5 Exception Handling

Java offers an excellent exception-handling mechanism which is easy to understand and implement. Java treats exceptions as objects and all the exceptions (errors) are arranged in a hierarchy. Its features include exception propagation, handling multiple exceptions based on inheritance hierarchy and providing user-defined exceptions. This chapter starts with a general discussion of exception-handling in object oriented programming languages and covers various concepts of Java's exception-handling features.

5.1 Introduction

An exception is an abnormal condition occurring during the execution of a program that causes it to deviate it from the normal execution path. When an exception occurs, it makes further execution of the program impossible. An exception can be defined as follows:

> An **exception** is an event that may cause abnormal termination of the program during its execution.

Different types of error—user errors, logic errors or system errors—can cause exceptions. Examples of errors include division by zero, out of array bounds, running out of virtual memory, opening an invalid file for reading and trying to read from an empty stack.

Whenever an error occurs during the execution of a program, it *throws an exception*. If the exception is not handled properly, the program execution is abruptly terminated, causing serious problems. In some books, the phrased 'throws an exception' is also sometimes replaced by 'raises an exception' or 'triggers an exception'.

Run-time error management in programming languages is a challenging task. In traditional programming languages such as C, Pascal and Fortran, common errors are handled by providing if-else statements. Detection of errors and handling them is often confusing, tedious and time consuming. On other hand, object oriented programming languages such as C++ and Java treat errors (or error conditions) as objects and provide exception handlers that are separate from the basic code so that the errors can be tackled efficiently.

What is involved in handling exceptions? Exception handling is a mechanism that enables programs to detect and handle errors before they occur rather than allowing them to occur and suffering the consequences. It is designed for dealing with synchronous errors such as an attempt to divide by zero.

The need for error handling can be better understood by studying Program 5.1 and analysing the outputs corresponding to different inputs.

Program 5.1 A common exception: dividing by zero.

```
public class Divide
{
    public static void main(String[ ] args)
    {
        System.out.println("\nProgram starts here\n");
        int a,b,c;

        a = Integer.parseInt(args[0]);
        b = Integer.parseInt(args[1]);
        c = a/b;
        System.out.println(c + "=" + a +"/"+b);
        System.out.println("\nProgram Ends here");
    }
}
```

Program 5.1 shows the following outputs for two different sets of input:

```
C:\javatest>java Divide 4 2
Program starts here
2 = 4/2
Program Ends here

C:\javatest>java Divide 4
Program starts here
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
    at Divide.main(Divide.java:10)

C:\javatest>java Divide a 4
Program starts here
Exception in thread "main" java.lang.NumberFormatException: a
    at java.lang.Integer.parseInt(Integer.java:426)
    at java.lang.Integer.parseInt(Integer.java:476)
    at Divide.main(Divide.java:9)

C:\javatest>java Divide 4 0
Program starts here
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Divide.main(Divide.java:11)

C:\javatest>
```

The program will be executed successfully (as in the case of the first run, that is, java Divide 4 2) if two conditions are met:

1. If number of arguments passed at run-time are two
2. If the arguments passed are both valid numbers

A run-time error occurs if any of the following takes place:

1. If the number of arguments passed is less than two (more precisely, not equal to two)
2. If any of the arguments passed is not a number
3. If the second argument is zero

So when the program terminates abnormally, the last statement,

```
println("Program Ends here")
```

is never executed.

To prevent interruptions of the normal flow of a program from exceptions, such as the ones discussed above, exception handlers in the program identify the exceptions and handle them appropriately. Exception handlers provide applications with many benefits, more than the traditional error-handling mechanisms. If there is no user-provided exception handler, the Java run-time system provides a default mechanism to terminate the program.

What is an exception handler? An *exception handler* is a piece of code used to deal with the exceptions, either to fix the error or abort execution in a sophisticated way. It is invoked when an exception is encountered. Exception handlers catch the exceptions raised during the execution of the code and handle them according to the instructions given in the handler code. In Java, exception handlers are represented as catch blocks.

The catch block in Java is used to handle exceptions that are raised during program execution.

5.2 Basics of Exception Handling in Java

The strong capability of Java to handle exceptions makes it possible for the programmer to monitor the program for *exceptional conditions* within it and transfer control during such conditions to a special exception-handling code that he or she provides along with the main code.

Java's exception-handling mechanism is geared to situations in which the method that detects an error is unable to deal with it. Such a method will throw an exception. There is no guarantee that there will be an 'Exception Handler' that is specifically designed to process that particular exception. If there is, the exception will be caught and handled.

A catch block contains Java code to handle exceptions so that the program will not terminate abnormally. Look at the following code:

```
public static float Divide(float a , float b)
{
    float c;
    try
    {
        c = a/b;
    }
    catch (Arithmetic Exception e)
    {
        System.out.println("The exception is" +e);
    }
}
```

In the above method, divide(), when the statement in the try block attempts to divide by zero (by passing zero value for b), the exception ArithmeticException will be raised. Immediately, the catch block catches this exception and handles it. The catch block displays the string in the println() method.

When an exception occurs, as described in the previous paragraph, Java creates an exception object and performs an action called 'throwing an exception'. The run-time system searches through the *catch* block, which is a block of code that is designed to handle the exception. It searches, starting from the method in which the exception occurred and searches backwards up to the call stack. This search continues till the run-time system finds a method that contains an appropriate exception handler. An exception handler is said to be appropriate if the type of the exception thrown is the same as the type of exception that can be dealt with by the handler or one of its sub-classes. The way to handle an exception is first to look for an appropriate handler associated with the block that has been thrown. If there is no appropriate handler, within that block the search progresses up through the call stack until an appropriate handler is found which can handle the exception. If no appropriate exception handler is found, the run-time system stops and the program terminates.

Once an exception is thrown, the block in which the exception is thrown expires and control cannot return to the throw point.

That is, it is an unconditional transfer of control. Thus Java uses the termination model of exception handling rather than the resumption model. In the resumption model, control would return to the point at which the exception was thrown and the program would resume execution. (This is similar to a function call.)

When an exception occurs, it is possible to communicate information to the exception handler from the vicinity of the location of the exception. The exception object generally contains

information about the exception, such as its type and the state of the program when the exception occurred. The three important features that an exception usually carries are the following:

1. The type of exception—determined by the class of the exception object.
2. Where the exception occurred—the stack trace.
3. Context information—the error message and other state information.

5.3 Exception Hierarchy

When a condition causes an exception to be thrown, that exception is thrown either by the Java virtual machine or by the Java throw statement. Exceptions are represented by objects instantiated from the class java.lang. Throwable or one of its sub-classes. Java also allows users to define classes that can throw their own (user-defined) exception objects as needed in their design; however, this new class must extend the throwable class or one of its sub-classes.

5.3.1 Throwable class hierarchy

Figure 5.1 shows a class hierarchy diagram that illustrates some of the exception handling classes and the relationships between them. The Throwable class is a super-class in the exception hierarchy. There are two sub-classes of Throwable class: Error and Exception. The difference between the two is that the Error class deals with exceptional events from which it is generally not possible to recover, whereas the Exception class deals mostly with events from which it is possible to recover.

An exception of the Error type is non-recoverable and should not be caught. Applications should not try to catch the objects of Error class. They usually cause the Java virtual machine

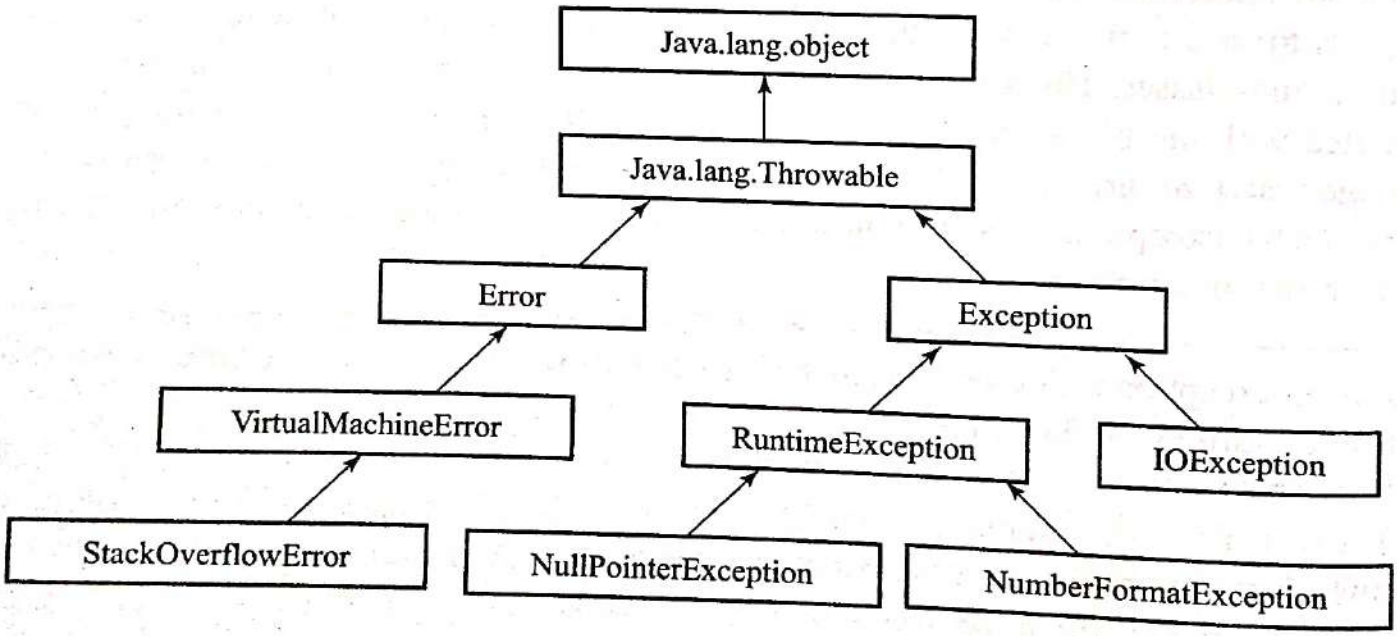


Fig. 5.1 Exception inheritance hierarchy (partial).

to display a message and then exit. These type of exceptions indicate a serious problem in the application and causes abnormal termination without handling. An example of a non-recoverable error is out of memory. These errors are operating system and hardware-dependent and hence cannot be handled by programmers.

