SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

School of Computing
Department of Computer Science and Engineering

UNIT - I

Java Programming-SBSA1302

UNIT-I

An Overview of Java - Features of Java - Object oriented concepts - Lexical Issues - Data Types - Variables - Type Conversions and Casting - Arrays - Operators - Control Statements

1. OOP Concepts:

- OOP treats data as a critical element
- It does not allow the data to flow freely around the system
- It ties data more closely to the functions that operate on it and protects it from unintentional modification by other functions
- OOP allows us to decompose a problem into a number of entities called **objects**
- and build the data and functions around these entities
- The combination of data and methods make up an object.

Object = Data + Method

Features of OOP

- Emphasis on data rather than procedure
- Programs are divided into objects
- Data is hidden cannot be accessed by external functions
- Objects may communicate with each other through methods
- New data and methods can be easily added whenever necessary
- Follows bottom up approach in program design

History of Java

- James Goesling is the Inventor of Java.
- Java was introduced in 1991 by Sun Micro system.
- Java was designed for the development of software for consumer electronic devices like TV,VCRs and such other electronic machines.

Milestones of java

- 1990- Sun Micro System Introduced java
- 1991- Team Announced its name as OAK
- 1995- It renamed as JAVA
- 1997- JDK 1.1 released
- 1998-Oracle

FEATURES OF JAVA

Simple:

- Java will be easier if u understand the concepts of object oriented programming
- Java inherits the c/c++ syntax.
- It is easy to learn.
- Programming in java is easier than c++

Object-Oriented:

- Java is a pure object oriented language.
- Everything is an object in java.
- All programs and data reside inside objects and classes.
- Object models in java is simple and easy extend.

Distributed:

- Java is designed for the distributed environment of the Internet, because it handles
- TCP/IP protocols.
- The original version of Java (Oak) included features for intra address- space messaging. This allowed objects on two different computers to execute procedures remotely.
- Java revived these interfaces in a package called *Remote Method Invocation (RMI). This* feature brings an unparalleled level of abstraction to client/ server programming.

Robust:

- Many languages not focus on memory management or exceptional behavior of the program, so it failed.
- These issues are clearly identified by the java with the techniques such as Garbage collection and Error Handling.
- In Garbage collection java uses a Thread to free the objects which are not in use.
- In Exception handling java uses exclusively written codes to correct the exception situation.

Secure.

- To download a normal program, it is a chance for a virus.
- In addition to viruses, another type of malicious program exists that must be guarded against.
- To Java-compatible Web browser, you can safely download Java applets without fear of viral infection or malicious intent.
- Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer.
- So, there is no security breaches.

Architectural neutral or platform independent

- One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine.

- Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction.
- The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, anytime, forever"

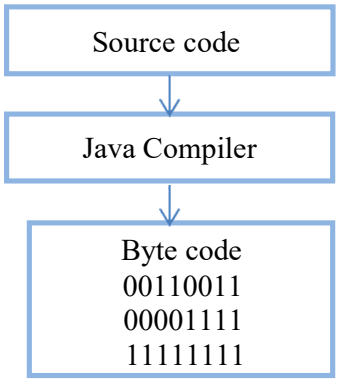

Fig:1.1 Byte Code Conversion

Portable

- Java compiler generates a code called Byte code which is used by any machine
- In java the size of the primitive data types are machine independent.
- Compile and Interpreted
- Generally, computer languages are compiled or interpreted.
- Java is a both compiler and interpreter language.

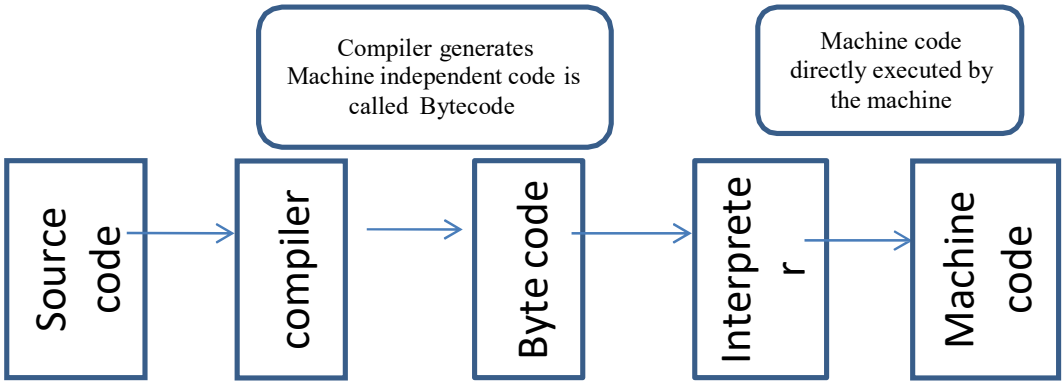

Fig: 1.2 Source code to Machine code generation

High Performance

- Java interpreter uses byte codes. So the performance is high.
- The Speed is also high comparable to other languages.

Multithreaded

- Multithreaded means handling more than one job at a time.

Dynamic

- It is a capable of linking dynamically new classes, methods and objects.
- Java support functions written in other languages such as c and c++. These functions are called native methods.
- During runtime native methods can link dynamically.

LEXICAL ISSUES

- Java programs have a collection of whitespace, identifiers, comments.

Whitespace

- Java is a free-form language. This means that you do not need to follow any special indentation rules.
- In Java, whitespace is a space, tab, or newline.

Identifiers

- It is used for class names, method names and variable names.
- Identifier consists of sequence of uppercase and lowercase letters, numbers or the underscore and dollar-sign characters.
- Identifiers must not begin with a number.
- Java is a case sensitive

Example of valid identifiers are

CseJava result a1 $unit cse_a1_mark1

Example of Invalid identifiers are

2cse 2-cse a1-cse

Literals

- Constant value in java by created using a **literal** representation.

Example:

100 99.2 ‘S’ ‘II cse’

Comment Line

Single Line(/Program1/)

Multi line(/*Line1
Lin2..*/)

Documentation
Comment(/**…….*/)

Separators

- In Java most commonly used separator is Semicolon(;).
- List of separators are:

Symbol	Name	Purpose
()	Parenthesis	Define expressions, Control Statements
{}	Braces	Initialization of arrays
[]	Brackets	For declaring arrays
;	Semicolon	Terminate statements
,	Comma	Used to separate variables and also in FOR loop
.	Period	Separate Package names from subpackage &
		Separate a variable or method from a reference variable

Tabel 1.1 Separators

CONSTANTS

- Constants in java refer to fixed values that do not change during the execution of a program

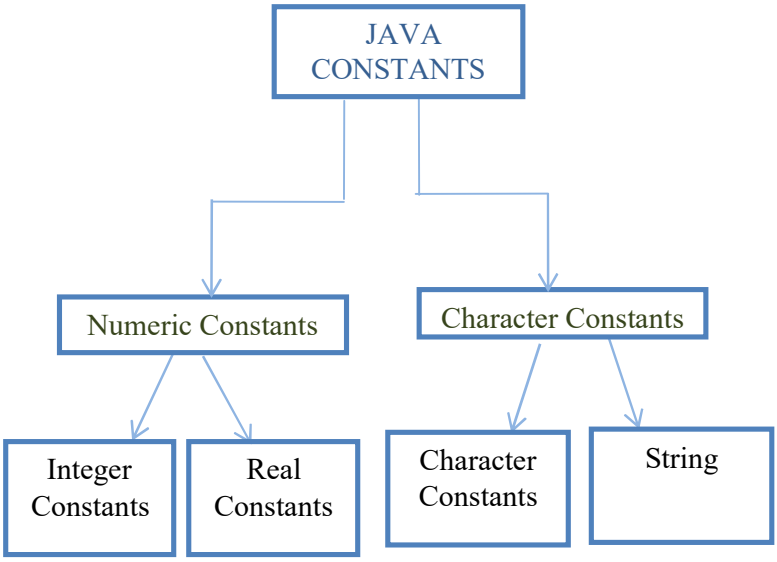

Figure 1.3. Types of Constants

DATA TYPES

- Data type specifies the size and type of values that can be stored.

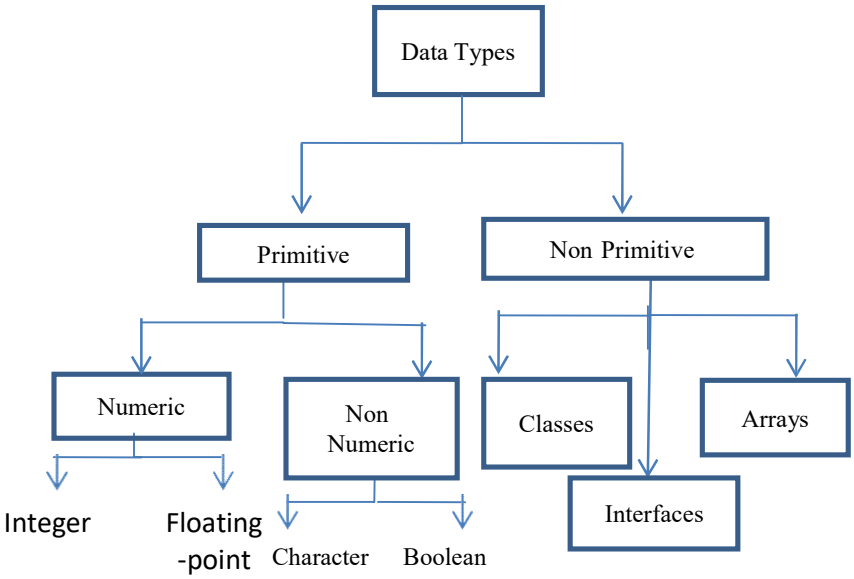

Fig: 1.4. Data Types

Integer

- Integer types can hold whole numbers such as 123,-96 and 5639

Float

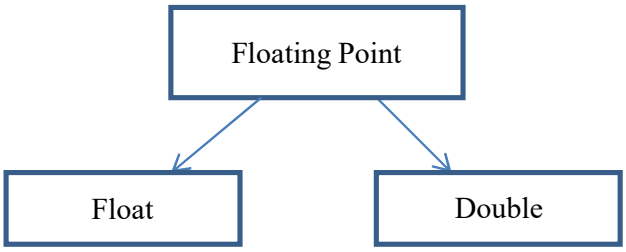

Fig: 1.5. Float Types

Character Type

- Java provides a character data type called CHAR.
- The char type assumes a size of 2bytes but, basically,it can hold only a single character.

Boolean Type

- 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 of storage.
- Boolean values are often used in selection and iteration statements.

OPERATORS

- Operator is a symbol that tells the computer to perform certain mathematical or logical manipulation.
- Operators are used in programs to manipulate data and variables

Types

- ✓ Arithmetic Operator
- ✓ Relational Operator
- ✓ Logical Operator
- ✓ Assignment Operator
- ✓ Increment and decrement Operator
- ✓ Conditional Operator
- ✓ Bitwise Operator

✓ Special Operator

Arithmetic Operator

- Arithmetic operators are used to construct mathematical expressions as in algebra.
- These can operate on any built-in numeric data type of java.
- We cannot use these operators on boolean type.

Operator	Meaning
+	Addition or unary Plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo Division

Table 1.2. Arithmetic Operator

Syntax:

Var1 + Var2

Ex:

```
import java.io.*;
class arith
{
public static void main(String args[])
{
int a,b,c;
a=20;
b=5;
c=a+b;
System.out.println("the output is:"+c);
c=a-b;
System.out.println("the output is:"+c);
c=a*b;
```

```
System.out.println("the output is:"+c);
c=a/b;
System.out.println("the output is:"+c);
c=a%b;
System.out.println("the output is:"+c);
}
}
```

Output:

```
the output is:25
the output is:15
the output is:100
the output is:4
the output is:0
```

Relational Operator

- Relational operators are used to take decisions by comparing two quantities depending on their relations.

Ex: compare age of two persons or price of times.

- These comparisons done with the help of relational operators

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

Table1 1.3 Relational Operator

Syntax:

Ae1 < Ae2

Logical operators

- The logical operators && and || are used for combining two or more relations.
- To combines Two or more relational expressions is termed as logical expression or compound relational expression

Operator	Meaning
&&	Logical And
||	Logical OR
!	Logical Not

Table 1.4. Logical Operator

Syntax:

(var1 > var2) && (var1 >var3)

ASSIGNMENT OPERATOR

- It is used to assign the value of an expression to a variable.(=)

Advantage of shorthand

- More concise and easier to read
- More efficient code

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

Table 1.5 Assignment Operator

Syntax:

v op = exp1;

Increment and Decrement Operator

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

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

++m or m++

---m or m---;

Conditional Operator

- The pair ? : is a ternary operator available in java. This operator is used to construct conditional expression 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 if it is false expression exp3 is evauated.

CONTROL STATEMENTS

- When a program breaks the sequential flow and jumps to another part of code, it is called branching.
- When branching is based on a particular condition ,it is known as conditional branching.
- If branching takes place without any decision, it is known as unconditional branching.

Java language possesses such decision making capabilities and supports the following statements known as control or decision making statements.

1) if statement
2) switch statement
3) conditional operator statement

Types

- Simple if
- If-else
- Nested if-else
- Else if ladder

Simple-If

- Simple if statement is used to execute or skip one statement or group of statements for a particular condition.
- If the condition is true both the statement-block and the statement-x are executed in sequence.

Syntax:

```
if (test condition)
{
Statement block;
}
Statement-x;
```

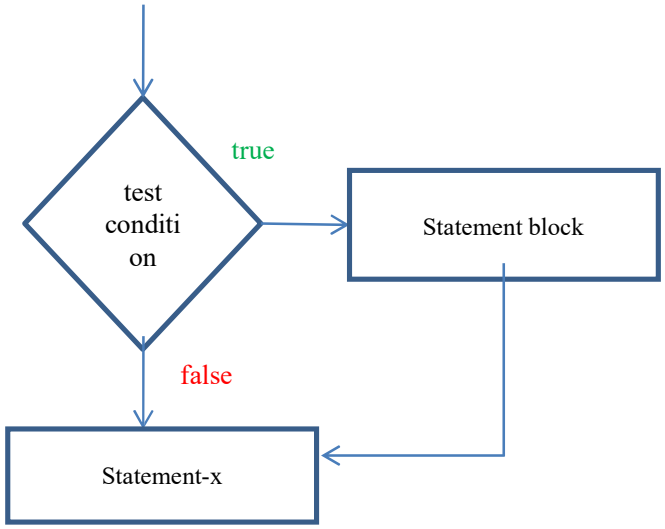

Fig. 1.6 Flowchart of simple if control

EX:

```
Import java.util.*;

Class simpleif
{
Public static void main(String args[])
{
Int age=18;
If(age>=18)
{
System.out.println("elligble for vote");
}
```

```
System.out.println("not elligble for vote");
}
}
```

Output:

elligble for vote

not elligble for vote

If-Else

- if else statement is used to execute one group of statements , if it is true condition. Otherwise it executes false statement.

Syntax:

```
if (test condition)
{
True Statement;
}
Else
{
False Statement;
  }
  Statement-x;
```

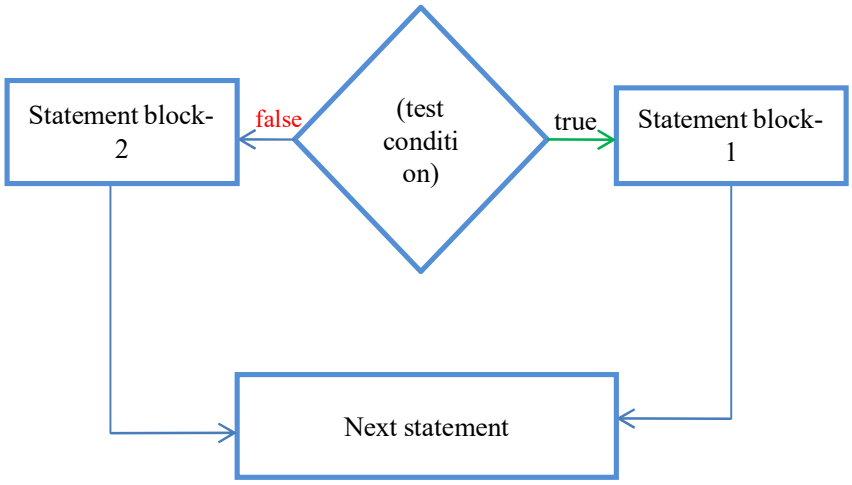

Fig. 1.7 Flowchart of if-else control

Ex:

```
Import java.util.*;

Class simpleif
{
Public static void main(String args[])
{
Int age=18;
If(age>=18)
{
System.out.println(“elligble for vote”);
}
Else
{
System.out.println(“not elligble for vote”);
}
}
```

Output:
elligble for vote

Nested-if

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

Syntax:

```
if (test condition1)
{
if (test condition2)
{
Statement block-1;
}
else
{
Statement block-2;
 } }
else
{
Statement block-3;
 }
statement-x;
```

If condition 1 is false ,the statement block 3 will be executed; otherwise it continuous to perform the second test. If condition 2 true, the statement-1 will be executed; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

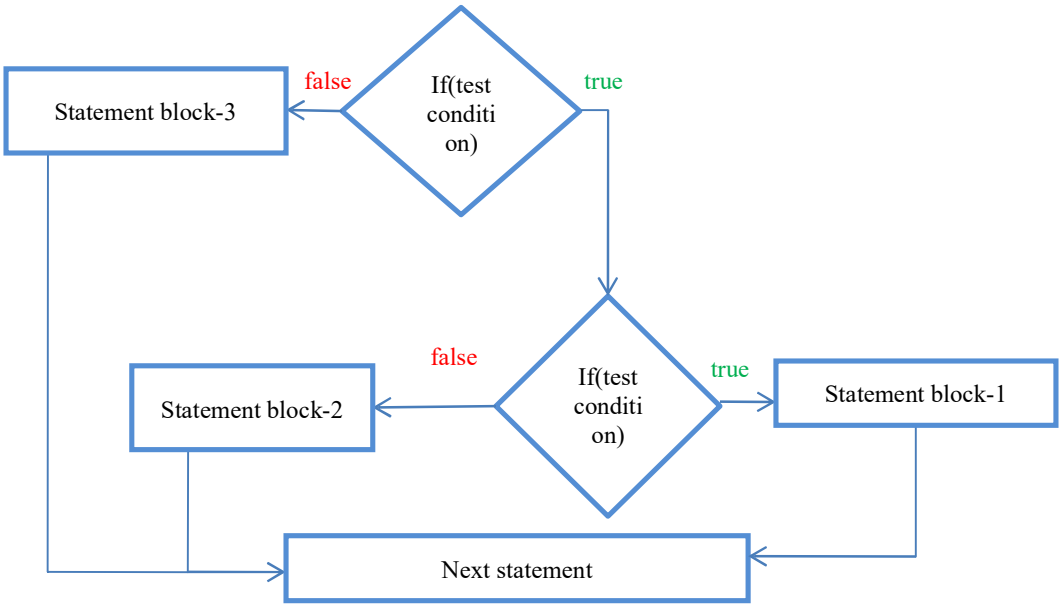

Fig. 1.8 Flowchart of Nested if control

Ex:

```
Import java.util.*;
Class simpleif
{
Public static void main(String args[])
{
Int a=10,b=5,c=3;
if (a>b)
{
if (a>c)
{
System.out.println(“a is greatest”);
}
else
{
System.out.println(“c is greatest”);
 } }
else
{
If(b>c)
```

```
{
System.out.println(“b is greatest”);
 }
Else
{
System.out.println(“c is greatest”);

}}
```

Output:
a is greatest

Else-if Ladder

- Else if ladder statement is used to take multiway decision.

