1 **History of Java**
Features of Java
The Java Environment
The Java Virtual Machine (JVM)

1.1 History of Java

- Java is a general-purpose, object-oriented programming language developed by Sun Microsystems of USA in 1991.
- Originally called Oak by James Gosling, one of the inventors of the language, Java was designed for the development of software for consumer electronic devices like TVs, VCRs, toasters and such other electronic machines.
- The goal had a strong impact on the development team to make the language simple, portable and highly reliable.
- The Java team which included Patrick Naughton discovered that the existing languages like C and C++ had limitations in terms of both reliability and portability. However, they modeled their new language Java on C and C++ but removed a number of features of C and C++ that were considered as sources of problems and thus made Java a really simple, reliable, portable, and powerful language.
  - **1990** Sun Microsystems- decided to develop special software that could be used to manipulate consumer electronic devices. A team of Sun Microsystems programmers beaded by James Gosling was formed to undertake this task.
  - **1991** After exploring the possibility of using the most popular object-oriented language C++. The team announced a new language named “Oak".
  - **1992** The team, known as Green Project team by Sun. demonstrated the application of their new language to control a list of home appliances using a hand-held device with a tiny touch-sensitive screen.
  - **1993** The World Wide Web (WWW) appeared on the Internet and transformed the text-based Internet into a graphical-rich environment. The Green Project team carne up with the idea of developing Web applets (tiny programs) using the new language that could run on all types of computers connected to Internet.
  - **1994** The team developed a Web browser called "HotJava" to locate and run applet programs on Internet. HotJava demonstrated the power of the new language, thus making it instantly popular among the Internet users.
  - **1995** Oak was renamed "Java", due to some legal snags. Java is just a name and is not an acronym. Many popular companies including Netscape and Microsoft announced their support to Java.
  - **1996** Java established itself not only as a leader for Internet programming but also as a general-purpose, object-oriented programming language. Sun releases Java Development Kit 1.0.
  - **1997** Sun releases Java Development Kit 1.1 (JDK 1.1).
  - **1998** Sun releases the Java 2 with version 1.2 of the Software Development Kit (SDK 1.2).
  - **1999** Sun releases Java 2 Platform. Standard Edition (J2SE) and Enterprise Edition (J2EE).
  - **2000** J2SE with SDK 1.3 was released.
  - **2002** J2SE with SDK 1.4 was released.
  - **2004** J2SE with JDK 5.0 (instead of JDK 1.5) was released. This is known as J2SE 5.0.

- The most striking feature of the language is that it is a *platform-neutral language.* Java is the first programming language that is not tied to any particular hardware or operating system. Programs developed In Java Can be executed anywhere on any system. We can call Java as a revolutionary technology because it has brought in a fundamental shift in how we develop and use programs. Nothing like this has happened to the software industry before.

1.2 Features of Java

- The Inventors of Java wanted to design a language which could offer solutions to some of the problems to encounter in modern programming. They wanted the language to be not only reliable, portable and distributed but also simple, compact and interactive. Sun Microsystems officially describes Java with the following attributes:

- Java 2 Features
  - Compiled and Interpreted
  - Platform-Independent and Portable
  - Object-Oriented
  - Robust and Secure
  - Distributed
  - Familiar. Simple and Small
  - Multithreaded and interactive
  - High Performance
  - Dynamic and Extensible
- Additional Features of J2SE 5.0
  - Ease of Development
  - Scalability and Performance
  - Monitoring and Manageability
  - Desktop Client
  - Core XML Support
  - Supplementary character support
  - JDBC RowSet

- Although the above appears to be a list of buzzwords, they aptly describe the full potential of the language. These features have made Java the first application language of the World Wide Web. Java will also become the premier language for general purpose stand-alone applications.

1.2.1 Compiled and Interpreted

Usually a computer language is either compiled or interpreted. Java combines both these approaches thus making Java a two-stage system. First lava compiler translates Source code into what is known as *bytecode* instructions. Bytecodes are not machine instructions and therefore, in the second stage, Java interpreter generates machine code that can be directly executed by the machine that is running the Java program. We can thus say that Java is both a compiled and an interpreted language.

1.2.2 Platform-Independent and Portable

The most significant contribution of Java over other languages is its portability. Java programs can be easily moved from one computer system to another, *anywhere* and *anytime.* Changes and upgrades in operating systems, processors and system resources will not force any changes in Java programs. This is the reason why Java has become a popular language for programming on Internet which interconnects different kinds of systems worldwide.

We can download a Java applet from a remote computer onto our local system via Internet and execute it locally. This makes the Internet an extension of the user's basic system providing practically unlimited number of accessible applets and applications.

Java ensures portability in two ways. First, Java compiler generates bytecode instructions that can *be* implemented on any machine. Secondly, the size of the primitive data types is machine independent.

1.2.3 Object-Oriented

Java is a true object-oriented language. Almost everything in Java is an *object.* All program code and data reside within objects and classes. Java comes with an extensive set of *classes,* arranged in *packages,* which we can use in our programs by inheritance. The object model in Java is simple and easy to extend.

1.2.4 Robust and Secure

Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run-time checking for data types. It is designed as a garbage-collected language relieving the programmers virtually all memory management problems. Java also incorporates the concept of exception handling which captures series errors and eliminates any risk of crashing the system.

Security becomes an important issue for a language that is used for programming on Internet. Threat of viruses and abuse of resources are everywhere. Java systems not only verify all memory access but also ensure that no viruses are communicated with an applet. The absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization.

1.2.5 Distributed

Java is designed as a distributed language for creating applications on networks. It has the ability to share both data and programs. Java applications can open and access remote objects on Internet as easily as they can do in a local system. This enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

1.2.6 Simple, Small and Familiar

Java is a small and simple language. Many features of C and C++ are either redundant or secures of unreliable code are not part of Java. For example, Java does not use pointers, preprocessor header files, goto statement and many others. It also eliminates operator overloading and multiple inheritance. To make the language look familiar to the existing programmers, it was modeled on C and C++ languages. Java uses many constructs of C and C++ and therefore, Java code “looks like a C++" code. In fact Java is simplified version of C++.

1.2.7 Multithreaded and Interactive

Multithreaded means handling multiple tasks simultaneously. Java supports multithreaded programs. This means we need not wait for the application to finish one task before beginning other. For example, we can listen to an audio clip while typing document and at the same time download an applet from a distant computer.

This feature greatly improves the interactive performance of Graphical applications. The Java runtime comes with tools that support multiprocess synchronizations and construct smoothly running interactive systems.

1.2.8 High Performance

Java performance is impressive for an interpreted language, mainly due to the use of intermediate bytecode. According to sun, java speed is comparable to the native C/C++, Java architecture is also designed to reduce overheads during runtime. Further, the incorporation of multithreading enhances the overall execution speed of Java program.

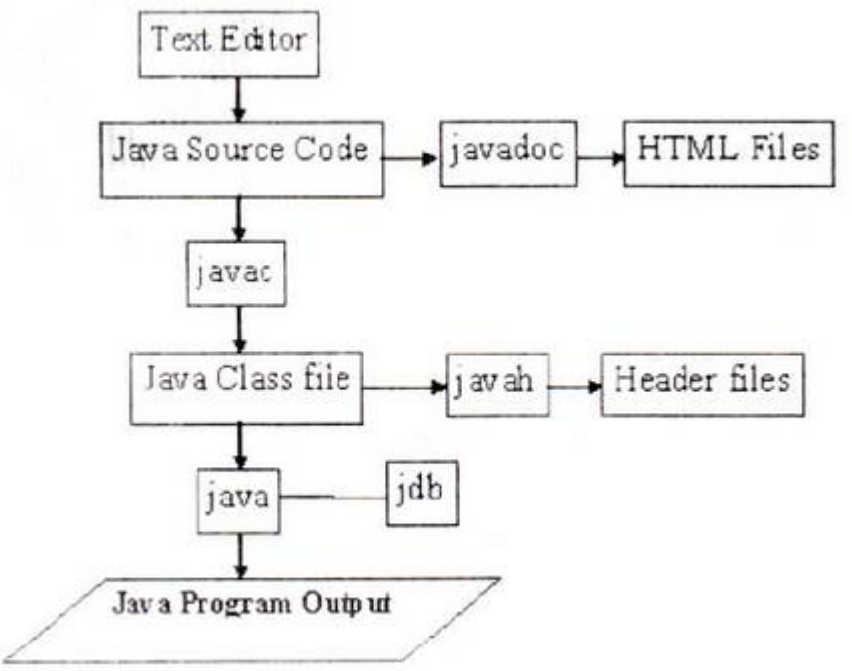
1.2.9 Dynamic and Extensible

Java is dynamic language. Java is capable of dynamically linking in new class libraries, methods, and objects. Java can also determine the type of class through a query, making it possible to either dynamically link or abort the program, depending on the response.

Java programs support functions written in other language such as C and C++. These functions are known as native methods. This facility enables the programmers to use the efficient functions available in these languages. Native methods are linked dynamically at runtime.

1.3 The Java Environment

- The java environment includes a large number of development tools and hundreds of classes and methods.
- The development tools are part of the system known as *Java Development Kit (JDK)* and the classes and methods are part of the *Java Standard Library (JSL),* also known as the *Application Programming Interface(API).*



- The Java Development Kit comes with a collection of tools that are used for developing and running Java program.

Java Development Tools	
Tool	***Description***
Appletviewer	Enables us to run java applet (without actually using a java-compatible browser).
Java	Java interpreter, which runs applets and applications by reading and interpreting bytecode files.
Javac	The java compiler, which translates java source code to bytecode files that the interpreter can understand.
javadoc	Creates HTML format documentation from java source code files.
javah	Produces header files for use with native methods.
javap	Java disassemble, which enables us to convert bytecode files into a program description.
Jdb	Java debugger, which helps us to find errors in our programs.

- **Application Programming Interface(API)**
  The java standard library (or API) includes hundreds of classes and methods grouped into several functional packages.

Most Common Packages	
Package	***Description***
Language Support Package	A collection of classes and methods required for implementing basic features of java.
Utility Package	A collection of classes to provide utility functions such as date and time functions.
Input/Ouput Package	A collection of classes required for input/output manipulation.
AWT Package	The Abstract Window Toolkit package contains classes that implements platform-independent graphical user interface.
Applet Package	This includes a set of classes that allows us to create java applets.
Networking Package	A collection of classes for communicating with other computers via internet.

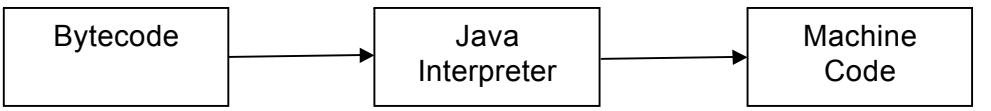
1.4 The Java Virtual Machine (JVM)

- All language compilers translate source code into machine code for a specific computer. Java compiler also does the same thing. Then, how does java achieve architecture neutrality?
- The answer is that the java compiler produces an intermediate code known as *bytecode* for a machine that does not exist.
- This machine is called the *Java Virtual* Machine (JVM) and it exists only inside the computer memory. It is a simulated computer within the computer and does all major functions of a real computer.

- The process of compiling a java program into bytecode which is also referred to as *virtual machine code.*

Process of compilation (Converting source code into bytecode)

- The virtual machine code is not machine specific. The machine specific code (known as machine code) is generated by java interpreter by acting as an intermediary between the virtual machine and the real machine.
- Remember that the interpreter is different from different machines..



Process of interpretation (Converting bytecode into machine code)

- Above figure illustrates how java works on a typical computer. The java object framework (Java API) acts as intermediary between the user programs and the virtual machine which in turn acts as the intermediary between the operating system and the java object framework.

2 Structure of a Java program
A simple Java program
Implementing a Java program

2.1 Structure of a Java program

- A java program may contain many classes of which only one class defines a main method. Classes contain data members and methods that operate on the data members of the class.
- Method may contain data type declarations and executable statements. To write a java program, we first define classes and then put them together.

Documentation Section
Package Statement
Import Statement
Interface Statements
Class Definitions

Main Method Class
{
Main Method Definition
}

2.1.1 Documentation Section

- The documentation section comprises a set of comment lines giving the name of the program, the author and other details, which the programmer would like to refer to at a later stage.
- Comments must explain why and what id classes and how of algorithms. This would greatly help in maintaining the program.
- In addition to the two styles of comments, java also uses a third style of comment /**….*/ known as documentation comment. This form of comments is used for generating documentation automatically.

2.1.2 Package Statement

- The first statement allowed in a java file is a package statement. This statement declares a package name and informs the compiler that the classes defined here belong o this package.
- For example: package student;
- The package statement is optional. That is, our classes do not have to be part of package

2.1.3 Import Statement

- The next thing after a package statement (but before any class definitions) may be a number of import statements. This is similar to the #include statement in C.
- For example: import student.test;
- This statement instructs the interpreter to load the test class contained in the package student. Using import statements, we can have access to classes that are part of other named package.

2.1.4 Interface Statements

- An interface is like a class but included a group of method declarations. This is also an optional section and is used only when we wish to implement the multiple inheritance features in the program.

2.1.5 Class Definitions

- A java program may contain multiple class definition. Classes are the primary and essential elements of a java program. These classes are used to map the objects of real-world problems. The number of classes used depends on the complexity of the problem.

2.1.6 Main Method Class

- Since every Java stand-alone program requires a main method as its starting point, this class is the essential part of a java program.
- A simple java program may contain only this part. The main method creates objects of various classes and establishes communications between them.
- On reaching the end of main, the program terminates and the control passes back to the operating system.

2.2 A simple Java program

- The best way to learn a new language is to write a few simple example programs and execute them. We begin with a very simple program that prints a line of text as output.

```
class SampleOne
{
     public static void main()
     {
          System.out.println("java is better than c++");
     }
}
```

- Above is simplest of all java programs. It brings out some salient features of the language. Let us therefore discuss the program line by line and understand the unique feature that constitutes a java program.

2.2.1 Class Declaration

- The first line

  Class SampleOne

  Declares a class, which is an object- oriented construct.
- Java is true object- oriented language and therefore, everything must be placed inside a class.
- Class is keyword and declares that a new class definition follows. SampleOne is a java identifier that specifies the name of the class to be defined.

2.2.2 Opening Brace

- Every class definition in java begins with an opening brace "{" and ends with a matching Closing brace "}", appearing in the last line in the example. This is similar to C++ class constructs.
- (Note that a class definition in C++ ends with a semicolon)

2.2.3 The Main Line

- The third line

  Public static void mian(String args[])

  Defines a method named main.
- Conceptually, this is similar to the main() function in c/c++. Every Java application program must include main() method. This is the starting point for the interpreter to begin the execution of the program.
- A java application can have any number of classes but only one of them must include a main method to initiate the execution.
- (Note that java applets will not use the main method at all)

- **Public (Access Specifier) :** the keyword public is an access specifier that declares the main method as unprotected and therefore making it accessible to all other classes. This is similar to the c++ public modifier.
- **Static:** next appears the keyword static, which declares this method as one that belongs to the entire class and not a part of any objects of the class. The main must always be declared as static since the interpreter uses this method before any objects are created. (allows to main method call without create object)
- **Void:** the type modifier void state that the main method does not return any value(but simply prints some text to the screen)
- All parameters to a method are declared inside a pair of parentheses. Here, String args[] declares parameter named args, which contains an array of objects of the class type String.

2.2.4 The Output Line

- The only executable statement in the program is

  System.out.println(“”);

  This is similar to the printf() statement of c or cout<< construct of C++.
- Since java is true object oriented language, every method must be part of an object. The println method is a member of the out inner class, which is a static data member of System class. This line prints specified string to the screen.
- Note the semicolon at the end of the statement. Every java statement must end a semicolon.

2.3 Implementing a Java Program

- Implementation of a java application program involves a series of steps. They include :
  - Creating the program
  - Compiling the program
  - Running (executing, interpreting) the program
- Remember that, before we begin creating the program, the Java Development Kit (JDK) must be properly installed on your system.

2.3.1 Creating The Program

- We should follow some rules before creating the java program.
  1. *File name* and *class name* which contains main() method must be *identical* (same and case sensitive).
  2. It suggested to give *first character of each word* of file name and class name should *uppercase*(i.e. Test, StudentDetail, SampleOne)
  3. Method name’s *first word should be lowercase then every word’s first character should be uppercase*(i.e. getData(), getStudentDetail())
- We can create a program using any text editor. Assume that we have entered the following program:

```
Class SampleOne
{
      Public static void main()
      {
            System.out.println("Hello");
      }
}
```

- We must save this in a file called SampleOne.java ensuring that the filename contains the class name properly.
- This file is called the *source file.* Note that all java source files will have the extension **java.**
- Note also that if a program contains multiple classes, the file name must be the class name of the class containing the **main** method.

2.3.2 Compiling The Program

- To compile the program, we must run the java compiler **javac,** with the name of the source file on the command line as shown below:

```
Javac SampleOne.java
```

- If everything is OK, the **javac** compiler creates a file called **SampleOne.class** containing the *bytecode* of the program. Note that the compiler automatically names the *bytecode* file as <classname>.class.

2.3.3 Running The Program

- We need to use the java interpreter to run a stand-alone program. At the prompt, type :

```
Java SampleOne
```

- Note when we compile, we should give *.java* extension. When we run it, we must not give the extension because it will use *.class* file.
- Now, the interpreter looks for the main method in the program (SampleOne.class) and begins execution from there. When executed, our program displays the following :

  Hello

2.3.4 Machine Netural

- The compiler converts the source code files into bytecode files. These codes are machine-independent and therefore can be run on any machine.
- That is, a program compiled on an IBM machine will run on a Macintosh Machine.
- *Java interpreter* reads the bytecode files and translates them into machine code for the specific machine on which the java program is running.
- The interpreter is therefore specially written for each type of machine.

3 Tokens
Comments
Constants
Variables
Data Types

3.1 Tokens

- A Java program is basically a collection of classes. A class is defined by a set of declaration statements and methods containing executable statements. Most statements contain expressions, which describe the action carried out on data. Smallest individual units in program are known as *tokens*. The compiler recognizes them for building up expressions and statements.
- Java language includes 5 types of tokens:
  (1) Reserved Keywords
  (2) Identifiers
  (3) Literals
  (4) Operators
  (5) Separators

3.1.1 Reserved Keywords

- Keywords are essential part of a language definition. They implement specific features of the language. Java language has.reserved.60 keywords.
- Since keywords have specific meaning in Java, we cannot use them as names for variables, classes, methods and so on.
- All the keywords are to be written in lower case letters. Since java is case-sensitive, one can use these words as identifiers by changing one or more letters to upper-case. Java Keywords are as follow:

abstract	boolean	break	byte	byvalue	case
cast	catch	char	class	const	continue
default	do	double	else	extends	false
final	finally	float	for	future	generic
goto	if	implements	import	inner	instanceof
int	interface	long	native	new	null
operator	outer	package	private	protected	public
rest	return	short	static	super	switch
synchronized	this	threadsafe	throw	throws	transient
true	try	var	void	volatile	while

3.1.2 Identifiers

- Identifiers are programmer-designed tokens. They are used for naming classes, methods, variables, Objects, labels, packages and interfaces in a program. Java identifiers should obey the following rules: -
  (1) They can have alphabets, digits and the underscore and dollar sign character
  (2) They must not begin with a digit
  (3) Uppercase and lowercase letters are distinct
  (4) They can be any length

- Identifier must be meaningful, short enough to be quickly and easily typed and long enough to be descriptive and easily read.

3.1.3 Literals

- Literals in Java are a sequence of characters (digits, letters and other characters) that represent constant value to be stored in variables. Java language specifies five major types of literals.
  (1) Integer literal
  (2) Floating point literal
  (3) Character literal
  (4) String literal
  (5) Boolean literal
- Each of them has a type associated with it. The type describes how the values behave and how they are stored.

3.1.4 Operators

- An operator is a symbol that takes one or more arguments and operates on them to produce a result. Operators are of many types and are considered in detail in later.

3.1.5 Separators

- Separators are symbols used to indicate where groups of code are divided and arranged. They basically defined the shape and function of our code.
- Parentheses ()
- Braces {}
- Brackets []
- Semicolon ;
- Comma ,
- Period.