An exception of the Exception type indicates an abnormal condition that must be properly handled to prevent program termination. An example of this type of error is divide by zero.

In rest of the chapter, unless specified otherwise, we focus on the class Exception compare and its sub-classes.

The Java Exception class is represented as follows:

```
Public class Exception extends Throwable
{
    public Exception()
    {
        super( );
    }
    public Exception(String s)
    {
        super( s);
    }
}
```

There are several sub-classes of the Error class and the Exception class. Many of these sub-classes themselves have numerous sub-classes. For instance, the class ArithmeticException is a sub-class of RuntimeException which is again a sub-class of Exception. Similarly, VirtualMachineError is a sub-class of Error which indicates that the JVM is broken or has run out of resources necessary for it to continue operating. Figure 5.1 shows the inheritance hierarchy between a few classes that are meant for throwing exceptions. (A comprehensive list of exceptions is given in section 5.10.) Classes in Java are organized in the form of groups, and each group forms a sub-tree of the inheritance tree.

Note that the Throwable class is the super-class in the inheritance hierarchy. The classes Throwable, Error and Exception are found in the java.lang package. Some exception classes such as IOException and FileNotFoundException are defined in packages such as java.io.

The most commonly used exceptions are mentioned and described in Table 5.1.

5.4 Constructors and Methods in Throwable Class

5.4.1 Constructors

There are four constructors in the Throwable class:

- Throwable( )
- Throwable(String message)

Table 5.1 Commonly used exception classes.

Exception Name	Description
ArrayOutofBoundsException	An array index is out of bounds.
FileNotFoundException.	The requested file is not found.
InterruptedException.	A thread has been interrupted.
ClassNotFoundException	A class cannot be found.
NullPointerException	A field or method access for a null object
NumberFormatException	A number is in illegal format
NegativeArraySizeException	A negative number was used for an array size.
ArithmeticException	An arithmetic exception like division by zero.

- Throwable(String message, Throwable cause)
- Throwable(Throwable cause)

The first two allow constructing an exception object with or without a String message encapsulated in the object.

5.4.2 Methods

There are several methods in the Throwable class. Three useful methods of the Throwable class that provide information about an exception are the following:

- getmessage( )—Returns the message that was encapsulated when the object was initiated. It returns null, if there is no message.
- tostring()—returns a brief description about the exception of throwable object.
- printstacktrace( )—prints the stack trace.

StackTrace is a list of the methods executed in sequence that lead to the exception and it is typically used to show the (unhandled) run-time errors on the screen.

The methods of the Throwable class that deal with StackTrace are the following:

- fillInStackTrace()
- getStackTrace()
- printStackTrace()
- setStackTrace(StackTraceElement[ ] stackTrace)

When an exceptional condition occurs within a method, the method may instantiate an exception object and hand it to the run-time system to deal with it. This is called throwing an exception.

5.5 Unchecked and Checked Exceptions

Based on the severity of the error, Exception classes are categorized into two groups: *unchecked* exceptions and *checked* exceptions.

Unchecked exceptions are those that can be either handled or ignored. If the programmer ignores an unchecked exception, the program will terminate when such an error occurs. On the other hand, if a handler is provided for an unchecked exception, the result of the occurrence of such an error will depend on the code written in the exception handler. An example of an unchecked exception is the class RuntimeException (and its sub-classes), which is a sub-class of the class Exception. It is a very important class in Java programming.

Checked exceptions are a type of exceptions that cannot be ignored. These exception classes represent routine abnormal conditions. The programmer should anticipate and catch them to avoid abnormal termination of the program. These exceptions are checked for by the compiler and thus the program code must handle or declare checked exceptions, otherwise the program will not compile. Thus, checked exceptions should be treated more carefully.

All exceptions instantiated from the Exception class, or from sub-classes of Exception except RuntimeException and its sub-classes, must be dealt with in one of the following ways:

- They must be handled with a try block followed by a catch block, or
- they must be declared in the throws clause of any method that can throw them.

5.6 Handling Exceptions in Java

Exception handling in Java is accomplished by using five keywords: try, catch, throw, throws and finally.

In Java, the code that may generate (or throw) an exception is enclosed in a try block. The try block can be followed immediately by one or more ***catch*** blocks with a ***finally*** block as the last block. The try block can also end without a catch block, and the catch blocks need not always have finally as the last block.

Each catch block specifies the type of exception it can catch and contains an exception handler code. After the last catch block, an optional finally block provides code that is always executed regardless of whether or not an exception occurs. If there are no catch blocks following a try block, the finally block is required.

When an exception is thrown, program control goes out of the try block and the catch blocks are searched in the order described earlier for an appropriate handler. If the exception type thrown matches the parameter type in one of the catch blocks, the code for that catch block is executed. If no exceptions are thrown in the try block, the exception handlers (catch blocks) for that block are skipped and the program resumes execution after the last catch block. If a finally block appears after the last catch block, it is executed, whether or not an exception is thrown.

We can specify with a throws clause the exceptions a method throws. An exception can be thrown from statements in the method or from a method that is called directly or indirectly from the try block. The point at which the throw is executed is called the throw point.

The Java statements that deal with exception handling are described in the following sections. The general syntax of try-catch-finally statement is as shown below:

```
try
{
    // java statements capable of throwing exceptions
}
catch( Exception_1 e)
{
    // java statements to handle the exception Exception_1
}
finally
{
    // cleanup code or exit code (optional)
}
```

5.6.1 Try block

The statements in a program that may raise exception(s) are placed within a try block. A try block is a group of Java statements, enclosed within braces { }, which are to be checked for exceptions. A try block starts with the keyword try.

To handle a run-time error and monitor the results, we simply enclose the code inside a try block. If an exception occurs within the try block, it is handled by the appropriate exception handler (catch block) associated with the try block. If there are no exceptions to be thrown, then try will return the result of executing the block. A try block should have one (or more) catch block(s) or one finally block or both. If neither is present, a compiler error occurs which says 'try' without 'catch' or 'finally'.

Exception handlers are put in association with a try block by providing one (or more) catch block(s) directly after the try block.

5.6.2 Catch block

A catch block is a group of Java statements, enclosed in braces { } which are used to handle a specific exception that has been thrown. catch blocks (or handlers) should be placed after the try block. That is, a catch clause follows immediately after the try block.

When an exception occurs in a try block, program control is transferred to the appropriate catch block. This transfer of control is unconditional, in the sense that the control will not return to the statement just after the throw point in the try block.

A catch block is specified by the keyword catch followed by a single argument within parentheses ( ). The argument type in a catch clause is from the Throwable class or one of its sub-classes. Java initializes the argument (parameter) to refer to the caught object. Program 5.2 illustrates the use of the try-catch statements.

Program 5.2 Using the try-catch statements.

```
public class Divide
{
    public static void main(String[ ] args)
    {
        System.out.println("\nProgram Execution starts here\n");
        int a,b,c;
        try
        {
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
            c = a/b;
            System.out.println( a + "/" + b + "=" + c );
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("\nProgram Execution completes here");
    }
}
```

The output of the above program is as follows:

```
C:\javatest>java Divide 4 2
Program Execution starts here
4 / 2 = 2
Program Execution completes here

C:\javatest>java Divide a 1
Program Execution starts here
java.lang.NumberFormatException: a
Program Execution completes here

C:\javatest>java Divide 4 0
Program Execution starts here
java.lang.ArithmeticException: / by zero
Program Execution completes here

C:\javatest>java Divide 4
Program Execution starts here
```

```
java.lang.ArrayIndexOutOfBoundsException
Program Execution completes here

C:\javatest>
```

Once the code in the catch block is completely executed, program control is passed to the executable statement immediately following the catch block, which is a print statement, in the above example.

Note that while writing multiple catch statements, super-class exception types should follow the sub-class exception types, otherwise the unreachable code exception will arise during compile time.

5.6.3 Finally block

The final step in setting up an exception handler is providing a mechanism for cleaning up the program before control is passed to different part of the program. This can be achieved by enclosing the clean-up logic within the finally block. This is described in the following way.

The code within the finally block is always executed regardless of what happens within the try block. That is, if no exceptions are thrown, then no part of the code in the catch blocks is executed but the code in the finally block is executed.

If an exception is thrown, first the code in the corresponding catch block (exception handler) is executed and then the control is passed to the finally block and the code in the finally block is executed. Note that there is only one finally block per one try block. Program 5.3 illustrates the use of the try-catch-finally statements.

Program 5.3 Using the try-catch-finally statements.

```
public class Divide
{
    public static void main(String[ ] args)
    {
        System.out.println("\nProgram Execution starts here\n");
        int a,b,c;
        try
        {
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
            c = a/b;
            System.out.println( a + "/" + b + "=" + c );
        }
        catch(Exception e)
```

```
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("Finally blocks always get executed");
        }
        System.out.println("\nProgram Execution completes here");
    }
}
```

The output of Program 5.3 is given below:

```
C:\javatest>java Divide 4 2
Program Execution starts here
4 / 2 = 2
Finally blocks always get executed
Program Execution completes here

C:\javatest>java Divide 4 0
Program Execution starts here
java.lang.ArithmeticException: / by zero
Finally blocks always get executed
Program Execution completes here

C:\javatest>
```

Generally the finally block is used to close the files that have been opened, connections to the databases that have been opened, sockets in the networks that have been opened or for releasing any system resources. Note that if the code in the catch block terminates the program by executing System.exit(0), the code in the finally block will not be executed.

In the above code listing, a temporary file is created in the try block and the block also has some code that can potentially throw an exception. Irrespective of whether the try block succeeds, the temporary file has to be closed and deleted from the file system. This is accomplished by closing and deleting the file in the finally block.

