MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution

DEPARTMENT OF COMPUTER APPLICATION

SUBJECT NAME: PROGRAMMING IN JAVA

SUBJECT CODE: SAZ4A

PREPARED BY: PROF. S.ANITHA

Title of the Course/	**Paper-IX**		**PROGRAMMING IN JAVA**
Core	**II Year & Fourth Semester**	Credit: 4	
Objective of the course	This course introduces the basic concepts of programming in JAVA		
Course outline	**Unit-1**: Introduction to Java-Features of Java-Basic Concepts of Object Oriented Programming-Java Tokens-Java Statements-Constants-Variables-Data Types- Type Casting-Operators-Expressions-Control Statements: Branching and Looping Statements.		
	Unit-2:Classes, Objects and Methods - Constructors - Methods Overloading-Inheritance-Overriding Methods-Finalizer and Abstract Methods-Visibility Control –Arrays, Strings and Vectors-StringBuffer Class-Wrapper Classes		
	Unit-3: Interfaces-Packages-Creating Packages-Accessing a Package-Multithreaded Programming-Creating Threads-Stopping and Blocking a Thread-Life Cycle of a Thread-Using Thread Methods-Thread Priority-Synchronization-Implementing the Runnable Interface		
	Unit-4: Managing Errors and Exceptions-Syntax of Exception Handling Code-Using Finally Statement-Throwing Our Own Exceptions-Applet Programming-Applet Life Cycle-Graphics Programming-Managing Input/Output Files: Concept of Streams-Stream Classes-Byte Stream Classes-Character Stream Classes – Using Streams-Using the File Class-Creation of Files-Random Access Files-Other Stream Classes.		
	Unit-5 : Network basics –socket programming – proxy servers – TCP/IP – Net Address – URL – Datagrams -Java Utility Classes-Introducing the AWT: Working with Windows, Graphics and Text- AWT Classes-Working with Frames-Working with Graphics-Working with Color-Working with Fonts-Using AWT Controls, Layout Managers and Menus.		

Unit-1: Introduction to Java-Features of Java-Basic Concepts of Object Oriented Programming-Java Tokens-Java Statements-Constants-Variables-Data Types- Type Casting-Operators-Expressions-Control Statements: Branching and Looping Statements.

Java Features

1. Compiled and Interpreted
2. Platform Independent and Portable
3. Object-Oriented
4. Robust
5. Secure
6. portable
7. Distributed
8. Architecture-Neutral
9. Familiar, Simple and Small
10. Multithreaded and Interactive
11. High Performance
12. Dynamic and Extensible

1. Compiled and Interpreted

computer language is either compiled(c programming) or interpreted. Java combines both these approaches thus making Java a two-stage system.Java compiler (javac) translates source code into bytecode instructions.when a program is compiled to an intermediate form (bytecode) and then interpreted by a virtual machine(JVM).Thus we can say Java is both compiled and Interpreted.

2. Platform Independent and Portable

Java is platform independent(that is, architecture-neutral), meaning that you can write a program once and run it on any computer. platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments.

3. Object-Oriented

Object-oriented programming (OOP) is a programming language model organized around objects rather than "actions" and data rather than logic. Java is true object oriented programming language. Almost everything in Java is an object. All program code and data reside within object and class.

4. Robust

Java is a robust language. The program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. Robust simply means strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

5. Secure

Security is the most important part of every language which is used as programming on internet. Java has better security for every purpose. Java system not only verifies all memory access but also ensure that no viruses are communicated with an applet. In java memory locations can not be accessed without proper authorization.

6. Portable

Java Byte code can be carried to any platform. No implementation dependent features. Everything related to storage is predefined, example: size of primitive data types

7. Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. This is mainly designed for network application. It has the ability to share both data and programs. Java enables multiple programmers at multiple remote locations to collaborate and work together in a single program.

8. Architecture-Neutral

Java compiler(javac) generates an architecture-neutral object file format(byte code), which makes the compiled code(byte code) executable on many processors, with the presence of Java Runtime Environment(JRE).

The bytecode is similar to machine instructions but is architecture neutral and can run on any platform that has a Java Virtual Machine (JVM). The goal of Java is: ***"write once; run anywhere, anytime, forever."***

9. Familiar, Simple and Small

Java is simple and easy to learn.

It is also a small and simple language for that Java eliminates pointers, preprocessor header files etc.

10. Multithreaded and Interactive

Java was designed to meet the real-world requirement of creating interactive, networked programs.

Multithreading means handling multiple tasks simultaneously. It supports multithreaded programs, which allows you to write programs that do many things simultaneously.

For this feature, we need not wait for the application to finish one task before beginning another. For example, we can listen to any song while doing any program at the same time we can download any file from the internet. so this is an example of multi threading and an interactive performance.

11. High Performance

Java performance is impressive for an interpreted language. Java allows us for the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode.

12. Dynamic and Extensible

Java is dynamic language because it is capable of linking in new class libraries, methods, and objects.

it is extensible because it supports native methods(methods which are written in another language like c/c++).Native methods are linked dynamically at runtime

Character Set

The character set is a set of alphabets, digits and some special characters that are valid in Java language.

The smallest unit of Java language is the characters need to write java tokens.

These character set are defined by Unicode character set.

Character Set Consists Of

Alphabets

Java accepts both lowercase and uppercase alphabets as variables and functions.

Uppercase: A B C..... X Y Z

Lowercase: a b c ..... x y z

Digits

0 1 2 3 4 5 6 7 8 9

Special Characters

+ _ ( ) { } [ ] \ | / > < ; etc.

White Spaces

Java Tokens

Token

Token is the smallest individual unit of a program.
Following are the tokens used in Java programming language.

- Keywords
- Identifiers
- Literals
- Operators
- Separators

Keywords
Keywords are the reserved words used for special purpose and hence must not be used as normal identifier name.
Following are the keywords used in Java.

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this

Identifiers

Identifiers are used for the names givens to different parts of the program like the name of the variables, methods, arrays etc.

Identifiers name are user-defined and must follow the given rules.

- First character of the identifier must be a letter or underscore
- Identifier names must only use letters (A-Z and a-z), digits (0-9) and underscore _
- Can't use white spaces
- Can't use keywords

Example: age, _score, isGameOver etc.

Literals

Literals in java are sequence of characters (digits, letters and other characters) that represent constant values to be stored in variables. Java language specifies five major type of literals. They are below :

- Integer literal
- Floating Point literal
- Character literal
- String literal
- Boolean literal

Constant	**Type of Value Stored**
Integer Literals	Literals which stores integer value
Floating Literals	Literals which stores float value
Character Literals	Literals which stores character value
String Literals	Literals which stores string value
Boolean Literals	Literals which stores true or false

Operators

An operator is a symbol that takes one or more arguments and then performs some operations and returns some result.

Example: The addition operator + takes two numbers and returns the sum.

Separators

These are Special Symbols used to Indicate the group of code that is either be divided or arrange into the Block The Separators includes Parentheses, Open Curly braces Comma, Semicolon, or either it will be period.

Symbol	**Name**	**Purpose**
( )	Parentheses	Used to contain lists of parameters in method definition and invocation .Also used for defining

		precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

Comments

We use comments to describe the code, highlight points that can be referred by the developer and to prevent a piece of code from getting compiled.
There are two type of comments in Java languages.

- Single line comments
- Multi line comments

We use the two forward slash to create single line comments.

```
// this is a single line comment
```

We use the /* to start a multi line comment and */ to end a multi line comment.

```
/**
 * this is a
 * multi line comment
 */
```

Constants

Constants are the fixed values that never change during execution of a program.
Following are type of constants in Java.

- Integer constants
- Real or Floating point constants
- Character constants
- String constants

Integer constants

These are the numbers without any decimal part. And they can be in decimal (bas 10), octal (base 8) or hexadecimal (base 16) number system.

Decimal integer constants uses ten digits from 0 to 9.
Example: -99, 0, 100 are decimal integer constants.

Octal integer constants starts with 0 and uses eight digits from 0 to 7.
Example: 01, 010, 011 are octal integer constants.

Hexadecimal integer constants uses ten digits from 0 to 9 and six letters from A to F. We can also use small letter from a to f.
Example: 0xA, 0xff, 0x0b are hexadecimal integer constants.

Real or Floating point constants

These are the numbers with decimal part.
Example: -10.12, 0.5, 100.234 are all real constants.

We can represent real constants in exponential i.e scientific notation.
Syntax: mantissa e exponent

Where, mantissa is either a real number or integer.And exponent is an integer with plus and minus symbol.

In the following example we are expressing 123000 in exponential form.
123000 = 1.23e5
Similarly, we can represent -0.012 in exponential form as follows.
-0.012 = -1.2e-2

Character constants

Character constants are single character enclosed in single quotes.
Example: 'a', 'b', 'c', '1', '2' etc are character constants.

String constants

String constants are sequence of characters enclosed in double quotes.
Example: "Hello World", "A", "B", "1" etc are string constants.
'a' is a character constant as it is using single quotes.
"a" is a string constant as it is using double quotes.

Escape Sequence

These are special backslash character constants in Java.
Some of the escape sequence is listed below.

Character Constant	Description
'\b'	Backspace
'\f'	Form feed
'\r'	Carriage return
'\t'	Horizontal tab
'\v'	Vertical tab

'\n'	New line

Variables

A variable is a named memory location to hold some value.
The value stored in a variable can be changed any time we want.
Example of variables: isGameOver, name, age etc.

Rules for variable names

To name variables we have to follow the given rules.

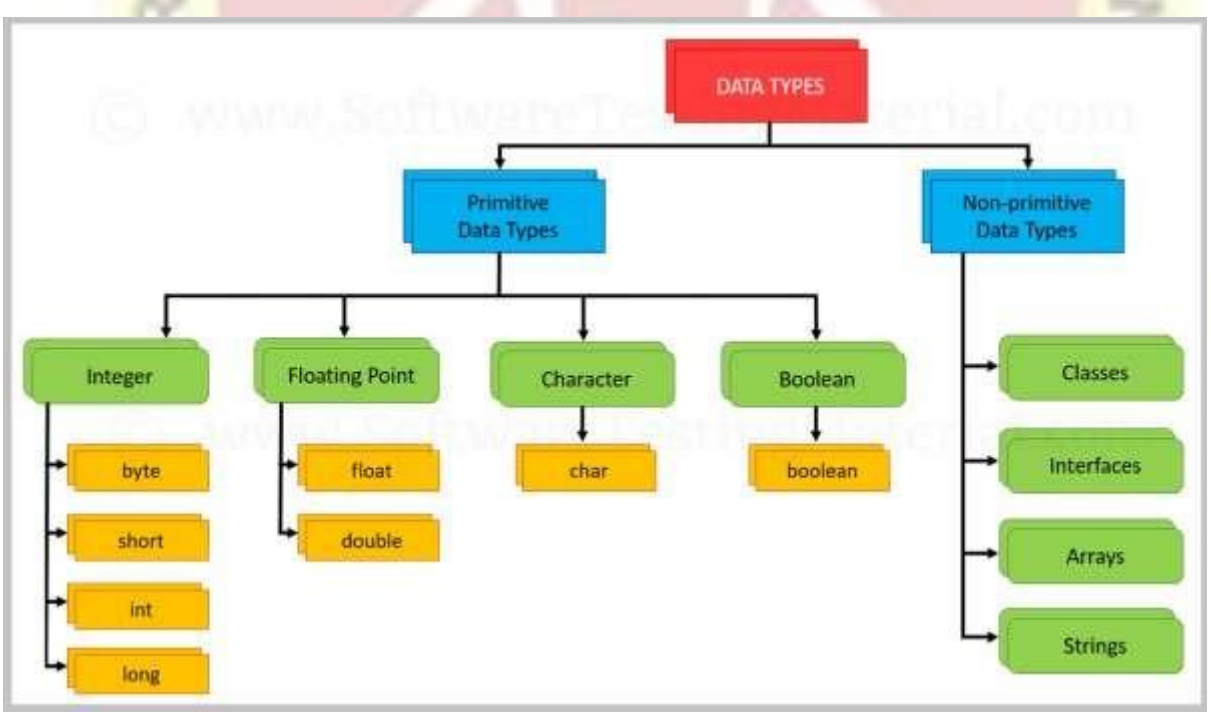
- First character must be a letter or underscore
- Can use letters (A-Z and a-z), digits (0-9) and underscore _
- Must not be a keyword
- Must not contain white space
- Variable name can be of any length

Variable names are case-sensitive so, isGameOver and isgameover are treated as two separate variables.

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, Stings and Arrays.



Primitive Data Type:

There are 8 primitive data types such as byte, short, int, long, float, double, char, and boolean. Size of these 8 primitive data types would not change from one OS to other.
byte, short, int & long – stores whole numbers
float, double – stores fractional numbers
char – stores characters
boolean – stores true or false

1. Integer types

Range of Integer types

KEYWORD	SIZE (bytes)	EXAMPLE
byte	1	byte b, c
short	2	Short b, c
int	4	int b, c
long	8	long b, c

2. Floating point types

Range of Floating point types

KEYWORD	SIZE (bytes)	EXAMPLE
float	4	float a, b
double	8	double pi

3. Character type

These take 2 bytes of memory space and we use the char keyword to create a character data type variable.

Example:

```
char ch = 'y';
```

4. Boolean type

Boolean data type can take only two values true and false and we use the boolean keyword to create a boolean type variable.

Example:

boolean isGameOver = true;

Non-primitive Data Type:

Non-primitive data types include Classes, Interfaces and Arrays

Operator in Java

An operator is a symbol that perform certain mathematical or logical manipulation. Operators are used in the program to manipulate data and variables. Java operators can be divided into following categories:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment and decrement operators
6. conditional operator
7. Bitwise Operators
8. Special Operators

1. Arithmetic Operators

- The basic arithmetic operations in Java Programming are addition, subtraction, multiplication, and division.
- Arithmetic Operations are operated on Numeric Data Types as expected.
- Arithmetic Operators are "Binary" Operators i.e they operate on two operands. These operators are used in mathematical expressions.

Operator	Description	Example
+ (Addition)	Adds two operands	5 + 10 =15
- (Subtraction)	Subtract second operands from first. Also used to Concatenate two strings	10 - 5 =5
* (Multiplication)	Multiplies values on either side of the operator.	10 * 5 =50

/ (Division)	Divides left-hand operand by right-hand operand.	10 / 5 =2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	5 % 2 =1
++ (Increment)	Increases the value of operand by 1.	2++ gives 3
-- (Decrement)	Decreases the value of operand by 1.	3-- gives 2

2. Relational Operators

- Relational Operators are used to checking relation between two variables or numbers.
- Relational Operators are Binary Operators.
- Relational Operators returns "Boolean" value .i.e it will return true or false.

Operators	Descriptions	Examples
== (equal to)	This operator checks the value of two operands, if both are **equal,** then it returns true otherwise false.	(2 == 3) is not true.
!= (not equal to)	This operator checks the value of two operands, if both are **not equal,** then it returns true otherwise false.	(4 != 5) is true.
> (greater than)	This operator checks the value of two operands, if the left side of the operator is **greater,** then it returns true otherwise false.	(5 > 56) is not true.
< (less than)	This operator checks the value of two operands if the left side of the operator is **less,** then it returns true otherwise false.	(2 < 5) is true.
>= (greater than or equal to)	This operator checks the value of two operands if the left side of the operator is **greater or equal**, then it returns true otherwise false.	(12 >= 45) is not true.
<= (less than or equal to)	This operator checks the value of two operands if the left side of the operator is **less or equal**, then it returns true otherwise false.	(43 <= 43) is true.

3. Logical Operators

Operator	Description	Example

&& **(logical and)**	If both the operands are non-zero, then the condition becomes true.	(0 && 1) is false
|| **(logical or)**	If any of the two operands are non-zero, then the condition becomes true.	(0 || 1) is true
! **(logical not)**	Logical NOT Operator Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(0 && 1) is true

4. Assignment Operators

Assignment operator is used to assign value to a variable.

The assignment operator (=) is the most commonly used binary operator in Java. It evaluates the operand on the right hand side and then assigns the resulting value to a variable on the left hand side. The right operand can be a variable, constant, function call or expression. The type of right operand must be type compatible with the left operand.

The general form of representing assignment operator is

Variable = expression/constant/function call

```
a = 3; //constant

x = y + 10; //expression
```

Compound assignment operators in Java

Compound-assignment operators provide a shorter syntax for assigning the result of an arithmetic or bitwise operator. They perform the operation on the two operands before assigning the result to the first operand.

The following are all possible assignment operator in java:

1. += (compound addition assignment operator)
2. -= (compound subtraction assignment operator)
3. *= (compound multiplication assignment operator)
4. /= (compound division assignment operator)
5. %= (compound modulo assignment operator)
6. &= (compound Bitwise & assignment operator)

7. |= (compound Bitwise | assignment operator)

8. ^= (compound Bitwise ^ assignment operator)

9. >>= (compound right-shift assignment operator)

10. >>>=(compound right-shift filled 0 assignment operator)

11. <<=(compound left-shift assignment operator)

5. Increment and decrement operators

- Increment and Decrement Operators are Unary Operators.
- Increment Operator is Used to Increment Value Stored inside Variable on which it is operating.
- Decrement Operator is used to decrement value of Variable by 1 (default).

Types of Increment and Decrement Operator :

- Pre Increment / Pre Decrement Operator
- Post Increment / Post Decrement Operator

Syntax :
++ Increment operator : increments a value by 1
-- Decrement operator : decrements a value by 1

Pre-Increment Operator :
"++" is written before Variable name.
Value is Incremented First and then incremented value is used in expression.

Operator	Description	Example
++ (Increment)	Increases the value of operand by 1.	2++ gives 3
-- (Decrement)	Decreases the value of operand by 1.	3-- gives 2

Example:

```
class PreIncrement
 {
  public static void main(String args[ ])
{
    int num1 = 1;
    int num2 = 1;
    --num1;
    --num2;
    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
```

```
  }
}
```

Output :

```
num1 = 0
num2 = 0
```

- **Pre Increment :** First Increment Value and then Assign Value.
- **++num1** will increment value inside "**num1**" variable. New value is assigned to itself.
- New Value of num1 = 0.

Example:

```
class PostIncrement
{
  public static void main(String args[ ])
{
    int num1 = 1;
    int num2 = 1;

    num1++;
    num2++;

    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
  }
}
```

Output :

```
num1 = 2
num2 = 2
```

- **Post Increment :** Increment Value of Variable After Assigning
- **num1++** will increment value inside "**num1**" variable after assigning old value to itself.
- New Value of num1 = 2.

6. Conditional Operators

The Conditional operator is the only ternary (operator takes three arguments) operator in Java. The operator evaluates the first argument and, if true, evaluates the second argument. If the first argument evaluates to false, then the third argument is evaluated. The conditional operator is the expression equivalent of the if-else statement.

The general form:

Expression1 ? Expression2 : Expression3

Expression ? value if true : value if false

Example:

```
class TernaryOperatorsDemo
 {
  public static void main(String args[ ])
{
        int x = 10, y = 12, z = 0;
        z = x > y ? x : y;
        System.out.println("z : "+z);
  }
  }
```

7. Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators −

Assume integer variable A holds 60 and variable B holds 13 then –

Operator	Description	Example
& (bitwise and)	Bitwise AND operator give true result if both operands are true. otherwise, it gives a false result.	(R & S) will give 12 which is 0000 1100
| (bitwise or)	Bitwise OR operator give true result if any of the operands is true.	(R | S) will give 61 which is 0011 1101
^ (bitwise XOR)	Bitwise Exclusive-OR Operator returns a true result if both the operands are different. otherwise, it returns a false result.	(R ^ S) will give 49 which is 0011 0001
~ (bitwise compliment)	Bitwise One's Complement Operator is unary Operator and it gives the result as an opposite bit.	(~R) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	R << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	R >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	R >>>2 will give 15

8. Special Operators

Java provides some special operators as given below.

1. instanceof Operator

This operator is used to know if an object is an instance of a particular class or not. This operator returns “true” if an object given at the left hand side is an instance of the class given at right hand side. Otherwise, it returns “false”.

Example

circle instanceof Shape

The above statement checks if the object “circle” is an instance of the class “Shape”. If yes, it returns “true”, else returns “false”

2. Dot operator

This operator is used to access the variables and methods of a class.

Example 1

student.mark

Here we are accessing the variable “mark” of the “student” object

Example 2

student.getMarks()

Here we are accessing the method “getMarks” of the “student” object.

This operator also can be used to access classes, packages etc.

Type Casting

Definition:

Type casting is a process of converting one data type into another data type.

syntax of type casting:

data_type variableName = (data_type) varName;

Where, varName is a variable that we are type casting into given data_type and saving the result in other variable variableName.

In the following example we are type casting variable of type int into type double.

```
// variable of type int
int a = 10;

// variable of type double
double d;

// type cast int to double
d = (double) a;
```

1. Automatic (Implicit) Type Casting

In Java, if destination variable has enough space to accommodate value of the source variable then Java will automatically perform the type casting.

For example a variable of type int is of 4 bytes while a variable of type byte is of 1 byte size. So, the byte variable will be automatically type casted into int.

```
// byte variable
byte b = 10;
// int variable
```

```
int i;
// automatic type casting
i = b;
```

2. Explicit Type Casting

In this we explicitly mention the data type that we want to cast into.

For explicit type casting we use the following syntax.

(type) expression

In the following example we are type casting variable of type float to int.

```
// float variable
float f = 10.11F;

// int variable
int i;

// explicit type casting
i = (int) f;
```

Widening and Narrowing

The process of assigning a smaller data type to a larger data type is known as widening.

The process of assigning a larger data type to a smaller data type is known as narrowing.

Narrowing leads to loss of information.

Example:

```
class Math {
  public static void main(String args[ ])
{
    int a = 5;
    int b = 2;
    float result = (float) a/b;
    System.out.println("Result: " + result);
  }
}
```

Output:

Result: 2.5

Control Statements: Branching and Looping Statements

Java Control statements control the order of execution in a java program, based on data values and conditional logic. There are three main categories of control flow statements;

- **Selection statements: if, if-else and switch.**
- **Loop statements: while, do-while and for.**

- **Transfer statements: break, continue, return**

We use control statements when we want to change the default sequential order of execution

Selection statements:

If Statement:

This is a control statement to execute a single statement or a block of code, when the given condition is true and if it is false then it skips **if** block and rest code of program is executed .

Syntax:

```
if(conditional_expression){
<statements>;
...;
...;
}
```

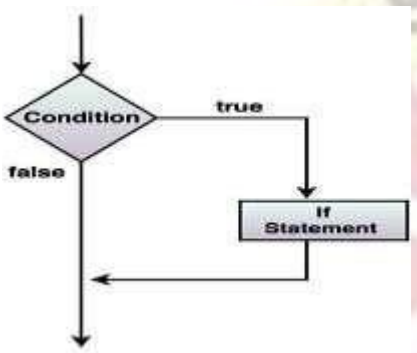


Fig: Flow diagram of if statement

Example: If n%2 evaluates to 0 then the "if" block is executed. Here it evaluates to 0 so if block is executed. Hence **"This is even number"** is printed on the screen.

```
   int n = 10;
if(n%2 = = 0)
{
  System.out.println("This is even number");
}
```

2. **If-else Statement:**

The **"if-else"** statementis an extension of if statement that provides another option when 'if' statement evaluates to "false" i.e. else block is executed if **"if"** statement is false.