3.2 Comments

- Java permits both single-line comments and multi-line comments available in C++. The single line comments begins with //. For longer comments, we can create long multi-line comments by starting with /* and ends with */.

3.3 Constants

- Constants in Java refer to fix values that do not change during the execution of a program.
- Integer constant refers to a series of digits. There are 3 types of integers, namely, decimal integer (set of 0 to 9), octal integer (set of 0 to 7 and leading 0), hexadecimal integer (set of digits preceded by OX and 0 to 9 and A to F)
- Real constants represented by numbers containing fractional parts like 17.235. Such numbers are called real (floating point) constants.
- Single character constants contain a single character enclosed within a pair of single quote mark. As for example

  'A' '5' ';' ' '

- A String constant is a sequence of characters enclosed between double quotes. The character may be alphabets, digits, special characters and blank spaces. As for example

  "Hello Javal, "1997" "5+7" "S"
- Backslash character constants are supported by Java used in output methods. As for example
  - '\b' back space
  - '\f' form feed
  - '\n' new line
  - '\r' character return
  - '\t' horizontal space
  - '\" single quote
  - '\"' double quote ;.
  - '\\' backslash

3.4 Variables

- A variable is an identifier that denotes storage location to store data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during the execution of the program.
- A variable name can be chosen by the programmer in a meaningful way so as to reflect what it represents in the program.
- Variable may consist of alphabets, digits, and underscore and dollar character, subject to the following conditions.

  (a) They must not begin with digit.

  (b) Uppercase and lowercase are distinct

  (c) It should not be keyword.

  (d) White space is not allowed.

  (e) Variable names can be of any length.

3.5 Data Types

- Every variable in java has a data type. Data types specify the size and type of values that can be stored.

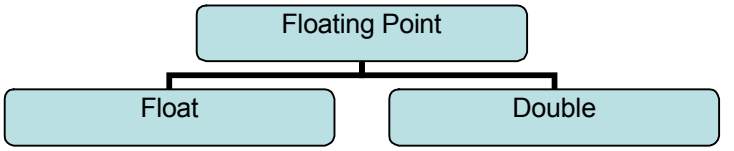
3.5.1 Integer Types

- Integer types can hold whole numbers such as 123,-95 and 5698. The size of the value that can be stored depends on the integer data types we choose. Java supports four types c integer byte, short, int, long. Java does not support the concepts of unsigned types and therefore all Java values are signed meaning they can be positive or negative.

Type	***Size***	***Min. value***	***Max. value***
Byte	One byte	-128	127
Short	Two byte	-32,768	32,767
Int	Four byte	-2,147,483,648	2,147,483,647
Long	Eight byte	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

3.5.2 Floating Point Types

- Floating point numbers are treated as double-precession quantities. To force then to be in single-precession mode, we must append f or F to the number.
- e.g) 132.45f or 125.36F



Type	*Size*	*Min. value*	*Max. value*
Float	Four byte	3.4e-038	3.4e+038
Double	Eight byte	1.7e-308	1.7e+308

- Double precession types are used when We need greater precession in storage of floating point numbers. All the mathematical function such as sin, cos and sqrt return double type values.
- Floating point data types supports a special value known as Not-a-Number (NaN). NaN used to represent the result of operations such as dividing zero by zero, where an actual number is not produced.

3.5.3 Character Type

- In order to store character constants in memory, java provides a character data type called char. The char type assumes a size of 2 bytes but, basically, it can hold only a single character.

3.5.4 Boolean Types

- Boolean type is used when we want to test a particular condition during the eexecution of the program. There are only two values that a Boolean type can take: true or false.
- Boolean type is denoted by the keyword Boolean and uses only one bit storage.

4 Scope of Variables Type Casting

4.1 Scope of Variables

- Java variables are actually classified into three kinds:
  - Instance variables
  - Class variables
  - Local variables
- Instance variables are declared inside a class. Instance variables are created when the objects are instantiated and therefore they are associated with the objects. They take different values for each object. On other hand class variables are global to a class and belong to the earlier set of objects that class creates. Only one memory location is created for each class variable.

- Variables declared and used inside methods are called local variables. Local variables can also be declared inside program blocks that are defined between an opening brace {and a closing brace}. These variables are visible to the program only from the beginning of its program block to the end of the program block. the area of the program where the variable is accessible is called its scope.

4.2 Type Casting

- We often encounter situations where there is a need to store a value of one type into variable of another type. In such situations, we must cast the value to be stored by proceeding with the name in parentheses.

  type variable1 = (type) variable2;
- The process of converting one data type to another is called casting.
- e.g

```
int m=50;
byte n=(byte)m;
long count=(long)m;
```

- Cast that result in no loss of information

From	*To*
byte	short, char, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

- Four integer types can be cast to and other type except Boolean. Casting into smaller type may result in a loss. of data. Similarly, the float and double can be cast to any other type except Boolean. Again casting Into smaller type may result in a loss of data. Casting floating point value to integer will result in a loss of the fractional part.

- **Automatic Conversation**
- For some types, it is possible to assign a value of one type to a variable of a different type without a cast.
- Java does the conversion of the assigned value automatically. This is known as automatic type conversion. Automatic type conversion is possible only if the destination type has enough precession to store the source value.
- For example, int is large enough to hold a byte value. therefore,

```
Byte b = 50;
Int a = b;
```

  are valid statement.

5

Arithmetic Operators
Relational Operators
Logical Operators
Assignment Operators
Increment/decrement Operators
Conditional Operators
Ternary Operator & Special Operators

- Java support a rich set of operators. We have used several of them. Such as +,-,*,/. 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. They usually form a part of mathematical or logical expressions.
- Java operators can be classified into a number of related categories as below:

5.1 Arithmetic Operators

- Arithmetic operators are used to construct mathematical expressions as in algebra. Java provides all the basic arithmetic operators.
- They are listed in below table. The operators +,-,*, and / all work the same way as they do in other languages. These can operate on any built-in numeric data type of java. We cannot use these operators on boolean type.
- The unary minus operator, in effect, multiplies its single operand -1. Therefore, a number preceded by a minus sign changes its sign.

Arithmetic operator	
Operator	*Meaning*
+	Addition or unary plus
-	Substraction or unary minus
*	Multiplication
/	Division
%	Modulo division (reminder)

- Arithmetic operators are used as shown below:

  A+b a-b
  A*b a/b
  A%b -a*b

- Here a and b may be variable or constants and are known as operands.

5.2 Relational Operators

- We often two quantities, and depending on their relation, take certain decisions. For exa, we may compare the age of two persons, or the price of two items, and so on.

- These comparisons can be done with the help of relational operators. We have already used the symbol '<' meaning 'less than' an expression such as

  A<b or x<20

  Containing a relational operator is termed as a relational expression. The value of relational expression is either true or false. For example, if x=10 then

  X<20 is true

  While

  20<x is false
- Java supports six relational operators in all. These operators and their meanings are shown as below.

Relational Operator	
Operator	*Meaning*
<	Is Less than
<=	Is less than equal to
>	Is greater than
>=	Is greater than equal to
==	Is equal to
!=	Is not equal to

- When arithmetic expressions are used on their side of a relational operator, the arithmetic expressions will be evaluated first and then the result compared. That is arithmetic operator have a higher priority over relational operator.

```
class RelationalOperator
{
        Public static void main(String args[])
        {
                Float a=15 of, 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+b is " + (b=a+c));
        }
}
```

- The output of program would be:
  A=15
  B=20.75
  C=15
  A< b is true
  a>b is false
  a==c is true
  a<=c is false
  a>=b is false
  a != c is true
  b == a+c is false
- Relational expressions are used in decision statement such as, if and while to decide the course of action of a running program .

5.3 Logical Operators

- In addition to the relational operators, java has three logical operator, which are given in table.

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

- The logical operators && and || are used when we want to form compound by combining two relations. an example is:
  a>b && x==10
- An expression of this kind which combines two or more relational expressions is termed as a logical expressions or a compound relational expression. Like the simple relational expressions, a logical expression also yields a value of true or false, according the follow truth table.

Truth Table			
Op -1	Op – 2	Value of the expression	
		Op-1 && op-2	Op -1 || op-2
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

- The logical expression given above is true only if a>b and x=10 are true. If either (or both) of them are false the expression is false.

5.4 Assignment Operators

- Assignment operators are used to assign the value of an expression to a variable . We have seen the usual assignment operator, '='. In addition, java has a set of 'shorthand' assignment operator which are used in the form.

  v op= exp;

- Where v is a variable, exp is an expression and op is a java binary operator. The operator op= is known as the shorthand assignment operator .
- The assignment statement

  V op = exp;

  Is equivalent to

  V=v op(exp);

  With v accessed only once. Consider example

  X+= y+1;
- This is same as the statement

  X= x+(y+1);
- The shorthand operator += means 'add +1 to x' increment x by y+1 for y=2 , the above statement becomes

  X+=3;
- And when this statement is executed, 3 is added to x. if the old value of x is, say 5, and then the new value of x is 8. Some of the commonly used shorthand assignment operators are illustrated in below table.

Assignment Operator	
Statement with simple Assignment operator	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

- The use of shorthand assignment operator has three advantages :
  1. What appears on the left hand side need not be repeated and therefore it become easier to write .
  2. The statement is more concise and easier to read.
  3. Use of shorthand operator results in a more efficient code.

5.5 Increment/decrement Operators

- Java has two very useful operator not generally found in many other language. These are the increment and decrement operators:

  ++ and --
- The operator ++ adds 1 to the operand while -- subtracts 1. Both are unary operator and are used in the following form :

  ++m or m++;

  --m or m--;

  ++m; is equivalent to m = m+1; (or m+=1;)

  --m; is equivalent to m = m-1; (or m-=1;)

- We use the increment and decrement operators extensively in for and while loops.
- While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right – hand side of an assignment statement. Consider the following :

```
M = 5;
Y=++m;
```

- In this case, the value of y and m would be 6. Suppose, if we rewrite the above statement as

```
M = 5;
Y= m++;
```

- Then, the value of y would be 5 and m would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On other hand, postfix operators first assigns the value to the variable on left and then increment the operator.

```
class Increment Operator
{
        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);
        }
}
```

- Output of program is as follows :

```
M=10
N=20
++m=11
N++ = 20
M=11
N=21
```

- Similar is the case, when we use ++(or --) in subscripted variables. That is, the statement

```
A[i++] =10
```

- Is equivalent to

```
A[i]  = 10
i = i+1
```

5.6 Conditional Operators (Ternary 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 where exp1, exp2 and exp3 are expressions.
- The operator ?: works as follows: exp1 is evaluated first. If it is nonzero (true), then the expression Exp2 is evaluated and becomes the value of the conditional expression.
- If exp1 is false, exp3 is evaluated and its value becomes the value of the conditional expression. Note that only of the expressions (either exp2 or exp3) is evaluated . for example, consider the following statements:

```
A = 10;        B =15;
X = (a>b) ? a : b;
```

- In this example, x will be assigned the value of b. this can be achieved using the if ….else statement as follows :

```
If (a>b)
        X= a;
Else
        X= b;
```

5.7 Special Operators

- Java supports some special operator of interest such as instanceof operator and member selection operator(.) .
- **Instanceof operator**
- The instanceof operator is an object 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.
- Example:

  Person instanceof student

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

- **Dot operator**
- The operator (.) is used to access the instance variables and methods of class objects. Example:

```
Person1.age          //reference to the variable age
Person1.salary()     //reference to the method salary()
```

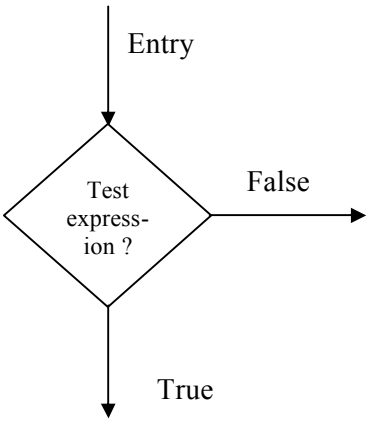
  It is also used to access classes and sub- packages from a package.

6	**Decision making: if statement** **if…else statement** **Nesting of if…else** **The else if ladder** **Switch statement**

- The if statement is a powerful decision making statement and is used to control the flow of execution of statement. It is basically a two-way decision statement and is used in conjunction with an expression. It takes the following from:

  If(test expression)

- It allows the computer to evaluate the expression first and then, depending on whether the value of the expression (relation or condition) is 'true' or 'false', it transfers the control to a particular statement, this point of program has to paths to follow, one for the true condition and the other for the false condition as shown below:
-



- Example:

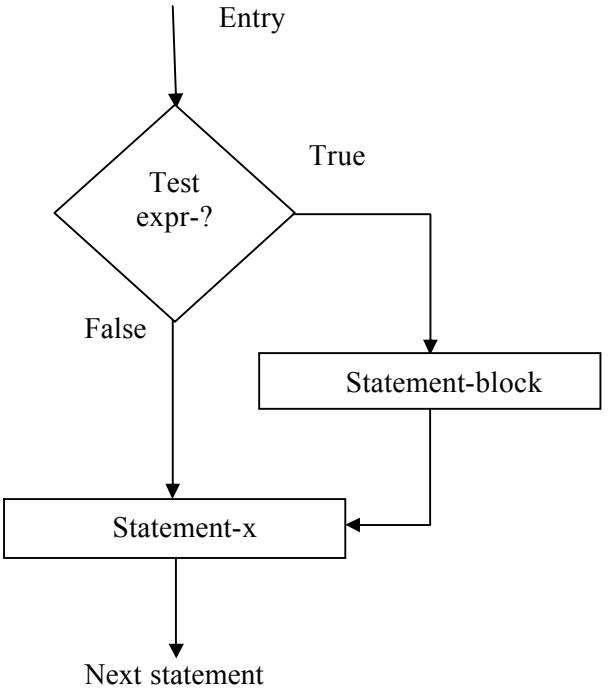
```
If(bank balance is zero)
        Borrow money
If(room is dark)
        Put on lights
```

6.1 Simple if statement

- The general form of a simple if statement is

```
If(test expression)
{
        Statement-block
}
        Statement-x;
```

- The 'statement-block' may be a single statement or a group of statements. If the test expression is true, the statement-block will be executed; otherwise the statement-block will be skipped and the execution will jump to the statement-x.
- It should be remembered that when the condition is true both the statement-bock and the statement-x are executed in sequence. This is illustrated below:



- Example:

```
Int a=10;
int b=10;
If(a==b)
{       System.out.println("both are same");      }
String str1='a';
String str2='a';
If(str1 equals(str2))
{       System.out.println("both are same:");      }
```

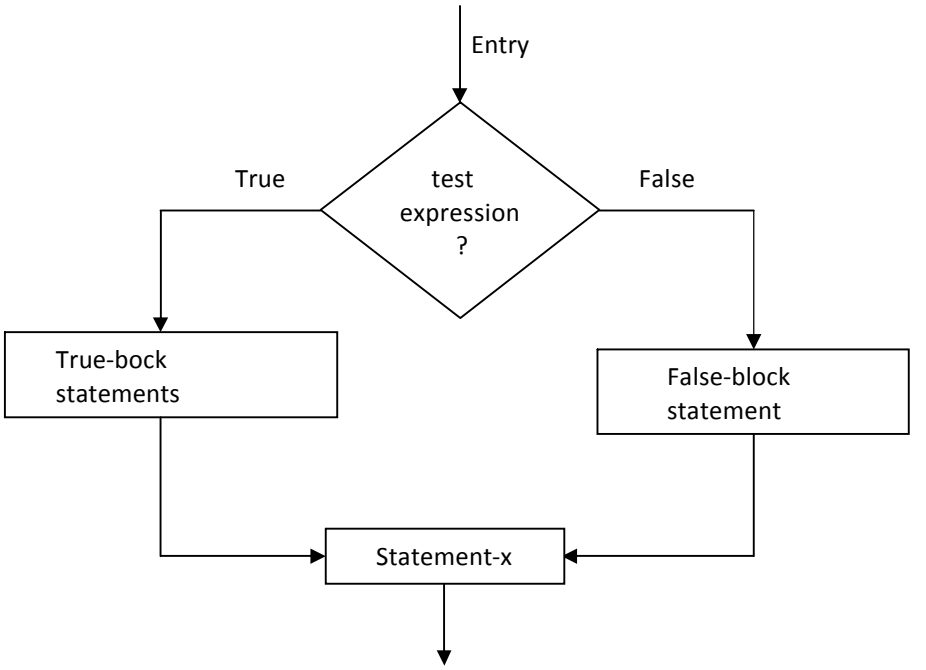
6.2 if….else statement

- The if……else statement is an extension of the simple if statement. The general form is:
- Syntax:

```
If(test expression)
{
        True-block statement(s)
}
Else
{
        False-block statement(s)
}
Statement-x
```

- If the test expression is true, then the true-block statement(s) immediately following the if statement, are executed; otherwise, the false-block statement(s) are executed. In either case, either true-block or false-block will be executed, not both.

This is illustrated in bellow. In both cases, the control is transferred subsequently to the statement-x.

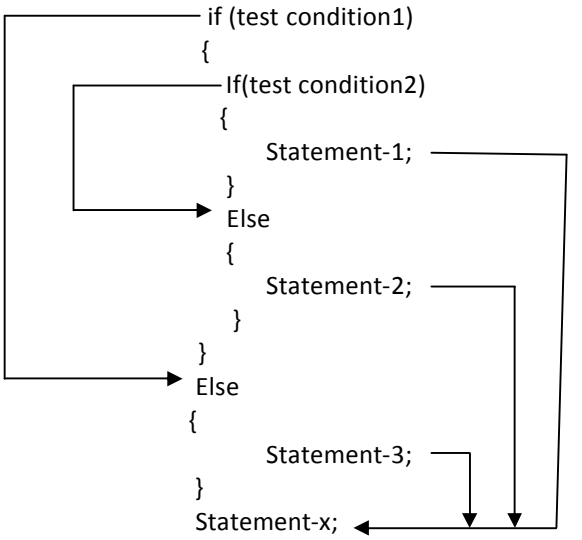


- Example:

```
String str1='a';
String str2='a';
If(str1 equals(str2))
{       System.out.println("both are same");      }
Else
{       System.out.println("both are not same");          }
```

6.3 Nesting of if….else statement

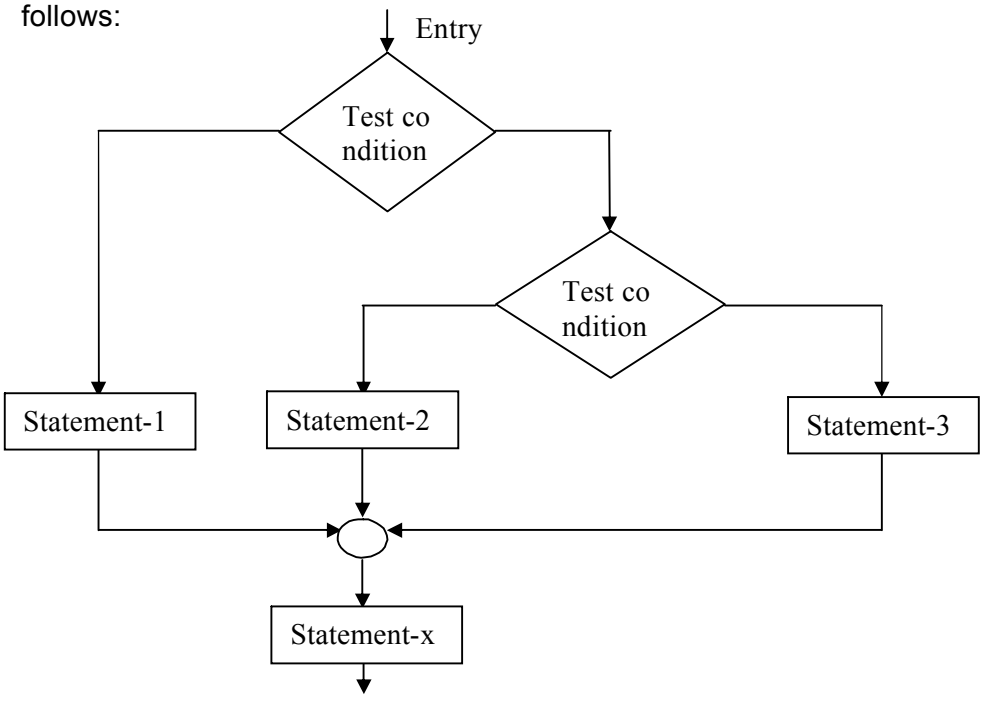
- When a series of decision are involved, we may have to use more than one if…..else statement in nested form as follows:



