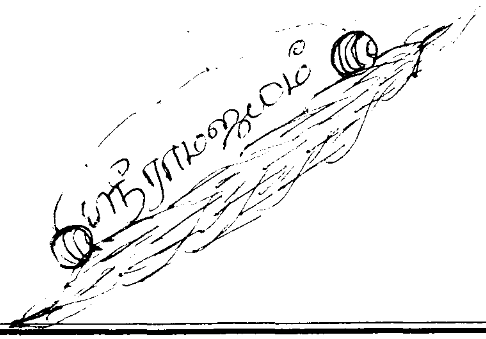
```
int main()
{
    iteml p(100,5,140.0);
    item2 q;
    float sum;
    sum=p;
    q=p;
    cout<<" Iteml code and its type"<<"\n";
    q.output();
    cout<<"\n Available Item"<<"\n";
    cout<<"Amount= "<< sum<<"\n\n";
    cout<<"Item2 code and its type\n";
    q.output();
    return 0;
}
```

OUTPUT

```
Item 1 code and its type
Ino: 100
Amount: 700
Available Items
amount= 14000
Item2 code and its type
ino: 100
amount: 700
```



Chapter-3

JAVA

Introduction – Features of Java – Difference between C++ and JAVA – Bytecode and JVM – JAVA class Libraries – Structure of Java program – First Java Program – Java Development Kit – Compilation and Running a JAVA program – Variables and Data types – Constants – Operators – Keywords – Expressions – Conditional and Control Statements – Classes – Objects – Constructors – Overloading – Inheritance – Subclass – Overriding – Final variables and methods – Visibility control – Arrays – Strings – Vectors – Wrapper class – Interfaces – Packages – Multi threaded programming – Exceptions – Applet Programming – Files in JAVA – Viva Voce Questions – Answers for Viva voce Questions – Lab Exercises – Answer for Lab Exercises

3.1 INTRODUCTION

JAVA was developed by a team of programmers at **SUN MICROSYSTEMS** in 1991. The team consists of 5 members, they are **James Gosling, Patrick Naughton, Chris Warth, Ed Frank** and **Mike Sheridan.**

This Language was initially called as **"Oak"**, but was renamed as "JAVA" in 1995.

The main goal of **JAVA** was to develop a language that could be used to write software for different consumer electronic devices such as Microwave ovens and Remote controls.

JAVA is a platform independent language, because they would run in different execution environments. The **JAVA** allows the same software to execute without change on a hetrogeneous environments.

The **JAVA's** philosophy **"WRITE ONCE RUN ANYWHERE"** provided a tremendous advantage.

static Is the keyword indicates that this method is associated with **First** class.

void It is the prototype keyword indicates that this method does not return any value.

System.out.println( ): Is the output statement which prints specified message on the screen.

Before executing a JAVA program, the program must be compiled using the **javac** statement as follows:

```
C:\JDK1.2\BIN>javac First.java<Enter>
```

If the command prompt returns without errors, the file **First.class** was created in the same directory which contains the byte codes.

To run a program, do the following:

```
C:\JDK1.2\BIN>java First<Enter>
```

Now, the above program will give the result as given below:

OUTPUT:

```
First Java Program
```

3.9 JAVA ENVIRONMENT

To write a program JAVA provides number of tools and hundreds of classes and methods.

The tools that JAVA provides are part of the **Java Development Kit (JDK)** and the classes and methods are part of the **Java Standard Library (JSL)** this is also called as the **Application Programming Interface (API)**.

3.9.1 Java Development Kit (JDK)

To run a JAVA program the Java Development Kit comes with a collection of tools given below.

Tool	Used to
java	Interpret JAVA programs
javac	Compile JAVA programs
javap	Disassemble JAVA programs
javah	Include 'C' header files.
jdb	Debug Java Programs
javadoc	Create HTML documents
appletviewer	View JAVA Applets

3.9.2 Application Programming Interface (API)

Java library contains many packages which contains number of methods and classes grouped into API given below:

3.9.2.1 Language support Package

It contains many classes and methods for supporting **basic features** of Java.

3.9.2.2 Utilities Package

This package contains classes which support the **date** and **time.**

3.9.2.3 Input and Output Packages

It contains the collection classes which support the **input** and **output** operation.

3.9.2.4 Networking Package

Networking package contains collection classes which support the **communication process** with other computers via Internet.

3.9.2.5 AWT Package (Abstract Window Toolkit)

This classes support the implementation of platform-independent Graphical User Interface (GUI).

3.9.2.6 Applet package

This package contains the number of classes to create and view various **Applets**.

These class libraries summarize their primary function. These are grouped into various packages. A Package is a collection of classes. The various JAVA class libraries or packages are shown below:

Package	Used to
java.io	Supports Input and Output
java.applet	Supports to build Applets
java.net	Supports to enable Networking.
java.util	Supports to Utilities
java.lang	Supports to Java functionality.
java.beans	Supports for Java software components
java.awt	Supports to design Abstract Window Tool kit (AWT) to construct GUI environment.
java.awt.event	Support to handle Events from AWT components
java.awt.image	Support to perform Image processing

3.8 STRUCTURE OF JAVA PROGRAM

Java progam is the construction of classes and class contains data members and methods that operates on. The following structure shows the JAVA program.

```
Documentation section
Package section
Import section
Interface section
Class definition
Main method class
{
        Main method definition;
}
```

Where,

Documentation section: This section usually contains the comment lines used to specify the name of the program, author and other details etc. This section is optional.

Package section: This section is used to declare the package name that are used to inform the compiler, that classes defined here belongs to this package. This section is optional.

Import section: This section is used to import the specified class or package like as we include a file in C++ using **#include** statement. This section is optional.

Interface section: This section is used to implement the interface concept and includes a group of method declaration. This section is optional.

Class definition: The class is the primary and essential element of a JAVA program and the JAVA program contains multiple class definitions. These classes are used to map the objects. This is also optional.

Main method definition: The **main( )** specifies the starting point of a Java program. This is essential part of the Java program. This is used to create objects of various classes and also used to establish communication between them.

3.8.1 First JAVA Program

Open a text editor (Usually notepad) to create a file that contains the program named as **First.Java.** The file name must match the class that we declare. **JAVA** is case sensitive so be careful while entering both the name of the file and its contents.

Eg:

Indicates that this method is associated with first class

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

Where,

class Every JAVA program must contains atleast one class specification. Here a new class **First** is declared.

main( ) All the JAVA application begins execution at **main( )**, it accepts one argument called **arg** that is an array of **Strings** object. The Interpreter starts the execution from the **main( )**.

public: Is the access specifier, indicates that the method can be called by code, outside the **First** class.

3.2 FEATURES OF JAVA

The designers of JAVA Programming language, wanted to develop a language which offers solutions to some of the problems encountered in modern programming language. Then the team of Sun Microsystem describes the JAVA with the following features.

1. Object Oriented Programming language.
2. Platform independent and Portable language.
3. Simple and small to unit programs.
4. Multithreading and Inheritance features.
5. Quick response that is dynamic and extensible.
6. Strong, safe and secure.
7. Distributed Language
8. Interpreted Language

3.3 DIFFERENCE BETWEEN JAVA AND C++

The **JAVA** language was modelled after the C++ language. The **JAVA** does not offers some of the features available in C++ for the benefit of programmers. Some of the few major differences are given below:

1. Pointers concept not available in **JAVA.**
2. **JAVA** does not provides header files.
3. Operator overloading is not possible in **JAVA**.
4. Multiple Inheritance in **JAVA** is accomplished by a special feature is called **Interface**.
5. Instead of Destructors, **JAVA** offers **finalize( )** function.
6. There is no Template classes in **JAVA**.

3.4 BYTECODE AND JAVA VIRTUAL MACHINE (JVM)

The normal program exists in two forms

(i) Source Code (ii) Object Code

Source code: Is the textual version of the program written by the programming using a text editor.

Object code: The executable form of a program is called object code. This object code is executed by the computer.

This object code is different to a particular CPU, it can not be executed on a different platforms. JAVA eliminates this restriction by using Bytecode.

Byte Code: Like all other Computer languages, the **JAVA** program begins with source code and compiled it then and produces an object file containing byte code instead of creating object code. This bytecodes are not specific for particular CPU. These byte codes are designed to be interpreted by a **JAVA VIRTUAL MACHINE (JVM).** These bytecodes are executed by **JVM** on any platforms.

3.5 APPLICATIONS AND APPLETS

Using the JAVA Programming, we can build two types of programs.

i) Application: These applications are directly executed by Java Virtual Machine (**JVM**).

ii) Applets: Applets require a web browser to execute. This browser includes a JVM and also provides an execution environment for the applets. These are typically downloaded from a web server to a user machine.

3.6 CLASSES AND OBJECTS

The **Classes** and **Objects** are the basic building blocks of a **JAVA** program.

Class: A Class is a template from which objects are created i.e. objects are instances of a class. It contains variables declaration and methods declaration (and/or definitions).

Object: Object is a variable of the type of Class. It is a region of storage, that defines both **state** and **behaviour** and storage can be memory or disk.

State is a set of variables and contents they contain.

Behaviour is a set of methods and the logics they implemented. Thus, the object is a combination of data and associated methods.

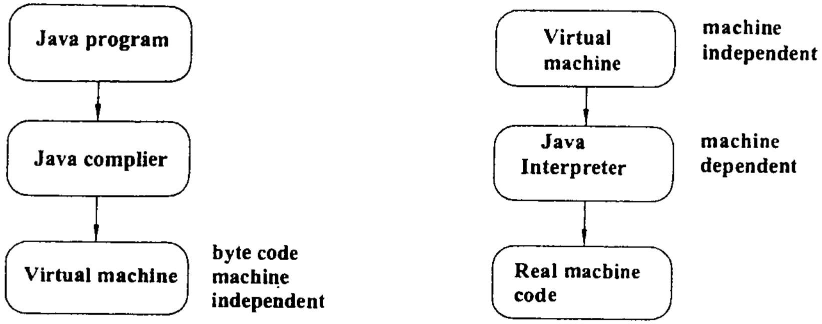
3.7 JAVA CLASS LIBRARIES

JAVA provides a rich set of class libraries, that provides a large number of classes and methods, that any **JAVA** program can use it.

These libraries are used for Input/Output, Mathematical, Networking, Events and many other capabilities.

3.10 COMPILATION AND RUNNING A JAVA PROGRAM

The Java source code is compiled by the Java compiler. Through the compiler the source code is converted into bytecode. This bytecode for a machine is called **Java Virtual Machine(JVM),** which is stored in computer memory. This Java Virtual Machine code is machine independent. This virtual machine code is converted into real machine code which is machine dependent by the Java interpreter.



3.11 VARIABLES

A Variable is a named memory location used to hold the content. All variables must be declared before they can be used. Variable declaration specifies the type and scope of variable.

3.11.1 Declaration of a Variable.

Syntax:

```
datatype var1,var2,...,varn;
```

Where,

datatype is the type of the data

var1,var2,...,var; are the list of variables.

A Variable can be choosen and declared by the programmer in a meaningful way, so it is very easy to identify what it represents in the program.