Syntax:

```
if(conditional_expression){
<statements>;
...;
...;
}
 else{
```

```
<statements>;
....;
....;
 }
```

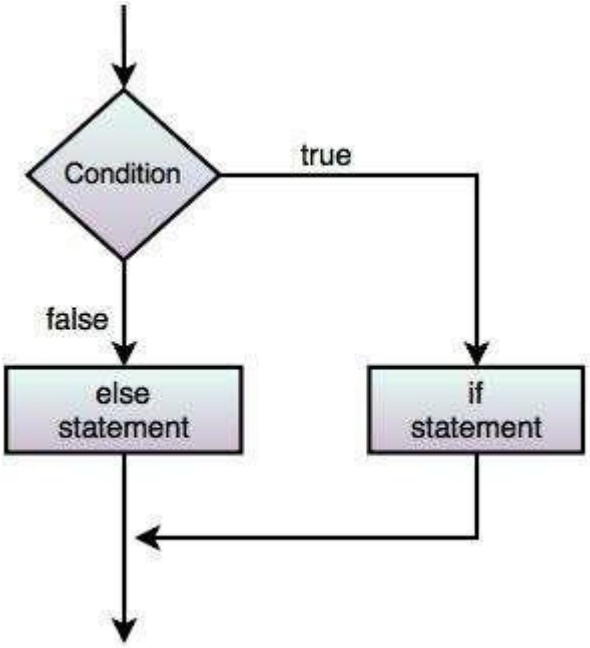


Fig: Flow diagram of if else statement

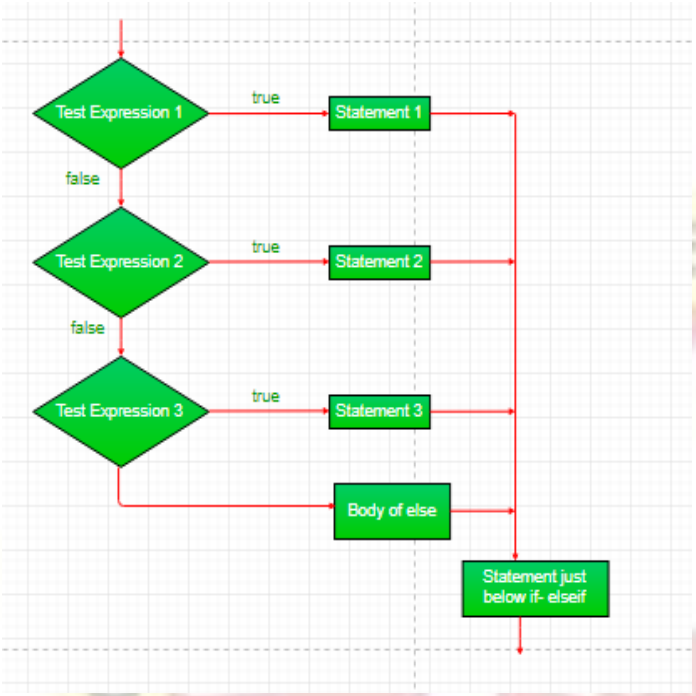
Example: If n%2 doesn't evaluate to 0 then else block is executed. Here n%2 evaluates to 1 that is not equal to 0 so else block is executed. So **"This is not even number"** is printed on the screen.

```
    int n = 11;
if(n%2 = = 0)
{
  System.out.println("This is even number");
}
else{
  System.out.println("This is not even number");
    }
```

3.**if-else-if ladder:** Here, a user can decide among multiple options.The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

```
    if (condition)
        statement;
    else if (condition)
```

```
    statement;
.
.
else
    statement;
```



Example:

```
class ifelseifDemo
{
    public static void main(String args[])
    {
        int i = 20;

        if (i == 10)
            System.out.println("i is 10");
        else if (i == 15)
            System.out.println("i is 15");
        else if (i == 20)
            System.out.println("i is 20");
        else
            System.out.println("i is not present");
    }
}
```

Output: i is 20

4.S witch Statement:

It is a multi-way branch with several choices. A switch is easier to implement than a series of if/else statements.

The switch statement begins with a keyword switch, followed by an expression that equates to a no long integral value. Following the controlling expression is a code block that contains zero or more labeled cases.

Each label must equate to an integer constant and each must be unique. When the switch statement executes, it compares the value of the controlling expression to the values of each case label. The program will select the value of the case label that equals the value of the controlling expression and branch down that path to the end of the code block.

If none of the case label values match, then none of the codes within the switch statement code block will be executed. Java includes a default label to use in cases where there are no matches.. Its general form is as follows:

Syntax:

switch (conditional- expression>)
{
case label$_1$: <statement$_1$>
case label$_2$: <statement$_2$>
...
case label$_n$: <statement$_n$>
default: <statement>
}

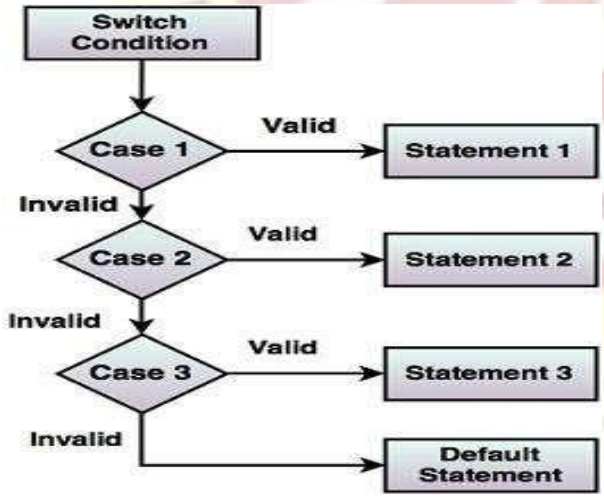


Fig: Flow diagram of switch statement

```
public class SwitchCaseStatementDemo
{
        public static void main(String[ ] args)
{
                int a = 10, b = 20, c = 30;
                int status = -1;
                if (a > b && a > c) {
                        status = 1;
                } else if (b > c) {
                        status = 2;
                } else {
                        status = 3;
                }
                switch (status) {
                case 1:
                        System.out.println("a is the greatest");
                        break;
                case 2:
                        System.out.println("b is the greatest");
                        break;
                case 3:
                        System.out.println("c is the greatest");
                        break;
                default:
                        System.out.println("Cannot be determined");
                }
        }
}
```

Output: c is the greatest

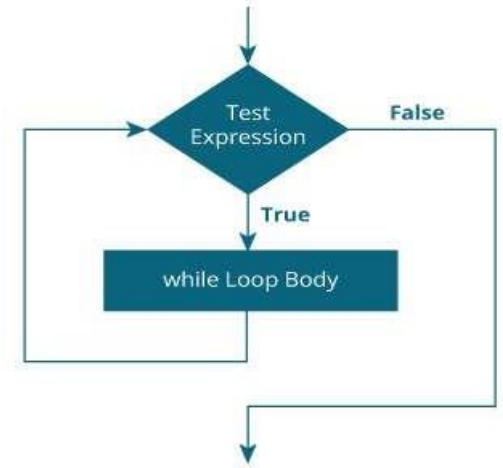
Looping Statements:

1. **while loop statements:**

This is a looping or repeating statement. It executes a block of code or statements till the given condition is true. The expression must be evaluated to a boolean value. It continues testing the condition and executes the block of code. When the expression results to false control comes out of loop.

Syntax:

```
 while(test-expression){
<statement>;
...;
...;
 }
```



// Program to find the sum of natural numbers from 1 to 100.

```
class sumofnos
 {
   public static void main(String[ ] args)
 {
      int sum = 0, i = 100;
      while (i != 0) {
         sum = sum + i;
         --i;
      }
      System.out.println("Sum = " + sum);
   }
}
```

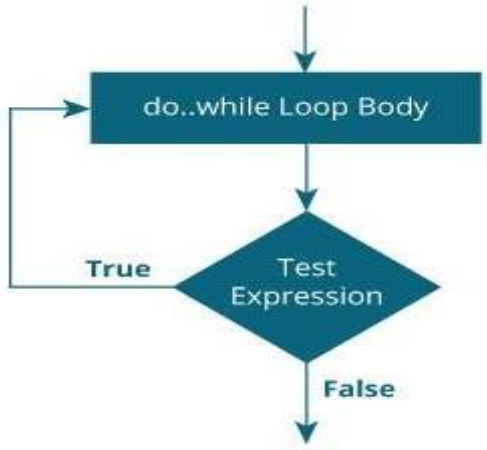
Output: Sum = 5050

Do-while Loop Statement

The do-while loop is similar to the while loop, except that the test is performed at the end of the loop instead of at the beginning. This ensures that the loop will be executed at least once. A do-while loop begins with the keyword do, followed by the statements that make up the body of the loop. Finally, the keyword while and the test expression completes the do-while loop. When the loop condition becomes false, the loop is terminated and execution continues with the statement immediately following the loop.

Syntax:

```
  do
{
  <statement>;
  ...;
  ...;
  }while (test-expression);
```




```
class DoWhile
{
    public static void main(String args[])
    {
        int a = 1;
        do
        {
            System.out.println("Value of a : "+a);
            ++a;
        }
        while (a<10);
    }
}
```

Output:
Value of a : 1
Value of a : 2
Value of a : 3
Value of a : 4
Value of a : 5
Value of a : 6
Value of a : 7
Value of a : 8
Value of a : 9

For loop

The for loop is a looping construct which can execute a set of instructions a specified number of times.

Syntax:

```
for (initialization; condition; increment/decrement)
{
    statements ;
}
```

initialization: The loop is started with the value specified.
condition: It evaluates to either 'true' or 'false'. If it is false then the loop is terminated.
increment or decrement: After each iteration, value increments or decrements.

```
public class ForLoopDemo {

	public static void main(String[] args) {
		System.out.println("Printing Numbers from 1 to 10");
		for (int count = 1; count <= 10; count++) {
			System.out.println(count);
		}
	}
}
```

Output

```
Printing Numbers from 1 to 10
1
2
3
4
5
6
7
8
9
10
```

Jump Statements:

Java supports three jump statements: break, continue, and return. These statements transfer control to another part of the program.

1. **Break statements:**

The break construct is used to break out of the middle of loops: for, do, or while loop. When a break statement is encountered, execution of the current loops immediately stops and resumes at the first statement following the current loop. That is, we can force immediate termination of a loop, bypassing any remaining code in the body of the loop.

Syntax:

```
break;  // breaks the innermost loop or switch statement.
break label;  // breaks the outermost loop in a series of nested loops.
```

```
public static void main(String args[] )
	{
		int i;
		i=1;
		while(true)
		{
			if(i >10) break;
			System.out.print(i+" ");
			i++;
		}
	}
}
```

Continue statements:

This command skips the whole body of the loop and executes the loop with the next iteration. On finding continue command, control leaves the rest of the statements in the loop and goes back to the top of the loop to execute it with the next iteration (value).

Syntax:

```
continue;
```

```
/* Print Number from 1 to 10 Except 5 */
class NumberExcept
{
        public static void main(String args[] )
        {
                int i;
                for(i=1;i<=10;i++)
                {
                        if(i==5) continue;
                        System.out.print(i +" ");
                }
        }
}
```

Return statements:

This statement is mainly used in methods in order to terminate a method in between and return back to the caller method. It is an optional statement. That is, even if a method doesn't include a return statement, control returns back to the caller method after execution of the method. Return statement mayor may not return parameters to the caller method.

Syntax:

```
return;
return values;
```

return; //This returns nothing. So this can be used when method is declared with void return type.
return expression; //It returns the value evaluated from the expression.

Unit-2:Classes, Objects and Methods - Constructors - Methods Overloading-Inheritance-Overriding Methods-Finalizer and Abstract Methods-Visibility Control –Arrays, Strings and Vectors-StringBuffer Class-Wrapper Classes

Classes, Objects and Methods

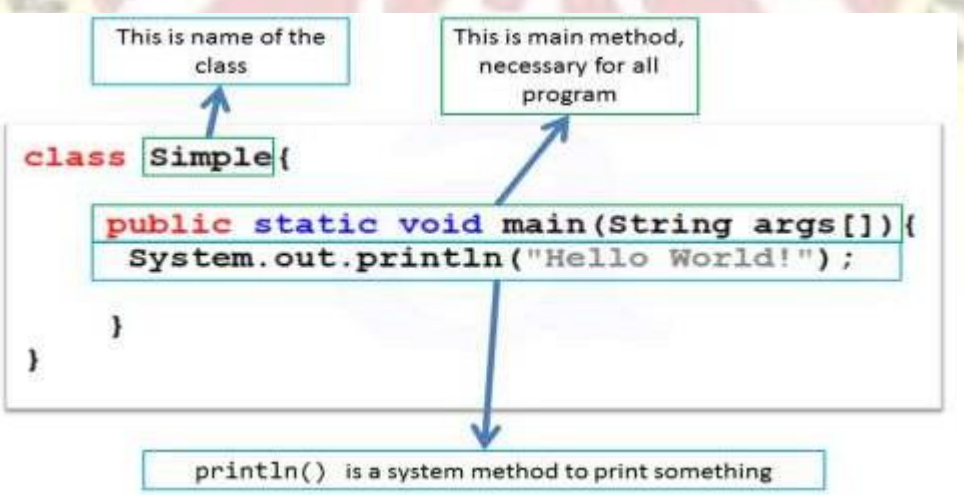
Object Definition: An object is an instance of a class.

Classes

- A class is a group of objects which have common properties.
- Class is a blueprint to create different objects.
- It encapsulates the object, class, methods, constructor, block and data member.
- class keyword is used to declare a class.

Syntax:

```
class ClassName [extends superclassname]
{
    datatype variablename ;//variable declaration
    datatype methodname( parameter )
    {
        method – body
    }
}
```



Example: Sample of a Class

```
public class Employee
{
    String Name;
    int EmpId;
    int age;
    double salary;
    void empDept()
    {
    }
    void empProject()
    {
    }
}
```

Object Creation

- Creation of an object is also called as instantiation of an object.
- The new operator is used to create an object of a class.
- If we want to create an object of the class and have the reference variable associated with this object, we must also allocate memory for the object by using the new operator. This process is called instantiating an object or creating an object instance.
- When we create a new object, we use the new operator to instantiate the object.
- The new operator returns the location of the object which we assign o a reference type.

Example:

```
Employee emp; // declare
emp=new Employee; // Instantiate;
```

The first statement declare a variable to hold the object reference and the sencond one actually assigns the object reference to the variable.The variable emp is now an object of the Employee class.

Both the statements can be combined into one as shown below:

Employee emp = new Employee ( );

Where emp is an object of Employee class. Employee ( ) is the default constructor of that class.

Example Java Program:

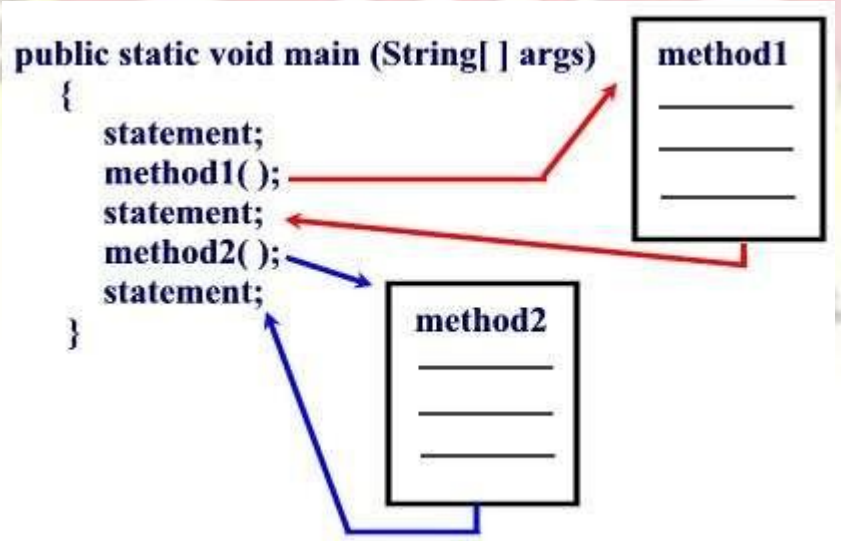
```
public class Cube {

        int length = 10;
        int breadth = 10;
        int height = 10;
        public int getVolume() {
                return (length * breadth * height);
        }
        public static void main(String[] args) {
                Cube cubeObj; // Creates a Cube Reference
                cubeObj = new Cube(); // Creates an Object of Cube
                System.out.println("Volume of Cube is : " + cubeObj.getVolume());
        }
}
```

Java - Methods

A Java method is a collection of statements that are grouped together to perform an operation. That is a method is like a function which is used to expose the behaviour of an object.

The methods are declared inside the body

A method is a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name.



Method definitions have four basic parts:

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

The syntax for defining a method is:

Method definition consists of a method header and a method body. The same is shown in the following syntax

modifier type methodname (parameter-list)

{

// Method body;

}

- **modifier** − It defines the access type of the method and it is optional to use.
- **type** − Method may return a value. It can be any data type.
- **methodname** − This is the method name. The method consists of the method name and the parameter list.
- **parameter-list** − The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **formal parameter -** The variables defined in the method header are known as formal parameters or simply parameters. A parameter is like a placeholder.
- **actual parameter -** When a method is invoked, you pass a value to the parameter. This value is referred to as an actual parameter or argument.
- **method body** − The method body defines what the method does with the statements.

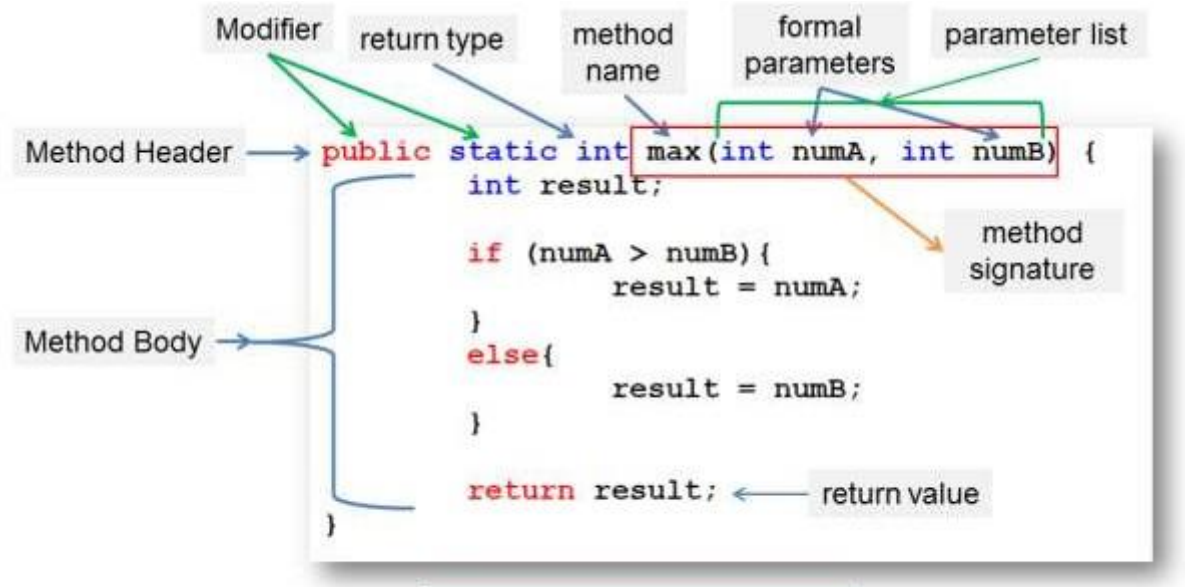


Figure: How to define a method

```
class MethodsExample{
        public static void main(String args[])
        {

    int a = 15;
    int b = 8;
    int c = max(a, b);
    System.out.println("Largest Value = " + c);

   }

  public static int max(int numA, int numB) {
        int result;

        if (numA > numB){
                result = numA;
        }
        else{
                result = numB;
        }

        return result;
  }
}
```

Output:
Largest Value = 15

Method Calling

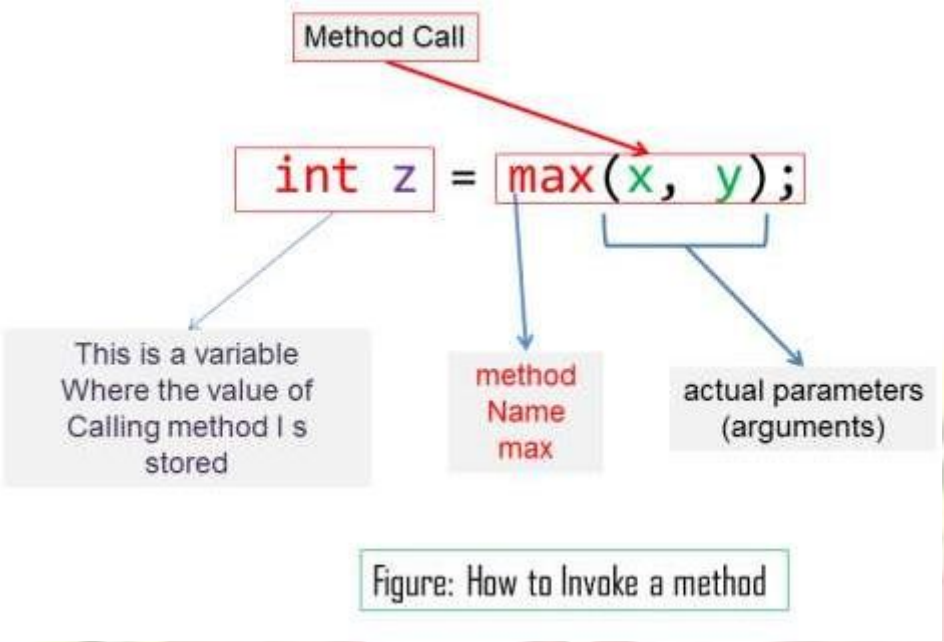


Figure: How to Invoke a method

In a method definition, we define what the method is to do. To execute the method, we have to call or invoke it.There are two ways in which a method is called depending on whether the method returns a value or not.

method returns a value
If a method returns a value, a call to the method is usually treated as a value. and this value will store in a variable For example,
int largestNumber = max(15, 8);
calls max(15, 8) and assigns the result of the method to the variable largestNumber.

method returns but not in a variable
The methods returning nothing or void is considered as call to a statement. consider an example

System.out.println(max(15, 8));

This is also a example of method calling, which prints the return value of the method call max(15, 8)

Void Method

. The void keyword allows us to create methods which do not return a value.

```
public class swappingExample {

   public static void main(String[] args) {
      int a = 30;
      int b = 45;
      System.out.println("Before swapping, a = " + a + " and b = " + b);

      // Invoke the swap method
      swapFunction(a, b);
      System.out.println("\n**Now, Before and After swapping values will be same here**:");
      System.out.println("After swapping, a = " + a + " and b is " + b);
   }

   public static void swapFunction(int a, int b) {
      System.out.println("Before swapping(Inside), a = " + a + " b = " + b);

      // Swap n1 with n2
      int c = a;
      a = b;
      b = c;
      System.out.println("After swapping(Inside), a = " + a + " b = " + b);
   }
}
```

Output

```
Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30

**Now, Before and After swapping values will be same here**:
After swapping, a = 30 and b is 45
```

Constructors

- A special type of method that is used to initialize the object, is known as constructor.
- They are invoked automatically when an object creates.
- Java compiler creates a default constructor for a class when any explicit constructor of that class is not declared.

Rules for creating Java constructor

There are basically three rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type. i.e Constructors do not have a return type — not even void
3. Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.

Types of Java constructors

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Default constructor

A constructor that have no parameter is known as default constructor.

The Default constructor is also known as no-args (no arguments) constructor because it has no parameter. The constructor is called "Default Constructor" when it doesn't have any parameter. The default constructor which takes no arguments and performs no special actions or initializations, when no explicit constructors are provided.

Syntax of default constructor:

```
ClassName( )
{

}
```

Example of Default Constructor

In this example, we are creating the no-arg constructor in the Sample class. It will be invoked at the time of object creation.

```
class Sample
{
   Sample()
{ // constructor
         System.out.println("this is inside constructor");
         }
//main method
 public static void main(String args[])
{
//calling a default constructor
  Sample b=new Sample();
  }
}
```

Output:

```
 this is inside constructor
```

Java parameterized constructor

A constructor which has a specific number of parameters is known as parameterized constructor. The Parameterized constructor is used to provide different values to the distinct objects.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
//Java Program to demonstrate the use of parameterized constructor
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n)
{
    id = i;
    name = n;
    }
    //method to display the values
    void display( )
{
System.out.println(id+" "+name);
}
      public static void main(String args[])
{
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

Output:

```
111 Karan
222 Aryan
```

Method Overloading

When a class has two or more methods by the same name but different parameters, it is known as method overloading.