Syntax:

```
if (test condition-1)
{
Statement block-1;
}
Else if (test condition-2)
{
Statement block-2;
}
----------
----------
Else if (test condition-n)
{
Statement block-n;
 }
Else
Default statement;
next statement;
```

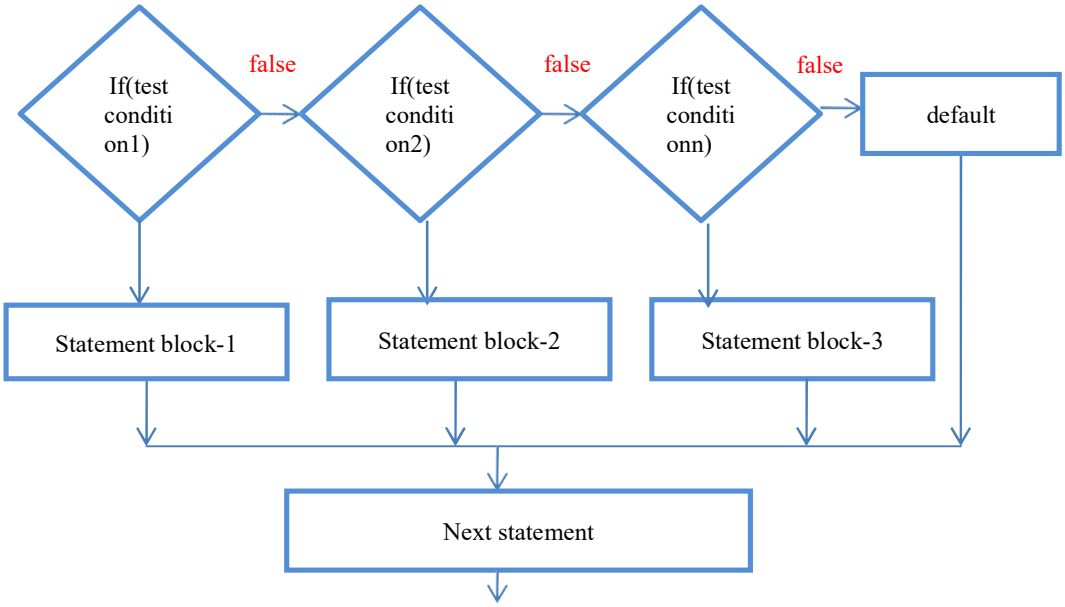

Fig. 1.9 Flowchart of Else if control

Ex :

```
import java.io.*;
class elseifladder
{
public static void main(String args[])
{
int mark=90;
if(mark>=90)
{
System.out.println("honour");
}
else if(mark>=80)
{
System.out.println(" Ist class with distinct");
}
else if(mark>=70)
{
System.out.println("2nd class");
}
else
{
System.out.println("Fail:");
}}}
```

Output:

Honour

Switch

- Java has a built-in multiway decision statement known as switch.
- The switch statement tests the value of a given variable against a list of case values and when a match is found, a block of statements associated with that case is executed.

Syntax:

```
switch (choice)
{
Case 1:
Statement-1;
Break;
Case 2:
Statement-2;
Break;
Case 3:
Statement-3;
Break;
Default:
Default statement;
}
```

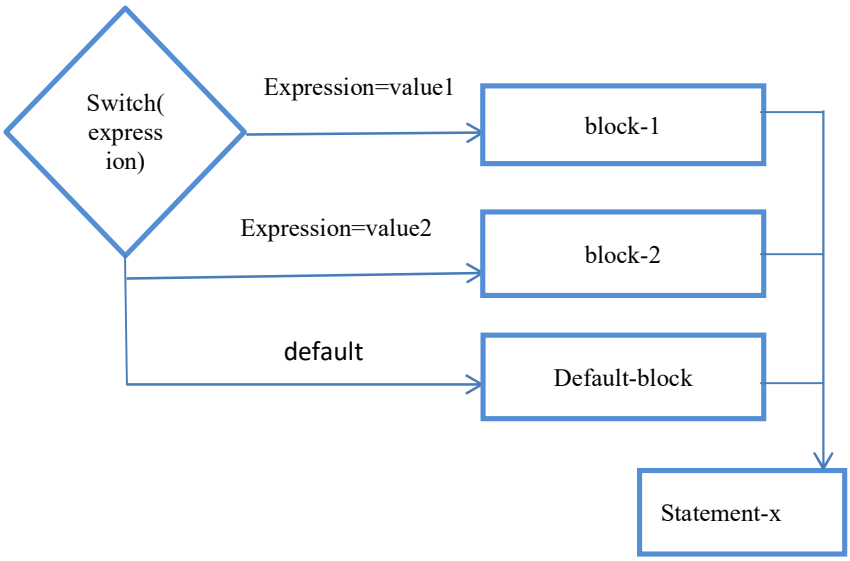

Fig. 1.10 Flowchart of switch

Ex:

```
import java.util.*;
class switch
{
public static void main(String args[])
{
int a,b,c,ch;
Scanner sc=new Scanner(system.in);
a=sc.nextInt();
b=sc.nextInt();
System.out.println("1.addition/n 2.Sub/n 3.mul/n 4.div/n");
switch(ch)
{
case 1:
c=a+b;
System.out.println("Addition"+c);
break;
case 2:
c=a-b;
System.out.println("sub"+c);
break;
case 3:
c=a*b;
System.out.println("mul"+c);
break;
case 4:
c=a/b;
```

```
System.out.println("div"+c);
break;
case 5:
System.exit(0);
}}}
```

Output:

```
10
20
1.addition
2.Sub
3.mul
4.div
Addition 30
```

ARRAY

- Array is a group of contiguous or related data items that share a common name.
- Array value is indicated by writing a number called index number or subscript in brackets after the array name.

Ex:

Salary[10]

Represents the salary of the 10^{th} employee.

The complete set of value is referred to as an array, the individual values are called elements. Arrays can be of any variable type.

Types

- One-dimentional array
- Two-dimentional array
- Multi -dimentional array

One-dimentional array

A list of items can be given one variable name using only one subscript is called a single-subscripted variable or a one dimensional array

Creation of array involves Three Steps

- Declaring array

- Creating memory locations
- Putting values into memory locations

Declaration of Arrays:

declaration of arrays in two forms

DataType arrayname[]; (or) DataType[] arrayname;

Ex:

```
Int      number[ ];
Float  average[ ];
```

Creation of memory space:

After declaring an array ,using new operator the memory will be allocated.

arrayname = new type[size];

ex:

number = new int[5];

average = new float[5];

Two combine the declaration and creation in one step

Int number = new int [5];

Float average = new float [5];

Putting values into memory locations

Put values into an array when it is created is called intialization of array

Arrayname[ ]={ list of values}

Ex:

Int a[]={ 10,20,30,40,50 }

Get Values from User

- Using for loop we can get the value of the array from user with Scanner class.

Steps

i) Create a object for Scanner class

```
Scanner sc = new Scanner(System.in);
```

ii) Using for loop to get value of the array

```
for(i=0 ; i<5; i++)
{
A[i] = sc.nextInt();
}
```

iii) Display the array value using for loop

```
for(i=0; i< 5; i++)
{ System.out.println(a[i]); }
```

Ex:

```
import java.util.*;
class oned
{
public static void main(String args[])
{
int a[]=new int[100],sum=0;
Scanner sc=new Scanner(System.in);
System.out.println("Enter the size of array");
int n=sc.nextInt();
System.out.println("enter the value of the array");
for(int i=0;i<n;i++)
{
a[i]=sc.nextInt();
sum=sum+a[i];
}
System.out.println("the output is");
for(int i=0;i<n;i++)
{
System.out.println(a[i]);
}
System.out.println("sum is"+sum);
}
}
```

Output:
Enter the size of array
2
enter the value of the array
1
2
the output is
1
2
sum is3

Two Dimentional Array

- Two dimensional array used to store values based on table(Ex:Marks of 5subjects for 3 students**)**

Declaration

Int a[][]= new int[3][3];

Initialization

Int a[][]={{1,2,3}{4,5,6}{7,8,9}};

Display the array

```
For(i=0;i<3;i++)
{
For(j=0;j<3;j++)
{
System.out.println(a[i][j]);}}
```

Get Values from User

- Using for loop we can get the value of the array from user with Scanner class.

Steps

Create a object for Scanner class

```
Scanner sc = new Scanner(System.in);
```

Using for loop to get value of the array

```
for(i=0 ; i<5; i++){
for(j=0;j<5;j++){
A[i] = sc.nextInt();
}}
```

Display the array value using for loop

```
for(i=0; i< 5; i++){
for(j=0;j<5;j++){
{ System.out.println(a[i][j]); }}
```

Ex:

```
import java.util.*;
class twod
{
public static void main(String args[])
{
int a[][]=new int [2][2];
int b[][]=new int [2][2];
int c[][]=new int [2][2];
Scanner sc= new Scanner(System.in);
System.out.println("enter matrix A value");
for(int i=0;i<2;i++)
{

        for(int k=0;k<2;k++)
        {

        a[i][k]=sc.nextInt();

        }
}
System.out.println("enter matrix b value");
for(int i=0;i<2;i++)
{

        for(int k=0;k<2;k++)
        {

        b[i][k]=sc.nextInt();
        }
}
System.out.println("output in c");
for(int i=0;i<2;i++)
{
        for(int k=0;k<2;k++)
        {
        c[i][k]=a[i][k]+b[i][k];
        }
}
for(int i=0;i<2;i++)
{
        System.out.println(" ");
        for(int k=0;k<2;k++)
```

```
        {
                System.out.print(c[i][k]+" ");
        }
}}}
```

Output:

enter matrix A value

1 2

3 4

enter matrix b value

2 3

2 4

output in c

3 5

5 8

TYPE CONVERSION AND CASTING

Type casting is used to store a value of one type into a variable of another type.

Syntax:

Type variable1 = (type) variable2;

Automatic Conversion

- It is possible to assign a value of one type to a variable of a different type without a cast.

```
Byte b=75;
Int a=b;
```

LOOPING

- The process of repeatedly executing a block of statements is known as Looping.

Following 4 steps for looping process

- Setting and intialization of counter
- Execution of statements in the loop
- Test for a specified condition for execution of the loop
- Incrementing the counter

Types

Depending on the position of the control statement in the loop ,a control structure may be classified either as the entry -controlled loop or as exit-controlled loop.

While Loop:

- The While is an entry-controlled loop statement

Syntax

The syntax of a while loop is

```
while(Boolean_expression) {
  // Statements
}
```

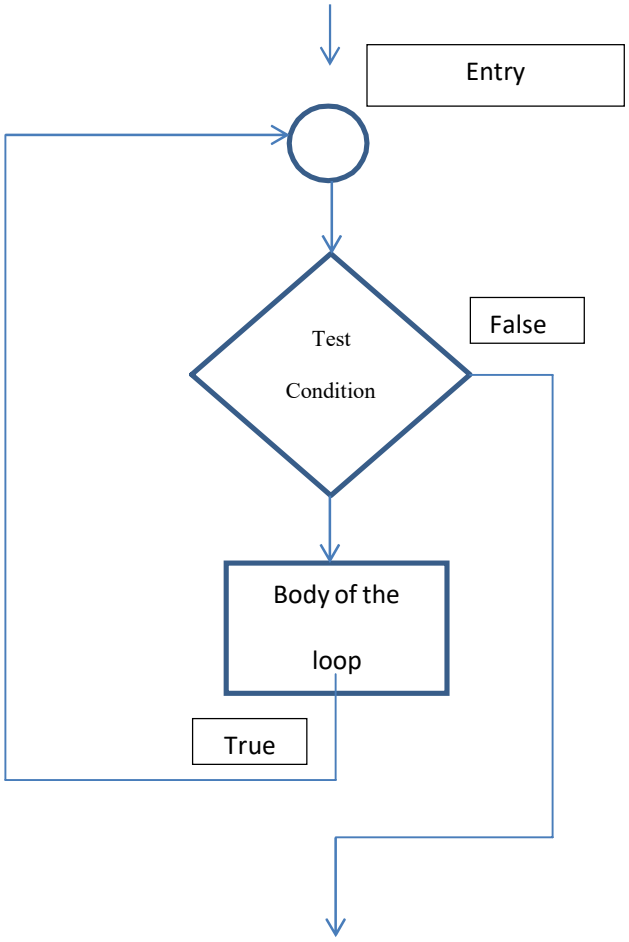

Fig. 1.11 While loop flow diagram

Structure of while loop

Initialization;

While(test condition){

Body of the loop;

Incre/decre

}

Ex: Sum of n numbers

```
Import java.util.*;
Class loop
{
Public static void main(String args[])
{
Int i,sum;
I=0;
While (i<5)
{
Sum=sum+I;
I++;
}
System.out.println(sum);
}
}
```

Output: 15

For Loop

- The For loop is an entry-controlled loop statement

Syntax

```
for (statement 1; statement 2; statement 3) {

  // code block to be executed

}
```

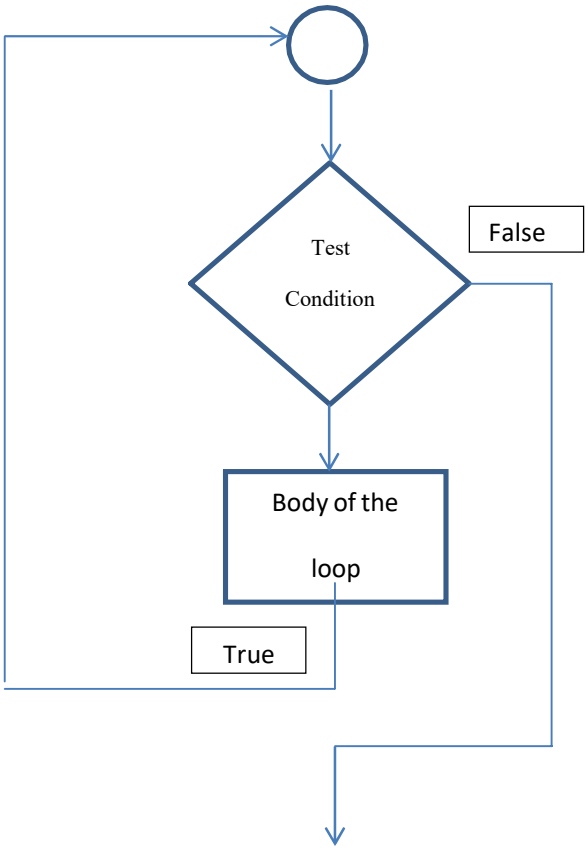

Fig. 12. Flow diagram – for loop

Structure of for loop

For(Initialization; test condition;incre/decre)

{

Body of the loop;

}

Ex: Sum of n numbers

```
Import java.util.*;
Class loop
{
Public static void main(String args[])
{
Int i,sum;
For(i=0; i<5;i++)
{
Sum=sum+I;
}
```

```
System.out.println(sum);
}
}
```

Output:

15

Do-While Loop

- The Do loop is an exit-controlled loop statement.

Syntax

```
Initialization;
Do
{
Statement;
Incre/decre
}while(test condition);
```

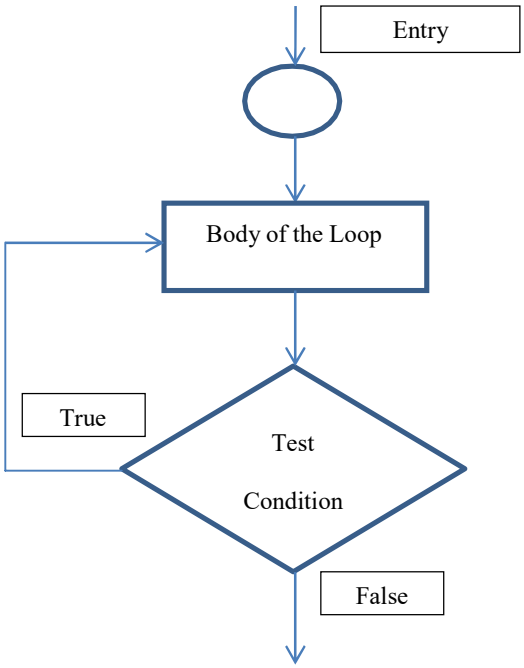

Fig.13. Flow diagram of Do-While Loop

Structure of do-while loop

Intialization;

do

{

Body of the loop;

Incre/decre;

}while(test condition);

Ex: Sum of n numbers

```
Import java.util.*;
Class loop
{
Public static void main(String args[])
{
Int sum;
I=0;
do
{
Sum=sum+I;
i++;
}while(i<5);
System.out.println(sum);
}
}
```

Output:

15

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

School of Computing

Department of Computer Science and Engineering

UNIT - II

Java Programming- SBSA1302

UNIT-II

Introduction to Classes and Objects - Declaring Objects and Methods - Constructors - this keyword - finalize() method- Overloading Methods - Overloading Constructors - Recursion - Access Control - Static and Final methods and variables - Nested and Inner class - String class - String Handling

OBJECTS AND CLASSES

Object:

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

Class

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- **Fields(Variable)**
- **Methods**

Fields :

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Methods:

In Java, a method is like a function which is used to expose the behavior of an object

Syntax of Class:

```
class classname
{
Fields;
methods;
}
```

Example:

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

Save: welcome.java
Compile: javac welcome.java
Interpret: java welcome
Output:
welcome.

