Chapter-1 Basics

- Javafeatures……
- JDK,Bytecode…….
- JVM,Token……..
- Datatype……
- Operators, Type Casting…….
- Garbage Collection…….
- Looping……...
- Decision statement……

- Chapter-1 Basics

History

- ✓ Java was created by a team of programmers at sun Microsystems of U.S.A in 1991.
- ✓ This language was initially called "Oak" by James Gosling but renamed "Java" in 1995.
- ✓ Java was designed for the development of software for consumer electronics devices like TVs, VCRs, and such electronics machines.
- ✓ When the World Wide Web became popular in 1994, sun realized that Java was the perfect programming language for the Web.
- ✓ Late 1995 and early 1996 they released Java & it was instant success.

JDK Editions

- Java Standard Edition (J2SE)
  - J2SE can be used to develop client-side standalone applications or applets.
- Java Enterprise Edition (J2EE)
  - J2EE can be used to develop server-side applications such as Java servlets and Java ServerPages.
- Java Micro Edition (J2ME).
  - J2ME can be used to develop applications for mobile devices such as cell phones.

What is Java?

- ◆ Java is a programming language that:
  - ■ Is exclusively object oriented
  - ■ Has full GUI support
  - ■ Has full network support
  - ■ Is platform independent
  - ■ Executes stand-alone or "on-demand" in web browser as applets

The java Developer's Kit(JDK)

Before you can start writing java program, you need to acquire and setup some kind of java programming software . Although several different products are available for development of java programe, the starting place for many new java programmers is the JDK.

JDK Tools:

1. A Compiler:

Javac,which takes a java program file and translates it into a form

which the computer know how to run.

2. An Interpreter

Java ,runw program that are created by the compiler.

3. A Debugger

Java debugger can help you to find error in your program.

4. A Disassembler

If you run in to a compiled java program that you would like to see

In java source code form, the java disassembler will do the

translation for you

5. A Profiler:

Profiler will provide the handy information about your program.

Importance of JAVA

1. Java applet

- An applet is a special kind of java program that is designed to be transmitted over the internet and automatically executed by a java-compatible web browser.

- Furthermore, an applet is downloaded on demand, just like an image, sound file, or video clip.

- The important difference is that an applet is an intelligent program, not just an animation or media file.

- In other words, an applet is a program that can react to user input and dynamically change-not just run the same animation or sound over and over.

2. Security

- Every time you download a normal program, you are risking a viral infection.
- Prior to java, most users did not download executable programs frequently, and those who did scanned them for viruses prior to execution. Even so, most users still worried about the possibilities of infecting their systems with a virus.
- In addition to viruses, another type of malicious program exists that must be guarded against. This type of program can gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system.

- Java answers both of these concerns by providing a "firewall" between a networked application and your computer.
- When you use a java-compatible web browser, you can safely download java applets without fear of viral infection or malicious intent.
- Java achieves this protection by confining a java program to the java execution environment and not allowing it access to other parts of the computer.
- The ability to download applets with confidence that no harm will be done and that no security will be broken is considered by many to be the single most innovative aspect of java.

3. Portability
   Many types of computers and operating systems are In use throughout the world and many are connected to the internet. For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. As you will soon see, the same mechanism that helps ensure security also helps create portability. Indeed, java's solution to these two problems is both elegant and efficient.

Features of JAVA

1. Simple
2. Secure
3. Portable
4. Object-oriented
5. Robust
6. Multithreaded
7. Architecture-natural
8. Interpreted
9. High performance
10. Distributed
11. Dynamic

1. Simple

- Java was designed to be easy for the professional programmer to learn and use effectively.
- Assuming that you have some programming experience, you will not find java hard to master.
- If you already understand the basic concepts of object oriented programming, learning java will be even easier.

- If you are an experienced C++ programmer, moving to java will require very little effort. Because java inherits the C/C++ syntax and many of the object oriented features of C++, most programmers have little trouble learning Java.

2. Security

Security is the benefit of java. Java system not only verifies all memory access but also ensure that no viruses are communicated with an applet.

3. Portable

The most significant contribution of java over other language is its portability. Java programs can be easily moved from one computer system to another.

Java ensures portability in two ways:

1. Java compiler generates byte code instruction that can be implemented on any machine.
2. The size of the primitive data types is machine-independent.

4. Object-Oriented

- Java is a true object oriented language. All program code and data reside
  within object and classes.
- The object model in java is simple and easy to extend.

5. Robust(healthy, strong)

- The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of java.
- To gain reliability, java has strict compile time and run time checking for codes.
- To better understand how java is robust, consider two main reasons for program failure: memory management mistakes and mishandled exceptional conditions.

1. Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or try to free some memory that another part of their code is still using. Java virtually

eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because java provides garbage collection for unused objects.)

2. Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found" and they must be managed with awkward and hard to read constructs. Java helps in this area by providing object oriented exception handling. In a well-written java program, all run-time errors can and should be managed by your program.

6. Multithreaded

- Java was designed to meet the real-world requirement of creating interactive, networked programs.
- To accomplish this, java supports multithreaded programming, which allows you to write programs that do many things simultaneously.
- The java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems.
- Java's easy to use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

7. Architecture-Neutral

- A central issue of java programmers was that code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow- even on the same machine.
- Operating system upgrades, and changes in core system resources can al combine to make a program malfunction.
- The java designer made several hard decisions in the java language and the java virtual machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever."

8. Interpreted

- Usually a computer language is either compiled or interpreted. Java combines these approaches thus making java a two-stage system.

- Java compiler translates source code into byte code instructions. Byte codes are not machine instructions and so java interpreter generates machine code that can be directly executed by the machine that is running the java program.
- We can thus say that java is both a compiled and an interpreted language.

9. High Performance
   Java performance is impressive for an interpreted language, mainly due to the use of intermediate byte code.
10. Distributed

- Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols.
- Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

11. Dynamic

- Java is capable of dynamically linking in new class libraries, methods and object.
- Java can also determine the type of class through a query, making it possible to either dynamically link or abort the program.

<u>Object-Oriented Programming</u>

Object

- ✓ An object is a region of storage that defines both state & behavior.
  - State is represented by a set of variables & the values they contain.
  - Behavior is represented by a set of methods & the logic they implement.
- ✓ Thus, an object is a combination of a data & the code that acts upon it.
- ✓ Objects are instance of a class.
- ✓ Objects are the basic runtime entities in an object-oriented system.
- ✓ Object take up space in memory and have an associated address like a record in Pascal or structure in C.
- ✓ The arrangement of bits or the data in an object's memory space determines that object's state at given moment.
- ✓ Objects are runtime instance of some class.
  For Example:
  Person p1,p2;
  p1 = new person();

p2 = new person();

Class

- ✓ A class is a template from which objects are created. That is objects are instance of a class.
- ✓ When you create a class, you are creating a new data-type. You can use this type to declare objects of that type.
- ✓ Class defines structure and behavior (data & code) that will be shared by a set of objects

OOP Principles

- ✓ All object-oriented programming languages provide mechanism that help you implement the object-oriented model.
- ✓ They are:
  1. Encapsulation
  2. Inheritance
  3. Polymorphism

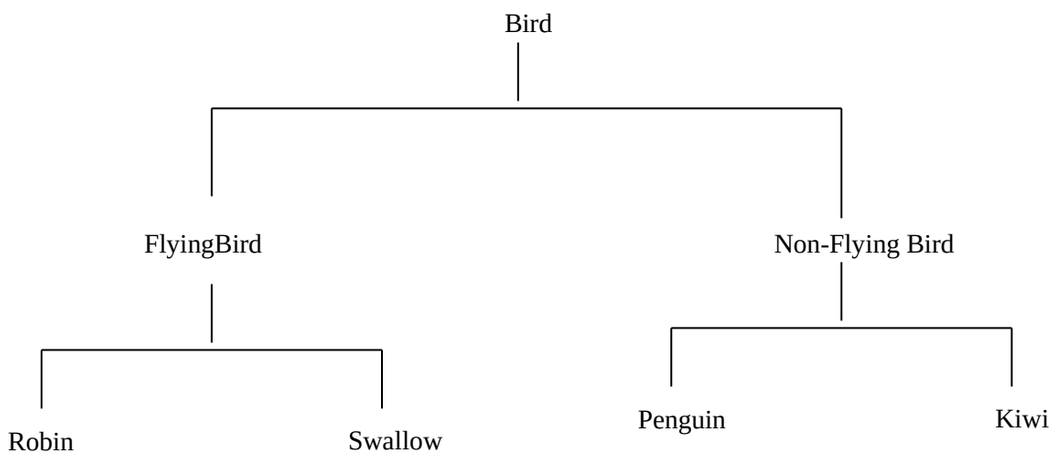
Encapsulation

- ✓ Encapsulation is the mechanism that binds code and data together and keeps both safe from outside interference and misuse.
- ✓ It works as a protective wrapper that prevents the code and data form being accessed by other code defined outside the wrapper.
- ✓ Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.
- ✓ In Java, the basis of encapsulation is the class.
- ✓ A class defines the structure and behavior that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class.
- ✓ For this reason, objects are sometimes referred to as instances of a class.
- ✓ When we create a class, we will specify the code and data that constitute that class. Collectively, these elements are called members of the class.
- ✓ The data defined by the class are referred to as member variables or instance variables.
- ✓ The code that operates on that data is referred to as member methods or just method.
- ✓ Each method or variable in a class may be marked private or public.
- ✓ The public interface of a class represents everything that external users of the class need to know, or may know.
- ✓ The private methods and data can only be accessed by code that is a member of the class. Therefore, any other

code that is not a member of the class cannot access a private method or variable.

Inheritance

- ✓ Inheritance is the process by which object of one class acquires the properties of another class.
- ✓ It supports the concept of hierarchical classification. For example, the bird robin is a part of the class flying Bird, which is again a part of the class Bird. Each derived class shares common characteristics with the class from it is derived.



- ✓ The concept of Inheritance provides the idea of reusability.
- ✓ This means that we can add additional features to an existing class without modifying it. This is possible by deriving new class form the existing one.
- ✓ The new class have the combined feature of both the classes.
- ✓ Each subclass defines only those features that are unique to it.
- ✓ Derived class is known as ‘sub class’ and main class is known as ‘super class’.

Polymorphism

- ✓ Polymorphism means the ability to take more than one form.
- ✓ A single function name can be used to handle different no and different types of argument.
- ✓ It plays an important role in allowing objects having different internal structures to share the same external interface.

Example

```
class Example
{
        public static void main(String args[])
        {
                System.out.println("First Example");
        }
}
```

example1

✓ class Example

Here name of the class is Example.

✓ public static void main(String args[])

public: The public keyword is an access specifier, which means that the content of the following block accessible from all other classes.

static: The keyword static allows main() to be called without having to instantiate a particular instance of a class.

void: The keyword void tells the compiler that main() does not return a value. The methods can return value.

main(): main is a method called when a java application begins,

- Java is case-sensitive
- Main is different from main
- Java compiler will compile classes that don't contain a main () method but java has no ways to run these classes.

✓ String args []

– Declares a parameter named args, which is an array of instance of the class string.
– Args[] receives any command-line argument present when the program is executed.

✓ System.out.println()

– System is predefined class that provides access to the system.
– Out is the output stream that is connected to the console.
– Output is accomplished by the built-in println() method. Println() displays the string which is passed to it.

Compilation

✓ Javac Example.java

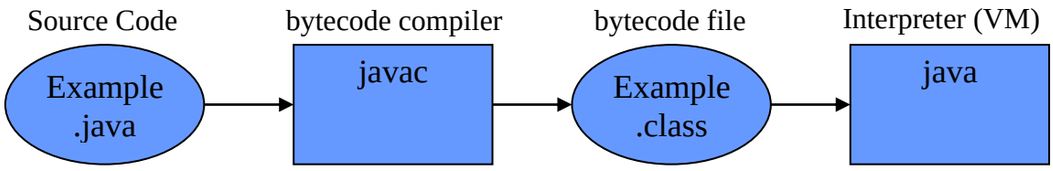
This command will compile the source file and if the compilation is successful it will generate a file named example.class containing bytecode. Java compilers translate java program to bytecode form.

✓ Java Example

The command called 'java' takes the bytecode and runs the bytecode on JVM environment in interpreted mode.

✓ Output

First Example



✓ When java source code is compiled, each individual class is put into its own output file named after the class and using the .class extension. This is why it is a good idea to give your java source files the same name as the class they contain – the name of the source file will match the name of the .class file.

✓ When you execute java program, you are actually specifying he name of the class that you want to execute. It will automatically search for a file by that name that has the .class extension. If it finds the file, it will execute the code contained in the specified class.

Java Virtual Machine

✓ All language compilers translate source code into machine code for a specific computer. Java compiler also does the same thing.

✓ Java compiler produces an intermediate code known as byte code for a machine that does not exist.

✓ This machine is called the Java Virtual Machine and it exists only inside the computer memory.

✓ The virtual machine code (Byte Code) is not machine specific.

✓ The machine specific code is generated by the Java Interpreter by acting as an intermediary between the virtual machine and the real machine as shown in fig. Interpreter is different for different machine.

Lexical Issues

- ✓ Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators and keywords.

1. Whitespace
   - ✓ Java is a free-form language. This means that you do not need to follow any special indentation rules.
   - ✓ For Example, the example program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already defined by an operator or separator.
   - ✓ In java, whitespace is a space, tab or a new line.
2. Identifiers
   - ✓ Identifiers are used for class names, method names and variable names.
   - ✓ An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers or the underscore and dollar-sign character.
   - ✓ hey must not begin with a number for fear that they be confused with a numeric literal.
   - ✓ Java is case-sensitive, so VALUE is a different identifier than value.
   - ✓ Some examples of valid identifiers are:
     - AvgTemp, Count, an, $test, this_is_ok
   - ✓ Invalid variable names include here;
     - 2count, high-temp, Not/ok
3. Literals
   - ✓ Literals in java are a sequence of characters (digits, letters, and other characters) that represent constant values to be stored in variables.
   - ✓ Java language specifies five major types of literals they are:
     - Integer Literals
     - Floating point Literals
     - Character Literals
     - String Literals
     - Boolean Literals

4. Comments
   - ✓ There are three types of comments defined by java,
     - Single line: - /_____ /
     - Multi line: - /* _____ */
     - Documentation comment: - /** _____ */

5. Separators
   - ✓ Separators are symbols used to indicate where groups of code are divided and arranged.
   - ✓ They basically define the shape and function of our code.

   1.Parentheses ()
   - o Parentheses are used to contain lists of parameters in method definition and invocation.
   - o Parentheses are used for defining precedence in expressions, containing expressions in control statements and surrounding cast types.
   - o Separators are symbols used to indicate where groups of code are divided and arranged.
   - o They basically define the shape and function of our code.

   2. Braces{}
      1. Braces are used to contain the values of automatically initialized arrays.
      2. Braces are used to define block of code for classes, methods and local scopes
   - o Separators are symbols used to indicate where groups of code are divided and arranged.
   - o They basically define the shape and function of our code.

   3. Brackets []
   - ✓ Separators are symbols used to indicate where groups of code are divided and arranged.
   - ✓ They basically define the shape and function of our code.
   - ✓ Parentheses ()
   - ✓ Braces {}
   - ✓ Brackets []
      - ✓ Brackets are used to declare array types, also used when dereferencing array values.

   - ✓ Separators are symbols used to indicate where groups of code are divided and arranged.
   - ✓ They basically define the shape and function of our code.
   - ✓ Parentheses ()
   - ✓ Braces {}
   - ✓ Brackets []
   - ✓ Semicolon ;
      - ✓ Semicolon is used for terminates statements.

- ✓ Separators are symbols used to indicate where groups of code are divided and arranged.
- ✓ They basically define the shape and function of our code.
- ✓ Parentheses ()
- ✓ Braces {}
- ✓ Brackets []
- ✓ Semicolon ;
- ✓ Comma ,
    - ✓ Separates consecutive identifiers in a variable declaration, also used to chain statements together inside a for statement.

- ✓ Separators are symbols used to indicate where groups of code are divided and arranged.
- ✓ They basically define the shape and function of our code.
- ✓ Parentheses ()
- ✓ Braces {}
- ✓ Brackets []
- ✓ Semicolon ;
- ✓ Comma ,
- ✓ Period .
    - ✓ Period is used to separate package names from sub packages and classes, also used to separate a variable or method from a reference variable.

Java Keywords

- ✓ There are 50 reserved keywords currently defined in java language.
- ✓ These keywords, combined with the syntax of the operators and separators, from the foundation of the java language.
- ✓ Keywords have specific meaning in java; we cannot use them as names for variables, classes, methods and so on.

Java Class Library

- ✓ We are using println(), print() methods which are members of the System class, which is a class predefined by Java that is automatically included in your programs.
- ✓ In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics.

- ✓ The standard class also provides support for windowed output. Thus, java is a combination of the java language itself, plus its standard classes. The class library provides provide much of the functionality that comes with java.

Data Types

- ✓ Every variable has a type, every expression has a type, and every type is strictly defined.
- ✓ All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- ✓ There are no automatic coercions or conversions of conflicting types as in some languages.
- ✓ The java compiler checks all expressions and parameters to ensure that the types are compatible.
- ✓ There are two types of data types
  - ✓ Primitive types
  - ✓ Non primitive types

Primitive Types

- ✓ Java provides eight primitive types of data:
  1. Byte
  2. Short
  3. Int
  4. Long
  5. Char
  6. Float
  7. Double
  8. Boolean
- ✓ The primitive types are also commonly referred to as simple types.
- ✓ These can be put in four groups:
  1. Integer
  2. Floating-point numbers
  3. Characters
  4. Boolean

Integer

- ✓ Java provides four integer types: byte, short, int, long.
- ✓ All of these are signed, positive and negative values.

Type	Size/bytes	Range
Byte	8	-128 to 127
Short	16	-32,768 to 32,767
Int	32	-2,147,483,648 to 2,147,483,647
Long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Floating-Point Types

- ✓ Floating-Point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision.
- ✓ For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type.
- ✓ There are two kinds of floating-point types, float and double, which represent single and double-precision numbers, respectively.

Type	Size/bytes	Range
Float	32	1.4e – 045 to 3.4e + 038
Double	64	4.9e - 324 to 1.8e + 308

Character

- ✓ The data type used to store characters is char.

Type	Size/bytes	Range
Char	16	0 to 65,536

Boolean

- ✓ Java has primitive type, called boolean, for logical values.
- ✓ It can have only one of two possible values, true or false.

Type	Size/bytes	Range
Boolean	1	True/False , Yes/No , 0/1

Variable

- ✓ The variable is the basic unit of storage in a java program.
- ✓ A variable is defined by the combination of identifiers, a type and an optional initialize.
- ✓ All variables have a scope, which defines their visibility and a life time.

  Declaring a Variable: -

  All variables must be declared before they can be used.

  The basic form of a variable declaration is shown here.

  Type identifier [= value] [, identifier [=value]...];

  Int a,b,c; // declare 3 integers

  Byte z = 22; // initialize z

```
Char x = 'X';      // the variable x has the value 'X'
```

Dynamic Initialization:

- ✓ Java allows variables to be initialized dynamically using any valid expression at the time the variable is declared.

Example:

```
class  DynamicInt
{
        public static void main(String args[])
        {
                double a= 3.0, b= 5.0;
                // c is dynamically initialized
                double c = Math.sqrt (a * a+b * b);
                System.out.println("The value of C is: - " + c);
        }
}
```

In above example method sqrt (), is the member of the Math class.

The Scope & Lifetime of Variable

- ✓ All the variables used have been declared at the start of the main () method. Java allows variables to be declared within any block.
- ✓ A block defines a scope. Thus, each time you start a new block, you are creating a new scope.
- ✓ A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.
- ✓ Many other computer languages define two general categories of scopes: Global and Local. In java, the two major scopes are those defined by a class and those defined by method.

- ✓ As a general rule, variables declared inside a scope are not visible (accessible) to code that is defined outside that scope.
- ✓ Thus, when you declare a variable within a scope you are localizing that variable and the scope rules provide the foundation for encapsulation.
- ✓ Scopes can be nested. For example, each time you create a block of code, you are creating a new rested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.
- ✓ To understand the effect of nested scopes consider the following program.

```
class Scope
{
	public static void main(String args[])
	{
		int x;
		x = 10;   // known to all code within main
		if (x == 10)
		{
			int y =20;  // known to only this block
			system.out.println ("x and y: - "+ x+" "+y);
			x = y*2;
		}
		y = 100;  // error y is not known here
		system.out.println ("x is "+x);
	}
}
```

- ✓ Variables are created when their scope is entered and destroyed when their scope is left.
- ✓ This means that a variable will not hold its value once it has gone out of scope.
- ✓ Therefore, variables declared within a method will not hold their values between calls to that method.
- ✓ A variable declared within a block will lese its value when the block is left. Thus, the lifetime of a variable is confined to its scope.
- ✓ If a variable declaration includes an initializer then that variable will be reinitialized each time the block in which it is declared is entered
- ✓ For example, consider the next program.

```
// demonstrate lifetime of a variable
class LifeTime
{
	public static void main (String args[])
	{
		int x;
		for (x=0 ; x< 3 ; x++)
		{    //y is initialized each time block is entered.
			int y = -1;
			system.out.println ("Y is: - "+y);
			y = 100;
			system.out.println ("Y is now: -"+ y);
		}
	}
}
```

Operators

- ✓ An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.
- ✓ Operators are used in programs to manipulate data and variables.
- ✓ Java operators can be classified into a number of related categories as below:
  1. Arithmetic operator
  2. Relational operator
  3. Logical operator
  4. Assignment operators
  5. Increment and decrement operator
  6. Conditional operators
  7. Bitwise operators
  8. Special operators

Arithmetic operator

- ✓ Java provides all the basic arithmetic operators.

Operator	Meaning
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

1. Integer Arithmetic

- ✓ When both the operands in a single arithmetic expression such as a+b are integers, the expression is called an integer expression, and the operation is called integer arithmetic.
- ✓ Integer arithmetic always yields an integer value.
- ✓ If we have a=14 and b=4 then
  a+b=18
  a-b=10
  a*b=56
  a/b=3 (decimal part truncated)
  a%b=2 (remainder of integer division)
- ✓ For modulo division, the sign of the result is always the sign of the first operand.
  -14%3 = -2
  -14%-3 = -2
  14%3 = 2

2. Real Arithmetic

- ✓ An arithmetic operation involving only real operands is called real arithmetic.
- ✓ A real operand may assume values either in decimal or exponential notation.
- ✓ Modulus operator % can be applied to the floating point data as well.

```
class FloatArith
{
	public static void main(String args[])
	{
		float a=20.5f,b=6.4f;
		System.out.println("a="+a);
		System.out.println("b="+b);
		System.out.println("a+b="+(a+b));
		System.out.println("a-b="+(a-b));
		System.out.println("a*b="+(a*b));
		System.out.println("a/b="+(a/b));
		System.out.println("a%b="+(a%b));
	}
}
```

3. Mixed-mode Arithmetic

- ✓ When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression.
- ✓ If either operand is of the real type, then the other operand is converted to real and the real arithmetic is performed. The result will be a real.

15/10.0 = 1.5
15/10 = 1

Relational Operator

- ✓ For comparing two quantities, and depending on their relation, we take certain decision.
- ✓ For example, we may compare the age of two persons, or the price of two items, and so on. These comparison can be done with the help of relational operators.

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

```
class Relational
{
	public static void main(String args[])
	{
		float a=15.0f,b=20.75f,c=15.0f;
```

```
            System.out.println("a="+a);
            System.out.println("b="+b);
            System.out.println("c="+c);
            System.out.println("a<b is "+(a<b));
            System.out.println("a>b is "+(a>b));
            System.out.println("a==c is "+(a==c));
            System.out.println("a<=c is "+(a<=c));
            System.out.println("a>=b is "+(a>=b));
            System.out.println("b!=c is "+(b!=c));
            System.out.println("b==a+c is "+(b==a+c));
      }
}
```

Logical Operator

- ✓ Java has three logical operators:

Operator	Meaning
&&	logical AND
||	logical OR
!	Logical NOT

```
class Logic
{
      public static void main(String args[])
      {
            int a=15,b=30,c=15,d=30;
            if (a==c && b==d)

                  System.out.println("Equal");
            else
                  System.out.println("Not Equal");
            if (a<b || a<c)
                  System.out.println("a is smaller than any other
values");
            else
                  System.out.println("a is greater than any other
values");
            if(a!=b)
                  System.out.println("the value of a and b are not
equal");
            else
                  System.out.println("The value of a and b are
equal");

      }
}
```

Assignment Operator

- ✓ Assignment operators are used to assign the value of an expression to a variable.

