CHAPTER

1

JAVA PROGRAMMING BASICS

Chapter Outlines

After comprehensive study of this chapter, you will be able to:

- To highlight importance, characteristics and architecture of java
- To introduce basic java concepts and syntax
- To demonstrate arrays in java
- To discus and program object oriented concepts by using java
- To describe concepts like packages and interfaces used in java
- To explain and program multithreading and exception handling in java
- To discuss concepts of file handling and program it by using java

1.1 INTRODUCTION

Java is general purpose, object oriented, high-level programming language, developed by Sun Microsystems in the year 1991. It is initially named as "Oak" by James Gosling, one of the inventors of the Java programming language. In 1995, "Oak" is renamed to "Java" because the name "Oak" did not survive legal registration. Java is a very powerful language and can meet all the requirements needed for the modern programming world. It is used to develop different programs that run in network, desktops, servers, embedded systems etc. Hence, we can say that java has applications in every sector in all environments.

1.2 BRIEF HISTORY OF JAVA

- In 1991, a research group working as part of Sun Microsystems's "Green" project headed by James Gosling was developing software to control consumer electronic devices. The goal was to develop a programming language that could be used to develop software for consumer electronic devices like TVs, VCRs, toasters, and the like. As a result the team announced a new language named "Oak".
- In 1992, the team demonstrated the application of their new language to control a list of home applications using a hand-held device with a tiny touch-sensitive screen.
- In 1994, the members of the Green project developed a World Wide Web (WWW) browser completely in Java that could run Java applets. The browser was originally called WebRunner, but is now known as HotJava.
- In 1995, Oak was renamed "Java" due to some legal snags. Many popular companies including Netscape and Microsoft announced their support to Java.
- In 1997, Sun released the Servlet API, which revolutionized server-side Web development.
- In 1999, Sun released the first version of the Java 2, Enterprise Edition (J2EE) specification that included Java Server Pages (JSP) and Enterprise JavaBeans (EJB) in a highly distributed enterprise middleware.

1.3 JAVA BUZZWORDS /Features of Java

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java buzzwords. Here are some terms that are used for java as its features:

- **Simple:** According to Sun, Java language is simple because, its syntax is similar to C and C++ and hence easy to learn after C and C++. Again confusing and rarely used features, such as explicit pointers, operator overloading etc, are removed from java.

- **Object Oriented:** Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior. Java is a true object-oriented programming language that supports encapsulation, inheritance, and polymorphism. Almost everything in Java is an object.
- **Distributed:** We can create distributed applications in java. Remote Method Invocation (RMI) and Enterprise Java Beans (EJB) are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.
- **Robust:** Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points make java robust.
- **Secure:** Security is an important concern, since Java is meant to be used in networked environments. Applets running in a Web browser cannot access a Web client's underlying file system or any other Web site other than that from which it originated. Again, Java programs run in a virtual machine that protects the underlying operating system from harm. Also, the absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization. . Furthermore, access restrictions are enforced (public, private) and byte codes are verified, which copes with the threat of a hostile compiler.
- **Architecture Neutral:** Since Java runs inside of a Virtual Machine, the program does not depend on the underlying operating system or hardware. Java code can be run on multiple platforms such as Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform independent code because it can be run on multiple platforms.
- **Portable:** We can carry java byte code to any platform and execute there. The phrase write once, run anywhere is the major concept for the portability that Java adheres. This is because byte code will run on any operating system for which there exists a compatible Virtual Machine.
- **Compiled and Interpreted:** Usually a computer language is either compiled or interpreted. Java combines both these approaches thus making Java a two-stage system. First, Java compiler translates source code into what is known as bytecode instructions. Byte codes are not machine instructions and therefore, in the second stage, Java interpreter generates machine code that can be directly executed by the machine that is running the Java program.
- **High Performance:** Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++). The performance gain is due to the JIT (Just In Time) compilation that cache the recurring process to speed up the interpretation.
- **Multithreaded:** We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.

1.4 JAVA ARCHITECTURE

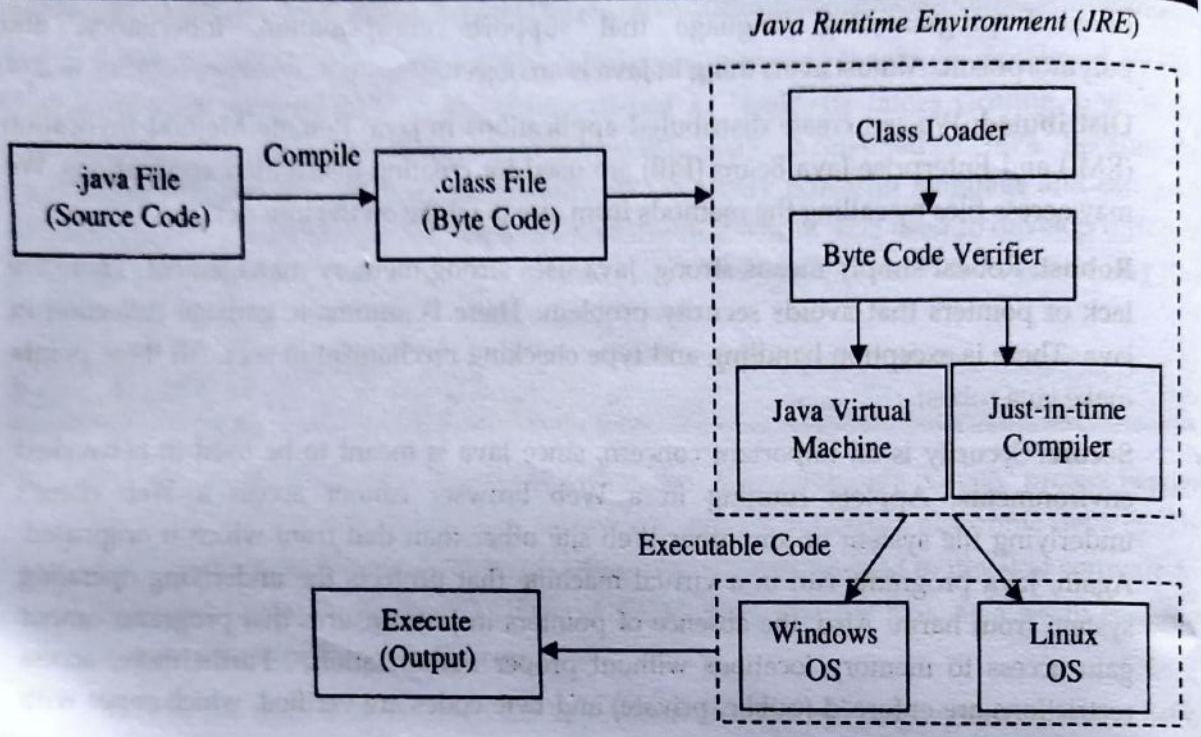


Fig 1.1 Java Architecture

Java source code is compiled into bytecode by Java compiler. Bytecode is not executable code for the target machine rather it is object code of java virtual machine (JVM). This bytecode will be stored in class files. During runtime, this bytecode will be loaded, verified and JVM interprets the bytecode into machine code which will be executed in the machine in which the Java program runs.

Class loader loads all the class files required to execute the program. Class loader makes the program secure by separating the namespace for the classes obtained through the network from the classes available locally. Once the bytecode is loaded successfully, then next step is bytecode verification by bytecode verifier.

The bytecode verifier verifies the byte code to see if any security problems are there in the code. It checks the byte code and ensures the followings.

- The code follows JVM specifications.
- There is no unauthorized access to memory.
- The code does not cause any stack overflows.
- There are no illegal data conversions in the code such as float to object references.

Once this code is verified and proven that there is no security issues with the code, JVM will convert the byte code into machine code which will be directly executed by the machine in which the Java program runs. JVM is the simulated computer within real computer. This is the component that makes java programming language platform neutral. All operating systems have this JVM that is responsible for generating executable code from bytecode generated by java compiler. If operating system does not incorporate JVM, we can install it.

You might have noticed the component "Just in Time" (JIT) compiler in above figure. This is a component which helps the program execution to happen faster. As we discussed earlier when the Java program is executed, the byte code is interpreted by JVM. But this interpretation is a slower process. To overcome this difficulty, JRE include the component JIT compiler. JIT makes the execution faster. Once the bytecode is compiled into that particular machine code, it is cached by the JIT compiler and will be reused for the future needs. Hence the main performance improvement by using JIT compiler can be seen when the same code is executed again and again because JIT make use of the machine code which is cached and stored.

1.5 SAMPLE JAVA PROGRAM

Here is the simple program in java and this can be written in any available text editors like notepad, wordpad, etc.

```
1. public class MyFirstJavaProgram
2. {
3.     public static void main(String args[])
4.   {
5.     System.out.println("This is the output of my first program");
6.   }
7. }
```

1.5.1 Dissection of the Program

Let's analyze the program that we created now in detail.

Class Declaration

Take a closer look on the first line which defines our class.

```
public class MyFirstJavaProgram
```

The keyword *"class"* is used to define a class. The statement *"class MyFirstJavaProgram"* defines a class with the name *"MyFirstJavaProgram"*. *"public"* is an access modifier which declares that our class is visible to all classes anywhere.

Opening and Closing Braces

Every class definition should begin with "{"and end with "}".

Main method

The main method of the class is defined as

```
public static void main(String args[])
```

This is the method the JVM calls to start executing our program. You can see the keywords *"public", "static" and "void"* used with the main method. The keyword *"public"* is an access modifier which declares that the method main is accessible to all other classes. The keyword *"static"* declares that the method belongs to the class and not specific to any of the objects of the

class. Thus, main method can be invoked by JVM directly through class name. The keyword *"void"* declares that the main method does not return any value. All parameters defined for any method are declared inside the brackets (). Here *"String args[]"* is a parameter to our main method. The statement fragment *"String args[]"* simply represent an a parameter named "args" which contains an array of String objects.

Output

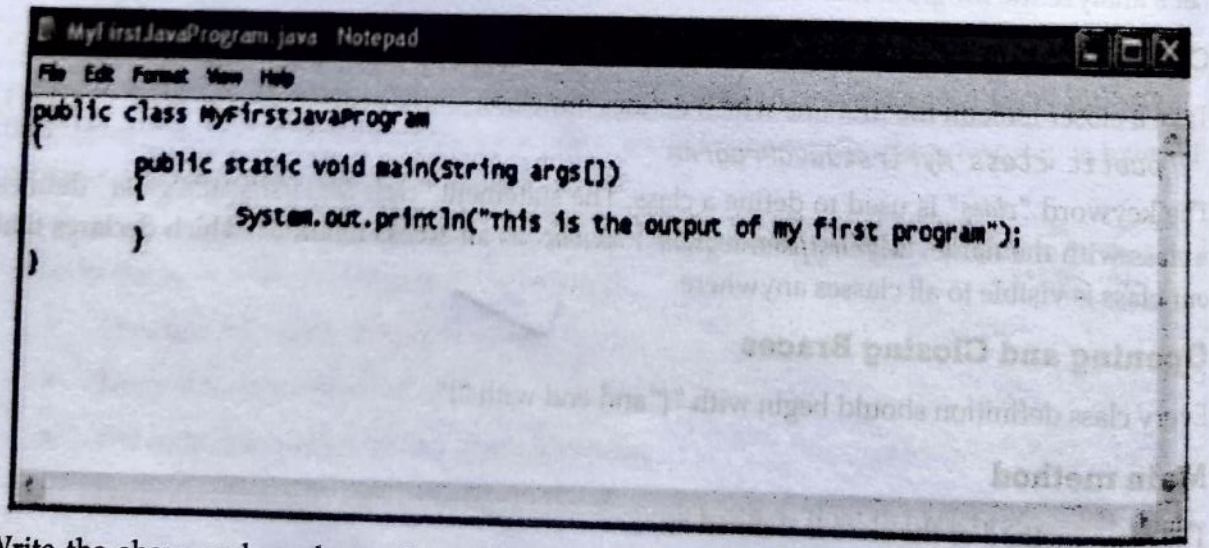
Take a closer look at the below statement.

```
System.out.println("This is the output of my first program");
```