Program for fields (variable)

```
class fields
{
int i;
String name;
public static void main(String[] args)
{
fields f=new fields();
System.out.println(f.i+"\n"+f.name);
}
}
```

save :fields.java
compile:javac fields.java
Interpret:java fields
Output:
0
null

2. Variable Intializtion through object

```
class fields1
{
int i;
String name;
public static void main(String[] args)
{
fields f=new fields();
f.i=10;
f.name="cse";
System.out.println(f.i+"\n"+f.name);
}
}
```

E:\basic pgms>javac fields1.java
E:\basic pgms>java fields1
10
cse

3. Program for Variables and Methods

```
class student
{
int rollno;
String name;
void getdata()
{
rollno=123;
name="cse";
}
void display()
{
System.out.println("Rollno:"+rollno+"\n"+"Name:"+name);
}
public static void main(String[] args)
{
student s=new student();
```

```
s.getdata();
s.display();
}
}
```

E:\basic pgms>javac student.java
E:\basic pgms>java student
output:
Rollno:123
Name:cse

4.Passing values to the Instance methods

```
class student1
{
int rollno;
String name;
void getdata(int r,String s)
{
rollno=r;
name=s;
}
void display()
{
System.out.println("Rollno:"+rollno+"\n"+"Name:"+name);
}
public static void main(String[] args)
{
student1 s1=new student1();
s1.getdata(111,"cse");
s1.display();
}
}
```

E:\basic pgms>javac student1.java
E:\basic pgms>java student1
Rollno:111
Name:cse

ACCESS SPECIFIERS:

The access specifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access specifiers:

1. Private: The access level of a private specifier is only within the class. It cannot be accessed from outside the class.
2. Default: The access level of a default specifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. Protected: The access level of a protected specifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. Public: The access level of a public specifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

CONSTRUCTORS

Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined,the constructor is automatically called immediately after the object is created, before the **new** operator completes.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Rules for creating Java constructor

There are three rules defined for the constructor.

- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors

There are two types of constructors in Java:

- Default constructor (no-arg constructor)
- Parameterized constructor

Syntax:

```
class rectangle
{
variable declaration;
rectangle()//construtor
{
}
}
```

Default Constructor

If there is no constructor available in the class It calls a default constructor. In such case, Java compiler provides a default constructor by default.

```
class Default
{
int i;
String s;
double d;
boolean b;
Default()
{
}
public static void main(String[] args)
{
Default d1=new Default();
System.out.println("Int:"+d1.i);
System.out.println("String:"+d1.s);
System.out.println("Double:"+d1.d);
System.out.println("Boolean:"+d1.b);
}
}
```

E:\basic pgms>javac Default.java
E:\basic pgms>java Default
Int:0
String:null
Double:0.0

Boolean:false

Parametarized Constructor:

To add parameters to the constructor is called parametarized constructor.

EX:

```
class circle
{
double rad,area;
double pi;
circle(double r, double  p)
{
rad=r;
pi=p;
}
void  calculate()
{
area=rad*rad*pi;
System.out.println(area);
}
public static void main(String[] args)
{
circle c=new circle(6.2,3.14);
c.calculate();
}
}
```

```
E:\basic pgms>javac circle.java
E:\basic pgms>java circle
120.70160000000001
```

METHOD OVERLOADING

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded,* and the process is referred to as *method overloading.* Method overloading is one of the ways that Java implements polymorphism.

Syntax:

```
class overload
{
void add(int a)
{
}
```

```
void add(int a,int  b)
{
}
double add(double a,double b)
{
}
}
```

EX:

```
class addition
{
int a,b,c;
void add()
{
a=20;b=30;
c=a+b;
System.out.println("simple function:"+c);
}
void add(int a, int b)
{
c=a+b;
System.out.println(" function with argument:"+c);
}
double add(double a,double b)
{
return(a+b);
}

public static void main(String[] args)
{
addition an=new addition();
an.add();
an.add(25,50);
System.out.println("return function:"+an.add(10.5,10.5));
}
}
```

E:\basic pgms>javac addition.java
E:\basic pgms>java addition
simple function:50
function with argument:75
return function:21.0

CONSTRUCTOR OVERLOADING

Same constructor name with different argument is called constructor overloading.

EX:

```
class conaddition
```

```
{
int a,b,c;
double d,e;
conaddition()
{
a=20;b=30;
System.out.println("simple Construtor:"+ (a+b));
}
conaddition(int  a, int b)
{
System.out.println("Construtor with argument:"+ (a+b));
}
 conaddition(double e,double f)
{
System.out.println("double constructor:"+ (e+f));
}
public static void main(String[] args)
{
conaddition an=new conaddition();
conaddition an1=new conaddition(10,20);
conaddition an2=new conaddition(10.5,20.5);
}
}
E:\basic pgms>javac conaddition.java
E:\basic pgms>java conaddition
simple Construtor:50
Construtor with argument:30
double constructor:31.0
```

STATIC

- To create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword **static**.
- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be **static**

Methods declared as **static** have several restrictions:

- They can only call other **static** methods.
- They must only access **static** data.
- They cannot refer to **this** or **super** in any way.

Ex:

```
class staticex
{
static int a = 3;
static int b;
static void meth(int x) {
```

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

E:\basic pgms>javac staticex.java
E:\basic pgms>java staticex
Static block initialized.
x = 42
a = 3
b = 12

FINAL KEYWORD

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

Syntax:

final int a;

```
final void display()
{
}
final class finalex
{
}
```

Final method and final class used in inheritance.

EX:

```
class finalex
{
final int a=20;
int b=10,c;
void display()
{
a=a+b;
System.out.println(" value  a is cannot changed"+a);
```

```
}
public static void main(String[] args)
{
finalex fe=new finalex();
fe.display();
}
}
```

```
E:\basic pgms>javac finalex.java
finalex.java:8: error: cannot assign a value to final variable a
a=a+b;
^
1 error
```

THIS KEYWORD:

this is a reference variable that refers to the current object.

Usage of this keyword

- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)
- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this can be used to return the current class instance from the method.

EX:

```
class thisex
{
int i,fa=1,f;
thisex(int f)
{
this.f=f;
}
int fact()
{
for(i=1;i<=f;i++)
{
fa=fa*i;
}
return fa;
}
public static void main(String[] args)
```

```
{
thisex te=new thisex(5);
System.out.println("factorial is:"+te.fact());
}
}
```

E:\basic pgms>javac thisex.java
E:\basic pgms>java thisex
factorial is:120

NESTED INNER CLASS

- To define class within another class is called nested class.
- Its groups all the classes logically and also achieves the Encapsulation.
- So, the code is more readable and maintainable
- The scope of the inner class always bound with its outer class
- Inner class can access the private members of the outer class.
- Inner classes are divided into two types

  1. static inner class

  2.non static inner class

Syntax:

```
class outerclass
{
Varaible declaration;
Method declaration;
Class innerclass
{
Variable declaration;
Method declaration;
}
}
```

Object Creation for non static inner class

```
class mainclass
{
Public static void main(String args[])
{
Outerclass o=new outerclass();
Outerclass.innerclass oi=o.new innerclass();
}
}
```

```
import java.util.*;
class outer//outer class or enclosing class
{
int x,y;
void getdata()
{
Scanner sc=new Scanner(System.in);
System.out.println("Enetr the value of x and y");
x=sc.nextInt();
y=sc.nextInt();
}
class inner//nesting class or non-static inner class
{
int c;
void display()
{
c=x+y;
System.out.println("Sum:"+c);
}
}
}
class mainclass
{
public static void main(String args[])
{
outer o=new outer();
outer.inner oi=o.new inner();
o.getdata();
oi.display();
}
}
```

Output:

```
enter the value of x and y
10
20
sum30
```

Object Creation for static inner class

Syntax:

```
Class outer
{
Void getdata()
```

```
{ }
Static class  inner
{
Static void dis()
{ }
}
class mainclass
{
Public static void main(String args[])
{
Outerclass.innerclass oi=new outerclass.innerclass();
Oi.getdata();
Inner.dis();
}
}
```

Ex:

```
import java.util.*;
class outerclass
{
  static  int  x,y;
 static void getdata()
{
Scanner sc=new Scanner(System.in);
System.out.println("enter the value of x and y");
x=sc.nextInt();
y=sc.nextInt();
}
  static class innerclass
{
 static int c;
 static void display()
{
c=x+y;
System.out.println("sum"+c);
}
}
}
class nestedclass
{
public static void main(String args[])
{
        //outerclass o=new outerclass();
outerclass.innerclass oi=new outerclass.innerclass();
outerclass.getdata();
```

```
oi.display();
}
}
enter the value of x and y
10
20
sum30
```

STRING

- String represents sequence of characters.
- To represent a string in java using character array.

Ex:

Char array[]=new char[4];

array[0]='j';
array[1]='a'
array[2]='v'
array[3]='a'

Limitations in character array :

not support string operations.

So java handle this problem using String class.

```
String s;
S=new String("iicse");
String s[]= new String[3];
```

STRING HANDLING METHODS

S1.LowerCase;
S1.UpperCase;
S1.replace('x','y');
S1.trim();
S1.equals(s2);
S1.equalgnoreCase(s2);
S1.length();
S1.chatAt(n);
S1.CompareTo(s2);
S1.concat(s2);
S1.substring(n);

S1.substring(n,m);

Tabel 1:String Handling Methods

RECURSION

- Recursion is the process of repeating items in a self-similar way.
- Most computer programming languages support recursion by allowing a function to call itself from within its own code.

Types

1. Direct
2. Indirect
3. Single
4. Multiple
5. Tail

Direct

A function call itself directly called direct method.

Procedure

1. Define base case(baecase->0)

2. Call recursively.

Problem: Sum of N numbers

N=0

Result=0

N=5

1+2+3+4+5=15

Ex: Fibonacci series

```
import java.util.*;
class recursive
{

int add(int s)
{
if(s==0)
return  0;
else if(s==1)
        return 1;
else
return add(s-1)+add(s-2);
```

```
}
public static void main(String args[])
{
        int s;
Scanner sc=new Scanner(System.in);
int sum=0;
recursive re=new recursive();
System.out.println("enter the value of s");
s=sc.nextInt();
for(int i=0;i<=s;i++)
{
        System.out.println(re.add(i));
}
}
}
```

Output:

Enter the value of s:5

120.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

School of Computing
Department of Computer Science and Engineering

UNIT - III

Java Programming- SBSA1302

Inheritance - Using super - Overriding Methods - Abstract classes - Final with Inheritance – Multi Threaded programming

UNIT-III

INHERITANCE

- ✓ Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object**.**
- ✓ Inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- ✓ Inheritance represents the IS-A relationship which is also known as a *parent-child* relationship.

Uses of inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Types of Inheritance in Java

Single Inheritance:

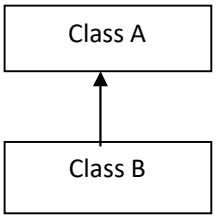

Fig. 3.1. Single Inheritance

Multilevel Inheritance

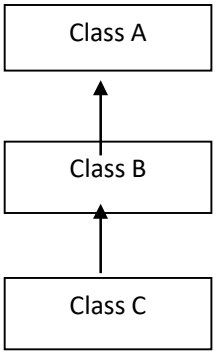

Fig. 3.2. Multiple Inheritance

Hierarchial Inheritance

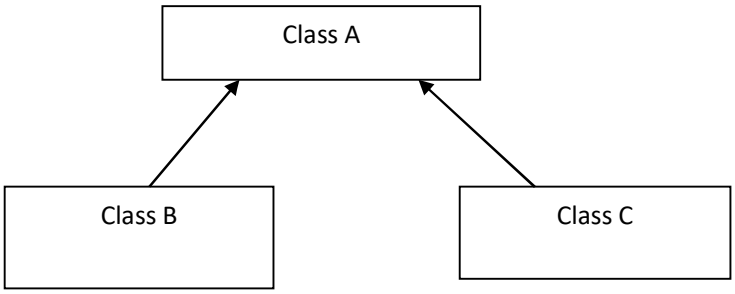

Fig. 3.3 Hierarchial Inheritance

Multiple Inheritance

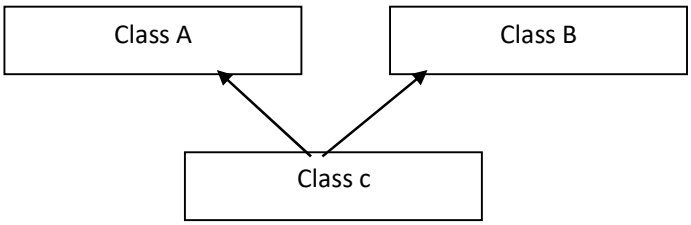

Fig. 3.4. Multiple Inheritance

Hybrid Inheritance

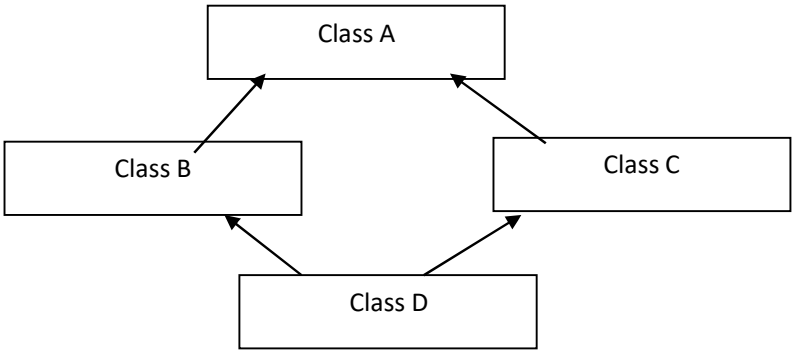

Fig. 3.5. Hybrid Inheritance

Single Inheritance:

one parent class one child class

Ex:

```
class add
{
   int x,y,result;
   public  void sum()
   {
      result=x+y;
   }
}
class sub extends add
{
   public void minus()
   {
      result=x-y;
   }
```

```
}
 public class Singleinherit {
   public static void main(String[] args) {
       sub obj=new sub();
      obj.x=10;
      obj.y=20;
      obj.sum();
      System.out.println(obj.result);
      obj.minus();
      System.out.println(obj.result);
   }
 }
```

E:\basic pgms>javac Singleinherit.java
E:\basic pgms>java Singleinherit

30

-10

Multilevel Inheritance:

When there is a chain of inheritance, it is known as *multilevel inheritance*.

EX:

```
class first
{
   String name;
   first()
   {
     name ="IstYear";
      System.out.println(name);
   }
   void dis()
   {
      System.out.println("6subjects+3lab");
   }
}
class second extends first
{
   second()
   {
      name="IInd year";
      System.out.println(name);
   }
   void dis()
```

```
    {
        System.out.println("6subjects+2lab");
    }
}
public class Multilevel extends second
{
Multilevel()
{
   name="Department";
   System.out.println(name);
}
void dis2()
{
System.out.println("HOD");
}


public static void main(String[] args) {
Multilevel obj=new Multilevel();
obj.dis();
obj.dis1();
obj.dis2();
}
}
```

E:\basic pgms>javac Multilevel.java

E:\basic pgms>java Multilevel

IstYear

IInd year

Department

6subjects+3lab

6subjects+2lab

HOD

Hierarchial Inheritance:

Two child class inherits one parent class is called Hierarchial inheritance

Ex:

```
class first
{
   String s="cse";
   void dis()
```

```
    {
       System.out.println("Name:"+s);
    }
}
class second extends first
{
int m[]={90,98,99,100,100,100};
int total;
void dis1()
    {
for(int i=0;i<m.length;i++)
{
 total=total+m[i];
}
System.out.println("Total:"+total);
    }
}

public class hierarchi extends first
{
void dis2()
{
System.out.println("IIchild");
}
public static void main(String[] args) {
second si=new second();
si.dis();
si.dis1();
//si.dis2();//error
}
 }
```

E:\basic pgms>javac hierarchi.java
E:\basic pgms>java hierarchi

Name:cse

Total:587

METHOD OVERRIDING:

- ✓ Base class and sup class have the same methods, the sub class method only executes.It is known as Overriding.
- ✓ Java has a method provision to executes the same methods of Base and sup class is called "super ".

Ex:

```
class student
{
int rno;
String name;
```

```
void display(int rno,String name)
{
System.out.println(rno+"\n"+name);
}
}
class mark extends student
{
int m1,m2,m3,tot,avg;
void display(int m1,int m2,int m3)
{
super.display(1234,"mahi");
tot=m1+m2+m3;
avg=tot/3;
System.out.println("tot:"+tot+"\n"+"avg:"+" "+avg);
}
}

class overriding
{
public static void main(String args[])
{
mark m1=new mark();
m1.display(90,92,98);
}
}
```

E:\javapgms>javac overriding.java

E:\javapgms>java overriding

1234

mahi

tot:280

avg: 93

ABSTRACT CLASSES

- ✓ Abstraction is a process of hiding the implementation details and showing only functionality to the user.

There are two ways to achieve abstraction in java

1. Abstract class
2. Interface

Rules:

- ✓ An abstract class must be declared with an abstract keyword.
- ✓ It can have abstract and non-abstract methods.
- ✓ It cannot be instantiated.
- ✓ It can have constructors and static methods also.
- ✓ It can have final methods which will force the subclass not to change the body of the method.

Ex:

```
abstract class phone
{
String s,city,add;
int i;
 void f1()
   {
       s= "cse";
      i=12345;
       city="chennai";
       add="sathaybama";
   }
   //abstract public void dis1();
   abstract  public void address();
}

public class Abs1 extends phone {

      public void address()
   {
      System.out.println(s+"\n"+i+"\n"+city+"\n "+add);
   }
   public static void main(String[] args) {
      Abs1 m1=new Abs1();
      m1.f1();
m1.address();
   }   }
```

E:\basic pgms>javac Abs1.java

E:\basic pgms>java Abs1

cse

12345

chennai

sathaybama

USING FINAL WITH INHERITANCE:

- ✓ To disallow a method from being overridden, specify final as a modifier at the start of its declaration.
- ✓ Methods declared as final cannot be overridden.

To implement final in three ways

1.Final Class

2.FInal method

3.Final variable

Ex:

```
final class finalex
{
final int i=10;
}
class inherit extends finalex
{
void display()
{
System.out.println(i);
}
public static void main(String[] args)
{
inherit fx=new inherit();
fx.display();
}
}
```

```
E:\basic pgms>javac inherit.java
inherit.java:5: error: cannot inherit from final finalex

class inherit extends finalex

                ^

1 error
```

MULTI THREADING

Multi threading is a program control. In this the program is divided into two or more independent subprograms called threads and are processed Parallely. Every program has atleast one thread.

In single processor system, multiple threads share the cpu time. This is achieved because a thread does not always need the cpu. That is, it might have to wait for user input or it might have to display something on the screen. During this time other threads take over the cpu. The previous thread can resume after the current thread comes out of the cpu.

The O/S is resposible for scheduled and allocating resources for threads.

Advantages of threads

1. it increases the speed of the execution
2. it allows to run more tasks simultaneously
3. it reduces the complexity of the large programs
4. it maximize cpu utilization

Life cycle of thread

While using thread it enters into the following states.

1. Newborn state
2. Runmode state
3. Running state
4. Blocked state
5. Dead State

Newborn State:

At once the thread object is created for the defined thread a new thread is born. This state of thread is called new born state.

from this state the new born thread can go to any one of the following state

a) Runmode State

b) Dead State.

if start() is called it goes to runmode state.if stop() is called i goes to dead state.

.