- ✓ Java has a set of 'shorthand' assignment operators which are used in the form

  var-name op= exp;

  which is equivalent to

  var-name=var-name op (exp);
- ✓ For the statement:

  x=x+(y+1);

  x+=y+1;

Statement with Simple assignment operators	Statement with Shorthand operator
a=a+1	a+=1
a=a*1	a*=1
a=a*(n+1)	a*=n+1
a=a/(n+1)	a/=n+1
a=a%b	a%=b

Increment and Decrement Operator

- ✓ java has two increment and decrement operators:

  ++ and - -
- ✓ The operator ++ adds 1 to the operand while - - subtracts 1. both are used in the following format:

  ++m; or m++;

  --m or m--;

Where:

++m is equivalent to m=m+1;

-- m is equivalent to m=m-1;

```
class Increment
{
    public static void main(String args[])
    {
        int m=10, n=20;
        System.out.println("m=" + m);
        System.out.println("n=" + n);
        System.out.println("++m=" + ++m);
        System.out.println("n++" + n++);
        System.out.println("m=" + m);
        System.out.println("n=" + n);
    }
}
```

Conditional Operator

- ✓ The character pair ? : is a ternary operator available in java. This operator is used to construct conditional expressions of the form

Exp1 ? Exp2: Exp3

- ✓ Consider the following example:

  a=10;

```
        b=15;
        x=(a>b) ? a: b;
```

It is same as:

```
        if (a>b)
                x=a;
        else
                x=b;
```

```
import java.util.Random;
class Condition
{
        public static void main(String args[])
        {
                Random random= new Random();
                float x= random.nextFloat();
                System.out.println("x=" +x);
                float y= random.nextFloat();
                System.out.println("y=" +y);
                float min=(x<y ? x : y);
                float max=(x>y ? x : y);
                System.out.println("min="+min);
                System.out.println("max="+max);
        }
}
```

Bitwise Operators

- ✓ Java has a distinction of supporting special operators known as bitwise operators for manipulation of data at values of bit level.
- ✓ These operators are used for testing the bits, or shifting them to the right or left.
- ✓ Bitwise operators may not be applied to float or double.

Operator	Meaning
&	bitwise AND
!	Bitwise OR
^	Bitwise exclusive OR
~	one's complement
<<	shift left
>>	shift right

- AND
  - 0101 AND 0011 = 0001
- OR
  - 0101 OR 0011 = 0111
- XOR
  - 0101 XOR 0011 = 0110

- NOT
  - NOT 0111 = 1000
- 0111 LEFT-SHIFT = 1110
- 0111 RIGHT-SHIFT = 0011

Logical Bitwise Operations

bit 1	bit 2	OR (|)	AND (&)	XOR (^)
0	0	0	0	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	0

- x = 12;
  y = 10;
  z = x & y; // z is 8
- It works like
  0000 1100
  & 0000 1010

  0000 1000 = 8 (decimal)

Special Operator

- ✓ Java supports some special operators of interest such as instanceof operator and member selection operator(.)
- ✓ Instanceof Operator
  - ✓ the instanceof is an object reference operator and returns true if the object on the left-hand side is an instance of the class given on the right-hand side. This operator allows us to determine whether the object belongs to a particular class or not.
  - ✓ For example:

  person instanceof student
  is true if the object person belongs to the class student; otherwise it is false.

- ✓ Dot operator
- ✓ The dot operator(.) is used to access the instance variables and methods of class objects.
- ✓ For example:

  person1.age //reference to the variable age
  person1.salary() //reference to the method salary()
- ✓ It is used to access classes and sub-packages form a package.

Arrays

- ✓ An array is a group of like-typed variables that are referred by a common name.
- ✓ Arrays of any type can be created and may have one or more dimensions.
- ✓ A specific element in an array is accessed by its index.
- ✓ Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

- ✓ A one-dimensional array is a list of like typed variables.
- ✓ To create an array, you first must create an array variable of the desired type.
- ✓ The general form of a one-dimensional array declaration is:

```
type var-name[ ];
int month_days[ ];
```

- ✓ The value of month_days is set to null, which represents an array with no value. To link month_days with an actual, physical array of integers, you must first allocate one using new and assign it to month_days.
- ✓ new is special character which allocates memory.

The general form of new as it applies to one-dimensional arrays appears as follows:

```
array-var=newtype[size];
```

Following example allocates a 12-element array of integers and line them to month_days.

```
month_days=newint[12];
```

after this statement executes, month_days will refer to an array of 12 integers.

In short, obtaining an array is a two step process.
1. you must declare a variable of the desired array type.
2. you must allocate the memory that will hold the array, using new, and assign it to the array variable.

```
class Array
{
    public static void main(String args[])
    {
        int month_days[ ];
        month_days= new int[12];
```

```
            month_days[0]=31;
            month_days[1]=28;
            month_days[2]=31
            month_days[3]=30
            month_days[4]=31
            month_days[5]=30
            month_days[6]=31
            month_days[7]=31;
            month_days[8]=30;
            month_days[9]=31;
            month_days[10]=30;
            month_days[11]=31;
            System.out.println ("April has" + month_days[3] +
"days.");
      }
}
```

- ✓ it is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

  int month_days[ ]=new int[12];

- ✓ Arrays can be initialized when they are declared.
- ✓ An array initializer is a list of comma-separated expressions surrounded by curly braces. The comma separates the values of the array elements.
- ✓ The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use new.

For example,

```
      Class AutoArray
{
      public static void main (String args[ ])
      {
            int month_days[]={31,28,31,30,31,30,31,31,
            30,31,30,31};
            System.out.println("April has" + month_days[3]   +
"days.");
      }
}
```

Multidimensional Arrays

- ✓ Multidimensional arrays are actually arrays of arrays.

- ✓ To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- ✓ For example, following declares two dimensional array:

```
int twoD[ ] [ ] = new int [4] [5];
```

- ✓ This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int.

Alternative Array Declaration

- ✓ There is a second form that may be used to declare an array:

```
type[ ] var-name;
```

- ✓ The square brackets follow the type specifier, and not the name of the array variable.
- ✓ For example, the following two declarations are equivalent:

```
int a1[ ] = new int[4];
int [ ] a1= new int[4];
char twod [ ] [ ] = new char [3] [4];
char [ ] [ ] twod = new char [3] [4];
```

- ✓ This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

```
int [ ] nums1, nums2, nums3;
```

- ✓ This creates 3 array variables of int type.

Type Conversion & Casting

- ✓ It is common to assign a value of one type to a variable of another type.
- ✓ If the two types are compatible then java will perform the conversion automatically. For example it is possible to assign an integer value to a long variable.
- ✓ However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For example there is no conversion defined from double to byte
- ✓ It is still possible to obtain a conversion between incompatible types.
- ✓ For that you must use a cast, which performs an explicit conversion between incompatible types.

Java's Automatic Type Conversion

- ✓ When one type of data is assign to another type of variable, an automatic type conversion will take place if the following two conditions are met:
  - The Two types are compatible.

- The destination type is larger than the source type.

- ✓ The int type is always large to hold all valid byte values. So no explicit cast statement is required.
- ✓ For widening conversions, the numeric types, including integer and floating-point types are not compatible with each other.
- ✓ However, numeric types are not compatible with char or Boolean. Char and Boolean data types are not compatible with each other.
- ✓ Java performs an automatic type conversion when storing a literal integer constant into variables of type byte, short or long.

Casting Incompatible Types

- ✓ Automatic type conversion will not fulfill all needs. For example if you want to assign an int value to a byte variable?
- ✓ This conversion will not be performed automatically, because a byte is smaller than int.
- ✓ This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.
- ✓ To create conversion between two incompatible types, you must use a cast.

- ✓ A cast is an explicit type conversion. It has this general form:

(target-type) value

- ✓ Here target-type specifies the desired type to convert the specified value to:
- ✓ For example, The following fragment casts an int to a byte. If the integer's value is larger then the range of byte, it will be reduced modules (the reminder of an integer division by the) byte's range.

```
int a;
byte b;
b = (byte) a;
```

- ✓ A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation.
- ✓ Integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.
- ✓ For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated.
- ✓ If the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

```
// demonstrate casts
```

```
class conversion
{
        public static void main (String args[])
        {
                byte b;
                int i = 257;
                double d = 323.142;
                System.out.println (“Conversion of int to byte: -”);
                b = (byte) i;
                System.out.println (“i and b: -“+i+” “+b);
                System.out.println (“\n Conversion of double to int: -”);
                i = (int) d;
                System.out.println (“d and i: -“+d+” “+i);
                System.out.println (“\n Conversion of double to byte: -
”);
                b = (byte) d;
                System.out.println (“d and b: -“+d+” “+b);
        }
}
```

Garbage Collection

- ✓ Objects are dynamically allocated by using the new operator.
- ✓ In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator.
- ✓ Java handles deallocation for you automatically. The technique that accomplishes this is called garbage collection.
- ✓ when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- ✓ Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because java provides garbage collection for unused objects.)

Finalize() Method

- ✓ Sometimes an object will need to perform some action when it is destroyed.
- ✓ For Ex, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed.
- ✓ To handle such situations, java provides a mechanism called finalization.
- ✓ By using finalization, we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- ✓ To add a finalizer to a class, you simply define the finalize() method. The java run time calls that method whenever it is about to recycle an object of that class.

```
Protected void finalize ()
{
        // finalize code
}
```

- ✓ It is important to understand that finalize() is only called just prior to garbage collection.
- ✓ It is not called when an object goes out-of-scope, for ex, .
- ✓ This means that you cannot know when- or even if- finalize() will be executed. therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on finalize() for normal program operation.

Method Overloading:

Method overloading enables you to specify different types of information (Parameter) to send to a method, To overload a method you declare another version with the same name but different parameter.

Example:

```
class A
{
        void display()
        {
                System.out.println("display() called");
        }
        void display(int i)
        {
                System.out.println("display(int) called");
        }
        void display(int i,int j)
```

```
	{
		System.out.println("display(int,int) called");
	}
	void display(double i)
	{
		System.out.println("display(double) called");
	}
	void display(double i,int j,int k)
	{
		System.out.println("display(double,int,int) called");
	}
}
class OverloadDemo
{
	public static void main(String args[])
	{
		A a=new A();
		a.display();
		a.display(5);
		a.display(5,10);
		a.display(10.0);
		a.display(5,5,5);

	}
}
```

Method Overriding:

In a class hierarchy, when a method in a subclass has the same name and type, then the method in the subclass is said to override the method in the
Super class. When a overridden method is called from within a subclass, it will always refer to the version of that method defined by the sub class. The version of the method defined by the super class will be hidden.

Example:

```
class A
{
	int i,j;
	A(int a, int b)
	{
		i=a;
		j=b;
	}
```

```
        void show()
        {
                System.out.println("I and j:" +i +" " +j);
        }
}
class B extends A
{
        int k;
        B(int a, int b, int c)
        {
                super(a,b);
                k=c;
        }
        void show()
        {
                System.out.println("k="+k);
        }
}


class Override
{
        public static void main(String args[])
        {
                B subob= new B(1,2,3);
                subob.show();
        }
}
```

Chapter-2 ClassFundas

- Class and Object……
- Constructor…….
- Overloading…….
- Overriding ……..
- Static…….
- Final……..
- Abstract…….....
- Inheritance ……
- All access specifiers……

Chapter-2 ClassFundas

Introduction of Classes, objects and methods

The General Form of a class

- ✓ A class is a template from which objects are created. That is objects are instance of a class.
- ✓ When you create a class, you are creating a new data-type. You can use this type to declare objects of that type.
- ✓ Class defines structure and behavior (data & code) that will be shared by a set of objects
- ✓ A class is declared by use of the class keyword. Classes may contain data and code both.
- ✓ The general form of al class definition:

```
class ClassName
    {
        type instance variable1;
        type instance variable2;

        type methodname1 (parameter list)
        {
            body of method;
        }
        type methodname2 (parameter list)
        {
            body of method;
        }
    }
```

- ✓ The data or variable are called instance variable. The code is contained within methods.
- ✓ The method and variables defined within a class are called member of the class.

```
class Box
{
    double width;
```

```
        double height;
        double depth;
}
class BoxDemo
{
        public static void main (String args[])
        {
                Box mybox = new Box ();
                double vol;
                mybox.width =10;
                mybox.height = 20;
                mybox.depth = 30;
                vol = mybox.width * mybox.height * mybox.depth;
                System.out.println ("Volume is: - "+vol);
        }
}
```

- ✓ Each object contains its own copy of each variable defined by the class.
- ✓ So, every Box object contains its own copy of the instance variables width, height and depth.
- ✓ To access these variables, you will use the dot (.) operator. The dot operator links the name of the object with the name of an instance variable.

Declaring Object

- Box mybox; // declare ref. to object which contains null value.
- mybox = new Box (); // allocate a Box object.

General form of a new

```
class var = new classname ();
mybox = new Box ();
```

Where:

class var = variable of the class type.
classname = name of the class.

- ✓ The classname followed by parentheses specifies the constructor for the class.
- ✓ A constructor defines what occurs when an object of a class is created.
- ✓ Most classes explicitly define their own constructors within their class definition but if no explicit constructor is specified then java will automatically supply a default constructor.
- ✓ This is the case with Box. This is default constructor.

Assigning Object Reference Variable

```
Box b1 = new Box ();
```

Box b2 = new b1;

- ✓ After this executes, b1 and b2 will both refer to the same object.
- ✓ The assignment of b1 to b2 did not allocate any memory or copy any part of the original object.
- ✓ It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

Introducing Methods

```
type name (parameter-list)
{
        body of method
}
```

Where:
Type : Specifies the type of data returned by the method. If the method does not return a value its return type must be void.
Name: Specifies the name of the method.
Parameter-list: It is a sequence of type & identifiers pairs separated by commas.

- ✓ Parameters are variables that receive the value of the argument passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.
- ✓ Methods that have a return type other than void return a value to the calling routine using the following form of the return statement;

Return value;

Types of Methods

1. Does not return value – void
2. Returning a value
3. Method which takes parameter

Does not return value – void

```
class Box
{
        double width, height, depth;
        void volume()
        {
                System.out.println("Volume is: -
"+width*height*depth);
        }
}
class BoxDemo
```

```
{
	public static void main (String args[])
	{
		Box mybox1 = new Box ();
		Box mybox2 = new Box ();

		mybox1.width =10;
		mybox1.height =20;
		mybox1.depth =15;
		mybox2.width =10;
		mybox2.height =15;
		mybox2.depth =25;
		mybox1.volume ();
		mybox2.volume ();
	}
}
```

Returning a value

```
class Box
{
	double width, height, depth;
	void volume()
	{
		return width*height*depth;
	}
}
class BoxDemo
{
	public static void main (String args[])
	{
		Box mybox1 = new Box ();
		Box mybox2 = new Box ();
		double vol;

		mybox1.width =10;
		mybox1.height =15;
		mybox1.depth =20;
		mybox2.width =2;
		mybox2.height =3;
		mybox2.depth =5;
		vol = mybox1.volume ();
		System.out.println (“Volume is: -“+vol);

		vol = mybox2.volume ();
		System.out.println (“Volume is: -“+vol);
	}
}
```

Method which takes parameter

```
int square ()
{
        return 10 * 10;
}
```

- ✓ It will return the square of 10 but this method is specified to only 10. If you modify the method so that it takes a parameter.

```
int square (int i)
        {
                return i * i;
        }
        int x;
        x = square (5);
```

- ✓ A parameter is a variable defined by a method that receives a value when the method is called. For example in square (), i is a parameter.
- ✓ An argument is value that is passed to a method when it is invoked. For example in square (100), passes 100 as an argument.

```
class Box
{
        double width, height, depth;
        double volume()
        {
                return width*height*depth;
        }
        void setDim (double w, double h, double d)
        {
                width = w;
                height = h;
                depth = d;
        }
}
```

```
class BoxDemo
{
        public static void main (String args[])
        {
                double vol;
                Box mybox1 = new Box ();
                Box mybox2 = new Box ();
```

```
            mybox1.setDim (10,15,25);
            mybox2.setDim (3,5,7);
            vol = mybox1.volume ();
            System.out.println ("Volume is: -"+vol);

            vol = mybox2.volume ();
            System.out.println ("Volume is: -"+vol);
        }
}
```

Constructor

- ✓ It can be tedious to initialize the entire variable in a class each time an instance is created.
- ✓ Even when you add functions like setDim (), it would be simpler and more concise to have all of the setup done at the time the object is first created.
- ✓ Because the requirement for initialization is so common, java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of constructor.
- ✓ A constructor initializes an object immediately upon creation.
- ✓ It has the same name as the class in which it resides and is syntactically similar to a method.
- ✓ Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.
- ✓ Constructor do not have return type not even void.

```
class Box
{
        double width, height, depth;
        Box ()
        {
            width = 10;
            height = 10;
            depth = 10;
        }
double volume()
        {
            return width*height*depth;
        }

}
class BoxDemo{
        public static void main (String args[])
        {
            double vol;
```

```
        Box mybox1 = new Box ();
        Box mybox2 = new Box ();

        vol = mybox1.volume ();
        System.out.println ("Volume is: -"+vol);

        vol = mybox2.volume ();
        System.out.println ("Volume is: -"+vol);
}
}
```

- ✓ mybox1 and mybox2 were initialized by the Box () constructor when they were created.
- ✓ Both will print the same value 1000.
  Box mybox1 = new Box ();
- ✓ Constructor for the class is being called. New Box() is calling the Box() constructor.

Parameterized Constructor

```
class Box
{
    double width, height, depth;
    Box (double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
    double volume()
    {
        return width*height*depth;
    }
}
class BoxDemo
{
    public static void main (String args[])
    {
        double vol;
        Box mybox1 = new Box (10,20,15);
        Box mybox2 = new Box (3,5,7);

        vol = mybox1.volume ();
        System.out.println ("Volume is: -"+vol);

        vol = mybox2.volume ();
        System.out.println ("Volume is: -"+vol);
}
}
```

this Keyword

- ✓ Sometimes a method will need to refer to the object that invoked it. To allow this, java defines this keyword.
- ✓ this can be used inside any method to refer to the current object.
- ✓ this is always a ref. to the object on which the method was invoked.
- ✓ Consider the following Example:

```
//redundant use of this
Box(double w, double h, double d){
        this.width=w;
        this.height=h;
        this.depth=d;
}
```

Instance Variable Hiding

- ✓ It is illegal in java to declare two local variables with the same name inside the same or enclosing scopes.
- ✓ We have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables.
- ✓ However, when a local variable has the same name as an instance variable, the local variable hides the instance variable.
- ✓ This is why width, height and depth were not used as the names of the parameters to the Box() constructor inside the box class.
- ✓ If they had been, them width would have referred to the formal parameter, hiding the instance variable width. While it is easier to use different names.

- ✓ this lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables.
- ✓ For example, here is another version of Box(), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name.

```
// use this to resolve name-space collisions.
Box(double width, double height, double depth)
{
        this.width=width;
        this.height=height;
        this.depth=depth;
}
```

Static

- ✓ Sometimes you want to define a class member that will be used independently without (of) any object of that class.
- ✓ Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- ✓ To create such a member, precede its declaration with the keyword static.
- ✓ When a member is declared static, it can be accessed before any objects of its class are created, and without reference to an object.
- ✓ You can declare both methods and variables to be static.
- ✓ The most common example of a static member is main().main() is declared as static because it must be called before any object exist.
- ✓ Instance variables declared as static are actually, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.
- ✓ Method declared as static have several restrictions:
  - ✓ They can only call other static methods.
  - ✓ They must only access static data.
  - ✓ They can not refer to this or super in any way.
- ✓ If you need to do computation in order to initialize your static variables, you can declare a static block which gets executed exactly once, when the class is first loaded.
- ✓ The following example shows a class that has a static method, static variables and static initialization block:
- ✓ As the UseStatic class is loaded, all of the static statements are run. first a is set to 3 then the static block executes and finally, b is initialized to a*4 or 12. then main() is called which calls metho(), passing 42 to x. the 3 println() statements refer to the two static variables a and b as well as to the local variable x.

```
class UseStatic
{
        static int a=3;
        static int b;
        static void meth(int x)
        {
                System.out.println("x=" +x);
                System.out.println("a=" +a);
```

```
        System.out.println("b=" +b);
    }
    static
    {
        System.out.println("Static block initialized");
        b=a*4;
    }
    public static void main(String args[])
    {
        meth(42);
    }
}
```

- ✓ Static block initialized
  X=42
  A=3
  B=12
- ✓ Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need to specify only name of their class followed by the dot operator.
- ✓ For Ex, if you wish to call a static method from outside its class, you can do so using the following:

Classname.method()

- ✓ Class name is the name of the class in which static method is declared. A static variable and method can be accessed in same way by use of the dot operator on the name of the class.

```
class StaticDemo
{
    static int a=42;
    static int b=99;
    static void callme()
    {
        System.out.println("a=" +a);
    }
}
class StaticByName
{
    public static void main(String args[])
    {
        StaticDemo.callme();
        System.out.println("b=" +StaticDemo.b);
    }
}
```

Output:
A=42
B=99

Final

- ✓ A variable can be declared as final.
- ✓ By writing final it prevents its contents from being modified. This means that you must initialize a final variable when it is declared.

For example:

- Final int FILE_NEW =1;
- Final int FILE_OPEN=2;

- ✓ Subsequent part of program can use FILE_OPEN, as if they were constants, without fear that a value has been changed.
- ✓ It is a common coding convention to choose all uppercase identifiers for final variables.
- ✓ Variables declared as final do not occupy memory on a per-instance basis. Thus, a final variable is essentially a constant.
- ✓ The keyword final can be applied to method, but its meaning is different than when it is applied to variables.

Using final with Inheritance

- ✓ The keyword final has 3 uses. First, it can be used to create the equivalent of a named constant. The other two uses of final apply to inheritance.
  1. Using final to Prevent Overriding
  2. Using final to prevent Inheritance

Using final to Prevent Overriding

- ✓ While method overriding is one of java's most powerful features, there will be times when you will want to prevent it from occurring.
- ✓ To disallow a method form being overridden, specify final as a modifier at the start of its declaration.
- ✓ Methods declared as final cannot be overridden.

```
Class A
{
    Final void meth()
    {
        System.out.println("this is a final method");
    }
}
Class B extends A
{
    Void meth()
    {
        // error cannot override
        System.out.println("Illegal");
    }
}
```

Because meth() is declared as final, it cannot be overridden in B. if you attempt to do so, a compile-time error will result.

- ✓ Methods declared as final can sometimes provide a performance enhancement: the compiler is free to inline calls to them because it "knows" they will not be overridden by a subclass.
- ✓ When a small final method is called, often the java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with final methods.
- ✓ Normally, java resolves call to methods dynamically, at run time. This is called late binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

Using final to prevent Inheritance

- ✓ Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final.
- ✓ Declaring a class as final implicitly declares all of its methods as final, too.
- ✓ As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.
- ✓ Ex of final class:

```
Final class A
{
      ----
}
Class B extends A
{
      //error can'subclass A
}
```

As the comments imply, it is illegal for B to inherit A since A is declared as final.

Nested & Inner classes

- ✓ It is possible to define a class within another class, such classes are known as nested classes.
- ✓ The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B is known to A, but not outside of A.

- ✓ A nested class (B) has access to the members, including private members, of class in which it is nested (A).
- ✓ However, the enclosing class does not have access to the members of the nested class.
- ✓ There are two types of nested classes:
  - Static
  - non-static (inner-class)

Static nested class

- ✓ A static nested class is one which has the static modifier applied because it is static, it must access the member of its enclosing class through an object. i.e. it can not refer to members of its enclosing class directly.
- ✓ Because of this reason, static nested classes are rarely used.