Eg:

```
String stdname;
int mark1, mark2, mark3;
float Arg;
```

3.11.2 Rules for Naming a variable

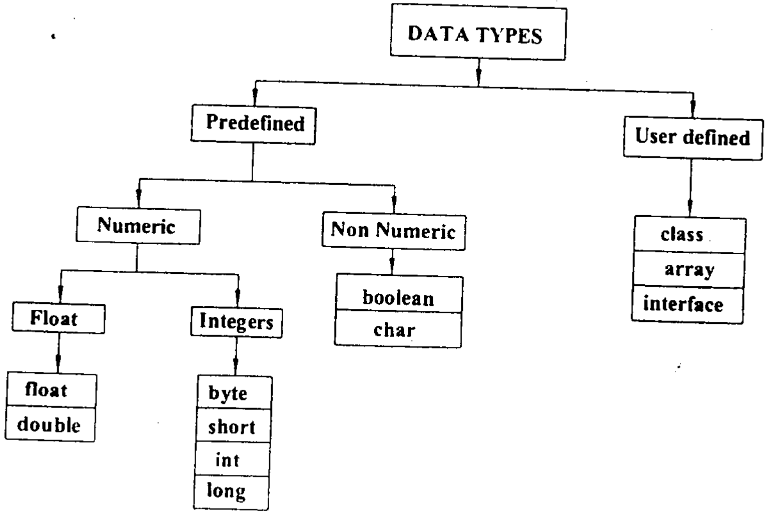
1. It must begin with alphabet.
2. It is a case sensitive i.e., the variable **NAME** is different from **name**.
3. The keywords not used as variables.
4. There is no blank space and special character in between the variable.
5. The length of the variable can be any length.

3.12 DATA TYPES

Data type is the type and size of the data, that we are going access with in the program.

The Data types in JAVA

The Java data types hierarchical structure is shown in below. The Java data types are mainly divided into two types, and those can be further divided as shown in figure below.



SIZES AND RANGES OF DATATYPES

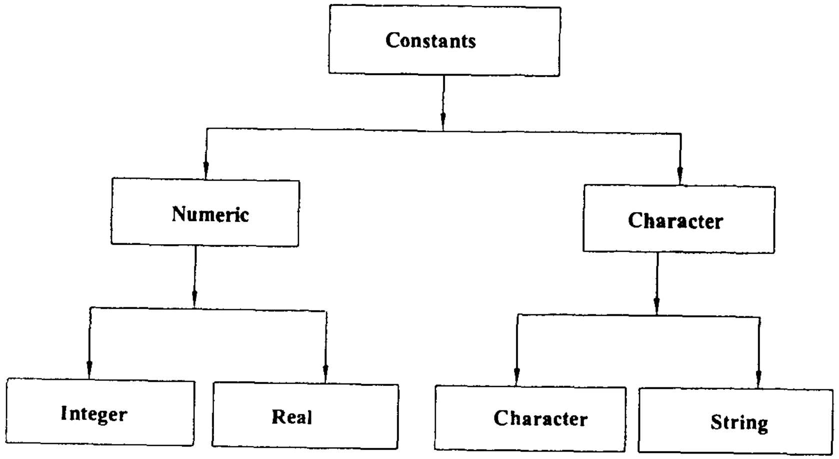
	Data type	Size in bytes	Range
8 bits	byte	1	–128 to +127
16 bits	short	2	–32,768 to +32,767
32 bits	int	4	–2,147, 483,648 to 2,147,483,647
64 bits	long	8	-9,233,372,036,854,775,808 to 9,223,372,036,854,775,807
32 bits	float	4	3.4E-38 to 3.4E+38
64 bits	double	8	1.7E-308 to 1.7E+308

Data type	Description
char	Used to hold 16 bit character data
int	Used to hold 32 bit signed integer numbers
float	Used to hold 32 bit signed floating point numbers.
double	Used to hold 64 bit signed floating point numbers.
short	Used to hold 16 bit signed integer numbers.
byte	Used to hold 8 bit signed integer numbers.
long	Used to hold 64 bit signed integer numbers.
boolean	Used to hold true or false values.

3.13 CONSTANTS

Constant is a fixed value, which does not vary during the execution. A Value assigned to a Variable is called Constant.

JAVA Constants



3.13.1 Numeric Constants

Numeric constants are formed with the help of digits only. These can be divided into two types.

i) Integer Constants: These constants are formed with the help of digits and which contains only integer part.

Eg: **123;**
045;
-23;

ii) Real Constants: These constants are formed with the help of digits and contains decimal part.

Eg: **12.75;**
-15.50;

3.13.2 Character Constants

Character constants are formed with the help of character only. These can be divided into two types.

i) Character Constants: The single character within single quotes called character constants.

Eg: **‘m’;**
‘y’;

ii) String Constants: The set of characters quoted in double quotes are called String constants.

Eg: **“MUNI”**
“GURU”
“LAK”

3.13.3 Backslash Character Constants

These character constants are also called as the escape sequences or execution character set. These are visibles in the program and not visible in output, but it carry out some specific operation in the output.

Constant	Meaning
‘\n’	New line
‘\b’	Backspace
‘\f’	Form feed
‘\r’	Carriage return
‘\t’	Horizontal tab
‘\’’	Single quote
‘\”’	Double quote
‘\\’	Back slash

For declaring the constants put variable name to the left of the “=” and put the value you want to assign to the variable to the right of the “=”. This assignment must be terminated by a semicolon. The constant does not vary during the execution.

Syntax:

```
variablename=value;
```

Eg: **n=10;**

3.14 OPERATORS

Operator is a symbols, which can do a particular operation on the operands.

Eg: **a+b;**

Where,

+ is the operator
a,b are the operands.

3.14.1 Arithmetic Operators

Arithmetic operators are used to carry out all the arithmetic operations.

JAVA Arithmetic operators

Operator	Meaning
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

3.14.2 Relational Operators

The Relational operators are used to compare the operands. The following are the relational operators available in JAVA.

Operator	Meaning
= =	Equal to
=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

3.14.3 Logical Operators

The logical operators are used to combine two or more relational expressions. The outcome of the Logical operator is boolean value.

Operator	Meaning
&&	Logical AND
||	Logical OR
==	Logical equal
!	Logical NOT
!=	Logical NOT equals

3.14.4 Increment and Decrement Operators

These are called unary operators, because they acts upon only one operand. i.e., These are used to increment or decrement only one operand content.

'++' is used to increment an operand.

'- -' is used to decrement an operand.

3.14.5 Ternary Operator

JAVA contains a ternary operator, that acts as an abbreviated form of an **if..then...else** statement.

Syntax:

```
variablename=expr1?expr2:expr3;
```

If the **expr1** is true the value of the **expr2** is taken, if it is false then the value of the **expr3** is taken.

3.14.6 Bitwise Operators

Bitwise operators are used to carry out operations on bits (Binary digits). JAVA provides, three categories of bitwise operators.

(i) Logical (ii) Shift (iii) Assignment

These operators may be applied only to Java's **int, char, byte, short int** and **long** data types.

Bitwise logical operators	
Operator	**Meaning**
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
~	Bitwise complement
!	Bitwise NOT

Shift operators	
Operator	**Meaning**
>>	shift right with sign extension
>>>	shift right with zero fill
<<	shift left with zero fill

There are another operators that combine a bitwise operations with assignment as shown below.

Operator	Meaning
&=	AND assignment
|=	OR assignment
^=	exclusive OR assignment
>>=	shift right with sign extension and assignment
>>>=	shift right with zero fill and assignment
<<=	shift left with zero fill and assignment

3.14.7 Special Operators

The JAVA provides two special operators that are used to manipulate on classes and methods of classes.

1. **Instanceof operator**
2. **Dot operator (.)**

3.14.7.1 The instanceof operator

This operator is used to find whether a object belongs to a particular class. It returns true if the object on the left hand side is belongs to the class on the right hand side.

Syntax:

```
object instanceof class;
```

The above statement is true, if the **object** belongs to the right hand side of the **class**. Otherwise it returns false.

3.14.7.2 The dot operator

This operator (.) is used to access the instance variable and methods of class objects.

Eg:

```
student.Mark;
student.Total( );
```

3.14.8 Operator Precedence and Associativity

If there are more than one operator used in an expression, then the operators are executed according to their precedence. If the operators have same precedence. Then they are evaluated either from left to right or right to left, depending on the level of precedence. The following table shows the precedence and its associativity of an operator.

Operator	Rank	Associativity
. ( ) [ ]	1	L to R
- ++ -- ! ~ type	2	R to L
* / %	3	L to R
+ -	4	L to R
<< >> >>>	5	L to R
< <= > >= instanceof	6	L to R
== !=	7	L to R
&	8	L to R
^	9	L to R
|	10	L to R
&&	11	L to R
||	12	L to R
?:	13	R to L
= op=	14	R to L

3.15 JAVA KEYWORDS

Keywords are the predefined words that are used to construct the statement in a language. Since, keywords having specified meaning, so they cannot be used as a variable. The Java keywords are given below:

Java Keywords

abstract
boolean
break
byte
cast
catch
char
class
const
continue
default
do
double
else
extends
final
finally
float
for
goto
if
implements
imports
instanceof
int
interface
long
native
new
package
private
protected
public
return
short
static
super
switch
synchronized
this
throw
throws
tangent
try
void
volatile
while

3.16 EXPRESSIONS

An expression is a combination of operators and operands arranged as per the syntax of the language. An expression may appear on the right side of an assignment statement.

Syntax:

```
variablename=expression;
```

Eg:

```
sum=a+b;
```

These Java expressions may contain variable, constants or both of them.

3.16.1 Type Conversion during Evaluation

In an expression the lower type is automatically converted into higher type. For an example if **char, byte, int,** and **long** are in an expression then the result is promoted to **long** in order to avoid the overflow.

Automatic Type conversion

	char	byte	short	int	long	float	double
char	int	int	int	int	long	float	double
byte	int	int	int	int	long	float	double
short	int	int	int	int	long	float	double
int	int	int	Int	int	long	float	double
long	long	long	long	long	long	float	double
float	float	float	float	float	float	float	double
double	double	double	double	double	double	double	double

We can convert the type using type casting is called forced conversion.

(typename)Expression;

Eg: a=(int)3.5

The value 3.5 is converted into integer by truncation.

3.17 CONDITIONAL STATEMENTS

In a program all the instructions are executed sequentially by default, when no repetition of some calculations are necessary. When in some situation we may have to change the execution order of statements based on condition or to repeat a set of statements until certain conditions are met. In such situation Conditional and Control statements are very useful.

'JAVA' provides the following conditional (decision making) statements.

i) *if* statement

ii) *if....else* statement

iii) *switch* statement

3.17.1 The *if* statement

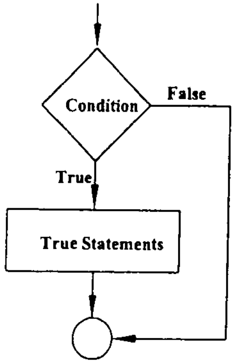
It is used to control the flow of execution of the statements and also used to test logically to find whether the condition is true or false.

Syntax :

```
if(condition is true)
{

        True statements;

}
```



If the condition is true, then the true statements are executed. The ***'True statements'*** may be a single statement or group of statements. If the condition is false then the true statements are not executed, instead the program skip past it. The condition is given by the relational operator like = =, != , < =, > =, etc.

Eg:

```
class ifc
{
        public static void main(String args[ ])
        {
                int i=5;
                if (i<10)
                        System.out.println("The Value of i is Less than 10");
        }
}
```

OUTPUT

The Value of i is Less than 10

If you want to execute multiple statements with in the ***if*** statements, that statements must be blocked with in braces ({ }).

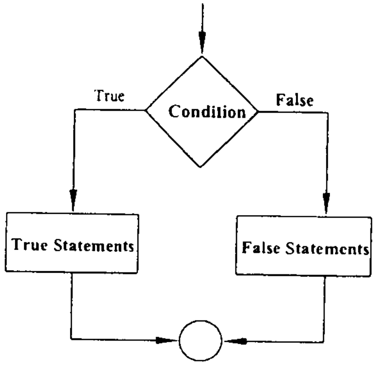
3.17.2 The *if...else* statement

It is used to control the flow of execution and also used to carry out the logical test and then pickup one of the two possible actions depending on the logical test.

