2.1 INTRODUCTION TO EXCEPTION HANDLING

Java applications are used in embedded systems software, that runs inside specialized devices like hand-held computers, cellular phones, and fancy toasters. In those kind of applications, it's especially important that software errors be handled strongly. As a programmer, we must know what constitutes a bad result, and what it means. It's often awkward to work around the limitations of passing error values in the normal path of data flow. An even worse problem is that certain types of errors can legitimately occur almost anywhere, and it's prohibitive and unreasonable to explicitly test them at every point in the software. Java offers an elegant solution to these problems with *exception handling*.

An ***exception*** is an abnormal condition occurring during the execution of a program that causes it to deviate from the normal execution path. When an exception occurs, it makes further execution of the program impossible. Thus, an exception can be defined as an event that may cause abnormal termination of the program during its execution.

Exception handling means to handle the exceptions (run-time errors) by the programmer himself to recover the computer from malfunction (i.e., computer hanging or computer deadlock) due to exceptions.

In Java, exception handling is done through five keywords, ***try, catch, throw, throws*** and ***finally***. To handle exceptions put the code that may generate an exception in a *try* block. When an exception is generated, the control leaves the try block and searches for the matching ***catch*** block to handle the exception.

Catch is known as ***exception handler*** which 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 hanlders catch the exceptions raised during the execution of the code and handle them according to the instructions given in the hanlder code. In Java, exception hanlders are represented as catch blocks.

Each catch block specifies the type of exception it can handle.

Example : ArithmeticException, ArrayIndexOutOfBoundsException etc.

Each try block is immediately followed by zero or more catch blocks. If the type of thrown exception matches the parameter type in one of the catch blocks then the code for that catch block is executed. If no exception is thrown in the try block, then the catch blocks for that block are skipped and the control goes to the statement after the last catch block. After last catch block, there is a ***finally*** block which is optional, that provides the code that always guarantees to execute regardless of whether or not an exception occur.

An exception can be thrown from a method or a statement in the method by specifying a *throws* clause. The point at which the exception is thrown is called the *throw point* and once the control leaves the throw point the control does not return to the throw point again. And there is no guarantee that there is an exception handler with each try block. If it is there, then exception will be caught and handled. Thus, the Java uses the "Termination Model of Exception Handling".

The below is the general form of an exception handling block,

```
try
{
    // Block of code for checking errors
}
catch(ExceptionTypeOne exOb)
{
    // Exception handler for ExceptionTypeOne
}
catch(ExceptionTypeTwo exOb)
{
    // Exception handler for ExceptionTypeTwo
}
Catch(ExceptionTypeThree exOb)
{
    // Exception handler for ExceptionTypeThree
}
finally
{
    // Block of code to be executed before try block ends
}
```

Here, 'ExceptionType' represents the type of exception that has occurred.

Example

```
class program
{
    Static void func( ) throws IllegalAccessException
    {
        try
        {
            System.out.println("From func( )");
            throw new IllegalAccessException("example");
        }
        finally
        {
            System.out.println("From finally block of func( )");
        }
    }
    Public static void main(String args[ ])
    {
        try
        {
            func( );
        }
        catch(Exception ex)
        {
            System.out.println("Exception caught" + ex);
        }
    }
```

Output

```
From func( )
From finally block of func( )
Exception caught java.lang.IllegalAccessException, example
```

In the program, the main( ) method will first call func( ). In the func( ) method, it is declared that it throws an IllegalAccessException. When an exception occurs within the try block, it will be thrown and the control will come out of try block. Then, the finally block is executed. The exception thrown from try block is caught in the catch block present in main( ) method. That is, after executing the finally block, the control will go to the catch block.

If an exception does not occur within the try block, the finally block will anyhow be executed after the try block. But, the control will not go to the catch block, as the exception is not thrown.

2.1.1 Necessity of Exception Handling

There are two types of errors, compilation errors and run-time errors. The compile time errors are the syntax errors, these errors occur at the compile-time and the program will not compile. The compilation errors are called bugs. These bugs are taken care of by the compiler. The errors may also occur at run-time (i.e., execution time). These errors halt the execution of the program and the execution is terminated. The program stops execution from the statement, which raised the error. The run-time errors are called exceptions. These exceptions are not handled by the Java runtime environment. These must be handled explicitly by the program to have the smooth execution of the program till the last statement. Some situations that may raise exceptions are,

(1) Attempt to open a non-existing file.

(2) The class file to be loaded is missing or is in the corrupt format.

(3) An operand exceeds its range. For example, a divisor in a division cannot be zero and the index of an array element cannot exceed the size of that array.

(4) The other end of the network connection does not exist.

(5) The network connection is disconnected due to some reason.

All these abnormal conditions must be handled properly otherwise program terminates abnormally. For this, Java comes with exception handling mechanism.

Example : Exception handling can be explained by considering an example program that contains a 'divide-by-zero' expression.

```
class ExceptionHandling
{
    public static void main(String args[ ])
    {
        int p, q;
        try
        {
            q=0;
            p=17/q;
            System.out.println("This line will never be printed");
        }
        catch(ArithmeticException ae)
        {
            System.out.println("Division-by-zero");
        }
        System.out.println("After executing catch statement");
    }
}
```

Output

```
Division-by-zero
After executing catch statement
```

The above program contains an exception within the try block. Because, the value of 'p' is computed by dividing the number 17 by 0, which is not possible. This exception will be thrown from try block. The catch block following this try block will catch this exception and will print the message. As the control will never go from catch block back to its associated try block, the line. "This line will never be printed" will never be printed on the output screen.

2.2 TYPES OF ERRORS

2.2.1 Compile Time Errors

Compile-time errors are the syntax errors that occur at the compilation of the program. These errors are detected and displayed on the screen by the compiler. The compile-time errors are called bugs. The process of finding bugs is called debugging. Debugging is easier as the Java compiler list the errors with their line numbers, class name, etc. When compile-time errors occur, the programmer must correct them and then recompile program to get the class file. This file would not be created if there are bugs in the program.

Most of bugs (i.e., compile-time errors) are due to typing mistakes. The various compile-time errors are,

(1) Missing parenthesis and semicolon.

(2) Missing opening or closing quotations.

(3) Use of variables without declarations.

(4) Misspelling of identifiers and keywords.

(5) Missing opening or closing braces.

(6) Assigning incompatible types.

2.2.2 Run Time Errors

The run-time errors are the errors that occur at the run-time of the program and cause the abnormal termination of the program. The run-time errors are called exceptions. There are three types of runtime errors,

(1) **Input Errors :** Input errors occur if the user provides unexpected inputs to the program. For example, if the program excepts an integer and the user provides it the string value. These errors can be prevented from occurring by prompting the user to enter the correct type of values.

(2) **System Errors :** System errors or hardware errors occur rarely. These errors occur due to unreliable system software or hardware malfunctions. These errors are beyond a programmer's control. He has a little control once such errors occurs.

(3) **Logical Errors :** Logical errors occur if the program is logically incorrect. These errors either generate incorrect results or terminate program abnormally. For example, a program for adding two numbers requires an addition operator (+), if the program supplies substraction operator (–) then this generates incorrect results. To debug the logical errors the program must be scanned to check the logical statements

2.3 BENEFITS OF EXCEPTION HANDLING

The benefits provided by exception handling mechanism are listed below,

(1) The try-catch clauses used in exception handling helps in separating the working or functional code from the error handling code.

(2) The exception handling mechanism provides a clear path for the error to propagate. That is, when a called method cannot manage a particular situation, it throws an exception and asks the calling method to deal with that situation.

(3) The exceptional handling mechanism implements powerful coding as it enrolls the compiler to make sure that the 'exceptional' situations are prevented and described.

The main advantages of the exception-handling mechanism in object-oriented programming over the traditional error-handling mechanisms are as follows,

(1) **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. The 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.

(2) **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 subclasses.

(3) **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.

2.4 TERMINATION OR PRESUMPTIVE MODELS

(1) ***Termination Model :*** In this model of exception handling, the programmer will have to explicitly invoke the same method in which the error has occurred, and was transferred to the catch block, so that the error can be handled. This explicit invocation is required, because few programming languages that use this kind of model do not allow the control to return back to the point, where the error has occurred.

(2) **Presumptive Model :** In this model of exception handling, the programmer will not have to explicity invoke the same method in which the error had occurred and was transferred to the catch block, so that the error can be handled. Because, the programming languages that use this kind of model will allow the control to return back to the point, where the error had occurred.

2.5 EXCEPTION HIERARCHY

The exception hierarchy consists of all the possible exceptions that can occur in a Java program. In this hierarchy, 'Object' is the root class and has a subclass called 'Throwable'. The 'Throwable' class in turn has two subclasses one is the 'Exception' class and the other is the 'Error' class. The java.lang package defines both the standard exception and error classes.

2.5.1 Exception Classes

The exception classes are two types. They are as follows,

(1) **Checked Exceptions :** Checked exceptions are the exceptions thrown by a method, if it encounters a situation which is not handled by itself. All classes that are derived from 'Exception' class, but not 'RuntimeException' class are checked exceptions. Whenever a method is declared or called it is checked by compiler to determine whether the method throws checked exceptions or not. The methods that throws checked exceptions must handle them by providing a *throws* clause containing the list of checked exceptions that it can throw in the method declaration or it must handle them by providing a try-catch block. If these exceptions are not handled, the compiler issues error message indicating that the exception must be caught or declared.

The list of Java's checked exceptions defined in java.lang package are given below,

(i) **ClassNotFoundException :** This exception is thrown, if the requested class does not exist or if the class name is invalid.

(ii) **CloneNotSupportedException :** This exception is thrown, if there is an attempt to clone (or create or copy of) an object that does not implement the 'Clonable' interface.

(iii) **IllegalAccessException :** This exception is thrown, if the requested class cannot be accessed.

(iv) **InstantiationException :** This exception is thrown, if there is an attempt to instantiate (or create) an object of an abstract class or interface.

(v) **InterruptedException :** This exception is thrown, if one thread interrupts the another thread.

(vi) **NoSuchFieldException :** This exception is thrown, if there is an attempt to access a field that does not exist.

(vii) **NoSuchMethodException :** This exception is thrown, if there is an attempt to a method call that does not exist.

(2) **Run-time Exceptions (Unchecked Exceptions) :** The run-time conditions that can generally occur in any Java method are represented as runtime exceptions. These exceptions are also called 'unchecked exceptions'. A java method does not require to declare that it will throw any of the run-time exceptions.

The standard run-time exception classes that are defined in java.lang package are,

(i) **ArithmeticException :** This exception will be thrown, when an exceptional arithmetic condition has occurred.

Example : Integer division by zero.

(ii) **ArrayStoreException :** This exception will be thrown, when trying to store a value in the array that does not have a type as that of the array.

(iii) **ClassCastException :** This exception will be thrown, while trying to cast a reference to an object of inappropriate type.

(iv) **IllegalArgumentException :** This exception will be thrown, when an illegal argument is passed to a method.

- **IllegalThreadStateException :** This exception will be thrown, when an operation is performed on a thread that is illegal with respect to the current state of the thread.

  Example : Trying to use a dead thread.

- **NumberFormatException :** This exception will be thrown, when the numeric information in a string cannot be parsed.

(v) **IllegalStateException :** This exception will be thrown, when a method is invoked while the run-time environment is not in an appropriate state to perform the operation that has been requested.

(vi) **IllegalMonitorStateException :** This exception will be thrown, if any of the wait( ), notifyAll( ) or notify( ) methods of the object are called from a thread, that does not possess the monitor of that object.

(vii) **IndexOutOfBoundsException :** There are two subclasses of this exception. They are,

- **ArrayIndexOutOfBoundsException :** This exception will be thrown, when an array object detects an out of range index. This condition occurs when the index is greater than or equal to the size of the array or less than zero.
- **StringIndexOutOfBoundsException :** This exception will be thrown, when an out of range index is detected by a 'String' object or a 'StringBuffer' object.

(viii) **NegativeArraySizeException :** This exception will be thrown, when an array of negative size is created.

(ix) **NullPointerException :** This exception will be thrown, when an object is accessed using a null object reference.

(x) **SecurityException :** This exception will be thrown, when an operation is performed that does not conform to the security policy that is being implemented by the object of 'Security Manager'.