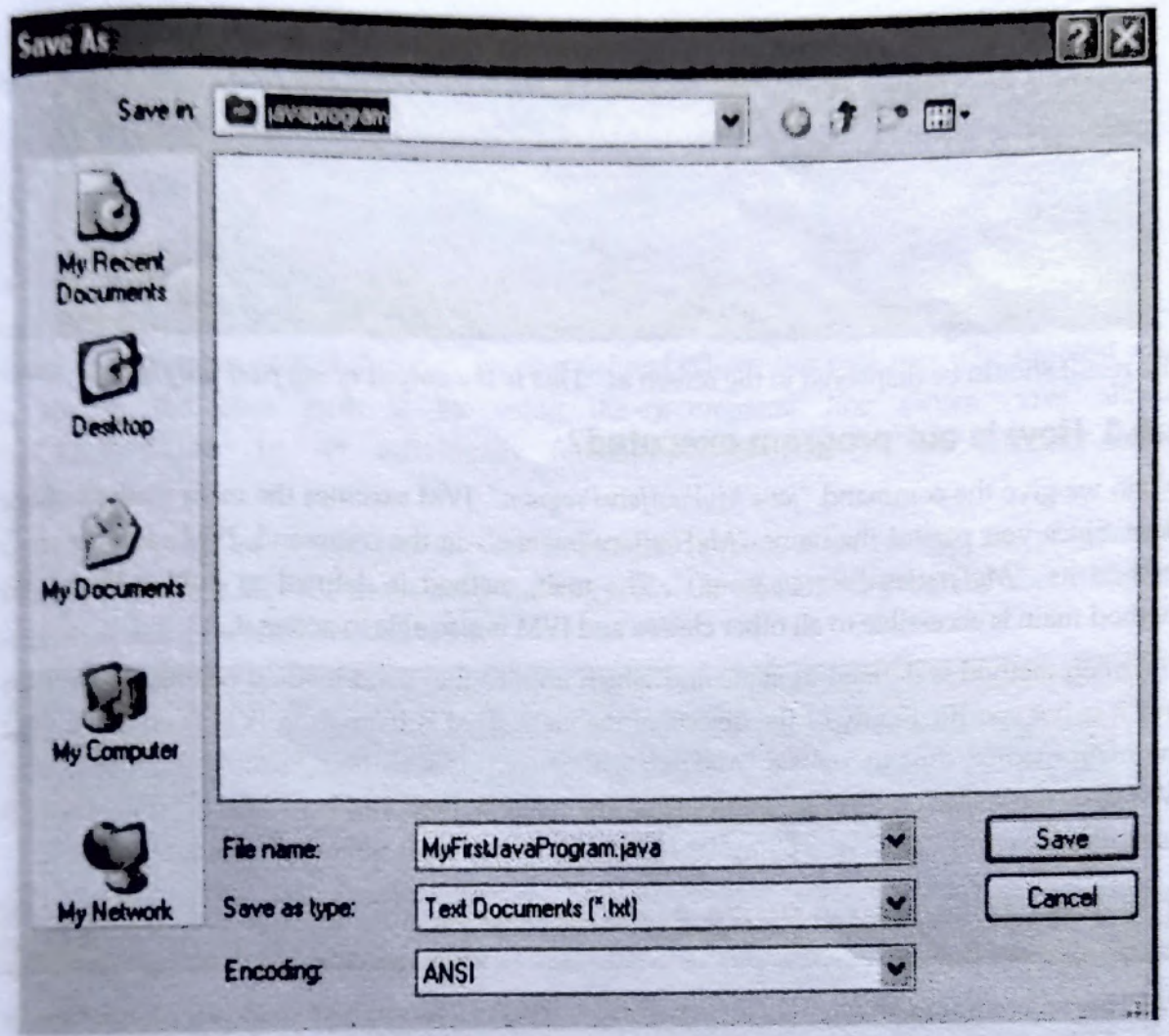
This is the code which outputs the message *"This is the output of my first program"* in the console. Here, *"System"* is a class which contains server useful methods. *"out"* is a static field of the *System* class which represents an object of *"PrintStream"* class. It represents an output stream which is already open and ready to output data. This output stream either points to display output or any other output destination specified by the user's host environment. When we run the program simply as we do now, this output stream points to the console where our message is printed. *"println"* is a method defined the class *"PrintStream"* which write the message given as its argument to the output stream and terminate the line. Here the message *"This is the output of my first program"* is given as the argument when we call the method "println" on the object of *"PrintStream"* class. As a result, the message is displayed in the console.

1.6 Compiling and Running Java Program

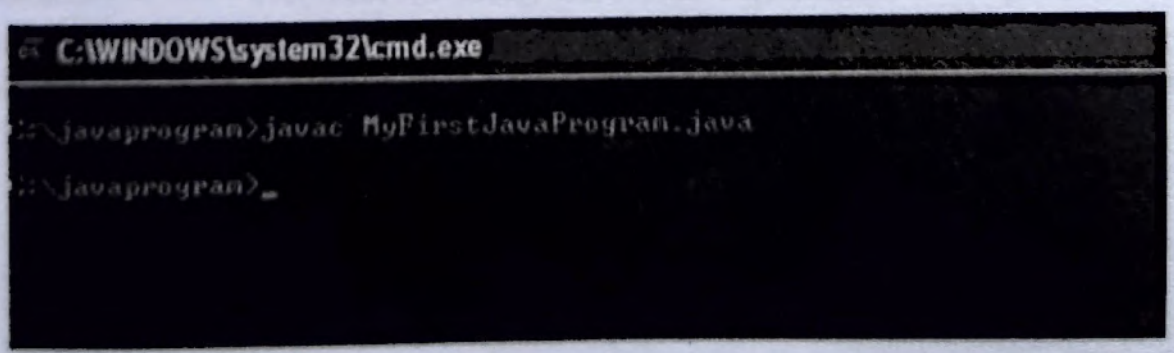
Your notepad file will look like the figure given below:



Write the above code and save it as a filename ***MyFirstJavaProgram.java*** in any convenient location. In this example, we are saving it under "C:\javaprogram".



Now, to compile the file, open a command prompt and navigate to the folder where you saved your ***MyFirstJavaProgram.java*** file. You are ready to compile here. To compile the java file use the tool ***'javac'*** that comes along with Java 2 SDK as shown below:



When this is done there are two possibilities: either there is no message and the prompt appears or there are messages that show the errors. If there are errors, your program will not compile and you must correct them to run the program.

After the successful compilation of ***MyFirstJavaProgram.java***, ***MyFirstJavaProgram.class*** file is created. This is the file that contains the byte-code. To execute the program use the tool ***'java'*** that comes along with Java 2 SDK as shown below:

The result should be displayed to the screen as *"This is the output of my first program"*

1.6.1 How is our program executed?

When we give the command *"java MyFirstJavaProgram"*, JVM executes the main method of our class. Since you passed the name *"MyFirstJavaProgram"* in the command, JVM calls the main method as *"MyFirstJavaProgram.main()"*. The main method is defined as public. Hence the method main is accessible to all other classes and JVM is also able to access it.

The main method is defined as static and which implies that main method belongs to the class and it is not specific to any of the objects of the class. That is the reason JVM is able to invoke the main method directly on the *"MyFirstJavaProgram"* class without creating any object of it. JVM does not expect any return values from the main method and hence we need to specify it using the keyword *"void"* which implies that main method does not return anything.

1.7 PATH AND CLASSPATH VARIABLES

PATH is an environment variable that tells java virtual machine where to look for tools, such as javac.exe, java.exe, javadoc.exe etc. We need to set value of PATH variable for using these tools conveniently. If we do not set the PATH variable, we need to specify the full path to the executable every time we compile or run it, such as:

```
C:\Program Files\Java\jdk-16.0.1\bin\javac MyClass.java
```

Thus, it is useful to set the PATH environment variable permanently to compile and run java programs without specifying full path of tools such as javac.exe, java.exe etc. We can set value of PATH variable permanently in windows as below:

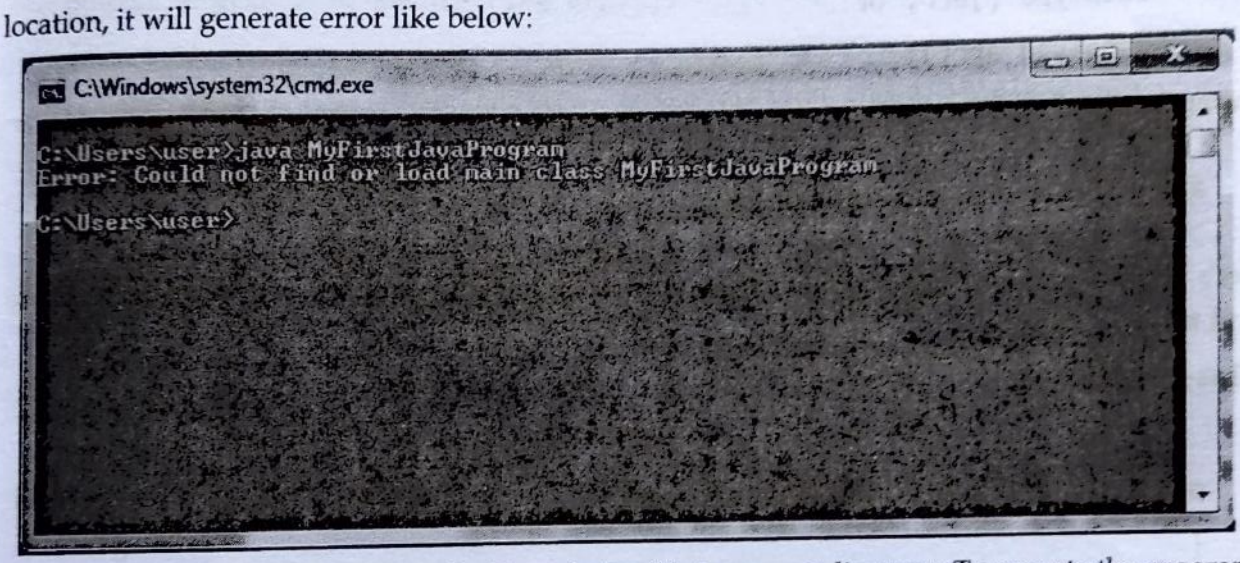
1. Go to Control Panel
2. Click on Settings
3. Click the ***Advanced*** system settings
4. Click ***Environment Variables.***
5. In the section System Variables, find the PATH environment variable and select it.
6. Click *Edit.* If the PATH environment variable does not exist, click *New.*
7. Click New
8. Specify the value of the PATH environment variable that looks like C:\Program Files\Java\jdk-16.0.1\bin
9. Click OK. Close all remaining windows by clicking OK.

Once the value of PATH variable is set, we can compile and run program without specifying full path of JVM tools like below:

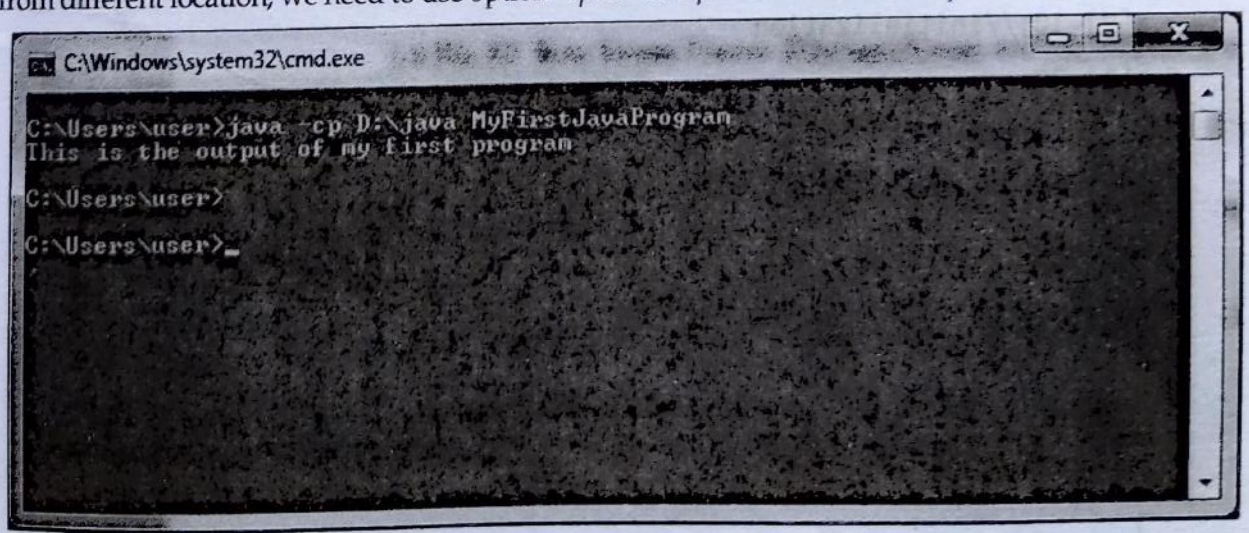
```
C:\users\user> javac MyClass.java
```

CLASSPATH is a parameter that tells the class loader where to look for user-defined classes and packages. It can be set either on the command line, or through an environment variable. Setting the CLASSPATH can be tricky and should be performed with care. The default value of the class path is current directory (denoted by period (.) symbol in windows). This means class loader only searches current directory for user defined classes and packages. The simplest way to specify the class path is by using the *-cp* command line switch. This allows the CLASSPATH to be set individually for each application without affecting other applications.