- The logic of execution is illustrated in above fig. the condition-1 is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the condition-2 true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement –x.
- Example:

```
Int  a=325,b=712,c=478;
System.out.println("largest value is :");
If(a>b)
{       If(a>c)
        {
                System.out.println("a");
        }
        Else
        {
                System.out.println("c");
        }
}
 else
{       If(c>b)
        {               System.out.println(c);       }
        else
          {             System.out.println(b);         }
}
```

- A commercial bank has introduced an incentive policy of giving bonus to all its deposit holder. The policy is as follows: a bonus of 2 per cent of the balance held on 31st December is given to every one, than Rs 5000. This logic can be coded as follows:

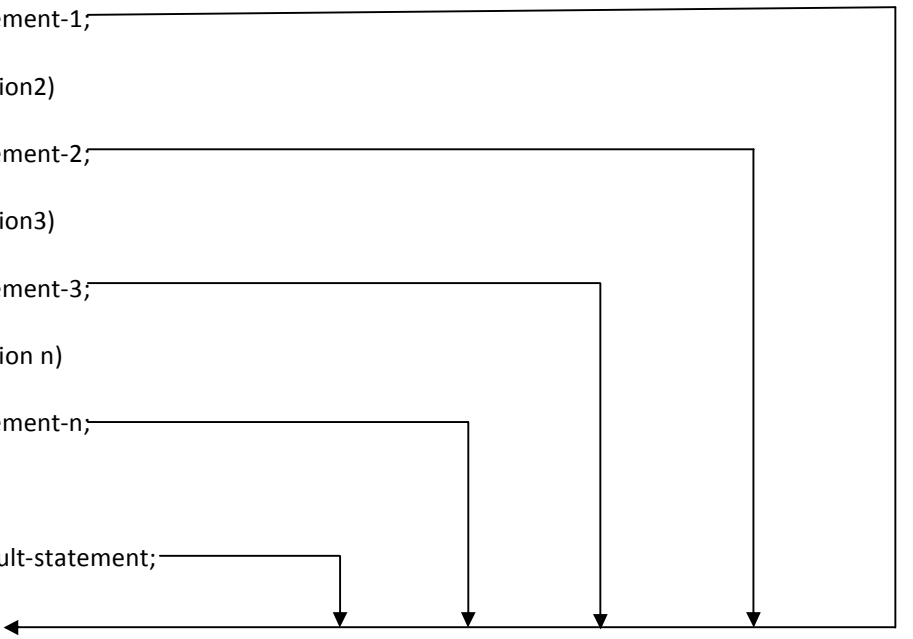


6.4 The if….else ladder

- There is another way of putting ifs together when multipath decisions are involved. A multipath decision is a chain of ifs in the statement which the statement associated with each else is an if. It takes the following general from:

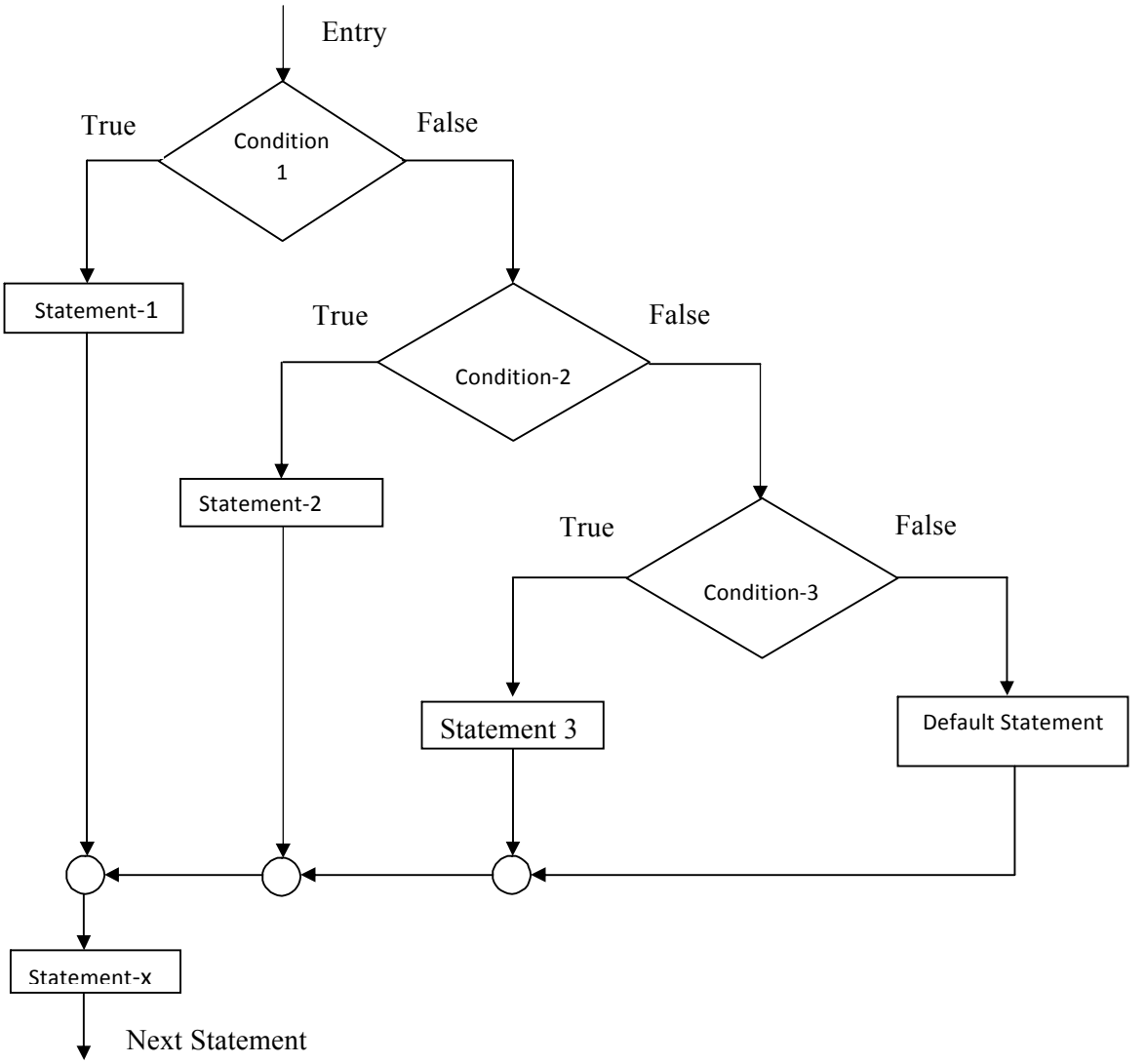
```
If(condition1)
{
        Statement-1;
}
Else if(condition2)
{
        Statement-2;
}
Else if(condition3)
{
        Statement-3;
}
Else if(condition n)
{
        Statement-n;
}
Else
{
        Default-statement;

Statement-x;
}
```



- The construct is known as the else if ladder. The condition are evaluated from the top (of the ladder), downwards. As soon as the true condition is found, the statement associated with is executed and the control is transferred to the statement-x (skipping the rest of the ladder). When all the n condition become false then the final else containing the default-statement will be executed above figure shows the logic of execution of else if ladder statements.
- Example:

```
Int a=10,b=51;
If(a>b)
{       System.out.println("a is bigger");
}
Else if(b>a)
{       System.out.println("b is bigger");
}
Else
{       System.out.println("both are same");
}
```

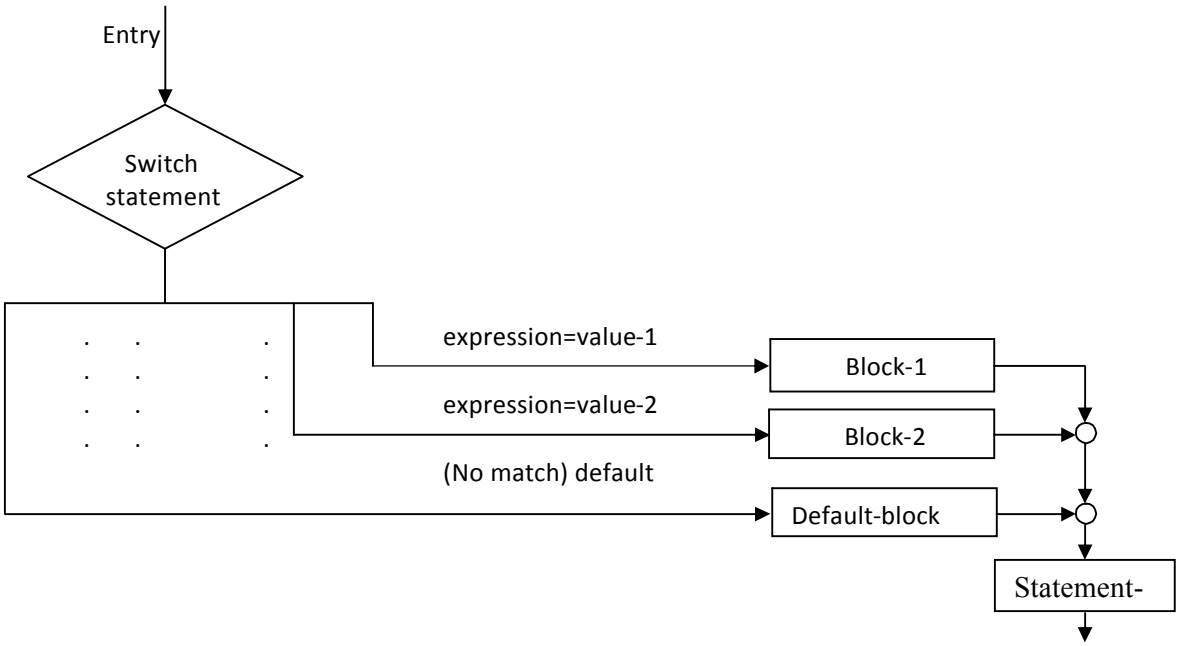


6.5 Switch statement

- We have seen that when one of the many alternatives is to be selected, we can design a program using if statement to control the selection. However, the complexity of such a program increases dramatically when the number of alternatives increases. The program becomes difficult to read and follow. At a times, it may confuse even the designer of the program. Fortunately, java has a built-in multiway decision statement known as switch. The switch statement tests the value of a given variable (or expression) against a list of case value and when a match is found, a block of statement associated with that case is executed. The general form of the switch statement is a shown below:

```
Switch(expression)
{
        Case value-1:
                Block-1
                Break;
        Case value-2:
                Block-2
                Break;
        ……..
        ……..
        Default:
                Default-block
                Break;
}
Statement-x;
```

- The selection process of switch statement is illustrated in the flowchart show in below:



7	**While** **do…while** **for & for each loop** **jumps in loops**

7.1 The While Statement

- The basic format of the while statement is

```
Initialization;
while (test condition)
{       Body of the loop        }
```

- The while is an entry-controlled ~op statement. The test condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.
- The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements.
- Consider the following code segment:

```
sum = 0;
n = 1;
whife (n<= 10)
(
sum =sum +n *n;
n = n + 1;
}
System.out.println ("Sum = " + sum);
```

- The body of the loop is executed 10 times for n = 1,2,…..,10 each time adding the square of the value of n, which is incremented inside the loop. The test condition may also be written as n < 11 the result would be the same.

7.2 THE DO STATEMENT

- The while loop construct that the condition will be tested first and then if condition is true will execute all the statements after while. On some occasions it might be necessary to execute the body of the loop before the rest performed. Such situation can be handled with the help of the do statement. This takes the form:

```
Initialization;
Do
{
        Body of the loop;
}
While(test condition)
```

- On reaching the do statement, the program proceeds to evaluate the body if the loop first. At the end of the loop, the test condition in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true when condition is false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.
- Since test condition is evaluated at the bottom of the loop, the do ... while construct provides and exit-controlled loop and therefore the body of the loop are always executed at the least once.

- Consider an example:

```
I=1;
Sum=0;

Do
{
        Sum=sum+I;
        I=i+2;
}
While(sum<40);
```

- The loop will be executed as long as the condition is true.

7.3 THE FOR STATEMENT

- The for loop is another entry-controlled loop that provides a more concise loop control structure. The general for is:

```
for ( initialization; test condition; increment)
{
body of the loop;
}
```

- The execution of the for statement is as follows:
  (1) Initialization of the control variables is done first, using assignment statements such as i=1 and count=0. The variable i and count are known as loop control variables.
  (2) The value of the control variable is tested using the test condition. The test condition is a relational expression, such as i<10 that determines when the loop will exit. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
  (3) When the body of the loop is executed, the control is transfer back to the "for" statement after evaluating the last statement in the loop. Now the control statement is incremented using an assignment statement such as i = i + 1 and the new (alue of control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. The process continues till the value of control variable fails to satisfy the test condition.
- Consider the following example:

```
for (x = 0 ; x < = 9 ; x = x + 1)
{
        System.out.print(x);
}
```

- The for loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons. Note that there is no semicolon at the end of the increment section.
- The for statement allows for negative Increments. For example,

```
for (x = 9 ; x <= 0; x = x - 1 )
        System.out.println(x);
```

- The for loop is execute 10 times and prints the digits 9 to 0 in one line. Note that braces are optional when the body of the loop contains only one statement.
- Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at first time. For example,

```
for(x=9; x< 9; x=x+1)
{
        System.out.print(x);
}
```

- will never be executed because the test condition fails at the very beginning itself.

Additional features of FOR LOOP:

- The "for loop" has several capabilities that are not found in other constructs. For example more than one variable can be initialized at a time in the “for" statement.

```
p = 1;
for (n = 0; n < 17; n ++)
```

can be rewritten as

```
for(n =0; p = 1; n < 17; n ++)
```

the initialization section has two parts p = 1 and n = 0 separated by comma.

- Like initialization section, the increment section, the increment section may also have more than one part. For example,

```
for (n = 0, p = 1; n < 17 ; n ++, m = m + 1)
```

is valid. The multiple arguments in the increment section are separated by commas.

- The third feature is that the test condition may have any compound relation and the testing need not be limited only to the loop control variable.

```
sum = 0;
for (n = 0; n < 17 && sum < 100 ; n++)
{
        ------------
        ------------
}
```

- The loop uses a compound test condition with the control variable n and external variable sum. The loop is executed as long as both the conditions n < 17 and sum < 100 are true.
- It is also permissible to use expressions In the assignment statements of initialization and Increment sections. For example, a statement of the type

```
for (x = (m+n)/2; x > 0; x = x/2) is valid.
```

- Also one can omit one of the sections, if required.

```
m = 5;
for ( ; m l= 100; )
{
        System.out.println(m);
        M=m+5;          }
```

Jumps in Loop

- Loop performs a set of operations repeatedly until the control variable fails to satisfy the test condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as certain condition occurs. Java permits a jump from one statement to the end or beginning of a loops as well as a jump out of a loop.

- **Jumping out of a loop**
- An early exit from a loop can be accomplished by using the break statement. This statement can also be used within while, do or for loop.
- When break statement encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, the break will exit only a single loop.
- The format of break statement is as follows:

```
break;
```

- **Skipping a part of loop:**
- During the loop operations, it may be necessary to skip a part of loop under certain conditions. Like break statement, Java supports another similar statement called the continue statement.
- However, unlike break which causes the loop to be terminated, the continue, as the name implies, causes the loop to be continues with the next iteration after skipping any statements in between. The continue statement tells the compiler, “SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATON". The format of continue statement is as follows:

```
continue;
```

- In the case of WHILE loop, continue causes the control to go directly to the test condition and then to continue the iteration process. In the case of for loop, the increment section of the loop is executed before the test condition is evaluated.

8 Array

- An array is a liked type variables that are referred by a common name.
- A specific element in an array is accessed by its index.

8.1 One-Dimensional array

- A one-dimensional array is, essentially, 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[ ];

- Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named month_days with the type “array of int”.
- You must allocate one using new and assign it to month_days. new is a special operator that allocates memory. You will look more closely at new in a later chapter, but you need to use it now to allocate memory for arrays. The general form of new as it applies to one-dimensional arrays appears as follows:
- array-var = new type[size];
- Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array. That is, to use new to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by new will automatically be initialized to zero.
- This example allocates a 12-element array of integers and links them to month_days.

```
month_days = new int[12];
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. The process is much the same as
- that used to initialize the simple types. An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate 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 Average {
    public static void main(String args[]) {
        double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
        double result = 0;
        int i;
        for(i=0; i<5; i++)
        {
            result = result + nums[i];
            System.out.println("Average is " + result / 5);
        }
    }
}
```

8.2 Multidimensional Arrays(Two Dimensional Arrays)

- In Java, 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, the following declares a two-dimensional array variable called twoD.

```
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**.

1 Defining a class, members of a class: variables and methods, creating objects, constructors, accessing class members

1.1 Defining a Class:

As stated earlier a class is a user-defined data type with a template that serves to define its properties. Once the class type has been defined, we can create "variables" of that using declaration that are similar to the basic type declaration . In java , these variables are terms as instances of classes, which are the actual objects. The basic form of a class definition is:

```
class classname [extends superclassname] [implements Interfacename]
{
        [ fields declaration ; ]
    [ methods declaration ; ]
 }
```

Everything inside the square brackets is optional. This means that the following would be a valid class definition:

```
class Empty
{
}
```

Because the body is empty, this class dose not contains any properties and therefore cannot do anything. We can, however compiler it and even create objects using it . C++ programmers may note that there is no semicolon after closing brace.

classname and *superclassname* are any valid java identifiers. The keyword extends indicates that the properties of the superclassname class are extended to the classname class. This concept is known as inheritance.

1.2 FIELDS DECLARATION:

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called instance variables because they are created whenever an object of the class is instantiated. We can declare the instance variables exactly the same way as we declare local variables. Example:

```
class Rectangle
{
    int length;
    int  width;
}
```

The class Rectangle contains two integer type instance variables. It is allowed to declare them in one line as.

```
int length, width;
```

Remember these variables are only declared and therefore no storage space has been created in the memory. Instance variables are also known as member variables.

1.3 METHODS DECLARATION:

A class with only data fields (and without methods that operator on that data) has no life. The objects created by such a class cannot respond to any messages. We must therefore add methods that are necessary for manipulating the data contained in the class. Methods are declared inside the body of the class but immediately after the declaration of instance variables. The general form of a methods declaration is

```
type methodname(parameter-list)
{
        Method-body;
}
```

Methods declarations have four basic parts:

1) The name of the method(methodname)
2) The type of the value the method returns(type)
3) A list of parameters(parameter-list)
4) The body of the method.

The type specifies the type of value the methods would return. This could be a simple data type such as int as well as any class type. It could even be void type, if the method does not returns any value. The *methodname* is a valid identifier. The parameter list is always enclosed in parentheses. This list contains variables names and types of all the values we want to give to the methods as input. The variables in the list are separated by commas. In the case where no input data are required, the declaration must retain the empty parentheses. Examples:

```
(int m, float x , float y)      // three parameters
( )                             //  Empty list
```

The body actually describes the operations to be performed on the data. Let us consider the rectangle class again and add a methods getdata() to it.

```
class Rectangle
{
        int length;
        int width;
        void getData (int x, int y )  // methods declaration                    {
        {
                length=x;
                width=y;
        }
}
```

Note that the method has a return type of **void** because it does not return any value. We pass two integer values to the method which are then assigned to the instance variables length and width. The getdata method is basically added to provide values to the instances variables. Notice that we are able to use directly length and width inside the method.

Let us some more properties to the class. Assume that we want to computer the area of the rectangle defined by the class. This can be done as follows:

```
class Rectangle
{
        int length,width;       //      combined declaration
        void getdata(intx,int y)
        {
                length =x;
                width =y;
        }
    int rectArea() // declaration  of another method
    {
        int area= length *width;
        return(area);
    }
}
```

The new method rectArea() computes area of the rectangle and returns the result. Since the result would be an integer, the return type of the method has been specified as int. also note that the parameter list is empty, remember that while the declaration of instance variable(and also local variables) can be combined as

int length, width;

The parameter list used in the method header should always be declared independently separated by commas, that is,

void getData (int x, y) //incorrect is illegal.

1.4 CREATING OBJECTS:

As pointed out earlier, an object in java is essentially a block of memory that contains space to store all the instance variables. Creating an object is also referred to as *instantiating* an object.

Objects in java are created using **new** operator. The **new** operator creates an object of the specified class and returns a reference to that object. Here is an example of creating an object of type **Rectangle**.