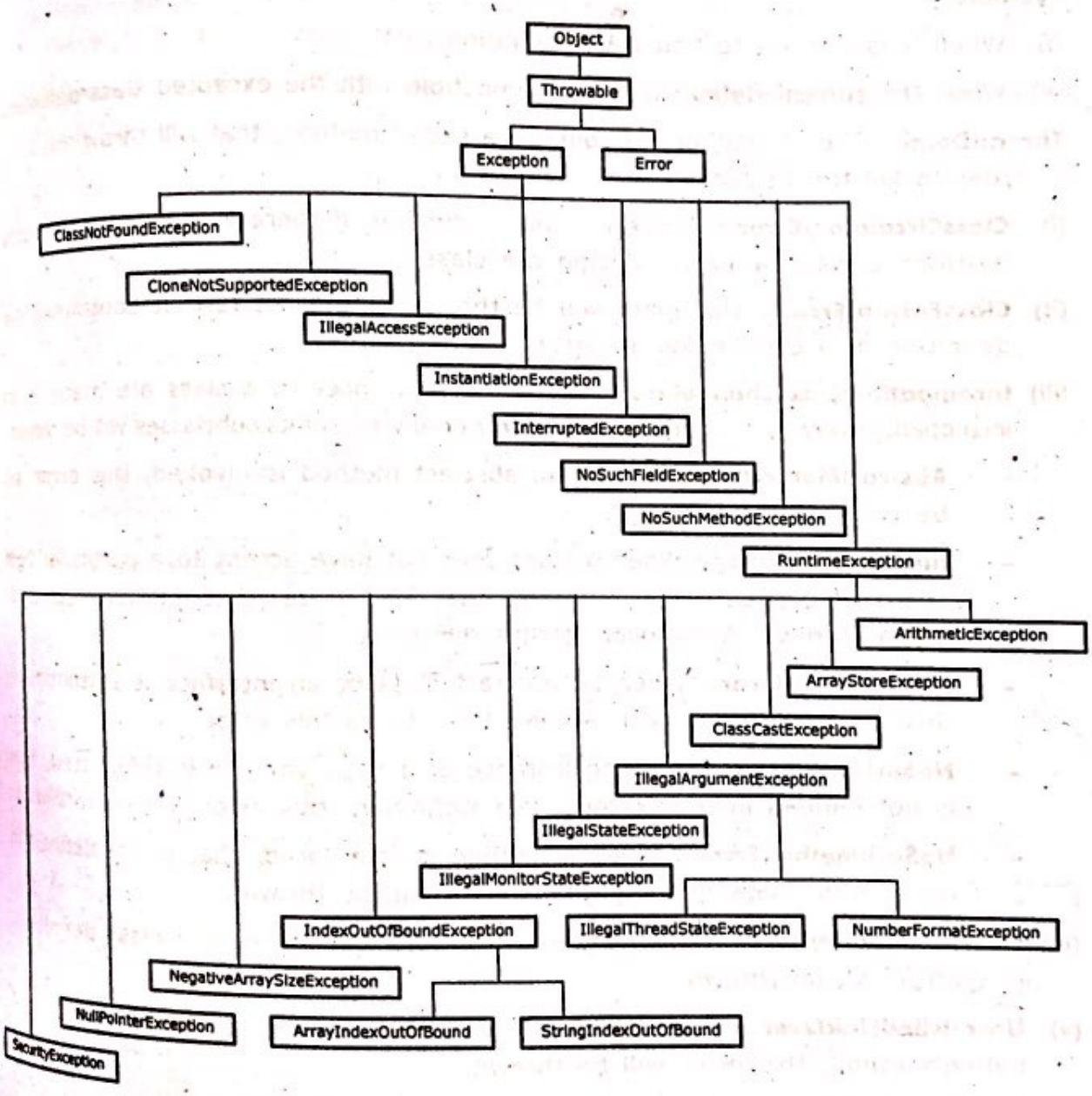

Fig. 2.5.1 Exception Class Hierarchy

2.5.2 ERROR CLASSES

Error Class consists of several subclasses. Whenever an unpredictable error is occurred, any of its appropriate subclasses will be thrown. As the errors have unpredictable nature, it is not required to declare them in the 'throws' clauses when the methods throw objects that are instances of 'Error' class or any one of its subclasses. The subclasses of Error Class are as follows,

(1) **LinkageError :** Whenever there is a problem in resolving the class reference, any one of appropriate subclass of this error will be thrown.

Example

(i) When it is difficult to find class definition (or)

(ii) When the current definition is not compatible with the excepted class definition.

(2) **ThreadDeath :** The 'thread' object contains a 'stop( )' method, that will throw this error in order to kill the thread.

(i) **ClassCirculatoryError :** This error will be thrown, if there is a circular reference between classes while initializing the class.

(ii) **ClassFormatError :** This error will be thrown, if the file format containing the definition of a class holds an error.

(iii) **IncompatibleClassChangeError :** When the references to a class are made in an incompatible way by another class, this error or any one of its subclasses will be thrown.

- **AbstractMethodError :** When an abstract method is invoked, this error will be thrown.
- **IllegalAccessError :** When a class does not have access to a particular field or a method, but it tries to access that filed or call that method, this error will be thrown. A compiler catches this error.
- **InstantiationError :** When an abstract class or an interface is instantiated, this error will be thrown. A compiler catches this error.
- **NoSuchFieldError :** When an instance or a class variable is referenced, that is not defined in the current class definition, this error will be thrown.
- **NoSuchMethodError :** When a method is referenced, that is not defined in the current class definition, this error will be thrown.

(iv) **NoClassDefFoundError :** When there is a problem in finding the class definition, this error will be thrown.

(v) **UnsatisfiedLinkError :** When there is a problem in finding the implementation of native method, this error will be thrown.

(vi) **VerifyError :** When the byte code verifier finds out, that a well formed class file holds a kind of security problem or internal inconsistency, this error will be thrown.

2.6 CONCEPTS OF EXCEPTION HANDLING

2.6.1 Usage of 'Try'

The statements in a program that may raise exceptions 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 run-time errors 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 executing block. A try block should have one (or more) catch blocks 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 blocks directly after the try block.

Example

```
class TryExample
{
    public static void main(String args[ ])
    {
        int a, b=c=2, d;
        try
        {
            d = b-c;
            a = 17/d;
        }
        catch(ArithmeticException e)
        {
            System.out.println("Division-by-zero exception");
        }
    }
}
```

Output

Division-by-zero exception

In the above program the try block contains a statement that divides the number 17 by 0. This will generate an exception, which will be caught by the catch block that follows the try block. The run-time error that could have occurred because of this exception, is thus handled and prevented.

2.6.2 Nested 'try' Statements

The try statements can be nested i.e., there can be one try statement inside the other try statement. Whenever the control goes to the try block its contents is pushed on the stack. If inner try block doesn't have its matching catch for an exception, the stack is not popped out and the catch block for next try statement are inspected for a match. This process goes on until the exception is caught or all of the nested try statements are finished. If no catch block handles the exception, then it is handled by the Java run time system (i.e., it terminates the program).

Example : The below program shows how the nested try statement works.

```
class NestedTry
{
   public static void main(String args[ ])
   {
      try
      {
         int x = args.length;
         System.out.println("x = " + x);
         int y = 2/x;                    //MaygenerateArithmeticException if
                                           x=0.
         System.out.println("y = " + y);
         try                             //Nested try block
         {
           if(x == 1)
             x = x/(x - x);
           if(x == 2)
           {
              int C[ ] = {1};
              C[2] = 32;
                                   //out-of-bounds exception
           }
         }
```

```
            catch(ArrayIndexOutOfBoundsException e)
            {
                System.out.println("Out of bounds exception" + e);
            }
            catch(ArithmeticException e)
            {
                System.out.println("Divide-by-zero" + e);
            }
        }
        catch(Exception e)
        {
            System.out.println("Caught an error" + e);
        }
    }
}
```

Output

```
x = 0
Caught an error java.lang.ArithmeticException : Divide by zero
```

In the above program, the statement y = 2/x in the outer try block throws an exception if no command line argument is passed. If only one argument is passed then a divide-by-zero exception is generated from within the nested try block. Since it is not handled by the inner block, so it will be passed on to the outer try block, where it is handled. If two command line arguments are passed then the inner try block generates an array out of bound exception, which is handled by the outer try block.

2.6.3 Usage of 'Catch'

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.

A catch block is specified by the keyword catch followed by a single argument within parenthess( ). The argument type in a catch clause is form the Throwable class or one of its subclasses. Java initializes the argument (parameter) to refer to the caught object.

The syntax for catch block is,

```
catch(Exception-type)
{
    // code to handle exception
}
```

The Exception-type here indicates the type of exception the catch block is going to handle.

Example : The program below illustrates the try and catch blocks,

```
public class JustTryCatch
{
    public static void main(String args[ ])
    {
        int i=7;
        int j=0;
        try
        {
            System.out.println(i/j);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arthmetic Exception Divide by zero error caught
            :" + e);
        }
        //rest of program
    }
}
```

Output

Arithmetic Exception Divide-by-zero error caught : java.lang.ArithmeticException : Divide-by-zero

Here, in the program, i and j are two integer variables with initial values of i=7 and j=0. Now if we try to divide i by j then obviously the result is an error. The statement which could arise exception is,

```
System.out.println(i/j);
```

Therefore it is enclosed in try block.

When the try block is executed an ArithmeticException is thrown to catch block. The 'ArithmeticException' says that the catch block is going to handle only arithmetic exceptions. The catch block handles the thrown exception and prints, "ArithmeticException Divide-by-zero error caught", there by returns to safe state and lets the rest of the program run.

If no exception is occurred in try block, catch block is not executed and rest of the program gets executed.

The Fig. 2.6.1 depicts the case if an error occurrs in try block,

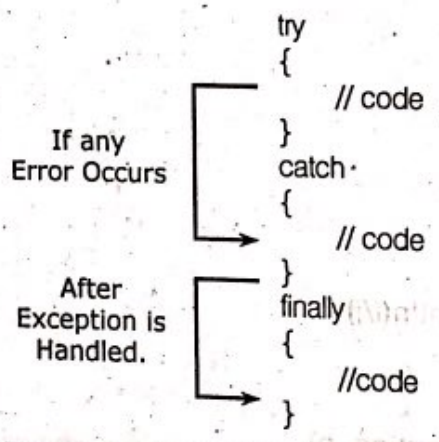

Fig. 2.6.1 Error in try Block

The Fig. 2.6.2 depicts the case if no error occurrs in try block,

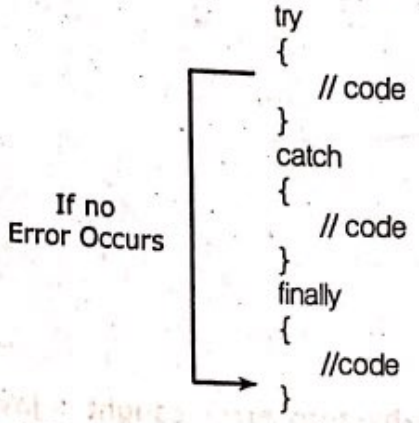

Fig. 2.6.2 No Error in try Block

2.6.4 Mulitple 'catch' Statements

There may be cases where in the statements in a try block can throw more than one exception which should be handled in the catch block. But a single catch block can't handle more than one exception. To handle these situations, we need to specify more than one catch block, each handling a different type of exception. So a single try block can have any number of catch blocks following it one after the other. When the try block throws an exception, then each catch block is inspected in order and the one whose type matches with that of thrown exception is executed and the rest of the catch blocks are skipped and the control goes to the next statement after the try/catch block.

It is possible that no handler will match with a particular thrown object. This causes to search for a exception handler in the next enclosing try block. As this process continues, it will be determined that there is handler matching particular thrown object in the stack. In such a case the non-GUI base application terminates applets and GUI-based applications return their regular event processing.

It is also possible that the exception handlers will provide a match to the type of exception. For this, there are several reasons,

(1) There can be an exception handler that catches any exception (i.e., Exception e).

(2) Due to inheritance, it is possible that the exception in subclass is handled by a handler specified in the subclass or by the handler specified in the superclasses of that subclass. The first exception handler which match the type of exception is executed.

Example : The program below shows the use of multiple catch blocks,

```
class MultiCatch
{
    public static void main(String args[ ])
    {
        int x=100, y=0, z;
        int a[ ]={10, 20, 30, 40, 50};
        try
        {
            System.out.println("Division :" + x/y);
            System.out.println(a[5]);
        }
```

```
            catch(ArithmeticException e)
            {
                System.out.println("Divide by zero error :" + e);
            }
            catch(ArrayIndexOutOfBoundsException e)
            {
                e.printStackTrace( );
            }
        }
    }
```

Output

Divide-by-zero error : java.lang.ArithmeticException : Divide-by-zero

In the above program, the try block has two statements, where each statement throws one type of exception. Where the first statement 'x/y' throws ArithmeticException that is handled by the first catch block in the program. When the exception occurs the control leaves the try block and goes to the matching catch block. So the second statement in try block is never executed. To make it execute, make 'y' some integer value, so that first statement is successfully executed and the control goes to the second statement, which raises an exception called ArrayIndexOutOfBounds Exception because it is trying to access an array index that does not exist at all. This exception is handled by the second catch block in the program.

2.6.5 Usage of 'throw'

'throw' is a Java keyword used in exception handling. Generally a try block checks for arrival of error and when an error occurs it throws the error and it is caught by the catch statement and then appropriate action will take place. Only the expressions thrown by the java run-time system are being caught, but throw statement allows a program to throw an expression explicitly.