5.6.4 Multiple catch blocks

Java allows having multiple catch blocks with one try block. Program 5.4 illustrates the use of multiple catch blocks associated with a single try block.

Program 5.4 Using multiple catch blocks.

```
public class Divide
{
    public static void main(String[ ] args)
    {
        System.out.println("\nProgram Execution starts here\n");
        int a,b,c;
        try
        {
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
            c = a/b;
            System.out.println( a + "/" + b + "=" + c );
        }
        catch(NumberFormatException e)
        {
            System.out.println("Arguments passed should be valid Numbers");
        }
        catch(ArithmeticException e)
        {
            System.out.println("Second Argument Should not be Zero");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Pass Proper Arguments");
        }

        System.out.println("\nProgram Execution Completes here");
    }
}
```

To see how Program 5.4 responds to different inputs, the reader can run the program by giving inputs such as (4, 2), (4, a), (4, 0) etc. (The output of Program 5.4 can be seen at http://www.universitiespress.com/downloads/books/81-7371-572-6.pdf)

When an exception occurs in the try block, the catch clauses are checked sequentially. A catch clause catches the exception if the thrown object is an instance of the type that is specified in the catch clause.

When multiple statements in a single try block throw different exceptions, different catch blocks must be written, one for each type of exception. On the other hand, each statement that might throw exception can be put into different try-catch blocks. Alternately, a single catch block can be defined as its parameter type, which is a super-class of all the exceptions that are to be handled. In some way, we have to identify particular exceptions and catch them within the

catch block. This can be achieved by using the instanceof operator as shown in the following example:

```
catch (Exception e)
{
    if (e instanceof ArithmeticException)
    {
        .....
     }
    else
    {
        if (e instanceof NumberFormatException)
        {
            .....
        }
        else
        .....
        .....
     }
}
```

If a single statement throws multiple exceptions, it is better to have multiple catch blocks associated with the try block.

> The order of catch blocks is very important when there are multiple catch blocks.

This has been discussed earlier at the start of section 5.6.

5.6.5 Nested try statements

The try statement can be nested. That is, one try statement can be placed inside the block of another try. Each time 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 catch handlers belonging to the next try statement are inspected for a match. 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. In Program 5.5 a pair of nested try statements have been used.

Program 5.5 Using nested try-catch statements.

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

```
    {
        System.out.println("\nProgram Execution starts here\n");
        int a,b,c;
        try
        {
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
            try
            {
                c = a/b;
                System.out.println( a + "/" + b + "=" + c );
            }
            catch(ArithmeticException e)
            {
                System.out.println("Second Argument Should not be Zero");
            }
          ////
        }
        catch(NumberFormatException e)
        {
            System.out.println("Arguments passed should be valid Numbers");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Pass Proper Arguments");
        }
        System.out.println("\nProgram Execution Completes here");
    }
}
```

5.6.6 The keyword throw

An exception can be caught only if it is identified, or, in other words, thrown. Exceptions can be thrown by the Java run-time system for the code that has been written. This can be achieved also by using the throw statement explicitly.

This statement starts with the keyword throw followed by a single argument. The argument must be an object instantiated from the Throwable class or its sub-class. The syntax for using the throw statement is the following:

```
throw <Exception object>
```

In a specific case where an instance of 'Exception object' is to be thrown, it takes the following form:

```
throw new <Exception object>
```

Usually the above statement is used to pass a string argument along with the exceptional object, so that the string can be displayed when the exception is handled.

Example

```
throw new IndexOutOfBoundsException("Index out of range");
```

The throw statement is generally written with a conditional statement such as an if statement and switch statement. It is more useful when it comes to throwing user-defined exceptions which cannot be recognized by the Java run-time system. This aspect is dealt with in section 5.8.

Program 5.6 illustrates the use of conditional statements:

Program 5.6 Using throw keyword.

```
public class TestThrow
{
    public static void main(String args[ ]) throws Exception
    {
        int number1 = 15, number2 = 10;
        if(number1 > number2) // Conditional statement.
            throw new Exception("number1 is 15");
        else
            System.out.println("number2 is 15");
    }
}
```

The output of Program 5.6 is

```
Exception in thread "main" java.Exception: number1 is 15
at TestThrow.main(TestThrow.java:8)
```

In Program 5.6 the keyword throw is used to throw an exception with the message. With throw, the programmer is allowed to throw the exception and the system prints the messages and terminates the program.

5.7 Exception and Inheritance

An exception handler designed to handle a specific type of object may be preempted by another handler whose exception type is a super-class of that exception object. This happens if the exception handler for that exception type appears earlier in the list of exception handlers. That is,

while using multiple catch statements, it is important to be aware of the order of exception classes and arrange them correctly.

> The exception sub-classes must come before any of their super-classes.

If a super-class exception is coded first in the catch block then, when an exception occurs, the super-class exception gets, executed whereas the sub-class exception catch block never gets executed. In Java, this is considered a compile-time error. Hence, the order of the catch blocks is important. Thus, the catch blocks that are written to handle exception types that are far from the root of the hierarchy should be placed first in the list of catch blocks, and the classes nearer to the root must be placed last. That is, the leaves come first and the root is last in the catch block order.

> The programmer must take care when providing handlers with exception types at different levels in the inheritance hierarchy.

Look at Program 5.7, which gives a compile time error because the general Exception is caught before Arithmetic Exception. Reversing the order of the catch blocks would ensure that the correct exception handler catches the exception. It is important to handle the right exception in the right context.

Program 5.7 Incorrect ordering of catch blocks.

```
public class Divide
{
    public static void main(String[ ] args)
    {
        System.out.println("\nProgram Execution starts here\n");
        int a,b,c;
        try
        {
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
            c = a/b;
            System.out.println( a + "/" + b + "=" + c );
        }
        catch(ArithmeticException e)
        {
            System.out.println("Exception caught:"+e);
        }
```

```
        catch(Exception e)
        {
            System.out.println("Second Argument Should not be Zero");
        }
    }
}
```

5.8 Throwing User-defined Exceptions

User-defined exceptions are useful to handle business application-specific error conditions. For example, when performing a withdrawal from a bank account, it is required to validate the minimum balance in that account. To handle this requirement, a user-defined exception MinimumBalance may be created. The user-defined exceptions can be generated using throw statement and caught them in the similar way as normal exceptions. When creating user-defined exceptions, the name of the exception type should not be a reserved exception type name. User-defined exceptions are provided by an application provider or a Java API provider.

We can write our own exceptions and throw and catch them wherever it is necessary. Some steps are followed in defining our exceptions. Generally, all the exceptions are sub-classes of the class Throwable. Also, Throwable already has two sub-classes—Exception and Error. User-defined exceptions should be sub-classes of Exception.

> Specifying class names of user-defined exceptions should be done carefully so that it does not match already existing exceptions.

If it is the same as predefined exception class names, a compilation error occurs.

An user-defined exception can be created as an instance of the class Throwable as follows:

```
Throwable UserException = new Throwable( );
```

or

```
Throwable UserExcepton = new Throwable('This is user exception Message');
```

Alternatively, the user-defined exception can be defined using the keyword extends as follows:

```
class UserException extends Exception
{
    public UserException()
    {
        ...
    }
    public UserException (String str)
```

```
    {
        ...
    }
    ...
}
```

Program 5.8 defines an user-defined exception UserException which extends the class Exception. Since there should be only one public class in a file, the class UserException is taken as default specifier. In Program 5.7 whenever the second parameter passed at run-time is zero then an instance of UserException is created and is thrown by using the keyword throw and it is caught by the catch block.

Program 5.8 Creating user-defined exceptions using extends.

```
class UserException extends Exception
{
    public UserException(String s)
    {
        super(s);
        System.out.println("From User Exception Constructor");
    }
}
public class Divide
{
    public static void main(String[ ] args)
    {
        System.out.println("\nProgram Execution starts here\n");
        int a,b,c;
        try
        {
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
            if (b==0)
            {
                throw new UserException("Second Argument should not be Zero...");
            }
            c = a/b;
            System.out.println( a + "/" + b + "=" + c );
        }
        catch(UserException e)
```

```
        {
            System.out.println("From UserException catch Statement");
            System.out.println(e);
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("\n Program Execution Completes here");
    }
}
```

The output of Program 5.8 is the following:

```
C:\javatest>java Divide 4 0
Program Execution starts here
From User Exception Constructor
From UserException catch Statement
UserException: Second Argument should not be Zero...
Program Execution Completes here
C:\javatest>
```

Note that if a user-defined exception class is a sub-class of RuntimeException, it would not be a checked exception. If it is a sub-class of Exception but not a sub-class of RuntimeException it will be a checked exception.

5.9 Redirecting and Rethrowing Exceptions

5.9.1 Redirecting exceptions using throws

Recall that the code capable of throwing an exception is kept in the try block and the exceptions are caught in the catch block. When there is no appropriate catch block to handle the (checked) exception that was thrown by an object, the compiler does not compile the program. To overcome this, Java allows the programmer to redirect exceptions that have been raised up the call stack, by using the keyword **throws.** Thus, an exception thrown by a method can be handled either in the method itself or passed to a different method in the call stack.

To pass exceptions up to the call stack, the method must be declared with a throws clause. All the exceptions thrown by a method can be declared with a single throws clause; the clause

consists of the keyword throws followed by a comma-separated list of all the exceptions, as shown below:

```
Void RedirectExMethod( ) throws Exception_A, Exception B, Exception C
{
    .....
}
```

Program 5.9 illustrates two methods redirecting an exception Exception from the method throwing it, that is, ConvertAndDivide, to the calling method, main.