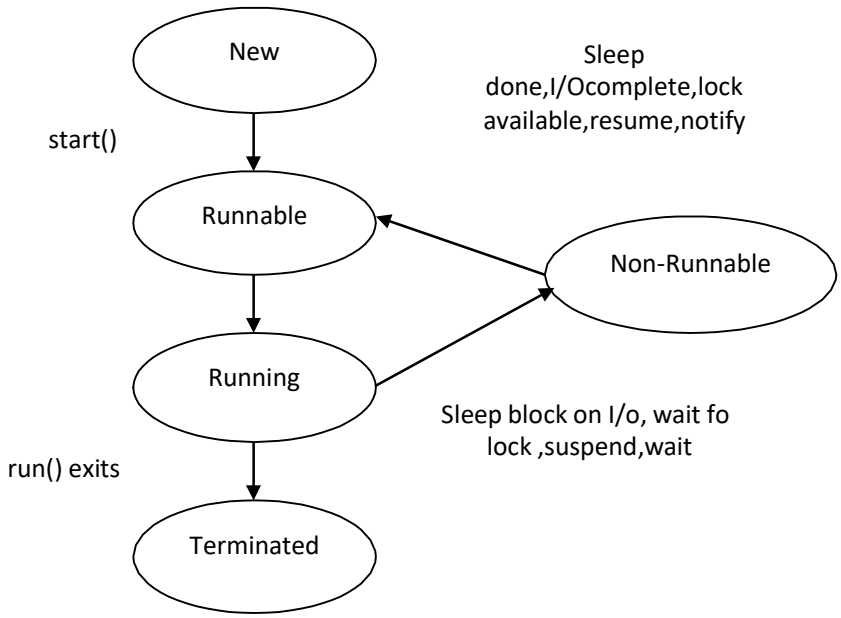

Fig. 3. 6:Thread Life Cycle

Creating a Thread

In the most general sense, you create a thread by instantiating an object of type **Thread**.

Java defines two ways in which this can be accomplished:

- ✓ You can extend the **Thread** class, itself.
- ✓ You can implement the **Runnable** interface.

Create a Thread using Thread class

- ✓ **T**o create a new class that extends Thread,and then to create an instance of that class**.**
- ✓ The extending class must override the run( ) method, which is the entry point for the new thread

Ex:

```
class firstthread  extends Thread
        {
        public  void  run()
                {
                        }}
```

- ✓ call **start( )** to begin execution of the new thread.

Ex:

```
class two extends Thread
{
   public void run()
   {
      for(int i=1;i<=5;i++)
      {

         System.out.println(i+"*2="+(i*2));
```

```
        }
    }
}
class three extends Thread
{
    public void run()
    {
                for(int i=1;i<=5;i++)
                {
System.out.println(i+"*3="+(i*3));

                }
    }
}
class wel extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println("welcome");
        } }}
public class Thread1 {
    public static void main(String[] args) {
         two t1=new two();
        three f1=new three();
        wel w1=new wel();
        t1.start();
        f1.start();
        w1.start();
    }}
```

```
E:\basic pgms\unit iii>javac Thread1.java
E:\basic pgms\unit iii>java Thread1
1*2=2
2*2=4
3*2=6
4*2=8
5*2=10
welcome
welcome
welcome
welcome
welcome
1*3=3
2*3=6
3*3=9
4*3=12
5*3=15
```

Output order is not same. Different compile time you can get different output.

Using Thread Methods Stop() and yield()

Ex:

```
class maths extends Thread
{
public void run()
{
for(int i=1;i<6;i++)
{
if(i==2) stop();
{
System.out.println("maths score is:"+i);
}
}
}
}
class science extends Thread
{
public void run()
{
for(int i=1;i<6;i++)
{
if(i==3)  yield();
{
System.out.println("science score is:"+i);
}}}}
class subthread
{
public static void main(String args[])
{
maths ma=new maths();
science sc=new science();
ma.start();
sc.start();
}}
```

```
E:\javapgms>javac subthread.java
Note: subthread.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
E:\javapgms>java subthread

science score is:1
science score is:2
science score is:3
maths score is:1
science score is:4
science score is:5
```

Imlementing Runnable Interface

- ✓ **T**o create a new class that implements Runnable,and then to create an instance of that class**.**
- ✓ The extending class must override the run( ) method, which is the entry point for the new thread

Ex:

```
class firstthread  implements Runnable
        {
        public  void  run()
                {
                        }}
```

- ✓ **Use Thread ()** to create a object for Thread class. In this to pass the object of each class.
- ✓ Use strart() to start execution.

Ex:

```
class two implements Runnable{
   public void run()
   {
      for(int i=1;i<=5;i++)
      {
           System.out.println(i+"*2="+(i*2));
      }   }}
class three implements Runnable
{
   public void run()
   {
            for(int i=1;i<=5;i++)
            {
System.out.println(i+"*3="+(i*3));
             }   }}
class wel implements Runnable
{
   public void run()
   {
      for(int i=1;i<=5;i++)
      {
         System.out.println("welcome");
      }   }}
public class Runnablethread {
  public static void main(String[] args) {
  two t1=new two();
Thread t=new Thread(t1);
t.start();
three f1=new three();
Thread th=new Thread(f1);
th.start();
```

```
  wel w1=new wel();
Thread th2=new Thread(w1);
th2.start(); }}

E:\basic pgms\unit iii>javac Runnablethread.java
E:\basic pgms\unit iii>java Runnablethread
1*2=2
2*2=4
3*2=6
4*2=8
5*2=10
1*3=3
2*3=6
3*3=9
4*3=12
5*3=15
welcome
welcome
welcome
welcome
welcome
```

Setting the Priority of a Thread:

- ✓ Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- ✓ higher-priority threads get more CPU time than lower priority threads.
- ✓ A higher-priority thread can also preempt a lower-priority one.

Metohds:

setPriority( ):

To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.

This is its general form:

final void setPriority(int *level*)

The value of *level* must be within the range
MIN_PRIORITY
MAX_PRIORITY
These values are 1 and 10, respectively.
To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **final** variables within **Thread**.

getPriority( ):

You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:

final int getPriority( )

EX:

```
public class Thprioity extends Thread {
     public void run()
   {
System.out.println(Thread.currentThread().getPriority());
   }
      public static void main(String[] args) {
       Thprioity obj=new Thprioity();
       Thprioity obj1=new Thprioity();
       Thprioity obj2=new Thprioity();
       obj.setPriority(Thread.MIN_PRIORITY);
       obj1.setPriority(5);
       obj2.setPriority(Thread.MAX_PRIORITY);
       obj.start();
        obj1.start();
        obj2.start();
       }   }
 E:\basic pgms\unit iii>javac Thprioity.java
 E:\basic pgms\unit iii>java Thprioity
 1
 10
 5
```

Synchronization.

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization.*

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.
2. Cooperation (Inter-thread communication in java)

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent.

Ex:

```
class Table{
void printTable(int n){//method not synchronized
 for(int i=1;i<=5;i++){
System.out.println(n*i);
try{
Thread.sleep(400);
}catch(Exception e){System.out.println(e);}
  }
 }
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}}
```

Output: 5
100
10
200
15
300
20
400
25

500

Java synchronized method

- ✓ If you declare any method as synchronized, it is known as synchronized method.
- ✓ Synchronized method is used to lock an object for any shared resource.
- ✓ When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output: 5

10
15
20
25
100
200
300
400
500

School of Computing

Department of Computer Science and Engineering

UNIT - IV

Java Programming- SBSA1302

Unit -IV

Packages - Access Protection – Importing Packages – Interfaces – Exception Handling – throw and throws. – Sting tokenizer – Date – Calendar- Random

4. INTERFACE:

- ✓ To acheive the multiple and hybrid inheritance, interfaces are used.
- ✓ interface can specify what a class must do, but not how it does it.
- ✓ Interfaces are syntactically similar to classes, but they lack instance variables,and their methods are declared without any body.
- ✓ Once it is defined, any number of classes can implement an interface. Also, one class
- ✓ can implement any number of interfaces.
- ✓ By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Defining an Interface:

- ✓ An interface is defined much like a class.

This is the general form of an interface:

access interface *name* {

return-type method-name1(*parameter-list*);

return-type method-name2(*parameter-list*);

type final-varname1 = value;

type final-varname2 = value;

// ...

return-type method-nameN(*parameter-list*);

type final-varnameN = value;

}

Rules for defining an interface

- ✓ Access is public, an interface can be used by any other code.
- ✓ The methods which are declared have no bodies(abstract methods)
- ✓ Variables can be declared inside of interface declarations.
- ✓ They are impicitly final and static means they cannot be changed by the implemented class.
- ✓ They must be initalized with a constant value.

Ex:

interface student

{

void display();

}

Implementing interfaces

- ✓ Once an interface has been defined, one or more classes can implement that interface.
- ✓ To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.

Ex:

```
interface  student
{
int i=10;
public void display(String s);
}
class A implements student
{
public void display(String s)
{
if(i%2==0)
{
System.out.println("My first interface"+s);
System.out.println("i is even:"+i);
}
}
}
class interfaceex
{
public static void main(String[] args)
{
A a1=new A();
a1.display("student");
}}
```

E:\basic pgms\unit ii>javac interfaceex.java
E:\basic pgms\unit ii>java interfaceex

My first interfacestudent

i is even:10

EXTENDING INTERFACES

One interface can inherit another by use of the keyword extends.

EX:

```
interface A {
void meth1();
void meth2();
}

interface B extends A {
void meth3();
}

class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
}
}
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

E:\basic pgms\unit ii>javac IFExtend.java
E:\basic pgms\unit ii>java IFExtend

Implement meth1().

Implement meth2().

Implement meth3().

Imlement Multiple Inheritance through Interface

Ex:

```
interface area
{
   final static float pi=3.14F;
   float compute(float x,float y);
}
class rectangle implements area
{
   public float compute(float x,float y)
   {
      return(x*y);
   }
}
class circle implements area
{
   public float compute(float x,float y)
   {
      return(pi*x*x);
   }
}
public class Interface1 {
   public static void main(String[] args) {
      rectangle rect=new rectangle();
      circle c=new circle();
      area a;
      a=rect;
      System.out.println("area of rectangle"+a.compute(10,20));
      a=c;
      System.out.println("area of circle"+a.compute(10,0));
   }
   }
```

output:
area of rectangle200.0

area of circle314.0

PACKAGES:

- ✓ A java package is a group of similar types of classes, interfaces and sub-packages.
- ✓ Package in java can be categorized in two form, built-in package and user-defined package.
- ✓ There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1. Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2. Java package provides access protection.
3. Java package removes naming collision.

Defining a package:

- ✓ To create a package is quite easy.
- ✓ simply include a package command as the first as the first statement in a Java source file.

This is the general form of the **package** statement:

```
package pack;
```

Here, pack is the name of the package.

Conditions for Create a package

- ✓ **pack** must be stored in a directory called **pack**.
- ✓ Remember that case is significant, and the directory name must match the package name exactly.
- ✓ Create a .java file and stored under the pack directory.We can create many different .java files and stored under the pack directory.

Simple package example

```
package pack;
public class a
{
public void msg()
{
```

```
        System.out.println("welcome");
        }
}
```

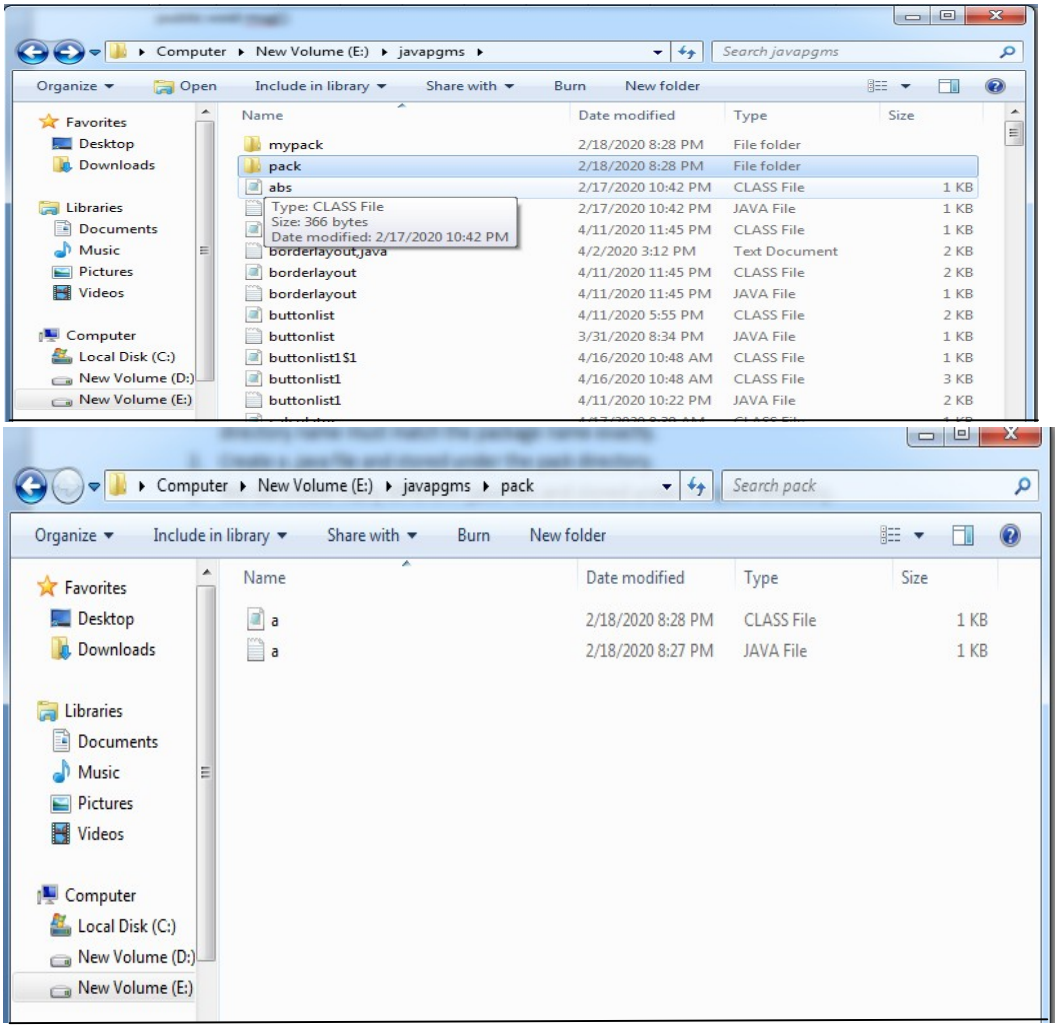

Fig. 4.1. Directory of package a

IMPORTING PACKAGES

- ✓ Java includes the **import** statement to bring certain classes, or entire packages, into visibility.
- ✓ Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program.
- ✓ In a Java source file, **import** statements occur immediately following the **package**
- ✓ statement (if it exists) and before any class definitions.

This is the general form of the import statement:

```
import  packagename.*;
or
import packagename.classname;
```

Ex:

import pack.*; (or) import pack.a;

Importing the package 'pack'

```
import pack.*;
public class calculator
{
public static void main(String args[])
{
a a1=new a();
a1.msg();
}}
```

Condition to save 'calculator.java' file

- ✓ Don't save claculator.java under 'pack' folder.
- ✓ To store the calculator.java in outside of the 'pack' folder.
- ✓ Make sure 'calculator.java' and 'pack' is in the same Drive("E: (or) c: (or) d:")

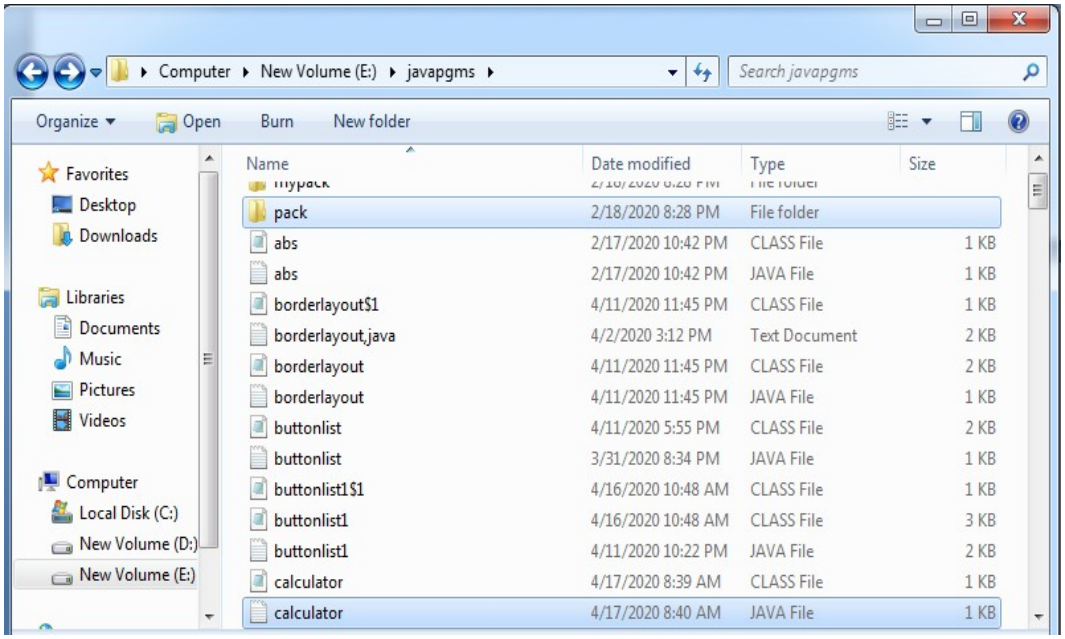

Fig. 4.2. Directory of calculator Package

How to RUN the Package

1. No need to run the package.
2. Run the .java file which imported the package.

E:\javapgms>javac calculator.java

E:\javapgms>java calculator

welcome

Access Two Package in a single class:

1. Create a package 'pack'.
2. Create a file 'a.java' and stored under 'pack'.