The syntax of throw statement is,

```
throw ThrowableInstance;
```

The *Throwableinstance* should be an object of Throwable class or its subclass. The int, char, string etc., type of objects cannot be thrown. The Throwable object can be created in two ways,

(1) As a parameter in to catch clause.

Example

```
catch(NullPointerException e)
{
        ---
        ---
        ---
        throw e;
}
```

(2) Using new operator.

Example

```
throw new NullPointerException;
```

When an error occurs the execution gets stopped at throw statement. The very next try block is checked to see if it has catch block, if not checking goes to the next try block. If no catch is matched the situation is handled by default exception handler. The default exception handler terminates the programs and displays information in stack trace.

Example

```
class ThrowUsage
{
    static int x=5, y=0, z;
    static int div( )
    {
        try
        {
            if(y == 0)                          //throws an exception
                throw new ArithmeticException("Division-by-zero");
            else
                return x/y;
        }
```

```
        catch(ArithmeticException ae)
        {
            System.out.println("ArithmeticException caught at div( )");
            throw ae;                                   // rethrows an exception
        }
    }
    public static void main(String args[ ])
    {
        try
        {
            z = div( );
            System.out.println("z = " + z);
        }
        catch(ArithmeticException ae)
        {
            System.out.println("Exception recaught at main( ) :" + ae);
        }
    }
}
```

Output

```
ArithmeticException caught at div( )
Exception Recaught at main( ) : java.lang.ArithmeticException : Division-by-zero
```

2.6.6 Usage of 'throws'

Generally any method may cause exceptions. But some methods cause exceptions, which could not be handled by itself. If this is the case with a method, then the method should atleast inform the method callers that it may throw exceptions of so and so types. A method can specify what type of exceptions it is going to throw by using the keyword *throws*.

The syntax of throws keyword is,

```
type method-name(parameter-list) throws Exceptiontype1, Exceptiontype2,.....
{
    ---
    ---
}
```

Example

```
import java.io.*;
class ThrowsUsage1
{
    static void test( ) throws IOException
    {
        System.out.println("Inside test( )\n throwing IO exception");
        throw new IOException("IO Exception");
    }
    public static void main(String args[ ])
    {
        try
        {
            test( );
        }
        catch(IOException e)
        {
            System.out.println("Caught :" + e);
        }
    }
}
```

Output

```
Inside test( )
throwing IO exception
Caught : java.io.IOException : IOException
```

2.6.7 'throw' Vs 'throws' Keywords

'throw' and 'throws' are keywords of Java used with exception handling.

Throw : Sometimes we can pass information to the user as exception messages. Suppose, in a bank, if an account holder tries to draw more amount than what his balance is, we can inform him of the same through a print statement on a monitor. The same message can also be passed as an exception message using any one of the predefined exception classes. The exception messages are displayed with throw keyword. With throw keyword, we can explicitly throw any exception(generally exceptions are thrown by the system). In the below statement, we have created an anonymous object of Exception and passed the message of parameter to the constructor.

Example : throw new Exception("Insufficient funds please");

Throws : 'throws' keyword can be used in two ways,

(1) **Claiming the Exception :** Sometimes the owner of a method wants to warn the users against the possible dangers (or exceptions) that may occur if they use that method. See how an author(the language designer) warns the possible exception that may occurs when a file is opened,

```
public FileInputStream(String filename) throws FileNotFoundException
```

He warns the possibility of not existing the file that the user wants to open. The author warns by placing the exception in the method (here, constructor) header itself with throws clause. Any one who wants to use the constructor has to handle the exception (using try-catch).

(2) **Alternative way for try-catch :** We can use throws as an alternative for a try-catch block, but this style is not robust.

Example : public static void main(String args[ ]) throws IOException

2.6.8 Usage of 'finally'

Whenever an exception occurs, the exception affects the flow of execution of the program. Sometimes some blocks may be bypassed by exception handling.

Consider a program, which has allocated memory to a particular variable with an intention to use it. The deallocation of memory for that varaible is at the end of the code. Now, suppose that due to the uncaught exception, the program execution has bypassed the deallocation part of code. This is undesirable and therefore inorder to make sure that some statements must be surely executed irrespective of exception, finally block has been introduced.

Every try block should be associated with atleast one catch or finally block. However, it is optional to have a finally block within a program. If a finally block is included in a program, then it will be executed after the try/catch block, but before the code following this try/catch block. The finally block will be executed irrespective of the occurrence of exceptions.

A finally block helps in closing opened files, deallocating the allocated memories etc.

The syntax of the finally block is,

```
try
{
    ---------
    ---------
}
catch(...)
{
    ---------
    ---------
}
catch(...)
{
    ---------
    ---------
}
finally
{
    ---------
    ---------
}
```

Example

```
class Exfinally
{
    static void proOne( )
    {
        try
        {
```

```
            System.out.println("Process One");
            throw new RuntimeException("Demo");
        }
        finally
        {
            System.out.println("Process One's finally");
        }
    }
    static void proTwo( )
    {
        try
        {
            System.out.println("Process Two");
            return;
        }
        finally
        {
            System.out.println("Process Two's finally");
        }
    }
    static void proThree( )
    {
        try
        {
            System.out.println("Process Three");
        }
        finally
        {
            System.out.println("Process Three's finally");
        }
    }
```

```
    public static void main(String args[ ])
    {
        try
        {
            proOne( );
        }
        catch(Exception e)
        {
            System.out.println("Caught the exception");
        }
        proTwo( );
        proThree( );
    }
}
```

Output

```
Process One
Process One's finally
Caught the exception
Process Two
Process Two's finally
Process Three
Process Three's finally
```

2.7 BUILT IN EXCEPTIONS

Java defines several exception classes inside the java.lang package. The most general of these exceptions are subclasses of the standard type 'RuntimeException'. Many exceptions derived from RuntimeException are available automatically because java.lang package is implicitly imported into all Java programs. Also it is not necessary for them to be included in any method's throw list. These are known as unchecked exceptions, because the compiler doesn't check to see if a method handles or throws these exceptions.

There are exceptions defined by java.lang, which must be included in a method's throws list if that method can generate one of these exceptions and doesn't handle itself. These are known as checked exceptions. The following are the various, checked and unchecked exceptions defined in java.lang package are given in Tables 2.7.1 and 2.7.2.

Table 2.7.1 Java's Unchecked RuntimeException Subclasses ... java.lang

S.No.	Exception	Meaning
(1)	ArithmeticException	Arithmetic error, such as divide-by-zero.
(2)	ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
(3)	ArrayStoreException	Assignment to an array element of an incompatible type.
(4)	ClassCastException	Invalid cast.
(5)	EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
(6)	IllegalArgumentException	Illegal argument used to invoke a method.
(7)	IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
(8)	IllegalStateException	Environment or application is in incorrect state.
(9)	IllegalThreadStateException	Requested operation not compatible with current thread state.
(10)	IndexOutOfBoundsException	Some type of index is out-of-bounds.
(11)	NegativeArraySizeException	Array created with a negative size.
(12)	NullPointerException	Invalid use of a null reference.
(13)	NumberFormatException	Invalid conversion of a string to a numeric format.
(14)	SecurityException	Attempt to violate security.
(15)	StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
(16)	TypeNotPresentException	Type not found.
(17)	UnsupportedOperationException	An unsupported operation was encountered.

Table 2.7.2 Java's Checked Exceptions Defined in java.lang

S.No.	Exception	Meaning
(1)	ClassNotFoundException	Class not found.
(2)	CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
(3)	IllegalAccessException	Access to a class is denied.
(4)	InstantiationException	Attempt to create an object of an abstract class or interface.
(5)	InterruptedException	One thread has been interrupted by another thread.
(6)	NoSuchFieldException	A requested field does not exist.
(7)	NoSuchMethodException	A requested method does not exist.

CREATING OWN EXCEPTION SUBCLASSES

Of course Java's built-in exceptions handle most common errors, but still we may require to create our own exception types for handling situations specific to our applications. It is very simple to do this, we must define a subclass of exception. Subclasses don't require to implement anything. It is their existence in the type system, which allows us to use them as exceptions. Exception class won't define any methods of its own. It inherit the methods provided by 'Throwable'. All exceptions (including which we create) have the methods defined by 'Throwable' available to them.

The Table 2.8.1 shows some of the important methods defined by Throwable.

Table 2.8.1 The Methods Defined by Throwable

S.No.	Method	Description
(1)	Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
(2)	Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
(3)	String getLocalizedMessage( )	Returns a localized description of the exception.
(4)	String getMessage( )	Returns a description of the exception.
(5)	StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives our program access to information about each element in the trace, such as its method name.
(6)	Throwable initCause (ThrowablecauseExc)	Associates causeExc with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
(7)	void printStackTrace( )	Displays the stack trace.
(8)	void printStackTrace (PrintStream stream)	Sends the stack trace to the specified stream.
(9)	void printStackTrace (PrintWriter stream)	Sends the stack trace to the specified stream.
(10)	void setStackTrace (StackTraceElementelements[ ])	Sets the stack trace to the elements passed in elements. This method is for specialized applications, not normal use.
(11)	String toString( )	Return a String object containing a description of the exception. This method is called by println( ) when outputting a Throwable object.

The example here declares an exception MyOwnException, the subclass of Exception class and uses it to report an error if one or both the arguments to add( ) method is less then zero. This class also overrides the toString( ) method defined by Throwable class to return the description of the exception.

Example

```
class MyOwnException extends Exception
{
    public MyOwnException( ) { }
    public String toString( )
    {
        return "MyOwnException type error";
    }
}
class CreateException
{
    static void add(int a, int b) throws MyOwnException
    {
        if((a<0)||(b<0))
            throw new MyOwnException( );
        System.out.println("Sum =" + (a+b));
    }
    public static void main(String args[ ])
    {
        try
        {
            add(50, 100);
            add(30, -20);
        }
        catch(MyOwnException moe)
        {
            System.out.println("Caught :" + moe);
        }
    }
}
```

```
class MyException
{
    static void comp(int a) throws SuperException
    {
        if(a == 100)
        throw new SuperException(a);
        System.out.println("completed");
    }
    public static void main(String args[ ])
    {
        try
        {
            comp(100);
        }
        catch(SuperException e)
        {
            System.out.println("Caught Super Exception");
        }
    }
}
```

Output

Caught Super Exception

2.9 INTRODUCTION TO 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 as 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.

2.9.1 Multithreading in Single Processor and Multiprocessor Systems

Multithreading means running more than one thread concurrently. In a single processor system, multithreading is achieved by time slicing that is the processor switches between different threads from time-to-time. Since a processor can process one task at a time, these threads will not run concurrently. The processor switches between threads so fast that it gives the illusion of simultaneity to an enduser.

On a multiprocessor system, multithreading is achieved by multiprocessing. In these systems, multiple processors processes different threads concurrently. Wherein each thread is executed independently with another thread running on some other processor.

In multiprocessor systems, the multithreading is achieved automatically. Here, operating system uses all the processors in the compiler to run any number of threads. Multithread applications takes the full advantage of any number of processors within the system and run multiple threads simultaneously and finish the task in less amount of time.

Multiprocessor systems runs the applications faster than single processor systems. Therefore they are used in projects to increase the performance. Since the singlethread applications use only one processor at any time, they may effect the system performance as most of the time of a processor is wasted in switching the application from one processor to another. An application must be multithreaded to achieve the maximum benefit of mutithreading operating systems and multiprocessor machines.

The operating systems windows NT, windows 95 and some versions of UNIX are multithread operating systems and can run either on a single processor system or a multiprocessor system.

2.9.2 Advantages of Multithreading

The advantages of multithreading are as follows,

(1) We can reduce the idle time of the microprocessor.

(2) Performance of processes is improved.

(3) During the synchronous lock acquisition call, where the single-threaded application no longer can use the processor, the operating system switches to a thread from another application on the computer that wants to use the processor.

(4) Developers also can prioritize threads to increase responsiveness for time-critical threads. Normally, high-priority threads execute for short periods of time and occur only on special occassion throughout the program.

(5) Reduces interference between execution and user interface.

Multithreaded applications can take full advantage of multiple processors to gain better performance through simultaneous execution of tasks.

2.3 Multitasking

Multithreading is a concept of dividing a program into two or more subprograms that run concurrently. Each subprogram is called a thread. Each thread executes separately ependent of other threads. Thus multithreading is a specialized form of multitasking.

There are two types of multitasking,

Process-based Multitasking : A process is a program, that is in execution by the processor. Process-based multitasking allows two or more programs to run concurrently by the processor. For example, using an text editor while running the Java compiler. In process-based multitasking, a program is the smallest dispatchable unit of code.

Thread-based Multitasking : Thread-based multitasking allows two or more tasks of a single program to run simultaneously by the processor. For example, the text editor formatting the text at the same time printing the text. Here both tasks are performed by two separate threads. In thread-based multitasking, a thread is the smallest dispatchable unit of code.

The Table 2.9.1 shows the comparison between process-based and thread-based ultitasking.