Program 5.9 Redirecting exceptions.

```
public class Divide
{
    public static void main(String[ ] args)
    {
        System.out.println("\nProgram Execution starts here\n");
        try
        {
            convertAndDivide(args[0],args[1]);
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage()+"\n");
            e.printStackTrace();
        }
        System.out.println("\nProgram Execution Completes here");
    }
    static void convertAndDivide(String s1,String s2) throws Exception
    {
        int a,b,c;
        a = Integer.parseInt(s1);
        b = Integer.parseInt(s2);
        c = divide(a,b);
        System.out.println( a + "/" + b + "=" + c );
    }
    static int divide(int x, int y) throws Exception
    {
        if (y==0)
```

```
        {
            throw new Exception("Second Argument is Zero...");
        }
        return x/y ;
    }
}
```

The output of Program 5.9 is the following:

```
C:\javatest>java Divide 4 0
Program Execution starts here
Second Argument is Zero...
java.lang.Exception: Second Argument is Zero...
    at Divide.divide(Divide.java:35)
    at Divide.convertAndDivide(Divide.java:27)
    at Divide.main(Divide.java:9)
Program Execution Completes here

C:\javatest>java Divide 4 a
Program Execution starts here
a
java.lang.NumberFormatException: a
    at java.lang.Integer.parseInt(Integer.java:426)
    at java.lang.Integer.parseInt(Integer.java:476)
    at Divide.convertAndDivide(Divide.java:26)
    at Divide.main(Divide.java:9)
Program Execution Completes here

C:\javatest>
```

The method getMessage() in Program 5.9 just prints the message that is given at the time of the throw statement. If no message is given, then the data value that is responsible for the exception is shown; printStackTrace() prints the all the names of the methods that are called in generating the exception. The method names are printed in the reverse order of their call.

While defining exception handlers, it is instructive to take into account the scope of a method. The scope of the exception-handling mechanism is not limited to the exceptions that can be thrown by the code written into the method; it extends to the methods that called the method in which the exception is thrown. That is, the scope also includes exceptions thrown by methods called by that method and so on.

5.9.2 Rethrowing an exception

An exception that is caught in the try block can be thrown once again and can be handled. The try block just above the rethrow statement will catch the rethrown object. If there is no try block just above the rethrow statement then the method containing the rethrow statement handles it. To propagate an exception, the catch block can choose to rethrow the exception by using the throw statement. Note that there is no special syntax for rethrowing. Program 5.10 illustrates how an exception can be rethrown.

Program 5.10 Rethrowing exceptions.

```
public class Divide
{
    public static void main(String[] args)
    {
        System.out.println("\nProgram Execution starts here\n");
        int a,b,c;
        try
        {
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
            try
            {
                c = a/b;
                System.out.println( a + "/" + b + "=" + c );
            }
            catch(ArithmeticException e)
            {
                System.out.println("Second Argument Should not be Zero");
                System.out.println("Rethrowing the object again");
                throw e;
            }
           ////
        }
        catch(NumberFormatException e)
        {
            System.out.println("Arguments passed should be valid Numbers");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Pass Proper Arguments");
        }
```

```
        System.out.println("\nProgram Execution Completes here");
    }
}
```

Often first-time readers may get confused about the use of the three keywords: **throw**, **throws** and **Throwable**. It is therefore useful to dwell on the three concepts together in order to clarify their meaning.

Throwable is a class. Though the **Throwable** class is derived from the java.lang.Object class in the Java class library, **Throwable** is the super-class of all classes that handle exceptions.

The keyword **throw** is a statement that throws an exception. Note that an exception can be thrown either by the **throw** statement or when an error occurs during the execution of any other statement.

The keyword **throws** is a clause specified in the method definition which indicates that the method throws the exceptions mentioned after the keyword **throws**, which are handled in the called methods.

5.10 Advantages of the Exception-Handling Mechanism

The main advantages of the exception-handling mechanism in object oriented programming over the traditional error-handling mechanisms are the following:

- *The separation of error-handling code from normal code* Unlike traditional programming languages, there is a clear-cut distinction between the normal code and the error-handling code. This separation results in less complex and more readable (normal) code. Further, it is also more efficient, in the sense that the checking of errors in the normal execution path is not needed, and thus requires fewer CPU cycles.
- *A logical grouping of error types* Exceptions can be used to group together errors that are related. This will enable us to handle related exceptions using a single exception handler. When an exception is thrown, an object of one of the exception classes is passed as a parameter. Objects are instances of classes, and classes fall into an inheritance hierarchy in Java. This hierarchy can be used to logically group exceptions. Thus, an exception handler can catch exceptions of the class specified by its parameter, or can catch exceptions of any of its sub-classes.
- *The ability to propagate errors up the call stack* Another important advantage of exception handling in object oriented programming is the ability to propagate errors up the call stack. Exception handling allows contextual information to be captured at the point where the error occurs and to propagate it to a point where it can be effectively handled. This is different from traditional error-handling mechanisms in which the return values are checked and propagated to the calling function.

The following is a list of exceptions under the Throwable class:

Throwable

Error

AWTError

LinkageError

ClassCircularityError
ClassFormatError
ExceptionInInitializerError
IncompatibleClassChangeError
NoClassDefFoundError
UnsatisfiedLinkError
VerifyError

ThreadDeath

VirtualMachineError

InternalError
OutOfMemoryError
StackOverflowError
UnknownError

Exception

AclNotFoundException
ActivationException

UnknownGroupException
UnknownObjectException

AlreadyBoundException
ApplicationException
AWTException
BadLocationException
ClassNotFoundException
CloneNotSupportedException

ServerCloneException

DataFormatException
ExpandVetoException
FontFormatException
GeneralSecurityException

CertificateException
CRLException
DigestException
InvalidAlgorithmParameterException
InvalidKeySpecException

- InvalidParameterSpecException
- KeyException
- KeyStoreException
- NoSuchAlgorithmException
- NoSuchProviderException
- SignatureException
- UnrecoverableKeyException

IllegalAccessException
InstantiationException
IntrospectionException
InvalidMidiDataException
InvocationTargetException
IOException

- ChangedCharSetException
- CharConversionException
- EOFException
- FileNotFoundException
- InterruptedIOException
- MalformedURLException
- ObjectStreamException
- ProtocolException
- RemoteException
- SocketException
- SyncFailedException
- UnknownHostException
- UnknownServiceException
- UnsupportedEncodingException
- UTFDataFormatException
- ZipExcept

LastOwnerException
LineUnavailableException
MidiUnavailableException
MimeTypeParseException
NamingException

- AttributeInUseException
- AttributeModificationException
- CannotProceedException
- CommunicationException
- ConfigurationException
- ContextNotEmptyException
- InsufficientResourcesException

InterruptedNamingException
InvalidAttributeIdentifierException
InvalidAttributesException
InvalidAttributeValueException
InvalidNameException
InvalidSearchControlsException
InvalidSearchFilterException
LimitExceededException
LinkException
NameAlreadyBoundException
NameNotFoundException
NamingSecurityException
NoInitialContextException
NoSuchAttributeException
NotContextException
OperationNotSupportedException
ReferralException
SchemaViolationException
ServiceUnavailableException
NoninvertibleTransformException
NoSuchFieldException
NoSuchMethodException
NotBoundException
NotOwnerException
ParseException
PartialResultException
PrinterException
PrinterAbortException
PrinterIOException
PrivilegedActionException
RemarshalException
RuntimeException
ArithmeticException
ArrayStoreException
CannotRedoException
CannotUndoException
ClassCastException
CMMException
ConcurrentModificationException
EmptyStackException
IllegalArgumentException

IllegalMonitorStateException
IllegalPathStateException
IllegalStateException
ImagingOpException
IndexOutOfBoundsException
MissingResourceException
NegativeArraySizeException
NoSuchElementException
NullPointerException
ProfileDataException
ProviderException
RasterFormatException
SecurityException
SystemException
UndeclaredThrowableException
UnsupportedOperationException
SQLException
BatchUpdateException
SQLWarning
TooManyListenersException
UnsupportedAudioFileException
UnsupportedFlavorException
UnsupportedLookAndFeelException
UserException
AlreadyBound
BadKind
Bounds
CannotProceed
InconsistentTypeCode
Invalid
InvalidName
InvalidName
InvalidSeq
InvalidValue
NotEmpty
NotFound
PolicyError
TypeMismatch
UnknownUserException
WrongTransaction

6 Multithreading

Multithreading is a powerful programming tool that makes it possible to achieve concurrent execution of multiple units of a program. Each portion of a program—designated a thread—may execute concurrently with others. Multithreading basically enables a program to do more than one task at a time and also to synchronize these tasks. Java builds thread support directly into the language. Multithreading support for Java includes thread creation, thread prioritizing, thread scheduling, resource locking (thread synchronization) and establishing inter-thread communication.

6.1 Introduction: an Overview of Threads

A thread is a single sequential flow of control within a program. It differs from a process in that a process is a program executing in its own address space whereas a thread is a single stream of execution within a process.

A thread can be defined as a process in execution within a program.

A thread by itself is not a complete program and cannot run on its own. But it runs within a program; however each thread has a beginning, and an end, and a sequential flow of execution at a single point of time. At any given instant within the run-time of the program, only one thread will be executed by the processor.

Multithreading allows a program to be structured as a set of individual units of control that run in parallel; however, the Central Processing Unit (CPU) can only run one process at a time. The CPU time is divided into slices and a single thread will run in a given time slice. Typically, the CPU switches between multiple threads and runs so fast that it appears as if all threads are running at the same time. It is better, as a programming practice, to identify different parts of the program that can perform in parallel and implement them into independent threads.

Multithreading differs from multitasking: multitasking allows multiple tasks (which can be processes or programs) to run concurrently whereas multithreading allows multiple units within a program (threads) to run concurrently.

Unlike the processes in some operating systems (for example, Unix), the threads in Java are 'light-weight processes' as they have relatively low overheads and share common memory space. This facilitates an effective and inexpensive communication between threads.

6.2 Creating Threads

In Java, threads are objects and can be created in two ways:

1. by extending the class Thread
2. by implementing the interface Runnable

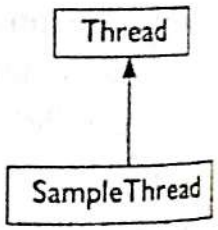
In the first approach, a user-specified thread class is created by extending the class Thread and overriding its run() method. In the second approach, a thread is created by implementing the Runnable interface and overriding its run() method. In both approaches the run() method has to be overridden. Usually, the code that is to be executed by a thread is written in its run() method. The thread terminates when its run() method returns.