Suppose the program *Example.java* is saved in location D:\java. If we compile this program, it will create *Example.class* file in the same directory. Now if we run the program from different location, it will generate error like below:



This is because class loader searches for *Example.class* file in current directory. To execute the program from different location, we need to use option *–cp* or *–classpath* with command java like below:



To modify the CLASSPATH permanently that works for every application, we can use the same procedure ad used for the PATH variable.

1.8 Arrays in Java

Normally, array is a collection of similar type of elements that have contiguous memory location. Java array is an object that contains elements of similar data type. We can store only fixed set of elements in a java array. Array in java is index based, for n-sized array first element of the array is stored at 0 index and last element is stored in index n-1. There are two types of array.

- Single Dimensional Array
- Multidimensional Array

1.8.1 Single Dimensional Array in java

Syntax to Declare an Array in java

```
dataType[] arr; or
dataType []arr; or
dataType arr[];
```

Instantiation of an Array in java

```
arr_ref_var=new datatype[size];
```

Let's see the simple example of java array, where we are going to declare instantiate, initialize and traverse an array.

```
class Testarray
{
   public static void main(String args[])
   {
      int a[]=new int[5];//declaration and instantiation
      a[0]=10;//initialization
      a[1]=20;
      a[2]=70;
      a[3]=40;
      a[4]=50;

      //printing array
      for(int i=0;i<a.length;i++)//length is the property of array
         System.out.println(a[i]);
      }
}
```

Output

```
10
20
30
40
50
```

We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

```
class Testarray
{
   public static void main(String args[])
   {
      int a[]={3,7,9};//declaration, instantiation and initialization

      //printing array
      for(int i=0;i<a.length;i++)//length is the property of array
         System.out.println(a[i]);
      }
}
```

Output

```
3
7
9
```

1.8.2 Multidimensional array in java

In such case, data is stored in row and column based index which is also known as matrix form.

Syntax to Declare Multidimensional Array in java

```
dataType[][] ar;        or
dataType [][]ar;        or
dataType ar[][];        or
dataType []ar[];
```

Example to instantiate Multidimensional Array in java

```
int[][] ar=new int[3][3];//3 row and 3 colum
```

Example of Multidimensional java array

Let's see the simple example to declare instantiate, initialize and print the two dimensional array.

```
import java.util.*;
class TestArray
{
   public static void main(String args[])
   {
      int a[][]=new int[3][3];//declaration and instantiation
      Scanner sc=new Scanner(System.in);
      System.out.println("Enter elements");
      for(int i=0;i<3;i++)
        for(int j=0;j<3;j++)
         a[i][j]=sc.nextInt();

      System.out.println("------Enter Elements Are----------\n");
      for(int i=0;i<3;i++)
      {
          for(int j=0;j<3;j++)
          {
         System.out.print(a[i][j]+" ");
          }
      System.out.println();
      }

   }
}
```

Output

```
Enter elements
7   8   9
4   5   6
1   2   3
------Enter Elements Are---------
7       8       9
4       5       6
1       2       3
```

We can also declare, instantiate and initialize the java array together by:

```
int a[]={{3,3,4},{4,8,9},{1,7,4}};//declaration, instantiation and initialization
```

1.8.3 The Arrays Class

The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

- **Public static int binarySearch(Object[ ] a, Object key):** Searches the specified array of Object (Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, (-(insertion point + 1).
- **Public static boolean equals(long[ ] a, long[ ] a2):** Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other primitive data types (Byte, short, Int, etc.)
- **Public static void fill(int[ ] a, int val):** Assigns the specified int value to each element of the specified array of ints. Same method could be used by all other primitive data types (Byte, short, Int etc.)
- **Public static void sort(Object[ ] a):** Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. Same method could be used by all other primitive data types (Byte, short, Int, etc.)
- **Public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length):** This method copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array. A subsequence of array components is copied from the source array referenced by **src** to the destination array referenced by **dest**. The number of components copied is equal to the **length** argument.

Example

```
import java.util.Arrays ;
public class ArrayMethods
{
   public static void main(String[] args)
   {
        int a[] = { 10, 5, 19, 12, 23, 15 };
        int b[] = new int[5];

        int pos=Arrays.binarySearch(a,19);
        System.out.println("Index of 19 is:" +pos);

        boolean flag=Arrays.equals(a,b);
        System.out.println("Are a and b equal:" +flag);

        System.arraycopy(a,0,b,0,5);
```

```
        System.out.println("********Elements of b are********:");
        for(int i=0;i<5;i++)
        {
            System.out.print(b[i]+"   ");
        }
        System.out.println();
        Arrays.sort(a);
        System.out.println("********After Sorting Elements of a
are********:");
        for(int i=0;i<5;i++)
        {
            System.out.print(a[i]+"   ");
        }
    }
}
```

Output

```
Index of 19 is:2
Are a and b equal:false
********Elements of b are********:
10  5  19  12  23
********After Sorting Elements of a are********:
5  10  12  15  19
```

1.9 For Each Loop

The Java for-each loop or enhanced for loop is introduced since J2SE 5.0. It provides an alternative approach to traverse the array or collection in Java. The advantage of the for-each loop is that it eliminates the possibility of bugs and makes the code more readable. It is known as the for-each loop because it traverses each element one by one. The drawback of the enhanced for loop is that it cannot traverse the elements in reverse order. Here, you do not have the option to skip any element because it does not work on an index basis. Moreover, you cannot traverse the odd or even elements only. But, it is recommended to use the Java for-each loop for traversing the elements of array and collection because it makes the code readable.

Syntax

```
for(data-type variable : array | collection)
{
   //body of for-each loop
}
```

Example

```
//Program to demonstrate for-each loop
class ForEachTest
{
   public static void main(String args[])
   {
        int a[]={12,13,14,44};
        for(int i:a)
        {
              System.out.println(i);
        }
 }
}
```

Output

```
12
13
14
44
```

1.10 Class and Objects

Class is blueprint or template of real world objects that specifies what data and what methods will be included in objects of that class. We can also say that class is description group of objects having similar properties. A class is also called user defined data type or programmers defined data type because we can define new data types according to our needs by using classes.

On the other hand, objects are instances of classes. We can say that object is variable of class type. Memory for instance variables is not allocated at the time of class declaration rather at the time of object creation. Thus we can say that objects have physical existence and classes are only concepts.

1.10.1 Declaring Classes

The general syntax of the class definition is given below:

```
[Access][Modifiers] class className [extends superClass] [implements interface1
interface2.....]
{     //body     }
```

Here,

- **Access-** It can be public, private, default or protected and defined access restrictions applied to the class.
- **Modifiers-** It can be static or final
- **ClassName-** It is the name of class, with the initial letter capitalized by convention.
- **Superclass-** It is the name of the class's parent, if any, preceded by the keyword *extends*. A class can only extend one parent.
- **Interfaces-** Theses are comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body-** The class body generally contains variables and methods, surrounded by braces, {}.

Data is encapsulated in a class by placing data fields inside the body of the class definition. The general form of variable declaration is:

```
[Access][Modifiers] data-type variableName;
```

Here,

- **Access:** It can be public, private, default or protected and defined access restrictions applied to the class.
- **Modifiers:** It can be static or final
- **Data type:** type of the variable. It may be primitive or user defined.
- **Variable name:** name of the variable. It can be any valid identifier

To manipulate data contained in the class we add methods inside the body of the class definition. The general form is:

```
[Access][Modifiers] return-type methodName(ArgumentList) [exceptions lists]
{
    //body
}
```

Here,

- **Access:** It can be public, private, default or protected and defined access restrictions applied to the class.
- **Modifiers:** It can be static or final
- **Return-type:** the data type of the value returned by the method or void if the method does not return a value.

- **Method name:** valid identifier with convention of starting with small letter, first word as a verb and the later words start with capital letter.
- **The parameter list in parenthesis:** a comma-separated list of input parameters, preceded by their data types, enclosed by parentheses. If there are no parameters, you must use empty parentheses.
- **Exception list:** List of exception classes that can be thrown from the method to its caller.
- **The method body, enclosed between braces:** the method's code, including the declaration of local variables, goes here.

Example

```
class Box
{
   private int l;
   private int b;
   private int h;
   public void setData(int x, int y, int z)
   {
      l = x;
      b=y;
      h=z;
   }
   public int findArea()
   {
      return l*b;
   }
   public int findVolume()
   {
      Return l*b*h ;
   }
}
```

1.10.2 Creating Objects

Objects are created using the **new** operator. The **new** operator creates an object of the specified class and returns a reference to that object. The main idea of using **new** is to create the memory that is required to hold an object of the particular type in run time. To create a usable object we must finish two steps: *declaring* the variable of its type and *instantiating* the object. The following example shows the creation of an object of type Rectangle.

```
Box bx; //declaring the variable of type Rectangle
bx = new Box(); //Instantiating an object
```

The execution of first statement creates the variable that holds reference of the class Box. It points nowhere (i.e. null) as shown below:

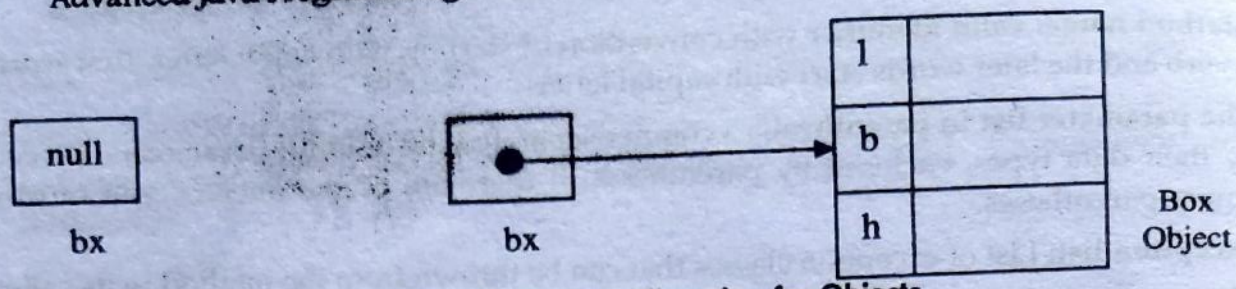


Fig 1.2 Memory allocation for Objects

When the second statement is executed then the actual assignment of object reference to the variable is done as shown in above figure (right). The above two steps can be combined into single one as below:

```
Box bx = new Box();
```

1.10.3 Using Class Members

When an object of the class is created then the members are accessed using dot operator as shown below:

```
bx.setData(3, 3,3);
bx.findArea();
bx.findVolume();
```

Not all the members can be accessed from outside the class using the dot operator. Here, we cannot access length and breadth from outside the class because they are private.

Complete Example

```
class Box
{
   private int l;
   private int b;
   private int h;
   public void setData(int x, int y, int z)
   {
      l = x;
      b=y;
      h=z;
   }
   public int findArea()
   {
      return l*b;
   }
   public int findVolume()
   {
      return l*b*h ;
   }
}
```

```
class BoxDemo
{
   public static void main(String a[])
   {
      Box b=new Box();
      b.setData(5,3,3);
      int area=b.findArea();
      System.out.println("Area of box="+area);
      int vol=b.findVolume();
      System.out.println("Volume of box="+vol);
   }
}
```

Output

Area of Box=15

Volume of Box=45

1.11 Overloading Methods

In Java, we can define different methods with the same name but either with different number of parameters or with different type of parameters, which is called method overloading. This is one of the examples of **polymorphism**. In this case the return type of the method does not matter.