It is used to execute some statement when the condition is true and execute some other statements, when the condition is false.

Syntax :

```
if(condition)
{
        True statements;
}
else
{
        False statements;
}
```



Eg :

```
import java.io.*;
class con1
{
public static void main(String args[ ])
{
try
{
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter Value for i...");
int i=Integer.parseInt(br.readLine( ));
if (i<10)
            System.out.println("The Numbr is Less than 10");
else
            System.out.println("The Number is Greater  than 10");
}
catch(IOException e)
{
      System.out.println(e);
}
}
}
```

OUTPUT

```
Enter Value for i 15
The number is Greater than 10
```

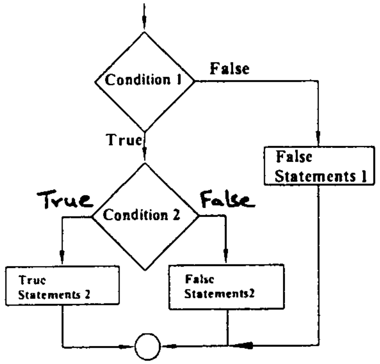
There is only one statement in the **if** block (or) **else** block the braces are optional. But if there is more than one statement we must use the braces.

3.17.3 Nested *if....else* statement

When a series of ***if..else*** statements are occurred in a program, we can write an entire ***if..else*** statement in another ***if..else*** statement called nesting, and the statement is called ***nested if.***

Syntax :

```
if(condition 1)
{
        if(condition 2)
        {
                True statement2;
        }
        else
        {
                False statement2;
        }
}
else
{
                False statement 1;
}
```



Eg:

```
import java.io.*;
class con2
{
public static void main(String args[ ])
{
try
{
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter Value for i...");
int i=Integer.parseInt(br.readLine( ));
if (i==10)
      System.out.println("Play Cricket");
else
           if (i==15)
                System.out.println("Play Foot Ball");
           else
                System.out.println("Don't Play");
      }
      catch(IOException e)
      {
           System.out.println(e);
      }
      }
}
```

OUTPUT

```
               Enter Value for i...25

               Don't Play
```

Note that the second *if-else* statement is nested in the first *else* statement. If the condition in the first *if* statement is *false*, then the condition in the second *if* statements is checked, if it is false the final *else* statement is executed.

Similarly, we can construct any number of *if..else* statements in nested fashion, that is called *if..else ladder.*

If you want to test more than one condition in *if* statement the logical operator are used as specified below. These are used to combine the results of two or more conditions.

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

Eg:

```
import java.io.*;
class con3
{
public static void main(String args[ ])
{
try
{
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter Value for i...");
        int i=Integer.parseInt(br.readLine( ));
        if ((i>10) &&(i<20))
            System.out.println("Number Between 10 and 20 ");
        else
            System.out.println("Number > 20 or < 10");
    }
    catch(IOException e)
    {
        System.out.println(e);
    }
}
}
}
```

OUTPUT

```
          Enter Value for i....15
          Number between 10 and 20
```

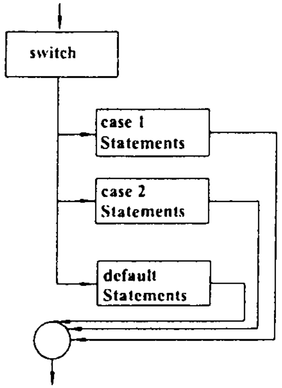
3.17.4 The *switch* statement

The **switch** statement is used to pickup or execute a particular group of statements from several available group of statements. It allows us to make a decision from the number of choices.

It is a multiway decision statement, it tests the value of given variable or expression against a list of case values and when a match is found, a block of statements associated with that **case** is executed.

Syntax:

```
switch(expression)
{
        case constant 1 :
                    block1;
                    break;
        case constant 2 :
                    block2;
                    break;
                    .
                    .
                    .
        default :
                    default block;
                    break;
}
```



The integer expression following the keyword ***switch*** is any *'JAVA'* expression that must yield an integer value. It must be an integer constant like 1,2,3 or an expression that evaluates to an integer.

The keyword ***case*** is followed by an integer or a character constant. Each constant in each case must be different from all the other.

First the integer expressions following the keyword ***switch*** is evaluated. The value it gives is searched against the constant values that follow the *case* statements. When a match is found, the program executes the statements following the ***case***. If no match is found with any of the ***case*** statements, then the statements following the ***default*** are executed.

Rules for writing switch( ) statement

i) The expression in ***switch*** statement must be an integer value or a character constant.

ii) No real numbers are used in expression.

iii) Each ***case*** block and ***default*** blocks must end with ***break*** statements.

iv) The ***default*** is optional

v) The ***case*** keyword must terminate with colon (:).

vi) No two ***case*** constants are identical.

Eg:

```
import java.io.*;
class con4
{
public static void main(String args[ ])
{
try
{
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter Value for i...");
int i=Integer.parseInt(br.readLine( ));
switch(i)
{
          case 1:
               System.out.println("i in case 1");
               break;
          case 2:
               System.out.println("i in case 2");
               break;
          default:
                    System.out.println("i in default");
                    break;
}    }
     catch(IOException e)
     {
          System.out.println(e);
     }
}
}
```

<u>**OUTPUT**</u>

```
Enter Value for i....2
I am in case 2
```

Here, for variable *'i'* match is found in *case 2*, hence the statements in *case 2* is executed.

3.18 CONTROL STATEMENTS - LOOPS

In some situations there is a need to repeat a set of instructions in specified number of times (or) until a particular condition is being satisfied. This repetitive operations are done through a loop control structure.

The following are the loop structures available in 'JAVA'

i) *while..*

ii) *do...while*

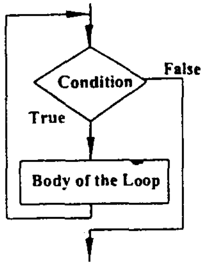
iii) *for...*

3.18.1 The *while* loop

It is a repetitive control structure, and executes the statements with in the body until the condition becomes false.

Syntax:

```
while(condition)
{
        ........
        body of the loop ;
        ........
}
```



The statements within the while loop will be executed till the condition is true, when the condition becomes false the control passes to the first statement that follows the body of the *while* loop.

Eg: Addition of numbers upto 10

```
class con5
{
        public static void main(String args[ ])
        {
                int i=1,sum=0;
                while(i< =10)
                {
                        sum=sum+i;
                        i=i+1;
                }
                System.out.println("Sum is upto 10 ..."+sum);
        }
}
```

OUTPUT

```
        The sum is upto 10.... 55
```

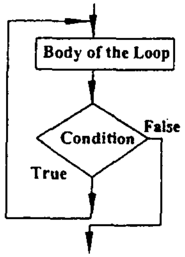
Here, first check whether *'i'* is less than or equal to 10. At first *i=1* so the condition is true, now ***sum*** is added with the value of *'i'* and get **sum** as one, than *'i'* is incremented by 1, then it checks the condition again. This process is going on till the condition is false. When the condition is false the **System.out.println( )** statement is executed and display the result.

3.18.2 The *do..while* loop

It is also repetitive control structure and executes the body of the loop once irrespective of the condition, then it checks the condition and continues the execution until the condition becomes false.

Syntax:

```
do
{
        ..........
        body of the loop;
        ..........
} while(condition);
```



Here, the statements with in the body of the loop is executed once, then it checks for the condition, if it is true, then it executes body until the condition becomes false.

Eg: Addition of numbers upto 10.

```
class con6
{
public static void main(String args[ ])
{
 int i=1,sum=0;
do
{
        sum=sum+i;
        i=i+1;
}
while(i< =10);
System.out.println("Sum of the numbers upto ..."+sum);
} }
```

OUTPUT

```
Sum of the numbers upto 10 ...55
```

Difference between ***while*** and ***do while*** statements.

while	do....while
1. This is the top tested loop	1. This is the bottom tested loop
2. The condition is first tested, if the condition is true then the block is executed until the condition becomes false.	2. It executes the body once, after it checks the condition, if it is true the body is executed until the condition becomes false.
3. Loop is not executed if the condition is false.	3. Loop is executed atlease once even though the condition is false.
4. It has no termination.	4. It has the termination.

3.18.3 The *for* loop

The *for* loop is another repetitive control structure, and is used to execute set of instruction repeatedly until the condition becomes false.

The assignment, incrementation and decrementation and condition checking is done in *for* statement only, where as other control structures are not offered all this features in one statements only.

Syntax :

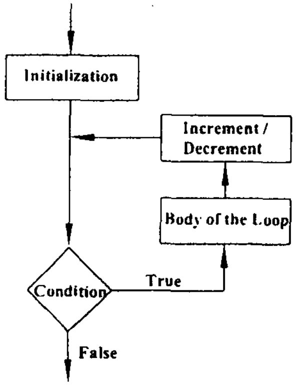
```
for(initialise counter;test condition;increment /decrement counter)
{
...
body of the loop;
...
}
```

for loop has three parts

1. ***Initialise counter*** is used to initialize counter variable.

2. ***Test condition*** is used to test the condition.

3. ***Increment/decrement counter*** is used to increment or decrement counter variable.



If there is a single statement within the *for* loop, the blocking with braces is not necessary, if more than one statement in body of the loop, the statements within the body must be blocked with braces.

Eg: Addition of number upto 10

```
class con7
{
        public static void main(String args[ ])
        {
                int sum=0;
                for (int i=1; i<=10;i++)
                        sum=sum+i;
                System.out.println("The addition of numbers upto 10 is..."+sum);

        }
}
```

OUTPUT

```
The addition of numbers upto 10 is... 55.
```

Note that the initialisation, testing and incrementation of loop is done in the *for* statement itself.

3.18.4 Nested *for* loops

Like *if* statement *for* loop also nested. The loop with in the loop is called nested loop.

To understand how nested loops work, look at the program given below

```
class con8
{
        public static void main(String args[ ])
        {
                for (int i=1; i<=3;i++)
                {
                        System.out.println("\n");
                        for (int j=1;j<=3;j++)
                                System.out.println("\t"+j);
                }
        }
}
```

OUTPUT

```
        1    2    3
        1    2    3
        1    2    3
```

Here for each value of *'i'* the inner loop is executed completely, with in the inner loop *'j'* taking from 1 to 3. The inner loop terminates when the value of *'j'* is greater than 3. Where *'\n'* causes the control to the next line for each entry of the *'i'* loop.

3.18.5 The *break* statement

The ***break*** statement is used to terminate the loop. When the keyword ***break*** is used inside any 'JAVA' loop, control automatically transferred to the first statement after the loop. A ***break*** is usually associated with an *if* statement

Syntax :

break;

Eg:

```
class con9
{
        public static void main(String args[ ])
        {
                for (int i=1; i<=10;i++)
                {
                        System.out.print(i);
                        if (i==6)
                        break;
                }
}
```

OUTPUT

```
        1 2 3 4 5 6
```

Here the ***System.out.print()*** statement print value of *'i'* upto 6 when *'i'* reaches 6 the ***if*** statement is true. So the ***break*** statement transfers the control to the outside of the ***for*** loop.

3.18.6 The *continue* Statement

In some situations, we want to take the control to the beginning of the loop, bypassing the statements inside the loop which have not yet been executed, for this purpose the ***continue*** is used. When the statement ***continue*** is encountered inside any 'JAVA' loop control automatically passes to the beginning of the loop.

Syntax:

continue;

Like ***break*** statement the ***continue*** statements also associate with *if* statement.