```
Rectangle rect1;            //declare the object
rect1=new Rectangle();      //instance the object
```

The first statement declared a variable to hold the object references and the second one actually assigns the objects references to the variables. The variable **rect1** is now an object of Rectangle class.

Both statements can be combined into one statement as below:

```
Rectangle  rect1 = new  Rectangle();
```

The method Rectangle() is the default constructor of the class. We can create any number of objects of Rectangle. Example:

```
Rectangle rect1= new Rectangle();
```

Rectangle rect2= new Rectangle();

It is important to understand that each object has its own copy of the instance variables of its class. This means that any changes to the variables of one object have no effect on the variables of another. It is also possible to create two or more references to the same object.

1.5 ACCESSING CLASS MEMBERS:

NOW we have created objects each containing its own set of variable, we should assign values to these variable in order to use them in our program. Remember all variable must be assigned values before they are used. Since we are outside the class., we cannot access the instance variable and the methods directly. To this, we used the concerned object and the dot operator a shown below:

```
Objectname.variablename=value;
Objectname.methodname(parameter-list);
```

Here objectname is the name of the object,variablename is the name of the instances variable inside the object that we wish to access, methodname is the method that we wish to call, and paremeter-list is a comma separated list of “actual values” that must match in type and number with the parameter list of the methodname declared in the class. The instances variable of the Rectangle class may be accessed and assigned the values as follow:

```
rect1.length=15;
rect1.width=10;
rect2.length=20;
rect2.width=12;
```

note that the two objects rect1 and rect2 store different value as shown below:

```
rect1.length=15;        rect2.length=20;
rect1.width=10;         rect2.width=12;
```

This is one way of assigning values to the variables in the objects. Another way and more convenient way of assigning values to the instance variable is to use a method that is declared inside the class.

In our case, the method getData can be used to do this work.we can call the getData method on any Rectangle object to set the values of both length, width. here is code segment to achieve this.

```
Rectangle rect1=new Rectangle();
rect1.getdata(15,10);
```

This code creates rect1 object and then passes in the values 15 and 10 for x and y parameter of the method gatdata. This method then assigning these values to length and width variables respectively . for the sake of convenience, the method is again shown below:

```
void getData( int x, int y)
```

```
{
        length=x;
        width=y;
}
```

Now that the object rect1 contains values for its variables , we can compute the area of the rectangle represented by rect1. This again can be done in two ways.

1) The first approach is to access the instance variables using the dot operator and compute the area. that is,
   int area1=rect1.length * rect1.width;
2) The second approach is to call methodrectarea declared inside the class. That is ,
   int area1=rect1.rectArea();

1.6 CONSTRUCTORS

We know that all objects that are created must be given initial values. We have done this earlier using two approaches. The first approach uses the dot operator to access the instance variables and then assigns values to them individually. It can be a tedious approach to initialize all the variables of all the objects.

The second approach takes the help of a method like **getData** to initialize each object individually using statements like,

```
rect1.getData(10,15);
```

It would be simpler and more concise to initialize an object when it is first created. Java supports a special type of method, called a constructor, that enables an object to initialize itself when it is created.
Constructors have the same name as the class itself. Secondly, they do not specify a return type, not even **void.** This is because they return the instance of the class itself.
Let us consider our **Rectangle** class again.

We can now replace the **getData** method by a constructor method as shown below :

```
class Rectangle
{
        int length;
        int width;
        Rectangle (int x, int y)  // Constructor method
        {
                length = x;
                width = y;
        }
        intrectArea()
        {
                return (length * width);
```

```
        }
    }
```

Program illustrates the use of a constructor method to initialize an object at the time of its creation.

Program *Application of constructors*

```
class Rectangle{
    int length, width ;
    Rectangle (int x, int y)                    // Defining constructor
    {
        length = x;
        width = y;
    }
    intrectArea() {
        return (length * width) ;
    }
}
class RectangleArea{
    public static void main (String args[ ])    {
        Rectangle rect1 = new Rectangle(15, 10);        //Calling constructor
        int area1 = rect1.rectArea();
        System.out.println("Area1 = "+ area1);
    }
}
```

Output of Program

```
        Area1 = 150
```

The constructor Rectangle defined in Program 8.2 can also be termed as *parameterized constructor*. This is because, at the time of object instantiation, the constructor is explicitly invoked by passing certain arguments. But, what if we want the constructor to automatically initialize the object variables with some default values at the time of object instantiation. The *default constructor* is used in such a situation.
It is declared in the same manner as a parameterized constructor with one characteristic difference that it does not take in any parametric values.

Program shows the use of default constructor :

```
class perimeter{
    int length;
    int width;
    perimeter()   {
        length = 0;
        breadth = 0;
```

```
        }
        // parameterized constructor
        perimeter(int x, int y)
        {
                length = x;
                breadth = y;
        }
        void cal_perimeter()
        {
                intperi;
                peri=2*(length + breadth);
                System.out.println("\nThe perimeter of the rectangle is :"+peri);
        }
}
class Ex_default_c{
        public static void main(String args[ ])      {
                perimeter p1=new perimeter();    // calling default constructor
                perimeter p2=new perimeter(5, 10); //calling parameterized constructor
                p1.cal_perimeter();
                p2.cal_perimeter();
        }
}
```

Output of Program 8.3 :

```
        The Perimeter of the rectangle is : 0
        The Perimeter of the rectangle is : 30
```

2 Static members v/s instance members

We have seen that a class basically contains two sections. One declares variables and the other declares methods. These variables and methods are called *instance variables* and *instance methods.* This is because every time the class is instantiated, a new copy of each of them is created. They are accessed using the objects (**with dot operator**).

Let us assume that we want to define a member that is common to all the objects and accessed without using a particular object. That is, the member belongs to the class as a whole rather than the objects created from the class. Such members can be defined as follows:

static int count;
static int max (int x, int y);

The members that are declared **static** as shown above are called *static members.* Since these members are associated with the class itself rather than individual objects, the static variables and static methods are often referred to as *class variables* and *class methods* in order to distinguish them from their counterparts, instance variables and instance methods.

Static variables are used when we want to have a variable common to all instances of a class. One of the most common examples is to have a variable that could keep a count of how many objects of a class have been created. Remember, Java creates only one copy for a static variable which can be used even if the class is never actually instantiated.

Like static variables, static methods can be called without using the objects. They are also available for use by other classes. Methods that are of general utility but do not directly affect an instance of that class are usually declared as class methods. Java class libraries contain a large number of class methods. For example, the **Math** class of Java library defines many static methods to perform math operations that can be used in any program. We have used earlier statements of the types.

```
float  x = Math.sqrt(25.0);
```

The method **sqrt()** is a class method (or static method) defined in **Math** class.

We can define our own static methods as shown in Program 8.4.

Program : *Defining and using* static members

```
class Mathoperation
{
        static float mul (float x, float y)
        {
                return x*y;
        }
        static float divide (float x, float y)
        {
                return x/y;
        }
}

class MathApplication
{       public void static main (string args [ ])
        {
                float a = Mathoperation.mul(4.0,5.0);
                float b = Mathoperation.divide(a,2.0);
                System.out.println("b = "+ b);
        }       }
```

Output of Program 8.4 :

```
b = 10.0
```

Note that the static methods are called using class names. In fact, no objects have been created for use. Static methods have several restrictions :

1. They can only call other **static** methods.
2. They can only access **static** data.
3. They cannot refer to **this** or **super** in way.

3 Introduction to inheritance, super keyword

Reusability is yet another aspect of OOP paradigm. It is always nice if we could reuse something that already exist rather than creating the same all over again. java supports this concept. Java classes can be reused in several ways. This is basically done by creating new classes, reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called *inheritance.* The old class is known as the *base class* or *super class* or *parent class* and the new one is called the *subclass* or *derived class* or *child class.* The inheritance allows subclasses to inherit all the variables and methods of their parent classes. Inheritance may take different forms:

- Single inheritance (only one super class)
- Multiple inheritance (several super classes)
- Hierarchical inheritance (one super class, many subclasses)
- Multilevel inheritance (Derived from a derived class)
  Java does not directly implement multiple inheritance. However, this concept is implemented using a secondary inheritance path in the form of interfaces.

Defining a Subclass

A subclass is defined as follows :

```
class subclassname extends superclassname
{
        variables declaration;
        methods declarartion;
}
```

The keyword **extends** signifies that the properties of the superclassname are extended to the subclassname. The subclass will now contain its own variables and methods as well those of the superclass. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it. Program 8.6 illustrates the concept of single inheritance.

Program Application of single inheritance

```
class Room
{
        int length;
        int width;
        Room (int x, int y)
        {
                length = x;
                breadth = y;
        }
        int area()
        {
                return (length * breadth);
        }
}
```

```
class BedRoom extends Room                    //Inheriting Room
{
        int height;
        BedRoom (int x, int y, int z)
        {
                super (x, y)                  //pass values to superclass
                height = z;
        }
        int volume ( )
        {
                return (length * breadth * height);
        }
}
class InherTest
{
        public static void main (String args [ ])
        {
                BedRoom room1 = new BedRoom ( 14, 12, 10);
                int area1 = room1.area( );          //superclass method
                int volume1 = room1.volume ( );     //subclass method
                System.out.println("Area1 = "+ area1);
                System.out.println("Volume1 = "+ volume1);
        }
}
```

The output of Program 8.6 is :

```
Area1 = 168
Volume1 = 1680
```

The program defines a class **Room** and extends it to another class **BedRoom**. Note that the class **BedRoom** defines its own data members and methods. The subclass BedRoom now includes three instance variables, namely, **length, bredth**and **height** and two methods, **area** and **volume.**

The constructor in the derived class uses the **super** keyword to pass values that are required by the base constructor. The statement

```
BedRoom room1 = new BedRoom (14, 12, 10);
```

calls first the **BedRoom** constructor method, which in turn calls the **Room** constructor method by using the **super** keyword.

Finally, the object **room1** of the subclass **BedRoom** calls the method **area** defined in the super class as well as the method volume defined in the subclass itself.

Subclass Constructor

A subclass constructor is used to construct the instance variables of both subclass and the superclass. The subclass constructor uses the keyword **super** to invoke the constructor method of the superclass. The keyword **super** is used subject to the following conditions.

- **Super** may only be used within a subclass constructor method
- The call to superclass constructor must appear as the first statement within the subclass constructor
- The parameter in the **super** call must match the order and type of the instance variable declared in the superclass.

Program illustrated the use of **super( )** method for passing parameter to a superclass.

```
class  Super
{
        int x;
        Super (int  x)
        {
                this.x = x;
        }
        void display( )                                  //method defined
        {
                System.out.println("Super x = "+ x);
        }
}
class Sub extends Super
{
        int y;
        Sub (int x, int y)
        {
                Super (x);
                this.y = y;
        }
        void display( )                                  //method defined again
        {
                System.out.println("Super x = " + x);
                System.out.println("Sub y = " + y);
        }
}
class OverrideTest
{
        public static void main (String args [ ])
        {
                String cname;
                String tname;
                Cname=br.readLine();
```

```
            Sub s1 = new Sub (100, 200);
            s1.display( );
        }
    }
```

The output of Program:

```
Super x = 100
Sub y = 200
```

Multilevel Inheritance

A common requirement in object-oriented programming is the use of a derived class as a super class. Java supports this concept and uses it extensively in building its class library. This concept allows us to build chain of classes.

The class **A** serves as a base class for the derived class **B** which in turn serves as a base class for the derived class **C.** The chain ABC is known as inheritance path.

A derived class with multilevel base classes is declared as follows.

```
class A
{
        . . . . . .
        . . . . . .
}
class B extends A    // First level
{
        . . . . . .
        . . . . . .
}
class C extends B    //Second level
{
        . . . . . .
        . . . . . .
}
```

Hierarchical Inheritance

Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below the level. A hierarchical classification of accounts in a commercial bank. This is possible because all the accounts possess certain common features.

```
class A
{
        . . . . . .
        . . . . . .
}
class B extends A
{
        . . . . . .
        . . . . . .
}
class C extends A          {
        . . . . . .
        . . . . . .
}
```

4 Interfaces: introduction

Introduction: Java does not support multiple inheritance, that is, classes in java cannot have more than one super class. However, the designers of java could not overlook the importance of multiple inheritance. Java provides an alternate approach known as Interfaces to support the concept of multiple inheritance.

Defining Interfaces:

An interface is basically a kind of class. It contains methods and variables but with a major difference. The difference is that interfaces define only abstract methods and final fields. This means that interfaces donot specify any code to implement these methods and data fields contain only constants. Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods.

Syntax for defining interface:

```
interface Interfacename
{
    variables declaration;
    methods declaration;
}
```

Here, interface is the keyword and Interfacename is any valid java variable.
Variables are declared as follows:

```
static final type variablename = value;
```

Note that all variables are declared as constants.

Methods declaration will contain only a list of methods without any body statements.

```
return-type methodname1(parameter_list);
```

Example:

```
interface item
{
        static final int code = 1001;
        static final String name = "Fan";
        void display();
}
```

Implementing interfaces

Interfaces are used as "superclasses" whose properties are inherited by classes. It is therefore necessary to create a class that inherits a given interface.

Syntax:

```
class classname implements interfacename
{
        //body of classname
}
```

More general form of implementation is like this:

```
class classname extends superclass implements interface1, interface2,…..
{
        //body of classname
}
```

5 Final variables, methods and classes, abstract methods and classes

FINAL VARIABLES AND METHODS

All methods and variables can be overridden by default in subclasses. If we wish to prevent the subclasses from overriding the members of the superclass, we can declare them as final using the keyword **final** as a modifier. Example :

```
final int SIZE = 100;
final void showstatus( ) { . . . . . .}
```

Making a method final ensures that the functionality defined in this method will never be altered in any way. Similarly, the value of a final variable can never be changed. Final variables, behave like class variables and they do not take any space on individual objects of the class.

FINAL CLASSES

Sometimes we may like to prevent a class being further subclasses for security reasons. A class that cannot be subclassed is called a final class. This is achieved in Java using the keyword **final** as follows :

```
final class Aclass { . . . . . }
finalclass Bclass extends Someclass { . . . . . }
```

Any attempt to inherit these classes will cause an error and the compiler will not allow it. Declare a class **final** prevents any unwanted extensions to the class. It also allows the compiler to perform some optimisations when a method of a finalclass is invoked.

ABSTRACT METHODS AND CLASSES

We have seen that by making a method **final** we ensure that the method is not redefined in a subclass. That is, the method can never be subclassed. Java allows us to do something that is exactly opposite to this. That is, we can indicate that a method must always be redefined in a subclass, thus making overriding compulsory. This is done using the modifier keyword **abstract** in the method definition. Example :

```
abstractclass Shape
{
        . . . . . .
        . . . . . .
        abstract void draw ( );
        . . . . . .
        . . . . . .
}
```

When a class contains one or more abstract methods, it should also be declared **abstract** as shown in the example above.

While using abstract classes, we must satisfy the following conditions :

- We cannot use abstract classes to instantiate objects directly. For example,
  Shape s = new Shape ( )
  is illegal because **shape** is an abstract class.
- The abstract methods of an abstract class must be defined in its subclass.
- We cannot abstract constructors or abstract static method.

6 Introduction to method overloading and overriding

METHODS OVERLOADING

In Java, it is possible to create method that have the same name, but different parameter lists and different definitions. This is called *method overloading*. Method overloading is used when objects are required to perform similar tasks but using different input parameter. When we call a method in an object, Java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as *polymorphism*.

To create an overloaded method, all we have to do is to provide several different method definitions in the class, all with the same name, but with different parameter lists. The difference may either be in the number or type of arguments. That is, each parameter list should be unique. Note that the method's return type does not play any role in this. Here is an example of creating an overloaded method.

```
class Room{
        float  length;
        float  breadth;
        Room (float x, float y)                                  //constructor1
        {
                length = x;
                breadth = y;
        }
        Room (float x)                                           //constructor2
        {
                length = breadth = x;
        }
        int area ()     {
                return (length * breadth);
        }
}
```

Here, we are overloading the constructor method **Room( ).** An object representing a rectangular room will be created as

```
Room room1 = new Room(25.0 , 15.0) ;        //using constructor1
```

On the other hand, if the room is square, then we may create the corresponding object as

```
Room room2 = new Room(20.0) ;               //using constructor2
```

OVERRIDING METHODS

We have seen that a method defined in a super class is inherited by its subclass and is used by the objects created by the subclass. Method inheritance enables us to define and use methods repeatedly in the subclasses without having to define the methods again in subclass.

However, there may be occasions when we want an object to respond to the same method but have different behaviour when that method is called. That means, we should override the method defined in the superclass. This is possible by defining a method in the subclass that has the same name, same arguments and same return type as a method in the superclass. Then, when that method is called, the method defined in the subclass is invoked and executed instead of the one in the superclass. This is known as overriding. Program 8.7 illustrates the concept of overriding. The method**display( )** is overridden.

Program Illustration of method overriding

```
class  Super
{
        int x;
        Super (int  x)
        {
                this.x = x;
        }
```

```
        void display( )                          //method defined
        {
                System.out.println("Super x = "+ x);
        }
}
class Sub extends Super
{
        int y;
        Sub (int x, int y)
        {
                super (x);
                this.y = y;
        }
        void display( )                          //method defined again
        {
                System.out.println("Super x = " + x);
                System.out.println("Sub y = " + y);
        }
}
class OverrideTest
{
        public static void main (String args [ ])
        {
                Sub s1 = new Sub (100, 200);
                s1.display( );
        }
}
```

The output of Program 8.7 :

```
        Super    x    =    100
        Sub      y    =    200
```

Note that the method display ( ) defined in the subclass is invoked.

1	Managing errors & exceptions: introduction, types of errors, exceptions, syntax of exception handling construct, multiple catch statements, the finally clause, defining and throwing user-defined exceptions, the throw statement

Rarely does a program run successfully at its very first attempt. It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing to program to produce unexpected results. Errors are the wrong that can make a program go wrong.

An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible error condition in the program so that the program will not terminate or crash during execution.

TYPES OF ERRORS

Errors may broadly be classified into two categories:

- Compile-time errors
- Run-time errors

Compile-Time Errors

All syntax errors will be detected and displayed by the java compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the **.class**class. it is therefore necessary that we fix all the errors before we can successfully compile and run the program.

Next Program illustration of compile-time errors

```
/* this program contains an error*/
class Error1
{
        public static void main ( String args [ ] )
        {
            System.out.println(“Hello java!”)          // Missing;
        }
}
```

The java compiler does a nice job of telling us where the errors are in the program. For example, if we have missed the semicolon at the end of print statement in program ,the following message will be displayed in the screen:

```
Error1. java  :7: ‘;’ expected
System.out.println (“Hello java!”)
                    ^
                   1 error
```

We can now go to the appropriate line, correct the error, and recompile the program. Sometimes, a single error may be the source of multiple errors later in the compilation. For example, use of an undeclared variable in a number of places will cause a series of errors of type “undefined variable”. We should generally consider the earliest errors as the major source of our problem. After we fix such an error, we should recompile the program and look for other errors.

Most of the compile-time errors are due to typing mistakes. Typographical errors are hard to find. We may have to check the code word by word, or even character. The most common problems are:

- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments / initialization
- Bad references to objects
- Use of = in place of == operator
- And so on

Other errors we may encounter are related to directory paths. An error such as

Javac : command not found

means that we have not set the path correctly. We must ensure that the path includes the directory where the java executables are stored.

Run-Time Errors

Sometimes, a program may compile successfully creating the .**class**file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time errors are:

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible class or type
- Trying to cast an instance of a class to one of its subclasses
- Passing a parameter that is not in a valid rang or value for a method
- Trying to illegally change the state of a thread
- Attempting to use a negative size for an array
- Using a null object reference as a legitimate object reference to access a method or a variable.
- Converting invalid string to a number
- Accessing a character that is out of bounds of a string
- And may more

When such errors are encountered, java typically generates an error message and aborts the program. Program illustrates how a run-time error causes termination of execution of the program.

Next Program illustration of run-time errors

```
class Error2{
        public static void main(String args[]){
                int a=10;
                int b=5;
                  int c=5;
                int x=a/(b-c);     //Division by zero
                System.out.println(x=”+x);
                int y=a/(b+c);
                System.out.prinln(“y=”+y);
          }
    }
```

Above Program is syntactically correct and therefore does not cause any problem during compilation.However while executing, it display the following message and stops without executing further statements.