Example

```
class Overloading
{
   public static void main(String[ ]args)
   {
      System.out.println(sum(5, 7));
      System.out.println(sum(5, 7, 2));
      System.out.println(sum(5.2f, 7));
      System.out.println(sum(5.3, 7.4));
   }
   static int sum(int a, int b)
   {
      return a+b;
   }
   static int sum(int a, int b, int c)
   {
      return a + b + c;
   }
```

```
    static float sum(float a, int b)
    {
        return a+b;
    }
    static double sum(double a, double b)
    {
        return a+b;
    }
}
```

Output

```
12
14
12.2
12.7
```

1.12 CONSTRUCTORS

Constructors are special type of methods that are invoked automatically at the time of object creation. Main job of constructor is to allocate memory for objects. Programmers use constructors to assign initial values to instance variables at the time of object creation. Constructor has same name as of class and looks like a method except that it has no return type. If programmer do not provide constructor, JVM will provide our program with default constructor. Name of constructor is same as the name of class but have no return type. Since constructor looks like method we can also provide arguments (parameters) to the constructor and it is also possible to overload constructors.

Example

```
class Rectangle
{
    private int length;
    private int breadth;
    public Rectangle()
    { // default constructor
        length=0;
        breadth=0;
    }
    public Rectangle(int l, int b)
    { //parameterized constructor
        length = l;
        breadth = b;
    }
    int findArea()
```

```
    {
        return length*breadth;
    }
    int findPerimeter()
    {
        return 2*(length+breadth);
    }
}
class MainRectangle
{
    public static void main(String [] args)
    {
        Rectangle rect1 = new Rectangle(10, 5);
        Rectangle rect2 = new Rectangle();
        System.out.println("First rectangle");
        System.out.println("Area:" + rect1.findArea());
        System.out.println("Perimeter:" + rect1.findPerimeter());
        System.out.println("\nSecond rectangle");
        System.out.println("Area:" + rect2.findArea());
        System.out.println("Perimeter:" + rect2.findPerimeter());

    }
}
```

Output

```
First rectangle
Area: 50
Perimeter: 30
Second rectangle
Area: 0
Perimeter: 20
```

1.13 KEYWORD "THIS"

"this" is a reference that always points to currently active object. It can be used for name conflict resolution when parameters and instance variables both have same name, as below. Besides this, we can access members of currently active object, which is not sent as argument, from methods with the help of this keyword. And we can also return reference of currently active object from methods by using this keyword.

```
class Person
{
    String name;
    int age;
    Person(String name, int age)
```

```
    {
        this.name=name;
        this.age=age;
    }
    Person()
    {

    }
    boolean compareAge(Person p)
    {
        if(this.age>p.age)
            return true;
        else
            return false;
    }
    Person min(Person p)
    {
        if(this.age<p.age)
            return this;
        else
            return p;
    }
}
class PersonDemo
{
    public static void main(String a[])
    {
        Person p1=new Person("Ramesh",29);

        Person p2=new Person ("Sonu",25);
        boolean flag=p1.compareAge(p2);
        if(flag)
            System.out.println("P1 is elder brother");
        else
            System.out.println("P1 is brother");

        Person p= new Person();
        p=p1.min(p2);

        System.out.println("Age of  brother is::"+p.age);
    }
}
```

Output

P1 is elder brother

Age of brother is::25

If we look at the constructor definition above we see the names *name* and *age* as both parameters and instance variables. Though normally in java declaring two variables within the same scope is illegal it is permissible to have same name as parameter of a method and instance variables. In this case, we use **this** keyword to identify the instance variables. Similarly, in method *compareAge* keyword ***this*** is used to access members pr object p1, which is not sent to the method as argument.

1.14 Static Members and Methods

The static modifier in java can be used with variables as well as methods. A class basically contains variables and a new copy of each variable is created when an object of that class is created or when that class is instantiated. These variables are called *instance variables*. Suppose that we want to define a variable that is common to all the objects and accessed without using a particular object if it is accessible. That is, the variable belongs to the class as a whole rather than the objects created from the class. For this, we have the solution that comes in form of the **static** keyword.

Member variables that are preceded with ***static*** keyword are called static variables. Unlike instance variables, memory for static variables is created at the time of class declaration and is accessible to all object of that class. These variables are also called class variables or non-instance variables. Similarly, methods that are preceded with ***static*** keyword are called static methods and are mainly used to access static variables. These methods can be invoked by using class name directly. Here is the program that illustrates the use of the static variables and methods.

```
class Student
{
   String name;
   int roll;
   String program;
   static int count;
   Student(String n, int r, String p)
   {
      System.out.println("Object Created");
      name = n;
      roll = r;
      program=p;
      count++;
   }
   void display()
   {
      System.out.println("Name:" + name);
      System.out.println("Roll Number:" + roll);
      System.out.println("Program:" + program);
      System.out.println();
   }
   static void displayCount()
   {
```

```
        System.out.println("Number of Students=" + count);
    }

}

class StaticDemo
{
    public static void main(String []args)
    {
        Student.displayCount();
        Student x = new Student("Sunil", 1, "CSIT");
        Student y = new Student("Roni", 2, "CSIT");
        Student z = new Student("Prakash", 3, "CSIT");
        Student.displayCount();
        System.out.println("\nStudent Records");
        x.display();
        y.display();
        z.display();

    }
}
```

Output:

```
Number of Students=0
Object Created
Object Created
Object Created
Number of Students=3

Student Records
Name:Sunil
Roll Number:1
Program:CSIT

Name:Roni
Roll Number:2
Program:CSIT

Name:Prakash
Roll Number:3
Program:CSIT
```

1.15 Access Modifiers

Keywords that are used to define visibility of classes, methods and member variables are called access modifiers. Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

- **Default Access Modifier - No keyword**: Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc. A variable or method declared without any access control modifier in a class is available to any other class in the same package but not outside of the package. Therefore, it is also called package private.
- **Private Access Modifier - private**: Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself. Private access modifier is the most restrictive access level. Class and interfaces cannot be private. Variables that are declared private can be accessed outside the class if public methods are present in the class. Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.
- **Public Access Modifier - public**: A class, method, constructor, interface etc declared public can be accessed from any other class. However if the public class we are trying to access is in a different package, then the public class still need to be imported. In case of inheritance, all public methods and variables of a class are inherited by its subclasses.
- **Protected Access Modifier - protected**: Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members class. The protected access modifier cannot be applied to class and interfaces.

Example:

(This program will not compile)

```
class Access
{
   private int a;
   int b;
   protected int c;
   public int d;
}
class AccessDemo
{
   public static void main(String a[])
   {
      Access x=new Access();
      x.a=10;//Generates compiler error because a is private and is not
             // accessible from ouside the class
      x.b=20;
      x.c=30;
      x.d=40;
   }
}
```

1.16 INHERITANCE

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order. A class that is derived from another class is called a *subclass* (also a *derived class, extended class,* or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

Inheritance uses the concept of code **reusability**. Once a super class is written and debugged, we can reuse the properties in this class in other classes by using the concept of inheritance. Reusing existing code saves time and money and increases program's reliability. An important result of reusability is the ease of distributing classes. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations. Inheritance represents the ***IS-A relationship*** between child and parent class. Note that, in Java every class is implicitly a subclass of Object class.

1.16.1 Types of Inheritance

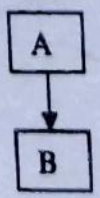
A class can inherit properties from one or more classes and from one or more levels. On the basis of this concept, inheritance may take following different forms:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance

Single Inheritance

In single inheritance, a class is derived from only one existing class. The general form and figure are given below:

```
class A
{
   members of A
}
class B  extends A
{
   own members of B
}
```



The keyword **extends** signifies that the properties of the superclass are extended to the subclass.

Multiple Inheritance

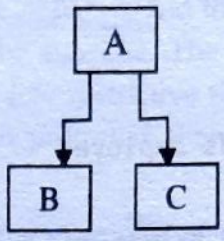
In this type, a class derives from more than one existing classes. Java classes do not support multiple inheritance. That is, a class cannot have more than one immediate superclass. However, Java provides an alternate approach known as **interfaces** to support the concept of multiple inheritance.

Hierarchical Inheritance

In this type, two or more classes inherit the properties of one existing class. The general form and figure are given below:

```
class A
{
   members of A
}
class B extends A
{
   own members of B
}

class C extends A
{
   own members of C
}
```



Multilevel Inheritance

The mechanism of deriving a class from another subclass class is known as multilevel inheritance. The process can be extended to an arbitrary number of levels. The general form and figure are given below:

```
class A
{
   members of A
}
class B extends A
{
   own members of B
}
class C  extends B
{
   own members of C
}
```

Example- Single inheritance

```
import java.util.*;
class Employee
{
   int eid;
   String ename;
   float salary;
   void getData()
   {
      Scanner sc=new Scanner(System.in);
      System.out.println("Enter id, name and salary");
      eid=sc.nextInt();
```

```
        ename=sc.next();
        salary=sc.nextFloat();
    }
    void showData()
    {
        System.out.println("ID:"+eid+"\nName:"+ename+"\nSalary:"+salary);

    }
}
class Programmer extends Employee
{
    int incentive;
    void getIncentive()
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter Incentive");
        incentive=sc.nextInt();
    }
    void showIncentive()
    {
        System.out.println("Incentive:"+incentive);
    }
    public static void main(String args[])
    {
        Programmer p=new Programmer();
        p.getData();
        p.getIncentive();
        System.out.println("!!!!!!!!!!!!Programmer Details!!!!!!!!!!!!");
        p.showData();
        p.showIncentive();
    }
}
```

Output

```
Enter id, name and salary
102
Suren
35000
Enter Incentive
12000
!!!!!!!!!!!!Programmer Details!!!!!!!!!!!!
ID:102
Name:Suren
Salary:35000.0
Incentive:12000
```

1.17 Method Overriding

The process of redefining the inherited method of the super class in the derived class is called method **overriding**. For example if we have a method called getData() in the super class called *Employee*, then the child class named *Programmer* can redefine the method to read values of member variables defined in *Programmer* class. The signature and the return type of the method must be identical in both the super class and the subclass. The overridden method in the subclass should have its access modifier same or less restrictive than that of super class. For example, if the overridden method in super class is protected, then it must be either **protected** or **public** in the subclass **but not private.**

Method overloading and method overriding are the examples of polymorphism. We can use *super* keyword to call overridden member of superclass from subclass. Keyword *super* always refers to the superclass immediately above the calling class. This is true even in multilevel inheritance. The example below shows method overriding

Example

```
import java.util.*;
class Employee
{
   int eid;
   String ename;
   float salary;

   void getData()
   {
      Scanner sc=new Scanner(System.in);
      System.out.println("Enter id, name and salary");
      eid=sc.nextInt();
      ename=sc.next();
      salary=sc.nextFloat();
   }
   void showData()
   {
      System.out.println("ID:"+eid+"\nName:"+ename+"\nSalary:"+salary);
   }
}
class Programmer extends Employee
{
   int incentive;
   void getData()//overridden
   {
      super.getData();//calling getData() of parent
      Scanner sc=new Scanner(System.in);
      System.out.println("Enter Incentive");
      incentive=sc.nextInt();
```

```
    }
    void showData()//overridden
    {
        super.showData(); //calling showData() of parent
        System.out.println("Incentive:"+incentive);
    }
    public static void main(String args[])
    {
        Programmer p=new Programmer();
        p.getData();
        System.out.println("!!!!!!!!!!!!Programmer Details!!!!!!!!!!!!");
        p.showData();
    }
}
```

Output

```
Enter id, name and salary
103
Roni
27000
Enter Incentive
10000
!!!!!!!!!!!!Programmer Details!!!!!!!!!!!!
ID: 103
Name: Roni
Salary: 27000.0
Incentive: 10000
```

1.18 Dynamic Method Dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Therefore, it is the mechanism of achieving *run-time polymorphism*. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred by a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

```
class A
{
   void callme()
   {
      System.out.println("Inside A's callme method");
```

```
    }
}
class B extends A
{
    void callme()
    {
        System.out.println("Inside B's callme method");
    }
}
class C extends B
{
    void callme()
    {
        System.out.println("Inside C's callme method");
    }
}
class Dispatch
{
    public static void main(String[]args)
    {
        A a = new A();
        B b = new B();
        C c = new C();
        A r;
        r = a;
        r.callme();
        r = b;
        r.callme();
        r = c;
        r.callme();
    }
}
```

Output:

Inside A's callme method
Inside B's callme method
Inside C's callme method

1.19 Execution of Constructors in Multilevel Inheritance

In the multilevel inheritance, constructors are called in order of derivation, from superclass to subclass. For example,

```
class A
{
    A()
    {
        System.out.println("Inside A's constructor");
```

```
    }
}
class B extends A
{
    B()
    {
        System.out.println("Inside B's constructor");
    }
}
class C extends B
{
    C()
    {
        System.out.println("Inside C's constructor");
    }
}
class ConsDemo
{
    public static void main(String[]args)
    {
        C c = new C();
    }
}
```

Output:

Inside A's constructor
Inside B's constructor
Inside C's constructor

In case of parameterized constructor, A subclass can call a constructor defined by its superclas by the use of following form of ***super***.

super(parameter-list);

Here, parameter-list specifies any parameters needed by the constructor in the superclass Keyword ***super*** always refers to the superclass immediately above the calling class. This is true even in multilevel inheritance. Also, **super** must always be the first statement executed inside a subclass constructor.