```
package pack;
public class a
{
public void msg()
{
System.out.println("welcome");
}
}
```

1. Create a package 'mypack'
2. Create class 'arith' and stored under 'mypack'

```
package  mypack;
import java.util.*;
public class arith
{
public void cal()
{
int a,b,c;
Scanner sr=new Scanner(System.in);
System.out.println("enter the value of a");
a=sr.nextInt();
System.out.println("enter the value of b");
b=sr.nextInt();
```

```
c=a+b;
System.out.println("Addition:"+c);
c=a-b;
System.out.println("Subtraction:"+c);
c=a*b;
System.out.println("Multipliction:"+c);
c=a/b;
System.out.println("Division:"+c);
}
}
```

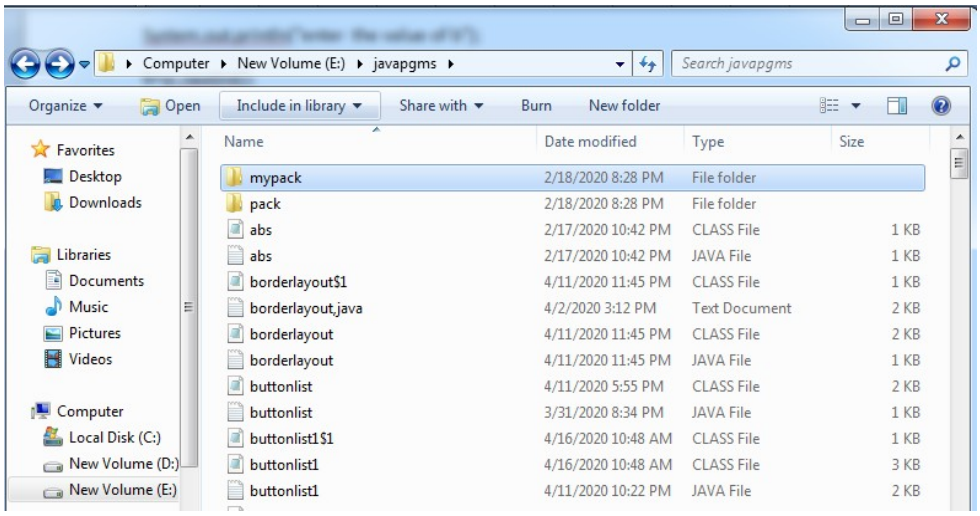

Fig 4.3 Directory of mypack and pack

Create a class'calculator' to access the package 'pack' and 'mypack'

```
import pack.*;
import mypack.*;
public class calculator
{
public static void main(String args[])
{
a a1=new a();
arith ar=new arith();
a1.msg();
ar.cal();
}
}
```

E:\javapgms>javac calculator.java
E:\javapgms>java calculator

welcome

enter the value of a

10

enter the value of b

20

Addition:30

Subtraction:-10

Multipliction:200

Division:0

ACCESS PROTECTION

- ✓ Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- ✓ Packages act as containers for classes and other subordinate packages
- ✓ Classes act as containers for data and code.

In packages, Java addresses four categories of visibility for class members:

- ✓ Subclasses in the same package
- ✓ Non-subclasses in the same package
- ✓ Subclasses in different packages
- ✓ Classes that are neither in the same package nor subclasses

The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package Subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Subclass Different	No	No	Yes	Yes

Package subclass				
Different package non-subclass	No	No	No	Yes

Tabel 4.1 Access Specifier of Package

Ex:

Private:

In 'pack' 'class a' and 'void msg' declared as private.

```
package pack;
 class a
{
void msg()
{
System.out.println("welcome");
}
}
```

E:\javapgms>javac calculator.java
calculator.java:7: error: a is not public in pack; cannot be accessed from outside package

a a1=new a();

^

calculator.java:7: error: a is not public in pack; cannot be accessed from outside package

a a1=new a();

^

2 errors

Protected:

```
package pack;
 protected class a
{
```

```
protected void msg()
{
System.out.println("welcome");
}
}
```

E:\javapgms>javac calculator.java
.\pack\a.java:2: error: modifier protected not allowed here

protected class a

^

calculator.java:7: error: a is not public in pack; cannot be accessed from outsi

de package

a a1=new a();

^

calculator.java:7: error: a is not public in pack; cannot be accessed from outsi

de package

a a1=new a();

^

3 errors

EXCEPTION HANDLING IN JAVA

The **exception handling in java** is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

Exception

Exception is an abnormal condition. In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL,Remote etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**.

Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

The classes that extend Throwable class except Runtime Exception and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend Runtime Exception are known as unchecked exceptions e.g. Arithmetic

Exception, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions

are not checked at compile-time rather they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, Assertion Error etc.

Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where Arithmetic Exception occurs

If we divide any number by zero, there occurs an Arithmetic Exception.

```
1. int a=50/0;//Arithmetic Exception
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
1. String s=null;
2. System.out.println(s.length());//NullPointerException
```

3) Scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur

NumberFormatException.

```
1. String s="abc";
2. int i=Integer.parseInt(s);//NumberFormatException
```

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

Syntax of java try-catch

```
try
{
//code that may throw exception
}
catch(Exception_class_Name ref)
{}
```

Syntax of try-finally block

```
try
{
//code that may throw exception
}
```

finally{}

Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only.You can use multiple catch block with a single try.

Problem without exception handling

```
public class Testtrycatch1
{
public static void main(String args[]
{
int data=50/0;//may throw exception
System.out.println("rest of the code...");
}
}
```

Output:
Exception in thread main java.lang.ArithmeticException:/ by zero

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

Ex:(with try catch block)

```
public class Testtrycatch2
{
public static void main(String args[])
{
try
{
int data=50/0;
}
catch(ArithmeticException e)
{
System.out.println(e);
}
System.out.println("rest of the code...");
}
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

rest of the code...

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code...

statement is printed.

Java catch multiple exceptions

Java Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multicatch block.

Ex.

```
public class TestMultipleCatchBlock
{
public static void main(String args[])
{
try
{
int a[]=new int[5];
a[5]=30/0;
}
catch(ArithmeticException e){System.out.println("task1 is completed");}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("task 2 compltd");
}
catch(Exception e){System.out.println("common task completed");}
System.out.println("rest of the code...");
}
}
```

Output:task1 completed
rest of the code...

Java nested try example

```
class Excep6
{
public static void main(String args[])
{
Try
{
Try
{
System.out.println("going to divide");
int b =39/0;
}catch(ArithmeticException e){System.out.println(e);
}
Try
{
int a[]=new int[5];
a[5]=4;
}catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
System.out.println("other statement);
}catch(Exception e){System.out.println("handeled");}
Syste m.out.println("normal flow..");
}
}
```

finally block

- ✓ It is a block that is used *to execute important code* such as closing connection,stream etc.
- ✓ Java finally block is always executed whether exception is handled or not.
- ✓ Java finally block must be followed by try or catch block.
- ✓ If you don't handle exception before terminating the program, JVM executes finally block(if any)

Need of finally

- ✓ Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

```
class TestFinallyBlock1

{

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

```
{
Try
{
int data=25/0;
System.out.println(data);
}
catch(NullPointerException e){System.out.println(e);}
finally{System.out.println("finally block is always executed");}
System.out.println("rest of the code...");
}
}
```

Output:finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero

THROW EXCEPTION

throw keyword

- ✓ The Java throw keyword is used to explicitly throw an exception.
- ✓ We can throw either checked or uncheked exception in java by throw keyword.
- ✓ The throw keyword is mainly used to throw custom exception.

The syntax of java throw keyword is given below.

1. **throw** exception;

Ex of throw IOException.

1. **throw new** IOException("sorry device error);

java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1
{
```

```
static void validate(int age)
{
if(age<18)
throw new ArithmeticException("not valid");
else
System.out.println("welcome to vote");
}
public static void main(String args[])
{
validate(13);
System.out.println("rest of the code...");
}
}
```

Output:
Exception in thread main java.lang.ArithmeticException:not valid

throws keyword

- ✓ The Java throws keyword is used to declare an exception.
- ✓ It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- ✓ Exception Handling is mainly used to handle the checked exceptions.
- ✓ If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of throws

return_type method_name() **throws** exception_class_name

{

//method code

}

Which exception should be declared

checked exception only, because:

• **unchecked Exception:** under your control so correct your code.

• **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of throws keyword

- ✓ Now Checked Exception can be propagated (forwarded in call stack).
- ✓ It provides information to the caller of the method about the exception.

```
import java.io.IOException;
class Testthrows1
{
void m()throws IOException
{
throw new IOException("device error");//checked exception
}
void n()throws IOException
{
m();
}
void p()
{
Try
{
n();
}catch(Exception e){System.out.println("exception handled");}
}
public static void main(String args[])
{
Testthrows1 obj=new Testthrows1();
obj.p();
System.out.println("normal flow...");
}
}
```

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between

final, finally and finalize are given below:

No. Final finally finalize

1) Final is used to apply restrictions on class, method and variable. Final classcan't be inherited, final method can't be overridden and final variable value can't be changed.

Finally is used to place important code, it will be executed whether exception is handled or

not.

Finalize is used to perform clean up processing just before object is garbage collected.

2) Final is a keyword. Finally is a block. Finalize is a method.

Java final example

```
class FinalExample
{
public static void main(String[] args)
{
final int x=100;
x=200;//Compile Time Error
}
}
```

Java finally example

```
class FinallyExample
{
public static void main(String[] args)
{
Try
{
int x=300;
}
catch(Exception e){System.out.println(e);}
finally{System.out.println("finally block is executed");}
}
}
```

Java finalize example

```
class FinalizeExample
{
public void finalize()
{
System.out.println("finalize called");
}
public static void main(String[] args)
{
FinalizeExample f1=new FinalizeExample();
FinalizeExample f2=new FinalizeExample();
f1=null;
```

```
f2=null;
System.gc();
}
}
```

School of Computing

Department of Computer Science and Engineering

UNIT - V

Java Programming- SBSA1302

UNIT-V

Applet - AWT Controls - Layout Managers and Menus- -Swing - JDBC/ODBC Connection

5. APPLET

Applets are small Java applications that can be accessed on an Internet server, transported over Internet, and can be automatically installed and run as apart of a web document.

Any applet in Java is a class that extends the java.applet.Applet class.

Differences between an applet and Java application

- ✓ An applet is a Java class that extends the java.applet.Applet class.
- ✓ A main() method is not invoked on an applet, and an applet class will not define main().
- ✓ Applets are designed to be embedded within an HTML page.
- ✓ When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- ✓ A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- ✓ The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- ✓ Applets have strict security rules that are enforced by the Web browser.
- ✓ The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- ✓ Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Lifecycle of Java Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

Lifecycle methods for Applet:

- ✓ The java.applet.Applet class 4 life cycle methods and java.awt.
- ✓ Component class provides 1 life cycle methods for an applet.

Four life cycle methods of applet.

1. public void init(): is used to initialized the Applet. It is invoked only once.
2. public void start(): is invoked after the init() method or browser is maximized. It is used to start the Applet.
3. public void stop(): is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. public void destroy(): is used to destroy the Applet. It is invoked only once. java.awt.Component class

The Component class provides 1 life cycle method of applet.

1. public void paint(Graphics g): is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

Example

```
import java.applet.*;
import java.awt.*;
/*
<applet code="simple" width=400 height=400>
</applet>
*/
public class simple extends Applet
{
public void paint(Graphics g)
{
g.drawString ("A Simple Applet", 25, 50);
}
}
```

```
E:\basic pgms\unit iv>javac simple.java
E:\basic pgms\unit iv>appletviewer simple.java
```

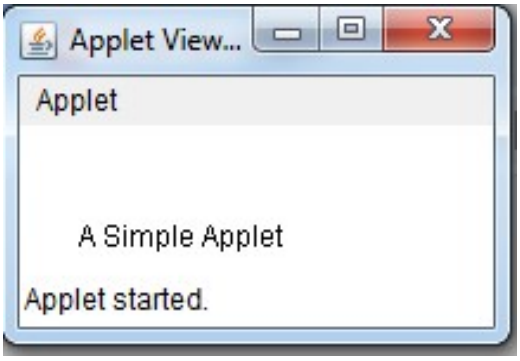

Fig. 5. 1 Output of simple.java by using appletviewer

Advantages of Applets

1. Very less response time as it works on the client side.
2. Can be run using any browser, which has JVM running in it.

Example of an Applet Skeleton

```
import java.awt.*;
import java.applet.*;
public class AppletTest extends Applet
{
public void init()
{
//initialization
}
public void start ()
{
//start or resume execution
}
public void stop()
{
//suspend execution
}
public void destroy()
{
//perform shutdown activity
}
public void paint (Graphics g)
{
//display the content of window
}
}
```

Example of an Applet

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

```
{
public void paint(Graphics g)
{
g.setColor(Color.red);
g.drawRect(100,100,50,50);
}
}
```

E:\basic pgms\unit iv>javac MyApplet.java

E:\basic pgms\unit iv>appletviewer MyApplet.java

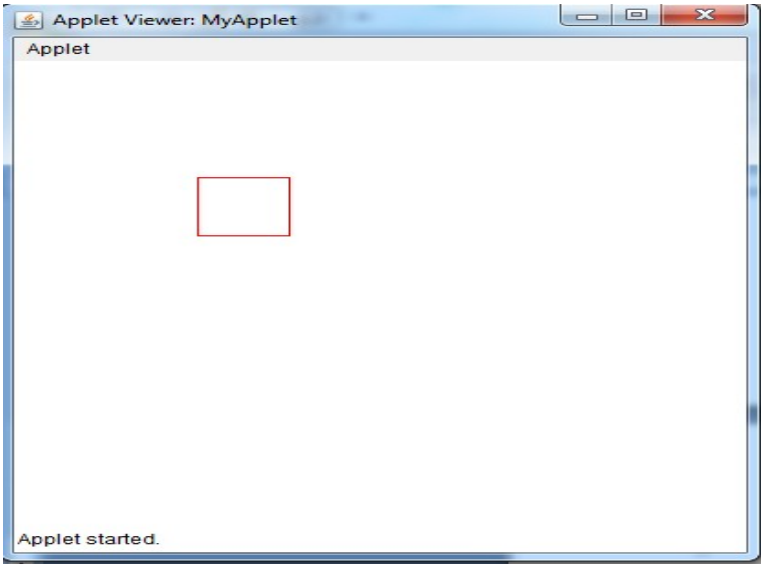

Fig. 5.2 Output of MyApplet.java

//Add two numbers In Applet with HTML

```
import java.awt.*;
import java.applet.*;
/*
<applet code="addition" width=400 height=400>
</applet>*/
public class addition extends Applet
{
public void paint(Graphics g)
{
int  a=10,b=20,c;
String s="";
c=a+b;
s="sum"+String.valueOf(c);
g.drawString(s,100,100);
```

```
}
}
```
E:\basic pgms\unit iv>javac addition.java
E:\basic pgms\unit iv>appletviewer addition.java

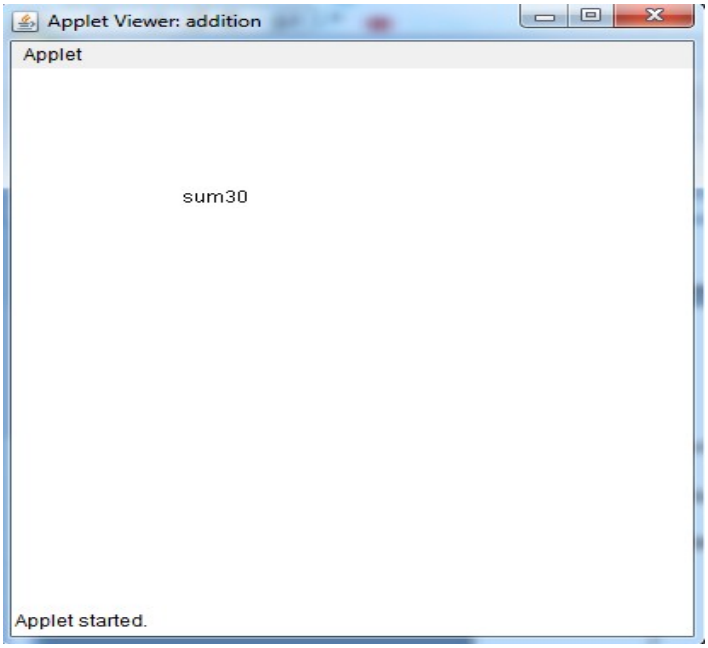

Fig. 5.3 Output of addition.java

How to run an Applet?

There are two ways to run an applet
1. By html file.
2. By appletViewer tool (for testing purpose).
Simple example of Applet by html file:
To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

```
import java.applet.Applet;
import  java.awt.Graphics;
public class First extends Applet
{
public void paint(Graphics g)
{
g.drawString("welcome",150,150);
}
}
```

myapplet.html

```
<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```

Simple example of Applet by appletviewer tool:

To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.

```
import java.applet.Applet;
import  java.awt.Graphics;
public class First extends Applet
{
public void paint(Graphics g)
{
g.drawString("welcome to applet",150,150);
}
}
```

DISPLAYING GRAPHICS IN APPLET

java.awt.Graphics class provides many methods for graphics programming.

Commonly used methods of Graphics class:

1. public abstract void drawString(String str, int x, int y): is used to draw the specified string.
2. public void drawRect(int x, int y, int width, int height): draws a rectangle with the specified width and height.
3. public abstract void fillRect(int x, int y, int width, int height): is used to fill rectangle with the default color and specified width and height.
4. public abstract void drawOval(int x, int y, int width, int height): is used to draw oval with the specified width and height.
5. public abstract void fillOval(int x, int y, int width, int height): is used to fill oval with the default color and specified width and height.
6. public abstract void drawLine(int x1, int y1, int x2, int y2): is used to draw

line between the points(x1, y1) and (x2, y2).

7. public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer): is used draw the specified image.

8. public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle): is used draw a circular or elliptical arc.

9. public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle): is used to fill a circular or elliptical arc.

10.public abstract void setColor(Color c): is used to set the graphics current color to the specified color.

11.public abstract void setFont(Font font): is used to set the graphics current font to the specified font.

Example of Graphics in applet:

```
import java.applet.Applet;
import java.awt.*;
public class GraphicsDemo extends Applet{
public void paint(Graphics g){
g.setColor(Color.red);
g.drawString("Welcome",50, 50);
g.drawLine(20,30,20,300);
g.drawRect(70,100,30,30);
g.fillRect(170,100,30,30);
g.drawOval(70,200,30,30);
g.setColor(Color.pink);
g.fillOval(170,200,30,30);
g.drawArc(90,150,30,30,30,270);
g.fillArc(270,150,30,30,0,180);
}
}
```

myapplet.html

```
 <html>
 <body>
<applet code="GraphicsDemo.class" width="300" height="300">
</applet>
 </body>
</html>
```

```
E:\basic pgms\unit iv>javac GraphicsDemo.java
```

E:\basic pgms\unit iv>appletviewer myapplet.html

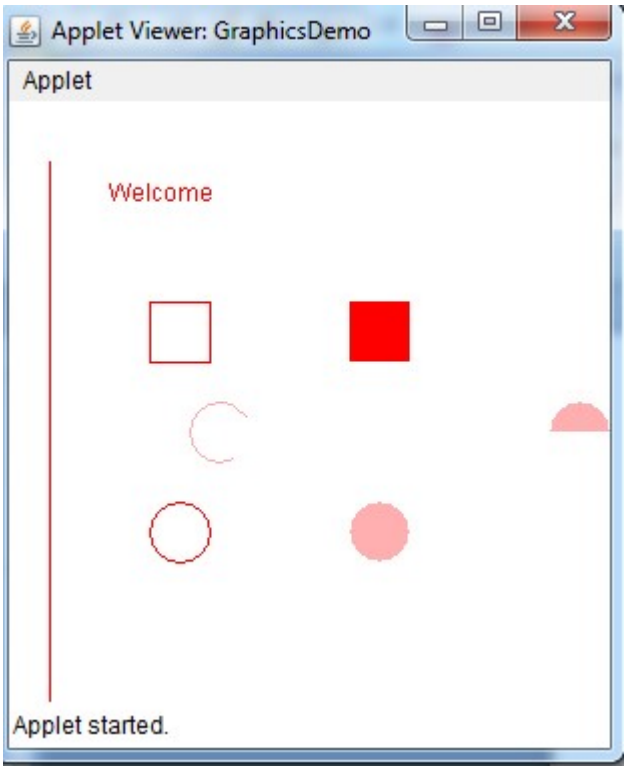

Fig. 5.4 Output of myapplet.java

PARAMETER IN APPLET

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter().

Syntax:

public String getParameter(" parameterName")

Example of using parameter in Applet:

```
import java.applet.Applet;
import java.awt.Graphics;
public class UseParam extends Applet{
public void paint(Graphics g){
String str=getParameter("msg");
g.drawString(str,50, 50);
 }
}
```

param.html
<html>
<body>
<applet code="UseParam.class" width="300" height="300">
 <param name="msg" value=" Sathyabama University">
 </applet>
 </body>
 </html>

E:\basic pgms\unit iv>javac UseParam.java

E:\basic pgms\unit iv>appletviewer param.html

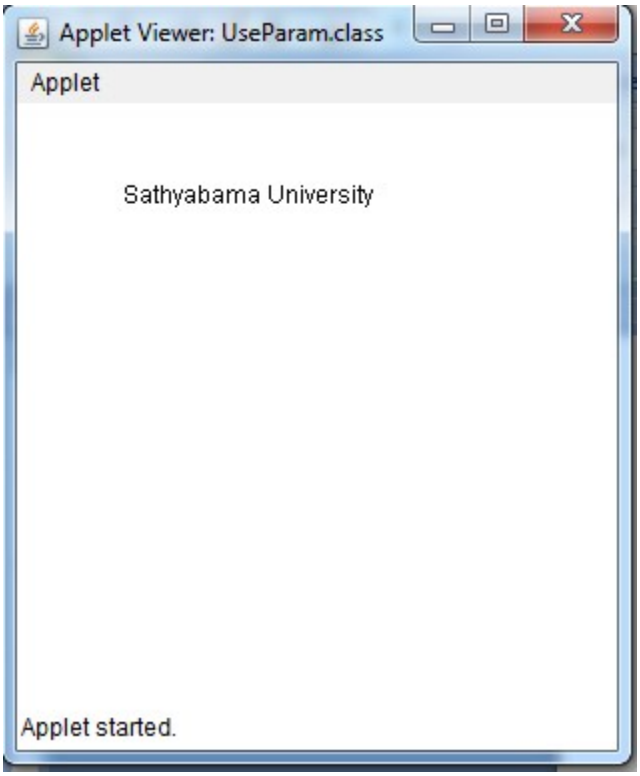

Fig. 5.5. Output of param.java

APPLET ARCHITECTURE

Use Parameters
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>

```
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
public class ParamDemo extends Applet{
String fontName;
int fontSize;
float leading;
boolean active;
// Initialize the string to be displayed.
public void start() {
String param;
fontName = getParameter("fontName");
if(fontName == null)
fontName = "Not Found";
param = getParameter("fontSize");
try {
if(param != null) // if not found
fontSize = Integer.parseInt(param);
else
fontSize = 0;
} catch(NumberFormatException e) {
fontSize = -1;
}
param = getParameter("leading");
try {

if(param != null) // if not found
leading = Float.valueOf(param).floatValue();
else
leading = 0;
} catch(NumberFormatException e) {
leading = -1;
}
param = getParameter("accountEnabled");
if(param != null)
active = Boolean.valueOf(param).booleanValue();
}
// Display parameters.
public void paint(Graphics g) {
```

```
g.drawString("Font name: " + fontName, 0, 10);
g.drawString("Font size: " + fontSize, 0, 26);
g.drawString("Leading: " + leading, 0, 42);
g.drawString("Account Active: " + active, 0, 58);
}
}
```

E:\basic pgms\unit iv>javac ParamDemo.java

E:\basic pgms\unit iv>appletviewer ParamDemo.java

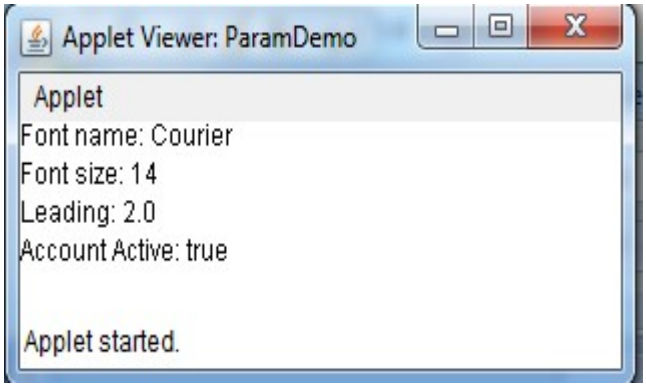

Fig. 5.6 Output of paramdemo.java

AWT

- ✓ AWT ('Abstract window Toolkit') is an API to develop GUI or window-based applications in java.
- ✓ Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system.
- ✓ The java.awt package provides classes for AWT api such as TextField, Label,TextArea, RadioButton, CheckBox, Choice, List etc.

CONTAINER:

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The containers are Frame, Dialog and Panel.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

METHODS OF COMPONENT CLASS

Method	Description

public void add(Component c)	**inserts a component on this component.**
public void setSize(int width,int height)	**sets the size (width and height) of the component.**
public void setLayout(LayoutManager m)	**defines the layout manager for the component.**
public void setVisible(boolean status)	**changes the visibility of the component, by default false.**

Tabel 5.1. Methods of Comoponent class

EX:

```
import java.awt.*;
import java.awt.event.*;
class framedemo
{
public static void main(String[] args)
{
Frame f=new Frame("my first frame");
f.setSize(500,500);
f.setLayout(null);
f.setVisible(true);
f.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
});
}
}
```

```
E:\javapgms>javac framedemo.java

E:\javapgms>java framedemo
```

Output

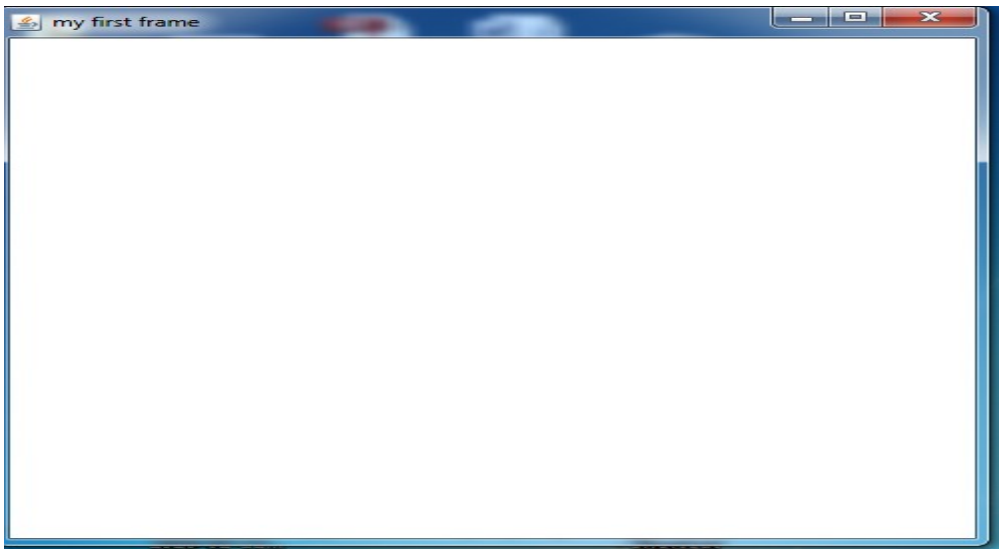

Fig. 5.7. Output of framedemo.java

LABEL

- ✓ 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( ):

Ex:

```
Label l1=new  Label();
```

Label(String str)

Ex:

```
Label l1=new  Label("Name");
```

Label(String str, int how)

Ex:

```
Label   l1=new    Label("Name",Label.LEFT or Label.RIGHT  or Label.CENTER
```

Methods

Void setText(String str);

String getText();

void setAlignment(int how)

int getAlignment( )

It has no Listener Class.

Ex:

```
import java.awt.*;
import java.awt.event.*;
class label
{
public static void  main(String[] args)
{
Frame f=new Frame("first");
Label l1=new Label("Name");
Label l2=new Label("age");
Label l3=new Label("gender");
l1.setBounds(100,50,70,30);
l2.setBounds(100,100,70,30);
l3.setBounds(100,120,70,30);
f.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}});
f.add(l1);
f.add(l2);
f.add(l3);
f.setSize(300,300);
f.setVisible(true);
f.setLayout(null);
}
}
```

```
E:\javapgms>javac label.java

E:\javapgms>java label
```

Output

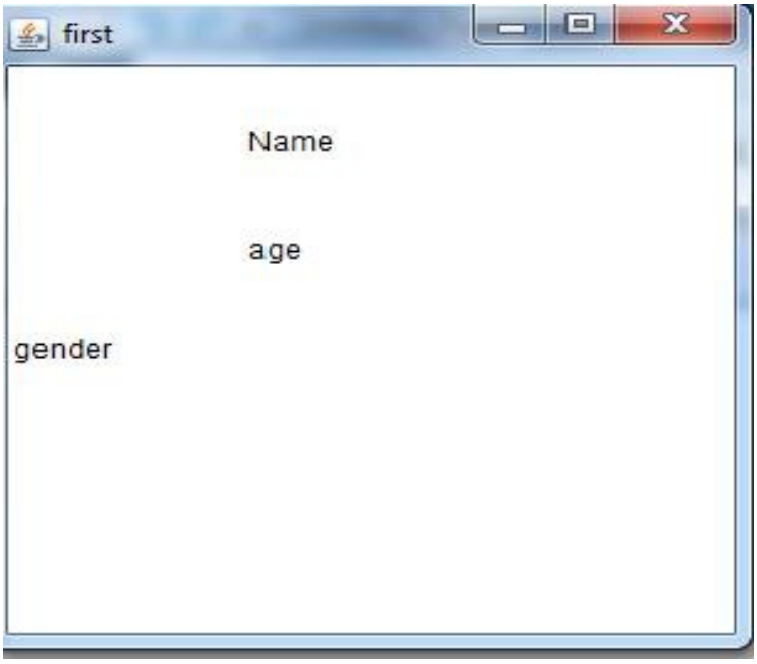

Fig. 5.8. Output of label.java

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:
The first version creates a default text field.
Textfield():
Ex:

```
TextField tf=new TextField( );
```

TextField(int numchars)

```
TextField  tf=new  TextField(10)
```

TextField(String str)

```
TextField  tf= new TextField("hai");
```

TextField(String str, int numChars)

```
TextField tf=new TextField("hai", 3);
```

METHODS

String getText();
Void setText("hello")
setBackground(Color.red)
getBackground()
getSelectedText()

EVENT

ActionListener()

METHOD

Void actionPerformed(ActionEvent e)
{
}

Ex:

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class textfield implements ActionListener
{
Frame f=new Frame();
Label l1,l2;
TextField t1,t2,t3;
textfield()
{
l1=new Label("Name :",Label.LEFT);
l2=new Label("Password:",Label.LEFT);
t1=new TextField(15);
t2=new TextField(15);
t2.setEchoChar('?');
t3=new TextField(40);
l1.setBounds(100,100,100,20);
l2.setBounds(100,200,100,20);
t1.setBounds(250,100,150,20);
t2.setBounds(250,200,150,20);
t3.setBounds(50,300,400,20);
f.add(l1);
```

```
f.add(l2);
f.add(t1);
f.add(t2);
f.add(t3);
t1.addActionListener(this);
t2.addActionListener(this);
f.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
});
f.setSize(450,450);
f.setLayout(null);
f.setVisible(true);
}
public void actionPerformed(ActionEvent  e)
{

t3.setText(t1.getText()+"  "+t2.getText()+"  "+"selected text is"+t1.getSelectedText());
}
public static void main(String[] arg)
{
new textfield();
}
}
```

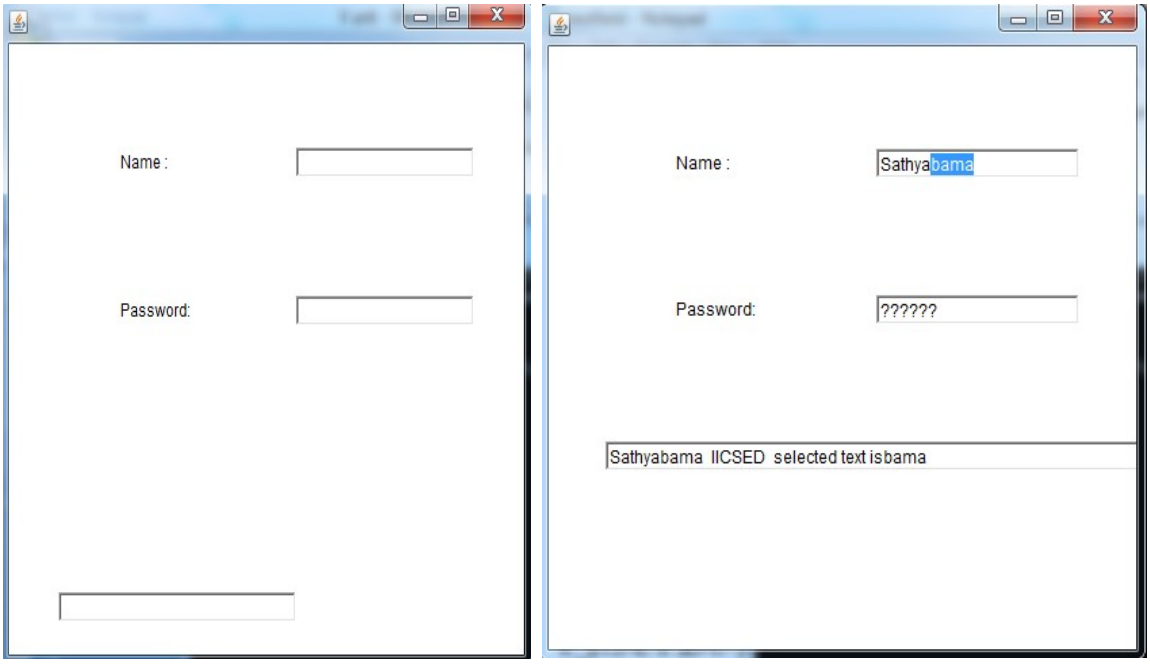

Fig. 5.9. Output of textfield.java

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  b= new Button();
```

It creates an empty button.

Button(String str)

```
Button  b= new  Button("click");
```

It creates a button that contains str as a label.

METHODS

setLabel():After a button has been created, you can set its label by calling setLabel(**).**

```
b.setLabel("click");
```

getLabel( ):

You can retrieve its label by calling getLabel( ).

```
String  str="";
str = b.getLabel();
```

Interface
ActionListener

Method
Object.addActionListener(this);
Public void actionPerformed(ActionEvent e)
{
}

Ex:
Simple Button

```
import java.awt.*;
import java.awt.event.*;
public class firstbutton
{
public static void main(String[] args)
{
Frame f=new Frame("buttonexample");
final TextField tf=new TextField();
tf.setBounds(50,50,150,200);
Button b=new Button("yes");
b.setBounds(50,100,60,30);
b.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent e)
{
tf.setText("clicked yes");
}
});
f.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
});
f.add(b);
f.add(tf);
```

```
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);
}
}
```

E:\javapgms>javac firstbutton.java

E:\javapgms>java firstbutton

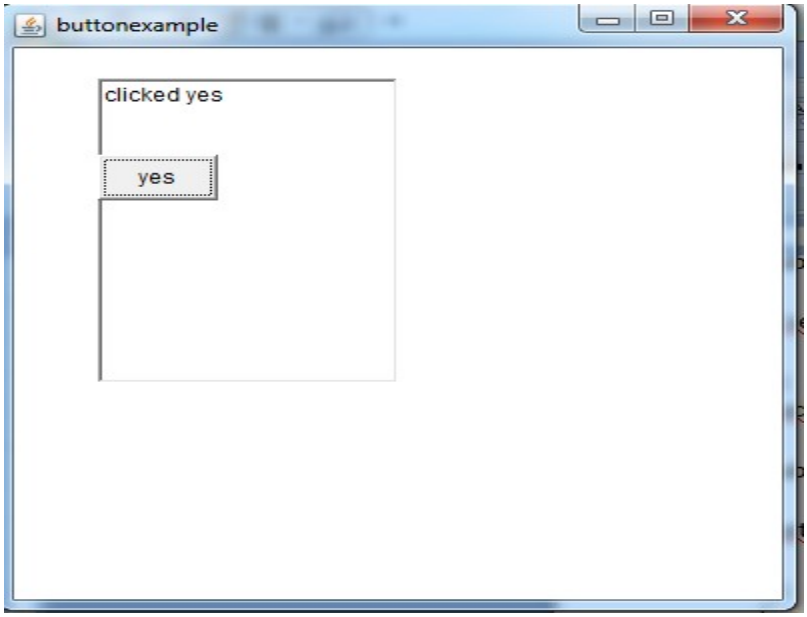

Fig. 5.10. Output of firstbutton.java

Button ArrayEx:

```
import java.awt.*;
import java.awt.event.*;
public class buttonlist  implements ActionListener
{
String s="";
Button y
public void init()
{
Button yes=new Button("yes");
Button no=new Button("no");
Button maybe=new Button("decided");
b[0]=(Button) add(yes);
b[1]=(Button) add(no);
```

```
b[2]=(Button) add(maybe);
for(int i=0;i<3;i++)
{
b[i].addActionListener(this);
}
}
public void actionPerformed(ActionEvent e)
{
for( int i=0;i<3;i++)
{
if(e.getSource()==b[i])
{
s="you pressed"+b[i].getLabel();
}
}
repaint();
}
public void paint(Graphics g)
{
g.drawString(s,6,100);
}
}
```

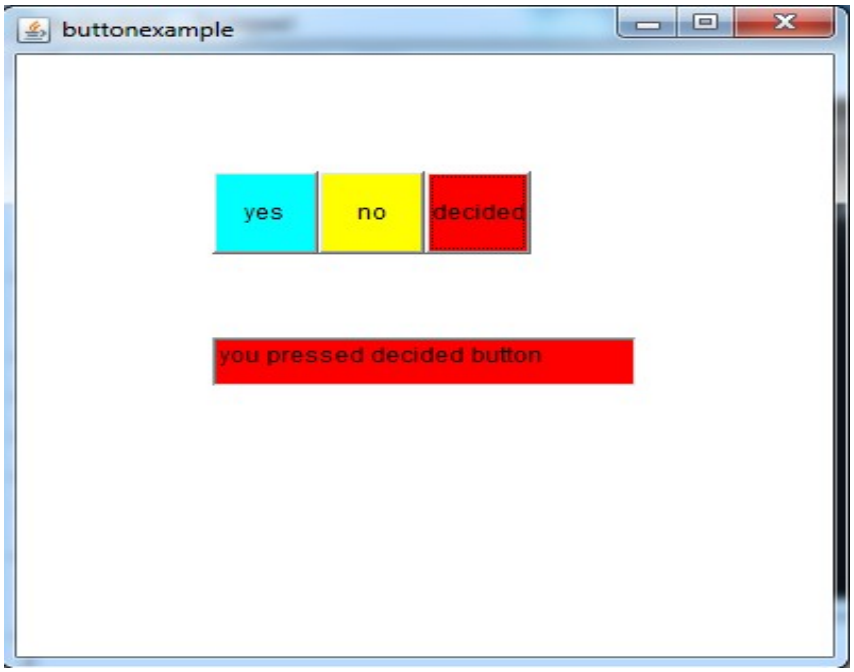

Fig. 5.11. Output of Buttonexample.java

Checkbox

- ✓ **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.
- ✓ Checkboxes are objects of the Checkbox class.

Checkbox supports these constructors:

Checkbox( ):