```
Java.lang.ArithmeticException: /by zero
      At Error2.main(error2.java:10)
```

When java run-time tries to execute a division by zero, it generates an error condition, which causes the programe to stop after displaying an appropriate message.

EXCEPTIONS

An exception is a condition that is caused by a run-time error in the program. When the java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it(i.e.informs us that an error has occurred).

If the execution object is not caught and handled properly, the interpreter will display an error message as shown in the output of above program and will terminate the program. If we want the program to continue with the exception of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as exception handling.

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

1. Find the problem (**Hit**the exception)
2. Inform that an error has occurred (**Throw**the exception)
3. Receive the error information (**catch**the exception)
4. Take corrective actions (**Handle**the exception)

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exception and to take appropriate actions.

When writing programs, we must always be on the lookout for places in the program where an exception could be generated. Some common exception that we must watch out for catching are listed in table

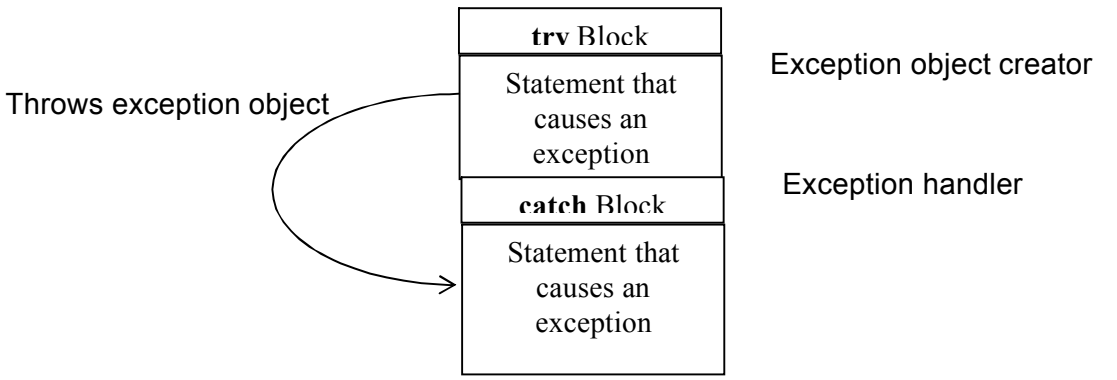
Exception Type	Cause of Exception
ArithmeticException	Caused by math errors such as division by zero
ArrayIndexOutOfBoundsException	Caused by bad array indexes.
Array StoreException	Caused when a programe tries to store the wrong type of data in an array
FileNot FoundException	Caused by an attempt to access a nonexistent file
IoException	Caused by general I/O failures, such as inability to read from a file
NullPointerException	Caused by referencing a null object
NumberFormatException	Caused when a conversion between strings and number fails
OutOfMemoryException	Caused when there's not enough memory to allocate a new object
SecurityException	Caused when an applet tries to perform an action not allowed by the browser's security setting
StackOverFlowException	Caused when the system runs out of stack space
StringIndexOutOfBoundsException	Caused when a program attempts to access a nonexistent character position in a string

Exception in java can be categorized into two types:

- **Checked exception:**These exception are explicitly handled in the code itself with the help of try-catch blocks. Checked exceptions are extended from the **java.lang.Exception**class.
- **Unchecked exception:** these exception are not essentially handled in the programe code; instead the JVM handles such exceptions. Unchecked exception are from the **java.lang.RuntimeException** class.

it is important to note that checked and unchecked exception are absolutely similar as far as their functionality is concerned; the difference lies only in the way they are handled.

SYNTAX OF EXCEPTION HANDLING CODE



Java use a keyword **try**to preface a block of code that is likely to cause an error condition ans throw" an exception. A catch block defined by the keyword **catch**"catches" the exception thrown" by thee try block and handles it appropriately. The catch block is added immediately after the try block. The following example illustrates the use of simple **try** and **catch** statements:

```
try{
        Statement;    // generates an exception
    }
catch  (Exception-type  e )
       {
                Statement;    // processes the exception
       }
     ……..
     ……
```

The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining, the remaining statements in the block are skipped and exception jumps to the catch block that is placed next to the try block.

The catch block too can have one or more statements that are necessary to process the exception. Remember that every **try** statement should be followed by at least one **catch** Statement; otherwise compilation error will occur.

Note that the catch statements works like a method definition. The catch statement is passed a single parameter, which is reference to the exception object thrown (by the try block). If the parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught and the default exception handler will cause the execution to terminate.
Next Program illustrates the use of try and catch blocks to handle an arithmetic exception. Note that program is a modified version of last program.

This Program Using try and catch for exception handling

```
Class Error3
{
        public static void main (String args[])
        {
                int a=10;
                int b=5;
                int c=5;
                int  x  , y ;
                try
                {
                        x=a/(b-c);     //Exception here
                }
```

```
            catch (ArithmeticException e)
                    {
                    System.out.println("Division by zero");
                    }
            y=a/(b+c);
            System.out.prinln("y="+y);
        }
    }
```

Program display the following output:

```
Division by zero
Y = 1
```

Note that the program did not stop at the point of exceptional condition. It catches the error condition, prints the error message, and then continues the execution, as if nothing has happened. Compare with the output of last program which did not give the value of y.

Next Program shown another example of using exception handling mechanism. Here, the try-catch block catches the invalid entries in the list of command line arguments.

```
class CLineInput{
    public static void main (String args[]){
            int invalid= 0 ; // Number of invalid arguments
            int number, count=0;
            for (int i=0;i<args.length;i++){
                    try{
                            Number = Integer.parseInt(args[i]);
                      }
                      catch ( NumberFormatException e){
                              Invalid =invalid +1 ;//Caught an invalid number
                                System.out.println("invalid number:"+ arg[i]);
                                //Skip the remaining part of the loop
                               Continue;
                       }
                      count = count+1;
             }
            System.out.println("valid numbers = +count);
           Sysatem.out.println(invalid numbers = +invalid);
        }
}
```

Note the use of the wrapper class **integer**to obtain an **int** number from a string:

```
Number = Integer.parseInt (args[i])
```

Remember that yhe number are supplied to the programe through the command line and therefore they are stored as string in the array **args[].** Since the above statement is placed in the try block, an exeption is thrown if the string is improperly formatted and the number is not included in the count.

When we run the program with the command line:

Java ClineInput 15 25.75 40 java 10.5 65

It produces the following output:

Invalid number: 25.75
Invalid number: java
Invalid number: 10.5
Valid numbers = 3
Invalid numbers = 3

There could be situations where there is a possibility of generation of multiple exception of different types within a particular block of the program code. We can use nested try statements in such situations. The execution of the corresponding catch blocks of nested try statements is done using a stack. The program below showed the example of nested try statements:

Program of Nested try statements

```
class eg_nested_try{
        public static void main(String args[]){
            try{
                    int a=2,b=4,c=2,x=7,z;
                    int p[]={2};
                    int p{3}=33;
              try {
                                z=x/(b*b)-(4*a*c));
                                System.out.println("the value of z is ="+z);
                    }
               catch (ArithmeticException e){
              System.out.println("Division be zero in Arithmetic expression");
                        }
               }
          }
     catch (ArrayIndexOutOfBounExcception e){
                System.out,println("array index is out-of-bounds);
      }
  }
}
```

Program displays the following output:

Array index out-of-bounds

MULTIPLE CATCH STATEMENTS

It is possible to have more than one catch statement in the catch block as illustrated below:

```
        try{
            statement  ;                    //generates an exception
          }
       catch  (Exception-Type-1 e){
            Statement;                  //processes exception type 1
```

```
        }
      catch  (Exception-Type-2 e){
            Statement;              //processes exception type 2
       }
      catch  (Exception-Type-N e){
            Statement;              //processes exception type N
    }
```

When an exception in a try block is generated, the java treats the multiple catch statement like cases in a switch statement. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.
Note that java does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion.

Example:

```
catch (Exception e);
```

the catch statement simply ends with a semicolon, which does nothing. This statement will catch an exception and then ignore it.

Next Program Using multiple catch blocks

```
class Error4
{
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);
             }
    }
```

Above Program uses a chain of catch blocks and, when run, produces the following output:

Array index error
Y = 2

Note that the array element a[2] does not exist because array a is defined to have only two elements, a[0] and a[1]. Therefore, the index 2 is outside the array boundary thus causing the block

Catch (ArrayIndexOutOfBoundsException e)

to catch and handle the error. Remaining catch blocks are skipped.

USING FINALLY STATEMENT

Java supports another statement known as **finally**statement that can be used to handle an exception that is not caught by any of the previous catch statements,**finally**block can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block shown as follows:

```
try
{
        ……..
        ……..
}

finally
{
……..
        ……..
}
```

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

When a **finally**block id defined, this is guaranteed to execute, regardless of whether or not in exception is thrown. As a result, we can use it to perform certain house-keeping operations such as closing files and releasing system resources.

In last Program, we may include the last two statements inside a **finally** block as shown below:

```
Finally         {
                int y = a[1]/a[0];
                System.out.prinln("Y=" + y);
        }
```

This will produce the same output.

THROWING OUR OWN EXCEPTION

There may be times when we would like to throw our own exceptions. We can do this by using the keyword **throw** as follows:

throw new **Throwabl'e subclass;**

Examples:

thrown new ArithmeticException () ;
thrown new NumberFormatException () ;

program demonstrates the use of a user-defined subclass of Throwable class. Note that **Exception**is a subclass of **Throwable** and therefore **MyException**is a subclass of **Throwable**class. An object of a class that extends **Throwable**can be thrown and caught.

Next Program Throwing our own exception

```
import java.lang.Exception;
class MyException extends Exception
{       MyException (String message)
        {       super(messge);
        }
}
class TestMyException
 {
        public static void main (String args[])
        {
                int  x=5, y=1000;
                try
                {
                        float z= (float) x / (float) y;
                        if (z < 0.01)
                        {
                                throw new MyException("Number is too small");
                        }
                }
                catch (MyException e)
                {
                        System.out.println("Caught my exception");
                System.out.println( e.getMessage() );
                }
                finally
                {System.out.println(" I am always here");
                }
        }
}
```

A run of above program produces:

Caught my exception
Number is too small
I am always here

The object e which contain the error message “Number is too small” is caught by the **catch** block which then displays the message using the getMessage() method.
Note that above Program also illustrates the use of **finally** block. The last line of output is produced by the **finally** block.
There could be situations where there is a possibilitythat a method might throw certain kinds of exception but there is no exception handling mechanism prevalent within the method. In such a case, it is important that the method caller is intimated explicitly that certain types of exceptions could be expected from the called method, and the caller must get prepared with some catching mechanism to deal with it.
The **throws** clause is used in such a situation. It is specified immediately after the method declarationstatement and just before the opening brace. The Program 13.8 shown an example of using the throws clause:
Program : Use of throws

```
class Example throws
{static void divide_m() throws ArithmeticException
{int x = 22 , y = 0.2;
z = x/y;
}
public static void main (String args[])
{
try
{
divide_m();
}
catch (ArithmeticException e)
{
System.out.println(“Caught the exception”+e);
}
}
 }
```

Above Program displays the following output:

Caught the exception java.lang.ArithmeticException: /by zero

2 Managing I/O files : introduction, concept of streams, Character stream classes

Introduction

Stream is an abstract demonstration of input or output device. By using stream, we can write or read data. To bring in information, a program is open a stream on an information source (a file, memory, a socket) and read information sequentially. In this unit, we will learn the concept of stream, I/O package.

Concept of Stream:

The Java Input/Output (I/O) is a part of **java.io** package. The **java.io** package contains a relatively large number of classes that support input and output operations. The classes in the package are primarily abstract classes and stream-oriented that define methods and subclasses which allow bytes to be read from and written to files or other input and output sources.

For reading the stream:

Open the stream
Read information
Close the stream

For writing in stream:

Open the stream
Write information
Close the stream

There are two types of stream as follows:

- Byte stream
- Character stream

Byte Streams:

It supports 8-bit input and output operations. There are two classes of byte stream

InputStream :The **InputStream** class is used for reading the data such as a byte and array of bytes from an input source. An input source can be a **file**, a **string**, or **memory** that may contain the data. It is an abstract class that defines the programming interface for all input streams that are inherited from it. An input stream is automatically opened when you create it. You can explicitly close a stream with the **close( )** method, or let it be closed implicitly when the object is found as a garbage.

OutputStream: The OutputStream class is a sibling to InputStream that is used for writing byte and array of bytes to an output source. Similar to input sources, an output source can be anything such as a file, a string, or memory containing the data. Like an input stream, an output stream is automatically opened when you create it. You can explicitly close an output stream with the **close( )** method, or let it be closed implicitly when the object is garbage collected.

Character Streams:

It supports 16-bit Unicode character input and output. There are two classes of character stream as follows:

- Reader
- Writer

These classes allow internationalization of Java I/O and also allow text to be stored using international character encoding.

Reader:

- BufferedReader
  - o LineNumberReader
- CharAraayReader
- PipedReader
- StringReader
- FilterReader
  - o PushbackReader
- InputStreamReader
  - o FileReader

Writer:

- BufferedWriter
- CharAraayWriter
- FileWriter
- PipedWriter
- PrintWriter
- String Writer
- OutputStreamWriter
  - o FileWriter

How Files and Streams Work:

Java uses **streams** to handle I/O operations through which the data is flowed from one location to another. For example, an **InputStream** can flow the data from a disk file to the internal memory and an **OutputStream** can flow the data from the internal memory to a disk file. The disk-file may be a text file or a binary file. When we work with a text file, we use a **character** stream where one character is treated as per byte on disk. When we work with a binary file, we use a **binary** stream.

The working process of the I/O streams can be shown in the given diagram.

Classes:

Standard Streams: Standard Streams are a feature provided by many operating systems. By default, they read input from the keyboard and write output to the display. They also support I/O operations on files.

Standard Input: - Accessed through **System.in** which is used to read input from the keyboard.

Standard Output: - Accessed through **System.out** which is used to write output to be display.

Standard Error: - Accessed through **System.err** which is used to write error output to be display.

System.in is a byte stream that has no character stream features. To use Standard Input as a character stream, wrap System.in within the InputStreamReader as an argument.
InputStreamReader inp= new InputStreamReader (System.in);

Working with Reader classes: Java provides the standard I/O facilities for reading text from either the file or the keyboard on the command line. The **Reader** class is used for this purpose that is available in the **java.io** package. It acts as an abstract class for reading character streams. The only methods that a subclass must implement are **read(char[], int, int)** and **close()**. The Reader class is further categorized into the subclasses.
The following diagram shows a class-hierarchy of the **java.io.Reader** class.
However, most subclasses override some of the methods in order to provide higher efficiency, additional functionality, or both.

InputStreamReader: An InputStreamReader is a bridge from byte streams to character streams i.e. it reads bytes and decodes them into Unicode characters according to a particular platform. Thus, this class reads characters from a byte input stream. When you create an InputStreamReader, you specify an InputStream from which, the InputStreamReader reads the bytes.

The syntax of InputStreamReader is written as:
InputStreamReader<variable_name>= new InputStreamReader (System.in)

BufferedReader:
The BufferedReader class is the subclass of the Reader class. It reads character-input stream data from a memory area known as a buffer maintains state. The buffer size may be specified, or the default size may be used that is large enough for text reading purposes. BufferedReader converts an unbuffered stream into a buffered stream using the wrapping expression, where the unbuffered stream object is passed to the constructor for a buffered stream class.
For example the constructors of the BufferedReader class shown as:
BufferedReader (Reader in): Creates a buffering character-input stream that uses a default-sized input buffer.
BufferedReader (Reader in, int sz): Creates a buffering character-input stream that uses an input buffer of the specified size.
BufferedReader class provides some standard methods to perform specific reading operations shown in the table. All methods throw an IOException, if an I/O error occurs.

Method	Return Type	Description
read( )	int	Reads a single character
read(char[] cbuf, int off, int len)	int	Read characters into a portion of an array.
readLine( )	String	Read a line of text. A line is considered to be terminated by (**'\n'**).
close( )	void	Closes the opened stream.

This program illustrates use of standard input stream to read the user input.

```
import java.io.*;
public class ReadStandardIO {
        public static void main(String[] args) throws IOException {
                InputStreamReader inp = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(inp); System.out.println("Enter
                text : ");
                String str = in.readLine();
                System.out.println("You entered String : "); System.out.println(str);
        }
}
```

Output of the Program:

```
C:\>javac ReadStandardIO.java
C:\>java ReadStandardIO
Enter text: this is an Input Stream
        You entered String: this is an Input Stream
```

3 Introduction to the concept of package, Java API packages, using the System package

Packages: Introduction

We have repeatedly stated that one of the main features of OOP is its ability to reuse the code already created. One way of achieving this is by extending the classes and implementing the interfaces. This is limited to reusing the classes within a program. What if we need to use classes from other programs without physically copying them into the program under development? This can be accomplished in Java by using what is known as packages, a concept similar to "class libraries" in other languages. Another way of achieving the reusability in Java, therefore, is to use packages.

Packages are Java's way of grouping a variety of classes and/or interfaces together. The grouping is usually done according to functionality. In fact, packages act as "containers" for classes. By organizing our classes into packages we achieve the following benefits:

1. The classes contained in the packages of other programs can be easily reused.
2. In packages, classes can be unique compared with classes in other packages. That is, two classes in two different packages can have the same name. They may be referred by their fully qualified name, comprising the package name and the class name.
3. Packages provide a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal use only.
4. Packages also provide a way for separating "design" from "coding". First we can design classes and decide their relationships, and then we can implement the Java code needed for the methods. It is possible to change the implementation of any method without affecting the rest of the design.

For most applications, we will need to use to different sets of classes, one for the internal representation of our program's data, and the other for external presentation purposes. We may have to build our own classes for handling our data and use existing class libraries for designing user interfaces. Java packages are therefore classified into two types. The first category is known as Java API packages and the second is known as user defined packages.

Java API packages

Java API provides a large number of classes grouped into different packages according to functionality. Most of the time we use the package available with the Java API. Below fig shows the functional breakdown of packages that are frequently used in the programs:

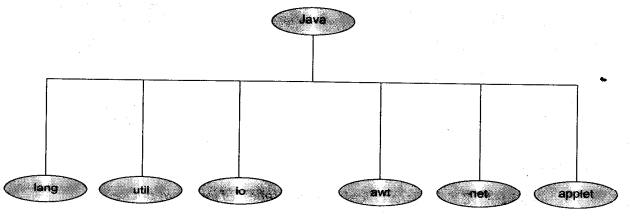


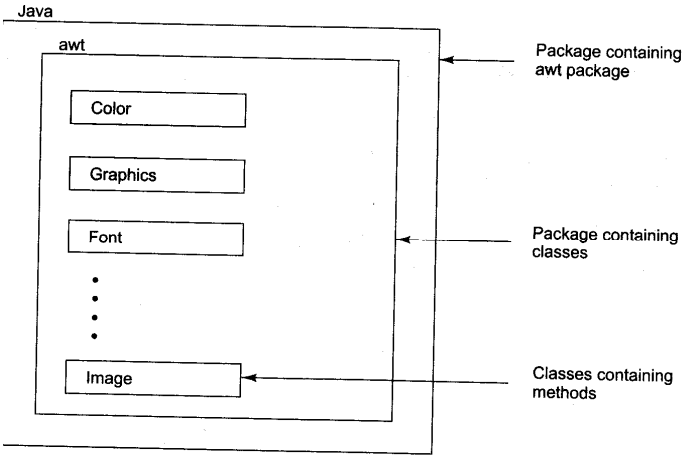
Fig. 11.1 *Frequently used API packages*

Table 11.1 Java System Packages and Their Classes

Package name	*Contents*
java.lang	Language support classes. These are classes that Java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, date, etc.
java.io	Input/output support classes. They provide facilities for the input and output of data.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.
java.net	Classes fof networking. They include classes for communicating with local computers as well as with internet servers.
java.applet	Classes for creating and implementing applets.

Using System Packages

The packages are organised in a hierarchical structure as illustrated in below Fig. This shows that package named java contains the package awt, which in turn contains various classes required implementing graphical user interface.



There are two ways of accessing the classes stored in a package. The first approach is to use the fully qualified class name of the class that we want to use. This is done by the package name containing the class and then appending the class name to it using the dot operator. For example, if we want to refer to the class Color in the awt package, then we may do so as follows:
java.awt.Colour

Awt is a package within the package java and the hierarchy is represented by separating the levels with dots. This approach is perhaps the test and easiest one if we need to access the class only once or when we need not have to access any other classes of the package.
But, in many situations, we might want to use a class in a number of places in the program or we may like to use many of the classes contained in a package. We may achieve this as follows:
import packagename.classname;
or
import packagename. *;

These are known as import statements and must appear at the top of the file, before any class declarations, import is a keyword.

4 Using java.lang (String, Math)

String

String manipulation is the most common part of many Java programs. Strings represent a sequence of characters. The easiest way to represent a sequence of characters in Java is by using a character array.
Example:

```
char  charArray[] = new char[4];
charArray[0] = 'J';
charArray[1] = 'a';
charArray[2] = 'v';
charArray[3] = 'a';
```

Although character arrays have the advantage of being able to query their length, they themselves are not good enough to support the range of operations we may like to perform on strings. For example, copying once character array into another might require a lot of book keeping effort. Fortunately, Java is equipped to handle these situations more efficiently.

In Java, string are class objects and implemented using two classes, namely, String and StringBuffer. A Java string is an instantiated object of String class. Java strings, as compared to C strings, are more reliable and predictable. This is basically due to C's lack of bonus-checking. A Java string is not a character array and is not NULL terminated. Strings may be declared and created as follows:

String stringName;
stringName = new String("string");
Example: String firstName;
firstName = new String("Anil");

These two statements may be combined as follows:
String firstName = new String("Anil");

Like arrays, it is possible to get the length of string using the length method of the String class.
int m = firstName.length();

Note that use of parentheses here. Java strings can be concatenated using the + operator.
Examples: String fullName= name1 + name2;
String city1 = "New" + "Delhi";
Where name1 and name2 are java strings containing string constants. Another example is System.out.println(firstName + "Kumar");

String Arrays
We can also create and use arrays that contain strings. The statement
String itemArray[]=new String[3];
Will create an itemArray of size 3 to hold three string constants. We can assign the strings to the itemArray element by element using three different statements or more efficiently using a for loop.

String Methods
The String class defines a number of methods that allow us to accomplish a variety of string manipulation task. Below table list some of the most commonly used string methods, and their tasks.

Method Call	Task Performed
s2 = s1.toLowerCase();	Converts the string s1 to all lowercase
s2 = s1.toUpperCase();	Converts the string s1 to all uppercase
s2 = s1.replace('x','y');	Replace all appearances of x with y
s2 = s1.trim()	Remove white spaces at the beginning and end of the string s1
s1.equals(s2)	Returns 'true' if s1 is equal to s2
s1.equalsIgnoreCase(s2)	Returns 'true' if s1 is equal to s2, ignoring the case of characters
s1.length()	Gives the length of s1
s1.charAt(n)	Gives nth character of s1
s1.compareTo(s2)	Returns negative if s1<s2, positive if s1>s2, o if s1 is equal to s2
s1.concat(s2)	Concatenates s1 and s2
s1.substring(n)	Gives substring starting from nth character
s1.substring(n,m)	Gives substring starting from nth character upto mth character(not including mth)
String.valueOf(p)	Creates a string object of the parameter p
p.toString()	Creates a string representation of the object p
s1.indexOf('x')	Gives the position of the first occurrence of 'x' in the string s1
s1.indexOf('x',n)	Gives the position of 'x' that occurs after nth position in the string s1

StringBuffer Class

StringBuffer is a peer class of String. While String creates string of fixed length, StringBuffer creates string of flexible length that can be modified in terms of both length and content. We can insert characters and substring in the middle of a string, or append another string to the end. Below table lists some of the methods that are frequently used in string manipulations.

Method Call	Task Performed
s1.setCharAt(n,'x')	Modifies the nth character to x
s1.append(s2)	Appends the string s2 at the end of s1
s1.insert(n,s2)	inserts the string s2 at the position n of the string s1
s1.setLength(n)	sets the length of the string s1 to n. if n<s1.length() s1 is truncated If n>s1.length() zeros are added to s1

Mathematical Functions

Mathematical functions such as cos, sqrt, log, etc. are frequently used in analysis of real-life problems. Java supports these basic math functions through Math class defined in the java.lang package. Below table lists the math functions defined in Math class. These functions should be used as follows:

Math.function_name()

Example: double y= Math.sqrt(x);

Function	Action
sin(x)	returns the sine of the angle x in radians
cos(x)	returns the cosine of the angle x in radians
tan(x)	returns the tangent of the angle x in radians
asin(y)	returns the angle whose sine is y
acos(y)	returns the angle whose cosine is y
atan(y)	returns the angle whose tangent is y
atan2(x,y)	returns the angle whose tangent is x/y
pow(x,y)	returns x raised to y (xy)
exp(x)	returns e raised to x (ex)
log(x)	returns the natural logarithm of x
sqrt(x)	returns the square root of x
ceil(x)	returns the smallest whole number greater than or equal to x(Rounding up)
floor(x)	returns the largest whole number less than or equal to x(Rounding down)
rint(x)	returns the truncated value of x
round(x)	returns the integer closest to the argument
abs(a)	returns the absolute value of a
max(a,b)	returns the maximum of a and b
min(a,b)	returns the minimum of a and b

Note that x and y are double type parameters, a and b may be ints, longs, floats and doubles.

1 Applet Introduction, Applet Architecture and Applet Skeleton – set background, foreground color

1.1 Applet Introduction

- All applets are subclasses of Applet. Thus, all applets must import java.applet. Applets must also import java.awt. Recall that AWT stands for the 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. The figures shown in this chapter were created with the standard applet viewer, called appletviewer, provided by the SDK. But you can use any applet viewer or browser you like. Execution of an applet does not begin at main( ). Actually, few applets even have main( ) methods. Instead, execution of an applet is started and controlled with an entirely different mechanism, which will be explained shortly. Output to your applet's window is not performed by System.out.println( ). Rather, it is handled with various AWT methods, such as drawString( ), which outputs a string to a specified X,Y location. Input is also handled differently than in an application.
- Once an applet has been compiled, it is included in an HTML file using the 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, simply 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. Here is an example of such a comment:

  ```
  /*
  <applet code="MyApplet" width=200 height=60> </applet>
  */
  ```

- This comment contains an APPLET tag that will run an applet called MyApplet in a window that is 200 pixels wide and 60 pixels high. Since the inclusion of an APPLET command makes testing applets easier, all of the applets shown in this book will contain the appropriate APPLET tag embedded in a comment.

1.2 Applet Architecture

- An applet is a window-based program. As such, its architecture is different from the so-called normal, console-based programs shown in the first part of this book. If you are familiar with Windows programming, you will be right at home writing applets. If not, then there are a few key concepts you must understand.
- First, applets are event driven. Although we won't examine event handling until the following chapter, it is important to understand in a general way how the event-driven architecture impacts the design of an applet. An applet resembles a set of interrupt service routines. Here is how the process works. An applet waits until an event occurs.

- The AWT notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return control to the AWT. This is a crucial point. For the most part, your applet should not enter a “mode” of operation in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the AWT run-time system. In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), you must start an additional thread of execution. (You will see an example later in this chapter.)
- Second, the user initiates interaction with an applet—not the other way around. As you know, in a nonwindowed program, when the program needs input, it will prompt the user and then call some input method, such as readLine( ). This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks a mouse inside the applet’s window, a mouse-clicked event is generated. If the user presses a key while the applet’s window has input focus, a keypress event is generated. As you will see in later chapters, applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.
- While the architecture of an applet is not as easy to understand as that of a console-based program, Java’s AWT makes it as simple as possible. If you have written programs for Windows, you know how intimidating that environment can be. Fortunately, Java’s AWT provides a much cleaner approach that is more quickly mastered.

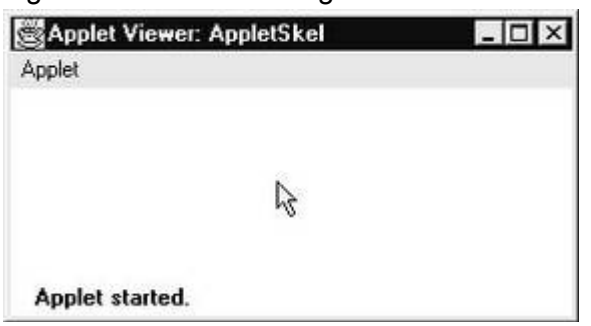
1.3 An Applet Skeleton (Life Cycle)

- All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods—init( ), start( ), stop( ), and destroy() are defined by Applet. Another, paint( ), is defined by the AWT Component class. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them. These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet {
// Called first.
```

```
        public void init() {
                // initialization
        }
    /* Called second, after init(). Also called whenever the applet is restarted. */
        public void start() {
                // start or resume execution
        }
    // Called when the applet is stopped.
        public void stop() {
                // suspends execution
        }
    /* Called when applet is terminated. This is the last method executed. */
        public void destroy() {
                // perform shutdown activities
        }
    // Called when an applet's window must be restored.
        public void paint(Graphics g) {
                // redisplay contents of window
        }
}
```

- Although this skeleton does not do anything, it can be compiled and run. When run, it generates the following window when viewed with an applet viewer:



1.4 Applet Initialization and Termination:

- It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the AWT calls the following methods, in this sequence:
    1. init( )
    2. start( )
    3. paint( )
- When an applet is terminated, the following sequence of method calls takes place:
    1. stop( )
    2. destroy( )
- Let's look more closely at these methods.

init( )

- The init( ) method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start( )

- The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. Whereas init( ) is called once—the first time an applet is loaded—start( ) is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start( ).

paint( )

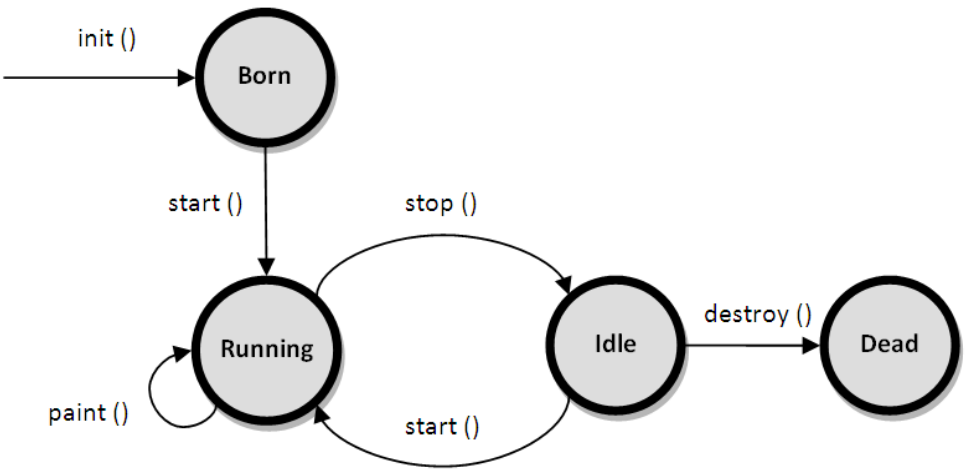
- The paint( ) method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called. The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop( )

- The stop( ) method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When stop( ) is called, the applet is probably running. You should use stop( ) to suspend threads that don't need to run when the applet is not visible. You can restart them when start( ) is called if the user returns to the page.

destroy( )

- The destroy( ) method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The stop( ) method is always called before destroy( ).



1.5 Simple Applet Display Methods

- As we've mentioned, applets are displayed in a window and they use the AWT to perform input and output. Although we will examine the methods, procedures, and techniques necessary to fully handle the AWT windowed environment in subsequent chapters, a few are described here, because we will use them to write sample applets.
- As we described in Chapter 12, to output a string to an applet, use drawString( ), which is a member of the Graphics class. Typically, it is called from within either update( ) or paint( ). It has the following general form:

```
void drawString(String message, int x, int y)
```

- Here, message is the string to be output beginning at x,y. In a Java window, the upper-left corner is location 0,0. The drawString( ) method will not recognize newline characters. If you want to start a line of text on another line, you must do so manually, specifying the precise X,Y location where you want the line to begin. (As you will see in later chapters, there are techniques that make this process easy.)
- To set the background color of an applet's window, use setBackground( ). To set the foreground color (the color in which text is shown, for example), use setForeground( ). These methods are defined by Component, and they have the following general forms:

```
void setBackground(Color newColor)
void setForeground(Color newColor)
```

- Here, newColor specifies the new color. The class Color defines the constants shown here that can be used to specify colors:

Color.black	Color.magenta
Color.blue	Color.orange
Color.cyan	Color.pink
Color.darkGray	Color.red
Color.gray	Color.white
Color.green	Color.yellow
Color.lightGray	

- For example, this sets the background color to green and the text color to red:

```
setBackground(Color.green);
setForeground(Color.red);
```

- A good place to set the foreground and background colors is in the init( ) method. Of course, you can change these colors as often as necessary during the execution of your applet. The default foreground color is black. The default background color is light gray.
- You can obtain the current settings for the background and foreground colors by calling getBackground( ) and getForeground( ), respectively. They are also defined by Component and are shown here:

```
Color getBackground( )
Color getForeground( )
```

- Here is a very simple applet that sets the background color to cyan, the foreground color to red, and displays a message that illustrates the order in which the init( ), start( ), and paint( ) methods are called when an applet starts up:

```
/* A simple applet that sets the foreground and
background colors and outputs a string. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/

public class Sample extends Applet{
String msg;
// set the foreground and background colors.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
msg = "Inside init( ) --";
}
// Initialize the string to be displayed.
public void start() {
msg += " Inside start( ) --";
}
// Display msg in applet window.
public void paint(Graphics g) {
msg += " Inside paint( ).";
g.drawString(msg, 10, 30);
}
}
```

2 java.awt package (Button, CheckBox, CheckBoxGroup, Choice, Color, Label, List, TextArea, TextField)

2.1 Control Fundamentals

- The AWT supports the following types of controls:
  - Labels
  - Push buttons
  - Check boxes
  - Choice lists
  - Lists
  - Text editing
- These controls are subclasses of Component.

2.1.1 Adding and Removing Controls

- To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling add( ), which is defined by Container. The add( ) method has several forms. The following form is the one that is used for the first part of this chapter:

  Component add(Component compObj)
- Here, compObj is an instance of the control that you want to add. A reference to compObj is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.
- Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call remove( ). This method is also defined by Container. It has this general form:

  void remove(Component obj)
- Here, obj is a reference to the control you want to remove. You can remove all controls by calling removeAll( ).

2.1.2 Responding to Controls

- Except for labels, which are passive controls, all controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor. As explained in Chapter 20, once a listener has been installed, events are automatically sent to it. In the sections that follow, the appropriate interface for each control is specified.

2.2 Using Label

- The easiest control to use is a label. A label is an object of type Label, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. Label defines the following constructors:
  - Label( )
  - Label(String str)
  - Label(String str, int how)
- The first version creates a blank label. The second version creates a label that contains the string specified by str. This string is left-justified. The third version creates a label that contains the string specified by str using the alignment specified by how. The value of how must be one of these three constants: Label.LEFT, Label.RIGHT, or Label.CENTER.
- You can set or change the text in a label by using the setText( ) method. You can obtain the current label by calling getText( ). These methods are shown here:
  - void setText(String str)
  - String getText( )
- For setText( ), str specifies the new label. For getText( ), the current label is returned. You can set the alignment of the string within the label by calling setAlignment( ). To obtain the current alignment, call getAlignment( ). The methods are as follows:

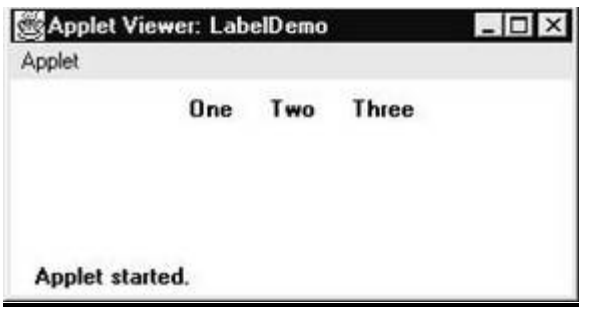
- void setAlignment(int how)
- int getAlignment( )

- Here, how must be one of the alignment constants shown earlier. The following example creates three labels and adds them to an applet:

```
// Demonstrate Labels
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/

public class LabelDemo extends Applet {
public void init() {
Label l1 = new Label();
l1.setText(“one”);
Label l2 = new Label("Two");
Label l3 = new Label("Three");
String str;
str=l3.getText();
// add labels to applet window
add(l1);
add(l2);
add(l3);
}
}
```

- Following is the window created by the LabelDemo applet. Notice that the labels are organized in the window by the default layout manager. Later, you will see how to control more precisely the placement of the labels.



2.3 Using Button

- The most widely used control is the push button. A push button is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type Button. Button defines these two constructors:
  - Button( )
  - Button(String str)

- The first version creates an empty button. The second creates a button that contains str as a label.
- After a button has been created, you can set its label by calling setLabel( ). You can retrieve its label by calling getLabel( ). These methods are as follows:
  - void setLabel(String str)
  - String getLabel( )
- Here, str becomes the new label for the button.

2.3.1 Handling Buttons

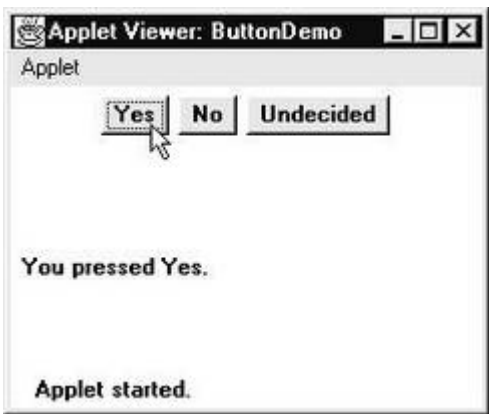
- Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the ActionListener interface. That interface defines the actionPerformed( ) method, which is called when an event occurs.
- An ActionEvent object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the string that is the label of the button. Usually, either value may be used to identify the button, as you will see.
- Here is an example that creates three buttons labeled “Yes,” “No,” and “Undecided.” Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the label of the button is used to determine which button has been pressed. The label is obtained by calling the getActionCommand() method on the ActionEvent object passed to actionPerformed().

```
// Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener {
String msg = "";
Button yes, no, maybe;
public void init() {
yes = new Button("Yes");
no = new Button("No");
maybe = new Button("Undecided");
add(yes);
add(no);
add(maybe);
yes.addActionListener(this);
```

```
no.addActionListener(this);
maybe.addActionListener(this);
}
public void actionPerformed(ActionEvent ae) {
String str = ae.getActionCommand();
if(str.equals("Yes")) {
msg = "You pressed Yes.";
}
else if(str.equals("No")) {
msg = "You pressed No.";
}
else {
msg = "You pressed Undecided.";
}
repaint();
}
public void paint(Graphics g) {
g.drawString(msg, 6, 100);
}
}
```

- Sample output from the ButtonDemo program is shown in below Figure.
- As mentioned, in addition to comparing button labels, you can also determine which button has been pressed, by comparing the object obtained from the getSource( ) method to the button objects that you added to the window. To do this, you must keep a list of the objects when they are added. The following applet shows this approach:

```
// Recognize Button objects.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonList" width=250 height=150>
</applet>
*/
```



```
public class ButtonList extends Applet implements ActionListener {
String msg = "";
Button bList[] = new Button[3];
public void init() {
Button yes = new Button("Yes");
Button no = new Button("No");
Button maybe = new Button("Undecided");
// store references to buttons as added
bList[0] = (Button) add(yes);
bList[1] = (Button) add(no);
bList[2] = (Button) add(maybe);
// register to receive action events
for(int i = 0; i < 3; i++) {
bList[i].addActionListener(this);
}
}
public void actionPerformed(ActionEvent ae) {
for(int i = 0; i < 3; i++) {
if(ae.getSource() == bList[i]) {
msg = "You pressed " + bList[i].getLabel();
}
}
repaint();
}
public void paint(Graphics g) {
g.drawString(msg, 6, 100);
}
}
```

- In this version, the program stores each button reference in an array when the buttons are added to the applet window. (Recall that the add( ) method returns a reference to the button when it is added.) Inside actionPerformed( ), this array is then used to determine which button has been pressed.
- For simple applets, it is usually easier to recognize buttons by their labels. However, in situations in which you will be changing the label inside a button during the execution of your program, or using buttons that have the same label, it may be easier to determine which button has been pushed by using its object reference.