Example : Calculate the sum of the given positive numbers.

```
import java.io.*;
class con11
{
public static void main(String args[ ])
{
try
{
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
int sum=0;
for(int i=1;i<=5;i++)
{
          System.out.println("Enter Value for n...");
          int n=Integer.parseInt(br.readLine( ));
          if (n<0)
               continue;
          else
               sum=sum+n;
     }
     System.out.println("GIven Numbers Sum is..."+sum);
     }
     catch(IOException e)
     {    System.out.println(e);
     } } }
```

Here, it asks the 5 numbers, for each number it checks whether it is less than zero, if it is less than zero it can not add and asks for next number (i.e. the control will automatically transfer to the *for* loop using the ***continue***), if it is greater then zero, it adds to the variable ***sum,*** finally after completing the loop it display the results.

3.19 MATHEMATICAL FUNCTIONS

The Mathematical functions are used to carry out mathematical operation that are frequently occured.

The following mathematical functions are available in the **java.lang** in **math** package.

Functions	Actions
sin(a)	Returns the sine of the angle at 'a' in radians
cos(a)	Returns the cosine of the angle at 'a' in radians
tan(a)	Returns the tangent of the angle at 'a' in radians
asin(a)	Return the angle whose sine is 'a'.
acos(a)	Return the angle whose cosine is 'a'.
atan(a)	Return the angle whose tangent is 'a'.
pow(a,b)	Return the 'a' to the power of 'b'.
exp(a)	Return the value 'e' raised to 'a'.
log(a)	Returns the value of natural logarithm of 'a'.
sqrt(a)	Returns the square root of 'a'.
ceil(a)	Returns the smallest value whose number greater than or equalto 'a' rounding up value.
floor(a)	Returns the greatest value whose number greater than or equal to 'a' rounding up value
rint(a)	Truncate the value of 'a'.
abs(a)	Returns the absolute value of 'a'.
max(a,b)	Returns the maximum value between 'a' and 'b'.
min(a,b)	Returns the minimum value between 'a' and 'b'.

Eg: The use of math function.

Syntax:

```
Math.functionname( );
```

Eg:

```
int a=Math.pow(x,y);
```

Return power of 'x' to the 'y' and assigns the result to 'a'.

3.20 CLASSES

A class is a user defined data type like structure in C language. In the class the members (or data types) and methods (member function that use the data) are defined. Once the class is defined we can create the object which are called instances of classes.

Syntax:

```
class classname
{
    variable declaration;
    methods declaration;
}
```

Note: Here note that there is no semicolon at the end of the brace but in the C++ language there is a semicolon at the end of the brace.

3.20.1 Declaring variables in a class

The variable declared in a class at that instances, no storage space has been created in the memory. The variable declared inside the class also called instance variable.

Syntax:

```
class student
{
    int rollno;
    int age;
}
```

Where, The class **student** contains two types of instances or variables of the type of **int**.

3.20.2 Declaring and defining methods in a class

A method which declared inside the class uses the data field had declared in the same class. This method are declared after the variable declaration.

Syntax :

```
type functionname(parameters)
{
    function body;
}
```

Where, The **type** specify the return type of the calling function from the called function. This could be a **int, float, char** (or) even be a **void**. The parameters list contains the data types and variable names which are passed into the function.

Eg:

```
class student
{
        int rollno;
        int age;
        void input(int r, int a)
        {
             rollno=r;
             age=a;
        }
}
```

In the above example the function **input(int r, int a)** return no value. So the function is declared as a **void**.

3.20.3 Creating objects

Creating an object is the process of allocating a block of memory that contains space to store all the instance variables of the class.

We can create the object by using the **new** keyword. The keyword **new** create an object for the specified class and return the reference to that object.

Syntax :

```
classname   objectname;              //object declaration
objectname=new classname( );         //Reference to the class name object
```

Eg:

```
student computer;
computer=new student( );
```

In the above statement the first statement declare the object. The second statement assign the object reference to the variable.

We can combine the two statements into one statement, as shown below:

```
classname objectname=new classname( );
```

3.20.4 Accessing class members

To access the class members, we must assign some values before accessing it in the program. The following statement shows how the objects are assigned.

Syntax:

```
objectname.variablename;
objectname.functionname(parameters);
```

Eg:

```
computer.rollno=100;
computer.age=50;
```

If you create another object

```
student chem;
chem=new student( );
```

Then

```
chem.rollno=200;
chem.age=75;
```

Another way of assigning values is by passing the parameters to function.

We can assign values through the function like,

```
computer.input(100,56)
```

It assign values to **rollno** and **age** through the **input** function.

3.20.5 Constructors

Constructor is a method that initialize each object when it is created. Constructor function have the same name of the class. In the previous approach uses the dot operator to access the instance variable and then assign them individually, so we can use the constructor to initialize objects.

Eg:

```
class add
{
      int a,b;
      add(int x,int y)          //constructor method
      {
          a=x;
          b=y;
      }
```

```
        int addition( )
        {
            return(a+b);
        }
}
class print
{
        public static void main(String args[ ])
        {
            add ob=new add(100,25);
            int sum=ob.addition( );
            System.out.println("sum="+sum);
        }
}
```

OUTPUT

```
        Sum=125
```

3.20.6 Methods overloading

Method overloading is the process of polymorphism. It means using the same function name for different purposes, the appropriate function is executed depending upon the argument passed to it.

Eg:

```
class Input
{
        int a,b;
        Input(int x, int y)
        {
            a=x;
            b=y;
        }
        Input(int x)
        {
            a=b=x;
        }
        int area
        {
            return(a*b);
        }
}
```

In the above example, we are overloading the **Input( )** constructor

If you want to find the area of the rectangle.

```
        Input I1=new Input(10,5);
            I1.area( );
```

If you want to find the area of the square.

```
        Input I2=new Input(10);
            I2.area( );
```

3.20.7 Static members

If you want to access a variable or method in all the classes, declared in the program then declare that variable or method as a **static**. In the **class** the **ordinary variable** and **methods** are called **instance variables** and **instance methods**. The **static variables** and **methods** in the **class** are called **class variables** and **class methods**. If you want to call the **static** variables (or) methods, no need for the object we can call directly. In Java **Math** class has number of **static** methods we can directly call it like **Math.square(2);**

Eg:

```
class mathfunction
{
        static int square(int a)
        {
            return(x*x);
        }
        static int double(int y)
        {
            return (y*2);
        }
}
class application
{
        public static void main(String arg[ ])
        {
            int a=mathfunction.square(4);
            int b=mathfunction.double(2);
            System.out.println("a="+a);
            System.out.println("b="+b);
        }
}
```

OUTPUT

```
        a=16
        b=4
```

Even though, we call the **static** methods without using the **object**. It has some restriction.

Restrictions of using static variables or methods

1. The **static** methods can only use **static** data.
2. The **static** method can only call another **static** method.
3. They cannot use as a super.

3.20.8 Nesting of methods

We can call a method by another method of the same class is called nesting methods.

Eg:

```
class addition
{
        int x,y;
        addition(int a, int b)                 //constructor method
        {
            x=a;
            y=b;
        }
        int add( )
        {
            return (x+y);
        }
        void display( )
        {
            int sum=add( );                    //calling a method
            System.out.println("Sum of value="+sum);
        }
}
class mainin
{
        public static void main(String arg[ ])
        {
            addition a=new addition(10,20)
            a. display( );
        }
}
```

OUTPUT

```
        Sum=30
```

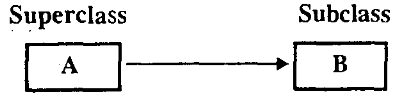
3.21 INHERITANCE

Inheritance is the process deriving a new class(sub class) from an existing class(superclass). The new class has the properties of the existing class with its own details.

This Inheritance property make the reusability of the existing class. The new class is called **subclass** (or) **child class**. The existing class is called **parent class** (or) **superclass**. There are many types in inheritance.

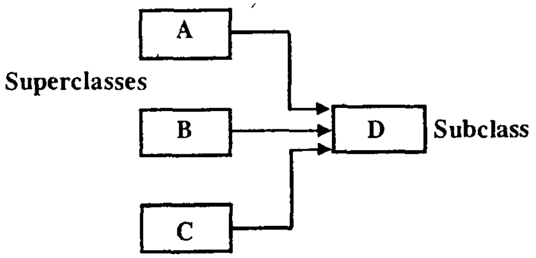
3.21.1 Single Inheritance

It is the process of deriving a subclass from only one superclass is called Single Inheritance (Single sub class from single superclass).



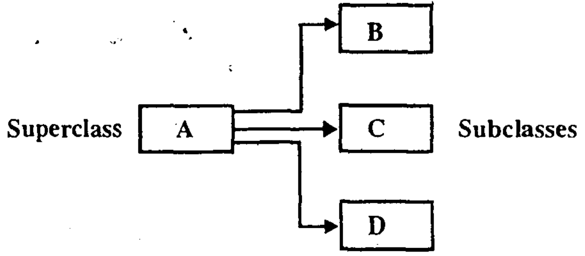
3.21.2 Multiple Inheritance

It is the process of deriving a subclass from many superclasses called Multiple Inheritance (Single sub class from many superclasses).



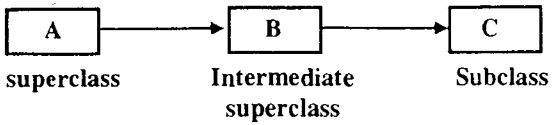
3.21.3 Hierarchical Inheritance

The superclass has many subclass is called hierarchial inheritance (Many sub classes from single superclass)



3.21.4 Multilevel Inheritance

It is the process of deriving a subclass from another subclass is called Multilevel inheritance. (Subclass from another sub class)



In the above example the subclass is derived from another derived class. Here derived class 'B' act as a base class for the subclass 'C'.

3.21.5 Defining a Subclass

The keyword **extends** inherits the property of superclass to the subclass. Subclass also has its own variables and methods. Now subclass will contain the property of superclass and its own property.

Syntax:

```
class subclassname extends superclassname
{
        variable declaration;
        methods declaration;
}
```

Eg:

```
class Add
{
        int x,y;
        Add(int a, int b)   // constructor method
        {
                x=a;
                y=b;
        }
        int addition( )
        {
                return (x+y);
        }
}
```

```
class exadd extends Add
{
        int z;
        exadd(int a, int b, int c)
        {
                super (a,b);   → calls constructor method in base class.   //passes value to the superclass.
                z=c;
        }
        int exaddition( )
        {
                return (x+y+z);
        }
}
class mainin
{
        public static void main(String ar[ ])
        {
                exadd a1=new exadd(10,20,30);
                int sum1=a1.Add( );     || X → addition     //superclass method
                int sum2=a1.exadd( );                       //subclass method
                System.out.println("sum1="+sum1);
                System.out.println("sum2="+sum2);
        }
}
```

OUTPUT:

```
        Sum1=30
        Sum2=60
```

The keyword **super** is used to call the constructor method of the superclass. **super** keyword must appear as the first statement of the subclass constructed. The parameter that passed from the **super( )** must match the order and type of the instance variables declared in the superclass.

3.21.6 Method Overriding

The method defined in superclass is inherited by its subclass and can be used through the object of subclass. This type of method Inheritance enables us to define and invoke methods repeatedly in subclasses without defining it again in subclass.

If, there may be a situation, when an object want to respond same method with different behaviour by calling the method. This is the process of override the method defined in superclass.

This overriding is possible by defining a method in subclass similarly defined in superclass. By Invoking the method, the subclass method is executed instead of superclass method.