```
Checkbox ch= new Checkbox();
```

Checkbox(String str):

```
Checkbox ch= new Checkbox("win98");
```

Checkbox(String str, boolean on)

```
Checkbox ch= new Checkbox("win98",true);
```

METHODS

boolean getState( )
void setState(boolean on)
String getLabel( )
void setLabel(String str)
getSource()

INTERFACE

ItemListener

METHOD

addItemListener(this);

public void itemStateChanged(ItemEvent e)
{
}

Ex:

import java.awt.*;

```
import java.awt.event.*;
public class checkbox implements ItemListener
{
Frame f;
Checkbox Win98, winNT, solaris, mac;
TextField tf;
checkbox()
{
f=new  Frame("checkbox");
tf=new TextField();
tf.setBounds(100,300,200,20);
Win98 = new Checkbox("win98");
Win98.setBounds(100,100,70,30);
winNT = new Checkbox("winNT");
winNT.setBounds(100,150,70,30);
solaris = new Checkbox("Solaris");
solaris.setBounds(100,200,70,30);
mac = new Checkbox("Mac");
mac.setBounds(100,250,70,30);
f.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
});
Win98.addItemListener(this);
winNT.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
f.add(Win98);
f.add(winNT);
f.add(solaris);
f.add(mac);
f.add(tf);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);
}

public void itemStateChanged(ItemEvent e)
{
```

```
if(e.getSource()==Win98)
{
tf.setText("you pressed"+" "+Win98.getLabel()+" "+Win98.getState());
}
if(e.getSource()==winNT)
{
tf.setText(winNT.getLabel()+" "+winNT.getState());
}
if(e.getSource()==solaris)
{
tf.setText(solaris.getLabel()+" "+solaris.getState());
}
if(e.getSource()==mac)
{
tf.setText(mac.getLabel()+" "+mac.getState());
}
}
public static void main(String[]  args)
{
new checkbox();
}
}
```

```
E:\javapgms>javac checkbox.java

E:\javapgms>java checkbox
```

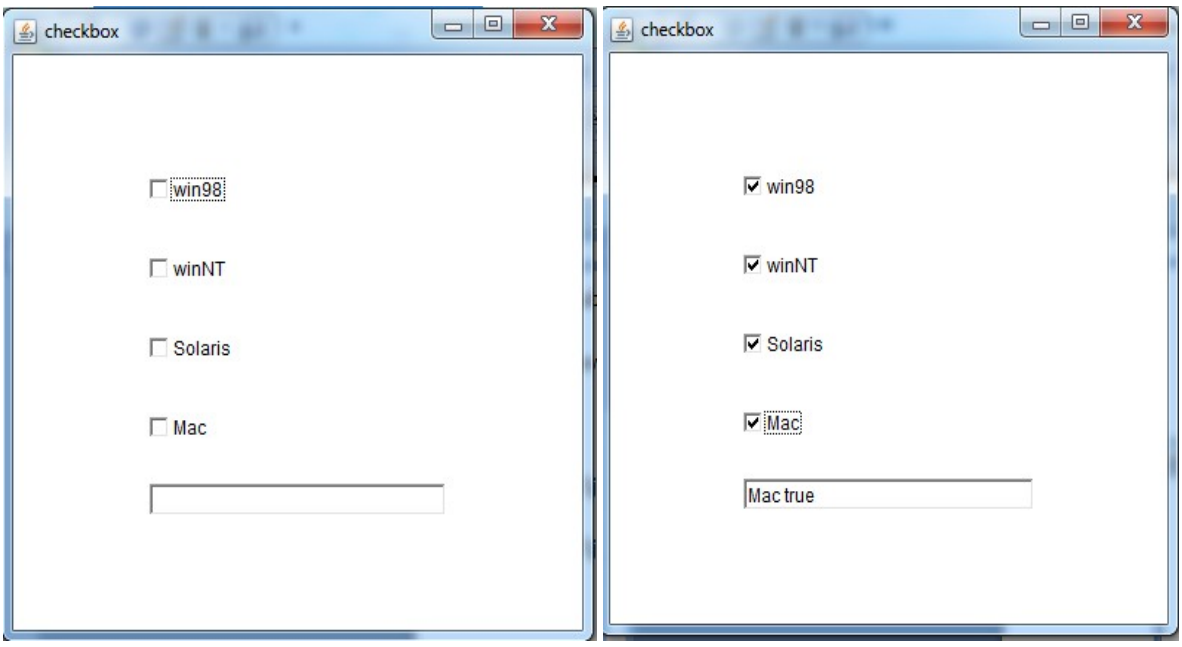

Fig. 5.12. Output of textfield.java

CHECKBOX GROUP

- ✓ 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

```
CheckboxGroup gp=new CheckboxGroup();
Checkbox ch=  new  Checkbox("win98",true,gp)
```

```
CheckboxGroup gp=new CheckboxGroup();
Checkbox ch=  new  Checkbox("win98",gp,true)
```

Ex:

```
import java.awt.*;
import java.awt.event.*;
public class checkboxgroup
{
public static void main(String[] args)
{
Frame f=new Frame("checkbox");
boolean b;
CheckboxGroup gp=new CheckboxGroup();
Checkbox Win98, winNT, solaris, mac;
Win98 = new Checkbox("Windows 98/XP", gp, true);
```

```
Win98.setBounds(100,100,50,50);
winNT = new Checkbox("Windows NT/2000",gp,false);
winNT.setBounds(100,150,50,50);
solaris = new Checkbox("Solaris",gp,true);
solaris.setBounds(100,200,50,50);
mac = new Checkbox("MacOS",gp,false);
mac.setBounds(100,250,50,50);
/*Win98.addItemListener(new ItemListener()
{
public void  itemStateChanged(ItemEvent e)
{
b=Win98.getState();

}
});
winNT.addItemListener(new ItemListener()
{
public void  itemStateChanged(ItemEvent e)
{
str=winNT.getState();
tf.setText(str);
}
});
solaris.addItemListener(new ItemListener()
{
public void  itemStateChanged(ItemEvent e)
{
tf.setText(solaris.getState());
}
});
mac.addItemListener(new ItemListener()
{
public void  itemStateChanged(ItemEvent e)
{
tf.setText(mac.getState());
}
});*/
f.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
```

```
}
});
f.add(Win98);
f.add(winNT);
f.add(solaris);
f.add(mac);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);
}
}
```

E:\javapgms>javac checkboxgroup.java

E:\javapgms>java checkboxgroup

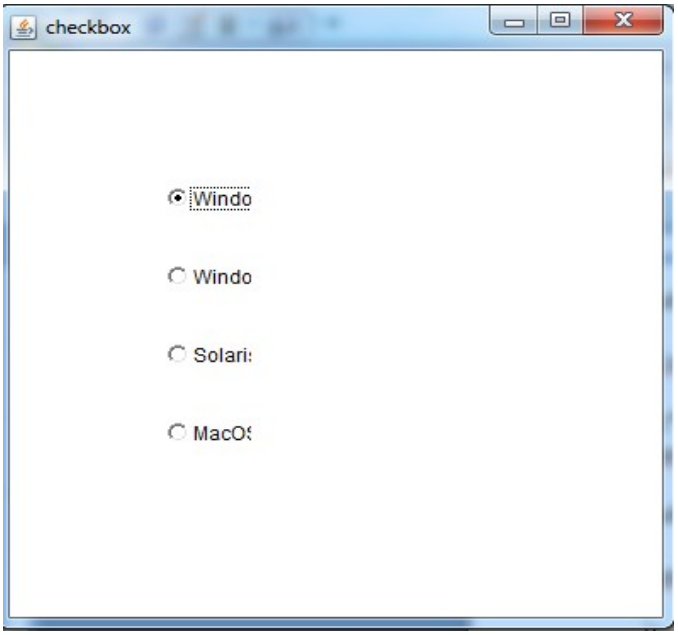

Fig.5.13. Output of checkboxgroup.java

LIST

- ✓ The List class provides a compact, multiple-choice, scrolling selection list.
- ✓ 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( )

The first version creates a List control that allows only one item to be selected at any one time

List(int numRows)

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).

List(int numRows, boolean multipleSelect)

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.

Ex:

```
import java.awt.*;
import java.awt.event.*;
public class list
{
public static void main(String[] args)
{
Frame f=new Frame();
final Label l=new Label();
l.setAlignment(Label.CENTER);
l.setSize(500,100);
Button b=new Button("show");
b.setBounds(200,150,80,30);
final List os=new List(4,false);
os.setBounds(100,100,70,70);
os.add("windows");
os.add("linux");
os.add("unix");
os.add("android");
b.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent e)
{
String data="os selected"+os.getItem(os.getSelectedIndex());
l.setText(data);
}});
f.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
```

```
});
f.add(os);
f.add(l);
f.add(b);
f.setSize(450,450);
//f.setLayout(null);
f.setVisible(true);
}
}
```

E:\javapgms>javac list.java

E:\javapgms>java list

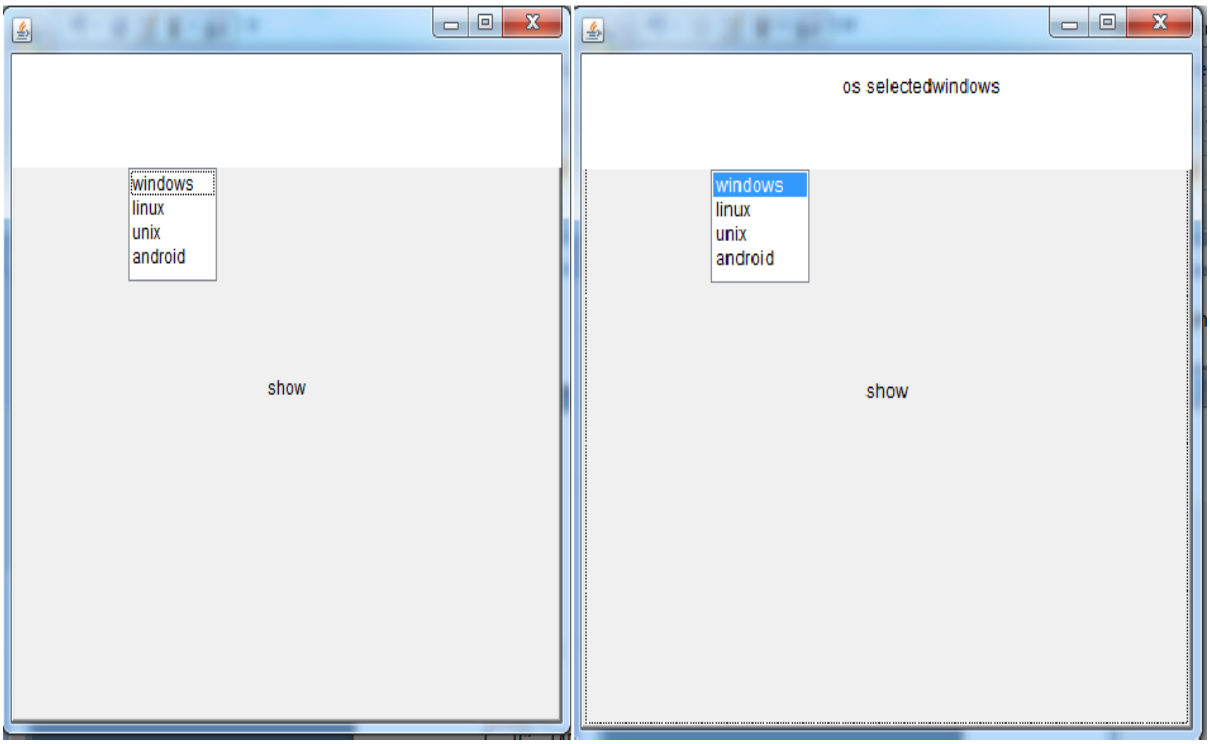

Fig 5.14 Output of List.java

TEXT AREA

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)

METHODS

getText( )
setText( )
getSelectedText( )
select( ),
isEditable( )
setEditable( )

Ex:

```
import java.awt.*;
import java.awt.event.*;
public class TextAreaDemo  {
public static void main(String[] args)
{
Frame f=new Frame();
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();
text.setBounds(100,100,200,200);
text.setText(val);

f.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
});
f.add(text);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);
```

```
}
}
```

E:\javapgms>javac TextAreaDemo.java

E:\javapgms>java TextAreaDemo

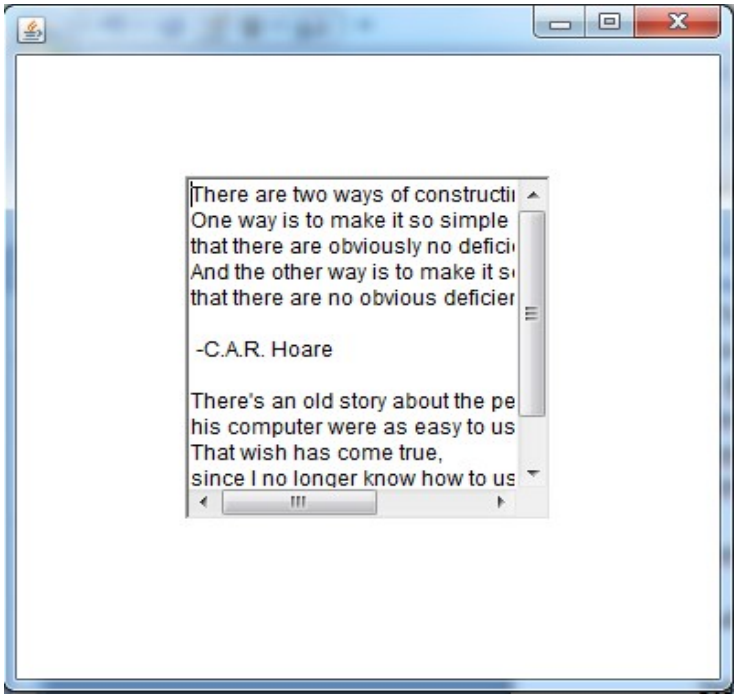

Fig. 5.14 Output of TextAreaDemo.java

Scrollbar

- ✓ *Scroll bars* are used to select continuous values between a specified minimum and
- ✓ maximum.
- ✓ Scroll bars may be oriented horizontally or vertically.
- ✓ A scroll bar is actually a composite of several individual parts.
- ✓ Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow.
- ✓ The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar.

Scrollbar defines the following constructors:

Scrollbar( )

It creates a vertical scroll bar.

Scrollbar(int *style*)

It allows you to specify the orientation of the scroll bar.

Scrollbar(int *style,* int *initialValue*, int *thumbSize*, int *min*, int *max*)

If *style* is Scrollbar.VERTICAL, a vertical scroll bar is created. If *style* is Scrollbar.HORIZONTAL, the scroll bar is horizonta**l**

Methods

int getValue( )
void setValue(int *newValue)*
int getMinimum( )

int getMaximum( )

void setUnitIncrement(int *newIncr*)

void setBlockIncrement(int *newIncr*)

Interface

AdjustmentListener interface

Object

AdjustmentEvent

Event Method
public void **getAdjustmentType( )**
{}

The types of adjustment events are as follows:
BLOCK_DECREMENT -A page-down event has been generated.
BLOCK_INCREMENT -A page-up event has been generated.
TRACK -An absolute tracking event has been generated.
UNIT_DECREMENT -The line-down button in a scroll bar has been pressed.
UNIT_INCREMENT -The line-up button in a scroll bar has been pressed.

Ex:

```
import java.awt.*;
import java.awt.event.*;
public class SBDemo implements AdjustmentListener
{
String msg = "";
Frame f;
Scrollbar vertSB, horzSB;
Label l;
SBDemo()
{
f=new Frame();
l=new Label();
l1=new Label();
l2=new Label();
vertSB = new Scrollbar(Scrollbar.VERTICAL,0, 1, 0, 30);
horzSB = new Scrollbar(Scrollbar.HORIZONTAL,0, 1, 0, 30);
vertSB.setBounds(100,100,50,100);
horzSB.setBounds(200,100,200,50);
l.setSize(200,200);
vertSB.addAdjustmentListener(this);
horzSB.addAdjustmentListener(this);
f.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
});
f.add(vertSB);
f.add(horzSB);
f.add(l);
f.setSize(450,450);
f.setVisible(true);
}
public void adjustmentValueChanged(AdjustmentEvent e)
{
if(e.getSource()==vertSB)

l.setText("Vertical Scrollbar value is:"+e .getValue());
else
l.setText("Horizontal Scrollbar value is:"+e .getValue());
}
```