2.4 Checkboxes

- A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the Checkbox class.
- Checkbox supports these constructors:
  - Checkbox( )

- Checkbox(String str)
- Checkbox(String str, boolean on)
- Checkbox(String str, boolean on, CheckboxGroup cbGroup)
- Checkbox(String str, CheckboxGroup cbGroup, boolean on)

- The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by str. The state of the check box is unchecked. The third form allows you to set the initial state of the check box. If on is true, the check box is initially checked; otherwise, it is cleared. The fourth and fifth forms create a check box whose label is specified by str and whose group is specified by cbGroup. If this check box is not part of a group, then cbGroup must be null. (Check box groups are described in the next section.) The value of on determines the initial state of the check box. To retrieve the current state of a check box, call getState( ). To set its state, call setState( ). You can obtain the current label associated with a check box by calling getLabel( ). To set the label, call setLabel( ). These methods are as follows:
  - boolean getState( )
  - void setState(boolean on)
  - String getLabel( )
  - void setLabel(String str)
- Here, if on is true, the box is checked. If it is false, the box is cleared. The string passed in str becomes the new label associated with the invoking check box. Handling Check Boxes Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).
- The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

```
// Demonstrate check boxes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=250 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener {
String msg = "";
Checkbox Win98, winNT, solaris, mac;
public void init() {
Win98 = new Checkbox("Windows 98/XP", null, true);
```

```
winNT = new Checkbox("Windows NT/2000");
solaris = new Checkbox("Solaris");
mac = new Checkbox("MacOS");
add(Win98);
add(winNT);
add(solaris);
add(mac);
Win98.addItemListener(this);
winNT.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie) {
repaint();
}
// Display current state of the check boxes.
public void paint(Graphics g) {
msg = "Current state: ";
g.drawString(msg, 6, 80);
msg = " Windows 98/XP: " + Win98.getState();
g.drawString(msg, 6, 100);
msg = " Windows NT/2000: " + winNT.getState();
g.drawString(msg, 6, 120);
msg = " Solaris: " + solaris.getState();
g.drawString(msg, 6, 140);
msg = " MacOS: " + mac.getState();
g.drawString(msg, 6, 160);
}
}
```

Sample output is shown in below.

2.5 CheckboxeGroup

- It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called radio buttons, because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type CheckboxGroup. Only the default constructor is defined, which creates an empty group.
- You can determine which check box in a group is currently selected by calling getSelectedCheckbox( ). You can set a check box by calling setSelectedCheckbox ( ). These methods are as follows:
  - Checkbox getSelectedCheckbox( )
  - void setSelectedCheckbox(Checkbox which)
- Here, which is the check box that you want to be selected. The previously selected check box will be turned off.Here is a program that uses check boxes that are part of a group:

```
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <applet code="CBGroup" width=250 height=200>
</applet> */
public class CBGroup extends Applet implements ItemListener {
String msg = "";
Checkbox Win98, winNT, solaris, mac;
CheckboxGroup cbg;
public void init() {
cbg = new CheckboxGroup();
Win98 = new Checkbox("Windows 98/XP", cbg, true);
winNT = new Checkbox("Windows NT/2000", cbg, false);
solaris = new Checkbox("Solaris", cbg, false);
mac = new Checkbox("MacOS", cbg, false);
add(Win98);
add(winNT);
add(solaris);
add(mac);
Win98.addItemListener(this);
winNT.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);          }
public void itemStateChanged(ItemEvent ie) {   repaint();       }
// Display current state of the check boxes.
public void paint(Graphics g) {
msg = "Current selection: ";
msg += cbg.getSelectedCheckbox().getLabel();
```

```
g.drawString(msg, 6, 100);
}
}
```

Output generated by the CBGroup applet is shown in below. Notice that the check boxes are now circular in shape.

2.6 Choice lists

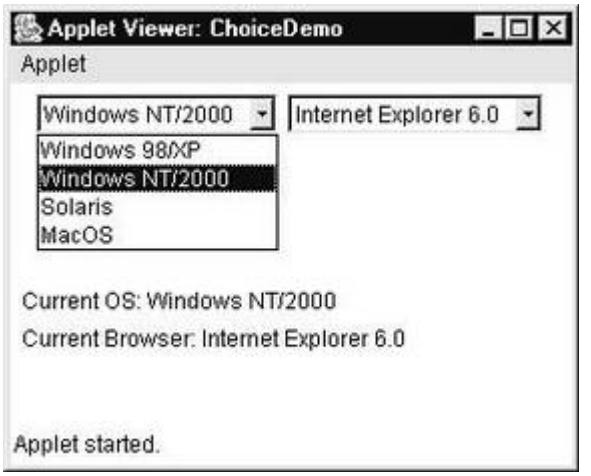
- The Choice class is used to create a pop-up list of items from which the user may choose. Thus, a Choice control is a form of menu. When inactive, a Choice component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object. Choice only defines the default constructor, which creates an empty list. To add a selection to the list, call add( ). It has this general form:
  - void add(String name)
- Here, name is the name of the item being added. Items are added to the list in the order in which calls to add( ) occur.
- To determine which item is currently selected, you may call either getSelectedItem () or getSelectedIndex( ). These methods are shown here:
  - String getSelectedItem( )
  - int getSelectedIndex( )
- The getSelectedItem( ) method returns a string containing the name of the item. getSelectedIndex( ) returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.
- To obtain the number of items in the list, call getItemCount( ). You can set the currently selected item using the select( ) method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:
  - int getItemCount( )
  - void select(int index)
  - void select(String name)
- Given an index, you can obtain the name associated with the item at that index by calling getItem( ), which has this general form:
  - String getItem(int index)

- Here, index specifies the index of the desired item.
- Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method.
- Here is an example that creates two Choice menus. One selects the operating system. The other selects the browser.

```
// Demonstrate Choice lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/
public class ChoiceDemo extends Applet implements ItemListener {
Choice os, browser;
String msg = "";
public void init() {
os = new Choice();
browser = new Choice();
// add items to os list
os.add("Windows 98/XP");
os.add("Windows NT/2000");
os.add("Solaris");
os.add("MacOS");
// add items to browser list
browser.add("Netscape 3.x");
browser.add("Netscape 4.x");
browser.add("Netscape 5.x");
browser.add("Netscape 6.x");
browser.add("Internet Explorer 4.0");
browser.add("Internet Explorer 5.0");
browser.add("Internet Explorer 6.0");
browser.add("Lynx 2.4");
browser.select("Netscape 4.x");
// add choice lists to window
add(os);
add(browser);
// register to receive item events
os.addItemListener(this);
browser.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie) {
```

```
repaint();
}
// Display current selections.
public void paint(Graphics g) {
msg = "Current OS: ";
msg += os.getSelectedItem();
g.drawString(msg, 6, 120);
msg = "Current Browser: ";
msg += browser.getSelectedItem();
g.drawString(msg, 6, 140);
}
}
```

Sample output is shown in below.



2.7 Lists

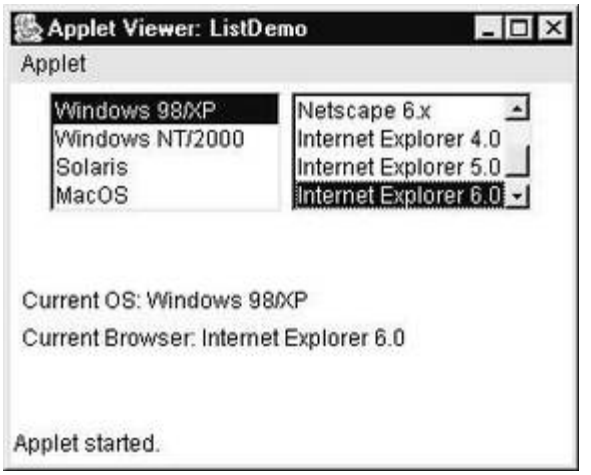
- The List class provides a compact, multiple-choice, scrolling selection list. Unlike the Choice object, which shows only the single selected item in the menu, a List object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. List provides these constructors:
  - List( )
  - List(int numRows)
  - List(int numRows, boolean multipleSelect)
- The first version creates a List control that allows only one item to be selected at any one time. In the second form, the value of numRows specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if multipleSelect is true, then the user may select two or more items at a time. If it is false, then only one item may be selected. To add a selection to the list, call add( ). It has the following two forms:
  - void add(String name)
  - void add(String name, int index)
- Here, name is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by index. Indexing begins at zero. You can specify –1 to add the item to the end of the list.

- For lists that allow only single selection, you can determine which item is currently selected by calling either getSelectedItem( ) or getSelectedIndex( ). These methods are shown here:
  - String getSelectedItem( )
  - int getSelectedIndex( )
- The getSelectedItem( ) method returns a string containing the name of the item. If more than one item is selected or if no selection has yet been made, null is returned. getSelectedIndex( ) returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned. It allow multiple selection, you must use either getSelectedItems() or getSelectedIndexes(), shown here, to determine the current selections:
  - String[ ] getSelectedItems( )
  - int[ ] getSelectedIndexes( )
- getSelectedItems( ) returns an array containing the names of the currently selected items. getSelectedIndexes( ) returns an array containing the indexes of the currently selected items. To obtain the number of items in the list, call getItemCount( ). You can set the currently selected item by using the select( ) method with a zero-based integer index. These methods are shown here:
  - int getItemCount( )
  - void select(int index)
- Given an index, you can obtain the name associated with the item at that index by calling getItem( ), which has this general form:
  - String getItem(int index)
- Here, index specifies the index of the desired item.
- To process list events, you will need to implement the ActionListener interface. Each time a List item is double-clicked, an ActionEvent object is generated. Its getActionCommand( ) method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an ItemEvent object is generated. Its getStateChange( ) method can be used to determine whether a selection or deselection triggered this event. getItemSelectable( ) returns a reference to the object that triggered this event. Here is an example that converts the Choice controls in the preceding section into List components, one multiple choice and the other single choice:

```
// Demonstrate Lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>
</applet>
*/
public class ListDemo extends Applet implements ActionListener {
List os, browser;
String msg = "";
public void init() {
```

```
os = new List(4, true);
browser = new List(4, false);
// add items to os list
os.add("Windows 98/XP");
os.add("Windows NT/2000");
os.add("Solaris");
os.add("MacOS");
// add items to browser list
browser.add("Netscape 3.x");
browser.add("Netscape 4.x");
browser.add("Netscape 5.x");
browser.add("Netscape 6.x");
browser.add("Internet Explorer 4.0");
browser.add("Internet Explorer 5.0");
browser.add("Internet Explorer 6.0");
browser.add("Lynx 2.4");
browser.select(1);
// add lists to window
add(os);
add(browser);
// register to receive action events
os.addActionListener(this);
browser.addActionListener(this);
}
public void actionPerformed(ActionEvent ae) {
repaint();
}
// Display current selections.
public void paint(Graphics g) {
int idx[];
msg = "Current OS: ";
idx = os.getSelectedIndexes();
for(int i=0; i<idx.length; i++)
msg += os.getItem(idx[i]) + " ";
g.drawString(msg, 6, 120);
msg = "Current Browser: ";
msg += browser.getSelectedItem();
g.drawString(msg, 6, 140);
}
}
```

- Sample output generated by the ListDemo applet is shown in below. Notice that the browser list has a scroll bar, since all of the items won't fit in the number of rows specified when it is created.



2.7 Using a TextField

- The TextField class implements a single-line text-entry area, usually called an edit control. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. TextField is a subclass of TextComponent. TextField defines the following constructors:
  - TextField( )
  - TextField(int numChars)
  - TextField(String str)
  - TextField(String str, int numChars)
- The first version creates a default text field. The second form creates a text field that is numChars characters wide. The third form initializes the text field with the string contained in str. The fourth form initializes a text field and sets its width. TextField (and its superclass TextComponent) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call getText( ). To set the text, call setText( ). These methods are as follows:
  - String getText( )
  - void setText(String str)
- Here, str is the new string.
- The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using select( ). Your program can obtain the currently selected text by calling getSelectedText( ). These methods are shown here:
  - String getSelectedText( )
  - void select(int startIndex, int endIndex)
- getSelectedText( ) returns the selected text. The select( ) method selects the characters beginning at startIndex and ending at endIndex–1. You can control whether the contents of a text field may be modified by the user by calling setEditable( ). You can determine editability by calling isEditable( ). These methods are shown here:
  - boolean isEditable( )
  - void setEditable(boolean canEdit)

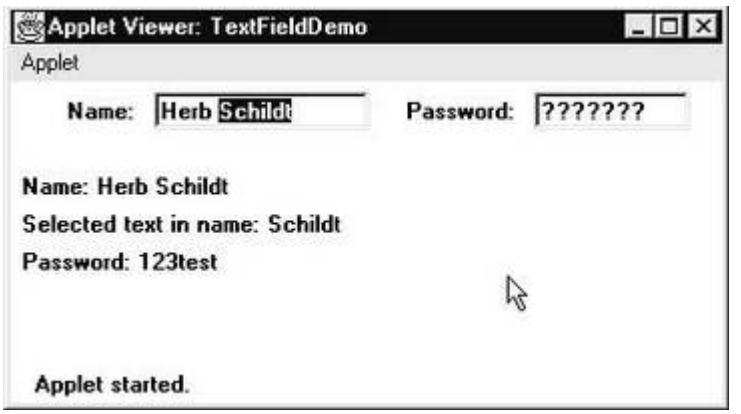
- isEditable( ) returns true if the text may be changed and false if not. In setEditable(), if canEdit is true, the text may be changed. If it is false, the text cannot be altered. There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling setEchoChar( ). This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the echoCharIsSet( ) method. You can retrieve the echo character by calling the getEchoChar( ) method. These methods are as follows:
  - void setEchoChar(char ch)
  - boolean echoCharIsSet( )
  - char getEchoChar( )
- Here, ch specifies the character to be echoed.

- Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field. However, you may want to respond when the user presses ENTER. When this occurs, an action event is generated.
- Here is an example that creates the classic user name and password screen:

```
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet
implements ActionListener {
TextField name, pass;
public void init() {
Label namep = new Label("Name: ", Label.RIGHT);
Label passp = new Label("Password: ", Label.RIGHT);
name = new TextField(12);
pass = new TextField(8);
pass.setEchoChar('?');
add(namep);
add(name);
add(passp);
add(pass);
// register to receive action events
name.addActionListener(this);
pass.addActionListener(this);
}
// User pressed Enter.
```

```
public void actionPerformed(ActionEvent ae) {
repaint();
}
public void paint(Graphics g) {
g.drawString("Name: " + name.getText(), 6, 60);
g.drawString("Selected text in name: "
+ name.getSelectedText(), 6, 80);
g.drawString("Password: " + pass.getText(), 6, 100);
}
}
```

- Sample output from the TextFieldDemo applet is shown in below.



2.8 Using a TextArea

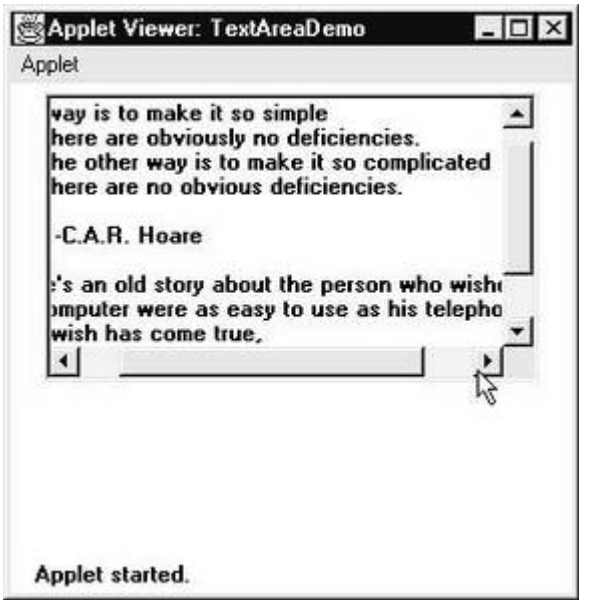
- Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called TextArea. Following are the constructors for TextArea:
  - TextArea( )
  - TextArea(int numLines, int numChars)
  - TextArea(String str)
  - TextArea(String str, int numLines, int numChars)
  - TextArea(String str, int numLines, int numChars, int sBars)
- Here, numLines specifies the height, in lines, of the text area, and numChars specifies its width, in characters. Initial text can be specified by str. In the fifth form you can specify the scroll bars that you want the control to have. sBars must be one of these values:
  - SCROLLBARS_BOTH SCROLLBARS_NONE
  - SCROLLBARS_HORIZONTAL_ONLY SCROLLBARS_VERTICAL_ONLY
- TextArea is a subclass of TextComponent. Therefore, it supports the getText( ), setText( ), getSelectedText( ), select( ), isEditable( ), and setEditable( ) methods described in the preceding section.
- TextArea adds the following methods:
  - void append(String str)

- void insert(String str, int index)
- void replaceRange(String str, int startIndex, int endIndex)

- The append( ) method appends the string specified by str to the end of the current text. insert( ) inserts the string passed in str at the specified index. To replace text, call replaceRange( ). It replaces the characters from startIndex to endIndex–1, with the replacement text passed in str. Text areas are almost self-contained controls. Your program incurs virtually no management overhead. Text areas only generate got-focus and lost-focus events. Normally, your program simply obtains the current text when it is needed.
- The following program creates a TextArea control:

```
// Demonstrate TextArea.
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/
public class TextAreaDemo extends Applet {
public void init() {
String val = "There are two ways of constructing " +
"a software design.\n" +
"One way is to make it so simple\n" +
"that there are obviously no deficiencies.\n" +
"And the other way is to make it so complicated\n" +
"that there are no obvious deficiencies.\n\n" +
" -C.A.R. Hoare\n\n" +
"There's an old story about the person who wished\n" +
"his computer were as easy to use as his telephone.\n" +
"That wish has come true,\n" +
"since I no longer know how to use my telephone.\n\n" +
" -Bjarne Stroustrup, AT&T, (inventor of C++)";
TextArea text = new TextArea(val, 10, 30);
add(text);
}
}
```

- Here is sample output from the TextAreaDemo applet:



3 Introduction to event handling

3.1The 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. Its concept is quite simple: a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to “delegate” the processing of an event to a separate piece of code.
- In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.
- The following sections define events and describe the roles of sources and listeners.

3.1.1 Events

- In the delegation model, 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 cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples.
- 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, software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

3.1.2 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 the name of the event and el is a reference to the event listener. For example, to remove a keyboard listener, you would call removeKeyListener( ).
- The methods that add or remove listeners are provided by the source that generates events. For example, the Component class provides methods to add and remove keyboard and mouse event listeners.

3.1.3 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 methods 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.
- Many other listener interfaces are discussed later in this and other chapters.

3.2 Event Classes

- The classes that represent events are at the core of Java's event handling mechanism. Thus, we begin our study of event handling with a tour of the event classes. As you will see, they provide a consistent, easy-to-use means of encapsulating events. 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( ) and toString( ). The getSource( ) method returns the source of the event. Its general form is shown here:
  - Object getSource( )
- As expected, toString( ) returns the string equivalent of the event. The class AWTEvent, defined within the java.awt package, is a subclass of EventObject. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its getID( ) method can be used to determine the type of the event. The signature of this method is shown here:
  - int getID( )
- Additional details about AWTEvent are provided at the end of Chapter 22. At this point, it is important to know only that all of the other classes discussed in this section are subclasses of AWTEvent. To summarize:
  ■ 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.