Table 2.9.1 Differences between Process-based and Thread-based Multitasking

S.No.	Process-based Multitasking	Thread-based Multitasking
(1)	More overhead is required in multitasking processes.	Less overhead is required in multitasking threads.
(2)	Each process has its own address space. Therefore, they are heavy weight.	Threads share the same address space and the same process. Therefore, they are light weight.
(3)	Interprocess communication is expensive and limited	Interthread communication is expensive.
(4)	Context switching between processes is costly.	Context switching between threads is cheaper.
(5)	It is not controlled by Java.	It is controlled by Java.

2.9.4 Differences Between Multithreading and Multitasking

Table 2.92 Differences between Multithreading and Multitasking

S.No.	Multithreading	Multitasking
(1)	More than one thread runs simultaneously.	More than one process runs simultaneously.
(2)	It's part of a program.	It's part of a program.
(3)	It's a light-weight process.	It's a heavy weight process.
(4)	Threads are divided into subthreads.	Process is divided into threads.
(5)	Within the process threads are communicated.	There is no communication between processors directly.

2.10 AN OVERVIEW OF THREADS

A *thread* is a single sequential flow of control within a program. It differs from a process. 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 by itself is not a complete program and cannot run on its own. 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.

2.10.1 Thread Life Cycle

A thread can be defined as a sequence of instructions which can run independently.

A thread is always in one of five states,

(1) newborn.

(2) runnable.

(3) running.

(4) dead and

(5) blocked.

Fig. 2.10.1 shows the life cycle of a thread,

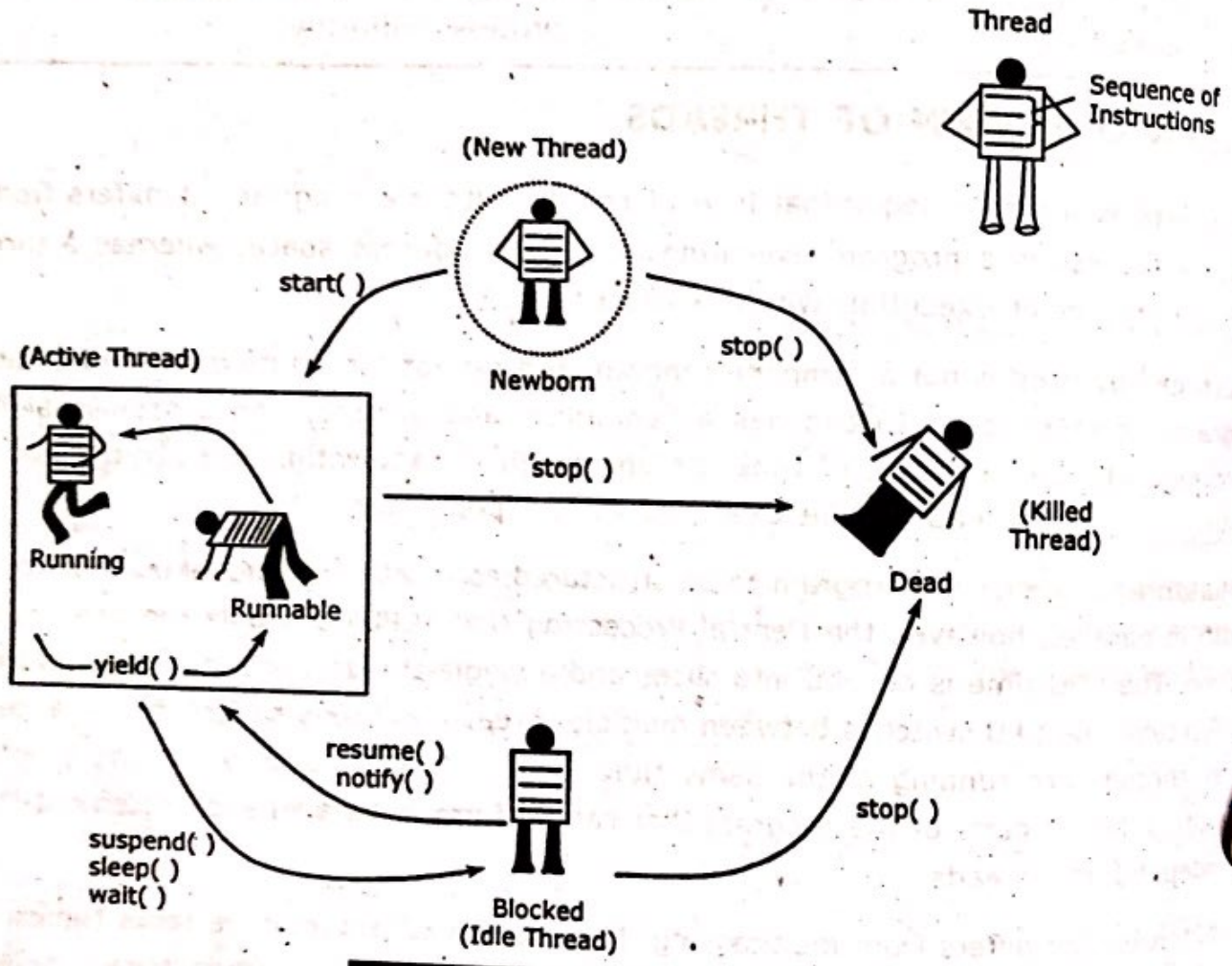

Fig. 2.10.1 Life Cycle of a Thread

(1) **The Newborn State :** When a thread is called, it is in 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.

(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 the CPU) and moves on 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.

(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,

(i) It dies (that is, the run( ) method exits).

(ii) It gets blocked to the input/output for some reason.

(iii) It calls sleep( ).

(iv) It calls wait( ).

(v) It calls yield( ).

(vi) It is preempted by a thread of higher priority.

(vii) 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,

(i) If the thread with the higher priority comes out of the sleeping state.

(ii) The I/O completes for a thread with a higher priority waiting for that I/O.

(iii) 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.

(4) **The Dead State :** A thread goes into the dead state in two ways,

(i) If its run( ) method exits.

The run( ) method exits when it finished execution naturally or throws an uncaught exception.

(ii) A stop( ) method is invoked.

The stop( ) method kills the thread.

A thread in the dead state cannot be executed further.

(5) The Blocked State : A thread can enter the blocked state when one of the following five conditions occurs,

(i) When sleep( ) is called.

(ii) When suspend( ) is called.

(iii) When wait( ) is called.

(iv) The thread calls an operation, (For example, during input/output, a thread will not return until the I/O operation completes).

(v) 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 operations, it will enter into the blocked state by giving the access of 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 into the blocked state.

(1) If a thread has been put to sleep( ), the specified timeout period must expire.

(2) If a thread has called wait( ), then some other thread using the resource for which the first thread is waiting must call notify( ) or nofityAll( ).

(3) If a thread is waiting for the completion of an input or output operation, then the operation must finish.

Example

```
class SharedBankAccount
{
    private double bal = 5000;
    public void getBal( )
    {
        return bal;
    }
    public void withDraw(double amt)
    {
        bal = bal-amt;
    }
}
```

```
public class AccessBankAccount implements Runnable
{
    private SharedBankAccount ac = new SharedBankAccount( );
    public static void main(String args[ ])
    {
        AccessBankAccount obj = new AccessBankAccount( );
        Thread first = new Thread(obj);
        Thread second = new Thread(obj);
        first.setName("First thread");
        second.setName("Second thread");
        first.start( );
        second.start( );
    }
    public void run( )
    {
        for(int i=0; i<5; i++)
        {
            withDrawing(1000);
        }
    }
    private void withDrawing(double amt)
    {
        string name = Thread.currentThread( ).getName( );
        if(ac.getbal( ) >= amt)
        {
            System.out.println(name + "is withdrawing");
            ac.withDraw(amt);
            System.out.println(name + "completed withdrawing");
            try
            {
                System.out.println(name + "is sleeping");
                Thread.s  /1000);
            }
```

```
                catch(InterruptedExecution e)
                {
                    e.printStackTrace( );
                }
                System.out.println(name + "woke up");
            }
            else if(ac.getBal( )<=0)
            {
                System.out.println("Attempt to overdraw amount");
            }
            else if(ac.getBal( ) < amt)
            {
                System.out.println("There is no sufficient amount for" + name);
            }
        }
    }
```

Output

```
First thread is withdrawing
First thread completed withdrawing
First thread is sleeping
Second thread is withdrawing
Second thread completed withdrawing
Second thread is sleeping
First thread woke up
First thread is withdrawing
First thread completed withdrawing
First thread is sleeping
Second thread woke up
Second thread is withdrawing
Second thread completed withdrawing
Second thread is sleeping
First thread woke up
```

First thread is withdrawing
First thread is sleeping
Second thread woke up
Attempt to overdraw amount

2.11 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 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.

2.11.1 By Extending Thread Class

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,

```
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("first");
        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.

2.11.2 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 subclass of classes other than Thread, using Runnable interface is the best way of creating threads.

The general form of code is 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.

2.11.3 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 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);
```

Example

```
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( ));
    }
}
```

Output

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

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.

2.12 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 thread) 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 reasons,

(1) The thread has not yet started.

(2) Stopped by another thread.

(3) Completion of the thread itself.

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

```
try
{
    t1.join( );
    t2.join( );
    t3.join( );
}
catch(InterruptedException e)
{
}
```

Here t1, t2, t3, 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 in the following form,

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

S.No.	Name of the Method	Function
(1)	getName( )	Returns the name of the thread
(2)	setPriority( )	To set the priority of the thread
(3)	getPriority( )	Returns the thread priority
(4)	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
(5)	join( )	Wait for a thread to terminate
(6)	run( )	Entry point for the thread
(7)	start( )	Start a thread by calling its run method
(8)	stop( )	Terminate the thread abruptly

In addition to the methods described in Table 2.14.1, there are some additional ethods. They include the sleep( ), suspend( ), resume( ), wait( ), notify( ), notifyAll( ) and ld( ) methods.

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(longmillisec) throws InterruptedException;

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

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.

(5) **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) **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.

2.14 THREAD PRIORITIES

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,

(1) Thread.MIN_PRIORITY 1 (minimum priority)

(2) Thread.MAX_PRIORITY 10 (maximum priority)

(3) Thread.NORM_PRIORITY 5 (default)

2.14.1 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.

Example : The program shows how to set and get the priorities of a thread.

```
public class Demo extends Thread
{
    public void run( )
    {
        for(int i=1; i<10; i++)
        {
            String str = Thread.currentThread( ).getName( );      //this.getName( );
            System.out.println(str + " : " + i);
        }
    }
```

```
    public static void main(String args[ ])
    {
        Demo obj1 = new Demo( );
        Demo obj2 = new Demo( );
        Demo obj3 = new Demo( );
        obj1.setName("First Thread");
        obj2.setName("Second Thread");
        obj3.setName("Third Thread");
        obj1.setPriority(NORM_PRIORITY);
        obj2.setPriority(MAX_PRIORITY);
        obj3.setPriority(MIN_PRIORITY);
        obj1.start( );
        obj2.start( );
        obj3.start( );
        System.out.println("Priority of 1st thread :"+obj1.getPriority( ));
        System.out.println("Priority of 2nd thread :"+obj2.getPriority( ));
    }
}
```

Output

```
Priority of 1st thread : 5
Second Thread : 1
First Thread : 1
Third Thread : 1
Priority of 2nd thread : 10
Second Thread : 2
First Thread : 2
Second Thread : 3
First Thread : 3
Second Thread : 4
First Thread : 4
Third Thread : 2
```

```
Second Thread : 5
Second Thread : 6
First Thread : 5
First Thread : 6
Second Thread : 7
First Thread : 7
Third Thread : 3
Second Thread : 8
First Thread : 8
Second Thread : 9
First Thread : 9
Third Thread : 4
Third Thread : 5
Third Thread : 6
Third Thread : 7
Third Thread : 8
Third Thread : 9
```

In the above program different priorities are set to three different threads. Each thread executes the loop for 10 times. The thread with the highest priority that is the second thread in this program will be executed first. The last 3 statements gets the priority associated with each thread.

2.14.2 Time Slicing

Some Java platforms support a concept called time-slicing and some do not support. 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.

2.14.3 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 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,

(1) A higherpriority thread becomes runnable.

(2) Its yields( ) or run( ) method exits.

(3) Its time-slice has expired (only on systems that support time-slicing).

2.15 SYNCHRONIZING THREADS

Synchronization is a process in which an access to a shared resource is allowed to only one thread at a time. Synchronization is required in multithreaded programming, when two or more threads tries to access the shared resources at the same time. If two threads are allowed to access the shared resource at the same time then the data will be changed, hence producing the wrong output.

Thread synchronization is important in multithreaded programming in order to maintain the data consistency. In Java, synchronization is achieved using 'synchronized' keyword. A block or a method that should be shared among different threads is declared as synchronized. A thread that calls the synchronized method gets the access to the shared resource. If another thread tries to call the same method will block until the first thread relinquish or leave that method. Once the thread leaves method then other threads waiting for that synchronized method can make a call. And hence in this way the data consistency is maintained.