In Java, methods and variables are inherited by a child class from a parent class by extending the parent. By extending the class Thread, however, one can only extend or inherit from a single parent class (in this case, the class Thread is the parent class). This limitation of using extends within Java can be overcome by implementing interfaces. This is the most common way to create threads. A thread that has been created can create and start other threads.

6.2.1 Creating a new thread extending Thread

The first method of creating a thread is simply by extending the Thread class. The Thread class is defined in the package java.lang. The class that inherits overrides the run( ) method of the parent Thread for its implementation. This is done as shown in the code fragment given below. By its side a representation of the inheritance that is being implemented.

```
public class SampleThread extends Thread
{
    public SampleThread(String name)
    {
        super(name);
    }
    public void run( )
    {
        // code to be run when this thread is started
    }
}
```



A thread can be started by applying the start() method on the thread object. The following code segment creates an object of the thread class and starts the thread object.

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

```
    {
        ……..
        ……..
        SampleThread st = new SampleThread();
        st.start();
        ……...
    }
}
```

Here, the thread object st of the thread class SampleThread is created as

```
SampleThread st = new SampleThread();
```

To start the thread object st, the start() method can be applied on this object as

```
st.start();
```

When the above statement is executed, the run() method of the SampleThread class is invoked. The start() method implicitly calls the run() method. Note that the run() method can never be called directly.

Look at Program 6.1, which creates a thread class ThreadExample which extends the class Thread and overrides the method Thread.run(). The run() method of this program is where all the work of the ThreadExample class thread is done. This instance of the class is created in the ExampleT class. The start() method on this instance starts the new thread. The child thread prints the values from 0 to 5.

Program 6.1 Using extends to write a single-thread program.

```
import java.lang.*;
class ThreadExample extends Thread
{
    public ThreadExample(String name)
    {
        super(name);
    }
    public void run()
    {
        System.out.println(Thread.currentThread());
        for(int  i=0;i<=5;i++)
            System.out.println(i);
    }
}
public class ExampleT
```

```
{
    public static void main(String args[ ])
    {
        ThreadExample t = new ThreadExample("First");
        t.start( );
        System.out.println("This is :" + Thread.currentThread());
    }
}
```

The output of Program 6.1 is as follows:

```
This is :Thread[main, 5, main]
Thread[First, 5, main]
0
1
2
3
4
5
```

The first line of the output shows the name of the thread (main), the priority of the thread (5) and the name of the ThreadGroup (main). In the second line, First, 5 and main are the name, priority and name of the ThreadGroup of the child thread.

The created thread does not automatically start running. To run a thread, the class that creates it must call the method start() of the Thread. The start() method then calls the run() method. When applying the start() method on a thread object, a new flow of control starts processing the program. The start() method can be invoked either from the constructor or any method in which the thread is created. Figure 6.1. shows the running of both main and child threads.

In Program 6.1 the main method creates an object a thread class ThreadExample. After executing the statement

```
ThreadExample t = new ThreadExample("First");
```

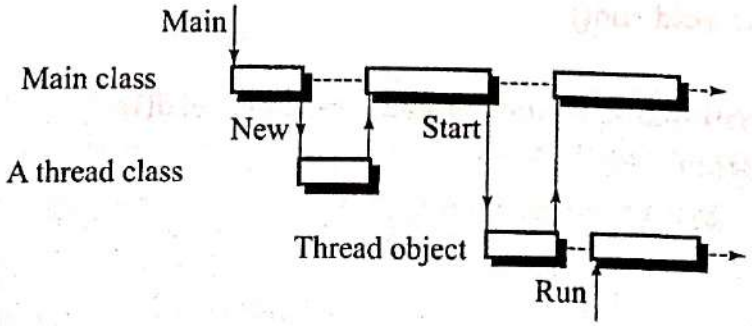


Fig. 6.1 Running the 'main' thread and newly created thread.

thread object *t* is in the newborn state of the thread life cycle (see section 6.3). When a thread is in newborn state, it does not hold any system resource and the thread object is said to be empty. A thread can be started only when it is in the newborn state by calling the start() method. Calling any method other than the start() method will cause an exception IllegalThreadStateException. The start() method creates the necessary system resources to run the thread, schedules the thread to run, and calls the thread's run() method. The thread object t calls the start() method of the ThreadExample class. Thread object t's run method is defined in the ThreadExample class. After execution of t.start() statement, the thread is in the runnable state. Henceforth, both the thread object as well as main thread are in the runnable state.

Every Java applet or application is multithreaded. For instance, main itself is a thread created by extending the Thread class.

6.2.2 Creating a thread implementing Runnable interface

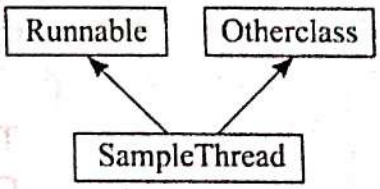
The interface Runnable is defined in the java.lang package. It has a single method—run().

```
public interface Runnable
{
    public abstract void run();
}
```

If we want multithreading to be supported in a class that is already derived from a class other than Thread, we must implement the Runnable interface in that class.

The majority of classes created that need to be run as a thread will implement Runnable since they may be extending some other functionality from another class. Whenever the class defining run() method needs to be the sub-class of classes other than Thread, using Runnable interface is the best way of creating threads. The syntax and the inheritance structure are given below.

```
public class SampleThread extends
OtherClass implements Runnable
{
    .....
    public void run()
    {
       // code to be run when this thread is started
    }
}
```



The class Thread itself implements the Runnable interface (package java.lang) as expressed in the class header:

```
public class Thread extends Object implements Runnable
```

As the Thread class implements Runnable interface, the code that controls the thread is placed in the run() method. In order to create a new thread with Runnable interface, we need to instantiate the Thread class. This thread class will have the following constructors:

```
public Thread(Runnable obj);
public Thread(Runnable obj, String threadname);
public Thread(ThreadGroup tg, Runnable obj, String threadname);
```

Here, obj is the object of the class which implements the Runnable interface, threadname is the name given to the thread and tg is the name of the ThreadGroup.

Program 6.2 illustrates the creation of threads using Runnable interface.

Program 6.2 Using Runnable interface to write a single-thread program.

```
class ThreadExample implements  Runnable
{
    Thread t;
    public ThreadExample(String threadname)
    {
        t = new Thread(this, threadname);
    }
    public void run()
    {
        System.out.println( Thread.currentThread() );
        for ( int  i = 0; i <= 5; i++ )
            System.out.println(i);
    }
}
public class ExampleT2
{
    public static void main(String args[ ])
    {
        ThreadExample obj = new ThreadExample("First");
        Obj.t.start( );
        System.out.println("This is :" + Thread.currentThread());
    }
}
```

The output of Program 6.2 is as shown below.

```
This is :Thread[main,5,main]
Thread[First,5,main]
```

```
0
1
2
3
4
5
```

In Program 6.2, in place of the Thread class constructor parameters we passed this and First. Here this refers to the ThreadExample class on which the thread is created. Here, the abstract run() method is defined in the Runnable interface and is being implemented.

By implementing Runnable, there is greater flexibility in the creation of the class ThreadExample.

In the above example, the opportunity to extend the ThreadExample class, if needed, still exists.

The method of extending the Thread class is good only if the class executed as a thread does not ever need to be extended from another class.

6.2.3 Stopping threads: the join() method

Generally, when the execution of a program starts the thread main is started first. Child threads are started after the main thread. So it is unusual to stop the main thread before the child threads. The main thread should wait until all child threads are stopped. The join() method can be used to achieve this. The syntax of this method is as follows:

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

The join() method waits until the thread on which it is called terminates. That is, the calling thread waits until the specified thread joins it. A thread (either main thread or child threads) calls a join() method when it must wait for another thread to complete its task. When the join() method is called, the current thread will simply wait until the thread it is joining with either completes its task or is not alive. A thread can be in the not alive state due to any one of the following:

- the thread has not yet started,
- stopped by another thread,
- completion of the thread itself

The following is a simple code that uses the join() method:

```
try
{
    t1.join();
```

```
        t2.join();
        t3.join();
    }
    catch(interupptedException e)
    {
    }
```

Here t(1), t(2), t(3) are the three child threads of the main thread which are to be terminated before the main thread terminates. If we check the isAlive() on these child threads after the join() method, it will return false.

There is another form of the join() method, which has a single parameter that specifies how much time the thread has to wait. This is the following:

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

6.2.4 Naming a thread

By default, each thread has a name. Java provides a Thread constructor to set a name to a thread. The name can be passed as a string parameter to this constructor, in the following manner:

```
Thread t = new Thread("First");
Thread t = new Thread(Runnable r, "SampleThread");
```

The setName method of the Thread class can also be used to set the name of the thread, in the following manner:

```
void setName(String threadname);
```

6.3 Thread Life-Cycle

A thread is always in one of five states: newborn, runnable, running, dead and blocked. Figure 6.2 shows the life-cycle of a thread.

6.3.1 The newborn state

When a thread is called, it is in the newborn state, that is, when it has been created and is not yet running. In other words, a start() method has not been invoked on this thread. In this state, system resources are not yet allocated to the thread. When a thread is in the newborn state, calling any method other than start() method causes an IllegalThreadStateException.