3.2.1 The ActionEvent Class

- 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 constructors:
  - 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 (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The when parameter specifies when the event occurred. The third constructor was added by Java 2, version 1.4.
- You can obtain the command name for the invoking ActionEvent object by using the getActionCommand( ) method, shown here:
  - String getActionCommand( )
- For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.
- The getModifiers( ) method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:
  - int getModifiers( )
- Java 2, version 1.4 added the method getWhen( ) that returns the time at which the event took place. This is called the event's timestamp. The getWhen( ) method is shown here.
  - long getWhen( )
- Timestamps were added by ActionEvent to help support the improved input focus subsystem implemented by Java 2, version 1.4.

3.2.2 The ItemEvent Class

- An ItemEvent is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. (Check boxes and list boxes are described later in this book.) 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.
- In addition, ItemEvent defines one integer constant, ITEM_STATE_CHANGED,
- that signifies a change of state. ItemEvent has this constructor:
  - ItemEvent(ItemSelectable src, int type, Object entry, int state)
- Here, src is a reference 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.
- The getItem( ) method can be used to obtain a reference to the item that generated an event. Its signature is shown here:

- Object getItem( )

- The getItemSelectable( ) method can be used to obtain a reference to the ItemSelectable object that generated an event. Its general form is shown here:
  - ItemSelectable getItemSelectable( )
- Lists and choices are examples of user interface elements that implement the ItemSelectable interface. The getStateChange( ) method returns the state change (i.e., SELECTED or DESELECTED) for the event. It is shown here:
  - int getStateChange( )

3.2.3 The KeyEvent Class

- A KeyEvent is generated when keyboard input occurs. There are three 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 any key is pressed or released. The last event occurs only when a character is generated. Remember, not all key presses result in characters. For example, pressing the SHIFT key does not generate a character.
- There are many other integer constants that are defined by KeyEvent. For example, VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ENTER	VK_ESCAPE	VK_CANCEL	VK_UP
VK_DOWN	VK_LEFT	VK_RIGHT	VK_PAGE_DOWN
VK_PAGE_UP	VK_SHIFT	VK_ALT	VK_CONTROL

- The VK constants specify virtual key codes and are independent of any modifiers, such as control, shift, or alt. KeyEvent is a subclass of InputEvent. Here are two of its constructors:
  - KeyEvent(Component src, int type, long when, int modifiers, int code)
  - KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)
- Here, src is a reference to the component that generated the event. The type of the event is specified by type. The system time at which the key was pressed is passed in when. The modifiers argument indicates which modifiers were pressed when this key event occurred.
- The virtual key code, such as VK_UP, VK_A, and so forth, is passed in code. The character equivalent (if one exists) is passed in ch. If no valid character exists, then ch contains CHAR_UNDEFINED. For KEY_TYPED events, code will contain VK_UNDEFINED.
- The KeyEvent class defines several methods, but the most commonly used ones are getKeyChar( ), which returns the character that was entered, and getKeyCode( ), which returns the key code. Their general forms are shown here:
  - char getKeyChar( )
  - int getKeyCode( )
- If no valid character is available, then getKeyChar( ) returns CHAR_UNDEFINED. When a KEY_TYPED event occurs, getKeyCode( ) returns VK_UNDEFINED.

3.2.4 The MouseEvent Class

- There are eight types of mouse events. The MouseEvent class defines the following integer constants that can be used to identify them:
  - MOUSE_CLICKED The user clicked the mouse.
  - MOUSE_DRAGGED The user dragged the mouse.
  - MOUSE_ENTERED The mouse entered a component.
  - MOUSE_EXITED The mouse exited from a component.
  - MOUSE_MOVED The mouse moved.
  - MOUSE_PRESSED The mouse was pressed.
  - MOUSE_RELEASED The mouse was released.
  - MOUSE_WHEEL The mouse wheel was moved (Java 2, v1.4).
- MouseEvent is a subclass of InputEvent. Here is one of its constructors.
  - 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 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 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 menu to appear on this platform. Java 2, version 1.4 adds a second constructor which also allows the button that caused the event to be specified.

- 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( )
- Alternatively, you can use the getPoint( ) method to obtain the coordinates of the mouse. It is shown here:
  - Point getPoint( )
- It returns a Point object that contains the X, Y coordinates in its integer members: x and y.
- The translatePoint( ) method changes the location of the event. Its form is shown here:
  - 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. Its signature is shown here:
  - int getClickCount( )
- The isPopupTrigger( ) method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:
  - boolean isPopupTrigger( )
- Java 2, version 1.4 added the getButton( ) method, shown here.
  - int getButton( )

- It returns a value that represents the button that caused the event. The return value will be one of these constants defined by MouseEvent.
  NOBUTTON BUTTON1 BUTTON2 BUTTON3
- The NOBUTTON value indicates that no button was pressed or released.

3.3 Sources of Events

- Below table lists some of the user interface components that can generate the events described in the previous section. In addition to these graphical user interface elements, other components, such as an applet, can generate events. For example, you receive key and mouse events from an applet. (You may also build your own components that generate events.) In this chapter we will be handling only mouse and keyboard events, but the following two chapters will be handling events from the sources shown in following Table.

Event Source	Description
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.

3.4 Event Listener Interface

- As explained, 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. Below table lists commonly used listener interfaces and provide a brief description of the methods that they define. The following sections examine the specific methods that are contained in each interface.

Interface	Description
ActionListener	Defines one method to receive action events.
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.

3.4.1 The 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)

3.4.2 The 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)

3.4.3 The KeyListener Interface

- This interface defines three methods. The keyPressed( ) and keyReleased( ) methods 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: key pressed and released.
- The general forms of these methods are shown here:
  - void keyPressed(KeyEvent ke)
  - void keyReleased(KeyEvent ke)
  - void keyTyped(KeyEvent ke)

3.4.4 The 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.
- The general forms of these methods are shown here:
  - void mouseClicked(MouseEvent me)
  - void mouseEntered(MouseEvent me)
  - void mouseExited(MouseEvent me)
  - void mousePressed(MouseEvent me)
  - void mouseReleased(MouseEvent me)

3.4.5 The 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 forms are shown here:
  - void mouseDragged(MouseEvent me)
  - void mouseMoved(MouseEvent me)

4 Introduction to JDBC

4.1 Java Database Connectivity

- JDBC (Java Database Connectivity) is a java API which enables the java programs to execute SQL statements. It is an application programming interface that defines how a java programmer can access the database in tabular format from Java code using a set of standard interfaces and classes written in the Java programming language.
- JDBC has been developed under the Java Community Process (JCP) that allows multiple implementations to exist and be used by the same application. JDBC provides methods for querying and updating the data in Relational Database Management system such as SQL, Oracle etc.
- The Java application programming interface provides a mechanism for dynamically loading the correct Java packages and drivers and registering them with the JDBC Driver Manager that is used as a connection factory for creating JDBC connections which supports creating and executing statements such as SQL INSERT, UPDATE and DELETE. Driver Manager is the backbone of the jdbc architecture.
- In short JDBC helps the programmers to write java applications that manage these three programming activities:
  1. It helps us to connect to a data source, like a database.
  2. It helps us in sending queries and updating statements to the database
  3. Retrieving and processing the results received from the database in terms of answering query.

4.2 JDBC has four Components:

1. **The JDBC API**: The JDBC API provides programmatic access to relational data from the Java programming language.

   The JDBC API is part of the Java platform, which includes the Java Standard Edition (Java SE) and the Java Enterprise Edition (Java EE). The JDBC 4.0 API is divided into two packages: java.sql and javax.sql. Both packages are included in the Java SE and Java EE platforms.
2. **JDBC Driver Manager**: The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver.
3. **JDBC Test Suite** — The JDBC driver test suite helps you to determine that JDBC drivers will run your program.
4. **JDBC-ODBC Bridge** — The Java Software bridge provides JDBC access via ODBC drivers.

4.3 Java Database Connectivity Steps

- Before you can create a java jdbc connection to the database, you must first import the java.sql package.
- import java.sql.*; The star (*) indicates that all of the classes in the package java.sql are to be imported.
- There are following six steps involved in building a JDBC application:

1) **Import the packages:** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using import java.sql.* will suffice.
2) **Register the JDBC driver:** Requires that you initialize a driver so you can open a communications channel with the database.
3) **Open a connection:** Requires using the DriverManager.getConnection()method to create a Connection object, which represents a physical connection with the database.
4) **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to the database.
5) **Extract data from result set:** Requires that you use the appropriate ResultSet.getXXX() method to retrieve the data from the result set.
6) **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

4.4 JDBC Driver

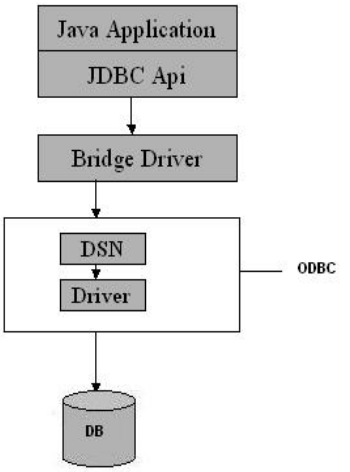
- Driver types are used to categorize the technology used to connect to the database. The driver class can be load by calling Class.forName() with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself. A client can connect to Database Server through JDBC Driver. Since most of the Database servers support ODBC driver therefore JDBC-ODBC Bridge driver is commonly used.
- The return type of the Class.forName (String ClassName) method is “Class”. Class is a class in java.lang package.

```
try
{
Class.forName(”sun.jdbc.odbc.JdbcOdbcDriver”); //Or any other driver
}
catch(Exception x)
{
        System.out.println( “Unable to load the driver class!” );
}
```

- **Types of JDBC drivers**
  1. JDBC-ODBC bridge plus ODBC driver, also called Type 1.
  2. Native-API, partly Java driver, also called Type 2.
  3. JDBC-Net, pure Java driver, also called Type 3.
  4. Native-protocol, pure Java driver, also called Type 4.

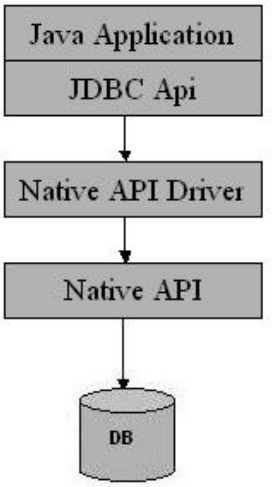
4.4.1 TYPE – 1: JDBC-ODBC Bridge

- This driver is implemented in the sun.jdbc.odbc.JdbcOdbcDriver class and comes with the Java 2 SDK, Standard Edition. Type 1 is the simplest of all but platform specific i.e only to Microsoft platform.
- This driver converts JDBC method calls into ODBC function calls. Type 1 drivers are written in Native code.



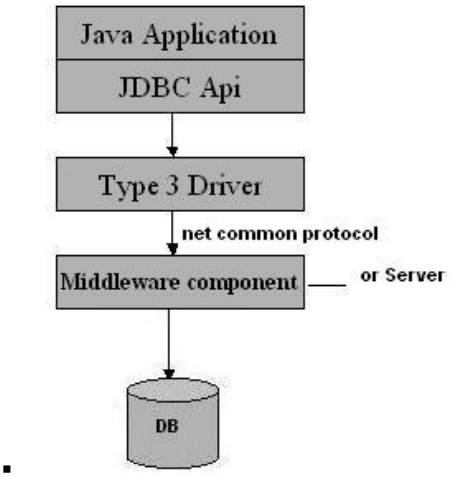
4.4.2 TYPE – 2 JDBC-Native API or Native Driver

- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database.
- This type of driver converts JDBC calls into calls to the client API for that database.
- The native API should change if the Database is changed because it is specific to a database.
- The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.



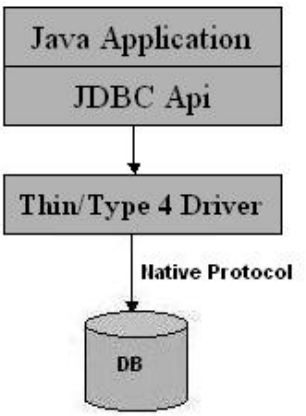
***4*.4.3 Type 3: JDBC-Net pure Java or Middleware Driver**

- The type 3 driver is written entirely in Java. In a Type 3 driver, a three-tier approach is used to accessing databases.
- The JDBC clients use standard network sockets to communicate with an middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.



4.4.4 TYPE – 4: Native-protocol or Pure Java driver or Pure Driver

- The type 4 driver is written completely in Java and is hence platform independent. It is installed inside the Java Virtual Machine of the client.
- It communicates directly with vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.
- MySQL's Connector driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers. Oracle thin driver - oracle.jdbc.driver. OracleDriver which connect to jdbc:oracle:thin URL format.



5 Working with Graphics

The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. The origin of each window is at the top-left corner and is 0,0. Coordinates are specified in pixels. All output to a window takes place through a graphics context. A *graphics context* is encapsulated by the Graphics class and is obtained in two ways:

It is passed to an applet when one of its various methods, such as paint( ) or update( ), is called.

It is returned by the getGraphics( ) method of Component.

The Graphics class defines a number of drawing functions. Each shape can be drawn edge-only or filled. Objects are drawn and filled in the currently selected graphics color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped. Let's take a look at several of the drawing methods.

Drawing Lines

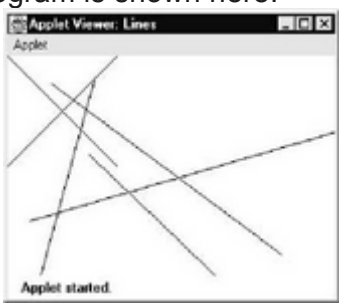
Lines are drawn by means of the drawLine( ) method, shown here:

void drawLine(int *startX*, int *startY*, int *endX*, int *endY*)

drawLine( ) displays a line in the current drawing color that begins at s*tartX*,*startY* and ends at *endX*,*endY*. The following applet draws several lines:

```
import java.awt.*;
import java.applet.*;
/* <applet code="Lines" width=300 height=200>
  </applet>*/
      public class Lines extends Applet {
        public void paint(Graphics g) {
        g.drawLine(0, 0, 100, 100);
        g.drawLine(0, 100, 100, 0);
        g.drawLine(40, 25, 250, 180);
        g.drawLine(75, 90, 400, 400);
        g.drawLine(20, 150, 400, 40);
        g.drawLine(5, 290, 80, 19);
        }
      }
```

Sample output from this program is shown here:



Drawing Rectangles

The drawRect( ) and fillRect( ) methods display an outlined and filled rectangle, respectively. They are shown here:

void drawRect(int *top*, int *left*, int *width*, int *height*)void fillRect(int *top*, int *left*, int *width*, int *height*)

The upper-left corner of the rectangle is at *top*,*left.* The dimensions of the rectangle are specified by *width* and *height.* To draw a rounded rectangle, use drawRoundRect( ) or fillRoundRect( ), both shown here:

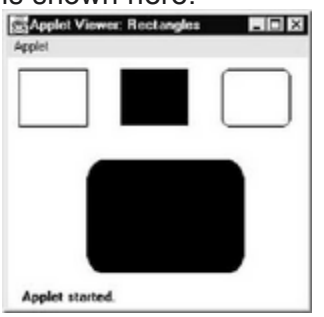
void drawRoundRect(int *top*, int *left*, int *width*, int *height*,int *xDiam*, int *yDiam*)
void fillRoundRect(int *top*, int *left*, int *width*, int *height*,int *xDiam*, int *yDiam*)

A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at *top*,*left.* The dimensions of the rectangle are specified by *width* and *height.* The diameter of the rounding arc along the X axis is specified by *xDiam.* The diameter of the rounding arc along the Y axis is specified by *yDiam.*

The following applet draws several rectangles:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="Rectangles" width=300 height=200>
</applet>
*/
public class Rectangles extends Applet {
public void paint(Graphics g) {
g.drawRect(10, 10, 60, 50);
g.fillRect(100, 10, 60, 50);
g.drawRoundRect(190, 10, 60, 50, 15, 15);
g.fillRoundRect(70, 90, 140, 100, 30, 40);
}
}
```

Sample output from this program is shown here:



Drawing Ellipses and Circles

To draw an ellipse, use drawOval( ). To fill an ellipse, use fillOval( ). These methods are shown here:

void drawOval(int *top*, int *left*, int *width*, int *height*)
void fillOval(int *top*, int *left*, int *width*, int *height*)

The ellipse is drawn within a bounding rectangle whose upper-left corner is specifiedby *top*,*left* and whose width and height are specified by *width* and *height.*To draw a circle, specify a square as the bounding rectangle.

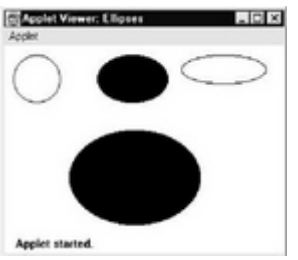
The following program draws several ellipses:

```
// Draw Ellipses
import java.awt.*;
import java.applet.*;
/*
<applet code="Ellipses" width=300 height=200>
</applet>
*/


    public class Ellipses extends Applet {
      public void paint(Graphics g) {
      g.drawOval(10, 10, 50, 50);
      g.fillOval(100, 10, 75, 50);
      g.drawOval(190, 10, 90, 30);
      g.fillOval(70, 90, 140, 100);
      }

}
```

Sample output from this program is shown here:



Drawing Arcs

Arcs can be drawn with drawArc( ) and fillArc( ), shown here:

void drawArc(int *top*, int *left*, int *width*, int *height*, int *startAngle*, int s*weepAngle*)

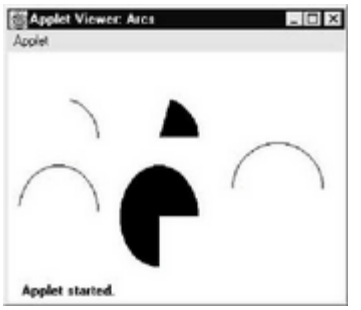
void fillArc(int *top*, int *left*, int *width*, int *height*, int *startAngle*,int *sweepAngle*)

The arc is bounded by the rectangle whose upper-left corner is specified by *top*,*left* and whose width and height are specified by *width* and *height.* The arc is drawn from *startAngle* through the angular distance specified by *sweepAngle.* Angles are specified in degrees. Zero degrees is on the horizontal, at the three o'clock position. The arc is drawn counterclockwise if *sweepAngle* is positive, and clockwise if *sweepAngle* is negative. Therefore, to draw an arc from twelve o'clock to six o'clock, the start angle would be 90 and the sweep angle 180.
The following applet draws several arcs:

```
// Draw Arcs
import java.awt.*;
import java.applet.*;
/*
<applet code="Arcs" width=300 height=200>
</applet>
*/
public class Arcs extends Applet {
```

```
public void paint(Graphics g) {
g.drawArc(10, 40, 70, 70, 0, 75);
g.fillArc(100, 40, 70, 70, 0, 75);
g.drawArc(10, 100, 70, 80, 0, 175);
g.fillArc(100, 100, 70, 90, 0, 270);
g.drawArc(200, 80, 80, 80, 0, 180);
}
}
```

Sample output from this program is shown here:



Drawing Polygons

It is possible to draw arbitrarily shaped figures using drawPolygon( ) and fillPolygon( ), shown here:

void drawPolygon(int *x*[ ], int *y*[ ], int *numPoints*)
void fillPolygon(int *x*[ ], int *y*[ ], int *numPoints*)

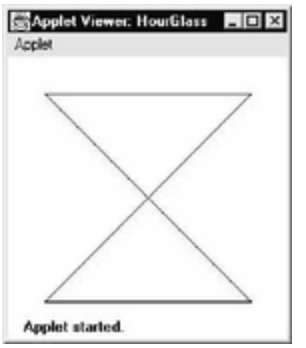
The polygon's endpoints are specified by the coordinate pairs contained within the *x* and *y* arrays. The number of points defined by *x* and *y* is specified by *numPoints.* There are alternative forms of these methods in which the polygon is specified by a Polygon object. The following applet draws an hourglass shape:

```
// Draw Polygon
import java.awt.*;
import java.applet.*;
/*
<applet code="HourGlass" width=230 height=210>
</applet>
*/


public class HourGlass extends Applet {

  public void paint(Graphics g) {
    int xpoints[] = {30, 200, 30, 200, 30};
    int ypoints[] = {30, 30, 200, 200, 30};
    int num = 5;
    g.drawPolygon(xpoints, ypoints, num);
    }
```

Sample output from this program is shown here:


Applet Viewer: HourGlass
Applet
Applet started.