Non-Static nested class(inner-class)

- ✓ The most imp type of nested class is the inner class.
- ✓ An inner class is a non-static nested class.
- ✓ It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static member of the outer class do.
- ✓ Thus, an inner class is fully within the scope of its enclosing class.
- ✓ The following program shows how to define and use an inner-class. The class named outer has one instance variable and method and defines one inner class called Inner.

```
class Outer
{
	int outer_x =100;
	void test()
	{
		Inner inner =  new Inner();
		inner.display();
	}
	class Inner
	{
		void display()
		{
		System.out.println ("Display Outer_X="+outer_x);
		}
	}
}
class InnerClassDemo
{
	public static void main (String args[])
	{
		Outer outer = new Outer();
		outer.test();
```

```
        }
}
```

- ✓ In the program, an inner class named Inner is defined within the scope of class Outer.
- ✓ Therefore, any code in class Inner can directly access the variable outer_x. method named display() is defined inside Inner. This method display outer_x on the output stream.
- ✓ The main() method of InnerClassDemo creates an instance of class outer and invokes its test() method. That method creates an instance of class Inner and the display() method is called.
- ✓ Inner class is known only within the scope of outer class.
- ✓ The java compiler generates an error message. If any code outside of class outer attempts to instantiate class Inner.
- ✓ An inner class has access to all of the members of its enclosing class, but the reverse is not true.
- ✓ Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

```
class Outer1
{
        int outer_x = 100;
        void test1()
        {
                Inner1 inner = new Inner1();
                inner.display();
        }
        class Inner1
        {
                //int y= 10;  // local to Inner
                void display()
                {
                        System.out.println("Display outer" +outer_x);
                }
        }
        void showy ()
        {
                //System.out.println (y);   // error y is not known here.
        }
}
class InnerClassDemo1
{
        public static void main(String args[])
        {
                Outer1 outer = new Outer1();
                outer.test1();
        }
}
```

❖ Abstract Classes

- ✓ A class that contains at least one abstract method and therefore can never be instantiated. Abstract classes are created so that other classes can inherit them and implement their abstract methods.
- ✓ Sometimes, a class that you define represents an abstract concept and as such, should not be instantiated.
- ✓ There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- ✓ That is, sometimes you will want to create a superclass that only defines generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- ✓ Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.
- ✓ You can require that certain methods be overridden by subclasses by specifying the abstract type modifier.
- ✓ These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass. Thus, a subclass must override them- it cannot simply use the version defined in the superclass.
- ✓ To declare an abstract method, use this general form:
  Abstract type name(parameter-list);
  As you can see, no method body is present.
- ✓ Any class that contains one or more abstract methods must also be declared abstract.
- ✓ To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.
- ✓ There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods.
- ✓ Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

```
Abstract class A
{
    Abstract void callme();
```

```
        Void callmetoo()
        {
                System.out.println("this is a concrete method");
        }
}
Class B extends A
{
        Void callme()
        {
                System.out.println("B's implementation of callme");
        }
}
Class AbstractDemo
{
        Public static void main(String args[])
        {
                B b=new B();
                b.callme();
                b.callmetoo();
        }
}
```

- ✓ Notice that no objects of class A are declared in the program. As mentioned, it is not possible to instantiate an abstract class.
- ✓ One other point: class A implements a concrete method called callmetoo(). This is perfectly acceptable.
- ✓ Abstract classes can include as much implementation as they see fit.
- ✓ Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because java's approach to run-time polymorphism is implemented through the use of superclass references.
- ✓ Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example.
- ✓ Using an abstract class, you can improve the Figure class shown earlier, since there is not meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares area() as abstract inside Figure.

```
Abstract class Figure
{
        Double dim1;
        Double dim2;
        Figure(double a, double b)
        {
```

```
            Dim1=a;
            Dim2=b;
        }
        Abstract double area();
}
Class Rectangle extends Figure
{
        Rectangle(double a, double b)
        {
            Super(a,b);
        }
        Double area()
        {
            System.out.println("Inside area for Rectangle.");
            Return dim1*dim2;
        }
}

Class Triangle extends Figure
{
        Triangle(double a, double b)
        {
            Super(a,b);
        }
        Double area()
        {
            System.out.println("Inside area for Triangle.");
            Return dim1*dim2/2;
        }
}
Class AbstractAreas
{
        Public static void main(String args[])
        {
            // Figure f= new Figure(10,10);// illegal now
            Rectangle r= new Rectangle(9,5);
            Triangle t= new Triangle(10,8);
            Figure figref;
            Figref=r;
            System.out.println("Area is" +figref.area());
            Figref=t;
            System.out.println("Area is" +figref.area());
        }
}
```

- ✓ As the comment inside main() indicates, it is no longer possible to declare objects of type figure, since it is now abstract. And all subclasses of figure must override area().
- ✓ To prove this to yourself, try creating a subclass that does not override area(). You will receive a compile-time error.
- ✓ Although it is not possible to create an object of type Figure, you can create a reference variable of type Figure. The variable figref is declared as reference to Figure. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

❖ Inheritance

- ✓ Inheritance is the process by which object of one class acquires the properties of another class.
- ✓ Inheritance allows the creation of hierarchical classifications.
- ✓ A class that is inherited is called a superclass. The class that does the inheriting is called a subclass.
- ✓ Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and add its own, unique elements.
- ✓ To inherit a class, you simply incorporate the definition of one class into another by using the extend keyword.
- ✓ The general form of class declaration that inherits a superclass is shown here.

```
Class subcls-name extends supercls-name
{
        // body of class.
}
```

- ✓ You can only specify one super class for any subclass.
- ✓ Java does not support the inheritance of multiple superclasses into a single subclass.
- ✓ We can create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

```
class A
{
        int i, j;
        void showij()
{
        System.out.println ("i and j: - "+i+" "+j);
}
}
```

```
class B extends A
{
	int k;
	void showk()
	{
		System.out.println("k="+k);
	}
	void sum()
	{
		System.out.println("i+j+k="+(i+j+k));
	}
}


class SimpleInheritance
{
	public static void main(String args[])
	{
		A superob= new A();
		B subob= new B();
		superob.i=10;
		superob.j=20;
		System.out.println("Content of superob:");
		superob.showij();
		System.out.println();

		subob.i=7;
		subob.j=8;
		subob.k=9;
		System.out.println("Content of subob:");
		subob.showij();
		subob.showk();
		System.out.println();
		System.out.println("sum of i,j and k in subob=");
		subob.sum();
	}
}
```

Output:

```
	Contents of superob:
	I and j= 10 20
	Content of subob:
	I and j= 7 8
	K=9
	Sum of I, j and k in sumob=
	I+j+k=24
```

Member Access and Inheritance:

- ✓ A subclass includes all the members of its superclass but it cannot access those members of the superclass that have been declared as private.

```
class A
{
	int i;
	private int j;
	void setij(int x,int y)
	{
		i=x;
		j=y;
	}
}
class B extends A
{
	int total;
	void sum()
	{
		total=i+j;  //error j is not accessible
	}
}
class Access
{
	public static void main(String args[])
	{
		B subob= new B();
		subob.setij(10,12);
		subob.sum();
		System.out.println("Total is"+subob.total);
	}
}


class Box
{
	double width;
	double height;
	double depth;

	Box (Box ob)
	{
		width=ob.width;
		height=ob.width;
```

```
            depth=ob.depth;
        }
        Box (double w, double h, double d)
        {
            width=w;
            height=h;
            depth=d;
        }
        Box ()
        {
            width=-1;
            height=-1;
            depth=-1;
        }
        Box (double len)
        {
            width=height=depth=len;
        }
        Double volume()
        {
            return width*height*depth;
        }
}


class BoxWeight extends Box
{
        double weight;
        BoxWeight (double w, double h, double d, double m)
        {
            width=w;
            height=h;
            depth=d;
            weight=m;
        }
}
class DemoBoxWeight
{
        public static void main(String args[])
        {
            BoxWeight mybox1= new BoxWeight(10,20,15,34.5);
            BoxWeight mybox2= new BoxWeight(2,3,4,0.076);
            double vol;
            vol=mybox1.volume();
            System.out.println("Volume of mybox1 is:"+vol);
```

```
            System.out.println("Weight of mybox1
is:"+mybox1.weight);
            System.out.println();
            vol=mybox2.volume();
            System.out.println("Volume of mybox2 is:"+vol);
            System.out.println("Weight of mybox2
is:"+mybox2.weight);
            System.out.println();
        }
}
```

Output:

Volume of mybox1 is:3000.0
Weight of mybox1 is: 34.5

Volume of mybox2 is: 24.0
Weight of mybox2 is:0.076

Using super:

- ✓ In the preceding ex, classes derived form Box were not implemented as efficiently as they could have been. For Ex. The constructor for BoxWeight explicitly initializes the width, height and depth fields of Box().
- ✓ Not only does this duplicate code found in its super class, which is inefficient, but it implies that a subclass must be granted access to these members.
- ✓ However, sometimes you want to create a super class that keeps the details of its implementation to itself (i.e. it keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own.
- ✓ Since encapsulation provides a solution to this problem. Whenever a subclass needs to refer to its immediate super class, it can do so by use of the keyword super.
- ✓ Super has 2 general forms

1. It calls the super class' constructor.
2. It is used to access a member of the superclass that has been hidden by member of subclass.

Super to call superclass Constructor

- ✓ A subclass can call a constructor defined by its superclass by use of the following form of super.

  super(parameter-list)

  Where parameter-list specifies any parameter needed by the constructor in the superclass.

- ✓ super() must always be the first statement executed inside a subclass constructor.
- ✓ To see how super() is used, consider this improved version of the BoxWeight() class:

```
class Box
{
        private double width;
        private double height;
        private double depth;
        Box(Box ob)
        {
                width=ob.width;
                height=ob.height;
                depth=ob.depth;
        }
        Box(double w, double h, double d)
        {
                width=w;
                height=h;
                depth=d;
        }
        Box()
        {
                width=-1;
                height=-1;
                depth=-1;
        }
        Box(double len)
        {
                width=height=depth=len;
        }
        Double volume()
        {
                return width*height*depth;
        }
}


class BoxWeight extends Box
{
        double weight;
        BoxWeight (BoxWeight ob)
        {
                super(ob);
                weight=ob.weight;
        }
```

```
        BoxWeight(double w, double h, double d, double m)
        {
                super(w,h,d);
                weight=m;
        }
        BoxWeight()
        {
                super();
                weight=-1;
        }
        BoxWeight(double len, double m)
        {
                super(len);
                weight=m;
        }
}


class DemoSuper
{
        public static void main(String args[])
        {
                BoxWeight mybox1= new BoxWeight(10,20,15,34.3);
                BoxWeight mybox2= new BoxWeight(2,3,4,0.076);
                BoxWeight mybox3= new BoxWeight();
                BoxWeight mycube= new BoxWeight(3,2);
                BoxWeight myclone= new BoxWeight(mybox1);
                double vol;

                vol=mybox1.volume();
                System.out.println("Volume of mybox1 is:"+vol);
                System.out.println("Weight of mybox1
is:"+mybox1.weight);
                System.out.println();
                vol=mybox2.volume();
                System.out.println("Volume of mybox2 is:"+vol);
                System.out.println("Weight of mybox2
is:"+mybox2.weight);
                System.out.println();
                vol=myclone.volume();
                System.out.println(vol);
                System.out.println(myclone.weight);
                System.out.println();
                vol=mycube.volume();
                System.out.println("volume of mycube is:"+vol);
```

```
            System.out.println("weight of mycube
is:"+mycube.weight);
            System.out.println();
            }
      }

BoxWeight(BoxWeight ob)
      Super(ob);
      Weight=ob.weight;
```

- ✓ Super() is called with an object of type BoxWeight not of type Box. This invokes the constructor Box(Box ob). A superclass variable can be used to reference any object derived form that class. Thus, we are able to pass a BoxWeight object ot he box constructor. Box only has knowledge of its own members.
- ✓ When a subclass calls super(), it is calling the constructor of its immediate superclass. Thus, super() always refer to the superclass immediately above the calling class. This is true in a multileveled hierarchy. Also, super() must always be the first statement executed inside a subclass constructor.

Second use for super:

- ✓ The second form of super acts somewhat like this,except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

Super.member
Member=can be either a method or an instance variable.

- ✓ This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy.

```
class A
{
      int i;
}
class B extends A
{
      int i;  //this I hides the I in A
      B(int a, int b)
      {
            super.i=a;
            i=b;
      }
      void show()
      {
```

```
            System.out.println("I in superclass:"+super.i);
            System.out.println("I in subclass:"+i);
        }
}
class UseSuper
{
        public static void main(String args[])
        {
            B subob=new B(1,2);
            subob.show();
        }
}
```

- ✓ Although, the variable I in B hides the I in A, super allows access to the I defiend in the superclass. Super can also be used to call methods that are hidden by a subclass.

Multilevel Hierarchy

- ✓ You can build hierarchies that contain as many layers of inheritance as you like.
- ✓ It is acceptable to use a subclass as a super class of another.
- ✓ For example, we have classes called A, B and C. C can be subclass of B, which is a subclass of A. when this type of situation occurs, each subclass inherits all of the traits found in all of its super classes. In this case, C inherits all aspects of B and A.

```
class Box
{
        private double width;
        private double height;
        private double depth;
        Box(Box ob)
        {
            width=ob.width;
            height=ob.height;
            depth=ob.depth;
        }
        Box(double w, double h, double d)
        {
            width=w;
            height=h;
            depth=d;
        }
        Box()
        {
            width=-1;
            height=-1;
```

```
            depth=-1;
        }
        Box(double len)
        {
            width=height=depth=len;
        }
        Double volume()
        {
            return width*height*depth;
        }
}


class BoxWeight extends Box
{
        double weight;
        BoxWeight(BoxWeight ob)
        {
            super(ob);
            weight=ob.weight;
        }
        BoxWeight(double w, double h, double d, double m)
        {
            super (w,h,d);
            weight=m;
        }
        BoxWeight()
        {
            super();
            weight=-1;
        }
        BoxWeight(double len, double m)
        {
            super(len);
            weight=m;
        }
}

class Shipment extends BoxWeight
{
        double cost;
        Shipment(Shipment ob)
        {
            super(ob);
            cost=ob.cost;
        }
```

```
        Shipment(double w, double h, double d, double m, double c)
        {
                super(w,h,d,m);
                cost=c;
        }
        Shipment()
        {
                super();
                cost=-1;
        }
        Shipment(double len, double m, double c)
        {
                super(len, m);
                cost=c;
        }
}

class DemoShipment
{
        public static void main(String args[])
        {
                Shipment ship1= new Shipment(10,20,15,10,3.41);
                Shipment ship2= new Shipment(2,3,4,0.76,1.28);
                Double vol;
                vol=ship1.volume();
                System.out.println("Volume of ship1 is:"+vol);
                System.out.println("Weight of ship 1
is:"+ship1.weight);
                System.out.println("Shipping cost: $"+ship1.cost);
                System.out.println();
                vol=ship2.volume();
                System.out.println("Volume of ship1 is:"+vol);
                System.out.println("Weight of ship 1
is:"+ship2.weight);
                System.out.println("Shipping cost: $"+ship2.cost);
        }
}
```

```
Volume of ship1 is: 3000.0
Weight of ship1 is: 10.0
Shipping cost: $3.41
Volume of ship1 is: 24.0
Weight of ship1 is: 0.76
Shipping cost: $1.28
```

- ✓ This example shows that super() always refers to the constructor in the closest superclass.

- ✓ The super() in shipment calls the constructor in BoxWeight. The super() in BoxWeight calls the constructor in Box.
- ✓ In a class hierarchy, if a superclass constructor requires paramenters, then all subclasses must pass those parameters.

Interface

- ✓ interface is similar to an abstract class in that its members are not implemented.
- ✓ In interfaces, none of the methods are implemented. There is no code at all associated with an interface.
- ✓ Once it is defined, any number of classes can implement an interface.
- ✓ One class can implement any number of interfaces.
- ✓ To implement an interface, a class must create the complete set of methods defined by the interface.
- ✓ Each class is free to determine the details of its own implementation.
- ✓ By providing the interface keyword, java allows you to fully utilize the "One interface multiple methods" aspect of polymorphism.
- ✓ Interfaces are designed to support dynamic method resolution at run time. For a method to be called from one class to another, both classes need to be present at compile time so the java compiler can check to ensure that the method signatures are compatible.
- ✓ Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritance in C++.

Defining an Interface

- ✓ The general form of an Interface:

```
Access-sp interface-name
{
        Return-type method-name(parameter-list);
        Type final_varname1=value;
}
```

Where

Access-sp is either public or not used.

- ✓ When no access specifier is used, then default access result and interface is only available to other members of the same package.
- ✓ When it is declared as public, the interface can be used by any other code.
- ✓ The methods which are declared have no bodies they end with a semicolon after the parameter list.

- ✓ Actually they are abstract method, there can be no default implementation of nay method specified within an interface.
- ✓ Each class that includes an interface must implement all of the methods.
- ✓ Variables can be declared inside of interface declarations. They are implicitly final and static, means they can not be changed by implementing it in a class. They must also be initialized with a constant value.
- ✓ All methods and variables are implicitly public if the interface, is declared as public.

Example:

```
Interface Callback
{
        void callback(int param);
}
```

Implementing Interfaces

- ✓ Once an interface has been defined, one or more classes can implement that interface.
- ✓ To implement an interface, include the implements clause in a class definition, and then create the methods defined b the interface.
- ✓ The general form of a class that includes the implements clause looks like this:

```
        Access class classname [extends superclass] [implements
interface, [,interface..]]
        {
                // class body
        }
```

- ✓ The methods that implement an interface must be declared as public.
- ✓ The type-signature of implementing method must match exactly the type signature specified in the interface.

```
Class Client implements Callback
{
        Public void callback(int p)
        {
                System.out.println("callback called with"+p);
        }
}
```

Note: when you implement an interface method, it must be declared as public.

- ✓ It is possible for classes that implement interfaces to define additional members of their own.

- ✓ For example, client implements callback() and adds the method nonIfaceMeth()

```
Class Client implements Callback
{
        Public void Callback(int p)
        {
                System.out.println("callback called with "+p);
        }
        Void nonIfaceMeth()
        {
                System.out.println("classes that implements interfaces
may also define other members, too.");
        }
}
```

Accessing Implementation through Interface References

- ✓ You can declare variable as object references that use an interface rather than a class type.
- ✓ Any instance of any class that implements the declared interface can be referred to by such a variable.
- ✓ When you call a method through one of these references, the correct version will be called based on the actual version will be called based on the actual instance of the interface being referred to.
- ✓ The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.
- ✓ The following example, calls the callback() method via an interface reference variable.

```
Class TestIface
{
        Public static void main(String args[])
        {
                Callback c= new Client();
                c.Callback(42);
        }
}
```

Output: callback called with 42

- ✓ Variable c is declared to be of the interface type callback, it was assigned an instance of client.
- ✓ Although c can be used to access the callback() method, it cannot access any other members of the client class.
- ✓ An interface reference variable only has knowledge of the method declared by its interface declaration.
- ✓ Thus, c could not be used to access nonIfaceMeth() since it is defined by client but not Callback.

```
Class AnotherClient implements Callback
{
        Public void callback(int p)
        {
                System.out.println("Another version of callback");
                System.out.println("p squared is" +(p*p));
        }
}
Class TestIface2
{
        Public static void main(String args[])
        {
                Callback c= new Client();
                AnotherClient ob= new AnotherClient();
                c.callback(42);
                c=ob;
                c.callback(42);
        }
}
```

Callback called with 42
Another version of callback
P squared is 1764

Partial Implementation

- ✓ If a class includes an interface but does not fully implement the method defined by that interface, then that class must be declared as abstract.

```
Abstract class Incomplete implements Callback
{
        Int a,b;
        Void show()
        {
                System.out.println(a +" " +b);
        }
        //- - -
}
```

Interfaces can be Extended:

- ✓ One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes.
- ✓ When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance.

- ✓ Any class that implements an interface must implement all methods defined by that interface, including any that are inherited form other interfaces.

```
Interface A
{
        Void meth1();
        Void meth2();
}
Interface B extends A
{
        Void meth3();
}
Class MyClass implements B
{
        Public void meth1()
        {
                System.out.println("Implement meth1");
        }
        Public void meth2()
        {
                System.out.println("Implement meth2");
        }
        Public void meth3()
        {
                System.out.println("Implement meth3");
        }
}
Class IFExtend
{
        Public static void main(String args[])
        {
                MyClass ob= new MyClass();
                Ob.meth1();
                Ob.meth2();
                Ob.meth3();
        }
}
```

Access Control

- ✓ Encapsulation links data with the code that manipulates it. Encapsulation provides another important attribute: Access Control

- ✓ Through encapsulation, you can control what parts of a program can access the member of a class. By controlling access, you can prevent misuse.
- ✓ How a member can be accessed is determined by the access specifier that modifies its declaration.
- ✓ Java supplies a rich set of access specifies. Some aspects of access control are related to inheritance or packages. Lets begin by examining access control as it applies to a single class.
- ✓ Java's access specifiers are:

1. Public
   When a member of a class is modified by the public specifier, then that member can be accessed by any other code.
2. Private
   When a member of a class is specified as private, then that member can only be accessed by other members of its class.
3. Protected
   If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.
4. Default
   when a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package.

- ✓ Now, we can understand why main (), has always been preceded by the public specifier. It is called by the code outside of the program. (by java run-time system).
- ✓ When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.
- ✓ In the classes developed so far, all member of a class have used the default access mode, which is essentially public usually, you will want to restrict access to the data members of a class allowing access only through methods. Also, there will be times when you will want to define methods, which are private to a class.
- ✓ An access specifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:
  public int i;
  private double j;
  private int myMethod (int a, char b);
- ✓ To understand the effects of public, private access considers the following program:

class Test

```
{
        int a;
        public int b;
        private int c;

        void SetC (int i)
        {
                c = i;
        }
        int getc()
        {
                return c;
        }
}
class AccessTest
{
        public static void main (String args[])
        {
                Test ob = new Test();
                ob.a = 10;
                ob.b = 20;
                //ob.c = 100;   // cause an error.
                ob.SetC (100);
                System.out.println("a, b and c: "+ob.a+" "+ob.b+"
"+ob.getc());
        }
}
```

Synchronized modifier:

The synchronized modifier is used to specify that method is thread safe . this means that only one path of execution is allowed in to a synchronized method at a time. In a multithreaded environment like java, it is possible to have many different path of execution running through same code. The synchronized modifier changed this rule by allowing only a thread access to a method at once, forcing the other thread to wait their turn.

Transient and volatile:

These are used to handle somewhat special situations.
When an instance variable is declared as transient, then its value need not persist when an object is stored.
For example:

```
Class t
{
        transient  int a;// will not persist
        int b;// will persist
}
```

Volatile:
The volatile modifiers tells the compiler that the variable modified by volatile can be changed unexpectedly by other parts of your program. one of these situations involves multithreaded program, sometimes two or more thread share the same instance variable.

Native:

The native modifier is used to identify method that have native implementation. The native modifiers informs the java compiler that a method's implementation
Is in an external C file. Its declaration has no body.

```
native int calc();
```

Chapter-3 Packages

- Java API Packages ……
- Accessing packages and use….
- Adding class to package …….
- Java.lang package classes ...…..
- Java.util package ……...

❖ Packages

- Packages are container for classes that are used to keep the class name space compartmentalized.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definition.
- Java provides a mechanism for partitioning the classname space into more manageable chunk. This mechanism is the package.
- The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class member that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each-other but not expose that knowledge to the rest of the world.

Defining a Package

- ✓ To create a package, simply include a package command as the first statement in java source file.
- ✓ Any classes declared within that file will belong to the specified package.
- ✓ The package statement defines a name space in which classes are stored.
- ✓ If you omit the package statement, the class names are put into the default package, which has no name.

- General form of package statement:
  Package pkg;

- ✓ Where, pkg is the name of the package. For example following statement creates a package called MyPackage must be stored in directory called MyPackage. The directory name must match the package name exactly.
- ✓ More than one file can include the same package statement.
- ✓ You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of multileveled package statement:
  Package pkg1[.pkg2[.pkg3]];

For example:

Package java.awt.image;

- ✓ This package needs to be stored in java/awt/image, java\awt\image or java:awt:image on your system.
- ✓ You can not rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

- ✓ How dose the java run-time system know where to look for packages that you create? – the answer has two parts:

1. by default, the java run-time system uses the current working directories as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found.
2. you can specify a directory path or paths by setting the CLASSPATH environmental variable.