6.3.2 The runnable state

A thread in the runnable state is ready for execution but is not being executed currently. Once a thread is in the runnable state, it gets all the resources of the system (as, for example, access to

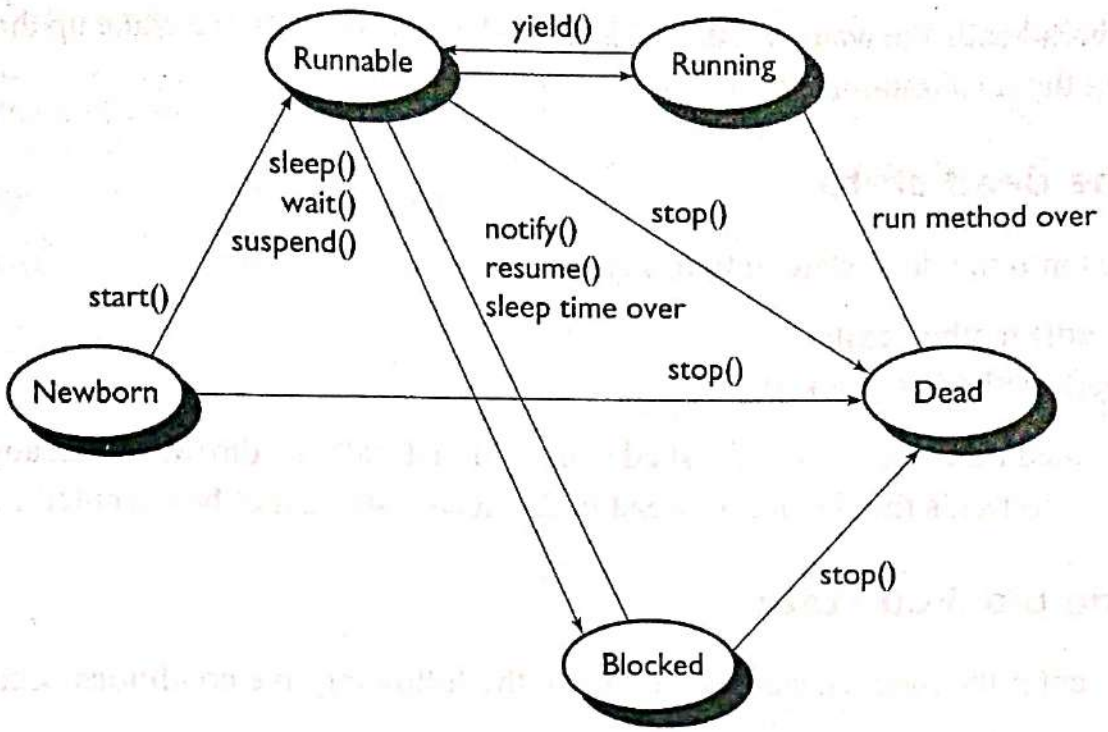


Fig. 6.2 Life-cycle of a thread.

the CPU) and moves on to the running state. All runnable threads are in a queue and wait for CPU access. When the start() method is called on the newborn thread, it will be in the runnable state. It is not in the running state yet because system resources such as the CPU (which might be serving another thread at that point) are not available.

6.3.3 The running state

After the runnable state, if the thread gets CPU access, it moves into the running state. The thread will be in the running state unless one of the following things occur:

- It dies (that is, the run() method exits).
- It gets blocked to the input/output for some reason.
- It calls sleep().
- It calls wait().
- It calls yield().
- It is preempted by a thread of higher priority.
- Its quantum (time slice) expires.

A thread with a higher priority than the running thread can preempt the running thread if any of the following situations arise:

- If the thread with the higher priority comes out of the sleeping state.
- The I/O completes for a thread with a higher priority waiting for that I/O.
- Either notify() or notifyAll() is called on a thread that called wait.

When a thread calls the wait() method it goes in the waiting state. To wake up this thread, some other running thread should notify it.

6.3.4 The dead state

A thread goes into the dead state in two ways:

- If its run() method exits.
- A stop() method is invoked.

The run() method exits when it finished execution naturally or throws an uncaught exception. The stop() method kills the thread. A thread in the dead state cannot be executed further.

6.3.5 The blocked state

A thread can enter the blocked state when one of the following five conditions occurs:

- When sleep() is called,
- When suspend() is called,
- When wait() is called,
- The thread calls an operation, (For example, during input/output, a thread will not return until the I/O operation completes.),
- The thread is waiting for monitor.

A thread in the blocked state waits for some action to happen so that it can get ready. That is, if the thread requires some of the I/O operation, it will enter into the blocked state by giving the access to the CPU to another thread. After completion of the I/O operations it will enter into the runnable state.

A thread must move out of a blocked state into the runnable (or running) state using the opposite of whatever phenomenon put it into the blocked state.

- If a thread has been put to sleep(), the specified timeout period must expire.
- If a thread has called wait(), then some other thread using the resource for which the first thread is waiting must call notify() or notifyAll().
- If a thread is waiting for the completion of an input or output operation, then the operation must finish.

6.3.6 Manipulating threads

Table 6.1 lists some of the methods used to manipulate threads.

In addition to the methods described in Table 6.1, there are some which need to be discussed briefly. They include the sleep(), suspend(), resume(), wait(), notify(), notifyAll() and yield() methods.

Table 6.1 Some methods that are provided in the Thread class to manipulate threads

Name of the method	Function
getName()	returns the name of the thread
setPriority()	to set the priority of the thread
getPriority()	returns the thread priority
isAlive()	determines whether the thread is still running or not; isAlive() returns true if the method is running, runnable or blocked. It returns false if the method is a new thread or dead
join()	wait for a thread to terminate
run()	entry point for the thread
start()	start a thread by calling its run method
stop()	terminate the thread abruptly

6.3.6.1 sleep()

A thread being executed can invoke sleep() to block the thread for some time and free the CPU. This thread goes into the sleep (blocked) state for the specified amount of time, after which it moves to the runnable state. The sleep method has the following prototypes:

```
public static void sleep (long millisec)  throws InterruptedException;
public static void sleep (long millisec, int nanosec)  throws InterruptedException;
```

6.3.6.2 suspend() and resume()

The Thread class has a method suspend() to stop the thread temporarily and a method resume() to restart it at the point at which it is halted. The resume() method must be called by some thread other than the suspended one. The suspend() method is not recommended in application programming.

6.3.6.3 wait(), notify() and notifyAll()

When a running thread calls wait(), it enters into a waiting (blocked) state for the particular object on which wait was called. A thread in the waiting state for an object becomes ready on a call to notify() issued by another thread associated with that object. All threads waiting for a given object become ready when receiving a call to notifyAll() from another thread associated with that object.

6.3.6.4 yield()

Some CPU-intensive operations of one thread may prevent other threads from being executed. To prevent this from happening, the first one can allow other threads to execute by invoking the yield() method. This method allows another thread of the same priority to be run. The thread from which yield() is invoked moves from the running state to the runnable state.

The methods sleep() and yield() are static methods.

6.4 Thread Priorities and Thread Scheduling

Each new thread inherits the priority of the thread that creates it. That is, when a new thread is created, the new thread has priority equal to the priority of the creating thread. Every Java thread has a priority between 1 and 10, and by default each thread is given normal priority, that is, 5. The constants defined in the Thread class are the following:

- Thread.MIN_PRIORITY 1 (minimum priority)
- Thread.MAX_PRIORITY 10 (maximum priority)
- Thread.NORM_PRIORITY 5 (default)

Higher the integer value assigned, higher is the priority of the thread.

6.4.1 setPriority and getPriority

A thread's priority can be set with setPriority method, which takes an int argument. getPriority method returns the thread's priority. If the argument to the setPriority method is not in the range from 1 to 10, it throws an IllegalArgumentException.

6.4.2 Time-slicing

Some Java platforms support a concept called time-slicing and some do not. Without time-slicing, each thread in a set of equal-priority threads runs to completion (unless the thread leaves the running state and enters the waiting, sleeping or blocked state) before that thread's peers get a chance to be executed. For more than two threads having equal priority the thread will be selected on round robin scheme. With time-slicing, each thread receives a brief amount of processor time, called a quantum, during which that thread can execute. At the end of the quantum, even if that thread has not been executed fully, the processor is taken away from that thread and given to the next thread of equal priority if such a thread is available. Note that, in this context, 'the thread's peers' refers to the active threads (which are alive). The 'next thread' means the next thread that is ready for execution. The scheduler decides which one is the next thread for obtaining processor time.

6.4.3 The scheduler

When many threads are ready for execution, the Java run-time system selects the highest priority runnable thread (fixed priority scheduling). The job of the Java scheduler is to keep a highest-

priority thread running at all points in time. If time-slicing is available, the scheduler ensures that several threads of equal priority each get executed for a quantum in a round robin fashion. That is, if there is more than one thread with the same priority, each thread gets a turn in round robin order. Lower priority threads are not run as long as there is a runnable higher priority thread. Java scheduling is also preemptive, that is, a higher priority thread preempts the lower priority one. Thus, a thread will run until one of the following conditions occurs:

- A higher priority thread becomes runnable.
- It yields, or its run method exits.
- Its time slice has expired (only on systems that support time-slicing).

6.5 Thread Synchronization

When more than one thread has to use a shared resource, Java finds a way of ensuring that only one thread uses the resources at one point of time; this is called *synchronization*.

In Java, each object is associated with a lock. The term lock refers to the access granted to a particular thread that has entered a synchronized method. Monitor refers to a portion of the code in a program. It contains one specific region (related to data or some resource) that can be occupied by only one thread at a time. This specific region within a monitor is known as *critical section*. A thread has exclusive lock from the time it enters the critical section to the time it leaves. That is, the data (or resource) is exclusively served for the thread, and any other thread has to wait to access that data (or resource). Some common terminology while using monitors is as follows: entering the critical section is known as *acquiring the monitor*, working with the critical section (data or resource) is known as *owning the monitor* and leaving the critical section is known as *releasing the monitor*.

The key concept in synchronization is the monitor. The monitor holds exclusive lock that can be owned by only one object at a time. The monitor allows only one thread to execute a synchronized method on the object at one time. This is accomplished by giving the ownership of the monitor to that object, or in other words, locking the object when the synchronized method is invoked. An object thus locked is also denoted to have obtained the lock.

When there are several synchronized methods attempting to act on an object, only one synchronized method may be active on an object at one instant of time and all other threads attempting to invoke synchronized methods must wait. Thus, other threads that attempt to lock the already locked monitor will be suspended. When the execution of that synchronized method is complete, the lock on the object is opened and the monitor allows the highest-priority runnable thread attempting to invoke a synchronized method to proceed.