Eg:

```
class A
{
        int x,y;
        A(int a,int b)
        {
              x=a;
              y=b;
        }
        void show( )
        {
              System.out.printin("x and y", +x""+y)
        }
}
class B extends A
{
        int z;
        B(int a,int b, int c)
        {
              super(a,b);
              z=c;
        }
        void show( )
        {
              System.out.println("Z;"+z);
        }
}
class mainin
{
        public static void main(String a[ ])
        {
              B showl=new B(10,20,30)
              show1.show( );
        }
}
```

<u>OUTPUT</u>

```
        Z: 30
```

In the above example when **show( )** method is invoked the superclass **show( )** method is overridden by subclass **show( )**, so it only display the Z value. If you want to display the superclass **show( )** method. Then

```
class B extends A
{
        int z;
        B(int a, int b, int c)
        {
              super(a,b)
              z=c;
        }
        void show(c)
        {
              super.show( );
              System.out.println("C:"+z);
        }

}
```

Then we get the output of superclass method.

If the arguments of the superclass **show( )** is not same as the **show( )** of the subclass then in that case, it is **Function Overloading.**

Eg:

```
class B extends A
{
        int z;
        B(int a, int b, int c)
        {
              super(a,b);
              z=c;
        }
        void show(String msg)
        {
              System.out.println(msg+z);
        }
}
```

```
class mainin
{
        public static void main (String arg[ ])
        {
              B showl=new B(10,20,30)
              showl.show( );
              showl.show("this is 2");
        }
}
```

OUTPUT

```
        x and y 10 20
        This is z 301
```

3.22 FINAL VARIABLES AND METHODS

By default all the variables and methods are overridden by subclass. To prevent the overriding of the members of a superclass, the keyword **final** is used.

If you declare the variable and method as **final** then it will never be modified in anyway.

Eg:

```
final int x=1;
final void show( )
{
      statement 1;
      statement 2;
      ....
}
```

If you declare the function as **final** it will never be modified.

3.22.1 Final classes

We can declare class as a **final** class for security reasons. If you declare class as a **final** class then it is not possible to derive subclass from that **final** class. If you attempt this the compiler will give some error.

3.22.2 Finalizer( ) methods

The **finalizer**( ) method is similar to **destructor** in C++, Java automatically frees the memory used by the object using garbage collecting system. Objects may has no resources, such as file descriptors or windows system fonts. The garbage collecting system cannot free these resources. To free this **finalizer()** method is used. When you call **finalizer**( ) at the end of class it will free the memory resource, which is occupied by the non-object resources, such as file descriptions (or) window system fonts etc.

3.22.3 Abstract Methods and Classes

Abstract method does the opposite job of **final**. When you declare a class as a **abstract** class, the methods in the class must be re-define in the subclass, thus making overriding process.

3.23 VISIBILITY CONTROL

JAVA provides five types of visibility modifiers or access modifiers as described below:

3.23.1 Public Access

If you specify the variables (or) methods as **public**, then that variables (or) methods can be accessed everywhere in the program.

3.23.2 Friendly Access

When you don't specify anything the default visibility is **friendly** level of access. The friendly access makes the fields visible only in the same package but in the case of **public** access the fields visible in all the packages. Packages is the group of related classes.

3.23.3 Protected Access

If you mention as **protected** class, then the fields are visibled in all the classes and subclasses in the same package, but also to subclasses in other packages.

3.23.4 Private Access

If you mention **private** class, then it is just like the **final** class we cannot inherit the class. They are accessible only within their own class.

3.23.5 Private protected Access

This access modifier specifies the visibility control in between the **protected** and **private** access. This makes the fields visible in all subclasses.

3.24 ARRAYS

Array is a group of related data items, that are stored in the contiguous memory location and in same name.

3.24.1 Declaration of Array

Like, ordinary variable, the array can also be declared before its scope.

Syntax:

```
datatype variablename[ ];
(or) data type[ ] variable name;
```

Here, we just declare the array. In the declaration, we don't mention the size and no memory space is created.

3.24.2 Creating memory space

After declaring the array the **new** keyword is used to create memory space for the array that already declared.

Syntax:

```
Array_name=new datatype[size of the array];
```

Eg:

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

3.23.3 One dimensional array

The list of item can be stored under a single variable with only one subscript such variable is called one dimensional array.

Syntax:

```
datatype variablename[ ]=new datatype [size of the array];
```

Eg:

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

The above statement create 3 space for a[ ] in the memory. If you store a[0]=1,a[1]=2,a[2]=3 .Then the datas are stored like this

```
a[0]=1
a[1]=2
a[2]=3
```

The above statement creates 3 memory space for the array variable **a[ ]**, We can combine the above two syntaxes in one statement, i.e., declaration and allocation of memory space.

```
Datatype arrayname=new datatype[size];
```

3.24.4 Initialization of arrays

The process of assigning values to array location is called initialization. We can initialize the array elements like ordinary variables.

Eg:

```
a[0]=1;
a[1]=2;
a[2]=3;
```

In JAVA, if you assign more than one value to the same variable like a[0]=1,a[0]=2, then the compiler will give an error.

Syntax:

```
type arrayname[ ]={values};
```

Where, values must be separated by commas.

Eg:

```
int marks[ ]={90,85,70,75}
```

Eg: Program to fine the ascending order

```
class ascending
{
public static void main(String arg[ ])
{
     int no={30,10,50,40};
     int len=no.length;
     System.out.println("The Given numbers:");
     for(int i=0; i<len;i++)
     {
          System.out.println(" "+no[i]);
     }
```

```
            for(int i=0; i<len; i++)
            {
                for(int j=i+1; j<len; j++)
                {
                    if(no[i]>no[j])
                    {
                        int t=no[i];
                        no[i]=no[j];
                        no[j]=t;
                    }
                }
            }
            System.out.println(“Ascending list”);
            for (int i=0; i<len; i++)
            {
                System.out.println(“ “+no[i]);
            }
        }
        }
```

OUTPUT

```
The given numbers
30      10    50    40
Ascending List
10      30    40    50
```

3.24.5 Two Dimensional Array

Two dimensional array is an array of one dimensional array. If you want to store 10 student marks and their Roll Nos, then use two dimensional array.

Eg:

```
int student[ ][ ]=new int [10][10];
int no[ ][ ]={{1,2,3},{4,5,6},{7,8,9}};
```

The above statement is stored in memory like this.

	Column [0]	Column [1]	Column [2]
Row [0]	1 [0][0]	2 [0][1]	3 [0][2]
Row [1]	4 [1][0]	5 [1][1]	6 [1][2]
Row [2]	7 [2][0]	8 [2][1]	9 [2][2]

Note, here all the datas are stored in the contiguous memory location.

Eg: Displaying Matrix form

```
class matrix
{
    public static void main(String args[ ])
    {
        int mat[ ][ ]={{1,2},{3,4}};
        System.out.println(“MATRIX”);
        for(int i=0; i<2; i++)
        {
            for(int j=0; j<2; j++)
            {
                System.out.print(“\t”+mat[i][j]);
            }
            System.out.println(“\n”);
        }
    }
}
```

OUTPUT

```
MATRIX
1     2
3     4
```

3.25 STRINGS

String represents a sequence of characters and, string manipulation is the most common part of JAVA program. The sequence of characters are stored in JAVA by using **String** class.

JAVA offers two class to manipulate strings, namely **String** and **StringBuffer**.

Syntax:

```
String stringname;
stringname=new String("string");
```

The above two statement may be combined in one statement as follows:

```
String stringname=new String("string");
```

Eg:

```
String name=new String("VRB");
```

We can get the string length as follows:

```
int l=name.length( );
```

We can easily concatenate the string using the '+' operator.

```
name=namefirst+namesecond;
Name= "first"+"Second";
```

3.25.1 String Array

The arrays of string can also be created

Syntax:

```
String stringname=new String[size of the array];
```

Creates a string array of specified size

Eg:

```
class stringdisplay
{
        static String name[ ]={"VRB","MUNI","GURU"};
        public static void main(String args[ ])
        {
              int len=name.length;
                     for (int i=0;i<len;i++)
              {
                     System.out.println(name[i]);
              }
        }
}
```

OUTPUT

```
VRB
MUNI
GURU
```

3.25.2 StringBuffer class

String class is usefui for fixed length string. **StringBuffer** class is used for flexible length and to modify both the content and length. By using this class, we can - insert characters and substring in the middle of the string, or append another string to the end of the string.

Eg:

```
class bufferstring
{
         public static void main(String args[ ])
         {
               Stringbuffer s1=new Stringbuffer("Happy Day");
               System.out.println(+s1);
               System.out.println("length of the string"+s1.length);
               for(int i=0;i<s1.length;i++)
               {
                      int a=i+1;
                      System.out.println("The"+a"character is"+s1.charAt(i));
               }
         String s2=new String(s1.toString( ));
         int P=s2.indexOf("day");
               s1.insert(P,"birth");
               System.out.println("attached"+s1);
               s1.setCharAt(5,'-');
               System.out.println("after set char"+s1);
               s1.append("to you");
               System.out.println("string after appended"+s1);
}
}
```

OUTPUT

```
Happy day
The string length is 9
The 1 character is H
The 2 character is a
The 3 character is p
The 4 character is p
The 5 character is y
The 6 character is
The7 character is d
The 8 character is a
The 9 character is y
The string s2 Happy day 6
Altered string Happy birthday
After setchar Happy_birthday
String after appended Happy_birth day to you
```

3.26 VECTORS

Vectors are similar to arrays that can hold objects of any type and any number. Vector class contained in the **java.util** package.

Syntax:

```
Vector a=new Vector(3);
Vector a=new Vector( )'
```

In the first statement we mention the size of the vector. If the size exceeds, the **Vector** class automatically increase the size.

The primitive data type can not be used directly in vectors, like **int, float, long, char** and **double**. So we need to convert these data types to objects using **Wrapper** class.

Methods in Vector class

Methods	Performance
list.addElement(object)	It adds the object to the list at the end.
list.element(5)	It gives the name of the 5th objects
list.size( )	It gives the number of objects
list.removeElement(object)	It removes the specified object from the list
list.removeElementAt(5)	It removes the object stored in the 5th position of the list
list.removeAllElements( )	It removes all the subject from the list.
list.copyInto(array)	It copies all the objects from the list to array
list.insertElementAt(object,2)	It inserts the items at 2nd position

Eg: Illustration of Vector

```
import java.util.*;                                   //Importing Vector class.
class Exvector
{
        public static void main(String args[ ])
        {
                Vector a =new Vector( );
                int len=args.length;
                for(int i=0;i<len;i++)
                {
                        a.addElement(args[i]);
                }
                a.insertElement("COMPUTER",2);
                int len=a.size( );
                String arraylist[ ]=new String[len];
                a.copyInto(arraylist);
                System.out.println("Subject list");
                for(int i=0;i<len;i++)
                {
                        System.out.println(arraylist[i]);
                }
        }
}
```

Give the Input in command line.After the compilation of the program give the input and displays the output as follows:

```
D:\java\java Exvector physics Maths Biology
Subject list
physics
COMPUTER
Maths
Biology
```

3.27 WRAPPER CLASS

Wrapper class is used to convert the primitive data types such as **float, int, char** and **double** into objects. The wrapper class is contained in **java.lang.**

Primitive type	Wrapper class
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long

The Wrapper class contains the number of methods for handling the primitive data types and objects as shown below:

	Conversion	
Obstructor	**Conversion**	**Action**
1) Integer	**Intobj=new Integer(iv);**	Convert the integer to integer object.
2) Float	**floatobj=new Float(fv);**	Convert the float to float object
3) Double	**Doubleobj=new Double(dv);**	Convert the double to double object
4) Long	**Longobj=newLong(lv);**	Convert the long to long object.