- Method overloading is nothing but compile time polymorphism in Java.
- It is checked by the compiler at compile time.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

```
class Adder
{
static  int add(int a,int b)
{return a+b;
}
static  int add(int a,int b,int c)
{return a+b+c;
}
}
class TestOverloading1
{
public static void main(String[] args)
{
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}
}
```

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder
{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}
}
```

Output:

22
24.9

Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. i.e Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

Purpose of Inheritance

1. **code reusability:** the same methods and variables which are defined in a **parent/super/base** class can be used in the **child/sub/derived** class.
2. It promotes polymorphism by allowing **method overriding**.

Disadvantages of Inheritance

- The main disadvantage of using inheritance is that the two classes (parent and child class) get **tightly coupled**.
- This means that if we change the code of parent class, it will affect to all the child classes which are inheriting/deriving the parent class, and hence, **it cannot be independent of each other**.

Syntax for defining subclass

A subclass is defined as follows:

class subclass-name extends superclass-name

{

variables declaration;

methods declaration;

}

extends is the keyword used to inherit the properties of a class. It indicates that we are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

A class which is inherited is called parent or super class and the new class is called child or subclass.

Let us see how extends keyword is used to achieve Inheritance.

```
class Vehicle.
{
  ......
}
class Car extends Vehicle
{
 .......  //extends the property of vehicle class.
}
```

Now based on above example. we can say that,

- **Vehicle** is super class of **Car**.
- **Car** is sub class of **Vehicle**.

Types of inheritance

There are three types of inheritance in Java.
1. Single level inheritance
2. Multilevel inheritance
3. Hierarchical inheritance
4.Multiple inheritance

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

NOTE :Multiple inheritance is not supported in java

1.Single Level Inheritance

When a class extends only one class, then it is called single level inheritance.

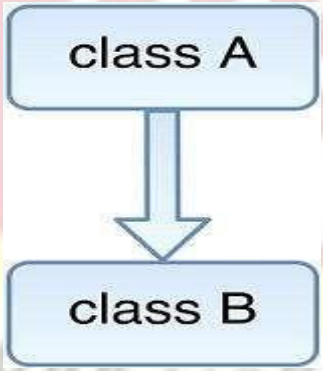


Fig: Single Level Inheritance

Syntax:

```
class A
{
   //code
}
class B extends A
{
   //code
```

}

Example: Sample program for single level inheritance

```
class Parent
{
    public void m1()
    {
       System.out.println("Class Parent method");
    }
}
public class Child extends Parent
{
   public void m2()
   {
      System.out.println("Class Child method");
   }
   public static void main(String args[])
   {
      Child obj = new Child();
      obj.m1();
      obj.m2();
   }
}
```

Output:

```
Class Parent method
Class Child method
```

2. Multilevel Inheritance

Multilevel inheritance is a mechanism where one class can be inherited from a derived class thereby making the derived class the base class for the new class.

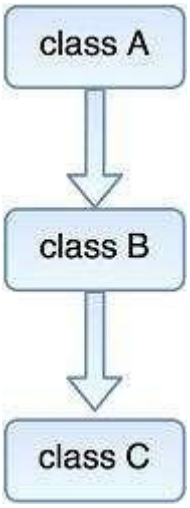


Fig: Multilevel Inheritance

Syntax:

```
class A
{
   //code
}
class B extends A
{
   //code
}
class C extends B
{
   //code
}
```

Example: Sample program for multilevel inheritance

```
class Grand
{
  public void m1()
  {
    System.out.println("Class Grand method");
   }
}
class Parent extends Grand
{
   public void m2()
   {
      System.out.println("Class Parent method");
    }
}
class Child extends Parent
{
   public void m3()
```

```
    {
      System.out.println("Class Child method");
    }
    public static void main(String args[])
    {
      Child obj = new Child();
      obj.m1();
      obj.m2();
      obj.m3();
    }
}
```

Output:
Class Grand method
Class Parent method
Class Child method

3. Hierarchical Inheritance

When one base class can be inherited by more than one class, then it is called hierarchical inheritance.

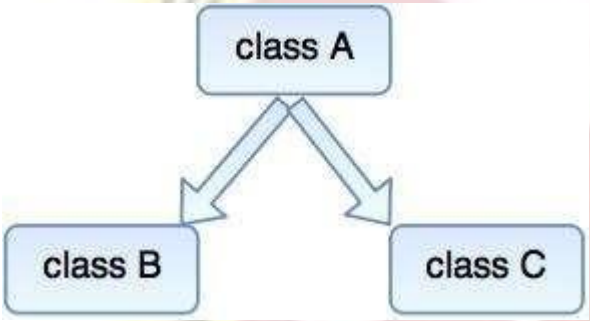


Fig: Hierarchical Inheritance

Syntax:

```
class A
{
   // code
}
class B extends A
{
   // code
}
class C extends A
{
   //code
}
```

Example: Sample program for hierarchical inheritance

```
class A
{
  public void m1()
```

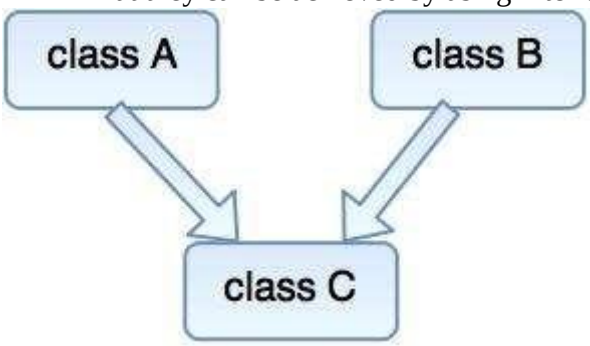
```
  {
    System.out.println("method of Class A");
  }
}
class B extends A
{
  public void m2()
  {
    System.out.println("method of Class B");
  }
}
class C extends A
{
  public void m3()
  {
    System.out.println("method of Class C");
  }
}
class D extends A
{
  public void m4()
  {
    System.out.println("method of Class D");
  }
}
public class MainClass
{
  public void m5()
  {
    System.out.println("method of Class MainClass");
  }
  public static void main(String args[])
  {
    A obj1 = new A();
    B obj2 = new B();
    C obj3 = new C();
    D obj4 = new D();
    obj1.m1();
    obj2.m1();
    obj3.m1();
    obj4.m1();
  }
}
```

Output:

```
method of Class A
method of Class A
method of Class A
method of Class A
```

4.Multiple Inheritances

- In multiple inheritance, one derive class can extend more than one base class.
  - Multiple inheritances are basically not supported by Java, because it increases thecomplexity of the code.
- But they can be achieved by using interfaces.



Example Java Program:

Method Overriding

When a method in a sub class has same name, same number of arguments and same return type as a method in its super class, then the method is known as overridden method. Method overriding is also referred to as runtime polymorphism.

The key benefit of overriding is the abitility to **define method that's specific to a particular subclass type**.

Example Java Program to illustrate the use of Java Method Overriding

```
class Vehicle
{
  //defining a method
  void run() {System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle
{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}
  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

Bike is running safely

Difference between Overloading and Overriding

(refer notes)
Visibility Control
(refer notes)
Static-(refer notes)
Arrays-(refer notes)

Garbage Collection in Java

Objects are dynamically allocated by using the new operator, the objects are destroyed and their memory released for later reallocation.

In Java destruction of object from memory is done automatically by the JVM. When there is no reference to an object, then that object is assumed to be no longer needed and the memory occupied by the object are released. This technique is called Garbage Collection.

finalize() method

Java provides a mechanism called finalization. finalize() method is called by garbage collection program before collecting object. It is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing.

finalize() method

The finalizer method is simply finialize() and can be added to any class. Java calls that method whenever it is needed to reclaim the space for that object. The finialize method should explicitly define the tasks to be performed.

Strings

String is probably the most commonly used class in java library. String represents a sequence of characters. String class is encapsulated under java.lang package. In java, every string that we create is actually an object of type **String**. One important thing to notice about string object is that string objects are immutable that means once a string object is created it cannot be altered.

The easiest way to represent a sequence of characters in java is by using a character array.

For example:

```
char[] chr = {'T', 'u', 't', 'o', 'r', 'i',  'a', 'l'};
```

String s = new String (chr);
or
String str = "Tutorial";

Creating String

There are two ways to create string in Java.
1. By using string literal
2. By using new keyword.

1. By using String Literals
It is simply created by assigning the string value within double quotes ("").

For example:
String s1 = "welcome";
String s2 = "Hello";

When we create string literals, JVM first checks if the string object already exists in string pool. If string already exits, then JVM does not create reference in pool.

2. By using new keyword
String object are also created by use of new keyword.

For example:
String s1 = new String ("Hello");
String s2 = new String ("welcome");

String Methods

The String class define a number of methods that allows us to accomplish a variety of string manipulations tasks.

1. String Methods.

Methods	Description
char charAt(int index)	Returns the character at specified index. The index value should be 0 to length ()-1.
int compareTo(Object o)	Compares the string and string object based on Unicode value.
int compareTo(String string)	Compare between two strings lexicographically means alphabetically.
static String copyValueof(char[ ] data)	Returns a string that represents the character sequence.
boolean endsWith(String suffix)	Returns “true” if the given index ends with string suffix otherwise “false”.
boolean equals(Object O)	This method is used to compare this string to specified object.
int length()	Returns total number of characters in a string.
String toLowerCase()	Converts all the characters in this string to lower case.

String toUpperCase()	Converts total characters of the given string in upper case.
String trim()	Return the copy of the string after eliminating leading and trailing spaces.
String replace(char oldChar, char newChar)	Returns the new string after replacing old character with new character.
String replaceAll(String regexp, String replacement)	Returns the new string replacing all the sequence of the characters matching regular expression and replacement string.
String substring(int beginIndex)	Returns the new string that is substring of that string.
String substring(int beginIndex, int endIndex)	Returns the substring of the given string. In this method begin index is inclusive and end index is exclusive.

Example:

```
public class StringDemo
{
    public static void main(String args[])
    {
        String s1 = "TutorialRide";
        String s2 = "Java Tutorial";
        // check string is empty or not
        System.out.println(s1.isEmpty());
        //find the length of string
        System.out.println(s1.length());
        System.out.println(s2.trim());
        System.out.println(s1.concat(".com"));
        System.out.println(s2.replace('J','V'));
    }
}
```

Output:

```
false
12
Java Tutorial
TutorialRide.com
Vava Tutorial
```

String Buffer

StringBuffer is a peer class of String. String creates strings of fixed length. StringBuffer creates a strings of flexible lenght that can be modified in terms of both length and content. We can insert characters and substrings in the middle of a string or append another string to the end. The following specifies the StringBuffer methods for string manipulations.

Methods	Description
append(s)	Append the specified string with the given string.
insert(n,s2)	Insert the string s2 at the position n of the string s1.
reverse()	Returns the reverse of string.
setlength(n)	sets the length of string to n.
setCharAt(n,x)	Modifies the nth character to x

The append( ) method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has several overloaded versions. Here are a few of its forms in syntax:

Syntax
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)

Program

```
class StringBufferExample{
public static void main(String args[]){

        StringBuffer strng=new StringBuffer("Hello ");
        strng.append("atnyla");//now original string is changed
        System.out.println(strng);//prints Hello Java

        }
}
```

Output: Hello atnyla

The **insert()** method inserts one string into another.

Program

```
// Demonstrate insert().
class insertDemo {
        public static void main(String args[])
 {
        StringBuffer strng = new StringBuffer("I Java!");
        strng.insert(2, "like ");
        System.out.println(strng);
 }
}
```

Output:I like Java!

Vectors

The Vectors class contained in the java.util package.
This class can be used to create a generic dynamic array known as vector that can hold objects of ant type and any number. These are also Called as dynamic Array of object data type .

The Java **Vector class** implements a growable array of objects where the size of the vector can grow or shrink as needed dynamically.

The vectors doesn't support primitives data types like int, float, char etc.
Vector proves to be very useful if we don't know the size of the array in advance.

Syntax:

```
Vector v=new Vector(); //declaring without size
Vector list=new Vector(3); //declaring with size
```

Advantages of Vector

- It can grow dynamically.
- It provides more powerful insertion and search mechanisms than arrays.
- Vectors can hold Objects of different classes.
- Vectors are synchronous.

Disadvantage of Vector

- Vector is Slow

Vector methods

Sr.No.	Method & Description
1.	**void addElement(item)** Adds the specified component to the end of this vector, increasing its size by one.
2.	**void clear()** Removes all of the elements from this vector.
3.	**void copyInto(Array)** Copies the components of this vector into the specified array.
4.	**Object get(int index)** Returns the element at the specified position in this vector.
5.	**void insertElementAt(Object obj, int index)** Inserts the specified object as a component in this vector at the specified index.
6.	**void removeAllElements()**

	Removes all components from this vector and sets its size to zero.
7.	**void removeElementAt(int index)** removeElementAt(int index).
8.	**void setElementAt(Object obj, int index)** Sets the component at the specified index of this vector to be the specified object.
9.	**int size()** Returns the number of components in this vector.

Example of Vector

```
import java.util.*;
class Vector_demo {
   public static void main(String[ ] args)
   {
      Vector vec = new Vector(7);
       // use add() method to add elements in the vector
      vec.add(1);
      vec.add(2);
      vec.add(3);
      vec.add(4);
      vec.add(5);
      vec.add(6);
      vec.add(7);
      Integer[] arr = new Integer[7];
        // copy componnent of vector int array arr
      vec.copyInto(arr);
      System.out.println("elements in array arr: ");
      for (Integer num : arr) {
         System.out.println(num);
      }
   }
}
```

Output:

```
elements in array arr:
   1
   2
   3
   4
   5
   6
   7
```

Java wrapper classes

Wrapper classes are used to convert any data type into an object.

The primitive data types are not objects; they do not belong to any class; they are defined in the language itself. Sometimes, it is required to convert data types into objects in Java language.
A wrapper class wraps (encloses) around a data type and gives it an object appearance. Wherever, the data type is required as an object, this object can be used. Wrapper classes include methods to unwrap the object and give back the data type. It can be compared with a chocolate.

Example: intValue() is a method of **Integer** class that returns an **int** data type.

- Wrapper classes allow primitive data types to be accessed as objects and objects as aprimitive data types.
- The various wrapper classes for primitive data type are: Boolean, Byte, Character,Double, Float, Integer, Long and Short.

- Wrapper classes make the primitive type data to act as objects.

Features of Wrapper class

- Wrapper classes convert numeric strings into numeric values.
- They provide a way to store primitive data into the object.
- The valueOf() method is available in all wrapper classes except Character.
  - All wrapper classes have typeValue() method. This method returns the value of theobject as its primitive type.

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Doubl

Example:

Integer wrapper class,

The following two statements illustrate the difference between a primitive data type and an object
of a wrapper class:

```
int x = 25;
Integer y = new Integer(33);
```

The first statement declares an int variable named x and initializes it with the value 25. The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y. The memory assignments from

these two statements are visualized in Figure 1.

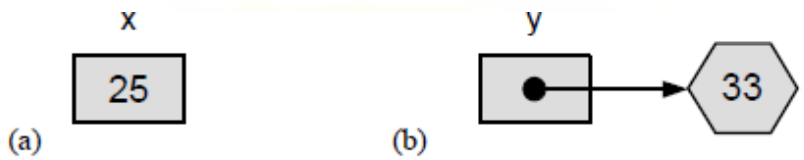


Figure 1. Variables vs. objects (a) declaration and initialization of an `int` variable (b) instantiation of an `Integer` object

Autoboxing

Autoboxing is the process of converting a primitive data type into corresponding wrapper class object. E.g. int to Integer.

Unboxing

Converting an object of wrapper class into corresponding primitive data type is known as unboxing. For example Integer to int.

Advantages of Wrapper Class in Java

The need for Wrapper Classes in Java is as follows:

- They are used to convert the primitive data types into objects (Objects are needed when we need to pass an argument in the given method).
- The **package** java.util contains classes which only handles objects, so wrapper class in Java helps in this case too.
- Data Structures store only objects and primitive data types.
- In **multithreading**, we need an object to support synchronization.

Example: Sample program for autoboxing

```
class AutoBoxDemo
{
   public static void main(String args[])
   {
      int x = 50;
      Integer y = Integer.valueOf(x);
     Integer z = x + y;
     System.out.println(x+"  "+y+"  "+z);
   }
}
```

Output: 50 50 100

Example: Sample program for unboxing

```
class UnBoxDemo
{
   public static void main(String args[])
   {
      Integer x = new Integer(15);
      int y = x.intValue();
      int z = x * y;
      System.out.println(x+"  "+y+"  "+z);
   }
}
```

Output:15 15 225

Unit-III

Interfaces-Packages-Creating Packages-Accessing a Package-Multithreaded Programming-Creating Threads-Stopping and Blocking a Thread-Life Cycle of a Thread-Using Thread Methods-Thread Priority-Synchronization-Implementing the Runnable Interface

Interfaces

An *interface* is a way of describing what classes should do, without specifying how they should do it. A class can implement more than one *interface.*

- Interface is similar to a class, but it contains only abstract methods.
- The Interfaces do not specify any code to implement these methods and data field contain only constants.
- It is the responsibility of class that implements an interface to define the code for implementation.
- By default the variables declared in an interface are public, static and final.
- Interface is a mechanism to achieve full abstraction.
- An interface does not contain any constructor.
- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.

Declaration

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>
{
```

```
    // declare constant fields
   // declare methods that abstract
}
```

Where,

interface is the keyword and <interface_name> is interface name.

The variables and methods are declared as follows:

static final datatype variable name=value; ----> for data member

returntype methodname(parameters)---> for method

Example: Sample of an interface

```
interface Employee
{
   static final Id = 101;
   static final String name = "ABC";
   void show();
   void getSalary(double salary);
}
```

Extending interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

syntax:

```
        interface name2 extends name1
{
   body of name2
}
```

Example: Sample program for extending interfaces

```
interface Base
{
   public void display ();
}
interface Derive extends Base
{
   public void show ();
}
```

Implementing interfaces

A class implements an interface. After that, class can perform the specific behavior on an interface.

The implements keyword is used by class to implement an interface.

Syntax:

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

Note: A class can implement more than one interface. Java can achieve multiple inheritances by using interface.

Example: Sample program to implements interface in Java

```
interface Drawable
{
void draw();
}
class Rectangle implements Drawable
{
public void draw(){System.out.println("drawing rectangle");
}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}

class TestInterface1{
public static void main(String args[])
{
Drawable d=new Circle();
d.draw();
}
}
```

Output:

```
drawing circle
```

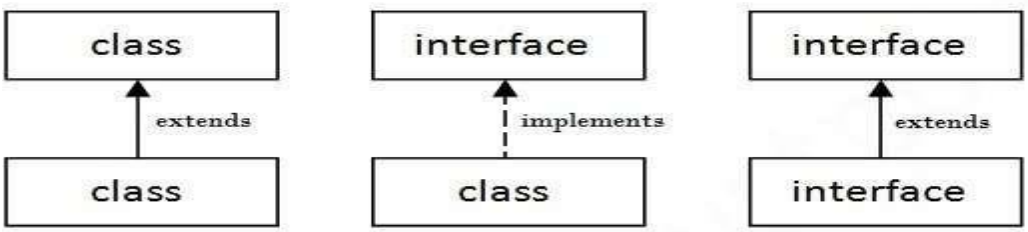
Example: Sample program to implements multiple inheritance

```
interface Vehicle
{
  void run();
}
interface Bike extends Vehicle
{
  void stop();
}


public class Demo implements Bike
{
  public void run()
  {
    System.out.println("Vehicle is running.");
  }
  public void stop()
  {
    System.out.println("Bike is stop.");
  }
  public static void main(String args[])
  {
    Demo obj = new Demo();
    obj.run();
    obj.stop();
  }
}
```

Output:
Vehicle is running.
Bike is stop.

Relationship between class and Interface

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Various forms of Interface Implementations:

Introduction to Packages

A package is a mechanism to group the similar type of classes, interfaces and sub-packages and provide access control. It organizes classes into single unit.
In Java already many predefined packages are available, used while programming.
For example: java.lang, java.io, java.util etc.

Advantages of Packages

- Packages provide code reusability, because a package has group of classes.
- It helps in resolving naming collision when multiple packages have classes with the same name.
- Package also provides the hiding of class facility. Thus other programs cannot use the classes from hidden package.
- One package can be defined in another package.

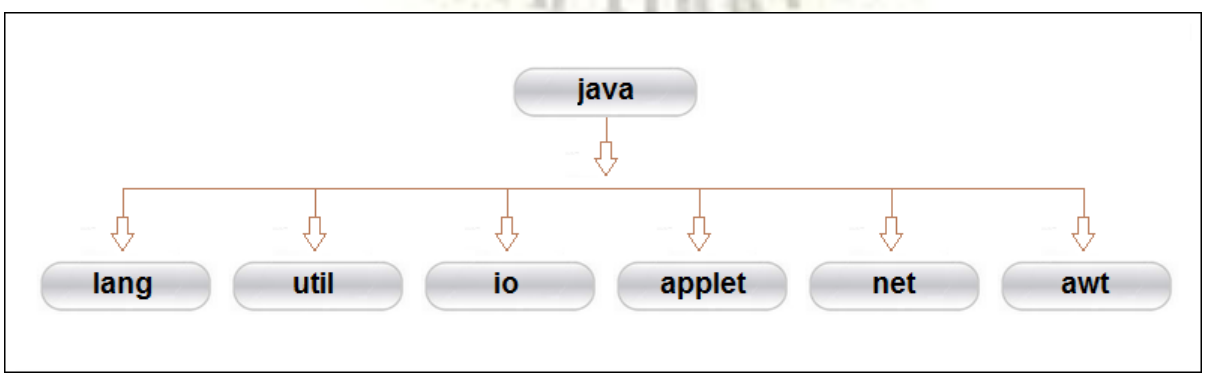
Types of Packages

There are two types of packages available in Java.

1. Built-in packages -Java API packages
Built-in packages are already defined in java API. For example: java.util, java.io, java,lang, java.awt, java.applet, java.net, etc.
2. User defined packages
The package we create according to our need is called user defined package.

Java API packages

Java **API(Application Program Interface)** provides a large numbers of classes grouped into different packages according to functionality. Most of the time we use the packages available with the the Java API. Following figure shows the system packages that are frequently used in the programs.



Java System Packages and Their Classes

java.lang	Language support classes. They include classes for primitive types, string, math functions, thread and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, data, etc.
java.io	Input/output support classes. They provide facilities for the input and output of data.
java.applet	Classes for creating and implementing applets.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

Creating and Accessing Packages

Creating a Package

We can create our own package by creating our own classes and interfaces together. The package statement should be declared at the beginning of the program.

Syntax:

```
package <packagename>;
class ClassName
{
……..
……..
}
```

Example: Creating a Package

```
// Demo.java
package p1;
public class Demo
{
  public void m1()
  {
     System.out.println("Method m1..");
  }
}
```

Here the package name is p1.The class Demo is now considered as a part of this package p1.

This program would be saved as Demo.java and located in the directory named as p1.
When the source file is compiled, java will create a .class file and store it in the same directory.

The .class file will be in a directory that has the same name as package and this directory should be a subdirectory of the directory where classes that will import the package is located.

Steps to create user defined package:

1. Declare the package at the beginning of a file using the form
   package packagename;
2. Define the class that is to be put in the package and declare it public.
3. Create a subdirectroy under the directory whre the main source files are stored.
4. Store the listing as the classname.java in the subdirectory created.
5. complie the file. This creates .class file in the subdirectory.

Accessing a package

The import keyword provides the access to other package classes and interfaces in current packages.
"import" keyword is used to import built-in and user defined packages in java program**.**
There are different 3 ways to access a package from other packages.

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If we use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
The import keyword is used to make the classes and interface of another package accessible to the current package.

Example

```
import java.util.*;
class Demo
{
   // statements
}
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example

```
import java.util.Scanner;
class Demo
```

```
{
     // statements
}
```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example

```
class Demo extends java.util.Scanner
{
   //statements
}
```

Example of package that import the packagename.*

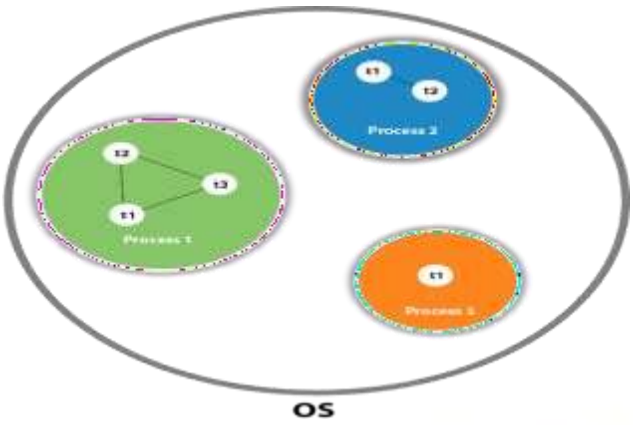
```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
Output:Hello
```

Multithreaded Programming

Thread

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



Multithreading

- The process of executing multiple threads simultaneously is known as **multithreading**.
- Multithreading means executing two parts of the same program concurrently.
- It is a part of multitasking.
- A program may contain more than one thread.