A thread executing in a synchronized method may determine that it cannot proceed, so it voluntarily calls wait(). This removes the thread from contention for the processor and from contention for the monitor object. The thread now remains in the waiting state while other threads try to enter the monitor object. When the thread executing the synchronized method completes, it can notify a waiting thread to become ready again so that now that thread can attempt to obtain the lock on the monitor object again and be executed. The notify() method acts as a signal to the

waiting thread that the condition to be satisfied for running the waiting thread is satisfied now. Then, the waiting thread can enter into the monitor. If a thread calls notifyAll(), then all threads waiting for the object become eligible to reenter the monitor (that is, they are all placed in a runnable state). Only one of those threads can obtain the lock on the object at a time.

The methods wait(), notify() and notifyAll() are the methods of the class Object. Since every user class inherits the class Object, these methods are available to all classes. Therefore, any object can enter into a monitor.

The monitor maintains a list of all threads waiting to enter the monitor object to execute synchronized methods. A thread is inserted in the list and waits for the object if that thread calls a synchronized method of the object while another thread is already executing is a synchronized method of that object. A thread is inserted in the list also if the thread calls the wait() method while operating inside the object. Threads executing synchronized methods explicitly call wait() inside the monitor upon completion of a synchronized method. Other threads that are blocked because the monitor was busy can now proceed to enter into the object. Threads that explicitly invoked wait() can only proceed when they are notified by the notify or notifyAll methods called by another thread. When it is acceptable for a waiting thread to proceed, the scheduler selects the thread with the highest priority.

The keyword synchronized is used to synchronize the threads and there are two ways to do it: by using synchronized methods and using a synchronized statement.

6.5.1 Synchronized methods

If two or more threads modify an object, the methods that carry the modifications are declared synchronized. Constructors need not be synchronized because a new object is created in only one thread. The general syntax for declaring a method synchronized is the following:

```
Modifier  synchronized  <return_type>  <method_name>(parameters)
```

To get a clear understanding of how this works, consider the Program 6.3.

Program 6.3 Using unsynchronized threads.

```
import java.io.*;
class Class_A
{
    void printValue()
    {
        try
        {
            for(int i = 0; i <= 5; i++ )
            {
                System.out.print( i + " " );
```

```
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException)
        {
        }
    }
}
class ThreadExample implements Runnable
{
    Class_A ob1;
    Thread t;
    ThreadExample(Class_A c)
    {
        this.ob1=c;
        t = new Thread(this);
    }
    public void run()
    {
        ob1.printValue();
    }
    public static void main(String[ ] args)
    {
        Class_A ca  = new Class_A( );
        ThreadExample one = new ThreadExample(ca);
        one.t.start();
        ThreadExample two = new ThreadExample(ca);
        two.t.start();
        ThreadExample three = new ThreadExample(ca);
        three.t.start();
    }
}
```

If we run the above example, we expect the following output:

```
0 1 2 3 4 5 0 1 2 3 4 5
0 1 2 3 4 5
```

But the result is unpredictable; it may result in the following:

```
0 0 0 1 1 1 2 2 2 3 3 3
4 4 4 5 5 5
```

The unpredictability of the output is a direct result of not synchronizing the printValue method. To see this and get the desired result let the printValue() method be declared synchronized.

```
synchronized void printValue()
```

Now, only one thread can execute in the method printValue( ) and the result will be as desired.

The following points are to be noted when implementing synchronized methods

- When a thread calls a non-static synchronized method, the thread acquires a lock on that object. Any other thread that tries to call the same method enters into the wait state.
- The synchronization of static methods is independent of the synchronization of non-static methods. That is, if a thread calls a static synchronized method, the other threads are free to call non-static synchronized methods.
- A sub-class can override a synchronized method in its super-class. The overridden method need not be synchronized.
- If a non-synchronized method of an object uses the super() method to call a synchronized method of its super-class, the object is blocked until the invocation has completed.

6.5.2 Synchronized statements

As an alternative to declaring a method synchronized, Java shows flexibility. The programmer can declare a part of the code as synchronized using synchronized statement. The general syntax of declaring a set of statements synchronized is as follows:

```
synchronized <object><statement>
```

In the case of synchronized statements, we need to specify an object to act as a monitor for this block. Each Java object can act as a monitor and hence any Java object can be specified synchronized.

In the example considered in Program 6.3 we can achieve the desired output in another way, that is by declaring the method synchronized by using synchronized statement. For this the code is modified as in Program 6.4.

Program 6.4

```
import java.io.*;
class Class_A
{
    void printValue()
    {
        try
        {
            for(int i = 0; i <= 5; i++ )
```

```
            {
                System.out.print( i + " " );
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e)
        {
        }
    }
}
class ThreadExample implements Runnable
{
    Class_A ob1;
    Thread t;
    ThreadExample(Class_A c)
    {
        this.ob1=c;
        t = new Thread(this);
    }
    public void run()
    {
        synchronized(ob1)
        {
            ob1.printValue();
        }
    }
    public static void main(String[ ] args)
    {
        Class_A ca = new Class_A();
        ThreadExample one=new ThreadExample(ca);
        one.t.start();
        ThreadExample two=new ThreadExample(ca);
        two.t.start();
        ThreadExample three=new ThreadExample(ca);
        three.t.start();
    }
}
```

The output of Program 6.4 is the following:

```
0 1 2 3 4 5 0 1 2 3 4 5
0 1 2 3 4 5
```

6.5.3 Deadlocks

Deadlocks occur on several occasions. These can be classified into one of the two following situations, namely, when two or more threads are waiting for two or more locks to be freed or circumstances in the program are such that the locks will never be freed.

Suppose one thread is a monitor of object A and another thread is monitor of the object B. In Figure 6.3, A and B are in the monitor because they are executing synchronized methods. At one instant, if A tries to access the synchronized method of B and B tries to access the synchronized method of A then they will enter into the Deadlock state.

If two or more threads are waiting for each other to unlock a synchronized block of code, where each thread relies on another to do the unlocking, then this situation is called deadlock situation. Deadlock situation occurs if the programmer is not careful while writing the synchronized methods.

6.6 Daemon Threads

A daemon thread is a thread that runs only to serve other threads. Daemon threads run in the background when CPU time is available that would otherwise go waste. For instance, the garbage collector in Java is a daemon thread. Unlike conventional user threads, daemon threads do not prevent a program from terminating.

A thread can be designated a daemon by calling the method setDaemon(true); A false argument means that the thread is not a daemon thread. A program can include a combination of daemon threads and non-daemon threads. When the program exits from execution then daemon threads only exists in the program. If a thread is to be designated a daemon, it must be done before its start method is called. Otherwise, it throws an IllegalThreadStateException exception. To find out whether a thread is a daemon thread or not, the isDaemon() method can be used. This method returns true if a thread is a daemon thread and returns false otherwise.

A daemon thread can only create a new daemon thread. Note that when the JVM starts, there is a single non-daemon thread which typically calls the main() method of the class. Each thread has a daemon flag, which is initialized to whatever its parent's flag was set to. Since the main thread of an applet or application has its flag set to false, the only way a thread can be designated a daemon thread is by invoking the following method:

```
setDaemon(true);
```

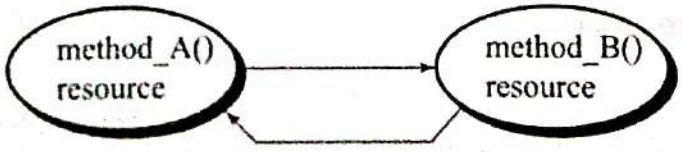


Fig 6.3 Deadlock state diagram.

Daemon threads are just like normal threads except that they are not taken into account when deciding if an application or applet has died. An application dies when all its non-daemon threads have died. The main purpose of daemon threads is to act as servers to the application/applet client. When their client dies, they have nothing left to do, so it is reasonable to kill the daemons also Daemon threads are typically used in image loading among other things.

6.7 Thread Groups

Sometimes it is useful to identify various threads as belonging to a thread group. Every thread object created in java is a member of the ThreadGroup object. In Java, every thread and every thread group joins some other thread group. That is, a thread which exists in a thread group can contain other thread groups. This kind of arrangement forms a hierarchical thread-group structure as shown in Figure 6.4.

As shown in Figure 6.4, the root of the thread-group structure is system thread group. JVM created main thread group, which is a sub-thread group of system thread group. JVM may create other threads under the system thread group based on the need. JVM also created a thread main under the main thread-group. The main thread executes the byte instructions in the main() method. By default, any created thread is a member of system thread group.

The class ThreadGroup contains methods for creating and manipulating thread groups. When creating the construction itself the group is given a unique name via a String argument. A thread