Converting the objects into primitive numbers	
1) **int iu=Intobj.intValue( );**	Convert the Integer object to primitive Integer.
2) **float fv=Floatobj.floatValue( );**	Convert the float object to primitive float.
3) **double dv=Doubleobj.doubleValue( );**	Convert the double object to primitive double.
4) **long lv=Longobj longValue( );**	Convert the Long object to primitive long.

Converting primitive numbers to string() method	
1) **S=Integer.toString(iv);**	Convert the Integer to string
2) **S=Float.toString(fv);**	Convert the float to string
3) **S=Double.toString(dv);**	Convert the double to string.
4) **S=Long.toString(lu);**	Convert the long to string.

Converting String Objectto numeric object

1)	**Intobj=Integer.Valueof(S);**	Convert the string object into int object
2)	**Floatobj=Float.Valueof(S);**	Convert the string object into float object
3)	**Doubleobj=Double.Valueof(S);**	Convert the string object to double object.
4)	**Long obj=Long.Valueof(S);**	Convert the string object to long object.

Converting string to primitive numbers

1)	**int i=Integer.parseInt(S);**	Convert the string to primitive Integer.
2)	**long l=Long.parseLong(S);**	Convert the string to primitive long.

Eg:

```
import java.io.*;
class example
{
        public static void main(String arg[ ])
        {
                int n=0;
                try
                {
                        DataInputStream in=new DataInputStream(System.in);
                        System.out.println("Enter the number");
                        String str=in.readLine( );
                        n=Integer.parseInt(str);
                }
                catch(IOException e)
                {
                        System.out.println(e);
                }
                System.out.println("The Entered number"+n);
        }
}
```

OUTPUT

```
Enter the number
3
The Entered number 3
```

3.28 INTERFACES

Java doesn't support the Multiple Inheritance. But in real life examples the usage of multiple inheritance is important. In C++ it is very complicate and difficult. In Java the multiple inheritance is done by using **interface**. The interface is also like class in **Java**. But the variable field must be **final** field and the method must be **abstract** method.

Syntax:

```
interface interfacename
{
        static final datatype variablename=value;
        returntype methodname(parameter list);
}
```

The method declaration will contain only the declaration and ends with semicolon. No codes within the method.

The class that implements the **interface** must have the code for the method. On the above, variable declaration the **static** and **final** are optional. Eventhough you didn't mention the variable as **static** and **final**, the interface class automatically assign the variables as constant.

Instead of the above variable declaration we can declare like this

datatype variablename=value;

3.28.1 Extending interface

Like classes, the interfaces can also be extended, means the interface can be subinterfaced from other interface. This is achieved by using the **extends** keyword.

Syntax:

```
interface subinterfacename extends superinterfacename
{
    variable declaration;
    abstract method declaration;
}
```

We can derive the subinterface from more than one interface using comma operator.

Eg:

```
interface Add
{
        int a=10;
        int b=20;
}
interface method
{
        void display( );
}
interface addition extends add, method
{
        ....
}
```

The above superinterface (or) subinterface the method codes must be definded only in the class. It is not possible to extend the interface with class it violates the rules of interface. So if you attempt this the compiler will give an error.

3.28.2 Implementing the interface

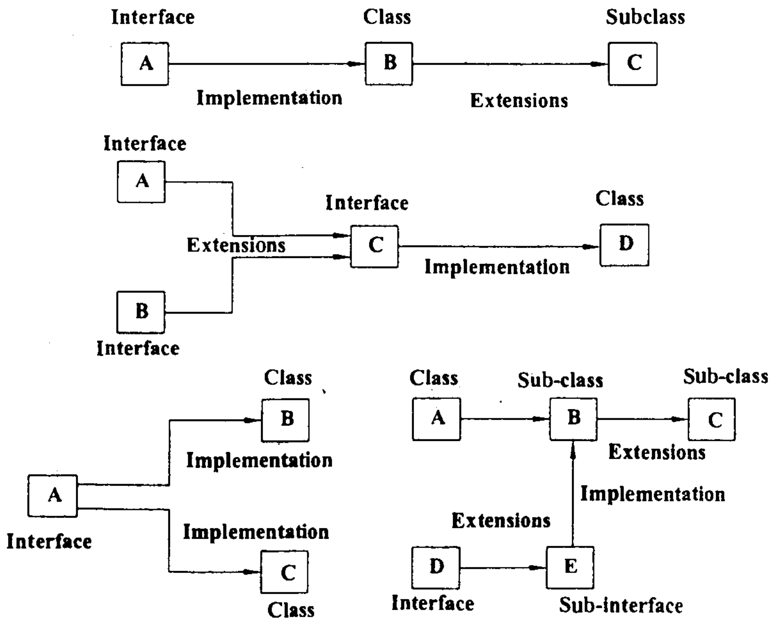
Interfaces are used ~~in~~ as superclass, whose properties are acquired or inherited by classes.

Syntax:

```
class classname implements interfacename
{
        variable declaration;
        method codes( );
}
```

In the above syntax the interface is used as a superclass (or) inherited by class.

The various methods of interface implementation is shown below:



Interface implementation in various forms

Eg:

```
interface Addition                     //Interface declared and defined
{
        int a=50;
        int manipulation(int x, int y);
}
class add implements Addition          //Interface Implementation
{
        public int manipulation(int x, int y)
        {
                return (a+x+y);
        }
}
class sub implements Addition
{
        public int manipulation(int x, int y)
        {
                return (a-x-y);
        }
}
class mainin
{
        public static void main(String angs[ ])
        {
                add add1=new add( );
                sub sub1=new sub( );
                Addition addition;
                addition=add1;
                System.out.println("Addition of three-
                                - numbers"+addtion.manipulation(10,20));
                addition=sub1;
                System.out.println("Substraction of three-
                                - number"+adddition.manipulation(10,0);
          }
}
```

OUTPUT

```
        Addition of three numbers 80
        Subtraction of three number 20
```

In the above example first the interface **Addition** is created after that, class objects **add** and **sub** created. In the **mainin** class, we created the instance of each class using the **new** operator.

Then, we declared the object **addition** of interface **Addition**. Then, we assign the reference to the **add** class object **addl** to **addition**. When, we called the **manipulation** method of **addition**, the **manipulation** method of **add** class is invoked.

3.29 PACKAGES

Java packaging is a concept similar to class libraries in other languages. Packages provide resusablity of another program classes, without physically copying it or without declaring it again.

Benefits of packages

1. By using the packages, we can reuse other program classes.
2. It provides hiding the classes, so other programs (or) packages cannot access the classes.

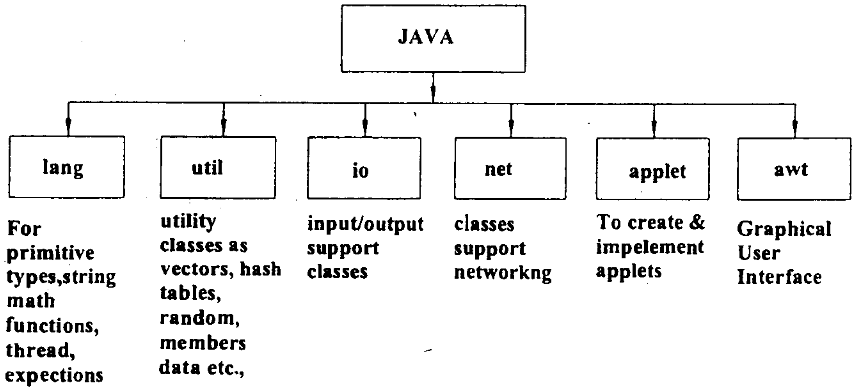
The JAVA packages are divided into two types.

Java API packages

User defined packages.

3.29.1 API (Application Programming Interface) Package

The API packages provides a large number of classes grouped into different packages. These packages are oftenly useful in most of our JAVA programming. These packages are also called System packages described below:

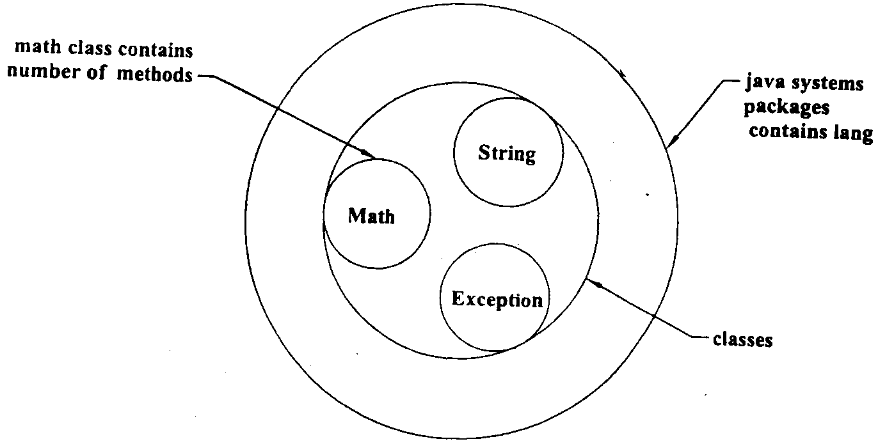

JAVA
lang
util
io
net
applet
awt
For primitive types,string math functions, thread, expections
utility classes as vectors, hash tables, random, members data etc.,
input/output support classes
classes support networkng
To create & impelement applets
Graphical User Interface

3.29.2 Java System Packages and their Contents

Package	Contents
java.lang	Containing the language support classes. These are automatically imported.
java.util	Containing the utilities classes, like vectors, tables, date, time etc.
java.io	Containing the Input/Output support classes.
java.awt	Containing the classes for implementing the Graphical User Interface (GUI) environment.
java.net	Containing the classes for Networking.
java.applet	Containing the classes for manipulating Applets.

3.29.3 The Hierarchial Packages Structure

The hierarchical structures of JAVA packages are shown below:


math class contains number of methods
java systems packages contains lang
String
Math
Exception
classes

If you want to access a class in the system package. There are two methods available. First method using dot method.

Eg:

java.awt.colour;

This method is the easiest way, when you access only one class. In many circumstances we use more than one class. By using the **import** method we can import required package.

Syntax:

```
import packagename.classname
```

Eg:

```
import java.awt.font;                //Imports font classes
```

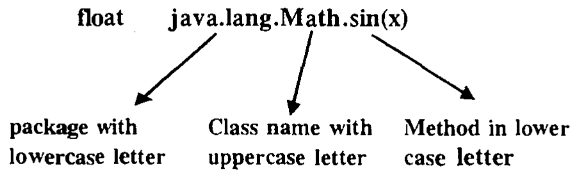
If you want to access all the classes, use the '*', to import.

Eg: **import java.awt.*;**

3.29.4 Naming method

Packages are named by following the Java naming rules.

Eg:


float java.lang.Math.sin(x)
package with lowercase letter
Class name with uppercase letter
Method in lower case letter

The above example invokes the **sin(x)** method of class **Math.**

For convection, all the packages in lower case letters and classes are in upper case letters.

3.29.5 User Defined Packages

We have learned about the system package, that are available, now we are going to discuss about the user defined package. These packages must be declared at the begining of the program except for comments and white spaces.

Syntax:

```
package packagel;                    //package declaration
public class classl;                 //class definition
{
      .....
      normal declarations;
      .....
}
```

3.29.6 Rules for creating user defined Java Packages

1. Declare the package at the beginning of the source file.
2. Declare the class as **public** that is used in the package.
3. Create a sub directory under the main directory, where the source files stored. Name the sub directory same as the package name.
4. Store the file in sub directory as same as the class name which declared as **public.**
5. Compile the file. This creates **.class** file in the subdirectory.

Then the packages are included in your program using the dot operator as follows.

```
Package firstpackagename.secondpackagename
```

Remember that the larger package subdirectoryname must be

```
firstpackagename\secondpackagename
```

3.29.7 Accessing user defined packages

The system packages are accessed in a program by using **import** keywords. Like system packages user defined packages are imported.