- ✓ For example, consider the following package specification:
  Package MyPack;
- ✓ In order for a program to find MyPack, one of two things must be true. Either the program is executed form a directory immediately above MyPack or CLASSPATH must be set to MyPack.

```
package MyPack;
class Balance
{
	String name;
	double bal;
	Balance(String n, double b)
	{
		name=n;
		bal=b;
	}
	void show()
	{
		if (bal>0)
		System.out.println("Name is:"+name +":$" +bal);
	}
}
class AccountBalance
{
	public static void main(String args[])
	{
		Balance current[]=new Balance[3];

		current[0]=new Balance("K.J.Fielding",123.23);
		current[1]=new Balance("will tell",157.02);
		current[2]=new Balance("Tom",-12.33);
		for(int i=0;i<3;i++)
		{
			current[i].show();
		}
	}
	}
```

- ✓ Compilation of program:

```
C:\javaprogs\MyPack>javac AccountBalance.java
C:\javaprogs>java MyPack.AccountBalance
```

- ✓ You will need to be in the directory above MyPack when you execute this command, or your CLASSPATH environment variable set appropriately.

```
C:\javaprogs>javac p\*.java
C:\javaprogs>java p.PackageDemo
Javac –d . P\*.java
```

Access Protection

- ✓ Anything declared public can be accessed form anywhere.
- ✓ Anything declared private cannot be seen outside of its class.

- ✓ Default: when a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package.
- ✓ If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.
- ✓ A class only has two possible access level: default and public. When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.

Chapter-4 Multithreding and Exception handling.

- Thread……..
- Implementing Thread using thread class and Runnable Interface…….
- Thread Methods and status…… (Running,Waiting, Slipping, suspend,Resume)…….
- Thread Priority……
- Synchronization and Deadlock in thread……..
- Exception handling using Try ,Catch,Throw,Throws , Finally…….

❖ Multithreading

- ✓ A thread is a single sequential flow of control within a program.
- ✓ Thread does not have its own address space but uses the memory and other resources of the process in which it executes. There may be several threads in one process.
- ✓ The Java Virtual Machine (JVM) manages these and schedules them for execution.
- ✓ The time needed to perform a context switch from one thread to another is substantially less than that required for performing such a change between processes.
- ✓ Multithreading is a conceptual programming paradigm where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel. For ex, one subprogram can display an animation on the screen while another may build the next animation to be displayed. This is something similar to dividing a task into subtasks and assigning them to different people for execution independently and simultaneously.

- ✓ A thread is similar to a program that has a single flow of control. It has a beginning, a body, and an end, and executes command sequentially. In fact, all main programs that we did can be called single threaded programs. Every program has at least one thread.
- ✓ Java enables us to use multiple flows of control in developing programs. Each flow of control may be thought of as a separate tiny program known as a thread that runs in parallel to others.
- ✓ A program that contains multiple flow of control is known as multithreaded program.
- ✓ Suppose in a java program with four threads, one main and three others. The main thread is actually the main method module, which is designed to create and start the other three threads, namely A,B,C.
- ✓ Once initiated by the main thread, the threads A,B, and C run concurrently and share the resource jointly.
- ✓ The ability of a language to support multithreads is referred to as concurrency. Since threads in java are subprograms of a main application program and share the same memory space, they are known as lightweight threads or lightweight processes.
- ✓ 'threads running in parallel' does not really mean that they actually run at the same time. Since all the threads are running on a single processor, the flow of execution is shared between the threads. The java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently.
- ✓ Multithreading enables programmers to do multiple things at one time. They can divide a long program into threads and execute them in parallel. For ex, we can send tasks such as printing into the background and continue to perform some other task in the foreground. This approach would considerably improve the speed of our programs.
- ✓ Threads are extensively used in java-enabled browser such as HotJava. These browsers can download a file to the local computer, display a web page in the window, output another web page to a printer and so on.

Creating Threads

- ✓ Thread class in the java.lang package allows you to create and manage threads. Each thread is a separate instance of this class.
- ✓ A new thread can be created in two ways:

1. by extending a thread class

- Define a class that extends Thread class and override its run() method with the code required by the thread.

2. by implementing an interface
   - Define a class that implements Runnable interface. The Runnable interface has only one method, run(), that is to be defined in the method with the code to be executed by the thread.

Extending the Thread class

- ✓ We can directly extend the Thread class

```
class Threadx extends Thread
{
public void run()
{
    //logic for the thread
}
}
```

- ✓ The class ThreadX extends Thread. The logic for the thread is contained in the run() method. That method may be ery simple or cmplex. It can create other objects or even initiate other threads.
- ✓ The program can start an instance of the thread by using the form shown here:
  - ThreadX tx= new ThreadX();
  - Tx.start();
  - newA().start();//we can write it like this also
- ✓ The first line instantiates the ThreadX class. The second line invokes the start() method of that object to start the thread executing. One of the actions of the start() method is to invoke the run() method. It is possible to create and start several instances of ThreadX that execute concurrently.

Another way to create a thread

- ✓ Declare a class that implements the Runnable interface. This method declares only one method as shown here:

```
public void run();
class RunnableY implements Runnable
{
    Public vod run()
    {
        // logic for thread
    }
}
```

- ✓ The application can start an instance of the thread by using the following code:

```
RunnableY ry = new RunnableY();
```

```
ThreadY ty= new Thread(ry);
Ty.start();
```

- ✓ 1st line instantiate the RunnableY class.
- ✓ 2nd line instantiate the Thread class. A reference to the RunnableY object is provided as the argument to the constructor.
- ✓ Last line starts the thread.

Thread Life Cycle

- ✓ During the life time of a thread, there are many states it can enter. They include:
    1. Newborn state
    2. Runnable state
    3. Running state
    4. Blocked state
    5. Dead state
- ✓ A thread is always in one of these 5 states. It can move from one state to another via a variety of ways as shown in fig.
    1. Newborn state
        - When we create a thread object, the thread is born and is said to be in newborn state. The thread is not yet scheduled for running. At this state, we can do only one of the following with it:
        - Schedule it for running using start() method.
        - Kill it using stop() method
        - If scheduled, it moves to the runnable state. If we attempt to use any other method at this stage, an exception will be thrown.
    2. Runnable state (start())
        - The runnable state means that the thread is ready for execution and is waiting for the availability of the processor.
        - That is, the thread has joined the queue of threads that are waiting for execution.
        - If all threads have equal priority, then they are given time slots for execution in round robin fashion. i.e. first-come, first-serve manner.
        - The thread that relinquishes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as time-slicing.
        - If we want a thread to relinquish control to another thread of equal priority before its turn comes, we can do so by using the yield() method.

    3. Running State

- Running means that the processor has given its time to the thread for its execution.
- The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread.
- A running thread may relinquish its control in one of the following situations:

1. It has been suspended using suspend() method. A suspended thread can be revived by using the resume() method. This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.
2. It has been made to sleep. We can put a thread to sleep for a specified time period using the method sleep (time), where time is in milliseconds. This means that the thread is out of the queue during this time period. The thread re-enter the runnable state as soon as this time period is elapsed.
3. It has been told to wait until some event occurs. This is done using the wait() method. The thread can be scheduled to run again using the notify() method.
4. Blocked State
   - A thread is said to be blocked when it is prevented form entering into the runnable state and subsequently the running state.
   - This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements.
   - A blocked thread is considered " not runnable" but not dead and therefore fully qualified to run again.
5. Dead State
   - A running thread ends its life when is has completed executing its run() method. It is a natural death.
   - However, we can kill it by sending the stop message to it at any state thus causing a premature death to it.
   - A thread can be killed as soon it is born, or while it is running, or even when it is in "not runnable" (blocked) condition.
   - The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing, t.join(); causes the current thread to pause execution until t's thread terminates. Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify. Like sleep, join responds to an interrupt by exiting with an InterruptedException.

Stopping a Thread

- ✓ Whenever we want to stop a thread form running further, we may do so by calling its stop() method like:
  ty.stop();
- ✓ This statement causes the thread to move to the dead state. A thread will also move to the dead state. A thread will also move to the dead state automatically when it reaches the end of its method.
- ✓ The stop() method may be used when the premature death of a thread is desired.

Blocking a Thread

- ✓ A thread can also be temporarily suspended or blocked form entering into the runnable and subsequently running state by using either of the following thread methods:
  - Sleep() – blocked for a specified time.
  - Suspend() – blocked until further orders.
  - Wait() – blocked until certain condition occurs.
- ✓ These methods cause the thread to go into the blocked state. The thread will return to the runnable state when the specified time is elapsed in the case of sleep(), the resume() method is invoked in the case of suspend(), and the notify() method is called in the case of wait().

Thread class

- ✓ Some of the constructors for Thread are as follows:
  - Thread()
  - Thread(Runnable r)
  - Thread(Runnable r, String s)
  - Thread(String s)
- ✓ Here, r is a reference to an object that implements the Runnable interface and s is a String used to identify the thread.

Methods of Thread class

Method	Description
• Thread currentThread()	returns a reference to the current thread
• Void sleep(long msec) • Throws InterruptedException	causes the current thread to wait for msec milliseconds
• Void sleep(long msec, int nsec) • Throws InterruptedException	causes the current thread to wait for msec milliseconds plus nsec nanoseconds

- Void yield() causes the current thread to yield control of the processor
- to other threads
- String getName() returns the name of the thread.
- Int getPriority() returns the priority of the thread
- Boolean isAlive() returns true if this thread has been started and has not
- Yet died. Otherwise, returns false.
- Void join() causes the caller to wait until this thread dies.
- Throws InterruptedException
- Void join(long msec) causes the caller to wait a max of msec until this thread dies.
- Throws InterruptedException if msec is zero, there is no limit for the wait time.
- Void join(long msec, int nsec) causes the caller to wait a max of msec plus nsec until this thread
- Throws Interruptedexception dies. If msec plus nsec is zero, there is no limit for the wait time.
- Void run() comprises the body of the thread. This method is overridden by subclasses.
- Void setName(String s) sets the name of this thread to s.
- Void setPriority(int p) sets the priority of this thread to p.
- Void start() starts the thread
- String toString() Returns the string equivalent of this thread.

❖ Thread Priority

- In java, each thread is assigned a priority, which affects the order in which it is scheduled for running.
- The threads of the same priority are given equal treatment by the Java scheduler and therefore, they share the processor on a first-come, first-serve basis.
- Java permits us to set the priority of a thread using the setPriority() method as follows:
  ThreadName.setPriority(intNumber);
- The intNumber may assume one of these constants or any value between 1 and 10.
- The intNumber is an integer value to which the thread's priority is set. The Thread class defines several priority constants:
  - MIN_PRIORITY=1
  - NORM_PRIORITY=5
  - MAX_PRIORITY=10
- The default setting is NORM_PRIORITY.

Synchronization

- ✓ When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.
- ✓ As an example, consider a bank account that is shared by multiple customers. Each of these customers can make deposits to or withdrawals form this account. Your application might have a separate thread to process the actions of each user.
- ✓ The solution to this problem is to synchronize the access to this common data.
- ✓ This can be done in two ways. First, a method can be synchronized by using the synchronized keyword as a modifier in the method declaration.
- ✓ When a thread begins executing a synchronized instance method, it automatically acquires a lock on that object.
- ✓ The lock is automatically relinquished when the method completes.
- ✓ Only one thread has this lock at any time.
- ✓ Therefore, only one thread may execute any of the synchronized instance methods for that same object at a particular time.
- ✓ If a second thread attempts to execute a synchronized instance method for that same object, the JVM automatically causes the second thread to wait until the first thread relinquishes the lock.
- ✓ When a thread begins executing a synchronized static method, it automatically acquires a lock on the associated Class object.
- ✓ Another way to synchronize access to common data is via a synchronized statement block. This has the following syntax:

```
synchronized(obj)
{
        //statement block
}
```

Here, obj is the object to be locked. If you wish to protect instance data, you should lock against that object. If you wish to protect class data, you should lock the appropriate Class object.

Dead Lock

- ✓ Deadlock is an error that can be encountered in multithreaded programs.

- ✓ It occurs when two or more threads wait for ever for each other to relinquish locks.
- ✓ Assume that thread1 holds lock on object1 and waits for a lock on object2. thread2 holds a lock on object2 and waits for a lock on object1. neither of these threads may proceed. Each waits forever for the other to relinquish the lock it needs.
- ✓ Deadlock situations can also arise that involve more than two threads. Assume that thread1 waits for a lock held by thread2. thread2 waits for a lock held by thread3. thread3 waits for a lock held by thread1.

Thread Communication

- ✓ Deadlock can occur if a thread acquires a lock and does not relinquish it.
- ✓ Now we will see how threads can cooperate with each other. A thread can temporarily release a lock so other threads can have an opportunity to execute a synchronized method or statement block. That lock can be acquired again at a later time.
- ✓ The class Object defines 3 methods that allow threads to communicate with each other. The wait() method allows a thread that is executing a synchronized method or statement block on that object to release the lock and wait for a notification from another thread.
  - ✓ Void wait() throws InterruptedException
  - ✓ Void wait(long msec) throws InterruptedException
  - ✓ Void wait(long msec, int nsec) throws InterruptedException
- ✓ The first form causes the current thread to wait indefinitely.
- ✓ The second form causes the current thread to wait for msec.
- ✓ The last form causes the current thread to wait for msec plus nsec.
- ✓ The notify() method allows a thread that is executing a synchronized method to notify another thread that is waiting for a lock on this object.
- ✓ If several threads are waiting, only one of these is selected. The selection criteria are determined by the implementer of the JVM. The signature of this method is shown here:
  void notify();
- ✓ The notifyAll() method allows a thread that is executing synchronized method to notify all threads that are waiting for a lock on this object. The signature of this method is shown here:

  void notifyAll();

- ✓ It is important to understand that when a thread executes the notify() or notifyAll() method, it does not relinquish its lock at that moment. This occurs only when it leaves the synchronized method.
- ✓ The net effect of the notify() and notifyAll() methods is that one thread resumes its execution of the synchronized method. It returns form the wait() method and continues executing the next statement.

Exception Handling

- ✓ An exception is an object that is generated at run-time to describe a problem encountered during the execution of a program.
- ✓ Some causes for an exception are integer division-by-zero, array index negative or out-of-bounds, illegal cast, interrupted I/O operation, unexpected end-of-file condition, missing file, incorrect number format.
- ✓ An exception is an abnormal condition that arises in a code sequence at run time or we can say an exception is a run-time error.
- ✓ In computer languages that do not support exception handling, errors must be checked and handled manually through the use of error codes.
- ✓ A java exception is an object that describes an exceptional condition that has occurred in a piece of code.
- ✓ When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.
- ✓ Java exception handling is managed via five keywords: try, catch, throw, throws and finally.

1. Try: program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown.
2. Catch: your code can catch this exception using catch and handle it in some rational manner.
3. Throw: system-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw.
4. Throws: any exception that is thrown out of a method must be specified by a throws clause.

5. Finally: any code that absolutely must be executed before a method returns is put in a finally block.

General form:

```
Try
{
        //block of code to monitor for errors…
}
Catch(ExceptionType1 exOb)
{
        // exception handling block
}
Finally
{
        // finally block
}
```

- ✓ The try statement contains a block of statements enclosed by braces. This is the code you want to monitor for exceptions. If a problem occurs during its executing, an exception is thrown.
- ✓ Immediately following the try block is a sequence of catch blocks. Each of these begins with the catch keyword. An argument is passed to each catch block. That argument is the exception object that contains information about the problem.
- ✓ If a problem occurs during execution of the try block, the JVM immediately stops executing the try block and looks for a catch block that can process that type of exception. Any remaining statements in the try block are not executed. The search begins at the first catch block. If the type of the exception object matches the type of the catch block parameter, those statements are executed. Otherwise, the remaining catch clauses are examined in sequence for a type match.
- ✓ When a catch block completes executing, control passes to the statements in the finally block. The java compiler ensures that the completes without problems, the finally block executed in all circumstances.
- ✓ When a try block completes without problems, the finally block executes. Even if a return statement is included in a try block, the compiler ensures that the finally block is executed before the current method returns.
- ✓ The finally block is optional. However, in some applications it can provide a useful way to relinquish resources. For example, you may wish to close files or databases at this point.
- ✓ Each try block must have at least one catch or finally block.

```
Class ExceptionTest
```

```
{
    Public static void main(String args[])
    {
        Int a=10;
        Int b=5;
        Int c=5;
        Int x,y;
        Try
        {
            X=a/(b-c);
        }
        Catch(ArithmeticException e)
        {
            System.out.println("Division by zero");
        }
        Y=a/(b+c);
        System.out.println("y="+y);
    }
}
```

Output:
Division by zero
Y=1

Displaying a Description of an Exception

- ✓ Throwable overrides the toString() method (defined by Object) so that it returns a string containing a description of the excaeption. You can display this description in a println() statement by simply passing the exception as an argument. For example, the catch block in the preceding program can be rewritten like this:

```
Catch (ArithmeticException e)
{
    System.out.println("Exception:" +e);
    A=0;
}
```

<u>Multiple catch Clauses</u>

- ✓ In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.
- ✓ When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- ✓ After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

- ✓ The following example traps two different exception types:

```
Class MultiCatch
{
    Public static void main(String args[])
    {
        Try
        {
            Int a=args.length;
            System.out.println("a=" +a);
            Int b= 42/a;
            Int c[]={1};
            C[42]=99;
        }
        Catch(ArithmeticException e)
        {
            System.out.println("divide by zero:" +e);
        }
        Catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Array index oob:" +e);
        }
    }
    System.out.println("After try/catch block");
    }
}
```

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

```
Class SuperSubCatch
{
    Public static void main(String args[])
    {
        Try
        {
            Int a=0;
            Int b=42/a;
        }
        Catch(Exception e)
        {
            System.out.println("Generic Exception catch.");
        }
```

```
            // ArithmeticException is a subclass of Exception.
            Catch(ArithmeticException e)
            {
                  // error-unreachable
                  System.out.println("this is never reached.");
            }
      }
}
```

- ✓ If you try to compile this program, you will receive an error message stating that the second catch statement is unreachable because the exception has already been caught. Since ArithmeticException is a subclass of Exception, the first catch statement will handle all exception based errors, including ArithmeticException. This means that the second catch statement will never execute. To fix the problem, revere the order of the catch statement.

```
Class ExceptionMulti
{
      Public static void main(String args[])
      {
            Int a[]={5,10};
            Int b=5;
            Try
            {
                  Int x=a[2]/b-a[1];
            }
            Catch(ArithmeticException e)
            {
                  System.out.println("Division by zero");
            }
            Catch(ArrayIndexOutOfBoundsException e)
            {
                  System.out.println("Array index error");
            }
            Catch(ArrayStoreException e)
            {
                  System.out.println("wrong data type");
            }
            Int y=a[1]/a[0];
            System.out.println("y=" +y);
      }
}
```

Throw

- ✓ We saw that an exception was generated by the JVM when certain run-time problems occurred.
- ✓ It is also possible for our program to explicitly generate an exception. This can be done with a throw statement. Its form is as follows:

```
throw object;
```

- ✓ Here, object must be of type java.lang.Throwable. Otherwise, a compiler error occurs.
- ✓ Inside a catch block, you may throw the same exception object that was provided as an argument. This can be done with the following syntax:

```
catch(ExceptionType param)
{
    throw param;
}
```

- ✓ Alternatively, you may create and throw a new exception object as follows:

```
throw new ExceptionType(args);
```

- ✓ Here, exceptionType is the type of the exception object and args is the optional argument list for its constructor.
- ✓ When a throw statement is encountered, a search for a matching catch block begins. Any subsequent statements in the same try or catch block are not executed.

Throws

- ✓ When you write a method that can throw exceptions to its caller,
- ✓ A Java language keyword valid only at the end of method declarations that specifies which exceptions are not handled within the method but rather passed up the call stack. Unhandled checked exceptions are required to be declared in a method's throws clause whereas unchecked exceptions are generally not declared.

Chapter-5 Event Handling

- Event delegation model or event class hierarchy……..
- All classes and interfaces of event delegation……
- programmes related to event handling……..

Event Handling

Delegation Event Model

- The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events.
- The delegation event model provides a standard mechanism for a source to generate an event and send it to a set of listeners.

Event

- An event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface.
- Some of the activities that causes events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list and clicking the mouse.
- Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. We are free to define events that are appropriate for our application.

Event Sources

- A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:
  public void addTypeListener(TypeListener el)
- Here, type is the name of the event, and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener(). The method that registers a mouse motion listener is called addMouseMotionListener(). When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners that register to receive them.
- Some sources may allow only one listener to register. The general form of such a method is this:
  public void addTypeLIstener(TypeListener el)
  throws java.util.TooManyListenersException
- Here,type is the name of the event, and el is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as unicasting the event.
- A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

Public void removeTypeListener(TypeListener el)

- Here, type is an object that is notified when an event listener. For example, to remove a keyboard listener, you would call removeKeyListener()

Event Listeners

- A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.
- The method that receive and process events are defined in a set of interfaces found in java.awt.event. For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

Event Classes

Event Object

- At the root of the Java event class hierarchy is EventObject, which is in java.util. it is the superclass for all events. Its one constructor is shown here:
  EventObject(Object src)
  here, src is the object that generates this event.
- EventObject contains two methods:
  - getSource()
  - toString()
- The getSource() method returns the source of the event. Its general form is shown here:
  Object getSource()
- toString() returns the string equivalent of the event.

AWTEvent

- The abstract AWTEvent class extends EventObject and is part of the java.awt package. All of the AWT event types are subclasses of AWTEvent.
- Constructor:
  AWTEvent(Object source, int id)
  here, source is the object that generates the event and id identifies the type of the event. The possible values of id are described in the remainder of this section.
- Two of its methods are:
  - Int getId()
  - String toString()
- The getId() returns the type of the event, and toString() returns the string equivalent of the event.
- EventObject is a superclass of all events.

- AWTEvent is a superclass of all AWT events that are handled by the delegation event model.
- The package java.awt.event defines several types of events that are generated by various user interface elements.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract subclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released, also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened or quit.

ActionEvent

- An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
- The ActionEvent class defines four integer constants that can be used to identify any modifiers associated with an action event: ALT_MASK, CTRL_MASK, META_MASK, and SHIFT_MASK.
- In addition, there is an integer constant, ACTION_PERFORMED, which can be used to identify action events.
- ActionEvent has these three constructor:
  ActionEvent(Object src, int type, String cmd)
  ActionEvent(Object src, int type, String cmd, int modifiers)
  ActionEvent(Object src, int type, String cmd, long when, int modifiers)
- Here,src is a reference to the object that generated this event. The type of the event is specified by type, and its command string is cmd. The argument modifiers indicates which modifier keys were pressed when the event was generated. The when parameter specifies when the event occurred.
- String getActionCommand() obtain the name for the invoking ActionEvent object by using this method. For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.
- Int getModifiers() method returns a value that indicates which modifier keys were pressed when the event was generated.
- Long getWhen() returns the time at which the event took place. This called the event's timstamp.

AdjustmentEvent

- An AdjustmentEvent is genrated by a scroll bar. There are five types of adjustment events.
- The AdjustementEvent class defines integer constants that can be used to identify them. The constants are:

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.

UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

- There is an integer constant, ADJUSTMENT_VALUE_CHANGED, that indicates that a change has occurred.
- The constructor is:
  AdjustmentEvent(Adjustable src, int id, int type, int data)
  here, src is a reference to the object that generated this event. The id equals ADJUSTMENT_VALUE_CHANGED. The type of the event is specified by type, and its associated data is data.
- The getAdjustable() method returns the object that generated the event. Its form is shown here:
  Adjustable getAdjustable()
- The type of the adjustment event may be obtained by the getAdjustmentType() method. It returns one of the constants defined by AdjustmentEvent.
  int getAdjustmentType()
- The amount of the adjustment can be obtained from the getValue() method, shown here:
  int getValue()
  for example, when a scroll bar is manipulated, this method returns the value represented by that change.

