and notify( )

Waiting and notifying provide means of communication between threads that synchronize on the same object. The threads execute wait( ) and notify( ) (or notifyAll( )) methods on the shared object for this purpose. These final methods are defined in the Object class, and therefore, inherited by all objects. These methods are referred as inter-thread communication methods.

These methods are used to avoid the polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, suitable action is taken. This wastes some CPU time. For example, consider the producer-consumer problem, where one thread is producing some data and another is consuming it. Here, the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. This situation is undesirable. We need to design it in such a way that when producer has produced some data consumer consumes it. This process goes on in sequence. In order to do this, we need to establish proper communication between the producer and consumer threads. This can be done by wait( ) & notify( ) methods.

- wait( ): It is used to tell the calling thread to give up the lock and go to sleep until some other thread enters the same lock and calls notify( ).
- notify( ): It wakes up the first thread that called wait( ) on the same object.
- notifyAll( ): It wakes up all the threads that called wait( ) on the same object. Here, the highest priority thread will run first.

These methods can only be executed on an object whose lock the thread holds, otherwise, the call will result in an IllegalMonitorStateException.

```
final void wait(long timeout) throws InterruptedException
final void wait(long timeout, int nanos)
          throws InterruptedException
final void wait() throws InterruptedException
```

The ‘timeout’ specifies the time needed to wait. A thread invokes wait( ) method on the object whose lock it holds. The thread is added to the wait set of the object.

```
final void notify()
final void notifyAll()
```

A thread invokes a notification method on the object whose lock it holds to notify thread(s) that are in the wait set of the object.

The following program shows the incorrect implementation of producer-consumer problem. It consists of four classes: 'Queue', it is a queue that we are trying to synchronize; 'Producer', the threaded object that is producing queue entries; 'Consumer', the threaded object that is consuming queue entries; and 'ProCon', the class that creates the single Queue, Producer and Consumer.

```
// Incorrect implementation of a producer and consumer.
class Queue
{
   int num;
   synchronized int consume()
   {
      System.out.println("Consumed: " + num);
      return num;
   }
   synchronized void produce(int n)
   {
      num = n;
      System.out.println("Produced: " + num);
   }
}
class Producer implements Runnable
{
   Queue q;
   Producer(Queue q)
   {
      this.q = q;
      new Thread(this, "Producer").start();
   }
   public void run()
   {
      int i = 0;
      while(i<5)
         q.produce(++i);
   }
}
class Consumer implements Runnable
{
   Queue q;
   Consumer(Queue q)
   {
      this.q = q;
      new Thread(this, "Consumer").start();
   }
```

```
    public void run()
    {
       while(q.consume()<4);
    }
  }
  class ProCon
  {
     public static void main(String args[])
     {
        Queue q = new Queue();
        new Producer(q);
        new Consumer(q);
     }
  }
```

Incorrect producer-consumer problem.

Output:

```
Produced: 1
Produced: 2
Consumed: 2
Consumed: 2
Produced: 3
Produced: 4
Produced: 5
Consumed: 5
```

Though the produce( ) and consume( ) methods on class Queue are synchronized, the producer is overrunning the consumer, as well as consumer is consuming the same queue value twice. Thus, we got the erroneous output shown above. Here the output will vary as per the platform varies. So in order to establish a meaningful communication between producer and consumer, we need to use wait and notify methods. It is effectively implemented in program as shown below:

```
// Correct implementation of a producer and consumer.
  class Queue
  {
     int num;
     boolean isSet = false;
     synchronized int consume()
     {
        if(!isSet)
        {
           try{
             wait();
           }catch(Exception e) { }
        }
           System.out.println("Consumed: " + num);
           isSet = false;
           notify();
```

```
            return num;
      }
      synchronized void produce(int n)
      {
         if(isSet)
         {
            try{
              wait();
            }catch(Exception e) { }
         }
         num = n;
         System.out.println("Produced: " + num);
         isSet = true;
         notify();
      }
   }
   class Producer implements Runnable
   {
      Queue q;
      Producer(Queue q)
      {
         this.q = q;
         new Thread(this, "Producer").start();
      }
      public void run()
      {
         int i = 0;
         while(i<5)
            q.produce(++i);
      }
   }
   class Consumer implements Runnable
   {
      Queue q;
      Consumer(Queue q)
      {
         this.q = q;
         new Thread(this, "Consumer").start();
      }
      public void run()
      {
         while(q.consume()<5);
      }
   }
   class ProConUsed
   {
       public static void main(String args[])
       {
          Queue q = new Queue();
          new Producer(q);
          new Consumer(q);
```

```
        }
    }
```

correct implementation of producer-consumer problem

Output:

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
```

Observe program. Inside consume( ), wait( ) is called. This causes its execution to suspend until the Producer notifies you that some data is ready. When this happens, execution inside consume( ) resumes. After the data has been obtained, consume( ) calls notify( ). This tells Producer that it is now okay to put more data in the queue. Inside produce( ), wait( ) suspends execution until the Consumer has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and notify( ) is called. This tells the Consumer that it should now remove it. Here the variable ‘isSet’ plays an important role to check whether the consume( ) or produce( ) has finished their task properly. The output obtained shown clean and synchronized behavior of producer-consumer problem.