Syntax:

```
import packagel.package2.classname;
```

Here **package2** is inside the **packagel** and **class** is inside the **package2.**

3.29.8 Working with a package

Eg:

```
package packl;
public class A
{
      public void display( )
      {
            System.out.println("I am in class A");
      }
}
```

Store the above program in a sub directory **packl** and name the file as **classA.java**. Then compile the file. It will create the class file **classA.class**.

```
import packl.*;
class packtest
{
        public static void main(String arg( )]
        {
                A a=new A( );
                a.display( );
        }
}
```

Store this file in **packl** sub director, then compile and run.

3.29.9 Subclasses using Packages

```
import packl.A;                    //imports the superclass A
class B extends A )
{
        public void displayB( );
        {
                System.out.println("I am in classB");
        }
}
class classextend
{
        public static void main(String arg[ ])
        {
                B b=new B( );
                b.display( );
                b.displayB( );
        }
}
```

OUTPUT

```
I am in ClassA
I am in Class B
```

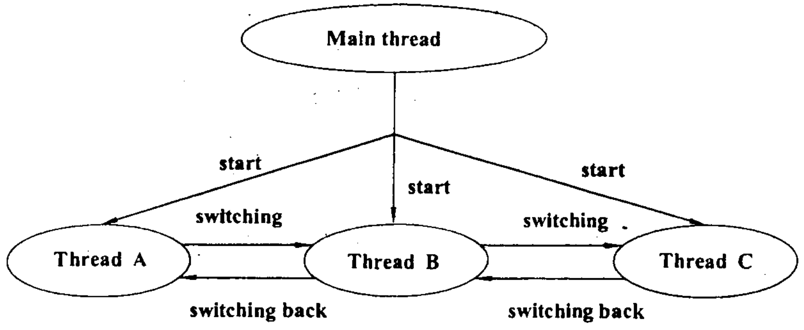
3.30 MULTI THREADED PROGRAMMING

The ability to execute several programs at a time is know as multithreaded programming. The small program is called **thread**. The threads running in parallel, does not mean that they all execute in the same time. The Java interpreter responsible for switching between the threads. A program that contains multiple flows of control is known as multithreaded programming.

The concept multithreaded programming makes the JAVA programming differ from other programming languages. It is useful to enable the programmer to do multiple things at one time. They can be divided into threads and execute them in parallel.

The multithreaded ability supports the concept of concurrency, means threads in subprogram and main program shares the same memory space, and are called as the **lightweight threads** or **lightweight processes**.

This concept is very preferable to use at the same time to do two or more things.



3.30.1 Creating threads

Threads can be created in the form of object by using the **run( )** method. This **run( )** method is the heart and soul of any thread.

Syntax:

```
public void run( )
{
        .....;
        .....;
}
```

Methods to create thread.

1. Define a class that extends thread class and override its **run( )** method with the code required by the thread, this process is called creating threads using threads class.
2. Define a class that implements runnable interface. The runnable interface has only one method, **run( )**, that is to be define in the method with the code to be executed by the thread. This process is called, creating threads by converting a class to a thread.

3.30.2 Extending the thread class

The class can be invoked as thread by extending the class **java.lang.thread.** This is used to access all the thread methods directly. It includes the following:

1. Declare the class as an extending class for the thread class.
2. Define the **run( )** method in the class, that is responsible for executing the sequence of code, that the thread will execute.
3. To execute the thread create a thread object and call the **start( )** method.

3.30.3 Declaring thread class

The extended thread class can be declared as follows:

Syntax:

```
class threadname extends Thread
{
        .....
        .....
        .....
}
```

3.30.4 The run( ) method

In order to execute our thread the existing one is override by the **run( )** method.

Syntax :

```
class threadname extends Thread
{
public void run( )
        {
                .....;
                .....;
        }

}
```

3.30.5 Starting new thread

We can start a new thread using the **start( )** method.

Syntax:

```
threadname objectname=new threadname( );
objectname.start( );
```

The first statement create a new object at class thread name, the second statement calls the **start( )** method to put a thread in runnable state.

We can combine the above statements into one statement as follows.

```
new threadname( ).start( );
```

Eg:

```
class A extends Thread
{
        public void run( )
         {
         char ch='A';
         for(int a=1;a<=5;a++,ch++)
         {
           System.out.println("\t From Thread A:"+ch);
         }
        System.out.println("Exit from A");
         }
}
```

```
class B extends Thread
{
        public void run( )
          {
        char cha='a';
        for(int b=1;b<=5;b++)
        {
           System.out.println("\t From Thread B:"+cha);
        }
        System.out.println("Exit from B thread");
        }
}
class Thread1
{
        public static void main(String sr[ ])
        {
        A threadA=new A( );
        B threadB=new B( );
        System.out.println("Start the Thread a");
        threadA.start( );
        System.out.println("Start the Thread b");
        threadB.start( );
        System.out.println("End of main thread");
        }
}
```

OUTPUT

```
        Start the Thread a
        Start the Thread b
        End of main thread
                From Thread A:A
                From Thread B:a
                From Thread A:B
                From Thread B:a
                From Thread A:C
                From Thread B:a
                From Thread A:D
                From Thread B:a
                From Thread A:E
                From Thread B:a
        Exit from A
        Exit from B thread
```

3.30.6 Stopping and blocking a thread

We can stop the running thread by simply calling the **stop( )** method.

Syntax:

```
threadname.stop( );
```

Actually, when the thread ends execution it automatically sent to the dead state as well as, when we call the **stop( )** method the thread sends to the dead state.

If you want to block or suspend a thread from execution. Then use one of the following method.

sleep( );
suspend( );
wait( );

If you call any one of the above method the thread is sent to block state. If you call **sleep( )** method after the particular time elapsed the thread is came to runnable state.

If you call **suspend( )** method, then the thread is came to runnable state after calling **resume( )** method.

If you call **wait( )** method, the thread will come to runnable state after you call **notify( )** method.

3.30.7 Various states of thread

During the execution time or lifetime of a thread, it can enter any one of the states as specified **below**:

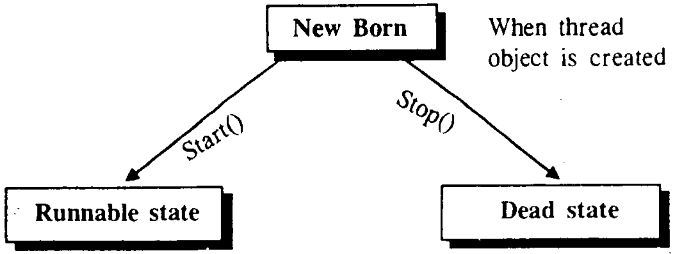
1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

3.30.7.1 Newborn state

When, we create the object of thread, then the thread is born, this is called newborn state. At this stage we can do only two operations.

1. We can start the method using **start( )**. Schedule it for running using start)
2. Destroy it using **stop( )** method.

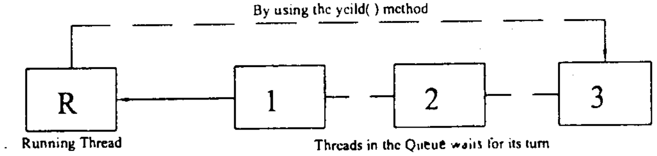
If you do any other operation an exception will be thrown



3.30.7.2 Runnable state

It means the thread is ready to execute and the thread is waiting for the availability of the processor, mean the thread joins in the queue of threads that are waiting for execution. If all the threads have equal priority then first come first serve method followed. This process of assuming time to a thread is called **time ~~slanting~~** method. If we want to transfer the control before the threads, this can be achieved by using the **yield( )** method.

Slicing


By using the yeild( ) method
R
1
2
3
Running Thread
Threads in the Queue waits for its turn

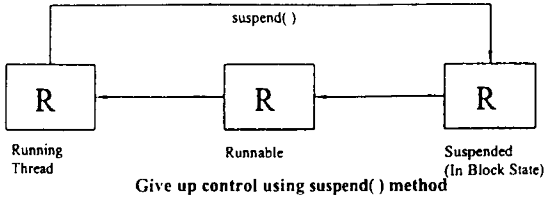
Give up control using yeild( ) method

Runnable thread.

3.30.7.3 Running state

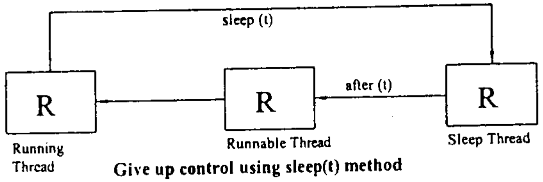
Running state is the thread which is under execution. The running thread may be give up its control in one of the following situations.

1. If you want to **suspend( )** the thread for some reason. The thread sends to the **block** state. After using the **resume( )** method the suspended thread comes to runnable state.



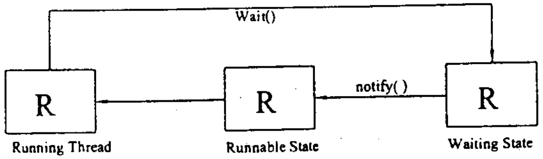
Give up control using suspend( ) method

2. If the thread is in **sleep** state using **sleep( )** for a specified period of time in milli seconds. The thread reenters the runnable state after the specified period is completed.



Give up control using sleep(t) method

3. If the thread is in wait state using **wait( )** method. Then the thread is come to runnable state using the **notify( )** method.



Glve up control using wait() method

3.30.7.4 Blocked state

If we prevented a thread into runnable state, this state is called blocked state. When the thread is suspended using sleeping (or) waiting, it is considered as **"not runnable"** state. But we must known the thread is not in dead state.

3.30.7.5 Dead state

After the thread finished its execution it sends to dead state. We can put the running thread into dead state using the **stop( )** method. We can kill it at any state (running, newborn, (or) block statement) using the **stop( )** method.

Eg: Illustration of threads methods

```
class A extends Thread
{
        public void run( )
        {
          char ch='A';
          for(int a=1;a<5;a++,ch++)
          {
          if(a==1) yield( );
          System.out.println("\t Thread A:"+ch);
          }
           System.out.println("Exit from thread A");
          }
}
class B extends Thread
{
        public void run( )
        {
          char cha='a';
         for(int b=1;b<=5;b++,cha++)
         {
           System.out.println("\t Thread B:"+cha);
           if(b==1)
           try
           {
              sleep(2000);
           }
            catch(Exception e){}
            if(b==3)stop( );
             }
        System.out.println("Exit from the thread B");
         }
}
```

```
class Threadfunctions
{
        public static void main(String args[ ])
        {
         A threadA=new A( );
         B threadB=new B( );
         System.out.println("Start the thread A");
         threadA.start( );
         System.out.println("Start the thread B");
         threadB.start( );
         System.out.println("End of the main thread");
        }

}
```

OUTPUT

```
        Start the thread A
Start the thread B
End of the main thread
        Thread A:A
        Thread B:a
        Thread A:B
        Thread A:C
        Thread A:D
Exit from thread A
        Thread B:b
        Thread B:c
```

3.30.8 Synchronization

Threads are using their own data and methods. In some situation one thread may try to read a record and another one is still writing to the same. Due to this we get the concentrated results. We can overcome this problems using the synchronization. This will keep watching on such locations, both read and write.

Syntax:

```
synchronized void update( )
{     .....
      .....
}
```

When the synchronised thread is created, the Java creates a **monitor** and handover the thread that calls the method first time.

As long as the thread holds the monitor, no other thread can be enter into the synchronised of code.