```
        public static void main(String[] args)
        {
        new SBDemo();
        }
}
```

E:\javapgms>javac SBDemo.java

E:\javapgms>java SBDemo

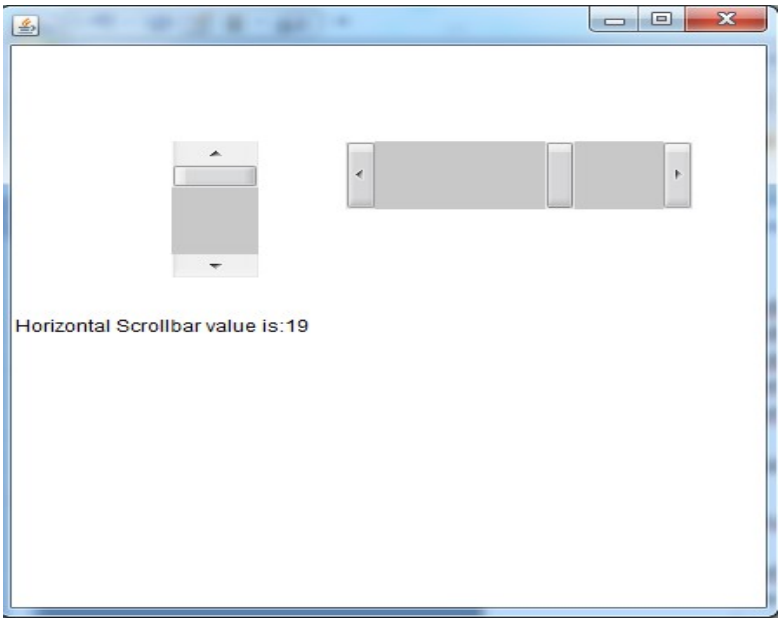

Fig 5.15 Output of SBDemo.java

LAYOUT MANAGERS

- ✓ A layout manager is an instance of any class that implements the **LayoutManager** interface.
- ✓ The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The **setLayout( )** method has the following general form:

void setLayout(LayoutManager *layoutObj*)

Types:

1.FlowLayout
2.GridLayout

3.BorderLayout

FlowLayout

- ✓ FlowLayout is the default layout manager.
- ✓ It implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom.
- ✓ When no more components fit on a line, the next one appears on the next line.
- ✓ A small space is left between each component, above and below, as well as left and right.

Constructors for FlowLayout:

FlowLayout( )

The first form creates the default layout, which centers components and leaves five pixels of space between each component.

FlowLayout(int *how*)

The second form lets you specify how each line is aligned. Valid values for *how* are as follows:

FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT

These values specify left, center, and right alignment, respectively.

FlowLayout(int *how*, int *horz*, int *vert*)

The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert,* respectively.

Ex:

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class flowlayout implements ActionListener
{
Frame f=new Frame();
Label l1,l2,l3;
TextField t1,t2,t3;
Button Add=new Button("add");
flowlayout()
{
l1=new Label("ist value");
l2=new Label("2nd value");
l3=new Label("result");
t1=new TextField(15);
t2=new TextField(15);
```

```
t3=new TextField();
l1.setBounds(100,100,70,70);
l2.setBounds(100,150,70,70);
l3.setBounds(100,200,70,70);
t1.setBounds(250,100,70,70);
t2.setBounds(250,200,70,70);
t3.setBounds(250,300,70,70);
Add.setBounds(200,400,70,70);
f.setLayout(new FlowLayout(FlowLayout.LEFT));
f.add(l1);
f.add(l2);
f.add(l3);
f.add(t1);
f.add(t2);
f.add(t3);
f.add(Add);
Add.addActionListener(this);
/*Add.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent  e)
{
int i=Integer.parseInt(t1.getText());
int j=Integer.parseInt(t2.getText());
int c=0;
if(e.getSource()==Add)
{
c=i+j;
}
t3.setText(String.valueOf(c));
}
});*/
f.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
});
f.setSize(450,450);
f.setLayout(null);
f.setVisible(true);
}
```

```
        public void actionPerformed(ActionEvent  e)
        {
        String s1=t1.getText();
        String s2=t2.getText();
        int i=Integer.parseInt(s1);
        int j=Integer.parseInt(s2);
        int c=0;
        if(e.getSource()==Add)
        {
        c=i+j;
        }
        String re=String.valueOf(c);
        t3.setText(re);
        }

        public static void main(String[] arg)
        {
        new flowlayout();
        }
}
```

```
E:\javapgms>javac flowlayout.java

E:\javapgms>java flowlayout
```

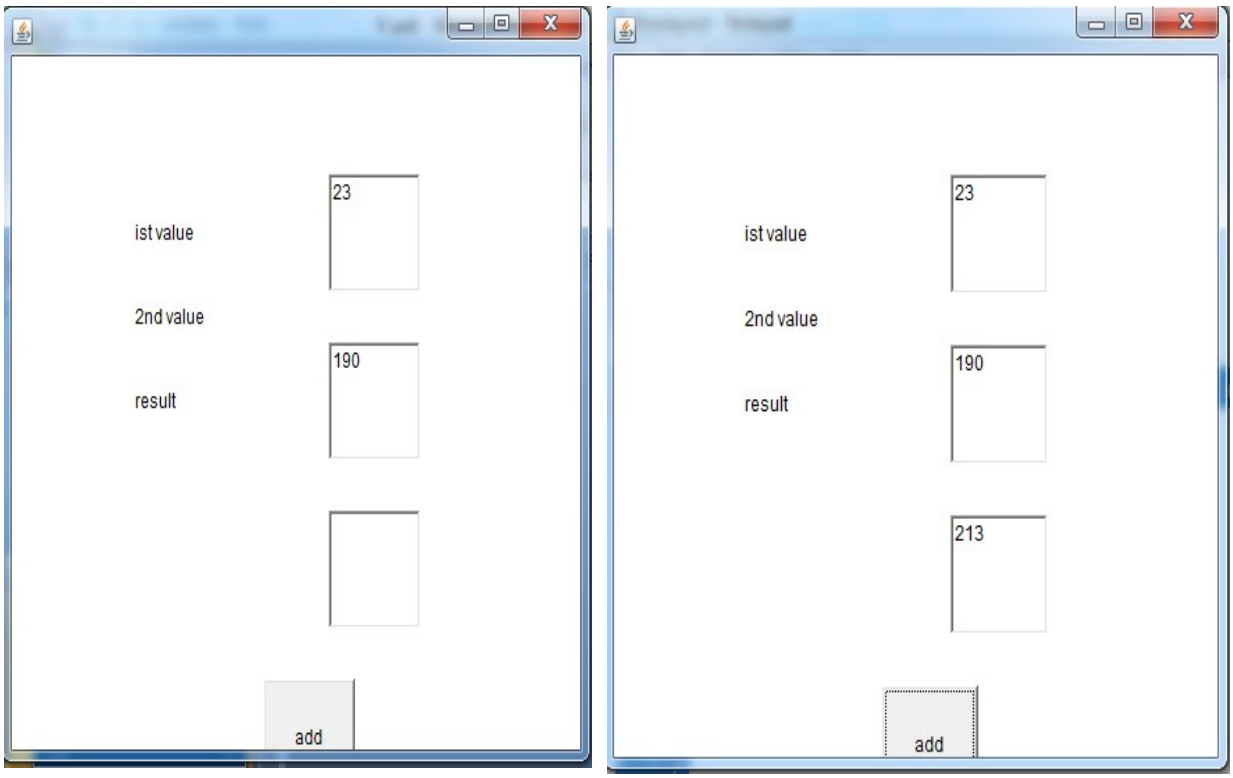

Fig. 5.16. Output of Flowlayout.java

BorderLayout

- ✓ The **BorderLayout** class implements a common layout style for top-level windows.
- ✓ It has four narrow, fixed-width components at the edges and one large area in the center.
- ✓ The four sides are referred to as north, south, east, and west. The middle area is called the center.

Constructors for BorderLayout:

BorderLayout( )

It creates a default border layout.

BorderLayout(int *horz*, int *vert*)

It allows you to specify the horizontal and vertical space left between components in horz and verz

BorderLayout defines the following constants that specify the regions:

BorderLayout.CENTER BorderLayout.SOUTH

BorderLayout.EAST BorderLayout.WEST

BorderLayout.NORTH

add():

It is used to add the components which is defined by **Container**:

void add(Component *compObj,* Object *region*);

Here, *compObj* is the component to be added, and *region* specifies where the component will be added.

Ex:

```
import java.awt.*;
import java.awt.event.*;
public class borderlayout {
Frame f;
String msg="";
borderlayout()
{
 f=new Frame("BorderLayout");
f.setLayout(new BorderLayout());
Button b1=new Button("north");
Button b2=new Button("south");
Button b3=new Button("east");
Button b4=new Button("west");
f.add(b1,BorderLayout.SOUTH);
f.add(b2,BorderLayout.NORTH);
f.add(b3,BorderLayout.EAST);
f.add(b4,BorderLayout.WEST);
msg = "The reasonable man adapts " +
"himself to the world;\n" +
"the unreasonable one persists in " +
"trying to adapt the world to himself.\n" +
"Therefore all progress depends " +
"on the unreasonable man.\n\n" +
" – George Bernard Shaw\n\n";
f.add(new TextArea(msg), BorderLayout.CENTER);
f.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
});
f.setSize(200,200);
f.setVisible(true);
}
public static void main(String[] args)
{
new borderlayout();
}
```

}

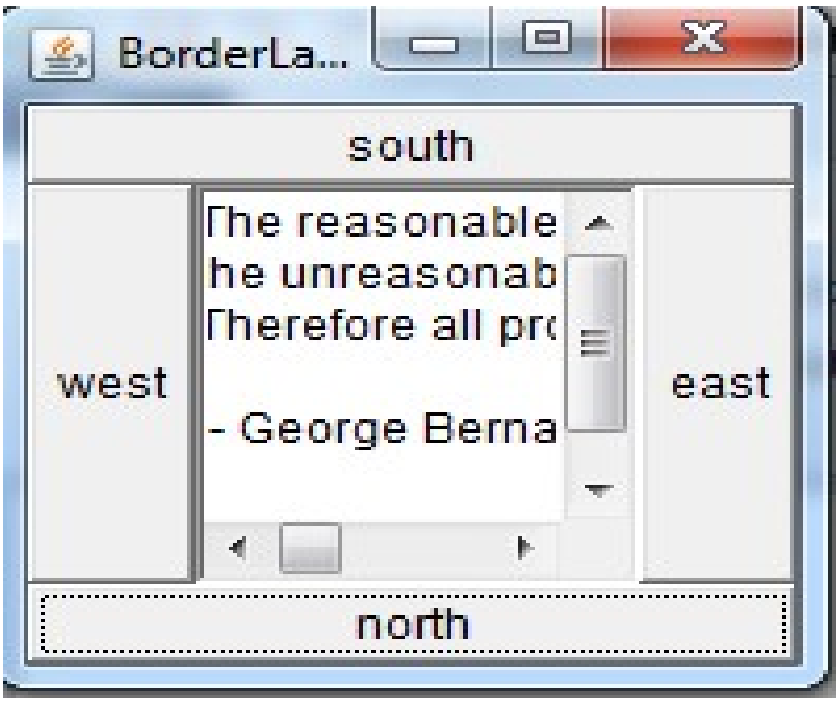

Fig. 5.17. Output of BorderLayout.java

GridLayout

- ✓ **GridLayout** lays out components in a two-dimensional grid.
- ✓ When you instantiate a **GridLayout**, you define the number of rows and columns.

Constructors for GridLayout

GridLayout( )

The first form creates a single-column grid layout

GridLayout(int *numRows*, int *numColumns*)

The second form creates a grid layout with the specified number of rows and columns.

GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

It specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length rows.

Ex:

```
import java.awt.*;
import java.awt.event.*;
public class gridlayout {
Frame f;
final int n = 3;
TextField tf;
gridlayout()
{
f=new Frame("");
f.setLayout(new BorderLayout());
```

```
        f.setLayout(new GridLayout(n, n));
        tf=new TextField();
        f.setFont(new Font("SansSerif", Font.BOLD, 24));
        for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
        int k = i * n + j;
        if(k >=0)
        f.add(new Button("" + k));
        }
        }
        f.add(new Button("+"));
        f.add(new Button("-"));
        f.add(new Button("*"));
        f.add(new Button("/"));
        f.add(new Button("="));
        f.add(tf,BorderLayout.NORTH);
        f.addWindowListener(new WindowAdapter()
        {
        public void windowClosing(WindowEvent e)
        {
        System.exit(0);
        }
        });
        f.setSize(200,300);

        f.setVisible(true);
        }
        public static void main(String[] args)
        {
        new gridlayout();
        }
}
```

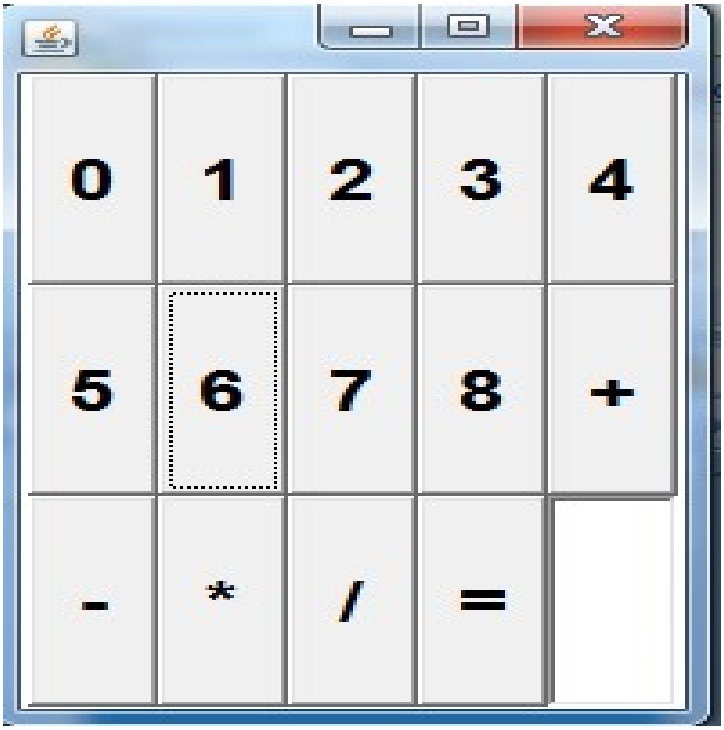

Fig. 5.18 Output of gridlayout.java

MENU BARS AND MENUS

A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu.

Menu Classes

To create a menu bar, first create an instance of **MenuBar**. This class only defines the default constructor. Next, create instances of **Menu** that will define the selections displayed on the bar.

1. MenuBar
2. Menu
3. MenuItem

Constructors for MenuItem:

MenuItem( )
MenuItem(String *itemName*)
MenuItem(String *itemName*, MenuShortcut *keyAccel*)

Constructors for Menu:

Menu( )
Menu(String *optionName*)
Menu(String *optionName*, boolean *removable*)

Add Menuitem to Menu

Once you have created a menu item, you must add the item to a **Menu** object by using **add( )**.The general form is

Menuobject.add(MenuItemobject)

Here, *item* is the item being added. Items are added to a menu in the order in which the calls to **add( )** take place. The *item* is returned.

Add Menu to Menubar

Once you have added all items to a **Menu** object, you can add that object to the menu bar by using this version of **add( )** defined by **MenuBar**:

Menubarobject. add(Menuobject)

Here, *menu* is the menu being added. The *menu* is returned.

Ex:

```
import java.awt.*;
import java.awt.event.*;
public class menudesign implements ActionListener
{
Frame f;
TextArea te;
MenuBar mb;
MenuItem New,Save,Cut,Copy,Paste,Close;
menudesign()
{
f=new Frame();
te=new TextArea();

New=new MenuItem("New");
Save=new MenuItem("Save");
Cut=new MenuItem("Cut");
Copy=new MenuItem("Copy");
Paste=new MenuItem("Paste");
Close=new MenuItem("Close");

//New.addActionListener(this);
//Save.addActionLitener(this);
Cut.addActionListener(this);
Copy.addActionListener(this);
Paste.addActionListener(this);
Close.addActionListener(this);

mb=new MenuBar();
```

```
Menu fi=new Menu("File");
Menu ed=new Menu("Edit");
Menu fo=new Menu("Format");
Menu ex=new Menu("exit");

fi.add(New);
fi.add(Save);
ed.add(Cut);
ed.add(Copy);
ed.add(Paste);
ex.add(Close);

te.setBounds(50,50,100,200);
//ta.setBackground(Color.cyan);

mb.add(fi);
mb.add(ed);
mb.add(fo);
mb.add(ex);
f.setMenuBar(mb);
f.add(te);
f.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
});
f.setSize(200,300);
f.setVisible(true);
}
public void actionPerformed(ActionEvent e)
{
String s=(String)e.getActionCommand();
if(s.equals("Cut"))
//te.Cut();
te.setText(s);
if(s.equals("Paste"))
//te.Paste();
te.setText(s);
if(s.equals("Copy"))
//te.Copy();
```

```
te.setText(s);
if(s.equals("Close"))
System.exit(0);
}
public static void main(String[] args)
{
new menudesign();
}
}
```

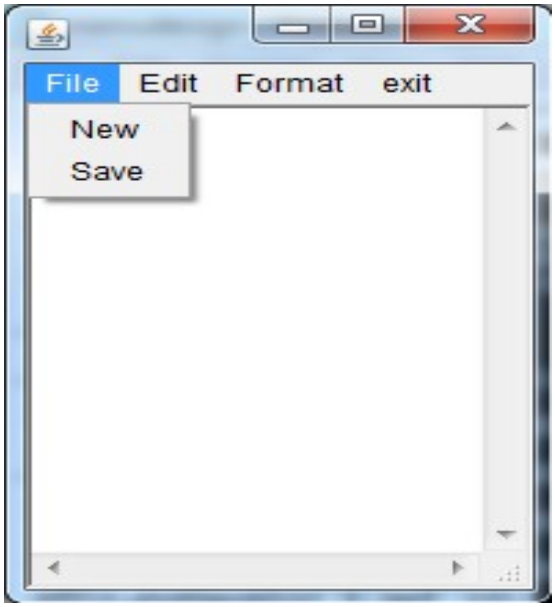

Fig.5.19 Output of Menudesign.java

DATABASE CONNECTIVITY

• ODBC driver –

- ✓ It require a driver to be loaded at runtime to connect to any data source.
- ✓ The driver is implemented as a class that is located and loaded at runtime.
- ✓ The ODBC driver for JDBC connections is named sun.java.odbc.JdbcOdbcDriver.
- ✓ It require an ODBC connection string to connect to the data source.
- ✓ To connect with an ODBC DSN, we require a connection string "jdbc:odbc:*ODBC DSN String*"
- ✓ The package containing the database related classes is contained in java.sql.
- ✓ Dynamically load the class sun.java.odbc.JdbcOdbcDriver as Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
- ✓ Connection conn = DriverManager.getConnection(database, "", "");
- ✓ The Statement must be created to execute a SQL query on the opened database.
- ✓ Statement s = conn.createStatement();
- ✓ To clean up after we are done with the SQL query, • To call s.close() to dispose the object Statement.

• **To call conn.close() for close the database**

Step 1 : Open Microsoft Access and select Student data base option and give the data base name as File name option as “student”
Step 2 : Create a table and insert your data into the table
Step 3 : Save the table with the desired name; in this article we save the following records with the table name student.

• ODBC connection string
• Import the classes to connect to the database
• Load the JDBC:ODBC driver
• Open the MS-Access database file in the application space
• Create a Statement object to execute the SQL query
• Cleanup after finishing the job
• Creating a Database

37120001 Aravind
37120002 Arjun
• Now Creating DSN of your data base –
Step 4 : Open your Control Panel and than select Administrative Tools.
Step 5 : Click on Data Source(ODBC)-->User DSN.
Step 6 : Now click on add option for making a new DSN.select Microsoft Access Driver (*.mdb. *.accdb) and than click on Finish
Step 7 : Make your desired Data Source Name and then click on the Select option.
Step 8 : Now you select your data source file for storing it and then click ok and then click on Create and Finish

Example

```
import java.sql.*;
class DatabaseDemo
{
public static void main(String ar[])
{
Try
{
String url="jdbc:odbc:veeradsn";
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection(url);
Statement st=c.createStatement();
st.executeUpdate("INSERT into std(Reg_no,Name) values("+37120003+",'Avinash'"+")");
```

```
ResultSet rs=st.executeQuery("select * from std");
while(rs.next())
System.out.println(rs.getString(1)+ " "+rs.getString("Name"));
}
catch(Exception ee)
{
System.out.println(ee);
}
}
}
```