For Example: A browser can serve a web page while downloading a file. Each of these actions is a thread.

Life Cycle of Thread

(diagram from book)

New State: Soon after the creation of a thread, it is said to be in the new state.

Runnable State: The life of a thread starts after invoking the start () method. The start() method invokes the run () method.

Running State: When the thread is currently running, it is in the state of running.

Blocked State: When other threads are holding the resources, the current thread is said to be in wait state

Dead State: When the thread has completed the execution of run () method, it is said to be in dead state

Creating Thread

There are two ways to create a thread in Java.
1. Extending the thread class.
2. Implementing runnable interface.

1. Extending Thread Class

- Declare the class as extending the **Thread** class
- Implement the run() method that is responsible for executing the sequence of code that the thread will execute
- create a thread object and call start() method to initiate the thread execution

1. Declaring the class
The Thread class can be extended as:

```
class  Mythread  extends  Thread
{
.....
.....
.....
}
```

2. Implementing the run() Method

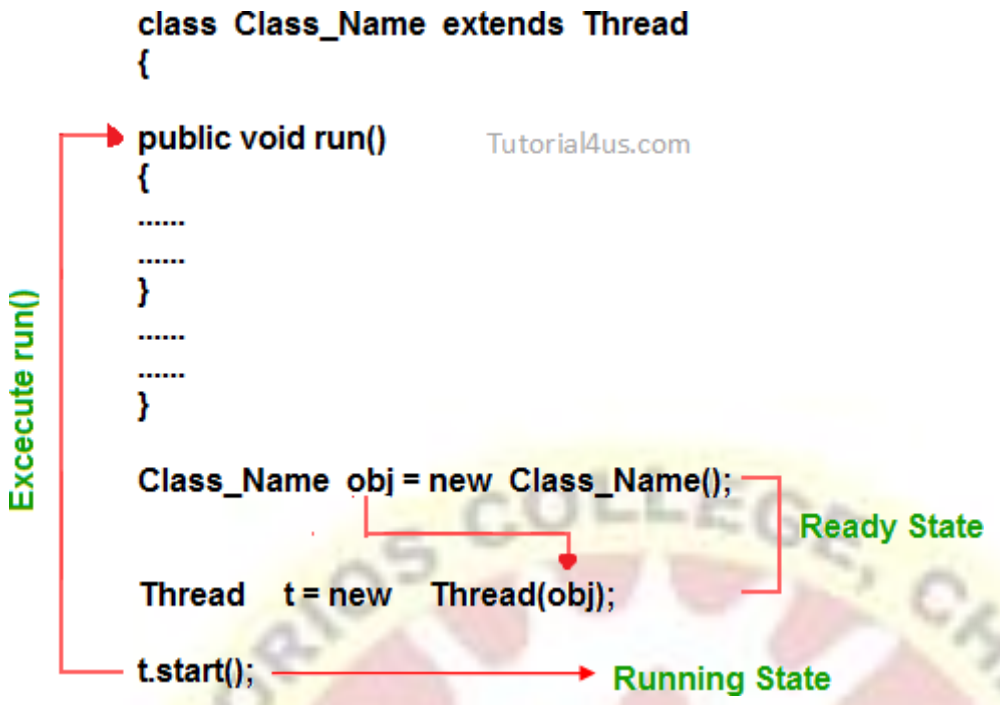
1. The run() method has been inherithed by the class Mythread.We have to Override this method in order to implement the code to be executed by our thread class .
   The run() will look as follows:

```
public void run()
{
  .....................
 .....................     //thread code
}
```

3. Starting New Thread

1. To Create and run an Instance of out thread class.
2. ie Mythread obj=new Mythread ();
   obj . start( );
   Call start() method of thread class to execute run() method.

All the 3 steps of **Extending Thread Class is shown in the figure as follows:**



Example:

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

Output: thread is running...

2. Implementing Runnable Interface

Runnable interface is available in java.lang package.

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

To implement Runnable interface, a class need only implement a single method called run( ), which is declared like this:

public void run( );

We must perform the following steps:

1. Declare the class as implementing the Runnable interface;
2. Implement the run() method.
3. Create a thread by definig an object that is instantiated from the runnable class as the target of the thres.
4. Call the thread's start( ) method to run the thread;

Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output:thread is running...

Thread Method

Following are the some important methods available in thread class.

Methods	Description
public void start( )	This method invokes the **run()** method, which causes the thread to begin its execution.
public void run( )	Contains the actual functionality of a thread and is usually invoked by **start()** method.
public static void	Blocks currently running thread for at least certain

sleep (long millisec)	millisecond.
public static void yield ( )	Causes the current running thread to halt temporarily and allow other threads to execute.
public static Thread currentThread ( )	It returns the reference to the current running thread object.
public final void setPriority (int priority)	Changes the priority of the current thread. The minimum priority number is 1 and the maximum priority number is 10.
public final boolean isAlive ( )	Used to know whether a thread is live or not. It returns true if the thread is alive. A thread is said to be alive if it has been started but has not yet died.

Thread Priorities

Every thread in Java has a priority that helps the thread scheduler to determine the order in which threads are scheduled. The threads with higher priority will usually run before and/or more frequently than lower priority threads.

Java permits us to set priority of a thread using the setPriority() method.

Syntax: Threadname. setPriority(int number);

Example:
Thread t=new Thread();
t.setPriority(any priority number between o to 10)
or
t.setPriority(Thread.MAX-PRIORITY)

Thread priorities is the integer number from 1 to 10 which helps to determine the order in which threads are to be scheduled. It decides when to switch from one running thread to another thread.

Priority range of **MIN_PRIORITY** is 1 and **MAX_PRIORITY** is 10. Default value of **NORMAL_PRIORITY** is 5

Syntax:
public static final int MAX-PRIORITY=10
MIN-PRIORITY,Which represents the minimum priority that a thread can have.

Syntax:
public static final int MIN-PRIORITY=0

NORM-PRIORITY,Which represent the default priority that is assigned to a thread.

Syntax:
public static final int NORM-PRIORITY=5

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread
{
 public void run()
{
   System.out.println("running thread name
is:"+Thread.currentThread().getName());
   System.out.println("running thread priority
is:"+Thread.currentThread().getPriority());

  }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
```

Output:running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1

Synchronization

At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**.

Definition: Allowing only one thread at a time to utilized the same resource out of multiple threads is known as thread synchronization

In java language thread synchronization can be achieve in two different ways.

1. Synchronized block
2. Synchronized method

Note: synchronization is a keyword

1. Synchronized block

Whenever we want to execute one or more than one statement by a single thread at a time(not allowing other thread until thread one execution is completed) than those statement should be placed in side synchronized block.

```
class  Class_Name  implement Runnable [extends Thread]
{
public void run()
{
synchronized(this)
{
.......
}
}
}
```

2. Synchronized block

Whenever we want to execute one or more than one statement by a single thread at a time(not allowing other thread until thread one execution is completed) than those statement should be placed in side synchronized block.

```
class  Class_Name  implement Runnable [ extends Thread]
{
public void run()
{
synchronized(this)
{
.......}}}
```

Unit-IV

Unit-4: Managing Errors and Exceptions-Syntax of Exception Handling Code-Using Finally Statement-Throwing Our Own Exceptions-Applet Programming-Applet Life Cycle-Graphics Programming-Managing Input/Output Files: Concept of Streams-Stream Classes-Byte Stream Classes-Character Stream Classes – Using Streams-Using the File Class-Creation of Files-Random Access Files-Other Stream Classes.

Managing Errors and Exceptions

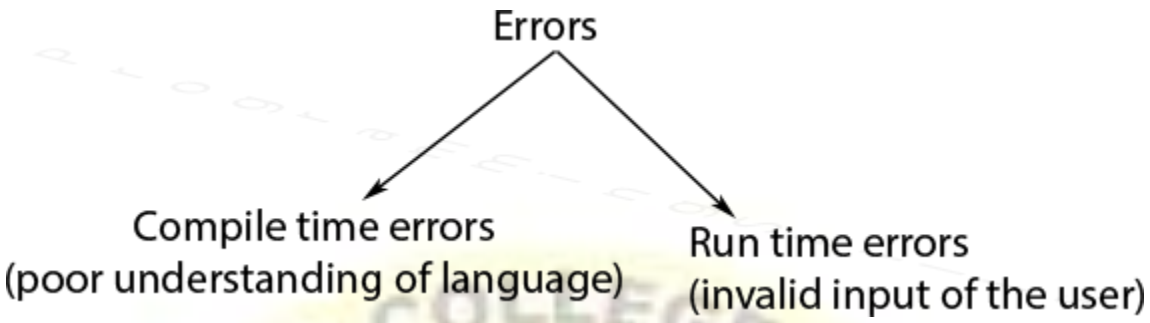
In general, the errors in programs can be categorized into three types:

1. **compile time:** Also called as Syntax errors when a program violates the rules of the programming language.

   Ex: missing a semi colon, typing a spelling mistake in keyword etc.

2. **Run-time errors:** Errors which are raised during execution of the program .

   Ex: dividing a number with zero, stack overflow, illegal type conversion, accessing an unavailable array element etc.



   Logic errors: Errors which are raised during the execution of the program due to mistakes in the logic applied in the program. These are the most severe kind of errors to detect.

   Ex: misplacing a minus sign in the place of plus sign, placing a semi colon at the end of loops etc.

Exception: An abnormal condition that occurs at run-time or a run-time error is known as an exception.

Exception Handling: Handling exceptions is known as exception handling. Instead of letting the program to terminate abnormally when an error occurs at run-time, we provide meaningful error messages to the users by handling the exception(s).

Exception Handling Fundamentals

In Java, abnormal conditions that occur in programs are known as exceptions. Java exceptions are objects which are *thrown* when an exception raises. Exceptions which are raised must be handled (caught) at some point.

There are five keywords used in Java for exception handling. They are:

i. try
ii. catch
iii. throw
iv. throws
v. finally

1. try: Statements that are supposed to raise exception(s) are placed inside

a *try* block.

2. catch: Code that handles exceptions are placed in *catch*blocks. A *try* block must be followed by one or more *catch*blocks or a single *finally* block.

3. throw: Most of the exceptions are thrown automatically by the Java run-time system. We can throw exceptions manually using *throw* keyword.

4. throws: If a method which raises exceptions doesn't want to handle them, they can be thrown to parent method or the Java run-time using the *throws* keyword.

5. finally: All the statements that should execute irrespective of whether a exception arises or not is placed in the *finally* block.

Types of Exceptions:

The exceptions are categorized into

1. **pre-defined exceptions (system exceptions)**
2. **user-defined exceptions**.

All exception classes available with Java are known as pre-defined exceptions or system exceptions. Exceptions that created by the programmer are known as user-defined exceptions.

The general form of an exception handling block looks as follows:

Syntax

```
try
{

    //code that cause exception;
}
catch(Exception_type  e)
{
    //exception handling code
}
```

Example : An examples to uses try and catch block in java

```
class Divide
{
      public static void main(String[] args)
      {
```

```
            try
            {
                  int a = 10, b = 0;
                  int c = a / b;
                  System.out.println("Result is: " + c);
            }
            catch(ArithmeticException e)
            {
                  System.out.println("b cannot be zero");
            }
      }
}
```

Output of above program is: b cannot be zero

Multiple catch blocks

If more than one exception is generated by the program then it uses multiple catch blocks. A single try block can contain multiple catch blocks.

Syntax

```
try
{
    // code which generate exception
}
```

```
catch(Exception_type1 e1)
{
    //exception handling code
}
catch(Exception_type2 e2)
{
    //exception handling code
}
```

Example : A program illustrating multiple catch block using command line argument

```
public class MultipleCatch
{
    public static void main(String args[])
    {
        try
        {
            int a = Integer.parseInt(args[0]);
            int b = Integer.parseInt(args[1]);
            int c = a / b;
            System.out.println("Result = "+c);
        }
```

```
            catch(ArithmeticException ae)
            {
                System.out.println("Enter second value except zero.");
            }
            catch (ArrayIndexOutOfBoundsException ai)
            {
                 System.out.println("Enter at least two numbers.");
            }
            catch(NumberFormatException npe)
            {
                  System.out.println("Enter only numeric value.");
            }
    }
}
```

Output:

```
10      5
Result = 2
10      0
Enter second value except zero.

5
Enter at least two numbers.

10      a
Enter only numeric value.
```

Using finally Statement

- The code present in finally block will always be executed even if try block generates some exception.
- Finally block must be followed by try or catch block.
- It is used to execute some important code.
- Finally block executes after try and catch block.

Syntax

```
try
{
    // code
}
catch(Exception_type1)
{
    // catch block1
}
Catch(Exception_type2)
{
```

```
    //catch block 2
}
finally
{
    //finally block
    //always execute
}
```

Example :

```
class FinallyTest
{
    public static void main(String args[])
    {
        int arr[] = new int[5];
        try
        {
            arr[7] = 10;
            System.out.println("Seventh element value: " + arr[7]);
        }
        catch(ArrayIndexOutOfBoundsException ai)
        {
            System.out.println("Exception thrown : " + ai);
        }
        finally
        {
            arr[0] = 5;
            System.out.println("First element value: " +arr[0]);
            System.out.println("The finally statement is executed");
        }
    }
}
```

Output:
Exception thrown: **java.lang.ArrayIndexOutOfBoundsException:** 7
First element value: 5
The finally statement is executed

Throwing Our Own Exceptions

The throw keyword in Java is used to explicitly throw our own exception.

Syntax:

throw new Throwable_subclass;

Example :

```
import java.lang.Exception;
class InvalidAgeException extends Exception
{
 InvalidAgeException(String s)
{
  super(s);
 }
}
class TestCustomException1{

   static void validate(int age)throws InvalidAgeException{
     if(age<18)
      throw new InvalidAgeException("not valid");
     else
      System.out.println("welcome to vote");
   }

   public static void main(String args[]){
      try{
      validate(13);
      }catch(Exception m){System.out.println("Exception occured: "+m);}

      System.out.println("rest of the code...");
  }
}
```

```
Output:Exception occured: InvalidAgeException:not valid
       rest of the code...
```

Applet Programming

- Applets are small Java applications that can be accessed on an Internet server, transported over Internet, and can be automatically installed and run as a part of a web document.
- After a user receives an applet, the applet can produce a graphical user interface.
- Any applet in Java is a class that extends the java.applet.Applet class.
- An Applet class does not have any main() method.
- It is viewed using JVM. The JVM can use either a plug-in of the Web browser or a separate runtime environment to run an applet application.
- JVM creates an instance of the applet class and invokes **init()** method to initialize an Applet.

Advantages of Applets

- Applets are supported by most web browser.
- Applets have very less response time, because it works at client side.
- Applets are quite secure because of their access of resources.
- An untrusted applet has no access to local machine.
- The size of applets is small, making it easier to transfer over network.

Limitations of Applets

- Applets require a Java plug-in to execute.
- Applets do not support file system.
- Applet itself cannot run or modify any application on the local system.
- It cannot work with native methods of system.
- An applet cannot access the client-side resources.

Difference between Java Applications and Applets

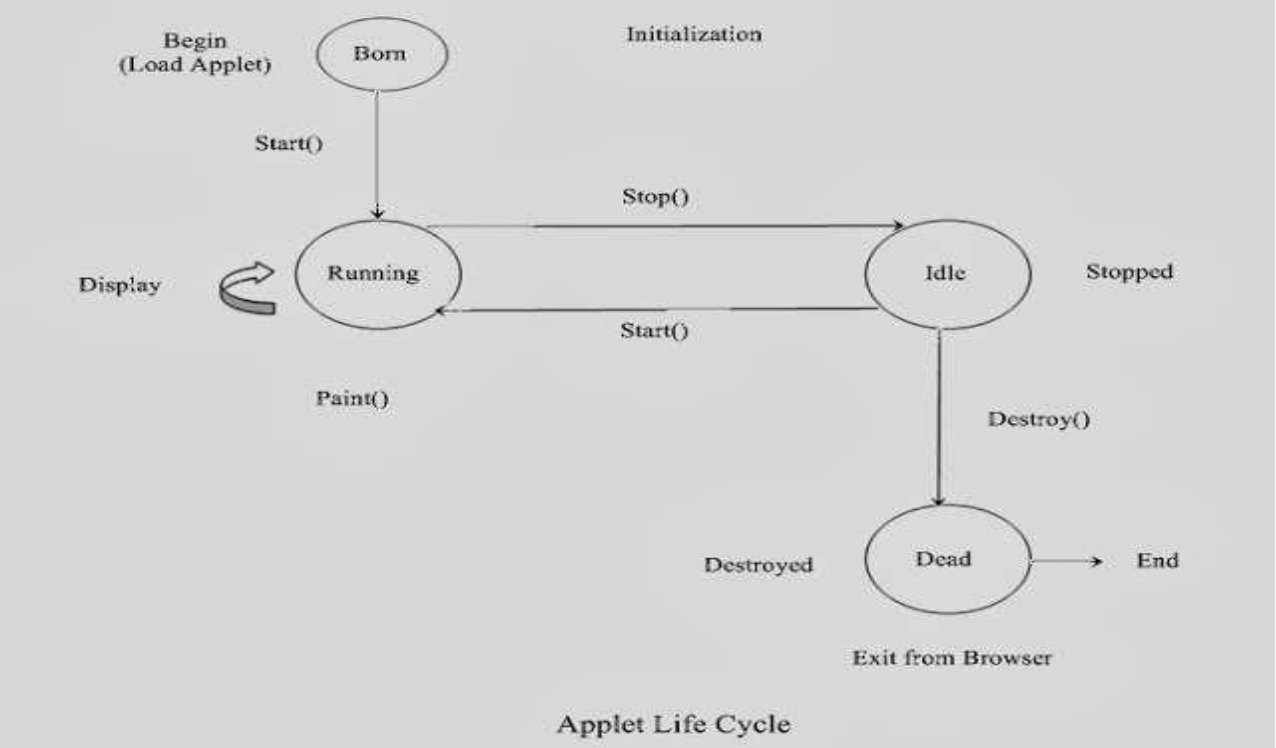
Applications	Applets
An application runs stand-alone.	An applet program runs under the control of browser.
It can access any data or software available on the system.	It cannot access anything on the system except browser's service.
Execution start from the main() method.	Execution start from the init() method because main() is not available.
It can read and write to the file system.	Cannot read and write to the file systems.
Does not provide any security.	An applet provides the security features.
Application run using Java interpreter.	It runs using applet viewer or on web browser.

Applet Life Cycle

Every Java applet inherits a set of default behaviours from the Applet class. When an Applet is created, it goes through different stages; it is known as applet life cycle. The Applets states include:

1. Born or initialization state
2. Running state
3. Stopped state
4. Destroyed state
5. Display state

The Applet undergoes a series of changes in its states as shown in the following figure as follows



An Applet's State Transition diagram

1.Initialization State

When a new *applet* is born or created, it is activated by calling *init()* method.

At this stage, We do the following ,if required.

- new objects to the *applet* are created
- initial values are set
- images or fonts loaded
- setup colors of the images
- An *applet* is initialized only once in its lifetime. We must override the init( ) method to provide any behavior mentioned above.

It's general syntax is:

```
public void init()
 {
    ..........................
    ..........................//Action to be performed
    ..........................
}
```

2.Running State

An *applet* enters the running state when the system calls the *start()* method. This occurs as soon as the *applet* is initialized. An *applet* may also start when it is in idle state. At that time, the *start()* method is overridden.

It's general syntax is:

```
public void start()
{
   ..........................
    ..........................//Action to be performed
    ..........................
}
```

3.Idle or Stopped State

An *applet becomes* in idle state when its execution has been stopped either *implicitly or explicitly*. An *applet* is *implicitly* stopped when we leave the page containing the currently running applet. An *applet* is *explicitly* stopped when we call *stop()* method to stop its execution.

It's general syntax is:

```
public void stop()
{
   ..........................
    ..........................//Action to be performed
    ..........................}
```

4.Dead State

An *applet* is in dead state when it has been removed from the memory. It destroys and removes the applet from the memory. It is invoked only once. This is end of the life cycle of applet and it becomes dead. This can be done by using *destroy()* method.

It's general form is:

```
public void destroyed()
 {
   ..........................
    ..........................//Action to be performed
    ..........................
}
```

Display State

Applet moves to display state whenver it has to perform some otput operations on the screen. The *paint()* method is called to perform the tasks like drawing, writing and creating colored backgrounds of the applet. It takes an argument of the graphics class. To use The graphics, it imports the package java.awt.Graphic.

It's general syntax is:

```
public void paint(Graphics g)
{
   ..........................
    ..........................//Display statements
    ..........................
}
```

where Graphics class contains the member functions that can be used to display the output to the browser.

Example to display a text String, the drawString() method is used.
g.drawString(String s, int x, int y)

where x and y are the coordinate positions on the screen and s is the string to be output.

Steps to Develop and Run an Applet Program

Developing an Applet Program

Steps to develop an applet program
1. Create an applet (java) code i.e. .java file.
2. Compile to generate .class file.
3. Design a HTML file with <APPLET> tag.

Running the Applet Program

There are two ways to run the applet program
1. By HTML
2. Appletviewer tool

Creating an Executable Applet

Executable applet is nothing but the .class file of applet, which is obtained by compiling the source code of the applet. Compiling the applet is exactly the same as compiling an application using following command.

javac appletname.java

The compiled output file called appletname.class should be placed in the same directory as the source file.

Designing a Web Page

Java applets are programs that reside on web page. A web page is basically made up of text and HTML tags that can be interpreted by a web browser or an applet viewer. Like Java source code, it can be prepared using any ASCII text editor. A web page is also called HTML page or HTML document. Web pages are stored using a file extension .html such as my Applet.html. Such files are referred to as HTML files. HTML files should be stored in the same directory as the compiled code of the applets.

A web page is marked by an opening HTML tag <HTML> and closing HTML tag </HTML> and is divided into the following three major parts:

1) Comment Section
2) Head Section
3) Body Section

Comment Section

This is very first section of any web page containing the comments about the web page functionality. It is important to include comments that tell us what is going on in the web page. A comment line begins with <! And

ends with a > and the web browsers will ignore the tesxt enclosed between them. The comments are optional and can be included anywhere in the web page.

Head Section

This section contains title, heading and sub heading of the web page. The head section is defined with a starting <Head> tag and closing </Head> tag.

```
<Head>
<Title> Welcome to Java Applet </Title>
</Head>
```

Body Section

After the Head Section the body section comes. This is called as body section because the entire information and behaviour of the web page is contained in it, It defines what the data placed and where on to the screen. It describes the color, location, sound etc. of the data or information that is going to be placed in the web page.

```
<Body>
<Center>
<H1>
Welcome to Java >
</H1>
</Center>
<BR>
<APPLET - - ->
</APPLET>
</Body>
```

Applet Tag

The <Applet...< tag supplies the name of the applet to be loaded and tells the browser how much space the applet requires. The ellipsis in the tag <Applet...> indicates that it contains certain attributes that must specified. The <Applet> tag given below specifies the minimum requirements to place the Hellojava applet on a web page.

```
<Applet
code = Hellojava.class
width = 400
Height = 200 >
</Applet>
```

This HTML code tells the browser to load the compiled java applet Hellojava.class, which is in the same directory as this HTML file. It also specifies the display area for the applet output as 400 pixels width and 200 pixels height. We can make this display area appear in the center of screen by using the CENTER tags as showsn below:

```
<CENTER>
<Applet>
----------------
----------------
----------------
</Applet>
</CENTER>
```

The applet tag discussed above specified the three things:

1) Name of the applet
2) Width of the applet (in pixels)
3) Height of the applet (in pixels)

Adding Applet to HTML File

To execute an applet in a web browser, you need to write a short HTML text file that contains the appropriate APPLET tag. Here is the HTML file that executes **SimpleApplet**:

```
<Applet code = Hellojava.class width = 400 Height = 200 >
</Applet>
```

The width and height attributes specify the dimensions of the display area used by the applet. After you create this file, you can execute the HTML file called RunApp.html (say) on the command line.