ComponentEvent

- A ComponentEvent is generated when the size, position, or visibility of a component is changed.
- There are four types of component events.
- The ComponentEvent class defines integer constants that can be used to identify them. The constants and their meaning are:

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

- ComponentEvent has this constructor:
  ComponentEvent(Component src, int type)

- Here, src is a reference to the object that generated this event. The type of the event is specified by type.
- ComponentEvent is the superclass wither directly or indirectly of ContainerEvent, FocusEvent, KeyEvent, MouseEvent, and WindowEvent.
- The getComponent() method returns the component that generated the event. It is shown here:
  Component getComponent()

ContainerEvent

- A ContainerEvent is generated when a component is added to or removed from a container.
- There are two types of container events. The ContainerEvent class defines int constants that can be used to identify them: COMPONENT_ADDED and COMPONENT_REMOVED.
- They indicate that a component has been added to or removed from the container.
- ContainerEvent is a subclass of ComponentEvent and has this constructor:
  ContainerEvent(Component src,int type,Component comp)
  here, src is a ref. to the container that generated this event.
  The type of the event is specified by type, and the comonent that has been added to or removed form the container is comp.

FocusEvent

- A FocusEvent is generated when a component gains or loses input focus.
- These events are identified by the integer constatns FOCUS_GAINED and FOCUS_LOST.
- FocusEvent is a subclass of ComponentEvent and has these constructors:
  FocusEvent(Component src, int type)
  FocusEvent(Component src, int type, boolean tempflag)
  FocusEvent(Component src, int type, boolean tempflag, Component other)
- Here,src is ref to the component that generated this event. The type of the event is specified by type. The argument tempflag is set to true if the focus event is temporary. Otherwise, it is set to false.
- The other component involved in the focus change, called the opposite component, is passed in other. Therefore, if a FOCUS_GAINED event occurred, other will refer to the component that lost focus. Conversely, if a FOCUS_LOST

event occurred, other will refer to the component that gains focus.

InputEvent

- InputEvent is a subclass of ComponentEvent and is the superclass for component input events.
- Its subclasses are KeyEvent and MouseEvent.
- The InputEvent class defines seven int constants that can be used to obtain information about any modifiers associated with this event.
- These are ALT_MASK,BUTTON1_MASK,BUTTON2_MASK,BUTTON3_MASK,CTRL_MASK,META_MASK,SHIFT_MASK.
- The isAltDown(), isControlDown(), isMetaDown(), and isShiftDown() methods test if these modifiers were pressed at the time the event was generated. The forms of these methods are shown here:
  boolean isAltDown(), boolean isControlDown()
  boolean isMetaDown(), boolean isShiftDown()

ItemEvent

- An ItemEvent is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.
- There are two types of item events, which are identified by the following integer constants:
  DESELECTED = the user deselected an item
  SELECTED = the user selected an item.
- ItemEvent(ItemSelectable src, int type,Object entry, int state)

here, src is a ref. to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by type. The specific item that generated the item event is passed in entry. The current state of that item is in state.

KeyEvent

- A KeyEvent is generated when keyboard input occurs.
- There are 3 types of key events, which are identified by these integer constants: KEY_PRESSED, KEY_RELEASED, and KEY_TYPED.
- The first two events are generated when an key is pressed or released. The last event occurs only when a character is generated.

MouseEvent

- There are eight types of mouse events. The MouseEvent class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	User clicked the mouse.
MOUSE_DRAGGED	User dragged the mouse.
MOUSE_ENTERED	Mouse entered a component.
MOUSE_EXITED	Mouse exited form a component.
MOUSE_MOVED	Mouse moved
MOUSE_PRESSED	Mouse was pressed
MOUSE_RELEASED	Mouse was released
MOUSE_WHEEL	Mouse wheel was moved

- MouseEvent is a subclass of InputEvent.

MouseEvent(Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean triggersPopup)

- Here, src is a reference to the component that generate dthe event. The type of the event is specified by type. The system time at which the mouse event occurred is passed in when. The modifiers argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in x and y. the click count is passed in clicks. The triggersPopup flag indicates if this event causes a pop-up mentu to appear on this platform.
- The most commonly used methods in this class are getX() and getY(). These return the X and Y coordinates of the mouse when the event occurred. Their forms are shown here:
  - Int getX()
  - Int getY()
- getPoint() is used to obtain the oordiantes of the mouse.

Point getPoint()

it returns a Point object that contains the X,Y coordinates in its integer members: x and y.

- The translatePoint() changes the location of the event.

void translatePoint(int x, int y)

here, the arguments x and y are added to the coordinates of the event.

- The getClickCount() method obtains the number of mouse clicks for this event.

int getClickCount()

MouseWheelEvent

- The mouseWheelEvent class encapsulates a mouse wheel event. It is a subclass of MouseEvent.
- MouseWheelEvent defiens two integer constants:

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

MouseWheelEvent(Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean triggersPopup, int scrollHow, int count)
Here, src is a ref. to the object that generated the event. The type of the event is specified by type. The system time at which the mouse event occurred is passed in when. The modifiers argumnt indicats which modifiers were pressed when the event occurred. The coordinates of the mouse are passed in x and y. the number of clicks the weeel has rotated is passed in clicks. The triggersPopup flag indicates if this event causes a pop-up menu to appear on this platform. The scrollHow value must be either WHEEL_UNIT_SCROLL or WHEEL_BLOCK_SCROLL. The no. of units to scroll is passed in amount. The count parameter indicates the number of rotational units that the wheel moved.

TextEvent

- Instance of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program.
- TextEvent defiens the integer constant TEXT_VALUE_CHANGED.
- TextEvent(Object src, int type)
  here, src is a ref. to the object that generated this event. The type of the event is specified by type.

WindowEvent

- There are ten types of window events.

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	Window has been closed.

WINDOW_CLOSING	User requested that the window be closed.
WINDOW_DEACTIVATED	Window was deactivated.
WINDOW_DEICONIFIED	Window was deiconified.
WINDOW_GAINED_FOCUS	Window gained input focus.
WINDOW_ICONIFIED	Window was iconified.
WINDOW_LOST_FOCUS	Window lost input focus.
WINDOW_OPENED	Window was opened.
WINDOW_STATE_CHANGED	The state of the window changed

- WindowEvent is subclass of ComponentEvent. It defiens several constructors:

WindowEvent(window src, int type)
Here, src is a ref. to the object that generated this event. The type of the event is type.
WindowEvent(Window src, int type, Window other)
WindowEvent(Window src, int type, int fromstate, int to state)
WindowEvent(Window src, int type, Window other, int fromstate, int to state)
Here, other specifies the opposite window when a focus event occurs. The fromState specifies the prior state of the window, and toState specifies the new state that the window will have when a windwo state change occurs.

- getWindow() returns the Window object that generated the event.

Sources of Events

Button	Generates action events when the button is pressed.
Checkbox	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.

List	Generates action events when an item is double-clicked; Generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scrollbar	Generates adjustment events when the scroll bar is manipulated.
Text Component	Generates text events when the user enters a character.
Window	Generates widow events when a window is activated, closed, deactivated, deiconified, iconified,opened or quit.

Event Listener Interface

- The delegation event model has two parts: sources and listeners.
- Listeners are created by implementing one or more of the interfaces defined by the java.awt.event package.
- When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment event.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
keyListener	Defines three methods to recognize when a key is pressed, released, or typed.

MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to rexognize when a window is activated, closed, deactivated, deicnified, iconified, opened ,or quit.

ActionListener Interface

- This interface defines the actionPerformed() method that is invoked when an action event occurs. Its general form is shown here:
  void actionPerformed(ActionEvent ae)

AdjustmentListener Interface

- This interface defines the adjustmentValueChanged() method that is invoked when an adjustment event occurs. Its general form is:
  Void adjustmentValueChanged(AdjustmentEvent ae)

ComponentListener Interface

- This interface defines four methods that are invoked when a component is resized, moved, shown or hidden. Their general forms are shown here:
  void componentResized(ComponentEvent ce)
  void componentMoved(ComponentEvent ce)
  void componentShown(ComponentEvent ce)
  void componentHidden(ComponentEvent ce)

ContainerListener Interface

- This interface contains two methods. When a component is added to a container, componentAdded() is invoked. When a component is removed from a container, componentRemoved() is invoked.
  void componentAdded(ContainerEvent ce)

void componentRemoved(ContainerEvent ce)

FocusListener Interface

- This interface defines two mehtods. When a component obtains keyboard focus, focusGained() is invoked. When a component loses keyboard focus, focusLost() is called. Their general form is shown here:
  void focusGained(FocusEvent fe)
  void focusLost(FocusEvent fe)

ItemListener Interface

- This interface defines the itemStateChanged() method that is invoked when the state of an item changes. Its general form is shown here:
  void itemStateChanged(ItemEvent ie)

KeyListener Interface

- This interface defines three methods. The keyPressed() and keyReleased() method are invoked when a key is pressed and released, respectively. The keyTyped() method is invoked when a character has been entered.
- For example, if a user presses and releases the A key, three events are generated in sequence key pressed, typed and released. If a user presses and releases the HOME key, two key events are generated in sequence keypressed and released.
  void keyPressed(KeyEvent ke)
  void keyReleased(KeyEvent ke)
  void keyTyped(KeyEvent ke)

MouseListener Interface

- This interface defines five methods. If the mouse is pressed and released at the same point, mouseClicked() is invoked. When the mouse enters a component, the mouseEntered() method is called. When it leaves, mouseExited() is called. The mousePressed() and mouseReleased() methods are invoked when the mouse is pressed and released, respectively.
  void mouseClicked(MouseEvent me)
  void mouseEntered(MouseEvent me)
  void mouseExited(MouseEvent me)
  void mousePressed(MouseEvent me)
  void mouseReleased(MouseEvent me)

MouseMotionListener Interface

- This interface defines two methods. The mouseDragged() method is called multiple times as the mouse is dragged. The

mouseMoved() method is called multiple times as the mouse is moved. Their general form:
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)

MouseWheelListener Interface

- This interface defines the mouseWheelMoved() method that is invoked when the mouse wheel is moved. Is general form is shown here:
  void mouseWheelMoved(MouseWheelEvent me)

TextListener Interface

- This interface defines the textChanged() method that is invoked when a change occurs in a text area or text field. Its general form is shown here:
  void textChanged(TextEvent te)

WindowFocusListener Interface

- This interface defines two methods: windowGainedFocus() and windowLostFocus(). These are called when a window gains or loses input focus. Their general forms are shown here:
  void windowGainedFocus(WindowEvent we)
  void windowLostFocus(WindowEvent we)

WindowListener Interface

- This interface defines seven methods. The windowActivated() and windowDeactivated() methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the windowIconified() method is called. When a windwo is opened or clsoed, the windwoOpened() or windwoClosed() methods are called, respectively. The windowClosing() mehtod is called when a window is being closed.

  void windowActivated(WindowEvent we)
  void windowClosed(WindowEvent we)
  void windowClosing(WindowEvent we)
  void windowDeactivated(WindowEvent we)
  void windowDeiconified(WindowEvent we)
  void windowIconofied(WindowEvent we)
  void windowOpened(WindowEvent we)

Chapter-6 Applets

- Applet class……
- Applet life cycle…….
- Applet context class…….
- Applet tag ……..
- Passing parameter to object…….
- Use of awt.graphics class and various method..

Applet

- Applet is a Java program that is embedded in an HTML document and runs in the context of a Java-capable browser.
- The Applet class is contained in the java.applet package. Applet contains several methods that give you detailed control over the execution of your applet.
- Java.applet also defines 3 interfaces: AppletContext, AudioClip and AppletStub.
- All applets are subclasses of Applet. Thus, all applets must import java.applet. Applet must also import java.awt (abstract window toolkit). Since all applets run in a window, it is necessary to include support for that window.
- Applets are not executed by the console based java run-time interpreter. Rather they are executed by either a web browser or an applet viewer called appletviewer, provided by the JDK.
- Once an applet has been compiled, it is included in an HTML file using APPLET tag. The applet will be executed by a java-enabled web-browser when it encounters the APPLET tag within the HTML file.
- To view and test an applet more conveniently, simple include a comment at the head of your java source code file that contains the APPLET tag. This way, your code is documented with the necessary HTML statements needed by your applet and you can test the compiled applet by starting the applet viewer with your java source code file specified as the target.
- Applet code uses the services of two classes, Applet and Graphics from the java class library. The Applet class which is contained in the java.applet package provides life and behaviour to the applet through its methods such as init(), start(), and paint(). Unlike with applications where java calls the main() method directly to initiate the execution of the program, when an applet is loaded, java automatically calls a series of Applet class methods form starting, running and stopping the applet code. The Applet class therefore maintains the lifecycle of an applet.
- The paint() method of the Applet class, when it is called, actually displays the result of the applet code on the screen. The output may be text, graphics, or sound. The paint()

method, which requires a Graphics object as an argument, is defined as follows:

- Public void paint(Graphics g)

- This requires that the applet code imports the java.awt package that contains the Graphics class. All output operations of an applet are performed using the methods defined in the Graphics class.
- import java.applet.Applet;

```
import java.awt.Graphics;
/*
<applet code="FirstApplet" width=200 height=200>
</applet>
*/
public class FirstApplet extends Applet
{
	public void paint(Graphics g)
	{
		g.drawString("This is my first applet.",20,100);
	}
}
```

Compilation of Applet program

- Javac FirstApplet.java
- Appletviewer FirstApplet.java
- The 1st and 2nd lines import the java.applet.Applet and java.awt.Graphics classes. Applet class is the superclass of all applets. The Graphics class is provided by the awt.
- The next four lines define a java comment. That comment is HTML source code. The applet tag specifies which class contains the code for this applet.
- It also defines the width and height in pixels of the display area. Applet viewer reads the HTML and interprets the info that is contained between the applet tags.
- The HTML is not part of the applet and is not used by a web browser. It is used only by the applet viewer.
- The next line declares that firstApplet extends Applet. Each applet that you create must extends this class.
- Paint() is responsible for generating the output of the applet. It accepts a Graphics object as its one argument. It is automatically invoked whenever the applet needs to be displayed.
- The actual output of the string is done by calling the drawstring() method of the Graphics object. x,y coordinate – at which to begin the string.

- Applet viewer executes an applet by using the HTML source code. Width and height attributes of the applet tag define the dimensions of the applet display area.
- Applet may also be executed by a web browser.

```
import java.applet.Applet;
import java.awt.Graphics;
public class SecondApplet extends Applet
{
        public void paint(Graphics g)
        {
                g.drawString("This is my 2nd applet.",20,100);
        }
}
```

Javac SecondApplet.java

```
<html>
<body>
<applet code="SecondApplet.java" width=200 height=200>
</applet>
</body>
</html>
```

SecondApplet.html

- On browser run the file SecondApplet.html

Applet Life Cycle

- Every java applet inherits a set of default behaviors from the Applet class. As a result, when an applet is loaded, it undergoes a series of changes in its state as shown in fig. The applet states include:

1. Born and initialization state
2. Running state
3. Idle state
4. Dead or Destroyed state

Initialization state

- Applet enters the initialization state when it is first loaded. This is achieved by calling the init() method of Applet class. The applet is born. At this stage we may do the following if required:
    1. Create objects needed by the applet
    2. Set up initial values
    3. Load images or fonts
    4. Set up colors
- The initialization occurs only once in the applet's life cycle. To provide any of the behaviors mentioned above, we must override the init() method.
  Public void init()

```
{
        Action
}
```

Running state

- Applet enters the running state when the system calls the start() method of Applet class.
- This occurs automatically after the applet is initialized.
- Starting can also occur if the applet is already in "stopped" (idle) state.
- For example, we may leave the web page containing the applet temporarily to another page and return back to the page. This again starts the applet running. Unlike init() method, the start() method may be called more than once. We may override the start() method to create a thread to control the applet.

```
Public void start()
{
        Action
}
```

Idle or Stopped state

- An applet becomes idle when it is stopped from running stopping occurs automatically when we leave the page containing the currently running applet. We can also do so by calling the stop() method explicitly. If we use a thread to run the applet, then we must use stop() method to terminate the thread. We can achieve by overriding the stop() method.

```
Public void stop()
{
        Action
}
```

Dead state

- An applet is said to be dead when it is removed from memory. This occurs automatically by invoking the destroy() method when we quit the browser. Like initialization, destroying stage occurs only once in the applet's life cycle. If the applet has created any resources, like threads, we may override the destroy() method to clean up these resources.

```
Public void destroy()
{
        Action
}
```

Display state

- Applet moves to the display state whenever it has to perform some output operations on the screen. This happens

immediately after the applet enters into the running state. The paint() method is called to accomplish this task. Almost every applet will have a paint() method. Like other methods in the life cycle, the default version of paint() method does absolutely nothing. We must therefore override this method if we want anything to be displayed on the screen.

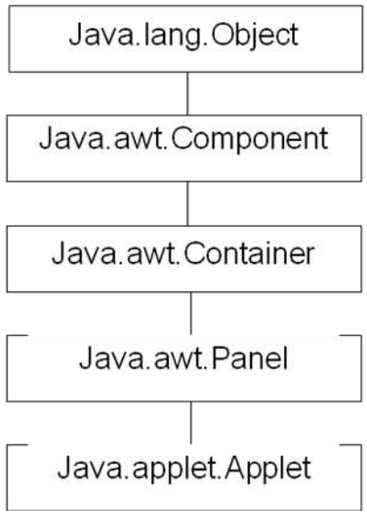
```
Public void paint( Graphics g)
{
        Display statement;
}
```

- It is to be noted that the display state is not considered as a part of the applet's life cycle. In fact, the paint() method is defined in the Applet class. It is inherited form the component class, a super class of Applet.

```
import java.applet.Applet;
import java.awt.Graphics;
/*<applet code="AppletLifeCycle" width=300 height=50>
</applet>
*/
public class AppletLifeCycle extends Applet
{
        String str="";
        public void init()
        {
                str+="init";
        }
        public void start()
        {
                str+="start";
        }
        public void stop()
        {
                str+="stop";
        }
        public void destroy()
        {
                System.out.println("destroy");
        }
        public void paint(Graphics g)
        {
                g.drawString(str,10,25);
        }
}
```

Applet class

- An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application.



- The Applet class must be the superclass of any applet that is to be embedded in a Web page or viewed by the Java Applet Viewer. The Applet class provides a standard interface between applets and their environment.

Method	Description
Void destroy()	Destroys this applet.
AppletContext getAppletContext()	Returns the applet context.
URL getCodeBase()	Returns the code base.
URL getDocumentBase()	Returns the document base.
Image getImage(URL url)	Returns an image object for the image at url.
Image getImage(URL url, String imgName)	Returns an Image object for the image named imgName relative to url.
String getParameter(String pName)	Returns the value of parameter pName.
Void init()	Initializes this applet.
Void showStatus(String str)	Displays str on the status line.
Void start()	Starts this applet.
Void stop()	Stops this applet.

```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
```

```
/*
<applet code="BackgroundForeground" width=300 height=100>
</applet>
*/
public class BackgroundForeground extends Applet
{
        public void paint(Graphics g)
        {
                setBackground(Color.yellow);
                setForeground(Color.blue);
                g.drawLine(0,0,200,200);
                g.fillRect(100,40,50,50);
        }
}
```

The AppletContext class

- The java.applet.AppletContext interface defines methods that allow an applet to interact with the context (or environment) in which it is executing. This context is provided by either a tool such as the appletviewer or a web browser.
- Methods defined by this interface are:

Method	Description
Applet getApplet(String appName)	Returns the applet named appName.
Enumeration getApplets()	Returns an enumeration of the applets in the current context.
AudioClip getAudioClip(URL url)	Returns an AudioClip object for the audio clip at url.
Image getImage(URL url)	Returns an image object for the image at url.
Void showDocument(URL url)	Retrieves and shows the document at url.
Void showDocument(URL url, String target)	Retrieves the document at url and displays it in target.
Void showStatus(String str)	Displays str in the status line.

- The firstform of showDocument() causes the web browser to retrieve and display the web page identified by url. The second form of showDocument() allows you to specify where this web page is displayed. The argument target may be "_self"(show in current frame), "_parent"(show in parent

frame), “_top”(show in top frame), and “_blank” (show in a new browser window). It may also equal the name of a frame.

- Many browsers allow you to divide their display area into frames. These are rectangular regions that can independently display different URLs. A web page can use the <frameset> tag to define how its display areas are to be divided into frames. A simple example of this technique is shown in the following listing. The display is divided into two columns. The left and right columns are 25 percent and 75 percent of the browser window, respectively.

```
<frameset cols=”25%,75%”>
<frame name=”left” src=”left.html”>
<frame name=”right” src=”right.html”>
</frameset>
```

- Framesets provide additional functionality. For example, you can specify a fixed width in pixels for a frameset or define nested framesets. Consult other texts for more information.
- The AppletContext interface is implemented by the applet viewer. However, these methods do not have the same functionality as you would find in a web browser environment. For example, if you use the showDocument() method in an applet and then execute that applet with the applet viewer, you will find that this method has no effect.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
/*
<applet code="ShowDocument" width=200 height=50>
</applet>
*/
public class ShowDocument extends Applet
{
	public void init()
	{
		AppletContext ac=getAppletContext();
		try
		{
			URL url=new URL("http://www.osborne.com");
			ac.showDocument(url,"frame2");
		}
		catch(Exception e)
		{
			showStatus("Exception:"+e);
```

```
            }
        }
        public void paint(Graphics g)
        {
            g.drawString("show document applet",10,25);
        }
}
```

Graphics Class

- A Graphics object encapsulates a set of methods that can perform graphics output. Specifically, it allows you to draw lines, ovals, rectangles, strings, images, characters, and arcs.
- Methods of Graphics class are:

Method	Description
Void drawArc(int x, int y, int w, int h, int degree0, int degrees1)	Draws an arc between degrees0 and degrees1. the center of the arc is the center of a rectangle with upper-left corner at coordinates x and y, width w and height h. zero degrees is at position 3pm on a watch. The angle increases in a counterclockwise direction.
Void drawImage(Image img, int x, int y, ImageObserver io)	Draws the image img so its upper-left corner is at x,y. Updates about the progress of this activity
Void drawLine(int x0,int y0,int x1, int y1)	Draws a line between the points at x0,y0 and x1,y1.
Void drawOval(int x, int y, int w, int h)	Draws an oval. The center of the shape is the center of a rectangle with upper-left corner at coordinates x and y, width w, and height h.
Void drawPolygon(int x[], int y[], int n)	Draws a polygon with n corners. The coordinates are given by the elements of x and y. the first and last points are automatically connected.
Void drawPolyline(int x[], int y[], int n)	Draws a polyline with n points. The coordinates are given by the elements of x and y. the first and last points are not automatically connected.

Void drawRect(int x, int y, int w, int h)	Draws a rectangle with upper-left corner at coordinates x and y width w and height h.
Void drawString(String str, int x, int y)	Draws str at location x,y.
Void fillArc(int x, int y, int w, int h, int degrees0, int degrees1)	Fills an arc between degrees0 and degrees1. the center of the arc is the center of a rectangle with upper-left corner at coordinates x and y, width w, and height h. zero degrees is at position 3pm on a watch.
Void fillOval(int x, int y, int w, int h)	Fills an oval. The center of the shape is the center of a rectangle with upper-left corner at coordinates x and y, width w, and height h.
Void fillPolygon(int x[], int y[], int n)	Fills a polygon with n corners. The coordinates are given by the elements of x and y.
Void fillRect(int x, int y, int w, int h)	Fills a rectangle with upper-left corner at coordinates x and y, width w and height h.
Color getColor()	Gets the color of the current object.
Font getFont()	Gets the font of the current object.
FontMetrics getFontMetrics()	Gets the font metrics of the current object.