In Java, a method or a block of statements within a method can be synchronized as follows,

```
public synchronized double getBalance(int accid, double amount)
{                    // a method is synchronized
    balance = balance + amount;
                     // other code
    return balance;
}
```

Synchronization makes the program run slower, thereby performance will be lost, because only one thread can execute the method at a time. We must synchronize sensitive or critical data only. For this Java comes with another format,

```
public double getBalance(int accid, double amount)
                    // a block of statements is synchronized
{
    synchronized(this)
    {
        balance = balance + amount;
    }
                    // other code
    return balance;
}
```

In the above method, only few statements of the method are synchronized, but not the whole method itself.

Example : In the below program, 10 customers exists, who operate on one joint account and calls deposit(int amount) method simultaneously. Each customer deposits Rs. 10 for 1 lakh times, that is, finally there must be 1 crore of rupees. If we remove the synchronized keyword from deposit( ) method, we will see different outputs when executed multiple times.

```
class Account
{
                    // a class that updates the balance whenever customer deposits
    private int balance=0;
    public synchronized void deposit(int amount)
    {
        balance += amount;
    }
    public int getBalance( )
    {
                        // to get the balance of the account whenever it is needed
        return balance;
    }
}
```

```
class Customer extends Thread          // Customer is a thread class
{
                                       // each joint-account holder deposits ₹ 10 for 100000 times
    Account account;
    public Customer(Account account)
    {
        this.account = account;
    }
    public void run( )
    {
        for(int i=0; i<100000; i++)
            account.deposit(10);
    }
}
class Bank
{
                                       // a class that create customer object and starts functioning
    private final static int NUMCUSTOMERS=10;
    public static void main(String args[ ])
    {
        Accou      count = new Account( );
        Customer customer[ ] = new Customer[NUMCUSTOMERS];
        for(int i=0; i<NUMCUSTOMERS; i++)
        {
            customer[i] = new Customer(account);
            customer[i].start( );
            try
            {                          // main thread to wait for all ch
                                       complete their execution
```

```
                customer[i].join( );
            }
            catch(InterruptedException e)
            {
                e.printStackTrace( );
            }
        }
        System.out.println("Current balance is :"+account.getBalance( ));
    }
}
```

Output

```
10000000
```

Example : The below program shows the importance of synchronization in multithreaded programming.

```
class First
{
    synchronized void call(String str)
    {
        System.out.print("[" + str);
        try
        {
            Thread.sleep(10000);
        }
            catch(InterruptedException e)
        {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}
```

```
class Second implements Runnable
{
    String str;
    First f;
    Thread t;
    public Second(First fir, String S)
    {
        f = fir;
        str = S;
        t = new Thread(this);
        t.start( )
    }
    public void run( )
    {
        f.call(str);
    }
}
class SynchDemo
{
    public static void main(String args[ ])
    {
        First f = new First( );
        Second S1 = new Second(f, "Hello");
        Second S2 = new Second(f, "World");
        Second S3 = new Second(f, "Synchronized");
        try
        {
            S1.t.join( );
            S2.t.join( );
            S3.t.join( );
        }
```

```
        catch(InterruptedException e)
        {
            System.out.println("Interrupted");
        }
    }
}
```

Output

[Hello]

[Synchronized]

[World]

As shown in the above program by calling sleep( ), the call( ) method allows another thread to get executed. This will result in the mixed-up output of three message strings. To prevent this, method call( ) is declared as synchronized. When the sleep( ) is called by call( ), it makes other threads to wait until the call( ) resumes and returns to the calling method. This prevents other threads to access the shared resource at the same time.

2.16 INTER THREAD COMMUNICATION

In order to avoid polling, java includes an elegant inter-thread communication mechanism by using wait( ), notify( ) and notifyAll( ) methods. These methods are defined in Object class as final. Thus, they are available to all the classes. These three methods can be called only from within a synchronized method. There are some rules for using these methods,

(1) wait( ) method tells the calling thread to give up the monitor and go to sleep till another thread enters the same monitor and calls notify( ).

(2) notify( ) method wakes up to the first thread, which called wait( ) on the same object.

(3) notifyAll( ) method wakes up all the threads, which called wait( ) on the same object.

Their general forms are as shown below,

```
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

Example : The below program illustrates interthread communication producer/consumer problem.

```
//Creation of class "Queue"
class Queue
```

```
{
    int m;
    boolean value = false;
    synchronized int get( )
    {
        while(!value)
        try
        {
            wait( );
        }
        catch(InterruptedException ie)
        {
            System.out.println("Caught interrupted exception");
        }
        System.out.println("Received :"+m);
        value = false;
        notify( );
        return m;
    }
    synchronized void put(int m)
    {
        while (value)
        try
        {
            wait( );
        }
        catch(InterruptedException ie)
        {
            System.out.println("Caught interrupted exception");
        }
        this.m = m;
        value = true;
        System.out.println("Produced :"+m);
        notify( );
}
```

```
}
//Creation of a threaded object, the 'Producer' generate entries of a 'Queue'
class Producer implements Runnable
{
    Queue q;
    Producer(Queue q)
    {
        this.q = q;
        new Thread(this, "The Producer").start( );
    }
    public void run( )
    {
        int j = 1;
        while(true)
        {
        if(j>5)
        system.exit(0);
            q.put(j++);
        }
    }
}
// Creation of a threaded object, the 'Consumer' to consume entries of a 'Queue'
class Consumer implements Runnable
{
    Queue q;
    Consumer(Queue q)
    {
        this.q = q;
        new Thread(this, "The Consumer").start(.);
    }
```

```
        public void run( )
        {
            while(true)
            {
                q.get( );
            }
        }
    }
    // Creation of a class IThreadComm to create objects of class Queue, Producer and
       Consumer
    class IThreadComm
    {
        public static void main(String args[ ])
        {
            Queue q = new Queue( );
            new Producer(q);
            new Consumer(q);
        }
    }
```

Output

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

In the program get( ), wait( ) are called. This causes its execution to suspend until the producer notifies you that some data is ready. When this happens, execution inside get( ) resumes. After the data has been obtained, get( ) calls notify( ). This tells producer that it is okay to put more data in the queue. Inside put( ), wait( ) suspends execution until the consumer has removed the item from the queue. When execution resumes, the next item of data is put in the queue and notify( ) is called. This tells the consumer that it should now remove it.

2.17 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 Fig. 2.17.1.

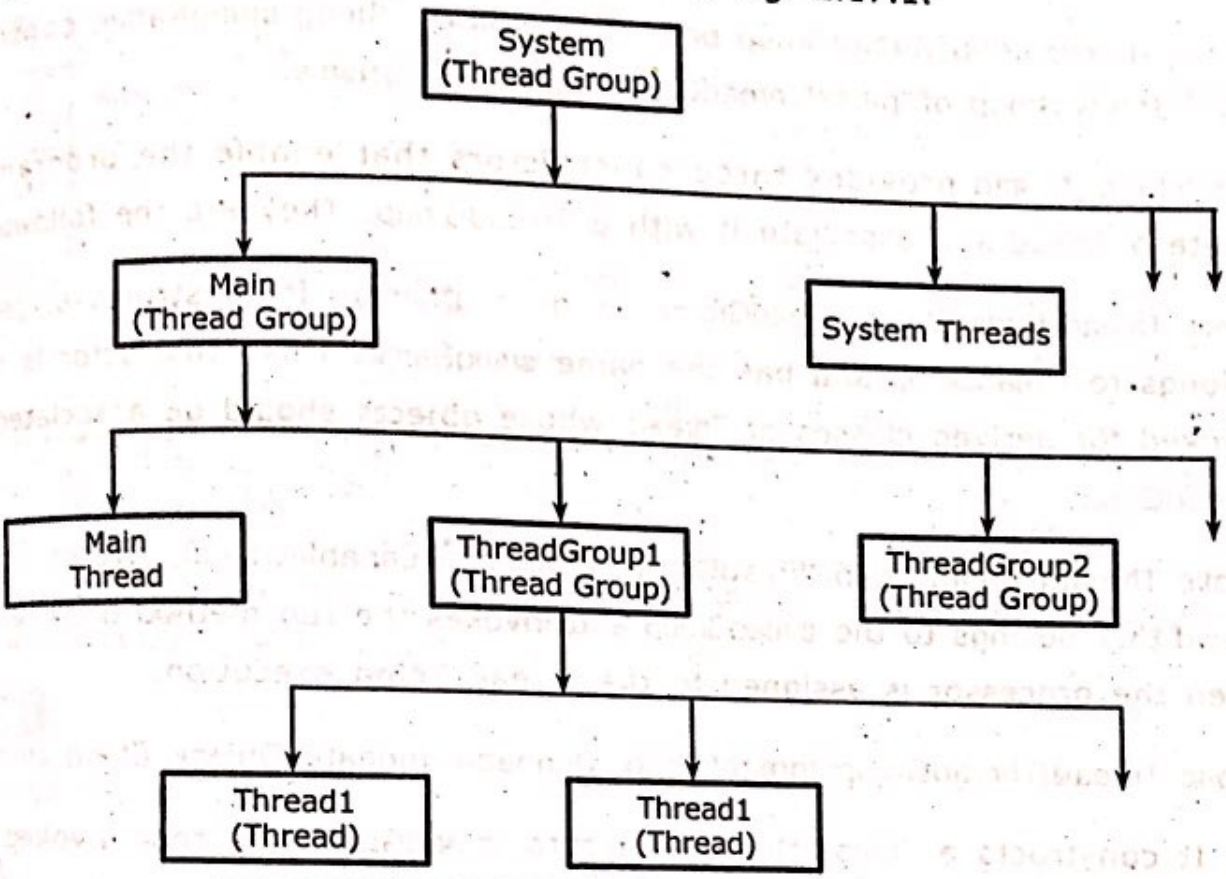

Fig. 2.17.1 Hierarchical Thread Group Structure

As shown in Fig. 2.18.1, the root of the thread group structure is 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 instruction 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 group can have members threads as well as other thread groups, and it forms a hierarchial 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,

(1) public ThreadGroup(String stringName) constructs a ThreadGroup with name stringName.

(2) 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,

(1) 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.

(2) 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 being execution.

(3) 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 being 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 class returns the total number of threads presently active. It includes the total number of threads active in the parent ThreadGroup as well as child ThreadGroup. The enumerate( ) method returns an array to 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 the group.

S.No.	Name of the Method	Function
(1)	getThreadGroup( )	Returns the reference of the ThreadGroup
(2)	getMaxPriority( )	Returns the maximum priority of the ThreadGroup
(3)	setMaxPriority( )	Sets a new maximum priority for a ThreadGroup
(4)	getName( )	Returns as a String the name of the ThreadGroup
(5)	getParent( )	Determines the parent of a ThreadGroup
(6)	ParentOf(ThreadGroup t)	Returns value true if t is the parent of the ThreadGroup, otherwise it returns false

EXAMPLE PROGRAM 1

Write a program to demonstrate the creation of thread groups and some methods which act on thread groups?

SOLUTION

```
// using thread groups
class TGroups
{
   public static void main(String[ ] args) throws Exception
   {
            //we should understand that the following statements are
              executed by
            //the main thread.
      Reservation res = new Reservation( );
      Cancellation can = new Cancellation( );
                  //create a Thread group with name
      ThreadGroup tg = new ThreadGroup("First Group");
                  //create 2 threads and add them to First Group
```

```
class Cancellation extends Thread
{
    public void run( )
    {
        System.out.println("I am cancellation thread");
    }
}
```

Output

```
Parent of tg1 = java.lang.ThreadGroup[name = First Group, maxpri=10]
Thread group of t1 = java.lang.ThreadGroup[name = First Group, maxpri=10]
Thread group of t3 = java.lang.ThreadGroup[name = Second Group, maxpri=7]
No of threads active in tg = 4
I am reservation thread
I am reservation thread
I am cancellation thread
I am cancellation thread
```

2.18 DAEMON THREADS

A daemon thread is a thread that runs only to serve other threads. Daemon threads run in the background, when CPUs 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 as 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, 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);
```

Daemon threads are just like normal threads except that they are not taken into account when deciding if an application or applet has died.

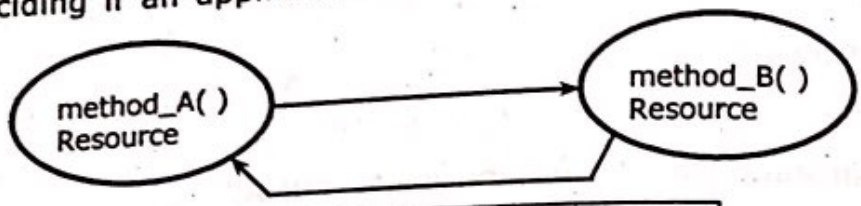

Fig. 2.18.1 Deadlock State Diagram

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.