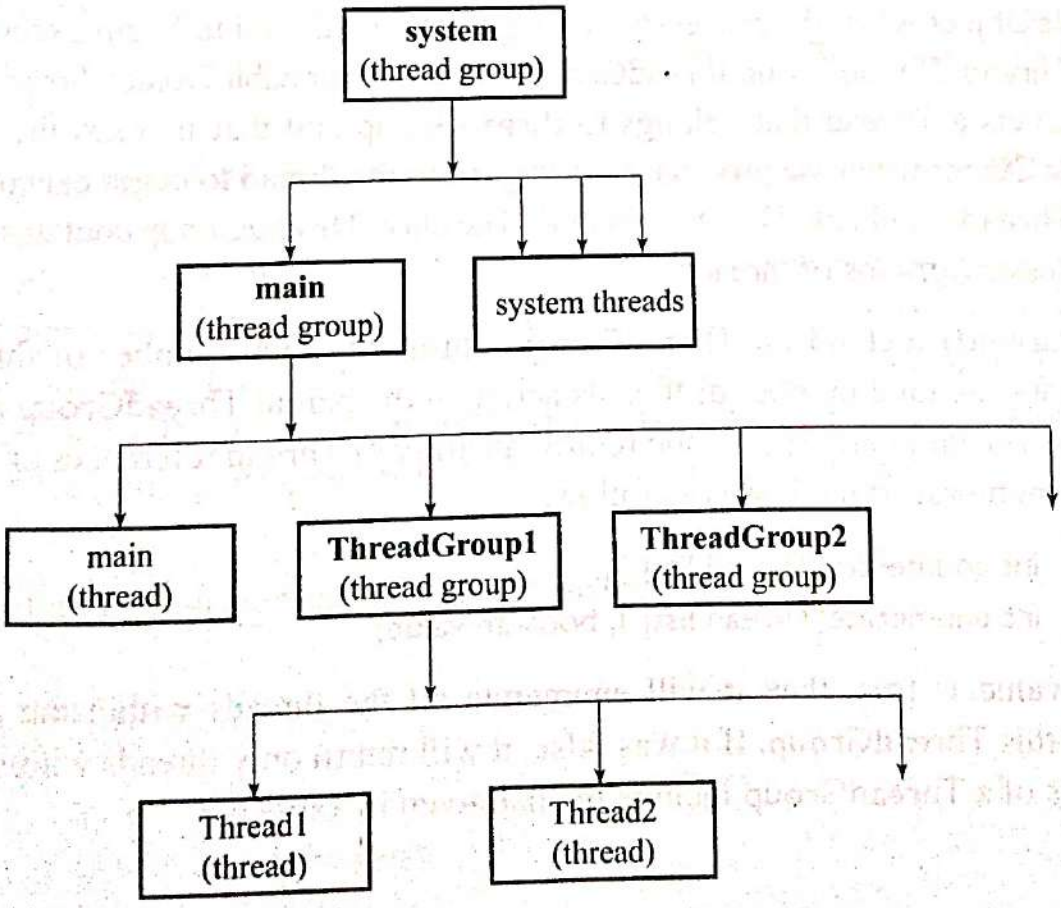


Fig. 6.4 Hierarchical thread-group structure.

group can have as members threads as well as other thread groups, and it forms a hierarchical layout. By default a newly created Thread object belongs to the same group as the Thread that creates it. The threads in a thread group can be dealt with as a group. It is possible to interrupt all the threads in a group. A thread group allows the member threads to act collectively when a method such as suspend(), resume() or stop() is invoked. This is also true for its sub-groups. That is, these operations work in a recursive manner.

A thread group can be the parent to a child thread group. Method calls sent to a parent thread group are also sent to all the threads in that parent's child thread groups. The class ThreadGroup provides the following two constructors.

- public ThreadGroup(String stringName)—constructs a ThreadGroup with name stringName.
- public ThreadGroup(ThreadGroup parentThreadGroup, String stringName)—constructs a child ThreadGroup of parentThreadGroup called stringName.

The class Thread provides three constructors that enable the programmer to instantiate a Thread and associate it with a ThreadGroup. They are the following:

- public Thread(ThreadGroup threadGroup, String stringName)
  It constructs a Thread that belongs to threadGroup and has the name stringName. This constructor is normally invoked for derived classes of Thread whose objects should be associated with a ThreadGroup.
- public Thread(ThreadGroup threadGroup, Runnable runnableObject)
  It constructs a Thread that belongs to the threadGroup and invokes the run method of runnableObject when the processor is assigned to the thread to begin execution.
- public Thread(ThreadGroup threadGroup, Runnable runnableObject, String stringName)
  It constructs a Thread that belongs to threadGroup and that invokes the run method of runnableObject when the processor is assigned to the thread to begin execution. The name of this Thread is indicated by stringName. The class ThreadGroup contains many methods for processing groups of threads.

The activeCount() method on ThreadGroup returns the total number of threads presently active. It includes the total number of threads active in the parent ThreadGroup as well as child ThreadGroups. The enumerate() method returns an array of Thread references of ThreadGroup. The syntax of enumerate() method is as follows:

```
int enumerate(Thread list[ ])
int enumerate(Thread list[ ], boolean value)
```

If boolean value is true, then it will enumerate all the threads within the group and also descendents of this ThreadGroup. If it was false, it will return only threads within this group.

The methods of a ThreadGroup include the list given in Table 6.2.

Table 6.2 The methods of a ThreadGroup.

Name of the method	Function
getThreadGroup( )	returns the reference of the ThreadGroup
getMaxPriority	returns the maximum priority of a ThreadGroup
setMaxPriority	sets a new maximum priority for a ThreadGroup
getName	returns as a String the name of the ThreadGroup
getParent	determines the parent of a thread group.
ParentOf(ThreadGroup t)	returns value true if t is the parent of the ThreadGroup, otherwise it returns false

6.8 Communication of Threads

In Java, inter-thread communication is achieved using the wait(), notify(), and notifyAll() methods of the Object class. These methods are final methods in the Object class and can be called only from within a synchronized code.

- wait() tells the calling thread to give up the monitor and move to sleep until some other thread enters the same monitor and calls notify().
- notify() wakes up the first thread that called wait() in the same object.
- notifyAll() wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

Consider an example of the queuing problem in which two queues are defined: one queue, producer queue, which produces some data, and the consumer queue, which consumes the data. Suppose the producer has to wait until the consumer is finished before it can generate more data. The consumer would waste many CPU cycles while it waited for the producer to produce. This may cause polling, which can be avoided by inter-thread communication. In I/O based programming, threads will wait in a loop and keep on checking whether the task is accomplished or not. This is called polling. Program 6.5 illustrates this queueing problem.

Problem 6.5 Using inter-thread communication I: Polling.

```
class Queue
{
    int n;
    synchronized  int get()
    {
        System.out.println("    Got:"+n);
```

```
        return n;
    }
    synchronized void put(int n)
    {
        this.n = n ;
        System.out.print("    put:"+n);
    }
}
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 < 10)
        {
            q.put( i++ );
        }
    }
}
class Consumer implements Runnable
{
    Queue q;
    Consumer(Queue q)
    {
        this.q = q;
        new Thread( this, "consumer" ).start( );
    }
    public void run( )
    {
        int k = 0;
        while( k < 10)
        {
            q.get( ) ;
```

```
            k++ ;
        }
    }
}
class SampleInterThread
{
    public static void main(String a[ ] )
    {
        Queue sample = new Queue( );
        Producer p =new Producer(sample);
        Consumer c =new Consumer(sample);
        System.out.println("hai");
    }
};
```

The output of Program 6.5 is the following:

```
put:0    Got:0    put:1    Got:1    put:2    Got:2
put:3    Got:3    put:4    Got:4    put:5    Got:5
put:6    Got:6    put:7    Got:7    Got:7    Got:7
put:8    put:9    Got:9
```

Here the result obtained is not the correct one. After the producer put 7, the consumer started and got the same 7 three times in a row. Then, the producer resumed and producer puts 8, 9 simultaneously and after that consumer consumes 9. Problem 6.6 gives the code that yields correct results.

Problem 6.6 Using inter-thread communication II.

```
class Queue
{
    boolean flag = true;
    int n;
    synchronized  int get()
    {
        if ( ! flag )
        {
            System.out.print("   Got:"+n ) ;
            flag = true;
```

```
            notify();
        }
        else
        {
            try
            {
                wait( );
            }
            catch(Exception e)
            {
            }
        }
        return n;
    }
    synchronized void put(int n)
    {
        if ( flag )
        {
            this.n = n;
            System.out.print("put:"+n);
            flag = false;
            notify( );
        }
        else
        {
            try
            {
                wait( );
            }
            catch(Exception e)
            {
            }
        }
    }
}
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 < 10 )
        {
            q.put( i++ );
        }
    }
}
class Consumer implements Runnable
{
    Queue q;
    Consumer(Queue q)
    {
        this.q = q;
        new Thread(this, "consumer").start( );
    }
    public void run(
    {
        int k = 0;
        while( k < 10 )
        {
            q.get( );
            k++;
        }
    }
}
class SampleInterThread
{
    public static void main(String a[ ] )
    {
        Queue sample = new Queue();
        Producer p = new Producer(sample);
        Consumer c = new Consumer(sample);
    }
};
```

The output of Program 6.6 is as shown below:

```
put:0    Got:0    put:1    Got:1    put:2    Got:2
put:3    Got:3    put:4    Got:4    put:5    Got:5
put:6    Got:6    put:7    Got:7    put:7    Got:7
put:8    Got:8    Put:9    Got:9
```

This concludes the discussion on multithreading. More programs illustrating the application of concepts discussed so far are presented below.

6.9 Sample Programs

Program 6.7 Writing all the properties of a thread.

```
import java.io.*;
class  Class_A extends Thread
{
    Thread t;
    public void startThread()
    {
        System.out.println("Starting child Thread");
        start();
    }
    public void run()
    {
        t=Thread.currentThread();
        System.out.println("Name of the thread is:"+t.getName());
        System.out.println("Priority of the thread:"+t.getPriority());
        System.out.println("Thread is Alive :"+t.isAlive());
        t.setName("First");
        t.setPriority(8);
        System.out.println("Changed Name:"+t.getName());
        System.out.println("Changed priority:"+t.getPriority());
        t.stop();
        System.out.println("Test whether thread is alive or not:"+t.isAlive());
    }
}
class ThreadExample5 extends Thread
```

```
{
    public static void main(String a[ ])
    {
        Class_A ob=new Class_A();
        System.out.println("Main thread name:"+Thread.currentThread().getName());
        ob.startThread();
        Thread maint=Thread.currentThread();
        System.out.println("Test main thread is alive or not:"+maint.isAlive());
    }
}
```

The output of the Program 6.7 is the following:

```
Main thread name: main
Starting child Thread
Test main thread is alive or not: true
Name of the thread: Thread-1
Priority of the thread:  5
Thread is Alive: true
Changed Name: First
Changed priority: 8
```

Program 6.8 Scrolling a sentence in an applet so that it looks like a moving banner.

```
import java.io.*;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*<applet code="ExampleApplet"  width=200 height=200>
    </applet>
*/
public class ExampleApplet extends Applet implements Runnable
{
    String msg="Welcome to thread programming";
    Thread t=null;
    int state;
    public void init()
    {
        System.out.println("This is init() method");
    }
```