Example

```
class Room
{
    int length;
    int breadth;
    Room(int x, int y)
    { //superclass constructor
```

```
        length = x;
        breadth = y;
    }
    int area()
    {
        return length * breadth;
    }
}

class BedRoom extends Room
{
    int height;
    BedRoom(int x, int y, int z)
    {
        super(x, y); //using super
        height = z;
    }
    int volume()
    {
        return length * breadth * height;
    }
}
class ConsSuper
{
    public static void main(String[]args)
    {
        BedRoom br = new BedRoom(5, 3, 4);
        System.out.println("Area = " + br.area());
        System.out.println("Volume = " + br.volume());
    }
}
```

Output

Area=15

Volume=60

1.20 FINAL MODIFIER

The *final* modifier in java can be used with classes, methods and member variables. It can be used for following purposes:

- **To Define Constants:** We can use *final* modifier with variables to declare them as constant. Once the value is assigned to final variable, we cannot change value at runtime. If we try to change the content of the final variable the compile time error occurs.

Example

```
class Circle
{
    private final float PI=3.1415f;
    private int r;
    Circle(int x)
    {
        PI=3.1416f; //Error
        r=x;
    }
    void area()
    {
        float a=PI*r*r;
        System.out.println("Area="+a);
    }
}
```

- **To Prevent Overriding:** We can prevent a method being overridden from subclass by declaring it as *final*.

```
class A
{
    final void show()
    {
        ……..
    }
}
class B extends A
{
    void show() //Error
    {
        ……..
    }
}
```

- **To Prevent Inheritance:** We can use *final* modifier with class declaration to prevent it from further inheritance. A *final* class cannot be further inherited.

```
final class A
{

}
class B extends A //Error
{

}
```

1.21 INTERFACE

In the Java programming, an *interface* is a reference type, similar to a class that can contain *only* constants, and method declarations. There are no method bodies. Interfaces cannot be instantiated – they can only be *implemented* by classes or *extended* by other interfaces. By default all variables defined in interfaces are constants and all the members in an interface are public implicitly so we can omit the public keyword there. Unlike classes, we cannot instantiate interfaces but we can create reference variable of interface type.

To use an interface, we must write a class that *implements* the interface. When an instantiable class implements an interface, it should either provide method body for each of the methods declared in the interface or the class should be declared abstract.

1.21.1 Defining an Interface

The syntax for defining an interface is very similar to that for defining a class. To define an interface, we use the keyword **interface** instead of **class**. The general form of an interface definition is:

```
[access] interface InterfaceName [extends Iname1,Iname2]
{
   constant declarations;
   method declarations;
}
```

Here

- **Access**: It is either ***public*** or not used. With no access specifier, the interface is only available to other members of the package in which it is declared.
- **InterfaceName**: It is the name of the interface, and can be any valid identifier.
- **Constant declarations**: These are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value.
- **Method declarations**: Method declarations will contain only a list of methods without body statements. They end with semicolon after the parameter list. These methods are implicitly **abstract**.

Example:

```
interface Shape
{
    float PI = 3.1415f;
    void area();
}
```

1.21.2 Extending Interfaces

Like classes, interfaces can also be extended. The new subinterface will inherit all the member of the superinterface in the manner similar to subclasses. We can also extend several interface into a single interface. When an interface extends two or more interfaces they are separated b commas as follows:

```
interface ItemConstants
{
    int code = 1001;
    String name = "Fan";
}
interface ItemMethods
{
    void display();
}
interface Item extends ItemConstants, ItemMethods
{
    ------
    ------
}
```

1.21.3 Implementing Interfaces

Interfaces are used as superclass whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. We use ***implements*** keyword to do this. Remember that a class can implement any number of interfaces. The general form is:

```
[access][modifier]class ClassName implements IName1,IName2,...]
{
    //definition of the class
    //all the methods in the interfaces must also be defined
}
```

Example

```
interface Shape
{
```

```
    float PI = 3.1415f;
    float findArea();
}
class Circle implements Shape
{
    int r;
    Circle(int x)
    {
        r=x;
    }
    public float findArea()
    {
        return PI * r * r;
    }
}
class CircleDemo
{
    public static void main(String[]args)
    {
        Circle c1 = new Circle(3);
        float area = c1.findArea();
        System.out.println("Area = " + area);
    }
}
```

Output

Area=28.2735

1.21.4 Uses of Interface for Achieving Multiple Inheritance

Interfaces can be used for multiple inheritance. In Java a class can extend only one class and if the situation comes where we need to gather the properties of two kinds of objects then Java cannot help us doing so by using classes. In this kind of situation interface can be used to achieve multiple inheritance. Note that we can extend only one class but can implement any number of interfaces. The method signature in the class must match the method signature of the interface.

Example

```
interface Sports
{
    float sportWt = 6.0f;
    void showSportWt();
}
```

```
class Test
{
      int roll;
      float part1, part2;
      void setData(int r, float p1, float p2)
      {
         roll = r;
         part1 = p1;
         part2 = p2;
      }
      void showData()
      {
         System.out.println("Roll Number: " + roll);
         System.out.println("Marks Obtained");
         System.out.println("Part1 = " + part1);
         System.out.println("Part2 = " + part2);
      }
}
class Results extends Test implements Sports
{
      float total;
      public void showSportWt()
      {
         System.out.println("Sports Wt = " + sportWt);
      }
      void display()
      {
         total = part1 + part2 + sportWt;
         showData();
         showSportWt();
         System.out.println("Total score = " + total);
      }
      public static void main(String[]args)
      {
         Results s1 = new Results();
         s1.setData(12, 27.5f, 56.0f);
         s1.display();
      }
}
```

Output

Roll Number: 12
Marks Obtained
Part1 = 27.5
Part2 = 56.0
Sports Wt = 6.0
Total score = 89.5

1.22 Abstract Classes and Methods

If we declare a class as abstract, then it is necessary for you to subclass it so as to instantiate the object of that class i.e. we cannot instantiate object of abstract class without creating its subclass. The keyword **abstract** is used to create a class as an abstract class and it can contain abstract methods. Likewise, if you need your method to be always overridden before it can be used, you can declare the method as an abstract method using **abstract** keyword and without the method definition i.e. end it with semicolon. Remember, if you declare some method as an abstract method, you must declare your class as abstract. See the example skeleton below:

```
public abstract class A
{
   ------
   ------
   public abstract void mymethod1();  //no definition
   void myMethod2()
   {
     //.....definition here
   }
}
```

Here we cannot instantiate object of A, if we need to instantiate we must inherit class A to some other class such that it overrides the abstract methods in class A. If the derived class does not define all the abstract methods of the super class then the derived class again should be *abstract*.

1.23 Interface versus Abstract Class

- An abstract class can contain non-static and non-final fields where as an interface has static final fields by default.
- An abstract class can contain at least one abstract method whereas an interface has all methods abstract by default. That is, an abstract class can have some implemented methods but an interface does not.
- A class can extend at most one abstract class whereas it can implement any number of interfaces.
- A subclass that implements interface need to override all methods declared in interfaces which is not true in case of abstract classes.

1.24 PACKAGES

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc. A Package can be defined as a grouping of related types(classes, interfaces, enumerations and annotations) providing access protection and name space management. Some of the existing packages in Java are:

- **java.lang** - bundles the fundamental classes
- **java.io** - classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, annotations are related. Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

1.24.1 Using Package

There are two ways of accessing classes in another package. One way is to add the full package name in front of every class name.

Example

```
jav.util.Date today = new java.util.Date();
```

This process is very cumbersome so the simpler and common approach is to use *import* keyword. Using this approach you do not have to give the classes their full names. In this approach either you can import a specific class or the whole package. The import statement will be given at the top of the file just below the package statement. Here is the example for importing the package:

Example

```
import java.util.*;
Date today = new Date();

//Also you can do
import java.util.Date;
Date today = new Date();
```

When importing a number of packages you need to pay attention to packages for a name conflict. For example, both the *java.util* and *java.sql* packages have a Date class. Suppose you write a program that imports both packages.

```
import java.util.*;
import java.sql.*;
```

If you now use the Date class, then you get a compile-time error:

```
Date today; // ERROR--java.util.Date or java.sql.Date?
```

The compiler cannot figure out which Date class you want. You can solve this problem by adding a specific import statement:

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

What if you really need both Date classes? Then you need to use the full package name with every class name.

```
java.util.Date deadline = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

Locating classes in packages is an activity of the compiler. The bytecodes in class files always use full package names to refer to other classes. It should be noted that Java compiler automatically imports java.lang package in source file.

1.24.2 Creating a package

When creating a package, you should choose a name for the package and put a **package** statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package. The ***package*** statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file. If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.

Example

```
package P1;
public class A
{
   public void show()
   {
      System.out.println("From Class A of Package P1");
   }
}
```

Compile above file and put the file ***A.class*** in a sub-directory called ***P1***.
Now, we can use above class from different package as below:

```
import P1.*;
public class B
{
   public static void main(String a[])
   {
      A x=new A();
      x.show();
   }
}
```

Output

From Class A of Package P1

1.25 Exception Handling

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. When an error occurs within a method, the method creates an object and hands it off to the runtime system. This object is called *exception object* and contains information about the error. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

After a method throws an exception, runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler. The exception handler chosen is said to *catch the exception*. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates.

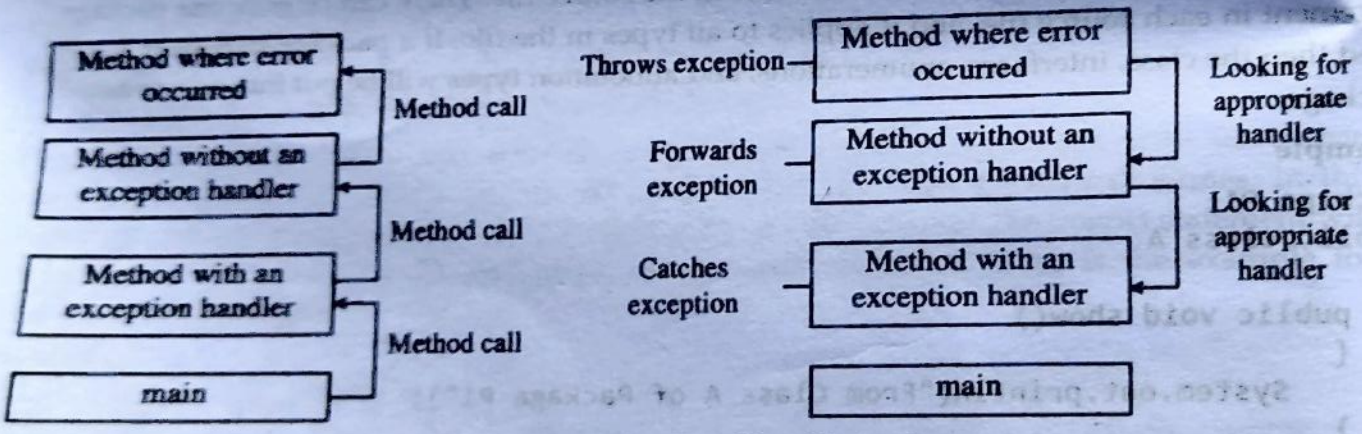


Fig. 1.3 Exception Handling Mechanism.

The core advantage of exception handling is to maintain the normal flow of the application. Exception normally disrupts the normal flow of the application that is why we use exception handling. If we perform exception handling, rest of the exception will be executed.

1.25.1 Categories of Exceptions

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner. There are three categories of exceptions:

- **Checked exceptions:** Checked exceptions are checked at compile-time and forces programmers to deal with the exceptions otherwise program will not compile. Normally theses are exceptions raised due to user error. They extend the java.lang.Exception class. IOException, SQLException etc are examples of checked exceptions.

- **Unchecked exception:** Unchecked exceptions are ignored at compile time and don't forces programmers to handle them. These exceptions are checked at runtime. They are logical programming errors and extend the java.lang.RuntimeException class. ArrayIndexOutOfBoundsException, ArithmeticException, NullPointerException etc are examples of unchecked exceptions.
- **Errors:** Errors are exceptional scenarios that are out of scope of application and it's not possible to anticipate and recover from them, for example hardware failure, JVM crash or out of memory error. They are also ignored at the time of compilation.

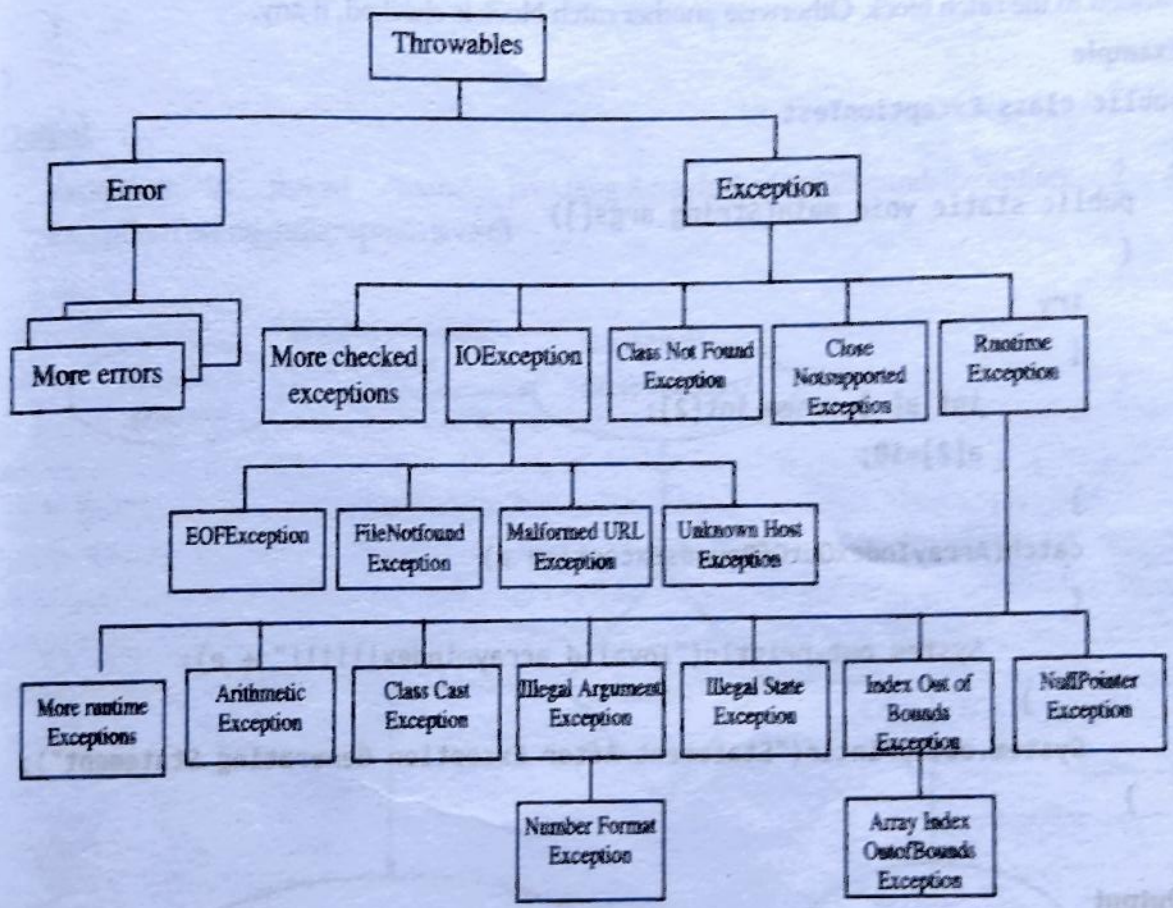


Figure 1.4 Exception Class Hierarchy.

1.25.2 Handling Exceptions with Try...Catch

A method catches an exception using a combination of the **try** and **catch** keywords. A try block is used to enclose the code that might throw an exception. It must be used within the method. Java try block must be followed by either catch or finally block. Java catch block is used to handle the Exception. It must be used after the try block only. We can also use multiple catch block with a single try. Syntax for using try/catch looks like the following:

```
try
{
    //Code
```

```
    }
    catch(ExceptionClass e)
    {
        //Catch block
    }
```

A catch statement involves declaring the type of exception we are trying to catch. If an exception occurs in the code enclosed in try block, the catch block that follows the try is checked. If the type of exception that occurred is matched with catch block, the exception is handed to the catch block. Otherwise another catch block is checked, if any.

Example

```
public class ExceptionTest
{
    public static void main(String args[])
    {
        try
        {
            int a[ ] = new int[2];
            a[2]=10;
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Invalid array index!!!!!!" + e);
        }
        System.out.println("Statement After Exception Generating Statement");
    }
}
```

Output

```
Invalid array index!!!!!!java.lang.ArrayIndexOutOfBoundsException: 2
Statement After Exception Generating Statement
```

The JVM firstly checks whether the exception is handled or not. If exception is handled by the application programmer, normal flow of the application is maintained. This means rest of the code is executed, which can be seen clearly from above example. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

This can be verified with following example. In this program exception is not handled by programmer and hence JVM handles the exception itself.

```
class ExceptionTest
{
    public static void main(String args[])
    {
          int a[ ] = new int[2];
           a[2]=10;
       System.out.println("Statement After Exception Generating Statement");
    }
}
```

Output

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2 at ExceptionTest.main(Exception.java:6)

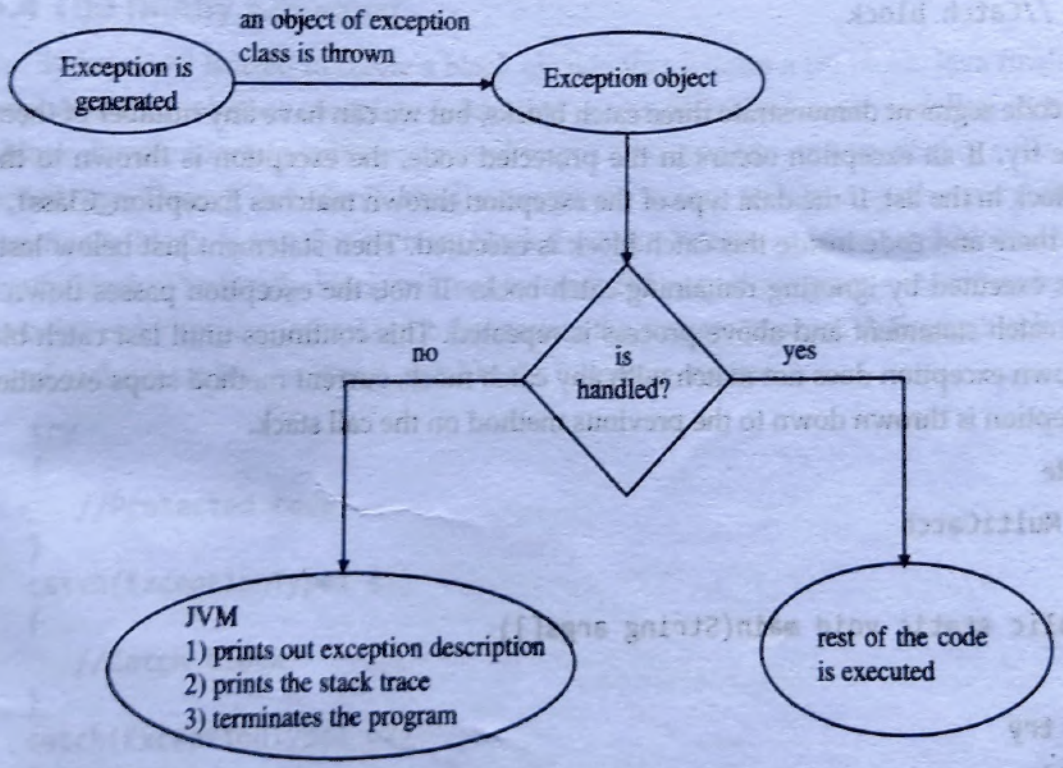


Fig 1.5 Effect of Exception Handling

1.25.3 Multiple Catch Blocks

A try block can be followed by any number of catch blocks. All catch blocks must be ordered from most specific to most general. For example, catch for ArithmeticException must come before catch for Exception. Even if there is possibility of occurring several exceptions, at a time only one exception is occurred and at a time only one catch block is executed. The syntax for multiple catch blocks looks like the following:

```
try
{
   //Protected code
}
catch(Exception_Class1 e1)
{
   //Catch block
}
catch(Exception_Class2 e2)
{
   //Catch block
}
catch(Exception_Class3 e3)
{
   //Catch block
}
```

Above code segment demonstrate three catch blocks, but we can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches Exception_Class1, it gets caught there and code inside this catch block is executed. Then statement just below last catch block is executed by ignoring remaining catch bocks. If not, the exception passes down to the second catch statement and above process is repeated. This continues until last catch block. If the thrown exception does not match with any catch mock, current method stops execution and the exception is thrown down to the previous method on the call stack.

Example

```
class MultiCatch
{
   public static void main(String args[])
   {
      try
      {
            int a[]=new int[5];
            a[5]=30/0;
       }
      catch(ArithmeticException e)
      {
         System.out.println("Division by Zero!!!!!!");
      }
      catch(ArrayIndexOutOfBoundsException e)
```

```
        {
            System.out.println("Invlaid array Index!!!!!");
        }
        catch(Exception e)
        {
            System.out.println("Error"+e);
        }
        System.out.println("rest of the code...");
    }
}
```

Output

Division by Zero!!!!!!

rest of the code..

1.25.4 The finally Keyword

The finally keyword is used to create a block of code that follows a try block. Java finally block is always executed whether exception is handled or not. This block must be followed by try or catch block. Finally block in java can be used to put cleanup code such as closing a file, closing connection etc. Even if we don't handle exception, before terminating the program, JVM executes finally block, if any. For each try block there can be zero or more catch blocks, but only one finally block. The finally block will not be executed only if program exits either by calling System.exit() or by causing a fatal error that causes the process to abort. A finally block appears at the end of the catch blocks, if any, and has the following syntax:

```
try
{
    //Protected code
}
catch(ExceptionType1 e1)
{
    //Catch block
}
catch(ExceptionType2 e2)
{
    //Catch block
}
catch(ExceptionType3 e3)
{
    //Catch block
}

finally
{
    //The finally block always executes.
}
```

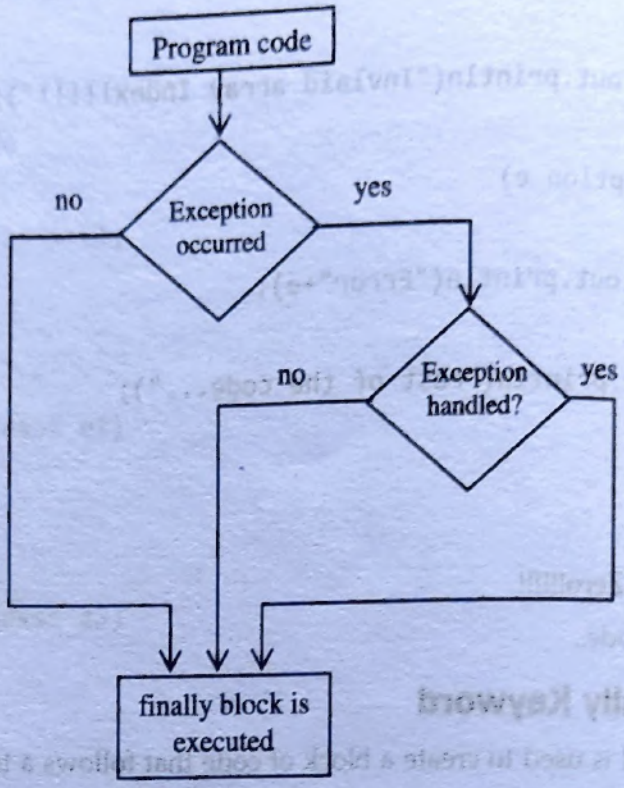


Fig. 1.6: Execution of finally Block

Example