```
c:\>appletviewer RunApp.html
```

Running the Applet

To execute an applet with an applet viewer, you may also execute the HTML file in which it is enclosed, eg.

```
c:\>appletviewer RunApp.html
```

Execute the applet the applet viewer, specifying the name of your applet's source file. The applet viewer will encounter the applet tag within the comment and execute your applet.

First Java Applet Program

The Following will be saved with named FirstJavaApplet.java:

```
import java.awt.*;
import java.applet.*;
public class FirstJavaApplet extends Applet
{
	public void paint(Graphics g)
	{
		g.drawString("My First Java Applet Program",100,100);
	}
}
```

After creating FirstJavaApplet.java file now you need create FirstJavaApplet.html file as shown below:

```
<HTML>
<HEAD>
 <TITLE>My First Java Applet Program</TITLE>
</HEAD>
<BODY>
 <APPLET Code="FirstJavaApplet.class" Width=300 Height=200>
 </APPLET>
</BODY>
</HTML>
```

After this you can compile your java applet program as shown below:

```
C:\>cd jdk1.4
C:\jdk1.4>cd bin
C:\jdk1.4\bin>edit FirstJavaApplet.java
C:\jdk1.4\bin>javac FirstJavaApplet.java
C:\jdk1.4\bin>appletviewer FirstJavaApplet.html
```

Output of FirstJavaApplet.class

Java Graphics Programming

Graphics is one of the most important features of Java.

Java applets can be written to draw lines, arcs, figures, images and text in different fonts and styles. Different colors can also be incorporated in display.

Every applet has its own area of the screen known as canvas, where it creates its display.

The size of an applet space is decided by the attributes of the <APPLET> tag.

The Graphics Class

The graphics class defines a number of drawing functions, Each shape can be drawn edge-only or filled. To draw shapes on the screen, we may call one of the methods available in the graphics class. The most commonly used drawing methods included in the graphics class are listed below. To draw a shape, we only need to use the appropriate method with the required arguments.

Drawing Methods of the Graphics Class

Sr.No	Method	Description
1.	clearRect()	Erase a rectangular area of the canvas.
2.	copyAre()	Copies a rectangular area of the canvas to another area
3.	drawArc()	Draws a hollow arc
4.	drawLine()	Draws a straight line
5	drawOval()	Draws a hollow oval
6	drawPolygon()	Draws a hollow polygon
7	drawRect()	Draws a hollow rectangle
8.	drawRoundRect()	Draws a hollow rectangle with rounded corners
9.	drawString()	Display a text string
10.	FillArc()	Draws a filled arc
11.	fillOval()	Draws a filled Oval
12.	fillPolygon()	Draws a filled Polygon
13.	fillRect()	Draws a filled rectangle
14.	fillRoundRect()	Draws a filled rectangle with rounded corners
15.	getColor()	Retrieves the current drawing color

16.	getFont()	Retrieves the currently used font
17.	getFontMetrics()	Retrieves the information about the current font
18.	setColor()	Sets the drawing color
19.	setFont()	Sets the font

Lines and Rectangles

Lines are drawn by means of the drawLine() method.

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

drawLine() displays a line in the current drawing color that begins at (start X, start Y) and ends at (endX, end Y).

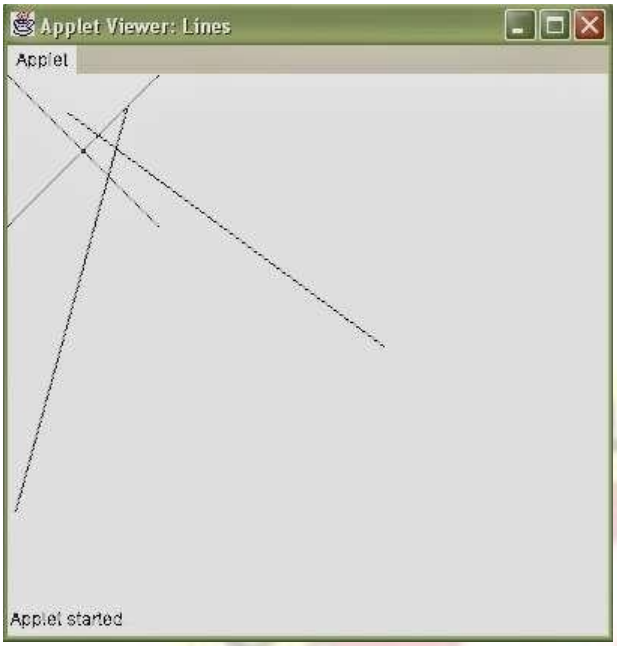
Example:

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

After this you can compile your java applet program as shown below:

```
C:\jdk1.4\bin>edit Lines.java
C:\jdk1.4\bin>javac Lines.java
C:\jdk1.4\bin>appletviewer Lines.java
```

Output:



Rectangle

The drawRect() and fillRect() methods display an outlined and filled rectangle, respectively.

Syntax

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

The upper-left corner of the rectangle is at(top,left). The dimensions of the rectangle are specified by width and height.

Use drawRoundRect() or fillRoundRect() to draw a rounded rectangle. A rounded rectangle has rounded corners.

Syntax

void drawRoundRect(int top, int left, int width, int height int Xdiam, int YDiam)
void fillRoundRect(int top, int left, int width, int height int Xdiam, int YDiam)

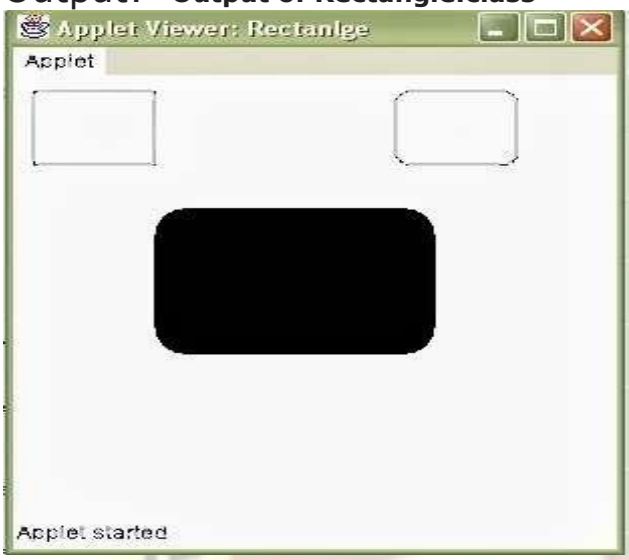
The upper-left corner of the rounded rectangle is at (top,left). The dimensions of the rectangle are specified by width and height. The diameter of the rounding are along the X axis are specified by x Diam. The diameter of the rounding are along the Y axis is specified by Y Diam.

Example:

```
import java.awt.*;
import java.applet.*;
/* <applet code="Rectanlge" width=300 Height=300></applet>*/
public class Rectanlge extends Applet
{
```

```
        public void paint(Graphics g)
        {
                g.drawRect(10,10,60,50);
                g.drawRoundRect(190,10,60,50,15,15);
                g.fillRoundRect(70,90,140,100,30,40);
        }
}
```

Output: **"Output of Rectangle.class"**



Circles and Ellipses

The Graphics class does not contain any method for circles or ellipses. To draw an ellipse, use drawOval(). To fill an ellipse, use fillOval().

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

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by (top,left) and whose width and height are specified by width and height. To draw a circle, specify a square as the bounding rectangle i.e get height = width.

The following program draws serveral ellipses:

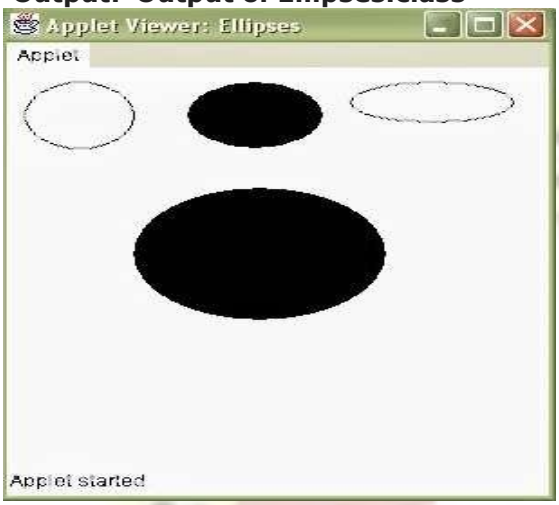
```
import java.awt.*;
import java.applet.*;
/*<applet code="Ellipses" width=300 Height=300></applet>*/
public class Ellipses extends Applet
{
        public void paint(Graphics g)
        {
                g.drawOval(10,10,60,50);
                g.fillOval(100,10,75,50);
                g.drawOval(190,10,90,30);
```

```
            g.fillOval(70,90,140,100);
        }
}
```

After this you can compile your java applet program as shown below:

```
c:\jdk1.4\bin\javac Ellipses.java
c:\jdk1.4\bin\appletviewer Ellipses.java
```

Output:"Output of Ellipses.class"



Drawing Arcs

An arc is a part of oval. Arcs can be drawn with draw Arc() and fillArc() methods.

Syntax

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

The arc is bounded by the rectangle whose upper-left corner is specified by (top,left) and whose width and height are specified by width and height. The arc is drawn from startAngle through the angular distance specified by sweepAngle. Angles are specified in degree. '0º' is on the horzontal, at the 30' clock's position. The arc is drawn conterclockwise if sweepAngle is positive, and clockwise if sweetAngle is negative.

The following applet draws several arcs:

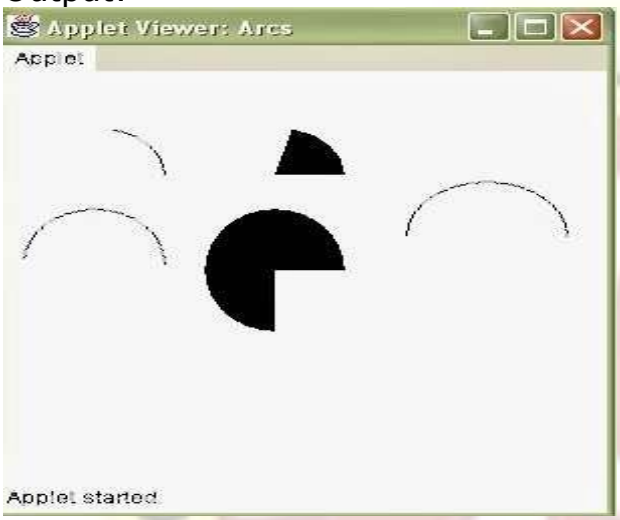
```
import java.awt.*;
import java.applet.*;
/* <applet code="Arcs" width=300 Height=300></applet>*/
public class Arcs extends Applet
{
        public void paint(Graphics g)
        {
```

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

After this you can comiple your java applet program as shown below:

```
c:\jdk1.4\bin\javac Arcs.java
c:\jdk1.4\bin\appletviewer Arcs.java
```

Output:



Drawing Polygons

Polygons are shapes with many sides. It may be considered a set of lines connected together. The end of the first line is the beginning of the second line, the end of the second line is the beginning of the third line, and so on. Use drawPolygon() and fillPolygon() to draw arbitrarily shaped figures.

Syntax

```
void drawPolygon(int ,[ ] , int y[ ], int numPoints)
void fillPolygon(int ,[ ] , int y[ ], int numPoints)
```

The polygon's endpoints are specified by the coordinate pairs contained within the x and y arrays. The number of points defined by x and y is specified by numPoints.

It is obvious that x and y arrays should be of the same size and we must repeat the first point at the end of the array for closing the polygon.

The following appplet draws an hourglass shape:

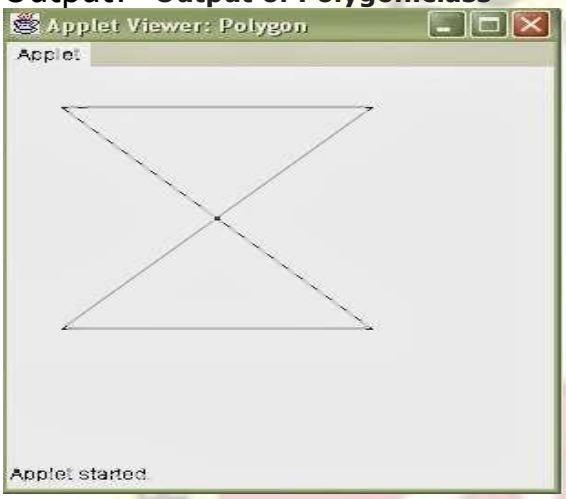
```
import java.awt.*;
import java.applet.*;
/*<applet code="Polygon" width=300 Height=300></applet>*/
public class Polygon extends Applet
{
```

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

After this you can comiple your java applet program as shown below:

```
c:\jdk1.4\bin\javac Polygon.java
c:\jdk1.4\bin\appletviewer Polygon.java
```

Output: **"Output of Polygon.class"**



University question:

We can draw Polygon in java applet by three ways :

Polygon are shapes with many sides.

1. drawLine(int x, int y, int x1, int y1) : In this method we would connect adjacent vertices with a line segment and also connect the first and last vertex.

A polygon may be considered a set of lines connected together. The end of the first line is the beginning of the second line, the end of the second is the begining of the third and so on. We can draw a polygon with n sides using the drawLine ( ) method n times in succession.

Example:

```
public void paint(Graphics g)
	{
		g.drawLine(10,20,170,40);
		g.drawLine(170,40,80,140);
		g.drawLine(80,140,10,20);
}
```

Output:

2. Draw polygon using drawPolygon( ) Method:

We can draw ploygon using drawPolygon( ) method of Graphics class. This method takes three arguments.

- An array of integers containing x coordinates
- An array of integers containing Y coordinates
- An integer for total number of points

The drawPolygon(int[ ] x, int[ ] y, int numberofpoints) : draws a polygon with the given set of x and y points.

Example:

```
 public void paint(Graphics g)
   {
      // x coordinates of vertices
      int x[ ] = { 10, 30, 40, 50, 110, 140 };
      // y coordinates of vertices
      int y[ ] = { 140, 110, 50, 40, 30, 10 };
      // number of vertices
      int numberofpoints = 6;
      // set the color of line drawn to blue
      g.setColor(Color.blue);
     // draw the polygon using drawPolygon function
      g.drawPolygon(x, y, numberofpoints);
   }
}
```

Output :

3. draws a polygon with the given object of Polygon class.

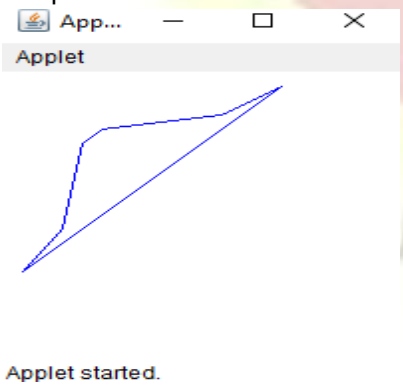
Another way of calling the methos of drawPolygon is to use a Polygon object. The polygon class enables us to treat the polygon as an object. It involves the following steps:

- Defining x coordinates values as an array
- Defining Y coordinates values as an array
- Defining number of points n
- Creating a Polygon object and initializing it with the above x,y,n values
- calling the method drawPolygon( ) or fillPolygon with the polygon object as an arguments.

Example: **drawPolygon(Polygon p)**