Example : The program below is an example of setting a thread as a daemon thread.

```
public class Demo extends Thread
{
    public void run( )
    {
        for(int x=0; x<5; x++)
            System.out.println(this.getName( )+" " +x);
    }
    public static void main(String args[ ])
    {
        Demo d1 = new Demo( );
        Demo d2 = new Demo( );
        d1.setName("Daemon Thread :");
        d2.setName("Normal Thread :");
        d1.setDaemon(true);
        d1.setPriority(Thread.MIN_PRIORITY);
        d1.start( );
        d2.start( );
    }
}
```

Output

```
Daemon Thread : 0
Normal Thread : 0
Daemon Thread : 1
Normal Thread : 1
Daemon Thread : 2
Normal Thread : 2
Daemon Thread : 3
Normal Thread : 3
Daemon Thread : 4
Normal Thread : 4
```

In the above program the thread d1 is set as a daemon thread and its priority is also set as minimum i.e., 1, since the daemon threads are given less priority. The daemon threads die as soon as the execution is over and before JVM is removed from the RAM. That is why the output of d1 will not be visible.

2.19 ENUMERATIONS

An enumeration is a simple list of name constants. In Java, an enumeration defines a class type and can have constructors, methods and instance variables.

An enumeration is created using the keyword '*enum*'.

Let us consider a simple enumeration that lists various size ranges.

```
enum Size {Small, Medium, Large, Extra-large};
```

Now you can declare variable of this type as,

```
Size s = Size.Medium;
```

A variable of type Size can hold only one of the values listed in the type declaration or the special value null that indicates that the variable is not set to any value at all.

The identifiers small, medium and so on are called enumeration constants. Each is implicitly declared as a public, static, final member of size. These constants are called self-typed in which 'self' refers to the enclosing enumeration.

2.19.1 Enumeration Fundamentals

(1) The syntax for creating an enumeration is,

```
enum enum_type
{
    Ordered list of named constants separated by commas
}
```

Example

```
enum Months
{
    Jan, Feb, Mar, Apr, May, June, July, Aug,
    Sept, Oct, Nov, Dec
}
```

Where,

Months specifies enumeration type

Jan-Dec specify enumeration constants of type 'Months'

(2) The procedure for declaring a variables of enum_type is same as that of the primitive types, as shown below,

Syntax : enum_type variable;

Example : Months m;

(3) The variable of type enum_type can be assigned by only those values defined by the enumeration. For example, the value of m can be either Jan, Feb, Mar,.... or Dec. The values are assigned to the enum variables using dot operator.

Syntax : enum_var = enum_type.value;

Example : m = Months.Mar

(4) A switch statement can also be controlled using an enum value by making all the constant values of an enum as case statements.

Example

```
switch(m)
{
    case Jan: :
    case Feb: :
    case Mar: :
        :
}
```

The enum constants used in the case statement need not be qualified by their enumeration type name as Months. Jan or Months. Feb. Because, the enum variable 'm' used in the switch expression provided the enum type for all its case constants.

(5) The value of an enum constant can be displayed using a println( ) statement by specifying its qualified name. That is, enum_type.value.

Example

The statement,

```
System.out.println("Current Month:" + Months.Mar);          //prints March
```

Example

```
enum Size {Small, Medium, Large, Extra-Large}
class EnumDemo
{
    public static void main(String args [ ])
    {
        Size s;
        s = Size.small;
        System.out.println("Value of s :" + s);          //Output an enum value
        System.out.println( );
        s = Size.Medium;
        if(s==Size.Medium)                              //Compare two enum values
        System.out.println("s is Medium\n");
        switch(s)                                       //Using an enum to control switch statement
        {
            Case Small :
                System.out.println("Small size");
                break;
            Case Medium :
                System.out.println("Medium size");
                break;
            Case Large :
                System.out.println("Large size");
                break;
```

```
                Case Extra-large :
                    System.out.println ("Extra-large size");
                    break;
            }
        }
    }
```

Output

```
Value of s : Small
s is Medium
Medium size
```

2.19.2 Method – values( ) and valueOf( )

The two predefined methods associated with enumeration are,

(1) **values( ) :** This method returns the list of all named constants associated with an enumeration in the form of an array. The general form of values( ) method is,

```
public static enum_type[ ] values( )
```

Where,

enum_type specifies the type of enumerator

Example : Consider an enumeration,

```
enum Months
{
    Jan, Feb, Mar, Apr, May, June, July, Aug,
    Sept, Oct, Nov, Dec
}
```

The call Months.values( ) returns Jan Feb Mar Apr May June July Aug Sept Oct Nov Dec

(2) **valueOf( ) :** This method takes a string(s) as argument and returns the enumeration constant corresponding to 's'. The general form of valueOf( ) method is public static enum_type valueOf(String s).

Example : The call, Months.valueOf("Mar") returns Mar.

```
//Enumeration of Months in a year
    enum Months
```

```
{
    Jan, Feb, Mar, Apr, May, June, July, Aug, Sept, Oct, Nov, Dec
}
class PreDefEnum
{
    public static void main(String args [ ])
    {
        Months m;
        System.out.println("The list of enum constants in Months are:");
        for(Months m1: Months.values())
            System.out.println(m1);
        m = Months.valueOf("Mar");
            System.out.println("The variable m holds the value:" + m);
    }
}
```

Output

```
The list of enum constants in months are,
Jan
Feb
Mar
Apr
May
June
July
Aug
Sept
Oct
Nov
Dec
The variable m holds the value : Mar.
```

2.19.3 'Java Enumerations are Class Types'

A *java enumeration is a class type* because it has most of the features of a class. It can define constructors, methods and instance variables, however, it cannot be instantiated using the 'new' keyword.

An enumeration considers all its constants as its objects. As a result, whenever an enumeration constant is created, the constructor for its corresponding enum will be called. Moreover, the enumeration constants can be associated with different values that can be assigned to the instance variable of the enumeration. These values are placed within the parentheses following their corresponding enumeration constants.

Example : Consider the following enumeration class type that defines an instance variable, a constructor and a method,

```
enum Months
{
    Jan(4), Feb(2), Mar(6), Apr, May, June(6), July(1), Aug(5), Sep, Oct(3), Nov(4), Dec(1);
    private int holidays;
    Months (int h)                          //Default constructor 1
    {
        holidays = h;
    }
    Months( )                               //Default constructor 2
    {
        holidays = 1;
    }
    int getHolidays( )                      //A method
    {
        return holidays;
    }
}
```

```
class EnumClassTest
{
    public static void main(String args[ ]);
    {
        Months m;
        System.out.println("The number of holidays in the month of" + Months.Mar +
        "is" + Months.Mar.getHolidays( ));
        System.out.println("The list of months and holidays:");
        for(Months m1: Months.values( ))
        System.out.println("The month" + m1 + "has" + m1.getHolidays( ) + "holidays");
    }
}
```

Output

```
The number of holidays in the month of Mar is 6
The list of months and holidays
The month Jan has 4 holidays
The month Feb has 2 holidays
The month Mar has 6 holidays
The month Apr has 1 holidays
The month May has 1 holidays
The month June has 6 holidays
The month July has 1 holidays
The month Aug has 5 holidays
The month Sept has 1 holidays
The month Oct has 3 holidays
The month Nov has 4 holidays
The month Dec has 1 holidays
```

The enumeration 'Months' defines,

(1) A list of months in a year along with number of holidays in each month.

(2) An instance variable, 'holidays', that holds the number of holidays in a month.

In the main( ) method of EnumClassTest class, at first, the variable m is declared as of type Months. This declaration calls the constructor of Months i.e., Months( ), for each constant. The first constructor requires an argument, thus, this constructor is called for those constants that specify the argument value, for example, Mar(6). It reads the argument in 'h' and stores it in 'holidays'. The second constructor does not require any argument, thus, this constructor is called for those constants that have not specified an argument value for example, Apr.

It assigns the value 1 to the variable 'holidays' for each of these constants.

These constructors associated a 'holidays' value with each of the enum constants, as a result their corresponding values can be obtained by calling getHolidays( ) method. For example, when the Months.Mar.getHolidays( ) method is called, it returns the value of 'holidays' associated with Mar i.e., 6. Similarly, Months.Apr.getHoliday( ) returns '1'.

- The enumeration class type has following limitations,

(1) Enums do not support inheritance, i.e., an enum cannot inherit any other class.

(2) Enums are not extendible, i.e., an enum cannot be made a super class.

2.19.4 Example Programs

EXAMPLE PROGRAM 1

2.20 I/O BASICS

The java.io package includes several classes for handling files. These can be cateogirzed as follows,

(1) File.
(2) InputStream.
(3) OutputStream.
(4) Reader.
(5) Writer.
(6) RandomAccessFile.

2.20.1 Stream Classes

Stream is a flow of data between source and destination through a program. If the data flows from a source to a program, it is called input stream. If the data flows from a program to destination, it is called output stream. The source may be a file on hard disk, keyboard or a socket connected to other system. The destination may be computer screen, a file or a socket. These streams can be handled using the classes and interfaces provided by java.io package.

Java defines four abstract classes, InputStream, OutputStream, Reader and Writer. The two classes InputStream and OutputStream handles the byte streams and two classes Reader and Writer handles the character streams. These classes gives the basic functionality that is common to all subclass streams. The byte stream classes are used to read and write bytes or binary objects. The character stream classes are used to read or write characters or string.

The byte stream classes handles byte oriented I/O such as reading or writing the data bytes or executable programs or numerical data. The classes that support byte stream include,

(1) InputStream.
(2) OutputStream.
(3) FileInputStream.
(4) FileOutputStream.
(5) ByteArrayInputStream.
(6) ByteArrayOutputStream.

(7) SequenceInputStream.

(8) FilterInputStream.

(9) FilterOutputStream.

(10) BufferedInputStream.

(11) BufferedOutputStream.

(12) PushbackInputStream.

(13) PrintStream.

(14) RandomAccessFile.

The InputStream, OutputStream, FilterInputStream and FilterOutputStream are abstract classes. The InputStream and OutputStream classes are at the top of hierarchy that supports byte streams. Character-based stream have several advantages over byte streams. They can read or write 16-bit unicode character at a time whereas byte streams can read or write 8 bit byte at a time. The various classes that support character streams include,

(1) Reader.

(2) Writer.

(3) FileReader.

(4) FileWriter.

(5) CharArrayReader.

(6) CharArrayWriter.

(7) BufferedReader.

(8) BufferedWriter.

(9) PushbackReader.

(10) PrintWriter.

Reader and Writer classes are at the top of hierarchy that support character streams.

(1) **InputStream Class :** InputStream class is an abstract class that defines the basic functionality for streaming byte input which is common to its subclasses. It defines the following methods. These methods throws an IOException on error.

(i) **int available( ) :** This method returns the number of bytes that are available for reading.

(ii) **void close( ) :** This method closes the InputStream. If the program tries to read further from it, an IOException is thrown.

(iii) **int read( ) :** This method reads and returns the integer representation of the next available byte from the InputStream.

(iv) **int read (byte buff[ ]) :** This method reads buff.length bytes from the InputStream and stores them into the byte array buff. On success it returns the number of bytes actually read.

(v) **int read (byte buff[ ], int index, int nBytes) :** This method reads nBytes bytes starting at index from the input stream and stores them into the byte array buff on success it returns the number of bytes actually read.

The read( ) method returns -1 if the end of file is reached.

(vi) **long skip(long nBytes) :** This method skips the number of bytes specified by nBytes and returns the number of bytes actually skipped.

(vii) **void mark (int nBytes) :** This method is used to mark the current position in the InputStream until it completes reading of nBytes of bytes.

(viii) **void reset( ) :** This method resets the mark that was previously set on the InputStream.

(ix) **boolean markSupported( ) :** This method determines whether the mark( ) or reset( ) are supported by invoking InputStream. It returns true if these are supported otherwise it returns false.

(2) **OutputStream Class :** OutputStream is an abstract class that defines the basic functionality for streaming byte output which is common to its subclasses. It defines the following methods. These methods throws an IOException on error.

(i) **void close( ) :** This method closes the OutputStream. If the program tries to write further to it, an IOException is thrown.

(ii) **void flush( ) :** This method flushes the OutputStream.

(iii) **void write(int b) :** This method writes a single byte to the OutputStream.

(iv) **void write (byte buff[ ]) :** This method writes an array of bytes specified by buff to the OutputStream.

(v) **void write(byte buff[ ], int index, int nBytes) :** This method writes nBytes starting at index from the byte array buff to the OutputStream.

2.21 READING CONSOLE INPUT

In Java, console input can be read using a byte stream or a character stream. Using a character stream to read console input is preferred. To read console input, the System.in must be wrapped in a BufferedReader object as shown below.

```
InputStreamReaderinputstream = new InputStreamReader(System.in);
BufferedReader obj = new BufferedReader(inputstream);
```

or equivalently as,

```
BufferedReader obj = new BufferedReader(new InputStreamReader(System.in));
```

Here, System.in refers to an object of type InputStream. The above code creates a BufferedReader object that is connected to the keyboard.

BufferedReader class defines two methods to read console input. They are read( ) and readLine ( ).