```
import java.applet.Applet;
import java.awt.Graphics;
/*
<applet code="DrawShapes" width=400 height=400>
</applet>
*/
public class DrawShapes extends Applet
{
	public void paint(Graphics g)
	{
		g.drawArc(20,20,160,160,0,135);
		g.drawOval(50,50,60,30);
		g.drawString("All shapes",10,15);
		g.fillRect(100,100,70,40);
	}
```

```
}
import java.applet.Applet;
import java.awt.Graphics;
/*
<applet code="DrawPolygon" width=400 height=400>
</applet>
*/
public class DrawPolygon extends Applet
{
	public void paint(Graphics g)
	{
		int n=5;
		int x[]=new int[n];
		int y[]=new int[n];
		x[0]=10;
		y[0]=100;
		x[1]=60;
		y[1]=10;
		x[2]=70;
		y[2]=140;
		x[3]=140;
		y[3]=90;
		x[4]=190;
		y[4]=10;
		g.drawPolygon(x,y,n);
	}
}
```

Using Colors

- The java.awt.Color class is used to work with colors. Each instance of this class represents a particular color. With the help of this class we can draw colored strings, lines, and shapes in an applet. It is possible to fill a shape such as an oval or rectangle with a color. We can also set the background and foreground colors of an applet.
- This class has following constructors:

Color(int red, int green, int blue)
Color(int rgb)
Color(float r, float g, float b)
Color c = new Color(255, 255, 240); this.setBackground(c);

- Here, red, green, and blue are int values that range between 0 and 255, inclusive. The argument rgb contains an encoding of a color in which the red, green, and blue components are specified in bits 23 to 16, 15 to 8 and 7 to 0, respectively. Finally, r,g, and b are float values that range between 0.0 and 1.0f inclusive.

- The Color class also defines several constants that represent specific colors. These are black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange pink, red, white, and yellow.
- Table shows some of the methods defined by Color class:

Method	Description
Static int HSBtoRGB(float h, float s, float b)	Returns an int encoding of a color whose hue, saturation, and brightness are specified are specified by h, s and b. these values range between 0.0f to 1.0f. Hue is the color. Its values represent red, orange, yellow, green, blue, indigo and violet as the hue varies from 0.0f to 1.0f. saturation represents the purity of the color. Brightness indicates the strength of the color. A brightness value of zero is black.
Static float[] RGBtoHSB(int r, int g, int b, float hsb[])	Returns an array of float elements with the hue, saturation, and brightness values for the color whose red, green, and blue components are specified by r, g and b. the argument hsb is the array in which these values are stored. A new array is allocated if hsb is null.
Color brighter()	Returns a brighter version of the current object.
Color darker()	Returns a darker version of the current object.
Static Color decode(String str) throws NumberFormatException	Returns a Color object corresponding to str. This argument must contain the decimal, octal or hex encoding of a color.
Boolean equals(Object obj)	Returns true if the current object and obj represent the same color value.
Int getBlue()	Returns the blue component of the

	current object.
Int getGreen()	Returns the green component of the current object.
Int getRGB()	Returns an int encoding of the current object.
Int getRed()	Returns the red component of the current object.

```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
/*
<applet code="BlueString" width=300 height=100>
</applet>
*/
public class BlueString extends Applet
{
	public void paint(Graphics g)
	{
		g.setColor(Color.blue);
		g.drawString("Blue string",100,50);
	}
}

import java.applet.Applet;
import java.awt.*;
/*
<applet code="ColorBars" width=300 height=300>
</applet>
*/
public class ColorBars extends Applet
{
	Color colors[]={Color.black, Color.blue, Color.cyan,
Color.darkGray,
Color.gray, Color.green,
Color.lightGray,Color.magenta, Color.orange, Color.pink,
Color.red,
Color.white, Color.yellow};
public void paint(Graphics g)
{
	int bar=300/colors.length;
	for(int i=0;i<colors.length;i++)
	{
		g.setColor(colors[i]);
		g.fillRect(i*bar,0,(i+1)*bar,300);
	}
```

```
}
}

import java.applet.Applet;
import java.awt.*;
/*
<applet code="ColorTriangle" width=400 height=300>
</applet>
*/
public class ColorTriangle extends Applet
{

	public void paint(Graphics g)
	{
		int n=3;
		int xdata[]=new int[n];
		int ydata[]=new int[n];
		xdata[0]=50;
		ydata[0]=150;
		xdata[1]=200;
		ydata[1]=50;
		xdata[2]=350;
		ydata[2]=150;
		int rgb=Color.HSBtoRGB(2.3f,1.2f,1.0f);
		g.setColor(new Color(rgb));
		g.fillPolygon(xdata,ydata,n);
	}
}
```

Displaying text

- We have seen that drawstring() method of the Graphics class is used to display text. We can also display strings by using different fonts. This is necessary in may applets. For example, tables and graphs use various fonts for titles, axes, data and other information.

java.awt.Font

- A font determines the size and appearance of characters in a string. Information about a font is encapsulated by the java.awt.Font class.
- The following is one of its constructors:
  Font(String name, int style, int ps)
  Here, name identifies the font. Some commonly used font names are Serif, SansSerif, and Monospaced. The style may be BOLD, ITALIC, or PLAIN. The point size of the font is ps.

- After a font has been created, you may then use it in a graphics context. This is done by calling the setFont() method of the Grahics class. This method has the following format:
- Void setFont(Font font)
- Here, font is a Font object. after this method is called, any strings that are output via the drawstring() method are displayed with that font.
- Java.awt.FontMetrics
- The FontMetrics class defines a font metrics object, which encapsulates information about the rendering of a particular font on a particular screen. It gets the dimension of the text.
- The java.awt.FontMetrics class allows you to get several metrics about the size of a font. In addition, you may also determine the size of a string that is displayed in that font. These quantities are provided in pixels. You will see that they are necessary to calculate the position at which to draw a string in an applet.
- The specific metrics that are available are the ascent, descent, leading and height. When a string is displayed, all of its characters are aligned to a horizontal baseline. Characters extend above and below that line. The number of pixels above the baseline is the ascent. The number of pixels below the baseline is the descent. The number of pixels between the descent of one line and the ascent of the next line is the leading. The sum of the ascent, descent, and leading is the height.
- The one constructor for this class is
  FontMetrics(Font font)
  Here, font indicates the font for which metrics are wanted.

Some of the methods of this class are:

Method	Description
Int charWidth(char c)	Returns the width of c.
Int charWidth(int i)	Returns the width of the character in the lowest 16 bits of i.
Int getAscent()	Returns the ascent.
Int getDescent()	Returns the descent.
Int getHeight()	Returns the height
Int getLeading()	Returns the leading
Int stringWidth(String	Returns width of str.

str)	

- To find out how many pixels some text will take on the screen you need FontMetrics.stringWidth(String). The trick is you cannot use the FontMetrics(Font f) constructor to get your FontMetrics object. You must use the Component.getFontMetrics(Font f) or the Graphics.getFontMetrics(Font f) method. Unfortunately, you can't get the FontMetrics object from the Font class alone, and then cache it along with the Font. For measuring height, FontMetrics is quite simple FontMetrics. getAscent, getHeight and getDescent are only approximate, and include a of lot of white space. getHeight even includes the suggested leading to the next line.
- For more accuracy, you need LineMetrics with takes a sample string of text and a FontRenderingContext. Even it so, it still include a lot of white space. I don't know if there is a way to get totally tight bounding box around some text, without rolling your own pixel based methods. LineMetrics is awkward to use. To get a dummy Graphics for FontMetrics, you could try Component.getGraphics(). If you are inside a paintComponent method, you can cast the Graphics object to a Graphics2D. If there is no GUI, you could create a dummy Graphics2D context like this:

```
import java.applet.Applet;
import java.awt.*;
/*
<applet code="FontDemo" width=200 height=200>
</applet>
*/
public class FontDemo extends Applet
{
	public void paint(Graphics g)
	{
		int baseline=100;
		g.setColor(Color.blue);
		g.drawLine(0,baseline,200,baseline);
		g.setFont(new Font("serif",Font.BOLD,36));
		g.setColor(Color.black);
		g.drawString("hello",5,baseline);
	}
}

import java.applet.Applet;
import java.awt.*;
/*
<applet code="FontMetricsDemo" width=200 height=200>
```

```
</applet>
*/
public class FontMetricsDemo extends Applet
{
	public void paint(Graphics g)
	{
		int baseline=100;
		g.setColor(Color.blue);
		g.drawLine(0,baseline,300,baseline);
		//draw string
		Font font=new Font("serif",Font.BOLD,36);
		g.setFont(font);
		g.setColor(Color.black);
		g.drawString("hello",5,baseline);
		g.setColor(Color.blue);
		//get fontMetrics
		FontMetrics fm=g.getFontMetrics(font);
		//draw line at baseline-ascent
		int ascent=fm.getAscent();
		int y=baseline-ascent;
		g.drawLine(0,y,300,y);
	}
}
```

Applet Dimensions

- We can display strings, lines and shapes in an applet by using the drawing methods of the Graphics class. Each of these methods required arguments to indicate where the output should be positioned.
- We can dynamically determine the dimensions of an applet. The data can then be used to calculate the arguments that should be passed to the drawing methods of the Graphics class. For example, you can display a circle at the center of an applet. If the applet is resized, the circle remains positioned at its center.
- The getSize() method is used to determine the size of an applet. It has the form shown here:
  Dimension getSize()
- A Dimension object encapsulates a width and height. The following are some of its constructors:
  Dimension(Dimension d)
  Dimension(int w, int h)
- Here, d is a Dimension object. The arguments w and h represent the width and height in pixels. The class has two instance variables width and height of type int.

```
import java.applet.Applet;
import java.awt.*;
```

```
/*
<applet code="DimensionDemo" width=200 height=200>
</applet>
*/
public class DimensionDemo extends Applet
{
	public void paint(Graphics g)
	{
		Dimension d=getSize();
		int x=d.width/2;
		int y=d.height/2;
		int radius=(int) ((d.width<d.height) ?
0.4*d.width:0.4*d.height);

		g.drawOval(x-radius, y-radius,2*radius,2*radius);
	}
}
```

Applet in a web page

- The applet viewer uses the HTML source code at the beginning of a .java file to display an applet. It is also possible to embed an applet in a web page. You can then supply the URL for that web page to a browser and it presents the applet. The complete syntax for an applet tag is shown in the following listing. Optional lines are enclosed in brackets.

```
<applet
[codebase=url]
Code=clsName
[alt=text]
[name=appName]
Width=wpixels
Height=hpixels
[align=alignment]
[vspace=vspixels]
[hspace=hspixels]
>
[<param name=pname1 value=value1>]
[<param name=pname2 value=value2>]
…
[<param name=pnameN value=valueN>]
</applet>
```

- The second line in this listing defines an optional parameter known as code base. This is the location form which the .class files for the applet are retrieved. You can assign a URL to code base. However, if code base is not specified in the <applet> tag, the .class files for the applet are retrieved form

the same location where the HTML document was obtained. That location is known as document base.

- The code parameter of the <applet> tag is required. It is assigned the name of the applet class, clsName. Browsers that cannot support the name of the applet class, clsName. Browsers that cannot support applets use text as an alternate representation of this applet. Each instance of an applet on a web page may be assigned a unique name shown above as appName.
- The width and height of the applet must be specified in wpixels and hpixels.
- The alignmet of the applet may have a value of LEFT,RIGHT,TOP,BOTTOM,MIDDLE,BASELINE,TEXT TOP,ABSMIDDLE OR ABSBOTTOM. These constants have the same meaning as they do in HTML when used to align images. The vertical and horizontal spacing around the applet may be specified as vspixels and hspixels.
- Parameters may be passed to an applet via a series of param tags. In the syntax shown above, the names of the paramenters are pname1 through pnameN. The values of these parameters are value1 through valueN.
- The following web page is retrieved by specifying the URL http://host1/page.html. the HTML file and photo1.jpg are retrieced form host1. however, the file Example1.class and any other .class files are retrieved form host2. the code base is the machine from which the .class files for the applet are retrieved. The document base is the machine form which all other files are retrieved.

```
<img src=”photo1.jpg” width=200 height=200>
<br>
<applet code=”example1” codebase=”host2” width=300
height=300>
</applet>
```

```
import java.applet.*;
import java.awt.*;
public class ParamTest extends Applet {
  public void paint(Graphics g) {
    String myFont   = getParameter("font");
    String myString = getParameter("string");
    int mySize      = Integer.parseInt(getParameter
("size"));
    Font f = new Font(myFont, Font.BOLD, mySize);
    g.setFont(f);
    g.setColor(Color.red);
```

```
    g.drawString(myString, 20, 20);
  }
}
```

```
<HTML>
<HEAD>
<TITLE>Passing parameters to Java applets</TITLE>
</HEAD>
<BODY>
<APPLET CODE="ParamTest.class" WIDTH="400"
HEIGHT="50">
   <PARAM NAME="font"    VALUE="Chiller">
   <PARAM NAME="size"    VALUE="24">
   <PARAM NAME="string"  VALUE="Hello, world ... it's me.
:)">
</APPLET>
</BODY>
</HTML>
```

Chapter-7 Input/Output

- Streams……
- Character streams and Byte streams……
- Character stream(Reader,Writer,FileReader FileWriter,BufferedReader,BufferedWriter, PrintWriter)……
- ByteStream(InputStream,OutputStream,FileI/O Stream,BufferedI\OStream, Data I/O Stream, Object I/O Stream,PrintStream) …….
- Random Access file , StringTokenizer……...

Streams

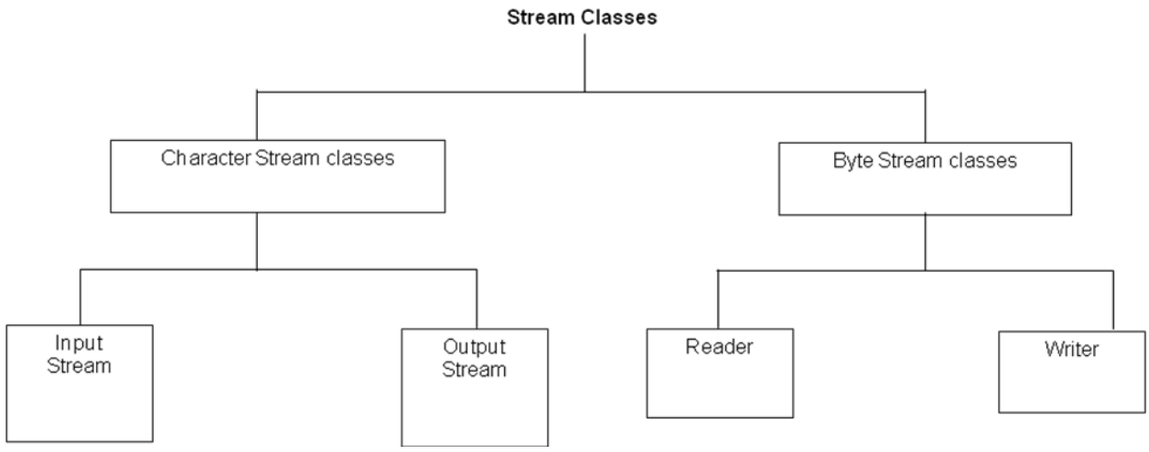
- A stream is an abstraction for a source or destination of data. (A stream is an abstraction that either produces or consumes information.)
- A stream is linked to a physical device by the Java I/O system.
- It enables you to use the same techniques to interface with different types of physical devices. For example, an input stream may read its data from a keyboard, file or memory buffer. An output stream may write its data to a monitor, file or memory buffer. – other types of devices may also be used as the source or destination for a stream.
- There are two types of streams:
    1. character Streams
    2. byte Streams

1. character Streams

- It allows you to read and write characters and strings.
- An input character Stream converts bytes to Character.
- An output character Stream converts character to byte.

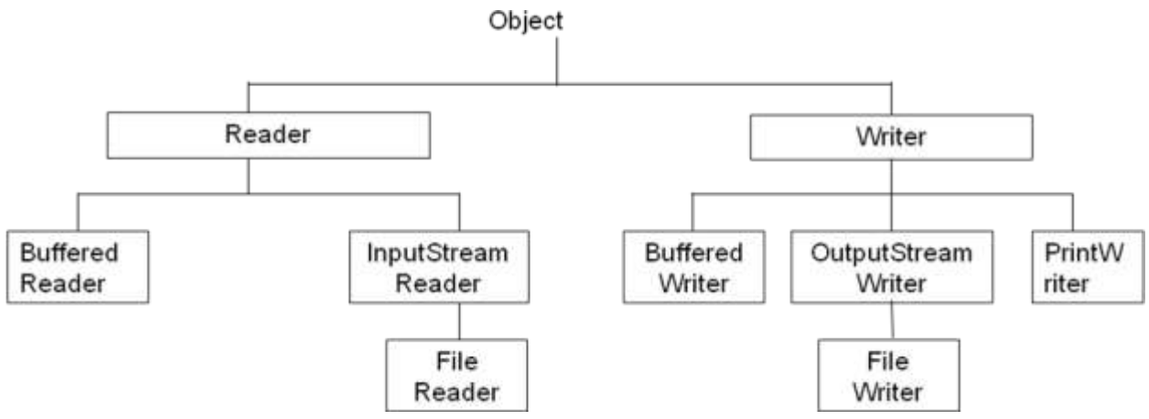
2. Byte Streams

- It allows you to read and write binary data.
- For ex, an application that simulates the behavior of an electric circuit can write a sequence of float values to a file. These would represent the value of a signal over a time interval.



Character Stream Classes

- Character stream allows you to read and write characters and strings.
- An input character stream converts bytes to characters.
- An output character stream converts characters to bytes.
- Java internally represents characters according to the 16-bit Unicode encoding. However, this may not be the encoding used on a specific machine.
- Character streams translate between these two formats.
- Ability to translate between Unicode and other encodings is an important feature because it enables you to write programs that operate correctly for an international marketplace.
- Fig shows a few of the character streams provided by the java.io package.



Some of the character stream classes

Reader

- Reader stream classes are designed to read characters from the file.

- The abstract Reader class defines the functionality that is available for all character input streams.
- Reader is an abstract class and therefore, we cannot create an instance of this class. Rather, we must use the subclasses that inherit form the Reader class. i.e. Buffered Reader and Input Stream Reader.
- Reader stream classes are designed to read characters from the file.
- The abstract Reader class defines the functionality that is available for all character input streams.
- Reader is an abstract class and therefore, we cannot create an instance of this class. Rather, we must use the subclasses that inherit form the Reader class. i.e. Buffered Reader and Input Stream Reader.

Methods of Reader class

Method	Description
Void close()	Closes the input stream. Further read attempts generate an IOException. Must be implemented by a subclass.
Void mark(int numChars)	Places a mark at the current point in the input stream that will remain valid until numChars characters are read.
Boolean markSupported()	Returns true if mark()/reset() are supported in this stream
Int read()	Reads a character from the stream. Waits until data is available.
Int read(char buffer[])	Attempts to read up to buffer-length characters into buffer and returns the actual number of characters that were successfully read. Waits until data is available.
Int read(char buffer[], int offset, int numChars)	Attempts to read up to numChars characters into buffer starting at buffer[offset] and returns the actual number of characters that were successfully read. Waits until data is available.
Boolean ready	Returns true if the next read() will not wait.
Void reset()	Resets the input pointer to the previously set mark.

Int skip(long numChars)	Skips over numChars bytes of input returning the number of characters actually skipped.

Input Stream Reader

- The Input Stream Reader class extends Reader. It converts a stream of bytes to a stream of characters. This is done according to the rules of a specific character encoding.
- The constructors provided by this class are as follows:
  InputStreamReader(InputStream is)
  InputStreamReader(InputStream is, String encoding)
  Where:
  Is= input stream
  Encoding = name of character encoding.
- The first form of the constructor uses the default character encoding of the user's machine.
- getEncoding() – returns the name of the character encoding. It has following syntax:
- String getEncoding()

File Reader

- The File Reader class extends InputStreamReader and inputs characters form a file.
- Its two most common constructors are:
  FileReader(String filepath)
  FileReader(File fileObj)
- Either can throw a FileNotFoundException.
- The program that reads a file is shown in the following listing. It accepts one command-line argument that is the name of the file to read. A FileReader object is created. Individual characters are obtained via read() and displayed via System.out.print(). The streakm is closed when all characters have been read.

```
import java.io.*;
class FileReaderDemo
{
	public static void main(String args[])
	{
		try
		{
			//create a file reader
			FileReader fr=new FileReader(args[0]);

			//read and display characters
			int i;
			while((i=fr.read())!=-1)
```

```
                {
                        System.out.println((char)i);
                }

                //close file reader
                fr.close();
        }
        catch(Exception e)
        {
                System.out.println("Exception:"+e);
        }
    }
}
```

Writer

- The writer stream classes are designed to perform all output operation on files.
- The writer class is an abstract class which acts as a base class for all the other writer stream classes.
- This base class provides support for all output operations by defining methods that are identical to those in output stream class.

Methods of Writer class

Method	Description
Void close()	Closes the output stream.
Void flush()	Writes any buffered data to the physical device represented by that stream.
Void write(int c)	Writes the lower 16 bits of c to the stream.
Void write(char buffer[])	Writes the characters in buffer to the stream.
Void write(char buffer[], int index, int size)	Writes size characters form buffer starting at position index to the stream.
Void write(String s)	Writes s to the stream.
Void write(String s, int index, int size)	Writes size characters form s starting at position index to the stream.

Output Stream Writer

- The output Stream Writer class extends writer.
- It converts a stream of characters to a stream of bytes.
- This is done according to the rules of a specific character encoding.

- Its constructors are like this:
  OutputStreamWriter(OutputStream os)
  OutputStreamWriter(OutputStream os, String encoding)
- Here, os is the output stream and encoding is the name of a character encoding. The first form of the constructor uses the default character encoding of the user's machine.
- The getEncoding() method returns the name of he character encoding. It has this syntax:
  String getEncoding()

File Writer

- The file writer class extends OuptputStreamWriter and outputs characters to a file.
- Its constructors are as follows:
  FileWriter(String filepath) throws IOException
  FileWriter(String filepath, Boolean append) throws IOException
  FileWriter(File fileObj) throws IOExcepiton
- The program that writes file is shown in the following listing. It accepts one command-line argument that is the name of the file to create. Twelve strings are written to the file by using the write() method of FileWriter.

```
import java.io.*;
class FileWriterDemo
{
	public static void main(String args[])
	{
		try
		{
			//create a file writer
			FileWriter fw=new FileWriter(args[0]);

			//write string to file
			for(int i=0;i<12;i++)
			{
				fw.write("Line" +i +"\n");
			}

			//close file writer
			fw.close();
		}
		catch(Exception e)
		{
			System.out.println("Exception:"+e);
		}
	}
}
```

Buffered Character Streams

- There are two classes BufferedWriter and BufferedReader.
- The advantage of buffering is that the number of reads and writes to a physical device is reduce. This improves performance.

BufferedWriter

- The BufferWriter class extends Writer and buffers output to a character stream. Its constructors are as follows:
  BufferedWriter(Writer w)
  BufferedWriter(Writer w, int bufSize)
- The first form creates a buffered stream using a buffer with a default size.
- In second, the size of the buffer is specified by bufsize.
- This class implements all of the methods defined by Writer. In addition, it provides the newLine() method to output a line separator. Its signature is shown below:
  Void newLine() throws IOException

The program that writes a file is shown in the following listing. It accepts one command-line argument that is the name of the file to create. A FileWriter object is created and passed as the argument to the BufferedWriter constructor. Twelve strings are written to the file by using the write() method of BufferedWriter.

```
import java.io.*;
class BufferedWriterDemo
{
	public static void main(String args[])
	{
		try
		{
			//create a file writer
			FileWriter fw= new FileWriter(args[0]);

			//create a buffered writer
			BufferedWriter bw= new BufferedWriter(fw);
			//write strings to the file
			for(int i=0;i<12;i++)
			{
				bw.write("Line "+i + "\n");
			}

			//close buffered writer
			bw.close();
		}
		catch(Exception e)
		{
```

```
                System.out.println("Exception:"+e);
            }
        }
}
```

BufferedReader

- The BufferedReader class extends Reader and buffers input form a character stream. Its constructors are as follows:
  BufferedReader(Reader r)
  BufferedReader(Reader r, int bufSize)
- The first form creates a buffered stream using a buffer with a default size.
- In second, the size of the buffer is specified by bufsize.
- This class implements all of the functionality defined by Reader. In addition, the readLine() method reads newline-terminated strings form a character stream. Its signature is:
  String readLine() throws IOException
- The program that reads a file is shown in the following program. It accepts one command-line argument that is the name of the file to red. A FileReader object is created and passed as the argument to the BufferedReader constructor. The readLine() method is used to obtain the individual lines in the file. Note that readLine() discards the newline character it reds. The file reader is closed after all lines have been displayed.