```
public class TestFinally
{
    public static void main(String args[])
    {
        try
        {
            int x=25/0;
            System.out.println("Quotient="+x);
        }
        catch(ArithmeticException e)
        {
            System.out.println(e.getMessage());
        }
        finally
        {
            System.out.println("Finally block is always executed");
        }
        System.out.println("rest of the code...");
    }
}
```

Output

/ by zero

Finally block is always executed

rest of the code...

1.25.5 The throws Keywords

If a method does not handle a checked exception by using try....catch...finally blocks, the method must declare it using the **throws** keyword. It throws the exception to immediate calling method in the hierarchy. In this case, either the calling method should have try.....catch block to handle the exception or should use throws keyword and forward the exception to its caller. This can be again forwarded up in the hierarchy. If all calling methods are using throws and no more calling method is available to handle the exception, JVM itself will handle the exception. The throws keyword appears at the end of a method's signature. By using the **throws** keyword, we can throw an exception, either by creating new instance of some exception class or by using exception object that is caught just.

Example

```
public class ThrowsDemo
{
   void divide(int x, int y) throws ArithmeticException
   {

      int r=x/y;  //Exception is occurred here  if y=0
      System.out.println("Quotient="+r);
   }
   public static void main(String args[])
   {
      ThrowsDemo t=new ThrowsDemo();
      try
      {
         t.divide(25,5);
         t.divide(25,0);
      }
      catch(ArithmeticException e)
      {
         System.out.println(e.getMessage());
      }
   }
}
```

Output

```
Quotient=5
/ by zero
```

In above program exception generated in *divide()* method is not handled there rather it is handed to it caller, *main()* method in this case. Thus method call statement is kept is try block in *main()* method and exception is handled by placing appropriate catch handler. If *main()* method does not handle exception, we can put throws clause just after main method and can hand job of exception handling to JVM.

1.25.6 The throw Keyword

The Java throw keyword is used to explicitly throw an exception. We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. Using throws we handle certain situations in our program. The example given below will validate if the age enter by the user is more than 18 or not. Like throws clauses, exception thrown by throw keyword should be handled in its caller method or caller should throw the exception again to its caller.

Example

```
import java.util.*;
public class ThrowTest
{
   void validate(int age)
   {
      try
      {
         if (age<18)
         throw new Exception("Invalid Age");
         else
         System.out.println("Save the age");
      }
      catch(Exception e)
      {
         System.out.println(e.getMessage());
      }
   public static void main(String args[])
   {
      ThrowTest t=new ThrowTest();
      Scanner sc=new Scanner(System.in);
      System.out.println("Enter Age");
      int a=sc.nextInt();
      t.validate(a);
   }
}
```

Output

First Run

Enter Age

22

Save the age

Second Run

Enter Age

16

Invalid Age

1.25.7 Creating Exception Classes

Java platform provides a lot of exception classes that we can use. If we need an exception type that isn't represented by those in the Java platform, we can write one of our own. When we create custom exceptions in Java, we extend either Exception class or RuntimeException class. If you want to write a runtime exception, you need to extend the RuntimeException class.

Example

```
class InvalidAgeException extends Exception
{
   InvalidAgeException(String s)
   {
      super(s);
   }
}
class TestCustomException1
{
   static void validate(int age)throws InvalidAgeException
   {
         if(age<18)
            throw new InvalidAgeException("not valid");
         else
            System.out.println("welcome to vote");
   }
   public static void main(String args[])
   {
         try
         {
            validate(13);
         }
         catch(Exception m)
         {
            System.out.println("Exception occured: "+m);
         }
         System.out.println("rest of the code...");
   }
}
```

Output

Exception occured: InvalidAgeException: not valid

rest of the code...

1.26 MULTITHREADING

The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program called a thread. Each thread has a separate path of its execution. Multithreading enables us to write programs in a way where multiple activities can proceed concurrently in the same program. Thus multithreading can be used to achieve multitasking.

1.26.1 Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable (blocked) and terminated. There is no running state. But for better understanding the threads, it is better to explain it in the 5 states. The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

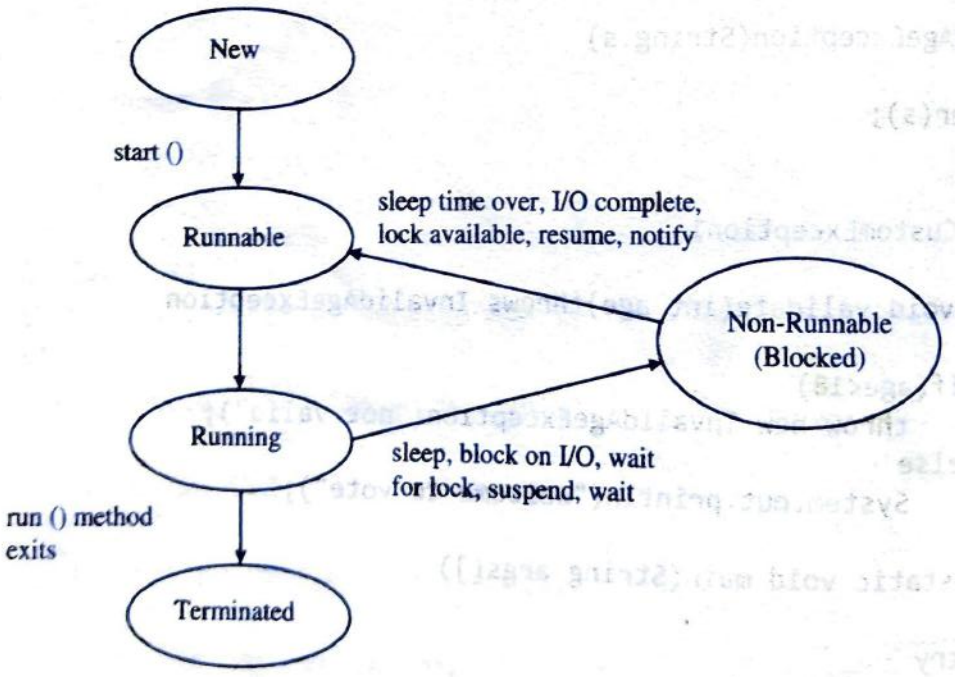


Figure 1.7: Thread State Transition.

- **New:** The thread is in new state if we create an instance of Thread class but before the invocation of start() method.
- **Runnable:** The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
- **Running:** The thread is in running state if the thread scheduler has selected it for execution.

- **Non-Runnable (Blocked):** This is the state when the thread is still alive, but is currently not eligible to run. It may be waiting for certain event to occur or for specified time to be elapsed.
- **Terminated:** A thread is in terminated or dead state when its run() method exits.

1.26.2 Writing Multithreaded Programs

A thread can be created in two ways: 1) By extending Thread class 2) By implementing Runnable interface.

Multithreading by Implementing Runnable Interface

One way of creating a thread is to create a class that implements the Runnable interface. We must need to give the definition of run() method. This run method is the entry point for the thread and thread will be alive till run method finishes its execution. Once the thread is created it will start running when start() method gets called. Basically start() method calls run() method implicitly. This way of thread implementation can be formalized into following three steps:

- As a first step we need to implement a run() method provided by Runnable interface. This method provides entry point for the thread and we need to put complete business logic inside this method. Following is simple syntax of run() method:

  public void run()

- At second step we need to instantiate a Thread object using the following constructor:

  Thread(Runnable th, String tname) or

  Thread(Runnable th)

  Here, *"th"* is an instance of a class that implements the Runnable interface and *"tname"* is the name given to the new thread.

- Once Thread object is created, we can start it by calling start() method, which executes a call to run() method. Following is simple syntax of start() method:

  void start()

Example

```
class RunnableDemo implements Runnable
{
   String tname;
   RunnableDemo(String n)
   {
        tname=n;
           System.out.println("Creating:"+tname);
   }
   public void run()
   {
      System.out.println("Running:"+tname );
```

```
        try
        {
                for(int i = 4; i > 0; i--)
            {
                        System.out.println("Thread:" + tname + "   Printing:"
i);
                                Thread.sleep(50);
                }
        }
        catch (InterruptedException e)
        {
                System.out.println("Thread " +  tname + " interrupted.");
            }
            System.out.println("Thread:" + tname + " exiting");
    }
}
public class TestThread
{
    public static void main(String args[])
    {
            RunnableDemo R1 = new RunnableDemo("One");
            Thread t1=new Thread(R1);
            t1.start();
               RunnableDemo R2 = new RunnableDemo("Two");
            Thread t2=new Thread(R2);
            t2.start();
    }
}
```

Output

```
Creating:One
Creating:Two
Running:One
Running:Two
Thread:One Printing:4
Thread:Two Printing:4
Thread:One Printing:3
Thread:Two Printing:3
Thread:One Printing:2
```

```
Thread:Two Printing:2
Thread:One Printing:1
Thread:Two Printing:1
Thread:One exiting
Thread:Two exiting
```

Multithreading by Extending Thread Class

This is the second way of creating a thread. Here we need to create a new class that extends the Thread class. The class should override the run() method which is the entry point for the new thread as described above. Then we need to call start() method to start the execution of a thread. This process can be formalized with following two steps.

- Override **run( )** method available in Thread class. This method provides entry point for the thread and we need to put our complete business logic inside this method. Following is simple syntax of run() method.

  public void run( )

- Once Thread object is created, start it by calling **start( )** method, which executes a call to run( ) method. Following is simple syntax of start() method:

  void start( );

Example

```
class ThreadDemo extends Thread
{
   String tname;
   ThreadDemo(String n)
   {
      tname=n;
         System.out.println("Creating:"+tname);
   }
   public void run()
   {
         System.out.println("Running:"+tname );
         try
      {
            for(int i = 4; i > 0; i--)
         {
                  System.out.println("Thread:" + tname + "   Printing:" +
i);
                  Thread.sleep(500);
            }
      }
```

```
        catch (InterruptedException e)
        {
                System.out.println("Thread " +  tname + " interrupted.");
            }
            System.out.println("Thread:" + tname + " exiting");
    }
}
public class TestThread
{
    public static void main(String args[])
    {

            ThreadDemo x = new ThreadDemo("t1");
            x.start();
                ThreadDemo y = new ThreadDemo("t2");
            y.start();
    }
}
```

Output

```
Creating:t1
Creating:t2
Running:t1
Thread:t1 Printing:4
Running:t2
Thread:t2 Printing:4
Thread:t1 Printing:3
Thread:t2 Printing:3
Thread:t1 Printing:2
Thread:t2 Printing:2
Thread:t1 Printing:1
Thread:t2 Printing:1
Thread:t1 exiting
Thread:t2 exiting
```

1.26.3 Thread Properties and Methods

We can get and set the values most of the thread properties. Major properties of Java threads are: Id, Name, Priority, currentThread, isAlive, etc. Some of the thread methods are static and others are non-static. Al already discussed, static methods can be invoked directly through

Thread class. But non-static methods need to be invoked by using object of Thread class. Following is the list of important methods available in the Thread class that van be used for getting and setting values of thread properties.

- **Public final void setName(String name):** Changes the name of the Thread object. There is also a getName() method for retrieving the name.
- **Public final void setPriority(int priority):** Sets the priority of this Thread object. The possible values are between 1 and 10. There is also a getPriority() method for retrieving the priority.
- **Public final boolean isAlive():** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion
- **Public static void yield():** Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled
- **Public static void sleep(long millisec):** Causes the currently running thread to block for at least the specified number of milliseconds
- **Long getId():**This method returns the identifier of this Thread.
- **Void join():**Waits for this thread to die.

Example

```
class RandFactThread extends Thread
{
   String tname;
   RandFactThread(String name)
   {
      this.setName(name);
         System.out.println("Creating Thread:"+name);
   }
   public void run()
   {
         int limit,i=0, n,fact;
            limit = (int) (Math.random() * 10+ 1);
         while(i<=limit)
            {
               n = (int) (Math.random() * 10 + 1);
               System.out.println("Number=" + n);
               fact=1;
               for(int k=1;k<=n;k++)
                  fact = fact*k;
                  System.out.println(this.getName()+":Factorial of "+n+"="
+ fact);
               i++;
            }
```

```
            System.out.println("Thread:" + this.getName() + " exiting");
    }
}
public class MultiThread
{
    public static void main(String args[]) throws Exception
    {
        RandFactThread x = new RandFactThread("fact1");
        System.out.println("ID of thread:"+x.getId());
        RandFactThread y = new RandFactThread("fact2");
        System.out.println("ID of thread:"+y.getId());
        x.setPriority(10);
        x.start();
            y.start();
        x.join();
        y.join();
        System.out.println("Exiting main thread");
    }
}
```

Output