```
public void paint(Graphics g)
   {
      // x coordinates of vertices
      int x[ ] = { 10, 30, 40, 50, 110, 140 };
      // y coordinates of vertices
      int y[ ] = { 140, 110, 50, 40, 30, 10 };
      // number of vertices
      int numberofpoints = 6;
      // create a polygon with given x, y coordinates
      Polygon p = new Polygon(x, y, numberofpoints);
      // set the color of line drawn to blue
      g.setColor(Color.blue);
       // draw the polygon using drawPolygon
      // function using object of polygon class
      g.drawPolygon(p);
   }
}
```

Output:



Managing Input /Output Files

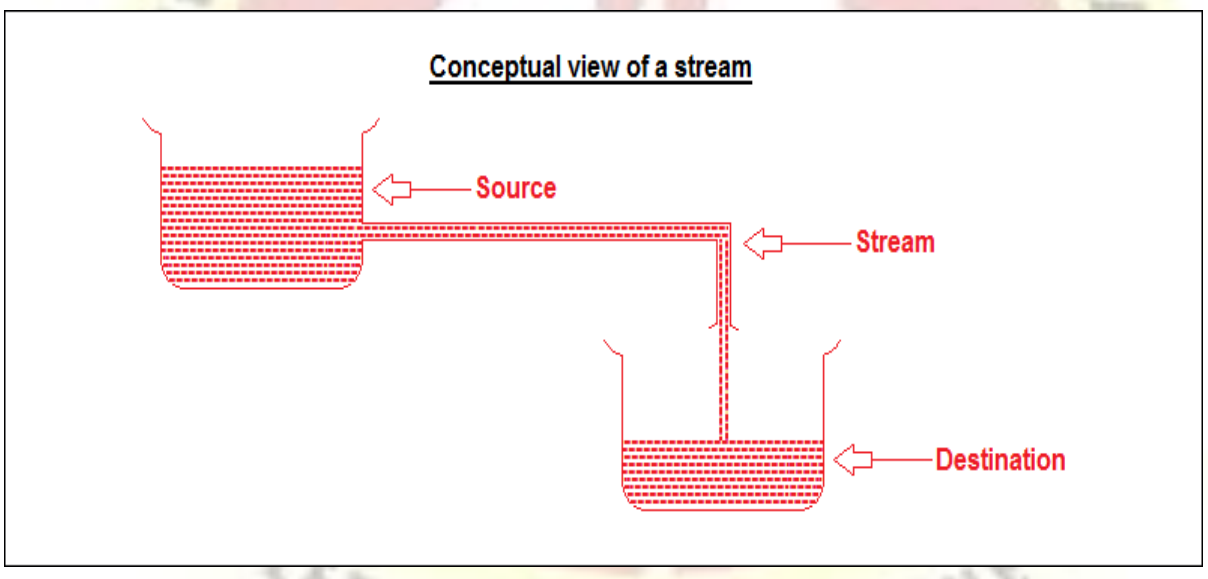
Input/output

Java I/O is a powerful concept, which provides the all input and output operations. Most of the classes of I/O streams are available in java.io package.

Streams

In Java, a **stream** is a **path** along which the **data flows**. Every stream has a **source** and a **destination**. We can build a complex file processing sequence using a series of simple stream operations. So Stream is the logical connection between Java program and file. In Java, stream is basically a sequence of bytes, which has a continuous flow between Java programs and data storage.

Two fundamental types of streams are **Writing streams** and **Reading streams**. While an **Writing streams** writes data into a **source**(file) , an **Reading streams** is used to read data from a **source**(file).

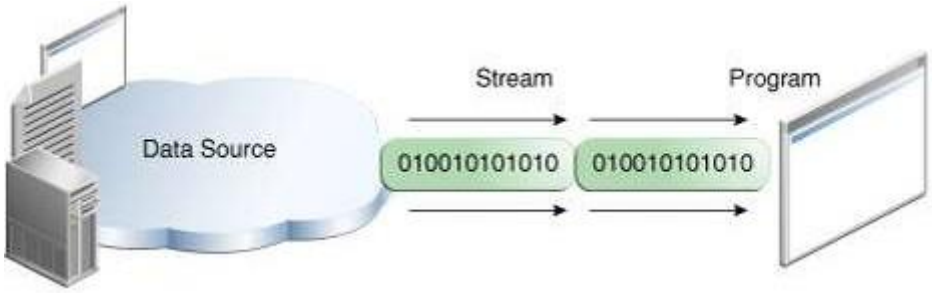


Types of Stream

Java Stream is basically divided into following types based on data flow direction.

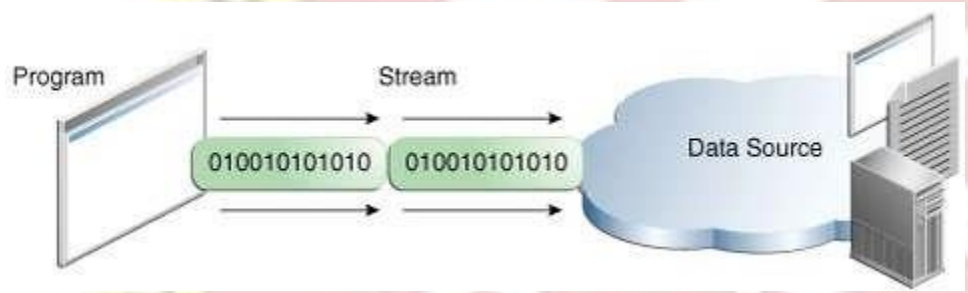
Input Stream

An Input stream extracts (i.e reads) data from the source(file) and sends it to the program. It is used to read the binary data from the source.



Output Stream

Output stream takes data from the program and sends(i.e writes) it to the destination(file). It is basically used to send out/write the data to destination.



Stream Classes

The java.io package contains a large number of stream classes that provide capabilities for processing all types of data. These classes may be categorized into two groups based on the data type on which they operate.

- **Byte stream classes**
- **Character stream classes**

Java encapsulates Stream under **java.io** package. Java defines two types of streams. They are,

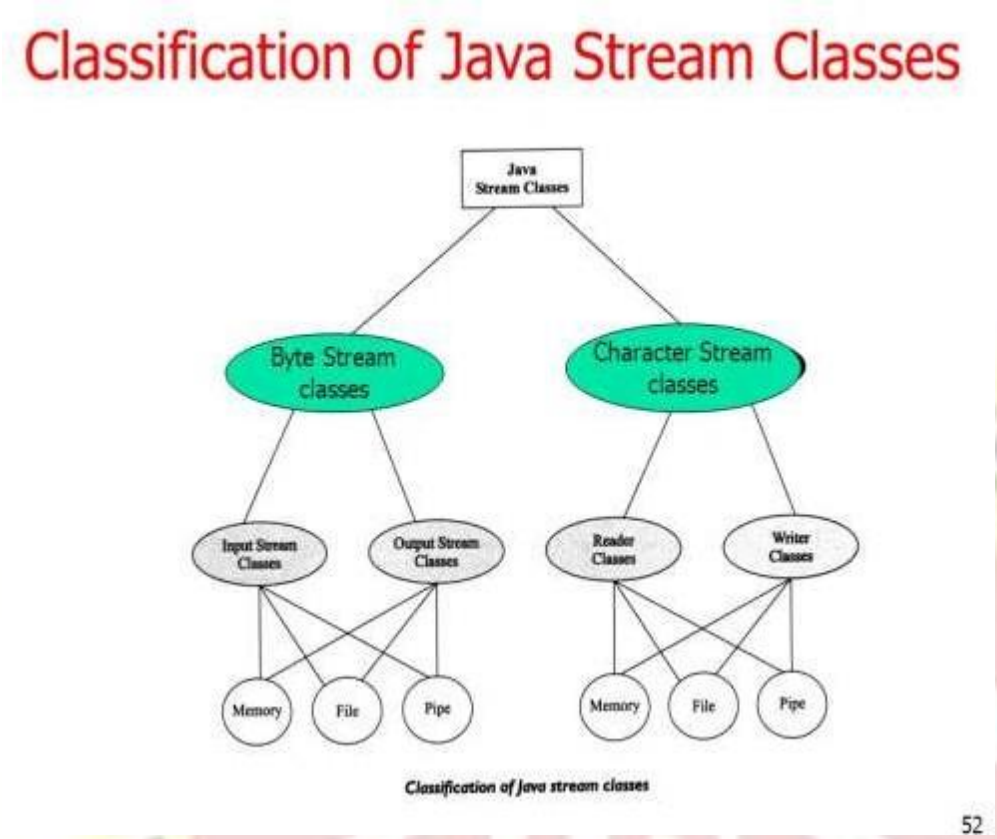
1.**Byte Stream :** It provides a convenient means for handling input and output of byte.
2.**Character Stream :** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.
These 2 groups are further be classified based on their purpose as shown in the following figure.

Byte Streams

Byte stream is used to input and output to perform 8-bits bytes. It has the classes like **FileInputStream** and **FileOutputStream.**

Character Streams

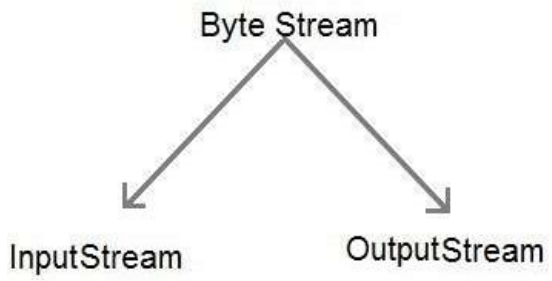
Character stream basically works on 16 bit-Unicode value convention. This stream is used to read and write data in the format of 16 bit Unicode characters



BYTE STREAM CLASSES

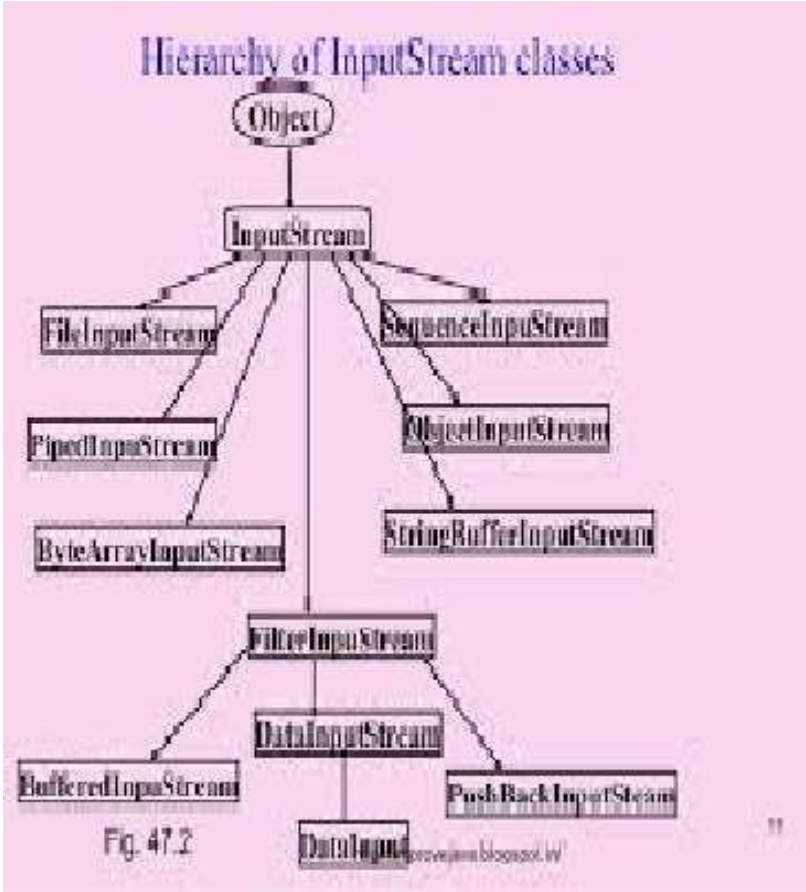
Byte Stream Classes

Byte stream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Java provides two kinds of byte stream classes: **input stream classes** and **output stream classes**.



INPUT STREAM CLASSES

Java's input stream classes are used to read 8-bit bytes from the stream. The InputStrearn class is the superclass for all byte-oriented input stream classes. All the methods of this class throw an IOException. Being an abstract class, the InputStrearn class cannot be instantiated hence, its subclasses are used.



Important Subclasses of InputStream Class

Name of the Class	Functionality
BufferedInputStream	Buffering input
LineNumberInputStream	Keeping track of how many lines are read
ByteArrayInputStream	Reading from an array
FileInputStream	Reading from a file
FilterInputStream	Filtering the input
PushbackInputStream	Pushing back a byte to the stream
PipedInputStream	Reading from a pipe
StringBufferInputStream	Reading from a String
DataInputStream	Reading primitive types

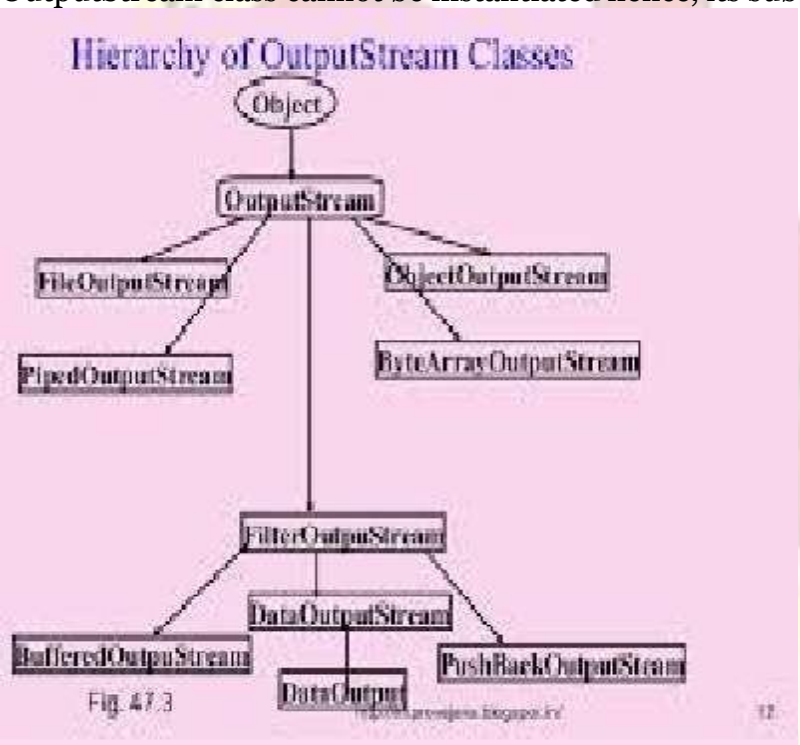
Important Methods in the InputStream Class

The Input Stream class defines various methods to perform reading operations on data of an input stream. Some of these methods along with their description are listed in Table

Methods	Usage
read( )	To read a byte from the input stream
read(byte b[ ])	To read an array of **b.length** bytes into array **b**
read(byte b[ ], int n, int m)	To read **m** bytes into **b** starting from **n'th** byte
close( )	To close the input stream

OUTPUT STREAM CLASSES

Java's output stream classes are used to write 8-bit bytes to a stream. The OutputStream class is the superclass for all byte-oriented output stream classes. All the methods of this class throw an IOException. Being an abstract class, the OutputStream class cannot be instantiated hence, its subclasses are used



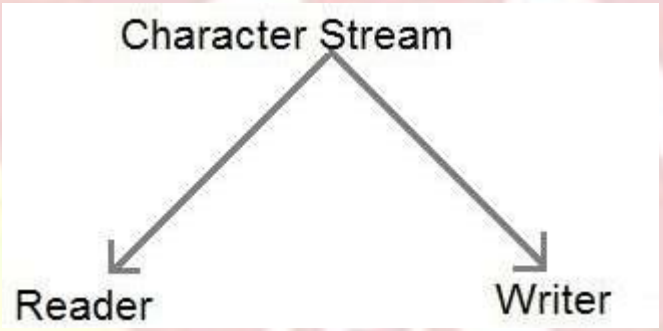
The OutputStream class defines methods to perform writing operations. These methods are discussed in Table

TableOutputStream Class Methods

.Method	**Description**
void write (int i)	writes a single byte to the output stream
void write (byte buffer [])	writes an array of bytes to the output stream
Void write(bytes buffer[],int loc, int nBytes)	writes 'nBytes' bytes to the output stream from the buffer b starting at buffer [loc]
void flush ()	Flushes the output stream and writes the waiting buffered output bytes
void close ()	closes the output stream. If an attempt is made to write even after closing the stream then it generates IOException

Character Stream Classes

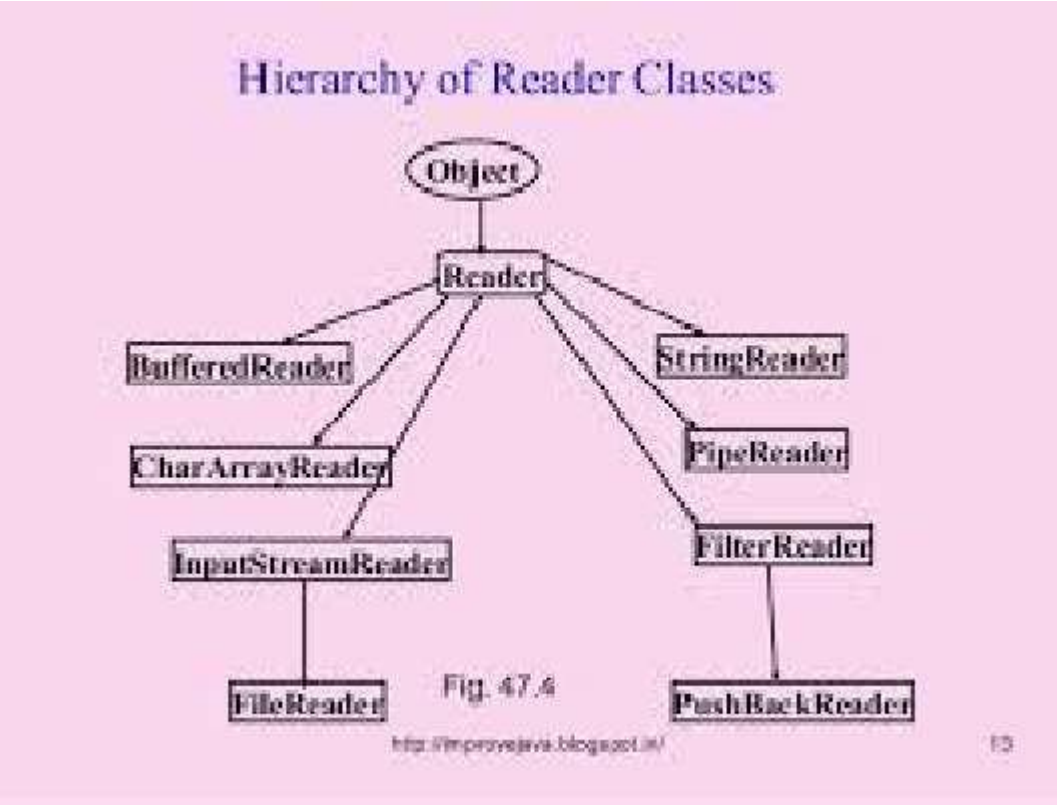
Character streams can be used to read and write 16-bit Unicode characters. Like byte streams, there are two kinds of character stream classes, namely, **reader stream classes**and **writer stream classes**.



READER STREAM CLASSES

Reader Stream Classes

Reader stream classes that are used to read characters include a super class known as **Reader** and a number of subclasses for supporting various input-related functions. Reader stream classes are functionally very similar to the input stream classes, except input streams use bytes as their fundamental unit of information, while reader streams use characters. The Reader class contains methods that are identical to those available in the InputStream class, except Reader is designed to handle characters. Therefore, reader classes can perform all the functions implemented by the input stream classes



Important Subclasses of Reader Class

Name of the Class	Functionality
BufferedReader	Buffering input
LineNumberReader	Keeping track of line numbers
CharArrayReader	Reading from an array
InputStreamReader	Translating byte stream into a character stream
FileReader	Reading from a file
FilterReader	Filtering the input
PushbackReader	Pushing back a character to the stream
PipedReader	Reading from a pipe
StringReader	Reading from a string

WRITER STREAM CLASSES

Writer Stream Classes

Like output stream classes, the writer stream classes are designed to perform all output operations on files. Only difference is that while output stream classes are designed to write bytes, the writer stream are designed to write character. The **Writer** class is an **abstract**class which acts as a base class for all the other writer stream classes. This

base class provides support for all output operations by defining methods that are identical to those in Outputstream class.

Important Subclasses of Writer Class

Name of the Class	Functionality
BufferedWriter	Buffering output
CharArrayWriter	Writing to an array
FilterWriter	Filtering the output
OutputStreamWriter	Translating character stream into a byte stream
FileWriter	Writing to a file
PrintWriter	Printing values and objects
PipedInputStream	Reading from a pipe
PipedWriter	Writing to a pipe
StringWriter	Writing to a String

Random Access Files

- It enables us to perform read and write operations on file.

- In this we use the file pointer to points the positions from where the reading or writing operation is performed.

- It defines operation modes (read / write).

1. r mode : Open for reading only.

2. rw mode : Open for reading and writing.

3. seek(file.length()) method : It jumps the file pointer at the specified location.

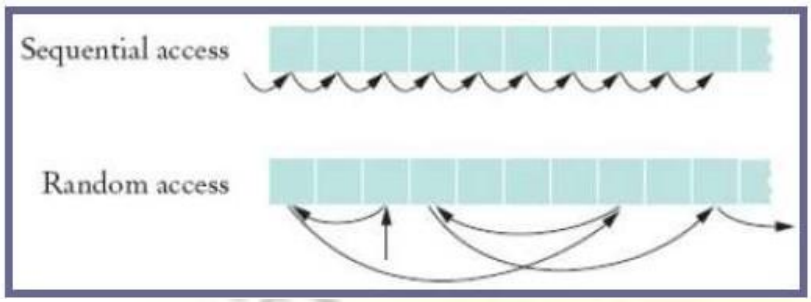
4. rand.close() method : It closes the random access file stream.

5. writeBytes( ) method : It simply writes the content into the file.

6. readByte() method : It reads a single byte from the file.

7. readLine() method : It reads the line from the file.

- Random access files are files in which records can be accessed in any order
  - Also called direct access files
  - More efficient than sequential access files



UNIT -V

Unit-5 : Network basics –socket programming – proxy servers – TCP/IP – Net Address – URL – Datagrams -Java Utility Classes-Introducing the AWT: Working with Windows, Graphics and Text- AWT Classes- Working with Frames-Working with Graphics-Working with Color-Working with Fonts-Using AWT Controls, Layout Managers and Menus.

Network basics

Networking is the concept of connecting multiple remote or local devices together. Java program communicates over the network at **application layer.**

All the Java networking classes and interfaces use java.net package. These classes and interfaces provide the functionality to develop system-independent network communication.

The java.net package provides the functionality for two common protocols:

TCP (Transmission Control Protocol)

TCP is a connection based protocol that provides a reliable flow of data between two devices. This protocol provides the reliable connections between two applications so that they can communicate easily. It is a connection based protocol.

UDP (User Datagram Protocol)

UDP protocol sends independent packets of data, called datagram from one computer to another with no guarantee of arrival. It is not connection based protocol.

Networking Terminology

i) Request and Response

When an input data is sent to an application via network, it is called **request**. The output data coming out from the application back to the client program is called **response**.

ii) Protocol

A **protocol** is basically a set of rules and guidelines which provides the instructions to send request and receive response over the network.

For example: TCP, UDP, SMTP, FTP etc.

iii) IP Address

IP Address stands for Internet protocol address. It is an identification number that is assigned to a node of a computer in the network.

For example: 192.168.2.01

Range of the IP Address

0.0.0.0 to 255.255.255.255

iv) Port Number

The **port number** is an identification number of server software. The port number is unique for different applications. It is a 32-bit positive integer number having between ranges 0 to 65535.

v) Socket

Socket is a listener through which computer can receive requests and responses. It is an endpoint of two way communication link. Every server or programs runs on the different computers that has a socket and is bound to the specific port number.

socket programming

Definiton: A socket is one endpoint of a two-way communication link between two programs running on the network.

Socket Programming is used to implement reliable, bidirectional , persistent ,point to point ,stream based connection between two computer on internet.

In Socket Programming , a Client program and a Server program establish a connection to one another. Each Program binds a socket to its end point of the connection.

To communicate , the client and the server each reads from and writes to the socket bound to the connection.

A server program creates a specific type of socket that is used to listen for client requests (server socket). In the case of a connection request, the program creates a new socket through which it will exchange data with the client using input and output streams. The socket abstraction is very similar to the file concept: developers have to open a socket, perform I/O, and close it. Figure 13.5 illustrates key steps involved in creating socket-based server and client programs.

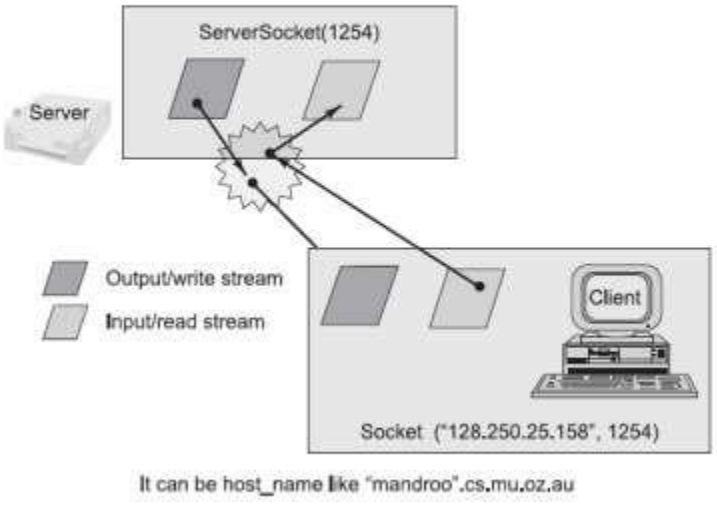


Fig. 13.5 Socket-based client and server programming

Steps for Server Program in Java

The steps for creating a simple server program are given in the following steps:

1. Open the Server Socket

ServerSocket server =new ServerSocket(PORT);

2.Wait for the Client Request

Socket client=server.accept();

where server is a ServerSocket object.

3. Establish the communication with the client.Create a I/O streams for communicating to the client.

DataInputStream is new DataInputstream(cilent,get.

InputStream());

DataOutputStream os new DataOutputstream(cilent,get. OutputStream());

4. Server and client communicate through the stream objects is and os.

String line=is.readline();

send to client: os.writeBytes("Hello");

5.Close Socket.

client.close();

Example:

Creation of a server program:

Let's see a simple of java socket programming in which client sends a text and server receives it.

```
// SimpleServer.java: A simple server program.
import java.net.*;
import java.io.*;
public class SimpleServer
{
public static void main(String args[]) throws IOException
 {
// Register service on port 1254
ServerSocket s = new ServerSocket(1254);
Socket s1=s.accept();
 // Wait and accept a connection
```

```
// Get a communication stream associated with the socket
OutputStream s1out = s1.getOutputStream();
DataOutputStream dos = new DataOutputStream (s1out);
// Send a string!
//method writes a string to the underlying output stream using modified UTF-8 encoding.
dos.writeUTF("Hi there");
// Close the connection, but not the server socket
dos.close();
s1out.close();
s1.close();
}
}
```

Steps for Client program in Java

The steps for creating a simple client program are:

1. Create a Socket Object:
   Socket client = new Socket(server, port_id);

2. Create I/O streams for communicating with the server.
   is = new DataInputStream(client.getInputStream());
   os = new DataOutputStream(client.getOutputStream());

3. Perform I/O or communication with the server:
   Receive data from the server: String line = is.readLine();
   Send data to the server: os.writeBytes("Hello\n");

4. Close the socket when done:
   client.close();

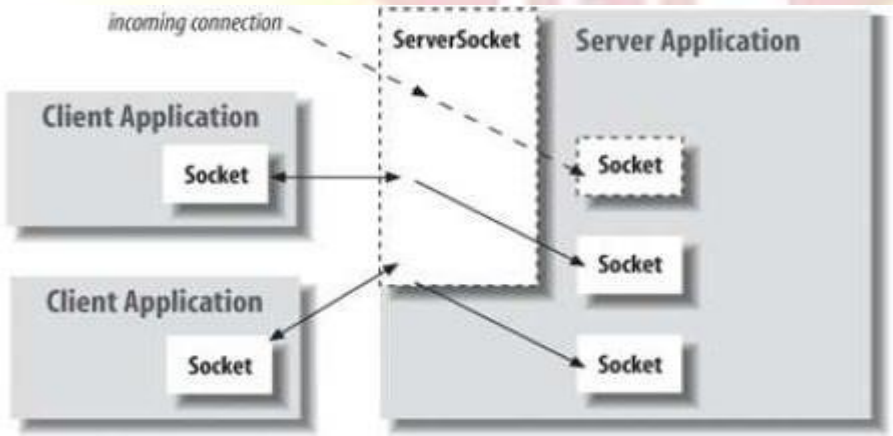
An example program illustrating establishment of connection to a server and then reading a message sent by the server and displaying it on the console is given below:

```
// SimpleClient.java: A simple client program.
import java.net.*;
import java.io.*;
public class SimpleClient
 {
    public static void main(String args[]) throws IOException
 {
// Open your connection to a server, at port 1254
   Socket s1 = new Socket("localhost",1254);
// Get an input file handle from the socket and read the input
InputStream s1In = s1.getInputStream();
DataInputStream dis = new DataInputStream(s1In);
String st = new String (dis.readUTF());
System.out.println(st);
// When done, just close the connection and exit
dis.close();
s1In.close();
s1.close();
}
}
```

TCP/IP

Socket Programming is used to provide the mechanism to connect two computers using TCP protocol . Socket provides an endpoint of two way communication link using TCP protocol. Java socket can be connection oriented or connection less. TCP provides two way communication, it means data can be sent across both the sides at same time.

TCP/IP Client and Server Sockets in Java



In Java , there are two classes to implement socket programming . These classes are :

1. Socket
2. ServerSocket .

TCP/IP Client Socket Class

Socket is class present in java.net package which represents TCP client . In order to create client we have to make object of Socket class in java. Following constructor can be used to create object of Socket class to provide the functionality of TCP/IP client.

1. public Socket(InetAddress local , int portNumber) - : it used to create a socket (client) using Inet Address object and a port.
2. public Socket(String hostName , int portNumber)- it used to create a socket (client) using hostName and a port.

Commonly Used Method Of The Socket Class:

Method	Description
public InputStream getInputStream()	returns the InputStream attached with this socket.
public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
public void close()	closes this socket
public int getPort()	return the invoking port to which the invoking Socket object is connected.
public InetAddress getInetAddress()	return the InetAddress associated with the Socket Object.

We can gain access to the input and output streams associated with a Socket by the use of the getInputStream() and getOutputStream()methods.

TCP/IP ServerSocket Class

This class represents the functionality of TCP/IP server in java. Object of ServerSocket is used to establish communication with the clients. Following constructor of ServerSocket class can be used to create TCP/IP server object .

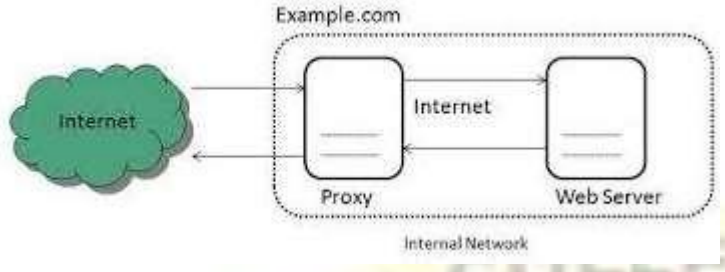
1. public ServerSocket(int portNumber)- it creates server socket on the specified port.
2. public ServerSocket(int portNumber , int maxQueueLength)- it creates server socket on the specified port with a queue length of QueueLength.

Commonly Used Methods Of ServerSocket Class

1. public Socket accept() - This method instruct TCP/IP server to establish network connection.
2. public void close() : This method is used to close the connection from the server.

Proxy Server

A proxy server is a server *(a* computer system or an application program) that acts as an intermediary for requests from clients seeking resources from other servers. A client connects to the proxy server, requesting some service, such as a file, connection, web page, or other resource, available from a different server.



Proxy servers offers the following basic functionalities:

- Firewall and network data filtering.
- Network connection sharing
- Data caching

Purpose of Proxy Servers

Following are the reasons to use proxy servers:

- Monitoring and Filtering
- Improving performance
- Translation
- Accessing services anonymously
- Security

Type of Proxies

Following table briefly describes the type of proxies:

Forward Proxies

In this the client requests its internal network server to forward to the internet.

Open Proxies

Open Proxies helps the clients to conceal their IP address while browsing the web.

Reverse Proxies

In this the requests are forwarded to one or more proxy servers and the response from the proxy server is retrieved as if it came directly from the original Server.

URL

The Java URL class represents a **pointer to a "resource" on the World Wide Web** . URL stands for **U**niform **R**esourse **L**ocator.

URL is a string of text that identifies all the resources on Internet, telling us the address of the resource, how to communicate with it and retrieve something from it.
A URL contains many information:

https://www.onlyjavatech.com/url-class-in-java
Protocol : In this case, http is the protocol .

Server name or IP Address: In this case, www.onlyjavatech.com is the server name.

Port Number: Port Number is optional. for HTTP services port number is 80. If the port number is not set explicitly then it returns -1.

File Name or directory name or resource: In this case, URL-class-in-java is the resource name

Commonly Used Constructor Of URL Class

Constructor	Description
public URL(String str)	Creates a URL object from the String str representation.
public URL(String protocol, String host, int port, String file)	Creates a URL object from the specified protocol, host, port number, and file.

Commonly Used Methods Of URL Class.

The java.net.URL class provides many methods. The important methods of URL class are given below.

Method	Description
public String getFile()	Gets the file name of this URL.
public String getHost()	Gets the host name of this URL, if applicable.
public String getPort()	Gets the port number of this URL.
public String getProtocol()	Gets the protocol name of this URL

public toString()	Constructs a string representation of this URL.
public URLConnection openConnection()	Returns the instance of URLConnection i.e. associated with this URL.