```
import java.io.*;
class BufferedReaderDemo
{
    public static void main(String args[])
    {
        try
        {
            //create a file reader
            FileReader fr=new FileReader(args[0]);
            //create a buffered reader
            BufferedReader br=new BufferedReader(fr);

            //send and display lines from file
            String s;
            while((s=br.readLine())!=null)
            System.out.println(s);

            //close file reader
            fr.close();
        }
        catch(Exception e)
```

```
		{
			System.out.println("Exception:"+e);
		}
	}
}
```

- This example shows how to use a buffered character stream to read input form the keyboard. The program executes an infinite loop that reads a string and displays the number of characters it contains. Each string must be terminated by a newline character.
- System.in is passed as the argument to the InputStreamReader constructor. This is done because System.in is an InputStream. The InputStreamReader object is then passed as the argument to the BufferedReader constructor.

```
import java.io.*;
class ReadConsole
{
	public static void main(String args[])
	{
		try
		{
			//create an input stream reader
			InputStreamReader isr=new InputStreamReader
(System.in);
			//create a buffered reader
			BufferedReader br=new BufferedReader(isr);
			//read and process lines from console
			String s;
			while((s=br.readLine())!=null)
			{
				System.out.println(s.length());
			}

			//close inputstream reader
			isr.close();

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

The PrintWriter class

- The PrintWriter class extends Writer and display string equivalents of simple types such as int, float, char, and objects.
- Its functionality is valuable because it provides a common interface by which many different data types can be output.
- This class has these four constructors:
  PrintWriter(OutputStream outputStream)
  PrintWriter(OutputStream outputStream, Boolean flushOnNewline)
  PrintWriter(Writer writer)
  PrintWriter(Writer writer, Boolean flushOnNewline)
- Here, flushOnNewLine controls whether Java flushes the output stream every time a newline (‘\n’) character is output. If flushOnNewline is true, flushing automatically takes place. If false, flushing is not automatic.
- The first and third constructors do not automatically flush.
- Java’s PrintWriter objects support the print() and println() methods for all types including Object. If an argument is not a simple type, the PrintWrier methods will call the object’s toString() method and then display the string that is returned from this method.

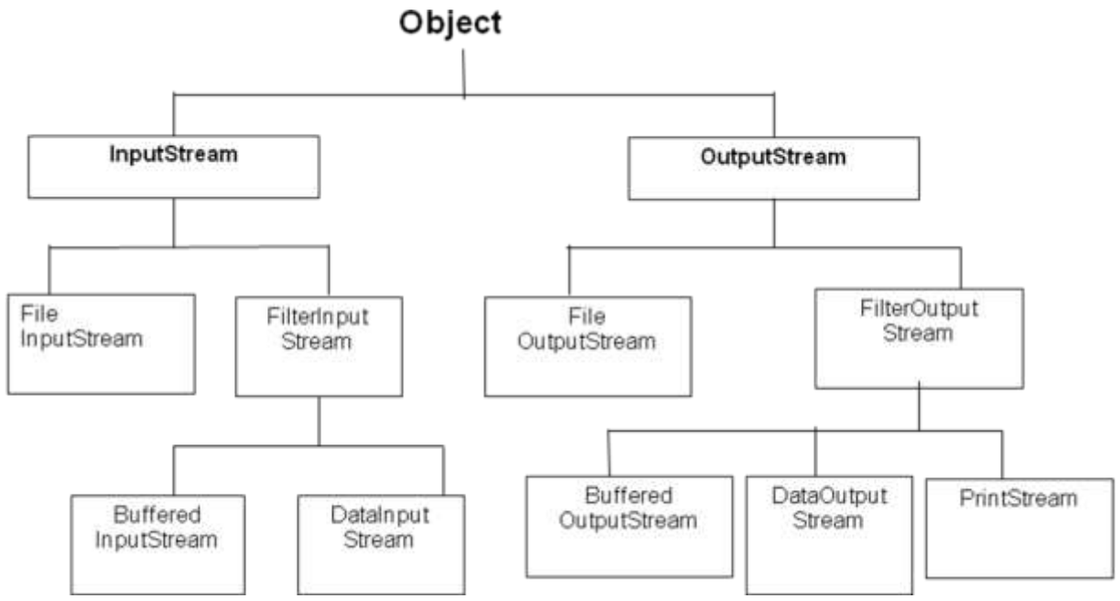
```
import java.io.*;
class PrintWriterDemo
{
	public static void main(String args[])
	{
		try
		{
			//create a print writer
			PrintWriter pw= new PrintWriter(System.out);
			//Experiment with some methods
			pw.println(true);
			pw.println('a');
			pw.println(500);
			pw.println(40000L);
			pw.println(45.67f);
			pw.println(45.67);
			pw.println("Hello");
			pw.println(new Integer("99"));
			// close print writer
			pw.close();
		}
		catch(Exception e)
		{
```

```
                System.out.println("Exception:"+e);
            }
        }
}
```

Byte Streams

- Byte streams allow a programmer to work with the binary data in a file.
- Figure shows a few of the byte streams provided by the java.io package. It includes OutputStream, FileOutputStream, DataOutputStream, and PrintStream classes that are used for output, while the InputStream, FileInputStream, FilterInputStream, BufferedINputStream, and DataInputStream classes are used for input.
- The OutputStream class defines the functionality that is available for all byte output streams.
- The following table summarizes the methods provided by this class.

Method	Description
Void close() Throws IOException	Closes the output stream.
Void flush() Throws IOException	Flushes the output stream.
Void write(int i) Throws IOException	Writes lowest-order 8 bits of I to the stream.
Void write(byte buffer[]) throws IOException	Writes buffer to the stream.
Void write(byte buffer[], int index, int size) Throws IOException	Writes size bytes form buffer starting at position index to the stream.



InputStream

- The abstract InputStream class defines the functionality that is available for all byte output streams.
- The methods provided by the InputStream class are:

Method	Description
Int available()	Returns the number of bytes currently available for reading.
Void close()	Closes the input stream.
Void mark(int numBytes)	Places a mark at the current point in the input stream. It remains valid until numBytes are read.
Boolean markSupported()	Returns true if mark()/reset() are supported. Otherwise, returns false.
Int read()	Reads one byte form the input stream.
Int read(byte buffer[])	Attempts to read up to buffer length bytes into buffer and returns the actual number of bytes that were successfully read.
Int read(byte buffer[], int offset, intnumBytes)	Attempts to read up to numBytes bytes into buffer starting at buffer[offset]. Returns the number of bytes successfully read.
Void reset()	Resets the input pointer to the previously set mark.
Int skip(long numBytes)	Skips numBytes of input. Returns the number of bytes actually skipped.

FileInputStream

- The fileInputStream class extends INputStream and allows you to read binary data form a file. Its most commonly used constructors are as follows:
  FileInputStream(String filepath) throws
FileNotFoundException
  FileInputStream(File fileObj) throws
FileNotFoundExcepiton
- Here, filepath is the full path name of a file and fileObj is a File object athat describes the file.

FilterInputStream

- The FilterInputStream class extends InputStream and filters an input stream. It provides this constructor:
  FilterInputStream(InputStream is)

Here, is is the input stream to be filtered.

- You do not directly instantiate FilterInputStream. Instead, you must create a subclass to implement the desired functionality.

BufferedInputStream

- The BufferedInputStream class extends FilterInputStream and buffers input form a byte stream. Its constructors are as follows:
  BufferedInputStream(InputStream is)
  BufferedInputStream(InputStream is, int bufSize)
- The first argument to both constructors is a reference to the input stream. The first form creates a buffered stream by using a buffer with a default size.
- In second, the size of the is specified by bufSize.

DataInputStream

- The DataInputStream class extends FilterInputStream and implements DataInput. It allows you to read the simple java types form a byte input stream. This class provides this constructor:
  DataInputStream(InputStream is)

Here, is is the input stream.

DataInput

- The DataInput interface defines methods that can be used to read the simple java types form a byte inpt stream.
- The following methods defined by this interface. All of these can throw an IOException.

Methods of DataInput

Method	Description
Boolean readBoolean()	Reads and returns a Boolean from

	stream.
Byte readByte()	Reads and returns byte from the stream.
Char readChar()	Reads and returns a char from the stream.
Double readDouble()	Reads and returns a double from the stream.
Float readFloat()	Reads and returns a float from the stream.
Void readFully(byte buffer[])	Reads bytes and fills buffer.
Void readFully(byte buffer[], int index, int size)	Reads size bytes and fills buffer starting at position index.
Int readInt()	Reads and return an int form the stream.
Long readLong()	Reads and returns a long from the stream.
Short readShort()	Reads and returns a short form the stream.
String readUTF()	Reads a string form the input. Characters are converted form UTF-8 format to Unicode. The string is returned form the method.
Int readUnsignedByte()	Reads and returns an unsigned byte from the stream.
Int readUnsignedShort()	Reads and returns an unsigned short form the stream.
Int skipBytes(int n)	Skips ahead n bytes in the stream.

- The program that reads a file is shown in the following listing. It accepts one cmdline argument that is the name of the file to read. A FileInputStream object is created. Bytes are obtained via read() and displayed via System.out.println(). The stream is closed when all bytes have been read.

```
import java.io.*;
class FileInputStreamDemo
{
        public static void main(String args[])
```

```
	{
		try
		{
			//create a file input stream
			FileInputStream fis= new
FileInputStream(args[0]);
			//read and display data
			int i;
			while((i=fis.read())!=-1)
			{
				System.out.println(i);
			}
			//close file input stream
			fis.close();
		}
		catch(Exception e)
		{
			System.out.println("Exception:"+e);
		}
	}
}
```

- The program that reads a file is shown in the following listing. It accepts one command-line argument that is the name of the file to read. A DataInputStream object is created. The various data types are obtained by calling some of its methods. The stream is closed when all data has been read.

```
import java.io.*;
class DataInputStreamDemo
{
	public static void main(String args[])
	{
		try
		{
			//create a file Input stream
			FileInputStream fis=new
FileInputStream(args[0]);
			//create a data Input stream
			DataInputStream dis= new DataInputStream(fis);
			//write various types of data
			System.out.println(dis.readBoolean());
			System.out.println(dis.readByte());
			System.out.println(dis.readChar());
			System.out.println(dis.readDouble());
			System.out.println(dis.readFloat());
			System.out.println(dis.readInt());
			System.out.println(dis.readLong());
```

```
                System.out.println(dis.readShort());

                //close buffered output stream
                fis.close();
            }
            catch(Exception e)
            {
                System.out.println("Exception:"+e);
            }
        }
}
```

OutputStream

- The OutputStream class defines the functionality that is available for all byte output streams.
- The methods provided by this class is as follows:

Method	Description
Void close() throws IOException	Closes the output stream.
Void flush() throws IOException	Flushes the output stream.
Void write(int i) throws IOException	Writes lowest-order 8 bits of I to the stream.
Void write(byte buffer[]) throws IOExcecption	Writes buffer to the stream.
Void write(byte buffer[], int index, int size) Throws IOException	Writes size bytes form buffer starting at position index to the stream.

FileOutputStream

- The FileOutputStream class extends OutputStream and allows you to write binary data to a file. Its most commonly used constructors are as follows:
  FileOutputStream(String filepath) throws IOException
  FileOutputStream(String filepath, Boolean append) throws IOException
  FileOutputStream(File fileObj) throws IOException
  Here, filepath is the full path name of a file and fileObj is a File object that describes the file. If append is true, characters are appended to the end of the file. Otherwise, the existing contents of the file are overwritten.

FilterOutputStream

- The FilterOutputStream class extends OutputStream. It is used to filter output and provides this constructor:
  FilterOutputStream(OutputStream os)

Here, os is the output stream to be filtered.

- You do not directly instantiate FilterOutputStream. Instead, you must create a subclass to implement the desired functionality.

BufferedOutputStream

- The bufferedOutputStream class extends FilterOutputStream and buffers output to a byte stream. Its constructors are as follows:
  BufferedOutputStream(OutputStream os)
  BufferedOutputStream(OutputStream os, int bufSize)
- The first argument to both constructors is a reference to the output stream. The first form creates buffered stream by using a buffer with a default size.
- In the second, the size of the buffer is specified by bufSize.

DataOutputStream

- The DataOutputStream class extends FilterOutputStream and implements DataOutput. It allows you to write the simple java types to a byte output stream. The class provides this constructor:
  DataOutputStream(OutputStream os)
  Here, os is output stream.

DataOutput

- The DataOutput interface defines methods that can be used to write the simple java types to a byte output stream.
- Following table shows the methods provided by the DataOutput interface. All of these can throw an IOException.

Method	Description
Void write(int i)	Writes I to the stream.
Void write(byte buffer[])	Writes buffer to the stream.
Void write(byte buffer[], int index, int size)	Writes size bytes from buffer starting at position index to the stream.
Void writeBoolean(Boolean b)	Writes b to the stream.
Void writeByte(int i)	Writes lowest-order 8 bits of I to the stream.
Void writeBytes(String s)	Writes s to the stream.
Void writeChar(int i)	Writes lowest-order 16 bits of I to the stream.
Void writeChars(String s)	Writes s to the stream.

Void writeDouble(double d)	Writes d to the stream.
Void writeFloat(float f)	Writes f to the stream.
Void writeInt(int i)	Writes I to the stream.
Void writeLong(long l)	Writes l to the stream.
Void writeShort(short s)	Writes s to the stream.
Void writeUTF(String s)	Writes s to the stream. Characters are converted from Unicode to UTF-8 encoding.

PrintStream

- The PrintStream class extends FilterOutputStream and provides all of the formatting capabilities we have been using from System.out since the beginning of the book. The static System.out variable is a PrintStream.
- PrintStream has these two constructor:
  PrintStream(OutputStream outputStream)
  PrintStream(OutputStream outputStream, Boolean flushOnNewline)
  Here, flushOnNewline controls whether java flushes the output stream every time a newline (‘\n’) character is output. If flushONNewline is true, flushing automatically takes place. If false, flushing is not automatic. The first constructor does not automatically flush.
- Java’s PrintStream objects support the print() and println() methods for all types including Object. If an argument is not a simple type, the PrintStream methods will call the object’s toString() method and then print out the result.
- The program that writes to a file is shown in the following listing. It accepts one command-line argument that is the name of the file to create. Twelve bytes are written to the file by using the write() method of FileOutputStream.

```
import java.io.*;
class FileOutputStreamDemo
{
	public static void main(String args[])
	{
		try
		{
			//create a file output stream
			FileOutputStream fos=new
FileOutputStream(args[0]);
			//write 12 bytes to the file
			for(int i=0;i<12;i++)
			{
				fos.write(i);
```

```
                }
                //close file output stream
                fos.close();
        }
        catch(Exception e)
        {
                System.out.println("Exception:"+e);
        }
    }
}
```

- The program that writes to a file is shown in the following listing. It accepts one command-line argument that is the name of the file to create.

```
import java.io.*;
class BufferedOutputStreamDemo
{
    public static void main(String args[])
    {
        try
        {
                //create a file output stream
                FileOutputStream fos=new
FileOutputStream(args[0]);
                //create a buffered output stream
                BufferedOutputStream bos=new
BufferedOutputStream
(fos);
                //write 12 bytes to the file
                for(int i=0;i<12;i++)
                {
                    bos.write(i);
                }
                //close buffered output stream
                bos.close();
        }
        catch(Exception e)
        {
                System.out.println("Exception:"+e);
        }
    }
}
```

- The program that writes to a file is shown in the following listing. It accepts one command-line argument that is the name of the file to create. Various types of data are written to the file by using the methods of DataOutputStream.

```
import java.io.*;
class DataOutputStreamDemo
{
	public static void main(String args[])
	{
		try
		{
			//create a file output stream
			FileOutputStream fos=new
FileOutputStream(args[0]);
			//create a data output stream
			DataOutputStream dos= new
DataOutputStream(fos);
			//write various types of data
			dos.writeBoolean(false);
			dos.writeByte(Byte.MAX_VALUE);
			dos.writeChar('A');
			dos.writeDouble(Double.MAX_VALUE);
			dos.writeFloat(Float.MAX_VALUE);
			dos.writeInt(Integer.MAX_VALUE);
			dos.writeLong(Long.MAX_VALUE);
			dos.writeShort(Short.MAX_VALUE)

			//close buffered output stream
			fos.close();
		}
		catch(Exception e)
		{
			System.out.println("Exception:"+e);
		}
	}
}
```

Random Access Files

- The stream classes examined in the previous sections can only use sequential access to read and write data in a file.
- The RandomAccessFile class allows you to write programs that can seek to any location in a file and read or write data at that point. This type of functionality is very valuable in some programs. For example, it can be used to manage a set of data records that are stored in a file.
- This class implements the DataInput and DataOutput interfaces. It also provides the methods summarized in the following table:

Method	Description

Void close()	Close the file.
Long getFilePointer()	Returns the current position of the file pointer. This identifies the point at which the next byte is read or written.
Long length()	Returns the number of bytes in the file.
Int read()	Reads and returns a byte from the file. Waits until data is available.
Int read(byte buffer[], int index, int size)	Attempts to read size bytes from the file and places these in buffer starting at position index. Returns the number of bytes actually read. Waits until data is available.
Int read(byte buffer[])	Reads bytes form the file and places these in buffer. Returns the number of bytes read. Waits until data is available.
Void seek(long n)	Positions the file pointer at n bytes form the beginning of the file. The next read or write occurs at this position.
Int skipBytes(int n)	Adds n to the file pointer. Returns the actual number of bytes skipped. If n is negative, no bytes are skipped.

- This example displays the last N bytes of a file. The first command-line argument is the filename, the second is the number of bytes. A RandomAccessFile object is created for the file. The seek() method positions the file pointer. The program then enters a loop in which readByte() is used to obtain bytes from the file. These are displayed via print().

```
import java.io.*;
class Tail
{
	public static void main(String args[])
	{
		try
		{
			//create random access file
			RandomAccessFile raf=new
RandomAccessFile(args[0],"r");
			//determine no of bytes to display at end of file
			long count=Long.valueOf(args[1]).longValue();

			//determine file length
			long position=raf.length();
			//seek to the correct position
```

```
                position-=count;
                if(position<0)
                        position=0;
                raf.seek(position);

                //read and display the bytes
                while(true)
                {
                        //read byte
                        try
                        {
                        byte b=raf.readByte();
                        //display as character
                        System.out.println((char)b);
                        }
                        catch(EOFException eofe)
                        {
                                break;
                        }
                }
        }
        catch(Exception e)
        {
                e.printStackTrace();
        }
    }
}
```

- Javac Tail.java
- Java Tail Tail.java 40

The Streamtokenizer class

- The StreamTokenizer class parses the data form a character input stream and generates a sequence of tokens.
- A token is a group of characters that represent a number or word. This functionality can be very valuable if you need to build parsers, compilers, or any program that processes character input.
- A constructor for this class is as follows:
  StreamTokenizer(Reader r)
  Here, r is a reader.
- The class defines four constants. TT_EOF and TT_EOL indicate end-of-file and end-of-line conditions, respectively. TT_NUMBER and TT_WORD indicate that a number or word has been read.
- Three instance variables provide valuable information.

- If the current token is a number, nval contains its value and ttype equals TT_NUMBER.
- If the current token is a string, sval contains its value and ttype equals TT_WORD. Otherwise, ttype contains the character that has been read.
- The general procedure to use a stream tokenizer is as follows:

1. Create a StreamTokenizer object for a Reader.
2. Define how characters are to be processed.
3. Call nextToken() to obtain the next token.
4. Read the ttype instance variable to determine the token type.
5. Read the value of the token form the sval, nval or ttype instance variable.
6. Process the token.
7. Repeat steps 3-6 until nextToken() returns StreamTokenizer.TT_EOF.

Method	Description
Void commentChar(int ch)	Indicates that ch starts a single-line comment.
Void eollsSignificant(boolean flag)	If flag is true, the end-of-line is treated as a token. Otherwise, it is treated as white space.
Int lineno()	Returns the current line number.
Void lowerCaseMode(Boolean flag)	If flag is true, the token is automatically converted to lowercase. Otherwise, it is not.
Int nextToken()	Returns TT_NUMBER if the next token is a number. Returns TT_WORD if the next token is word. Otherwise, returns the character that equals the next token.
Void ordinaryChar(int ch)	Specifies that ch should be treated as an ordinary character.
Void parseNumbers()	Specifies that numbers should be parsed.
Void pushback()	Pushes the current token back to the input stream.
Void quoteChar(int ch)	Defines ch as the quote character

	for string literals.
Void resetSyntax()	Specifies that all characters should be treated as ordinary characters.
Void slashSlashComments(Boolean flag)	If flag is true,// comments are ignored. Otherwise, they are not.
Void slashSlashComments(Boolean flag)	If flag is true, /* …*/ comments are ignored. Otherwise, they are not.
String toString()	Returns the string equivalent of the current token.
Void whitespaceChars(int c1, int c2)	Specifies that all characters in the range c1-c2 should be treated as white space.
Void wordChars(int c1, int c2)	Specifies that all characters in the range c1-c2 should be treated as word characters.

```
import java.io.*;
class StreamTokenizerDemo
{
	public static void main(String args[])
	{
		try
		{
		//create a file reader
		FileReader fr=new FileReader(args[0]);

		//create a buffered reader
		BufferedReader br=new BufferedReader(fr);

		//create a stream tokenizer
		StreamTokenizer s=new StreamTokenizer(br);
		//define period as ordinary character
		//s.ordinaryChar('.');
		//define apostrophe as word character
		//s.wordChars('\'','\'');
		//process tokens
		while(s.nextToken()!= StreamTokenizer.TT_EOF)
		{
			if(s.ttype==s.TT_WORD)
				System.out.println(s.sval);
```

```
                else if(s.ttype==s.TT_NUMBER)
                        System.out.println(s.nval);

                else
                        System.out.println((char)s.ttype);
                }

                //close file reader
                fr.close();
                }

        catch(Exception e)
        {
                System.out.println("Exception:"+e);
        }

        }
}
```

Chapter-8 Swing

- LayoutManager (FlowLayout,BorderLayout,CardLayout, BoxLayout, GridLayout,GridBagLayout,GroupLayout, SpringLayout)
- GUI with Swing,
- Container class,

- JApplet, Jlabel,JButton, JCheckBox,JChoice, JTextField ,JTextArea, JList,Jscrollbar,Jpannel, JFrame,JMenu,JMenuItem , JPassword, JRadioButton.

Layout manager:

Layout manager is a interface of AWT,not a class. The layout manager is notified each time you add a component to panel.

Whenever a panel need to be resized the layput manager is consulted via some layout method.each component that is being managed by a layout should properly implement the preferred size and minimum size method. These should return the comfortable and minimum size required to paint each component respectively. Java has some predefined layoutmanager, described here.

FlowLayout:

The flowlayout implements a simple layput style, which is similar to how words flow in a text editor. Component are laid out from the upper left corner,left to right and top to bottom. When no more component fit on a line, the next one appears on the next line. A small space is left between each component.

BorderLayout:

The border layout implements a common layout style for top level windows, which has four narrow , fixed-width component at the edges, and one large area in the center that grows and shrink in two dimensions to take up the slack. Each of these area is named with string. North,South,East,West represent the four sides, and center is the middle area.

Grid Layout:

The gridlayout creates an initiative layout of a simple uniform grid of components.
The constructor is used to define the number of row and coluns in the panel layout.

CardLayout:

The cardLayout is unique from the other layout manager in that it represents several different layouts that can be thought of as living on separate index cards that can be suffeled so that any one card is one top at a given time.this can be useful for interface that have optional comoponent which can be dynamically enabled and disabled upon user input.

Swing

In Part II, you saw how to build user interfaces with the AWT classes. Here, we will

take a tour of a supercharged alternative called Swing. Swing is a set of classes that
provides more powerful and flexible components than are possible with the AWT.
In addition to the familiar components, such as buttons, check boxes, and labels, Swing
supplies several exciting additions, including tabbed panes, scroll panes, trees, and
tables. Even familiar components such as buttons have more capabilities in Swing. For
example, a button may have both an image and a text string associated with it. Also,
the image can be changed as the state of the button changes.
Unlike AWT components, Swing components are not implemented by
platform-specific code. Instead, they are written entirely in Java and, therefore, are
platform-independent. The term lightweight is used to describe such elements.
The number of classes and interfaces in the Swing packages is substantial, and this
chapter provides an overview of just a few. Swing is an area that you will want to
explore further on your own.
The Swing component classes that are used in this book are shown here:
Class Description
AbstractButton Abstract superclass for Swing buttons.
ButtonGroup Encapsulates a mutually exclusive set of buttons.
ImageIcon Encapsulates an icon.
JApplet The Swing version of Applet.
JButton The Swing push button class.
JCheckBox The Swing check box class.
JComboBox Encapsulates a combo box (an combination of a drop-down list and text field).
JLabel The Swing version of a label.
JRadioButton The Swing version of a radio button.
JScrollPane Encapsulates a scrollable window.
JTabbedPane Encapsulates a tabbed window.
JTable Encapsulates a table-based control.
JTextField The Swing version of a text field.
JTree Encapsulates a tree-based control.