Read( ) : This method is used to read a character from a BufferedReader. It reads a character and returns it as an integer value. It returns -1 if the end of the input stream is reached. This method throws an IOException. It has the following general form.

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

Example : The following program reads characters from the console or keyboard using BufferedReader class.

```
import java.io. *;
class ReadChars
{
    public static void main(String args[ ]) throws IOException
{
    char ch;
    BufferedReader obj = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("Enter character, or 9 to quit.");
    System.out.println("characters entered are:");
    while((ch=(char)br.read( ))!='9')
    {
                                                    //cast int to char
      System.out.println(ch);
      }
    }
}
```

Output

```
Enter character, or 9 to quit
Hello9
characters entered are:
Hello
```

readLine( ) : This method is used to read a string from the keyboard. It reads a string and returns a String object. This method throws an IOException. It has the following general form.

```
String readLine( ) throws IOException
```

Example : The following program reads a string from the keyboard using BufferedReader class.

```
import java.io. *;
class ReadString
{
    public static void main(String args[ ]) throws IOExceptionString line;
    BufferedReader obj = new BufferedReader (new InputStreamReader(System.in));
    System.out.println("Enter multiple lines of text, or quit to stop.");
    do
    {
        line = obj.readLine( );                    //reads a line
        System.out.println(line);
        while(! line.equals("quit"));
    }
}
```

Output

```
Enter multiple lines of text, or quit to stop.
This is first line.
This is first line.
This is second line.
This is second line.
quit
```

Output

```
Enter number :
10.5
Enter number :
20.5
Enter number :
32.6
Enter number :
76.9
Enter number :
66.6
The elements are 10.5 20.5 32.6 76.9 66.6
```

2.22 WRITING CONSOLE INPUT

The write( ) is defined by PrintStream class, which is the type of object referenced by a byte stream, System.out. This method is used to write to console output i.e., monitor. It has the following general form,

```
void write(int bytevalue)
```

Write is a low level method that writes the byte specified by bytevalue to the stream. It writes only the low order eight bits even though the bytevalue is declared as an integer.

Example : The following example illustrates the use of write( ) method.

```
class WriteUsage
{
    public static void main(String args[ ])
    {
    int ch;
    ch = 'A';
    System.out.write(ch);
    System.out.write('\n');
    }
}
```

Output

```
A
```

2.23 READING AND WRITING FILES

In java, several classes and methods are used to read and write files. These methods read byte by byte from the source file and write byte by byte to the destination file.

Two stream classes that the programmer prefers very often are,

(1) FileInputStream.

(2) FileOutputStream.

The file must be opened before reading and writing files. The procedure for opening files is to create an object one of these stream classes.

Syntax

FileInputStream(String filename) throws FileNotFoundException.

FileOutputStream(String filename) throws FileNotFoundException.

The parameter "filename" passed to the constructor represents the name of the file to be opened. In case of "FileInputStream" class, if the file is not found then an exception FileNotFoundException is thrown. On the other hand, in case of "FileOutputStream" class, if the file doesn't exist, java tries to create a file with the specified name, if it fails, it throws "FileNotFoundException". An important point to be noted here is that, in earlier versions of java (i.e., before java 2), the "FileOutputStream" class used to throw "IOException" instead of "FileNotFoundException".

Reading Files : The read( ) method of FileInputStream class reads byte by byte from the source file.

Syntax : int read( ) throws IOException

This method reads a byte one at a time and returns it as ASCII integer value. It also returns -1 when EOF is encountered. If error occurs in reading, then an IOException is thrown.

Consider the following program that demonstrates the use of read( ) method.

```
/*program reads and displays the contents of a file*/
import java.io*;
class Display
  {
     public static void main(String args[ ]) throws IOException
        {
```

```
int K;
/*creating a stream object*/
FileInputStream fd;
try
    {
        /*trying to open a file */
        fd = new FileInputStream("myfile.txt");
    }
catch(FileNotFoundException e)
    {
        System.out.println("file does not exist");
        return;
    }
        /*This loop reads and displays the file contents until
          EOF is reached */
        do
            {
                K = fd.read( );
                if(K! = -1)
                System.out.print((char) i);
            }while(K! = -1);
        /*closing the file */
        fd.close( );
    }
}
```

Writing Files : The write( ) method of FileOutputStream writes byte by byte to a file.

Syntax : void write(int valueofbyte) throws FileNotFoundException

This method writes the byte represented by the valueofbyte to the destination file. It can thrown a FileNotFoundException, if any error occurs at the time of writing.

```
            catch(IOException e)
            {
            System.out.println("Error:" + e);
            }
        }
    }
```

Compile the program as given below.

C:\> javac CopyFile.java File1.txt File2.txt.

Output

File1.txt is copied to File2.txt

Open File2.txt to view the file.

2.24 PRINTWRITER CLASS

PrintWriter is a character stream. It is a subclass of Writer class. This class is used to output the formatted information to the specified stream. It has following constructors.

Constructors

```
PrintWriter(OutputStream outStream)
PrintWriter(Writer outStream)
PrintWriter(OutputStream outStream, boolean flush)
PrintWriter(Writer outStream, boolean flush)
```

The first two constructors output the text to the specified stream without flushing the buffer. The last two constructors flushes the output stream everytime a newline character (\n) is output. The flusing is automatic if 'flush' is true. It is false, flusing does not take place.

Methods : In addition to close( ), flush( ) and write( ) methods the PrintWriter class defines two methods print( ) and println( ) for all types of objects including Object class. If the arguments passed to these methods is not a simple type then it calls the toString( ) method of the invoking object to print the result. Print( ) method just prints a string passed to it. Whereas println( ) methods appends a new line character "\n" to the string. Therefore, after each println( ) the cursor will be at the new line.

In the following program, PrintWriter is linked to System.out. System.out is a stream, which prints the output at DOS prompt.

File Name: PrintWriterDemo.java

```
import java.io.*;
public class PrintWriterDemo
{
    public static void main (String args[ ]) throws IOException
    {
        PrintWriter pw = new PrintWriter(System.out);
        pw.println(true);
        pw.println('A');
        pw.println(500);
        pw.println(40000L);
        pw.println(45.67F);
        pw.println("Hello");
        pw.println(new Integer("99"));
        pw.close( );
    }
}
```

Output

```
true
A
500
40000
45.67
Hello
99
```

2.25 STRING HANDLING

A string is a collection of characters. Java implements strings as object type of String class. Java provides three types of string handling classes,

(1) String class.

(2) StringBuffer class.

(3) StringTokenizer class.

String and StringBuffer classes are defined in the java.lang package. StringTokenizer is defined in java.util package.

String handlings refers to string operations, character extraction, string comparison, searching string, modifying a string, etc. To handle all these operations there are several methods.

2.25.1 String Class

A string is an object of class String. String literals or constants are written as a sequence of characters in double quotes. A string may be assigned in declaration to string reference as,

```
String color = "blue";
```

It initializes string reference color to refer the anonymous string object blue.

Note

(1) Once we create a string object, we cannot change the contents of string instance.

(2) A variable declared as a string reference can be changed at anytime.

2.25.1.1 String Constructors

String class has several constructors for initializing the String objects in many ways. These are as follows,

(1) **String S = new String( );**

This is the default String constructor, which creates a new String object S. This new object contains no characters and hence its length is 0.

(2) **String(char chars[ ]);**

This constructor is used to initialize the string. This constructor initializes a string by an array of characters.

Example

```
char arr[ ] = {'a', 'b', 'c'};
String S = new String(arr);
```

(3) String(char chars[ ], int startindex, int numchars);

To initialize a string with subrange of a character array the above constructor is used.

Where, startindex is the index at which string subrange beings and numchars is the number of characters to use.

Example

```
char arr[ ] = {'a', 'b', 'c', 'd', 'e', 'f'};
String S = new String(arr, 2, 3);
```

(4) String(String obj);

We can construct a string object which is same as another string object by using this constructor.

Example

```
char arr[ ] = {'H', 'E', 'L', 'L', 'O'};
String S1 = new String(arr);
String S2 = new String(S1);
```

(5) There are two constructors that initializes a string when a byte array is given. They are,

```
String(byte asciichars[ ]);
String(byte asciichars[ ], int startindex, int numchars);
```

Where, asciichars is the array of bytes. The second form is used to specify a subrange.

These constructors are used for byte-to-character conversion.

(6) String(int codepoints[ ], int startindex, int numbers);

This constructor supports the extended unicode character set. In the above constructor codepoints is an unicode points array.

(7) String(StringBuilder obj);

This constructor is used to create strings from StringBuilder class object.

2.25.1.2 Special String Operations

(1) int length( ) : It returns the number of characters in a given string.

Example

```
class Len
{
```

```
    public static void main(String[ ] args)
    {
        String s1 = new String("hello");
        System.out.println(s1.length( ));
    }
}
```

Output

```
5
```

(2) **char charAt(int where) :** It returns the character at the specified location. 'where' must be a non-negative specified location within the string. The first element of a string is considered to be at position 'zero'.

Example

```
class Pos
{
    public static void main(String[ ] args)
    {
        String s1 = new String("hello java");
        char c = s1.charAt(7);
        System.out.println("Character =" + c);
    }
}
```

Output

```
Character = j
```

(3) **void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart) :** It extracts more than one character at a time and copy the characters of the string into a character array.

The first argument is the starting index from which characters are copied in the string. The second argument is the index that is one past of the last character to be copied from the string. The third argument is the array into which the characters are copied. The last argument is the starting index, where the copied characters are placed in the character array.

Example

```
class Get
{
    public static void main(String[ ] args)
    {
        String s1 = "java programming language";
        char c[ ] = new char[10];
        s1.getChars(3, 8, c, 0);
        for(int i=0; i<(8-3); i++)
        System.out.print(c[i]);
    }
}
```

Output

```
a pro
```

(4) **boolean equals(Object str) :** It compares the two strings and returns true, if the strings contains the same characters in the same order and false otherwise. The comparison is case-sensitive.

Example

```
class Equal
{
    public static void main(String[ ] args)
    {
        String s1 = new String("india");
        String s2 = s1;
        if(s1.equals(s2))
        System.out.println("Strings are equal");
        else
        System.out.println("Strings are not equal");
    }
}
```

Output

(5) **boolean equalsIgnoreCase(String str) :** It compares the two strings and returns true, if the strings contains the same characters in the same order and false otherwise, by ignoring the case characters.

Example

```
class Equal1
{
    public static void main(String[ ] args)
    {
        String s1 = new String("india");
        String s2 = new String("India");
        if(s1.equalsIgnoreCase(s2))
        System.out.println("Strings are equal");
        else
        System.out.println("Strings are not equal");
    }
}
```

Output

```
Strings are equal
```

(6) **int compareTo(String str) :** It returns '0' if the two strings are equal and 'negative number' if the invoking string is less than str and 'positive number' if the invoking string is greater than str that is passed as an argument. The comparison is case-sensitive.

Example

```
class Comp
{
    public static void main(String[ ] args)
    {
        Strings s1 =  new String("hello");
        Strings s2 = new String("india");
        if((s1.compareTo(s2)) > 0)
        System.out.println("s1 is greater than s2");
```

```
        else if((s1.compare[illegible] 0)
        System.out.println("[illegible] is greater than s1");
        else
        System.out.println("Two strings are equal");
    }
}
```

Output

```
s2 is greater than s1
```

If we want to ignore case differences when comparing two strings, we use the syntax of the form,

```
int compareToIgnoreCase(String str)
```

(7) regionMatches( ) : It compares a specific region inside a string with another specific region in another string. It follows two methods,

(i) boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars)

The first argument is the starting index of the string that invokes the method. The second argument is a comparison string. The third argument is starting index in the comparison string. The last argument is the number of characters to compare between the two strings.

(ii) boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)

If ignoreCase is true the method ignores the case of the objects being compared.

Example

(i)

```
class Rmatch1
{
    public static void main(String[ ] args)
    {
        String s1 = new String("happy birthday");
        String s2 = new String("Happy birthday");
        if(s1.regionMatches(0, s2, 0, 5))
        System.out.println("Match");
        else
        System.out.println("Not match");
    }
}
```

Output

```
Not match
```

(ii) class Rmatch2

```
{
    public static void main(String[ ] args)
    {
        String s1 = new String("happy birthday");
        String s2 = new String("Happy birthday");
        if(s1.regionMatches(true, 0, s2, 0, 5))
        System.out.println("Match");
        else
        System.out.println("Not match");
    }
}
```

Output

Match

(8) **int indexOf(int ch), int indexOf(String str), int indexOf(char ch) :** It search for the first occurrence of a character or substring. If the character or substring is found, the index of that character or substring in the string is returned. Otherwise, it returns '-1'.

Example

```
class Index
{
    public static void main(String[ ] args)
    {
        String letter = new String("Java programming language");
        System.out.println(letter.indexOf('v'));
        System.out.println(letter.indexOf("programming"));
        System.out.println(letter.indexOf('A'));
    }
}
```

Output

```
2
5
-1
```

(9) int indexOf(int ch, int startIndex), int indexOf(String str, int startIndex), int indexOf(char ch, int startIndex)

It searches the occurrence of character or substring from the startIndex argument.