```
Creating Thread:fact1
ID of thread:8
Creating Thread:fact2
ID of thread:9
fact2:::Number=3
fact1:::Number=7
fact1:::Factorial of 7 =5040
fact1:::Number=4
fact1:::Factorial of 4 =24
fact1:::Number=1
fact2:::Factorial of 3 =6
fact1:::Factorial of 1 =1
fact2:::Number=9
Thread:fact1 exiting
fact2:::Number=5
fact2:::Factorial of 5 =120
fact2:::Number=8
fact2:::Number=7
```

fact2:::Factorial of 7 =5040

Thread:fact2 exiting

Exiting main thread

1.26.4 Thread Priorities

We can set priorities of java threads. Java provides setPriority() and getPrority() methods for setting and reading priorities of threads. Value of priority can range from 1-10. Default value of priority is 5. Maximum priority of thread is 10 and minimum priority is 1. High priority threads can get more chances of execution from JVM. However, exact behavior depends upon JVM.

Example

```
class ThreadDemo extends Thread
{
        String tname;
        ThreadDemo(String n)
        {
            tname=n;
        }
        public void run()
        {
                for(int i = 9; i > 0; i--)
            {
                        System.out.println(tname + "  Printing:" + i);
                    }
        }
}
public class ThreadPriority
{
       public static void main(String args[])
       {
          ThreadDemo x = new ThreadDemo("T1");
          ThreadDemo y = new ThreadDemo("T2");
          ThreadDemo z = new ThreadDemo("T3");
          x.setPriority(10);
          y.setPriority(1);
          int p=z.getPriority();
          System.out.println("Priority of z:"+p);
          x.start();
          y.start();
          z.start();
       }
}
```

Output

Priority of z:5
T1 Printing:9
.
.
T1 Printing:1
T3 Printing:9
.
.
T3 Printing:1
T2 Printing:9
.
.
T2 Printing:1

1.26.5 Thread Synchronization

Synchronization means only one thread must be able to use shared resource at a time. If threads are not synchronized properly, it may lead to problem of race condition. Synchronized keyword is used in java for synchronizing threads.

Example

```
class Table
{
    synchronized void printTable(int n)
    {
        System.out.println("\nTable of "+n);
        for(int i=1;i<=10;i++)
        {
            System.out.println(i*n+"\t");
            try
            {
            Thread.sleep(100);
            }
            catch(Exception e)
            {
                System.out.println(e);
            }
        }
    }
}
class MyThread extends Thread
{
```

```
    Table t;
    int n;
    MyThread(Table t, int n)
    {
        this.t=t;
        this.n=n;
    }
    public void run()
    {
        t.printTable(n);
    }
}
class SynTest
{
    public static void main(String[] a)
    {
        Table t=new Table();
        MyThread x=new MyThread(t,4);
        MyThread y=new MyThread(t,5);
        MyThread z=new MyThread(t,9);
        x.start();
        y.start();
        z.start();
    }
}
```

Output

```
Table of 5
5  10  15  20  25  30  35  40  45  50
Table of 9
9  18  27  36  45  54  63  72  81  90
Table of 4
4  8   12  16  20  24  28  32  36  40
```

1.27 File Handling in Java / working with files.

Java I/O (Input and Output) is used to process the input and produce the output based on the input. Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations. A stream is a sequence of data. In Java a stream is composed of bytes.

In java, 3 streams are created for us automatically. All these streams are attached with console.

- **System.out:** standard output stream
- **System.in:** standard input stream
- **System.err:** standard error stream

Java application uses an output stream to write data to a destination, it may be a file, an array peripheral device or socket. And uses an input stream to read data from a source, it may be file, an array, peripheral device or socket.

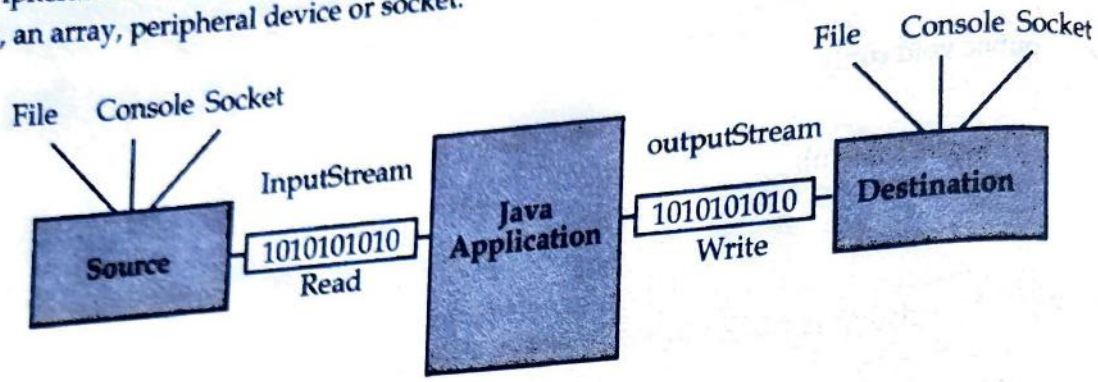


Fig. 1.8: Concept of Streams

1.27.1 Byte Stream Classes

The package java.io provides two set of class hierarchies - one for handling reading and writing of bytes, and another for handling reading and writing of characters. The abstract classes *InputStream* and *OutputStream* are the root of inheritance hierarchies handling reading and writing of bytes respectively. Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, *FileInputStream* and *FileOutputStream*. Following is an example which makes use of these two classes to copy an input file into an output file:

Example

```
import java.io.*;

public class IOStream
{
   public static void main(String args[]) throws IOException
   {
        FileInputStream in = null;
        FileOutputStream out = null;
        try
        {
             in = new FileInputStream("input.txt");
             out = new FileOutputStream("output.txt");
                  int c;
```

```
            while ((c = in.read()) != -1)
        {
            out.write(c);
            }
    }
    finally
    {
            in.close();
            out.close();
    }
  }
}
```

InputStream class defines the following methods for reading bytes

- ***int read():*** Reads the next byte of data from this input stream. The value byte is returned as an int in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned..
- ***int read(byte b[]):*** Reads up to b.length bytes of data from this input stream into an array of bytes. The read method of InputStream calls the read method of three arguments with the arguments b, 0, and b.length.
- ***int read(byte b[], int offset, int length):*** Reads up to len bytes of data from this input stream into an array of bytes, but starts atoffset bytes into the array.

Subclasses of InputStream implement the above mentioned methods.

OutputStream class defines the following methods for writing bytes

- ***void write(int b):*** Writes the specified byte to this output stream. The general contract for write is that one byte is written to the output stream. The byte to be written is the eight low-order bits of the argument b. The 24 high-order bits of b are ignored.
- ***void write(byte b[]):***Writes b.length bytes from the specified byte array to this output stream. The general contract for write(b) is that it should have exactly the same effect as the call write(b, 0, b.length).
- ***void write(byte b[], int offset, int length):*** Writes len bytes from the specified byte array starting at offset off to this output stream.

Subclasses of OutputStream implement the above mentioned methods.

1.27.2 Character Stream Classes

Java FileWriter and FileReader classes are used to write and read data from text files. These are character-oriented classes, used for file handling in java. Java Byte streams are used to perform input and output of 8-bit bytes; whereas Java **Character** streams are used to perform input and output for 16-bit Unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter.**

Example

```
import java.io.*;
public class FileReadWrite
{
   public static void main(String args[])throws IOException
   {
          FileWriter writer = new FileWriter("D:/hello.txt");
        writer.write("This\n is\n an\n example\n");
          writer.close();
          FileReader fr = new FileReader("D:/hello.txt");
        char [] a = new char[50];
        fr.read(a); //reads the content to the array
        for(char c : a)
          System.out.print(c); //prints the characters one by one
        fr.close();
   }
}
```

Output

This
is
an
example

1.27.3 Random Access File

RandomAccessFile class in java.IO API allows us to move back and forth in the file and we can read or write content in any required place of file. Before using RandomAccessFile class, we must instantiate it as below:

RandomAccessFile file=new RandomAccessFile(file-name,mode)

Once the class is instantiated we can move to the required location by calling seek() method and we can find the value of current position by calling getPointer() method.

Example

```
import java.io.*;
public class RAFDemo
{
   public static void main(String[] args)
   {
      try
      {
          // create a new RandomAccessFile with filename test
```

```
        RandomAccessFile raf = new RandomAccessFile("test.txt", "rw");

        // write something in the file
        raf.writeUTF("Hello World");
      raf.writeUTF("Hi");
        raf.seek(0);
        System.out.println(raf.readUTF());
      long pos=raf.getFilePointer();
      System.out.println("Current Position:" + pos);
              System.out.println(raf.readUTF());
        raf.seek(0);
        System.out.println(raf.readUTF());
      raf.seek(13);
        System.out.println(raf.readUTF());
      }
    catch (IOException ex)
    {
        ex.printStackTrace();
      }
  }
}
```

Output

```
Hello World
Current Position:13
Hi
Hello World
Hi
```

1.27.4 Reading and Writing Objects

Java object Serialization is an API that allows us to serialize Java objects. Serialization is a process to convert objects into a writable byte stream. Once converted into a byte-stream, these objects can be written to a file. The reverse process of this is called de-serialization. A Java object is Serializable if its class implement the java.io.Serializable interface. ObjectInputStream and ObjectOutputStream classes are used to read/write java objects.

Example

```
import java.io.*;
class Person implements Serializable
{
      private String name;
      private int age;
      private String gender;
```

```
    Person(String name, int age, String gender)
    {
       this.name = name;
       this.age = age;
       this.gender = gender;
    }
    @Override
    public String toString()
    {
    return "Name="+name+"\nAge="+age+"\nGender="+ gender;
    }
}
class RWObject
{
    public static void main(String[] args)
    {

       Person p1 = new Person("Ram", 30, "Male");
       Person p2 = new Person("Rita", 25, "Female");
       try
       {
          FileOutputStream fos = new FileOutputStream(new
File("myObjects.txt"));
          ObjectOutputStream oos = new ObjectOutputStream(fos);

          // Write objects to file
          oos.writeObject(p1);
          oos.writeObject(p2);
          oos.close();
          fos.close();
          FileInputStream fis = new FileInputStream(new
File("myObjects.txt"));
          ObjectInputStream ois = new ObjectInputStream(fis);

          // Read objects
          Person pr1 = (Person) ois.readObject();
          Person pr2 = (Person) ois.readObject();
```

```
            System.out.println(pr1.toString());
            System.out.println(pr2.toString());
            ois.close();
            fis.close();
        }
        catch (FileNotFoundException e)
        {
            System.out.println("File not found");
        }
        catch (Exception e)
        {
            System.out.println(e);
        }
    }
}
```

Output

Name=Ram

Age=30

Gender=Male

Name=Rita

Age=25

Gender=Female

EXERCISE

1. What is JVM? Explain architecture of java in detail along with suitable block diagram?
2. Explain the process of compiling and running and java program with sample example.
3. What is meant by PATH and CLASSPATH variable? Explain characteristics of java in brief.
4. What is array? How arrays in java are different? Explain different types of arrays with example.
5. What is meant by class and object? Explain object creation and member accessing mechanism with example.
6. Explain different access Specifiers used in java briefly with suitable example.

7. Explain the concepts behind static data members and static methods with example.
8. Write a program to model a cube having data member's length, breadth and height. U member functions of your own interest.
9. What is meant by overloading? Explain method overloading with suitable example.
10. What is dynamic polymorphism? Explain dynamic method dispatch with example.
11. How interfaces are different from class? Explain interface extension and implementatio with example.
12. What is package? Explain different ways of using packages. How can we crea packages? Explain.
13. What is constructor? Explain different types of constructors with suitable example.
14. How interfaces are different from abstract methods? Explain with suitable example.
15. Why multithreading is important? Explain thread life cycle with proper state diagram.
16. What are different ways of writing multithreaded programs? Explain with example each.
17. How exception is different from error? Explain at least five exception classes in brief.
18. What are different types of exceptions? Explain try....catch blocks with example.
19. Differentiate throws and throw keywords. When is finally block important? Explain with proper example.
20. What is meant by stream? Explain by stream and character stream classes with example
21. Write a program that read data of employees from keyboard and write it into the fi emp.doc
22. Why file handling is important? Explain any five classes used if file handling briefly.

□□□