The Swing-related classes are contained in javax.swing and its subpackages, such

as javax.swing.tree. Many other Swing-related classes and interfaces exist that are not
examined in this chapter.
The remainder of this chapter examines various Swing components and illustrates
them through sample applets.

JApplet

Fundamental to Swing is the JApplet class, which extends Applet. Applets that use
Swing must be subclasses of JApplet. JApplet is rich with functionality that is not
found in Applet. For example, JApplet supports various “panes,” such as the content
pane, the glass pane, and the root pane. For the examples in this chapter, we will not be
using most of JApplet’s enhanced features. However, one difference between Applet
and JApplet is important to this discussion, because it is used by the sample applets in
this chapter. When adding a component to an instance of JApplet, do not invoke the
add( ) method of the applet. Instead, call add( ) for the content pane of the JApplet
object. The content pane can be obtained via the method shown here:
Container getContentPane( )
The add( ) method of Container can be used to add a component to a content pane.
Its form is shown here:
void add(comp)
Here, comp is the component to be added to the content pane.

Icons and Labels

In Swing, icons are encapsulated by the ImageIcon class, which paints an icon from an
image. Two of its constructors are shown here:
ImageIcon(String filename)
ImageIcon(URL url)
The first form uses the image in the file named filename. The second form uses the

image in the resource identified by url.
The ImageIcon class implements the Icon interface that declares the methods
shown here:
Method Description
int getIconHeight( ) Returns the height of the icon
in pixels.
int getIconWidth( ) Returns the width of the icon
in pixels.
void paintIcon(Component comp, Graphics g,
int x, int y)
Paints the icon at position x, y on
the graphics context g. Additional
information about the paint
operation can be provided in comp.
Swing labels are instances of the JLabel class, which extends JComponent. It can
display text and/or an icon. Some of its constructors are shown here:
JLabel(Icon i)
Label(String s)
JLabel(String s, Icon i, int align)
Here, s and i are the text and icon used for the label. The align argument is either LEFT,RIGHT, CENTER, LEADING, or TRAILING. These constants are defined in theSwingConstants interface, along with several others used by the Swing classes.
The icon and text associated with the label can be read and written by the
following methods:
Icon getIcon( )
String getText( )
void setIcon(Icon i)
void setText(String s)
Here, i and s are the icon and text, respectively.
The following example illustrates how to create and display a label containing both
an icon and a string. The applet begins by getting its content pane. Next, an ImageIcon
object is created for the file france.gif. This is used as the second argument to the
JLabel constructor.
The first and last arguments for the JLabel constructor are the label text and the alignment. Finally, the label is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
```

```
/*


<applet code="JLabelDemo" width=250 height=150>
</applet>
*/
public class JLabelDemo extends JApplet {
public void init() {
// Get content pane
Container contentPane = getContentPane();
// Create an icon
ImageIcon ii = new ImageIcon("france.gif");
// Create a label
JLabel jl = new JLabel("France", ii, JLabel.CENTER);
// Add label to the content pane
contentPane.add(jl);
}
}
```

Output from this applet is shown here:

Text Fields

The Swing text field is encapsulated by the JTextComponent class, which extends
JComponent. It provides functionality that is common to Swing text components. One
of its subclasses is JTextField, which allows you to edit one line of text. Some of its
constructors are shown here:

```
JTextField( )
JTextField(int cols)
JTextField(String s, int cols)
JTextField(String s)
```

Here, s is the string to be presented, and cols is the number of columns in the text field.
The following example illustrates how to create a text field. The applet begins by
getting its content pane, and then a flow layout is assigned as its layout manager. Next,
a JTextField object is created and is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
```

```
public class JTextFieldDemo extends JApplet {
JTextField jtf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add text field to content pane
jtf = new JTextField(15);
contentPane.add(jtf);
}
}
```

Buttons

Swing buttons provide features that are not found in the Button class defined by the
AWT. For example, you can associate an icon with a Swing button. Swing buttons are
subclasses of the AbstractButton class, which extends JComponent. AbstractButton
contains many methods that allow you to control the behavior of buttons, check boxes,
and radio buttons. For example, you can define different icons that are displayed for
the component when it is disabled, pressed, or selected. Another icon can be used as a
rollover icon, which is displayed when the mouse is positioned over that component.
The following are the methods that control this behavior:
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)
Here, di, pi, si, and ri are the icons to be used for these different conditions.
The text associated with a button can be read and written via the following methods:
String getText( )
void setText(String s)
Here, s is the text to be associated with the button.
Concrete subclasses of AbstractButton generate action events when they are
pressed. Listeners register and unregister for these events via the methods shown here:
void addActionListener(ActionListener al)
void removeActionListener(ActionListener al)

Here, al is the action listener.
AbstractButton is a superclass for push buttons, check boxes, and radio buttons.
Each is examined next.
The JButton Class
The JButton class provides the functionality of a push button. JButton allows an icon,
a string, or both to be associated with the push button. Some of its constructors are
shown here:
JButton(Icon i)
JButton(String s)
JButton(String s, Icon i)
Here, s and i are the string and icon used for the button.
The following example displays four push buttons and a text field. Each button
displays an icon that represents the flag of a country. When a button is pressed, the
name of that country is displayed in the text field. The applet begins by getting its
content pane and setting the layout manager of that pane. Four image buttons are
created and added to the content pane. Next, the applet is registered to receive action
events that are generated by the buttons. A text field is then created and added to the
applet. Finally, a handler for action events displays the command string that is
associated with the button. The text field is used to present this string.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=300>
</applet>
*/
public class JButtonDemo extends JApplet
implements ActionListener {
JTextField jtf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add buttons to content pane
ImageIcon france = new ImageIcon("france.gif");
```

```
JButton jb = new JButton(france);
jb.setActionCommand("France");
jb.addActionListener(this);
contentPane.add(jb);
ImageIcon germany = new ImageIcon("germany.gif");
jb = new JButton(germany);
jb.setActionCommand("Germany");
jb.addActionListener(this);
contentPane.add(jb);
ImageIcon italy = new ImageIcon("italy.gif");
jb = new JButton(italy);
jb.setActionCommand("Italy");
jb.addActionListener(this);
contentPane.add(jb);
ImageIcon japan = new ImageIcon("japan.gif");
jb = new JButton(japan);
jb.setActionCommand("Japan");
jb.addActionListener(this);
contentPane.add(jb);
// Add text field to content pane
jtf = new JTextField(15);
contentPane.add(jtf);
}
public void actionPerformed(ActionEvent ae) {
jtf.setText(ae.getActionCommand());
}
}
```

Check Boxes

The JCheckBox class, which provides the functionality of a check box, is a concrete
implementation of AbstractButton. Its immediate superclass is JToggleButton, which
provides support for two-state buttons. Some of its constructors are shown here:

```
JCheckBox(Icon i)
JCheckBox(Icon i, boolean state)
JCheckBox(String s)
JCheckBox(String s, boolean state)
JCheckBox(String s, Icon i)
JCheckBox(String s, Icon i, boolean state)
```

Here, i is the icon for the button. The text is specified by s. If state is true, the check box
is initially selected. Otherwise, it is not.

The state of the check box can be changed via the following method:
void setSelected(boolean state)
Here, state is true if the check box should be checked.
The following example illustrates how to create an applet that displays four check
boxes and a text field. When a check box is pressed, its text is displayed in the text field.
The content pane for the JApplet object is obtained, and a flow layout is assigned as its
layout manager. Next, four check boxes are added to the content pane, and icons are
assigned for the normal, rollover, and selected states. The applet is then registered to
receive item events. Finally, a text field is added to the content pane.
When a check box is selected or deselected, an item event is generated. This is
handled by itemStateChanged( ). Inside itemStateChanged( ), the getItem( ) method
gets the JCheckBox object that generated the event. The getText( ) method gets the text
for that check box and uses it to set the text inside the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo extends JApplet
implements ItemListener {
JTextField jtf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Create icons
ImageIcon normal = new ImageIcon("normal.gif");
ImageIcon rollover = new ImageIcon("rollover.gif");
ImageIcon selected = new ImageIcon("selected.gif");
// Add check boxes to the content pane
JCheckBox cb = new JCheckBox("C", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
```

```
contentPane.add(cb);
cb = new JCheckBox("C++", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Java", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Perl", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
// Add text field to the content pane
jtf = new JTextField(15);
contentPane.add(jtf);
}
public void itemStateChanged(ItemEvent ie) {
JCheckBox cb = (JCheckBox)ie.getItem();
jtf.setText(cb.getText());
}
}
```

Radio Buttons

Radio buttons are supported by the JRadioButton class, which is a concrete
implementation of AbstractButton. Its immediate superclass is JToggleButton, which
provides support for two-state buttons. Some of its constructors are shown here:

```
JRadioButton(Icon i)
JRadioButton(Icon i, boolean state)
JRadioButton(String s)
JRadioButton(String s, boolean state)
JRadioButton(String s, Icon i)
JRadioButton(String s, Icon i, boolean state)
```

Here, i is the icon for the button. The text is specified by s. If state is true, the button is
initially selected. Otherwise, it is not.
Radio buttons must be configured into a group. Only one of the buttons in that

group can be selected at any time. For example, if a user presses a radio button that is
in a group, any previously selected button in that group is automatically deselected.
The ButtonGroup class is instantiated to create a button group. Its default constructor
is invoked for this purpose. Elements are then added to the button group via the
following method:
void add(AbstractButton ab)
Here, ab is a reference to the button to be added to the group.
The following example illustrates how to use radio buttons. Three radio buttons
and one text field are created. When a radio button is pressed, its text is displayed in
the text field. First, the content pane for the JApplet object is obtained and a flow
layout is assigned as its layout manager. Next, three radio buttons are added to the
content pane. Then, a button group is defined and the buttons are added to it. Finally,
a text field is added to the content pane.
Radio button presses generate action events that are handled by actionPerformed( ).
The getActionCommand( ) method gets the text that is associated with a radio button
and uses it to set the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet
implements ActionListener {
JTextField tf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add radio buttons to content pane
JRadioButton b1 = new JRadioButton("A");
b1.addActionListener(this);
contentPane.add(b1);
JRadioButton b2 = new JRadioButton("B");
```

```
b2.addActionListener(this);
contentPane.add(b2);
JRadioButton b3 = new JRadioButton("C");
b3.addActionListener(this);
contentPane.add(b3);
// Define a button group
ButtonGroup bg = new ButtonGroup();
bg.add(b1);
bg.add(b2);
bg.add(b3);
// Create a text field and add it
// to the content pane
tf = new JTextField(5);
contentPane.add(tf);
}
public void actionPerformed(ActionEvent ae) {
tf.setText(ae.getActionCommand());
}
}
```

Combo Boxes

Swing provides a combo box (a combination of a text field and a drop-down list)
through the JComboBox class, which extends JComponent. A combo box normally
displays one entry. However, it can also display a drop-down list that allows a user to
select a different entry. You can also type your selection into the text field. Two of
JComboBox's constructors are shown here:
JComboBox( )
JComboBox(Vector v)
Here, v is a vector that initializes the combo box.
Items are added to the list of choices via the addItem( ) method, whose signature is
shown here:
void addItem(Object obj)
Here, obj is the object to be added to the combo box.
The following example contains a combo box and a label. The label displays an
icon. The combo box contains entries for "France", "Germany", "Italy", and "Japan".
When a country is selected, the label is updated to display the flag for that country.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/
public class JComboBoxDemo extends JApplet
implements ItemListener {
JLabel jl;
ImageIcon france, germany, italy, japan;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Create a combo box and add it
// to the panel
JComboBox jc = new JComboBox();
jc.addItem("France");
jc.addItem("Germany");
jc.addItem("Italy");
jc.addItem("Japan");
jc.addItemListener(this);
contentPane.add(jc);
// Create label
jl = new JLabel(new ImageIcon("france.gif"));
contentPane.add(jl);
}
public void itemStateChanged(ItemEvent ie) {
String s = (String)ie.getItem();
jl.setIcon(new ImageIcon(s + ".gif"));
}
}
```

Tabbed Panes

A tabbed pane is a component that appears as a group of folders in a file cabinet. Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options.

Tabbed panes are encapsulated by the JTabbedPane class, which extends
JComponent. We will use its default constructor. Tabs are defined via the following
method:
void addTab(String str, Component comp)
Here, str is the title for the tab, and comp is the component that should be added to the
tab. Typically, a JPanel or a subclass of it is added.
The general procedure to use a tabbed pane in an applet is outlined here:
1. Create a JTabbedPane object.
2. Call addTab( ) to add a tab to the pane. (The arguments to this method define
the title of the tab and the component it contains.)
3. Repeat step 2 for each tab.
4. Add the tabbed pane to the content pane of the applet.
The following example illustrates how to create a tabbed pane. The first tab is titled
“Cities” and contains four buttons. Each button displays the name of a city. The second
tab is titled “Colors” and contains three check boxes. Each check box displays the name
of a color. The third tab is titled “Flavors” and contains one combo box. This enables
the user to select one of three flavors.

```
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/
public class JTabbedPaneDemo extends JApplet {
public void init() {
JTabbedPane jtp = new JTabbedPane();
jtp.addTab("Cities", new CitiesPanel());
jtp.addTab("Colors", new ColorsPanel());
jtp.addTab("Flavors", new FlavorsPanel());
getContentPane().add(jtp);
}
}
class CitiesPanel extends JPanel {
public CitiesPanel() {
JButton b1 = new JButton("New York");
add(b1);
JButton b2 = new JButton("London");
add(b2);
```

```
JButton b3 = new JButton("Hong Kong");
add(b3);
JButton b4 = new JButton("Tokyo");
add(b4);
}
}
class ColorsPanel extends JPanel {
public ColorsPanel() {
JCheckBox cb1 = new JCheckBox("Red");
add(cb1);
JCheckBox cb2 = new JCheckBox("Green");
add(cb2);
JCheckBox cb3 = new JCheckBox("Blue");
add(cb3);
}
}
class FlavorsPanel extends JPanel {
public FlavorsPanel() {
JComboBox jcb = new JComboBox();
jcb.addItem("Vanilla");
jcb.addItem("Chocolate");
jcb.addItem("Strawberry");
add(jcb);
}
}
```

Scroll Panes

A scroll pane is a component that presents a rectangular area in which a component
may be viewed. Horizontal and/or vertical scroll bars may be provided if necessary.
Scroll panes are implemented in Swing by the JScrollPane class, which extends
JComponent. Some of its constructors are shown here:
JScrollPane(Component comp)
JScrollPane(int vsb, int hsb)
JScrollPane(Component comp, int vsb, int hsb)
Here, comp is the component to be added to the scroll pane. vsb and hsb are int
constants that define when vertical and horizontal scroll bars for this scroll pane are
shown. These constants are defined by the ScrollPaneConstants interface. Some
examples of these constants are described as follows:
Constant Description

HORIZONTAL_SCROLLBAR_ALWAYS Always provide horizontal
scroll bar
HORIZONTAL_SCROLLBAR_AS_NEEDED Provide horizontal scroll bar,
if needed
VERTICAL_SCROLLBAR_ALWAYS Always provide vertical scroll bar
VERTICAL_SCROLLBAR_AS_NEEDED Provide vertical scroll bar,
if needed
Here are the steps that you should follow to use a scroll pane in an applet:
1. Create a JComponent object.
2. Create a JScrollPane object. (The arguments to the constructor specify the
component and the policies for vertical and horizontal scroll bars.)
3. Add the scroll pane to the content pane of the applet.
The following example illustrates a scroll pane. First, the content pane of the
JApplet object is obtained and a border layout is assigned as its layout manager. Next,
a JPanel object is created and four hundred buttons are added to it, arranged into
twenty columns. The panel is then added to a scroll pane, and the scroll pane is added
to the content pane. This causes vertical and horizontal scroll bars to appear. You can
use the scroll bars to scroll the buttons into view.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JScrollPaneDemo" width=300 height=250>
</applet>
*/
public class JScrollPaneDemo extends JApplet {
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new BorderLayout());
// Add 400 buttons to a panel
JPanel jp = new JPanel();
jp.setLayout(new GridLayout(20, 20));
int b = 0;
for(int i = 0; i < 20; i++) {
for(int j = 0; j < 20; j++) {
```

```
jp.add(new JButton("Button " + b));
++b;
}
}
// Add panel to a scroll pane
int v =
ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h =
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDE
D;
JScrollPane jsp = new JScrollPane(jp, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);
}
}
```

Trees

A tree is a component that presents a hierarchical view of data. A user has the ability to
expand or collapse individual subtrees in this display. Trees are implemented in Swing
by the JTree class, which extends JComponent. Some of its constructors are shown here:
JTree(Hashtable ht)
JTree(Object obj[ ])
JTree(TreeNode tn)
JTree(Vector v)
The first form creates a tree in which each element of the hash table ht is a child node.
Each element of the array obj is a child node in the second form. The tree node tn is the
root of the tree in the third form. Finally, the last form uses the elements of vector v as
child nodes.
A JTree object generates events when a node is expanded or collapsed. The
addTreeExpansionListener( ) and removeTreeExpansionListener( ) methods allow
listeners to register and unregister for these notifications. The signatures of these
methods are shown here:
void addTreeExpansionListener(TreeExpansionListener tel)
void removeTreeExpansionListener(TreeExpansionListener tel)
Here, tel is the listener object.

The getPathForLocation( ) method is used to translate a mouse click on a specific
point of the tree to a tree path. Its signature is shown here:
TreePath getPathForLocation(int x, int y)
Here, x and y are the coordinates at which the mouse is clicked. The return value is a
TreePath object that encapsulates information about the tree node that was selected by
the user.
The TreePath class encapsulates information about a path to a particular node in a
tree. It provides several constructors and methods. In this book, only the toString( )
method is used. It returns a string equivalent of the tree path.
The TreeNode interface declares methods that obtain information about a tree
node. For example, it is possible to obtain a reference to the parent node or an
enumeration of the child nodes. The MutableTreeNode interface extends TreeNode. It
declares methods that can insert and remove child nodes or change the parent node.
The DefaultMutableTreeNode class implements the MutableTreeNode interface.
It represents a node in a tree. One of its constructors is shown here:
DefaultMutableTreeNode(Object obj)
Here, obj is the object to be enclosed in this tree node. The new tree node doesn’t have a
parent or children.
To create a hierarchy of tree nodes, the add( ) method of DefaultMutableTreeNode
can be used. Its signature is shown here:
void add(MutableTreeNode child)
Here, child is a mutable tree node that is to be added as a child to the current node.
Tree expansion events are described by the class TreeExpansionEvent in the
javax.swing.event package. The getPath( ) method of this class returns a TreePath
object that describes the path to the changed node. Its signature is shown here:
TreePath getPath( )
The TreeExpansionListener interface provides the following two methods:
void treeCollapsed(TreeExpansionEvent tee)
void treeExpanded(TreeExpansionEvent tee)

Here, tee is the tree expansion event. The first method is called when a subtree is
hidden, and the second method is called when a subtree becomes visible.
Here are the steps that you should follow to use a tree in an applet:
1. Create a JTree object.
2. Create a JScrollPane object. (The arguments to the constructor specify the tree
and the policies for vertical and horizontal scroll bars.)
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane of the applet.
The following example illustrates how to create a tree and recognize mouse clicks on
it. The init( ) method gets the content pane for the applet. A DefaultMutableTreeNode
object labeled “Options” is created. This is the top node of the tree hierarchy. Additional
tree nodes are then created, and the add( ) method is called to connect these nodes to
the tree. A reference to the top node in the tree is provided as the argument to the
JTree constructor. The tree is then provided as the argument to the JScrollPane
constructor. This scroll pane is then added to the applet. Next, a text field is created
and added to the applet. Information about mouse click events is presented in this text
field. To receive mouse events from the tree, the addMouseListener( ) method of the
JTree object is called. The argument to this method is an anonymous inner class that
extends MouseAdapter and overrides the mouseClicked( ) method.
The doMouseClicked( ) method processes mouse clicks. It calls getPathForLocation( ) to translate the coordinates of the mouse click into a TreePath
object. If the mouse is clicked at a point that does not cause a node selection, the return
value from this method is null. Otherwise, the tree path can be converted to a string
and presented in the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
<applet code="JTreeEvents" width=400 height=200>
```

```
</applet>
*/
public class JTreeEvents extends JApplet {
JTree tree;
JTextField jtf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
// Set layout manager
contentPane.setLayout(new BorderLayout());
// Create top node of tree
DefaultMutableTreeNode top = new
DefaultMutableTreeNode("Options");
// Create subtree of "A"
DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
top.add(a);
DefaultMutableTreeNode a1 = new
DefaultMutableTreeNode("A1");
a.add(a1);
DefaultMutableTreeNode a2 = new
DefaultMutableTreeNode("A2");
a.add(a2);
// Create subtree of "B"
DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
top.add(b);
DefaultMutableTreeNode b1 = new
DefaultMutableTreeNode("B1");
b.add(b1);
DefaultMutableTreeNode b2 = new
DefaultMutableTreeNode("B2");
b.add(b2);
DefaultMutableTreeNode b3 = new
DefaultMutableTreeNode("B3");
b.add(b3);
// Create tree
tree = new JTree(top);
// Add tree to a scroll pane
int v =
ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h =
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDE
D;
JScrollPane jsp = new JScrollPane(tree, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);
```

```
// Add text field to applet
jtf = new JTextField("", 20);
contentPane.add(jtf, BorderLayout.SOUTH);
// Anonymous inner class to handle mouse clicks
tree.addMouseListener(new MouseAdapter() {
public void mouseClicked(MouseEvent me) {
doMouseClicked(me);
}
});
}
void doMouseClicked(MouseEvent me) {
TreePath tp = tree.getPathForLocation(me.getX(), me.getY());
if(tp != null)
jtf.setText(tp.toString());
else
jtf.setText("");
}
}
```

The string presented in the text field describes the path from the top tree node to the
selected node.

Tables

A table is a component that displays rows and columns of data. You can drag the
cursor on column boundaries to resize columns. You can also drag a column to a new
position. Tables are implemented by the JTable class, which extends JComponent.
One of its constructors is shown here:
JTable(Object data[ ][ ], Object colHeads[ ])
Here, data is a two-dimensional array of the information to be presented, and colHeads
is a one-dimensional array with the column headings.
Here are the steps for using a table in an applet:
1. Create a JTable object.
2. Create a JScrollPane object. (The arguments to the constructor specify the table
and the policies for vertical and horizontal scroll bars.)
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane of the applet.
The following example illustrates how to create and use a table. The content pane

of the JApplet object is obtained and a border layout is assigned as its layout manager.
A one-dimensional array of strings is created for the column headings. This table has
three columns. A two-dimensional array of strings is created for the table cells. You can
see that each element in the array is an array of three strings. These arrays are passed to
the JTable constructor. The table is added to a scroll pane and then the scroll pane is
added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTableDemo" width=400 height=200>
</applet>
*/
public class JTableDemo extends JApplet {

public void init() {
// Get content pane
Container contentPane = getContentPane();
// Set layout manager
contentPane.setLayout(new BorderLayout());
// Initialize column headings
final String[] colHeads = { "Name", "Phone", "Fax" };
// Initialize data
final Object[][] data = {
{ "Gail", "4567", "8675" },
{ "Ken", "7566", "5555" },
{ "Viviane", "5634", "5887" },
{ "Melanie", "7345", "9222" },
{ "Anne", "1237", "3333" },
{ "John", "5656", "3144" },
{ "Matt", "5672", "2176" },
{ "Claire", "6741", "4244" },
{ "Erwin", "9023", "5159" },
{ "Ellen", "1134", "5332" },
{ "Jennifer", "5689", "1212" },
{ "Ed", "9030", "1313" },
{ "Helen", "6751", "1415" }
};
// Create the table
JTable table = new JTable(data, colHeads);
// Add table to a scroll pane
```

```
int v =
ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h =
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDE
D;
JScrollPane jsp = new JScrollPane(table, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);
}
}
```