Example

```
class Indexof
{
    public static void main(String[ ] args)
    {
        String letter = new String("Java programming language");
        System.out.println(letter.indexOf('m', 2));
    }
}
```

Output

```
11
```

(10) int lastIndexOf(int ch), int lastIndexOf(char ch), int lastIndexOf(String str)

It searches the last occurrence of the character or substring. The search is performed from the end of the string towards the beginning of the string. If the character or substring is found, the index of that character in the string is return. Otherwise, it returns '-1'.

Example

```
class Lindexof
{
    public static void main(String[ ] args)
    {
        String letter = new String("Java programming language");
        System.out.println(letter.lastIndexOf('n'));
        System.out.println(letter.lastIndexOf("language"));
    }
}
```

Output

```
19
17
```

(11) int lastIndexOf(int ch, int startIndex), int lastIndexOf(String str, int startIndex), int lastIndexOf(char ch, int startIndex)

It searches the last occurrence of the character or substring starting with the startIndex to search the string from ending to the beginning.

Example

```
class Lindexof
{
    public static void main(String[ ] args)
    {
        String letter =  new String("Java programming language");
        System.out.println(letter.lastIndexOf('n', 11));
    }
}
```

Output

```
-1
```

(12) valueOf( ) : This is a static method. It is used to convert any data type including an object of any class into a string. It is overloaded in several different ways,

(i) static String valueOf(char chararray[ ])

(ii) static String valueOf(int i)

(iii) static String valueOf(double d)

(iv) static String valueOf(long l)

(v) static String valueOf(Object obj)

Example

```
int x = 10
String.valueOf(x);                    //returns string form of 10.
Date d = new Date( );
String.valueOf(d).                    //returns string form of date.
```

Example

File Name : ConvertingToString.java

```
import java.util.*;                   // for Date class
```

```
public class ConvertingToString
{
    public static void main(String args[ ])
    {
        int x = 10;
        double d = 10.5;
        Date today = new Date( );
        String str1 = String.valueOf(x);
        String str2 = String.valueOf(d);
        String str3 = String.valueOf(today);
        System.out.println(str1);
        System.out.println(str2);
        System.out.println(str3);
    }
}
```

Output

```
10
10.5
Thu Sep 16 11:51:21 GMT+0.5:30 2010
```

(13) String substring(int startIndex) : It returns a copy of the substring that begins that startIndex and return to the end of the invoking string.

Example

```
class Sub
{
    public static void main(String[ ] args)
    {
        String s = new String("Java programming language");
        System.out.println(s.substring(8));
    }
}
```

Output

```
gramming language
```

(14) **String substring(int startIndex, int endIndex) :** It returns all the characters from the beginning index upto the ending index.

Example

```
class sub2
{
    public static void main(String[ ] args)
    {
        String s = new String("program");
        System.out.println(s.substring(2, 5));
    }
}
```

Output

```
ogr
```

(15) **boolean startsWith(String str) and boolean endsWith(String str) :** The startsWith( ) method determines whether a given string begins with a specified string or not. Conversely, endsWith( ) method determines whether string ends with a specified string or not.

Example

```
class S14
{
    public static void main(String[ ] args)
    {
        String s = new String("Java is a program");
        if(s.startsWith("is"))
        System.out.println("Match");
        else
        System.out.println("Not match");
    }
}
```

Output

```
Not match
```

(16) String concat(String str) : It concatenates two string objects.

Example

```
class S15
{
    public static void main(String[ ] args)
    {
        String s = new String("Java");
        String s1 = s.concat("program");
        System.out.println(s1);
    }
}
```

Output

```
Javaprogram
```

(17) String replace(char original, char replacement) : It replaces all occurrences of one character in the invoking string with another character.

Example

```
class S16
{
    public static void main(String[ ] args)
    {
        String s = "Hello";
        System.out.println(s.replace('l', 'w'));
    }
}
```

Output

```
Hewwo
```

(18) String toLowerCase( ) : It converts characters into lowercase characters of a

Example

```
class S17
{
```

```
    public static void main(String[ ] args)
    {
        String s = "HELLO";
        System.out.println(s.toLowerCase( ));
    }
}
```

Output

```
hello
```

(19) **String toUpperCase( ) :** It converts all lowercase characters into uppercase characters of a string.

Example

```
class S17
{
    public static void main(String[ ] args)
    {
        String s = "hello";
        System.out.println(s.toUpperCase( ));
    }
}
```

Output

```
HELLO
```

(20) **String trim( ) :** It returns a copy of the invoking string from which any leading and trailing white space has been removed.

Example

```
class S18
{
    public static void main(String[ ] args)
    {
        String s = " hello";
        System.out.println(s.trim( ));
    }
}
```

Output

```
hello
```

(21) char[ ] toCharArray( ) : It returns a character array containing a copy of the characters in a string.

Example

```
class S20
{
    public static void main(String[ ] args)
    {
        String s = "hello";
        char ch[ ] = s.toCharArray( );
        for(int i=0; i<ch.length; i++)
        System.out.println(ch[i]);
    }
}
```

Output

```
h
e
l
l
o
```

(22) String toString( ) : It converts all objects into strings objects.

Example

```
class S21
{
    public static void main(String[ ] args)
    {
        Integer i = new Integer(10);
        System.out.println(i.toString( ));
    }
}
```

Output

```
10
```

2.25.2 StringBuffer Class

Once a string object is created then its contents cannot be changed. But the StringBuffer class provides the feature of creating and manipulating dynamic string information i.e., mutable strings. Every StringBuffer class is capable of storing a number of characters specified by its capacity. If the capacity of StringBuffer exceeds, then the capacity is automatically expanded to accommodate the additional characters.

2.25.2.1 StringBuffer Constructors

(1) **StringBuffer( ) :** It is default StringBuffer class constructor to create StringBuffer with no characters in it and an initial capacity of 16 characters.

(2) **StringBuffer(String str) :** It takes a string argument to create a StringBuffer containing the characters of the string argument. The initial capacity is equal to 16+number of characters in the given string as argument.

(3) **StringBuffer(int size) :** It takes an integer argument and creates a StringBuffer with no characters in it. The initial capacity is specified by the integer.

Example

```
class Sb
{
    public static void main(String[ ] args)
    {
        StringBuffer s1, s2, s3;
        s1 = new StringBuffer( );
        s2 = new StringBuffer("hello");
        s3 = new StringBuffer(45);
        System.out.println(s1.capacity( )+ " " + s2.capacity( )+ " " + s3.capacity( ));
    }
}
```

Output

```
16   21   45
```

2.25.2.2 StringBuffer Methods

(1) ***int length( ) :*** It returns the number of characters in a StringBuffer.

Example

```
class Sb1
{
```

```
    public static void main(String[ ] args)
    {
        StringBuffer s = new StringBuffer("hello");
        System.out.println(s.length( ));
    }
}
```

Output

5

(2) **int capacity( ) :** It returns the number of characters that can be stored in a StringBuffer without allocating more memory.

Example

```
class Sb2
{
    public static void main(String[ ] args)
    {
        StringBuffer s = new StringBuffer("hello");
        System.out.println(s.capacity( ));
        }
    }
```

Output

21

(3) **void ensureCapacity( ) :** It is provided to allow the programmer to guarantee that a StringBuffer has a minimum capacity, where capacity specifies the size of the buffer.

Example

```
class Sb3
{
    public static void main(String[ ] args)
    {
        StringBuffer s = new StringBuffer("hello");
        s.ensureCapacity(75);
        System.out.println(s.capacity( ));
    }
}
```

Output

75

(4) **void setLength(int len) :** It increases or decreases the length of StringBuffer.

Example

```
class Sb4
{
    public static void main(String[ ] args)
    {
        StringBuffer s = new StringBuffer("abcdefghijkl");
        s.setLength(5);
        System.out.println(s);
    }
}
```

Output

```
abcde
```

(5) **char charAt(int where) :** It returns a character at the specified where position.

Example

```
class Sb5
{
    public static void main(String[ ] args)
    {
        StringBuffer s = new StringBuffer("abcdefghijkl")
        System.out.println(s.charAt(7));
    }
}
```

Output

```
h
```

(6) **void setCharAt(int where, char ch) :** It takes an integer and a character argument and sets the character at the specified position of the character argument.

Example

```
class Sb6
```

(8) StringBuffer reverse( ) : It returns the reverse contents of the StringBuffer.

Example

```
class Sb8
{
    public static void main(String[ ] args)
    {
        StringBuffer s = new StringBuffer("abcdef");
        System.out.println(s.reverse( ));
    }
}
```

Output

```
fedcba
```

(9) append( ) : This method is overloaded 10 times in StringBuffer class to permit different data type values to get concatenated to the existing string in the string buffer.

Example

```
class Sb9
{
    public Static void main(String args[ ])
    {
        String s = "good bye";
        boolean b = true;
        int i = 7;
        double d = 33.333;
        StringBuffer sb = new StringBuffer("hello");
        sb.append(s);              //append( ) taking a string as parameter
        sb.append( ' ' );                  //to get a single space between words
        sb.append(i);              //concatenating an integer
        sb.append(b);              //append( ) taking boolean as parameter
        sb.append( ' ' );                  //append( ) taking a character as parameter
        sb.append(d);
        System.out.println(sb);
    }
}
```

(10) **delete( ) and deleteCharAt( ) :** StringBuffer also provides delete( ) and deleteCharAt( ) methods for deleting characters at any position in a StringBuffer. delete( ) takes two arguments, the starting subscript and the ending subscript. Ending subscript reads one-past. deleteCharAt( ) takes one argument, the index number of the character to delete.

Example

```
class Sb10
{
   public static void main(String args[ ])
   {                                          // to delete a range of characters
      StringBuffer sb1 = new StringBuffer("Sridhar");
      sb1.delete(3, 5);          // like substring, reads one past, deletes 'dh'
      System.out.println(sb1);  //prints Sriar
                                 //to delete a character at a particular index
      StringBuffer sb2 = new StringBuffer("Teja");
      sb2.deleteCharAt(2);              //deletes at index 2 (i.e., j is deleted)
      System.out.println(sb2);  //prints Tea
   }
}
```

Output

```
Sriar
Tea
```

2.25.3 StringTokenizer Class

In programming languages, statements are divided into individual pieces like identifiers, keywords and operators etc., called tokens. Java provides StringTokenizer class that breaks a string into its component tokens. This class is available in java.util package. Tokens are separated from one another by delimiters (special operators), white space characters such as blank, tab, new line and carriage return.

2.25.3.1 StringTokenizer Constructors

(1) StringTokenizer(String str)

(2) StringTokenizer(String str, String delimiters)

(3) StringTokenizer(String str, String delimiters, boolean delimiterAsToken)

```
    public static void main(String[ ] args)
    {
        StringBuffer s = new StringBuffer("abcdef");
        s.setCharAt(0, 'H');
        System.out.println(s);
    }
}
```

Output

```
Hbcdef
```

(7) **void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)**

It copies the substring of a StringBuffer into character array. sourceStart specifies the starting index from which characters should be copied in StringBuffer. sourceEnd specifies the index one past the last character to be copied from the StringBuffer. targetStart specifies the character array into which the characters are to be copied. targetStart specifies the starting location of the character array from where the first character should be placed.

Example

```
class Sb7
{
    public static void main(String[ ] args)
    {
        StringBuffer s = new StringBuffer("abc def");
        char ch[ ] = new char[s.length( )];
        s.getChars(3, s.length( ), ch, 0);
        for(int i=0; i<ch.length; i++)
        System.out.println(ch[i]);
    }
}
```

Output

```
def
```

In all constructors str is a string that is tokenized.

In the first constructor, the default delimiters are used.

In the second and third constructors delimiters in string specifies the delimiters.

In the third constructor if 'delimiterAsToken' is true then the delimiters are also returned as tokens when the string is tokenized otherwise, the delimiters are not returned. Delimiters are not returned as tokens by the first two constructors.

2.25.3.2 StringTokenizer Methods

(1) **int countTokens( ) :** Using the current set of delimiters the method determines the number of tokens left to be tokenized and returns the result.

(2) **boolean hasMoreTokens( ) :** It returns true if one or more tokens remain in the string and returns false if there are none.

(3) **boolean hasMoreElements( ) :** This method determines whether there are one or more tokens available in the string to extract. It returns true if there are one or more tokens, otherwise it returns false.

(4) **object nextElement( ) :** This method is used to obtain the next token. The token is returned as an object.

(5) **String nextToken( ) :** It returns the next token as string.

(6) **String nextToken(String delims) :** This method is used to obtain the next token as a string and set the delimiters string.

Example

```
import java.util.*;
class St
{
    public static void main(String[ ] args)
    {
        String s = new String("abc def gh");
        StringTokenizer t = new StringTokenizer(s);
        System.out.println(t.countTokens( ));
        while(t.hasMoreTokens( ))
        System.out.println(t.nextToken( ));
    }
```