```
import java.io.*;
import java.net.*;
class GetURL {
	public static void main(String[] args) throws Exception, IOException
{
		System.out.println("Enter url :");
		BufferedReader br = new BufferedReader(new
					InputStreamReader(System.in));
		String str = br.readLine();

		// create java.net.URL object and pass str
		URL url = new URL(str);
		String host = url.getHost();
		int port = url.getPort();
		String file = url.getFile();
		String prot = url.getProtocol();

		// print all the information
		System.out.println("Host :" + host);
		System.out.println("Port :" + port);
		System.out.println("File/Resource :" + file);
		System.out.println("Protocol :" + prot);
	}
}
```

Output:

```
Enter url :
https://www.onlyjavatech.com/URL-class-in-java
Host :www.onlyjavatech.com
Port :-1
File/Resource :/URL-class-in-java
Protocol :https
```

DATAGRAM

Definition: A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.

For implementing Datagrams concept of Connection-less communication Java provides two classes :
1) DatagramSocket
2) DatagramPacket

DatagramPacket object is the data container. It represents packet that contain data part which is transferred from one computer to another computer.
DatagramSocket is the mechanism used to send or receive the DatagramPackets.

DatagramPacket Class

This class provides facility for connection less transfer of messages from one system to another. For sending and receiving purpose, several constructors are used. One of them is as follows:

Syntax :

public DatagramPacket (byte[] buf, int length, InetAddress address, int port)
Parameters :
buf : byte array
length : length of message to deliver
address : address of destination
port : port number of destination

Methods Of DatagramPacket Class

int getPort()-Returns the port to which this packet is sent to or from which it was received..
byte[ ] getData()- Returns the data contained in this packet as a byte array. The data starts from the offset specified and is of length specified.
int getOffset()- Returns the offset specified.
int getLength()- Returns the length of the data to send or receive

DatagramSocket Class

DatagramSocket class in java provides functionality for sending and receiving the packet.Some Commonly used Constructors of DatagramSocket class
DatagramSocket( ) - it creates a datagram socket and binds it with the available Port Number on the localhost machine.
DatagramSocket(int port) - it creates a datagram socket and binds it with the given Port Number.

Commonly used methods are

void send(DatagramPacket p) Sends a datagram packet from this socket.
void receive(DatagramPacket p) It is used to receive the packet from a sender.

Java Utility Classes

The package `java.util` contains a number of useful classes and interfaces. Although the name of the package might imply that these are utility classes, they are really more important than that. In fact, Java depends directly on several of the classes in this package, and many programs will find these classes indispensable. The classes and interfaces in `java.util` include:

1. Calendar

The java.util.calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on.

To create a calendar object, it is required to call it via Calendar.getInstance().

Constructors of Calendar:

Calendar is a abstract class, and we cannot use the constructor to create an instance. Instead, we use the static method Calendar.getInstance() to instantiate an implementation sub-class.

1.Calendar.getInstance(): return a Calendar instance based on the current time in the default time zone with the default locale

2.Calendar.getInstance(TimeZone zone)- This constructor constructs a calendar with the specified time zone

3.Calendar.getInstance(TimeZone zone, Locale aLocale)- This constructor constructs a calendar with the specified time zone and locale.

Methods of Calendar:

1. The most important method in `Calendar` is `get(int calendarField)`, which produces an `int`. The `calendarField` are defined as `static` constant and includes:

- `get(Calendar.DAY_OF_WEEK)`: returns 1 (`Calendar.SUNDAY`) to 7(`Calendar.SATURDAY`).
- `get(Calendar.MONTH)`: returns 0 (`Calendar.JANUARY`) to 11 (`Calendar.DECEMBER`).
- `get(Calendar.DAY_OF_MONTH)`, `get(Calendar.DATE)`: 1 to 31
- `get(Calendar.HOUR_OF_DAY)`: 0 to 23
- `get(Calendar.MINUTE)`: 0 to 59
- `get(Calendar.SECOND)`: 0 to 59
- `get(Calendar.MILLISECOND)`: 0 to 999

2. `void add(int field, int amount)`: Adds or subtracts the specified amount of time to the given calendar field

3. void setTime(Date aDate): Sets this Calendar's time with the given Date instance

4. `Date getTime()`: return a `Date` object based on this `Calendar`'s value.

2. Random :

The Random class of java.util package contains methods in handling Random numbers as the class name implies. It is used to generate a stream of pseudorandom numbers.

Constructors of Random:

1. Random() - This creates a new random number generator.

2. Random(long seed) - This creates a new random number generator using a single long seed.

Methods of **Random:**

1. boolean nextBoolean()-This method returns the next pseudorandom, uniformly distributed boolean value from this random number generator's sequence.

2. int nextInt() -method is used to get the next pseudorandom, uniformly distributed int value from this random number generator's sequence.

3. float **nextFloat()-** method is used to get the next pseudorandom, uniformly distributed float value between 0.0 and 1.0 from this random number generator's sequence.

4. int nextInt(int n)- This method returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), from this random number generator's sequence.

3. Vector:

The **java.util.Vector** class implements a growable array of objects.

- It contains components that can be accessed using an integer index.
- The size of a Vector can grow or shrink as needed to accommodate adding and removing items.

Constructors of Vectors

1. Vector()- This constructor is used to create an empty vector.

2. Vector(int initialCapacity)- This constructor is used to create an empty vector with the specified initial capacity.

3.Vector(int initialCapacity, int capacityIncrement)- This constructor is used to create an empty vector with the specified initial capacity and capacity increment

Methods *of* Vectors

1. boolean add(E e) - This method appends the specified element to the end of this Vector.
2. void add(int index, E element)- This method inserts the specified element at the specified position in this Vector.
3. int capacity()- This method returns the current capacity of this vector.
4.void clear()-This method removes all of the elements from this vector.
5. void removeAllElements()-This method removes all components from this vector and sets its size to zero.

Introducing the AWT

Java AWT (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java. The java.awt package provides classes for AWT API such as Text Field, Label, Text Area, Radio Button, Check Box, Choice, List etc.

The `java.awt` package contains the *core* AWT graphics classes:

- GUI Component classes, such as `Button`, `Text Field`, and `Label`.
- GUI Container classes, such as `Frame` and `Panel`.
- Layout managers, such as `FlowLayout`, `BorderLayout` and `GridLayout`.
- Custom graphics classes, such as `Graphics`, `Color` and `Font`.

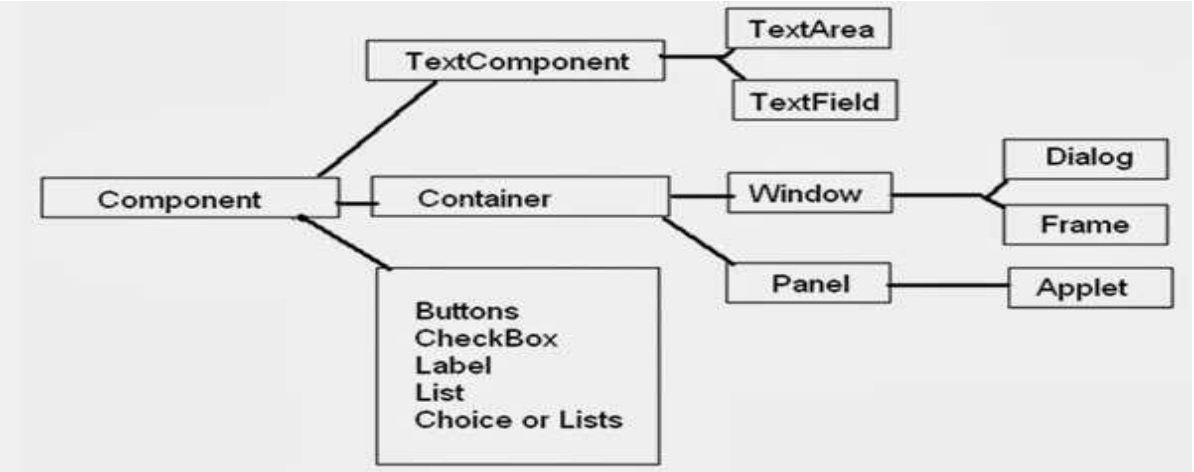
Window Fundamentals

Java Abstract window tool kit package is used for displaying the data within a GUI Environment. Features of AW T Package are as Followings:

1. It provides us a set of user interface components including windows buttons text fields scrolling list etc.
2. It provides us the way to laying out these above components.
3. It provides to create the events upon these components.

The main purpose for using the AWT is using for all the components displaying on the screen. AWT defines all the windows according to a class hierarchy those are useful at a specific level or we can say arranged according to their functionality.

The most commonly used interface is the panels those are used by applets and those are derived from frame which creates a standard window. The hierarchy of this AWT is:-



AWT hierarchy

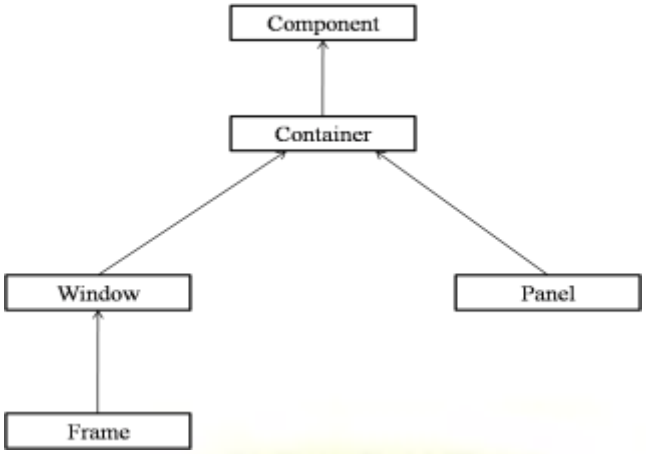
AWT CLASSES

The AWT framework contains many classes and interfaces using which we can create GUIs. The AWT package is *java.awt*. Some of the frequently used AWT classes are listed below:

Class	Description
AWTEvent	Encapsulates AWT events and is the root class for all AWT events
BorderLayout	The border layout manager
Button	Creates a push button control
Canvas	A blank window
CardLayout	The card layout manager
Checkbox	Creates a checkbox control
CheckboxGroup	Creates a group of checkboxes (radio buttons)
Choice	Creates a drop down list
Color	Manages colors in GUI programs
Component	An abstract super class for many AWT components
Container	A sub class of container which can hold other components
Dialog	Creates a dialog window
Dimension	Specifies the dimensions (width and height) of an object
FileDialog	Creates a window from which user can select a file
FlowLayout	The flow layout manager
Frame	Creates a standard window
Graphics	Encapsulates the graphics context
GridBagLayout	The grid bag layout manager
GridLayout	The grid layout manager
Image	Encapsulates graphical images
Label	Creates a label control to display static text
List	Creates a list control
Menu	Creates a menu control
MenuBar	Creates a menu bar control
MenuItem	Create a menu item
Panel	A sub class of container which can hold other components
PopupMenu	Creates a pop-up menu
Scrollbar	Creates a scroll bar control
ScrollPane	A container that provides scroll bars for a component
TextArea	Creates a mult-line text control
TextField	Creates a single-line text control
Window	Creates a window with no frame, title bar and title

Component and Container

The Component is the abstract class for many GUI control classes. Container is a sub class of Component class. The Component and various Container classes are arranged in a hierarchy as shown below:



Component

Component is the abstract class that encapsulates all the properties of a visual component. Except for menus, most of the GUI components are inherited from the Component class. The Components are elementary GUI entities, such as Button, Label, and TextField.

Container

Container is a sub class of the Component class which can be used to hold other components. A Container object can hold other Containers also. A Container is responsible for laying out (positioning) the components. The Containers, such as `Frame` and `Panel`, are used to hold components in a specific layout (such as `FlowLayout` or `GridLayout`).

Panel

Panel class is a concrete sub class of the Container class. A Panel object is a window without title bar, menu bar and border. Panel is the super class of Applet class and is capable of holding other components or containers.

Window

Window is a sub class of Container class. A Window creates a top-level container which can hold other components or containers.

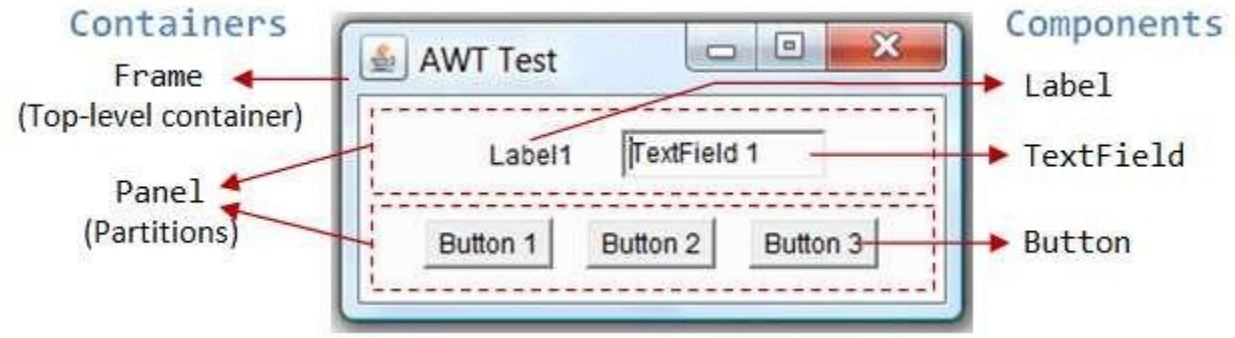
Frame

Frame is a concrete sub class of Window class. The Frame encapsulates a window. Frame contains a title bar, menu bar, borders and resizable corners. To create stand alone applications in Java, we generally use Frame. The frame is a container object, so GUI components can be placed in it.

Canvas

Canvas class is derived from the Component class. A Canvas encapsulates a blank window on which we can draw.

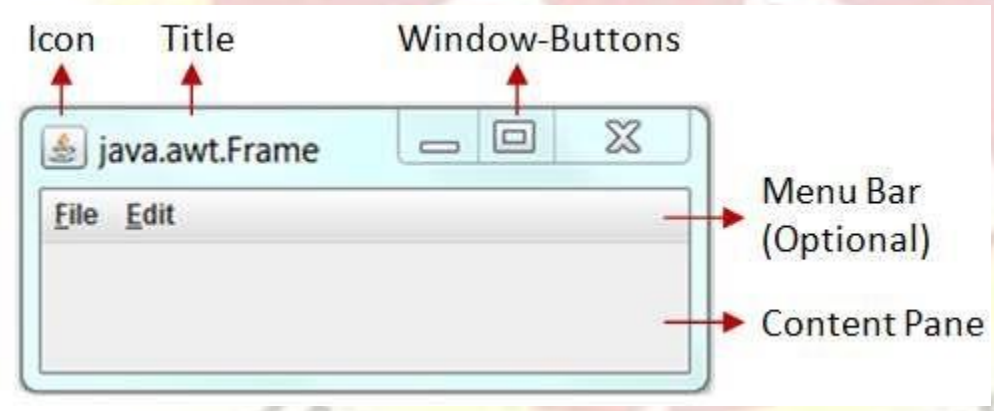
The following figure represents the container and components.



In the above figure, there are three containers: a Frame and two Panels. A Frame is the *top-level container* of an AWT program. A Frame has a title bar (containing an icon, a title, and the minimize/maximize/close buttons), an optional menu bar and the content display area. A Panel is a *rectangular area* used to group related GUI components in a certain layout. In the above figure, the top-level Frame contains two Panels. There are five components: a Label (providing description), a TextField (for users to enter text), and three Buttons (for user to trigger certain programmed actions).

Working With Frames Windows

Frame Class is a Sub Class of Window Class and frame class allows to Create a standard windows. The Frame Class Provides a Special Type of Window which has a title bar, menu bar , border and Resizing corners and it is a subclass of window class . The following figure specifies the frame.



The Various Methods those are Provided by the Frame Class are as follows :

1. **Frame class constructors:**

Frame() - Creates a Frame window with no name.

Frame(String title) - Creates a Frame window with a title specified by the title.

2. **Setting the Window Dimensions**

void setSize(int width, int height) – Used to specify the width and height of the frame window.

void setSize(Dimension size) – Used to specify the dimensions of the frame window.

Dimension getSize() – Returns the dimensions of the frame window.

3. Hiding and Showing a window

After a frame window has been created,it will not be visible until we call setVisible( ).

void setVisible(boolean visibleFlag) – *Makes the frame window visible if the boolean argument is true or non-visible if the boolean argument is false.*

4. To Display Frame:

Void show() : This is used for displaying or showing a Frame from a Main Window.

5. Setting a Window Title

We can change the title in a frame window using setTitle( ).

void setTitle(String title) – *Used to set the new title of the frame window*

6. Closing a Frame Windows

The windowClosing() method usec when the frame is attempted to close from the window's system menu or close button.

The following program code show how to create a frame window

```
import java.awt.*;
public class Framed1 extends Frame
{
	public static void main(String args[])
	{
		/* Creating a frame object */

		Frame frmobj=new Frame("My First Frame");
		frmobj.setSize(400,450);
		frmobj.setVisible(true);
	}
}
```

Note: The java.awt.* package should be included in the java programs to implement the functions of AWT classes in our program.

Working With Color

Adding AWT Colors improves the readability. This can be done by adding the color class to the java.awt package.
There are two formats to define Color, they are:

- RGB Model
- HSB Model

RGB Model: Red, Green and Blue are the primary colors in this model,through which many other colors can be formed by their combination. The value ranges between 0 and 255.If the value is not between the range, IllegalArgumentException is thrown.

Syntax: Color c = new Color(int red, int green, int blue);

HSB Model: Hue, Saturation and Brightness forms the HSB Model. Color Constructor will be passed with the values ranging between 0.0f and 1.0f. If the value is not between the range, IllegalArgumentException is thrown.

Syntax: Color c = new Color(float red, float green, float blue);

1. To Create a color Object using Constructors:

To use colors in our GUI applications, AWT provides the Color class. We can create a Color object by using any of the following constructors:

- Color(int red, int green, int blue)
- Color(int rgbValue)
- Color(float red, float green, float blue)

The first constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:
new Color(255, 100, 100); // light red.

The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor:

The final constructor, Color(float, float, float), takes three float values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue. Once we have created a color, we can use it to set the foreground and/or background color by using the setForeground( ) and setBackground( ) methods

2. To Get Color Values:

We can get values of red component, green component or blue component of the color by using the following methods:
int getRed()

int getGreen()

int getBlue()

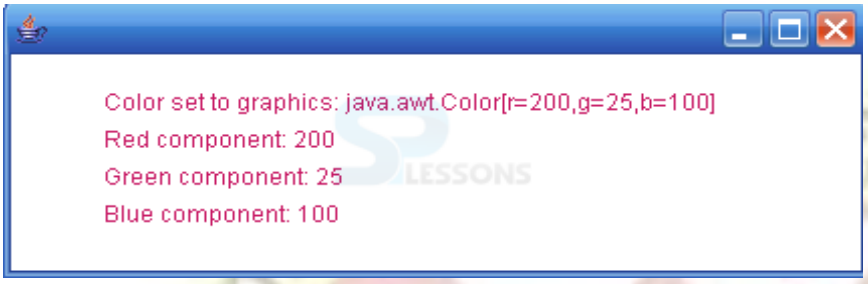
3. To Set New Color:
void setColor(Color newColor)

Example:

```
public class UsingColors extends Frame
{
  public void paint(Graphics g)
  {
    Color clr1 = new Color(200, 25, 100);
    g.setColor(clr1);
    g.drawString("Color set to graphics: " + g.getColor(), 50, 60);
    g.drawString("Red component: " + clr1.getRed(), 50, 80);
    g.drawString("Green component: " + clr1.getGreen(), 50, 100);
    g.drawString("Blue component: " + clr1.getBlue(), 50, 120);
```

```
  }
  public static void main(String args[])
  {
    Frame f1 = new UsingColors();
    f1.setSize(300, 200);
    f1.setVisible(true);
  }
}
```

Output:



Working with Fonts

The AWT supports multiple font Types. The Fonts have a family name, a logical name and a face name.

- Family Name - General name of the font
- logical name - Specifies the Category of the Font
- face name - Specifies the Specific Font

The Font class provides a method of specifying and using fonts.

The Font class constructor constructs font objects using the font's name, style (PLAIN, BOLD, ITALIC, or BOLD + ITALIC), and point size.

Syntax : Font (String FontName,String FontStyle, int FontSize)

Example: *Font("Arial",Font.ITALIC,20)*

The getName() method returns the logical Java font name of a particular font and the getFamily() method returns the operating system-specific name of the font. The standard Java font names are Courier, Helvetica, TimesRoman etc.

getFont() - It is a method used to get the font property

setFont(Font f) is used to set a font.

Font Style:

The font's style is passed with the help of the class variables Font.PLAIN, Font.BOLD, and Font.ITALIC. The combination Font.BOLD | Font.ITALIC specifies bold italics.

Example:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET CODE ="FontClass.class" WIDTH=300 HEIGHT=200>
/APPLET> */
public class FontClass extends java.applet.Applet
   {
      Font f;
      String m;
        public void init()
          {
            f=new Font("Arial",Font.ITALIC,20);
            m="Welcome to Java";
            setFont(f);
          }
            public void paint(Graphics g)
              {
                Color c=new Color(0,255,0);
                g.setColor(c);
                g.drawString(m,4,20);
              }
  }
```

OUTPUT

Using AWT Controls

AWT Controls are nothing but AWT Components that allows the user to communicate with the user in different ways.

AWT controls are:

1. Labels
2. Push buttons
3. Check boxes
4. Choice lists
5. Lists
6. Scroll bars
7. Text Area
8. Text Field

Adding and Removing Controls

Initially an instance of required control has to be created and in the next step instance should be added to the window by calling add() method.

Syntax
Component add(Component obj)

- The control can be removed from window by calling the method remove().

Syntax
Void remove(Component obj)

- All the controls can be removed at a time by calling the method removeAll().

The frequently-used are: AWT controls are as illustrated below.

Figure: AWT Controls

1. Label

A label is a GUI control which can be used to display static text. Label can be created using the *Label* class and its constructors which are listed below:

Label() - creates a blank label

Label(String str) - creates a label that contains the string specified by *str,* string is left-justified

Label(String str, int how) - creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of how must be one of these three constants: Label.LEFT, Label.RIGHT, or Label.CENTER.

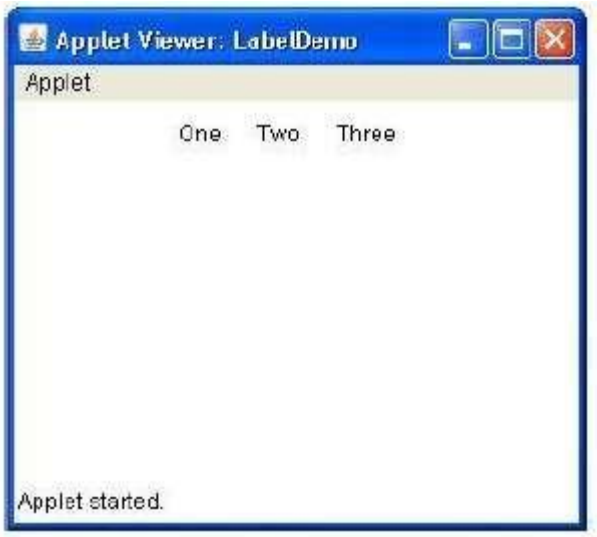
We can set or change the text in a label by using the setText( ) method. We can obtain the current label by calling getText( ). These methods are shown here:
  void setText(String str) - str specifies the new label
  String getText( )- the current label is returned

To obtain the current alignment, call getAlignment( ). The methods are as follows:
  void setAlignment(int how)
 int getAlignment( )
 Example:

```
import java.awt.*;
import java.applet.*;
/*<APPLET Code=" LabelDemo " Width=500 Height=200></APPLET>*/
public class LabelDemo extends Applet
{
public void init()
{
Label one = new Label("One");
Label two = new Label("Two");
Label three = new Label("Three");
// add labels to applet window
add(one);
add(two);
add(three);
}}
```

Output:

Following is the window created by the LabelDemo applet. Notice that the labels are organized in the window by the default layout manager.



2.BUTTONS

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 constructors which are given below:
Button() - creates an empty button with no label
Button(String str) creates a button with the given string as its label.

The methods available in the Button class are as follows:

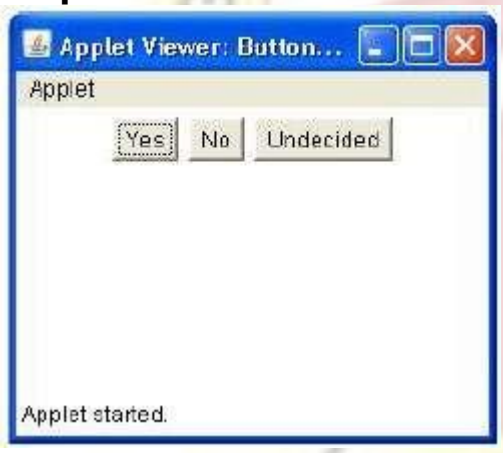
void setLabel(String str) – To set or assign the text to be displayed on the button.

String getLabel() – To retrieve the text on the button.

Example:

```
import java.awt.*;
import java.applet.*;
public class ButtonDemo extends Applet
{
String msg = "";
Button yes, no, maybe;
public void init()
{
yes = new Button("Yes");
no = new Button("No");
maybe = new Button("Undecided");
add(yes);
add(no);
add(maybe);
}
public void paint(Graphics g)
{
g.drawString(msg, 6, 100);
}
}
```

output:



3. Checkboxes

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

Checkbox supports these constructors:

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

The first form. The state of the check box is unchecked.

The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked.

The third form allows us to set the initial state of the check box. If on is true, the check box is initially checked; otherwise, it is cleared.

The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be null. The value of on determines the initial state of the check box.

In order to retrieve the current state of a check box, call getState( ). For setting its state, call setState( ). We can obtain the current label associated with a check box by calling getLabel( ). For setting the label, setLabel( ) is used. These methods are as follows:

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

Example:

```
import java.awt.*;
import java.applet.*;
public class CheckboxDemo extends Applet
{
String msg = "";
Checkbox Win98, winNT, solaris, mac;
public void init()
{
Win98 = new Checkbox("Windows 98/XP", null, true);
winNT = new Checkbox("Windows NT/2000");
solaris = new Checkbox("Solaris");
mac = new Checkbox("MacOS");
add(Win98);
add(winNT);
add(solaris);
add(mac);
}
public void paint(Graphics g)
{
}
}
```

Output:

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. These check boxes are often called radio buttons, because they act like the station selector on a car radio —only one station can be selected at any one time.

For creating a set of mutually exclusive check boxes, we must first define the group to which they will belong and then specify that group when we construct the check boxes. Check box groups are objects of type Check boxGroup. Only the default constructor is defined, which creates an empty group.

We can determine which check box in a group is currently selected by calling get Selected Checkbox( ). We can set a check box by calling set Selected Checkbox( ). These methods are as follows:

```
Checkbox getSelectedCheckbox( )

void setSelectedCheckbox(Checkbox wh)
```

Here, *wh* is the check box that we want to be selected. The previously selected check box will be turned off.

Here is a program that uses check boxes that are part of a group:

```
import java.awt.*;
import java.applet.*;
public class CBGroup extends Applet
{
String msg = "";
Checkbox Win98, winNT, solaris, mac;
CheckboxGroup cbg;
public void init()
{
cbg = new CheckboxGroup();
Win98 = new Checkbox("Windows 98/XP", cbg, true);
winNT = new Checkbox("Windows NT/2000", cbg, false);
solaris = new Checkbox("Solaris", cbg, false);
mac = new Checkbox("MacOS", cbg, false);
add(Win98);
add(winNT);
add(solaris);
add(mac);
}
public void paint(Graphics g)
{
msg = "Current selection: ";
msg += cbg.getSelectedCheckbox().getLabel();
g.drawString(msg, 6, 100);
}
}
```

Choice Controls

The Choice class is used to create a pop -up list of items from which the user may choose.

When inactive, a Choice component takes up only enough space to show the currently selected item.

When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left justified label in the order it is added to the Choice object.

Choice only defines the default constructor, which creates an empty list. In order to add a selection to the list, add( ) is used.

It has this general form:
void add(String name)

Here, name is the name of the item being added. Items are added to the list in the order in which calls to add( ) occur. In order to determine which item is currently selected, we may call either any of the following methods:

String getSelectedItem( )- method returns a string containing the name of the item.
int getSelectedIndex( )-) returns the index of the item.

Example:

```
import java.awt.*;
import java.applet.*;
public class ChoiceDemo extends Applet
{
Choice os, browser;
String msg = "";
public void init()
{
os = new Choice();
browser = new Choice();
os.add("Windows 98/XP");
os.add("Windows NT/2000");
os.add("Solaris");
```

```
os.add("MacOS");
browser.add("Netscape 3.x");
browser.add("Netscape 4.x");
browser.add("Netscape 5.x");
browser.add("Netscape 6.x");
browser.add("Internet Explorer 4.0");
browser.add("Internet Explorer 5.0");
browser.add("Internet Explorer 6.0");
browser.add("Lynx 2.4");
browser.select("Netscape 4.x");
add(os);
add(browser);
}
public void paint(Graphics g)
{
}
}
```

Lists

The List class provides a compact, multiple-choice, scrolling selection list. Unlike the Choice object, which shows only the single selected item in the menu, a List object can be constructed to show any number of choices in the visible Window. It can also be created to allow multiple selections. List provides these constructors:

List( )

List(int numRows)

List(int numRows, boolean multipleSelect)

The first version creates a List control that allows only one item to be selected at any one time.
In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed).

In the third form, if *multipleSelect* is true, then the user may select two or more items at a time. If it is false, then only one item may be selected.
For adding a selection to the list, we can call add( ). It has the following two forms:
void add(String name)
void add(String name, int index)
Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index*.
The get Selected Item( ) method returns a string containing the name of the item. If more than one item is selected or if no selection has yet been made, null is returned.
The get Selected Index( ) returns the index of the item.
Example:

```
import java.awt.*;
import java.applet.*;
public class ListDemo extends Applet
{
List os, browser;
String msg = "";
public void init()
{
os = new List(4, true);
browser = new List(4, false);
os.add("Windows 98/XP");
os.add("Windows NT/2000");
os.add("Solaris");
os.add("MacOS");
browser.add("Netscape 3.x");
browser.add("Netscape 4.x");
browser.add("Netscape 5.x");
browser.add("Netscape 6.x");
browser.add("Internet Explorer 4.0");
browser.add("Internet Explorer 5.0");
browser.add("Internet Explorer 6.0");
browser.add("Lynx 2.4");
browser.select(1);
add(os);
add(browser);
}
```

```
public void paint(Graphics g)
{
int idx[];
msg = "Current OS: ";
idx = os.getSelectedIndexes();
for(int i=0; i<idx.length; i++)
msg += os.getItem(idx[i]) + " ";
g.drawString(msg, 6, 120);
msg = "Current Browser: ";
msg += browser.getSelectedItem();
g.drawString(msg, 6, 140);
}
}
```

Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. Each end has an arrow that we 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. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value

Scrollbar defines the following constructors:

Scrollbar( )

Scrollbar(int style)

Scrollbar(int style, int iValue, int tSize, int min, int max)

The first form creates a vertical scroll bar. The second and third forms allow us to specify the orientation of the scroll bar.

If *style* is Scrollbar.VERTICAL, a vertical scroll bar is created.

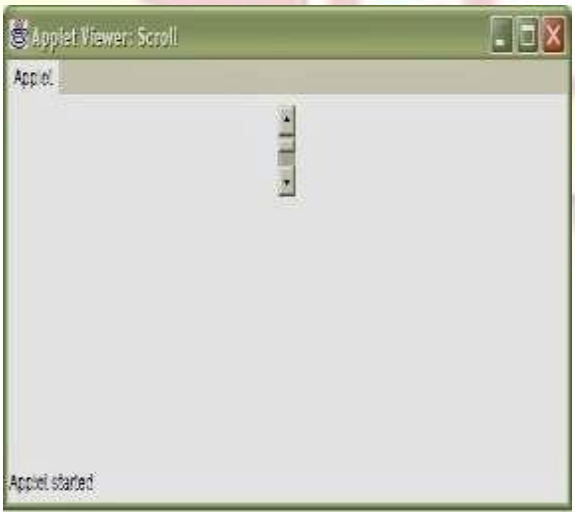
If *style* is Scrollbar.HORIZONTAL, the scroll bar is horizontal.

In the third form of the constructor, the initial value of the scroll bar is passed in *iValue*. The number of units represented by the height of the thumb is passed in *tSize*. The minimum and maximum values for the scroll bar are specified by min and max.

Example:

```
import java.awt.*;
import java.applet.*;
/*<APPLET Code="Scroll" Width=500 Height=200></APPLET>*/
public class Scroll extends Applet
{
 Scrollbar bar = new Scrollbar(Scrollbar.VERTICAL, 10, 0, 1, 100);
  public void init( )
 {
  add(bar);
 }
}
```

Output:



Text Fields

A text field or text box is a single line text entry control which allows the user to enter a single line of text. a text field can be created using the *TextField* class along with its following constructors:

TextField()-Empty TextBox which will never display any Text String into that TextBox

TextField(int numChars) - This will Create a TextField and determines the number of characters to be displayed at a Time in the textbox.

TextField(String str)- Creates a text field with a given string

TextField(String str, int numChars)- numChars specifies the width of the text field, and str specifies the initial text in the text field.

Example:

```
import java.awt.*;
import java.applet.*;
/*<APPLET Code="TextFieldTest" Width=500 Height=200></APPLET>*/
public class TextFieldTest extends Applet
{
	public void init( )
	{
		Label lblName = new Label("enter name");
		Label lblPhone = new Label("enter phone number");
		Label lblPasswd = new Label("enter password");
		TextField txtName = new TextField("your name here", 20);
		TextField txtPhone = new TextField(12);
		add(lblName);
		add(txtName);
		add(lblPhone);
		add(txtPhone);
	}
}
```

TextArea

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called TextArea. Following are the constructors for TextArea:

TextArea( )

TextArea(int numLines, int numChars)

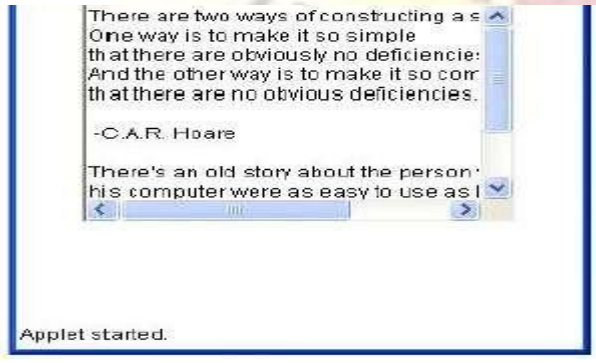
TextArea(String str)

TextArea(String str, int numLines, int numChars)

Here, *numLines* specifies the height, in lines, of the text area, and *numChars*specifies its width, in characters. Initial text can be specified by *str*.

Example

```
import java.awt.*;
import java.applet.*;
public class TextAreaDemo extends Applet
{
public void init()
{
String val = "There are two ways of constructing " +"a software design.n"
+"One way is to make it so simplen" +"that there are obviously no
deficiencies.n" +"And the other way is to make it so complicatedn" +"that
there are no obvious deficiencies.nn" +" -C.A.R. Hoarenn" +"There's an
old story about the person who wishedn" +"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.nn" +" -Bjarne Stroustrup, AT&T,
(inventor of C++)";
TextArea text = new TextArea(val, 10, 30);
add(text);
}
```




```
}
```

Layout Managers and Menus.

AWT Layout is defined as the order of placement of the components in a container. This arrangement can be done by **LayoutManager**

JAVA 's AWT Package Provides various Layouts to arrange the Components and also provide a method which is also called as setLayout, to set the Layout we have to use this Method and name of object of used Layout.

A layout manager determines how components will be arranged when they are added to screen or container , A Container may a Frame , Applet , Panel etc The AWT package includes classes for layout managers.
In Java Button, Checkbox, Lists, Scrollbars, Text Fields, and Text Area etc. positioned by the default layout manager. Using algorithm layout manager automatically arranges the controls within a window.

Layout manager are used to arrange components within the container. It automatically places the control at a particular position within window.

LayoutManager is an interface implemented by all the classes of layout managers.

Different layout managers available in AWT are:

1. FlowLayout
2. BorderLayout
3. GridLayout
4. CardLayout
5. GridBagLayout

1. FlowLayout

Flow layout is the default layout, which means if we don't set any layout in our code then layout would be set to Flow by default. Flow layout puts components (such as text fields, buttons, labels etc) in a row, if horizontal space is not enough to hold all components then Flow layout adds them in a next row and so on as shown in the following figure..

The flow layout manager arranges the components one after another from left-to-right and top-to-bottom manner. The flow layout manager gives some space between components. Flow layout manager instance can be created using anyone of the following constructors:

```
 FlowLayout()
FlowLayout(int how)
FlowLayout(int how, int hspace, int vspace)
```

In the above constructors, *how* specifies the alignment, *hspace* specifies horizontal space, and *vspace* specifies vertical space. Valid values for alignment are as follows:

```
FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
```

Example:

Here is a version of the CheckboxDemo applet shown, modified so that it uses left -aligned flow layout.

```
public class FlowLayoutDemo extends Applet
{
Checkbox Win98, winNT, solaris, mac;
public void init()
{
Win98 = new Checkbox("Windows 98/XP", null, true);
winNT = new Checkbox("Windows NT/2000");
solaris = new Checkbox("Solaris");
mac = new Checkbox("MacOS");
setLayout(new FlowLayout(FlowLayout.CENTER));
add(Win98);
add(winNT);
add(solaris);
add(mac);
}
}
```

Output:

2. *BorderLayout Manager*

The border layout manager divides the container area into five regions namely: north, south, east, west, and center. Default region is center. The following figure shows the border Layout.

Border layout instance can be created by using one of the below constructors:
BorderLayout()
BorderLayout(int hspace, int vspace)

In the above constructors, *hspace* signifies horizontal space between components and *vspace* signifies vertical space between components.It is uses add method to add the component at specified region.

Components of the BorderLayout Manager

BorderLayout.NORTH

BorderLayout.SOUTH

BorderLayout..EAST

BorderLayout.WEST

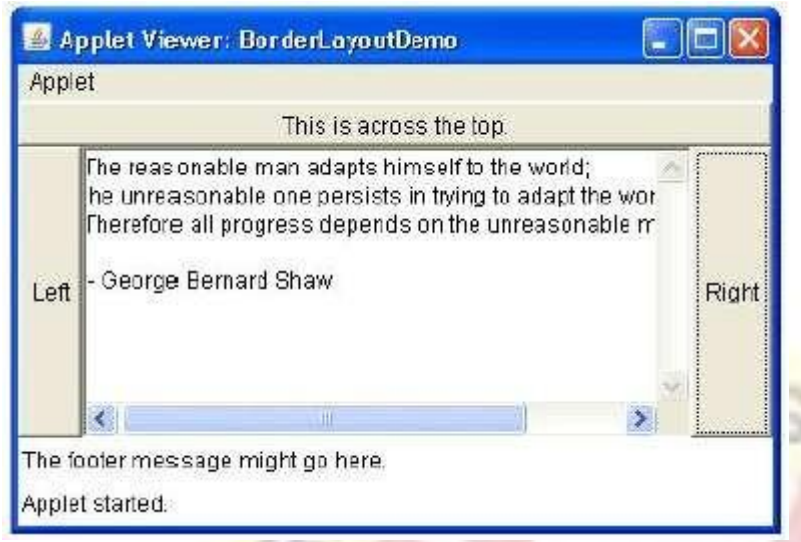
BorderLayout.CENTER

Example:

import java.awt.;*

```
import java.applet.*;
public class BorderLayoutDemo extends Applet
{
public void init()
{
setLayout(new BorderLayout());
add(new Button("This is across the top."),
BorderLayout.NORTH);
add(new Label("The footer message might go here."),
BorderLayout.SOUTH);
add(new Button("Right"), BorderLayout.EAST);
add(new Button("Left"), BorderLayout.WEST);
String 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.nn" +" - George Bernard Shawnn";
add(new TextArea(msg), BorderLayout.CENTER);
}
}
```

OUTPUT



3. GridLayout Manager

The grid layout manager arranges the components in a 2-dimensional grid. While creating the instance of GridLayout, we can specify the number of rows and columns in the grid. Care must be taken with the number of cells in the grid and the number of components being added to the grid. If they don't match, we might get unexpected output.

An instance of GridLayout can be created using one of the following constructors:

GridLayout()

GridLayout(int numRows, int numCols)

GridLayout(int numRows, int numCols, int hspace, int vspace)

In the above constructors, *numRows* and *numCols* specifies the number of rows and columns in the grid, *hspace* and *vsapce* specifies the horizontal space and vertical space between the components.

Here is a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:

4. CardLayout Manager

CardLayout is used to display a **single component at a time** among multiple components. This layout helps in saving the space.

The CardLayout class contains several layouts in it. The Cardlayout manages the components in form of stack and provides visibility to only one component at a time.

CardLayout Constructors

Constructor	Description
CardLayout ()	It constructs a new card layout with the gap of zero size.
CardLayout (int hgap, int vgap)	It is also used to create a new card layout with the specified horizontal and vertical gaps.

Commonly used methods of CardLayout class

- **public void next(Container parent):** is used to flip to the next card of the given container.
- **public void previous(Container parent):** is used to flip to the previous card of the given container.
- **public void first(Container parent):** is used to flip to the first card of the given container.
- **public void last(Container parent):** is used to flip to the last card of the given container.
- **public void show(Container parent, String name):** is used to flip to the specified card with the given name.

5. GridBagLayout Manager

GridBagLayout layout manager is the most powerful and flexible of all the predefined layout managers but more complicated to use.
The grid bag layout manager can be used to create an uneven grid i.e., number of columns in each row can differ. Also the size of components within a cell can be different.

The location and size of each component are specified by a set of constraints that are contained in an object of type **GridBagConstraints**. These constraints include height, width, and placement of a component.

The general process to work with **GridBagLayout** is, first set the layout of the container to **GridBagLayout**, then set the constraints for each component using **GridBagConstraints**, and then add each component to the container.

Menu Bars and Menus

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices.

Each choice is associated with a dropdown menu. It uses the classes for : ***MenuBar***, ***Menu***, and ***MenuItem***.

A menu bar contains one or more Menu objects.

Each Menu object contains a list of MenuItem objects.

Each MenuItem object represents something that can be selected by the user.

Creating a menu bar is straight forward. we have to simply create an instance of Java.awt.MenuBar which is a container for menus. It has only one constructor which is the default constructor as follows,

public MenuBar ()

Once you create the menu bar, we can add it to a frame with the setMenuBar () method of Frame class. One should remember that each application has only one menu.

The following statements create a menu bar and add it to the top of the frame.

```
Menubar menuBar = new MenuBar();

setMenubar(menuBar);
```

These statements are part of the class that extends Frame.

Initially the menubar is empty and you need to add menus before using it. Each menu is represented by an instance of class Menu. In order to create a Menu instance, you have to use the following constructor.

public Menu(String str)

where str represents the label for the menu. For example: In order to create a File menu, use the following statement

Menu menuFile = new Menu(“File");

Similarly to create Edit and View menu, use the statements

Menu menuEdit = new Menu(“Edit");

Menu menuView = new Menu ("View");

Once the Menu objects are created, we need to add them to the menu bar. For this, you have to use

the add () method of the MenuBar class whose syntax is as follows,

Menu add (Menu menu)

where menu is Menu instance that is added to the menu bar. This method returns a reference to the menu. By default, consecutively added menus are positioned in the menu bar from left to right. This makes the first menu added the leftmost menu and the last menu added the rightmost menu. If you want to add a menu at a specific location, then use the following version of add ()method inherited from the Container class.

Component add (Component menu, int index)

where menu is added to the menu bar at the specified index. Indexing begins at 0, with 0 being the leftmost menu. For example: In order to add menu instance menuFile to the menuBar use the following statement,

menuBar.add(menuFile);

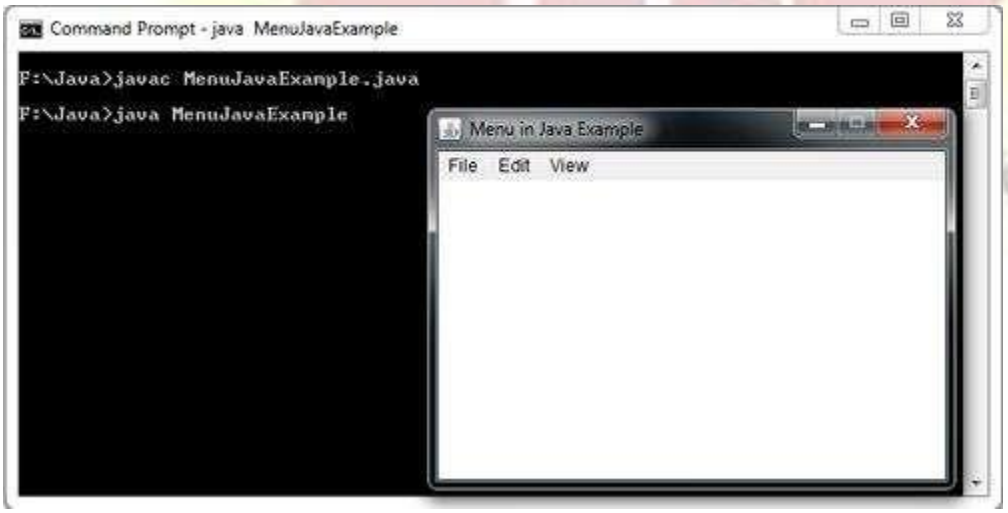
Similarly, add other Menu instances menuEdit,menuView using the following statements,

menuBar.add(menuEdit);

menuBar.add(menuView);

```
import java.awt.*;
class MenuExample extends Frame
```

```
{
     MenuExample()
    {
        MenuBar menuBar = new MenuBar();
        setMenuBar(menuBar);
        Menu menuFile = new Menu("File");
        Menu menuEdit = new Menu("Edit");
        Menu menuView = new Menu("View");
        menuBar.add(menuFile);
        menuBar.add(menuEdit);
        menuBar.add(menuView);
        MenuItem itemOpen = new MenuItem("Open");
        MenuItem itemSave = new MenuItem("Save");
        MenuItem itemExit = new MenuItem("Exit");
        menuFile.add(itemOpen);
        menuFile.add(itemSave);
        menuFile.add(itemExit);
        MenuItem itemcopy = new MenuItem("Copy");
        menuEdit.add(itemcopy);
   }
}
 class MenuJavaExample
 {
       public static void main(String args[])
      {
          MenuExample frame = new MenuExample();
          frame.setTitle("Menu in Java Example");
          frame.setSize(350,250);
          frame.setResizable(false);
          frame.setVisible(true);
      }
 }
```



*****