> ***Note:*** *A monitor is a like key and the thread that holds the key can only open the lock.*

Whenever, the thread completes its work using synchronized method, it will handover the monitor to the next thread that is ready to use same resource.

Whenever, two or more threads are waiting to gain control over a resource, this state is possibly called as deadlock.

3.31 EXCEPTION HANDLING

Errors are the common problems in programming, an error may produce unexpected result and abnormal termination (or) even crash the system. These errors may be classified into two types as follows:

3.31.1 Compile-Time errors

This type of errors are caused during compilation. Like missing of semicolons, misspelling of identifiers and keywords, missing double quotes in strings, type assignments, use of = in place of = =operator. Whenever the compiler causes an error, it won't create the class file. So it is necessary to edit the errors and recompile it.

3.31.2 Run-Time errors

After the successful compilation, class file created. Due to the wrong logic (or) stack overflow we cann't get the correct result during run-time. This errors are called run-time errors. Like dividing by error, converting the invalid string into number etc.

3.31.3 Exceptions

The Exception is a run-time error in the program and it creates an exception object and throws. If this exception object is not caught and handled properly, the interpreter will display an error message. In order to execute the other code, the exception object thrown by the error condition must be caught and handled correctly.

The exception handling mechanism helps the programmers to detect and report an exceptional circumstance, so that appropriate action can be taken.

The ec eption handling mechanism must follow the following tasks.

1. Detect the problem (Hit the exception).
2. Inform that problem has occured (throw the exception)
3. Receive the error information (at h the exception).
4. Take correct action (Handle the exception).

3.31.4 Java exceptions

Java provides the following predefined ec eption, that we must wat h out for at hing.

Exception	Cause of Exception
Arithmetic Exception	Math errors, **Eg:** division by zero
ArrayIndexOutOfBoundException	Due to bad array index.
ArrayStoreException	When you store the wrong type of data in an array.
FileNotFoundException	Due to an attempt to access a non existent file
IOException	Due to I/O failure.
NumberFormatException	Due to conversion failure between strings and number.
OutOfMemoryException	No Memory, when allocate a new object
SecurityException	when an applet trues to perform an action not allowed by the browser's security setting.
StockOverflowException	due to the overflow of stack.
StringIndexOutofBoundsException	When a program attempts to access a non existent character position in a string.
NullPointerException	When a reference has null object.

3.31.5 Declaring an Exception

The declaration of an exception handling includes **throwing** and **catching** the exceptions.

Syntax:

```
try
{
        statements that causes an exception;
}
catch( )
{
        statement that handles the exception;
}
```

The keyword **try** throw an exception if an error condition is met. The thrown exception is caught by the **catch** keyword. The **try** block contains one (or) more statement that could generate an exception. If any statement generate an exception, the other statements in the block are skipped and jumps to the **catch** block after the **try** block. Every **try** statement must have one **catch** statement.

Every **catch** statement has a single parameter, which is reference to the exception object thrown by the **try** block. If the **catch** parameter match with the type of exception object, then the exception is caught by the **catch** block and execute the **catch** statements. Otherwise, the default exception handler will cause the execution to terminate.

Eg: Division by zero will cause an error.

```
class exception
{
        public static void main(String args[ ])
        {
                int x=5,y=0,z=56;
        try
        {
                a=x/y;
        }
        catch (ArithmeticException e)
        {
                System.out.println("division by zero will cause an error");
        }
                y=x/z;
                System.out.println("The value at y is"+y);
        }
}
```

In the above example, when you divide by zero it will cause an error, due to the exception handling the error is handled by the **catch** block, and continues the execution.

OUTPUT

```
division by zero will cause an error.
y=1
```

3.31.6 More than one catch statement

We can use number of **catch** statements in catch block of an exception.

Syntax:

```
try
{
        Statement 1;
        Statement 2;
        ....
}
catch(exception-Type 1e)
{
        Statement;
}
catch (Exception-Type 2e)
{
        Statement;
}
        ....
        ....
```

More than one **catch** statement works like the **switch** statement. The first statement whose parameter matches with the exception object, will be executed, and the remaining statements will skipped.

Eg:

```
class error
{
        public static void main(String args[ ])
        {
        int i[ ]={5,10,20,10};
        int j=5;
        try
        {
                int z=i[4]/j-i[1];
        }
        catch(ArithmeticException e)
        {
        System.out.println("Integer Division by zero will cause an error");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
        System.out.println("The Array index is not sufficient");
        }
        catch(ArrayStoreException e)
        {
        System.out.println("you are trying to store wrong data type");
        }
        j=i[2]/i[3];
        System.out.println("j="+j);
        }
}
```

OUTPUT

```
        The Array index is not sufficient
        j=2
```

In the above program, we are handling the array more than its index. So the program will throw an exception. Which is handled by the **ArrayIndexOutOfBoundException**. So the exception is handled and the next statement followed is executed that gives the value if j.

3.31.7 The *finally* statement

The **finally** statement can be used to handle an exception that is not caught by any of the previous **catch** block, this block can be used to handle any exception within **try** block.

Eg:

```
try
{       ....
        ....
}
catch (Exception-Type 1)
{       ....
        ....
}
catch (Exception-Type 2)
{       ....
        ....
}
finally
{       ....
        ....
}
```

When the block is defined as **finally**, it is guaranty executed, without regarding whether it is thrown or not.

3.31.8 User Defined Exceptions

We can thrown our own exception by using the keyword **throw**.

Syntax:

```
throw new throwable_subclass;
```

Eg:

```
import java.lang.*;
class error1 extends Exception
{
        error1(String statement)
        {
        super(statement);
        }
}
```

```
class ourexception
{
        public static void main(String args[ ])
        {
        char ch='g';
        try
        {
                if(ch=='g')
                {
                  throw new error1("the character is 'g'");
                }
        }
        catch(error1 e)
        {
                System.out.println(e.getMessage( ));
        }
        finally
        {
                System.out.println("End of the program");
        }

        }
}
```

OUTPUT

```
        the character is 'g'
        End of the program
```

In this program, we throw our exception using the keyword **throw**. The object '**e**' contains the statement the character is '**g**' is caught by the **catch** statement. Then the **finally** block execute the statement **"End of the program"**.

3.32 APPLET PROGRAMMING

3.32.1 Applet

Applet is a small program in Java, which are mainly used in Internet. Applets are run by **Appletviewer** (or) any other web browser which support the Java. We can perform arithmetic operations, display graphics, play sounds, accept user input by using the Java applets.

3.32.2 Difference between Applet and Application program

1. Applets automatically call certain methods of applet class to start and execute the applet code. It does not contain **main( )**.
2. Applets cannot run independently. They are run from inside a web page using **HTML** tags.
3. Applets cannot read from (or) write to the files in the local computer.
4. Applets cannot communicate with other servers on the network.
5. Applets are restricted to user the libraries from other language such as C or C++.

3.32.3 Developing Java applets

Before, developing the Java applets, we must have the Java applet viewer or Java enabled web browser. We must follow the following steps to develop an applet.

1. Write the applet code. **(.java file)**
2. Create executable applet. (**.class file**)
3. Using the HTML tags create the web page.
4. Create the applet tag.
5. Incorporate the applet tag into the web page.
6. Create the HTML file.
7. Run the applet code.

3.32.4 General form of an Applet

Syntax:

```
import java.awt.*;
import java.applet.*;
.....
.....
public class applet_classname extends Applet
{
        public void paint (Graphics g)
        {
                .....
                .....
        }
}
```

In the above syntax, the first two statement **java.awt.***; and **java.applet.*** import all the classes in the package **awt** and **applet.** No need for the **main( )** function, Java automatically calls a series of **Applet** class methods for starting, running and stopping the applet code.

The method **paint( )** of the applet class is used to display the result of the applet code on the screen. The **paint( )** method required a **Graphics** object as an argument.

```
public void paint (Graphics g)
```

The package **java.awt** contains the **Graphics** class. The **Graphics** class contains the method to give all output operations of an applet.

The **Applet** class name is the main class for the applet. When the applet is loaded, Java creates an instance of this class and then a series of applet class methods are called on that instance to execute the code.

3.32.5 States of Applet

Each Java applet acquires a set of behaviours from the Applet class. The java applet undergoes number of states when it is loaded, as follows.

1. Initialization state.
2. Running state
3. Idle state
4. Displayed or dead state

3.32.5.1 Initialization state

When, the applet is first loaded, it is called as initialization state.

Syntax:

```
public void init( )
{
      .....
      .....
}
```

We can initialize the applet by using the **init( )** class. At this stage, we can do the following things.

1. Create the objects needed by the applet.
2. We can insert the images (or) fonts.
3. Initialise the values.
4. Set up colors.

3.32.5.2 Running state

The applet said to be in running state, when it is called by using the **start( )** method of applet class.

Syntax:

```
public void start( )
{
      .....
      .....
}
```

When you call the **start( )** method of applet class, the applet enters into the running state automatically after the initialization. We can use the **start( )** method to start the applet which are in idle state. So we can use more than one **start( )** method call unlike the **init( )** method.

3.32.5.3 Idle (or) Stopped state

An applet is said to be idle state or stopped state, when it is stopped from the running. We can stop the applet by calling the method **stop( ).**

Syntax:

```
public void stop( )
{
      .....
      .....
}
```

3.32.5.4 Dead state

When the applet is removed from the memory, the applet is said to be dead state. We can do this by making the **destroy( )** method.

```
public void destroy( )
{
        .....
        .....
        .....
}
```

3.32.5.5 Display state

The applet is going to display state, when we invoke **paint( )** method. This is happened in the running state to display the output.

Syntax:

```
public void paint(Graphics g)
{
    .....
    .....
}
```

3.32.6 Execution of an Applet

Compiling an applet is similar to the application program save the file as **filename.java** compile the Java file using the command **javac filename.java** then **.class** file is created. Now execute the applet using **appletviewer filename.java.**

3.32.7 Webpage Designing

To run Java applet, it is necessary to build the web page. It is basically made up of with HTML tags, that are interpreted by appletviewer or web browser. It consists of three parts.

1. Comment section 2. Head section 3. Body section

3.32.7.1 Comment section

Comments are used to identify the program element. This section contains the comments about the web page.

3.32.7.2 Head section

The Head section is usually used to give the title of the web page.

Syntax:

```
<HEAD>
            <TITLE>Java Applets</TITLE>
</HEAD)
```

The content in the **title** will appear in the **title bar** of the web browser or applet.

3.32.7.3 Body section

This section contains all about our web page. Like how it appears, color, location, sound etc.

Eg:

```
<BODY>
<CENTER><H1>"JAVA PROGRAMMING"</CENTER>
<APPLET>
</APPLET>
</BODY>
```

3.32.8 APPLET TAG

This is used to specify the name of the applet to be loaded and specifies how much space it requires.

Syntax:

```
<APPLET>
CODE=filename.class
WIDTH=500
HEIGHT=400
</APPLET>
```

In the above syntax the **<APPLET>** include the code of the applet file to be loaded and tells the compiler how much space requires for the applet. Ensure that the **HTML** file and **filename.class** must be in same directory.

Eg:

```
import java.awt.*;
import java.applet.*;
public class Example extends Applet
{
        public void paint (Graphics g)
{
        g.drawstring("Java Applet",10,100);
}
}
```

Save the file as **Example.java** and compile the file. If there is no error then **Example.class** is created. Then write HTML program as follows:

```
<HTML>
<HEAD>
<TITLE>Welcome</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1>Applet World</Hl>
</CENTER>
<BR>
<CENTER>
<APPLET CODE=Example.class WIDTH=400 HEIGHT=200>
</APPLET>
</CENTER>
</BODY>
</HTML>
```

Save the file as **Example.html** in the same directory.

To run the above program, we must have the Java supported web browser **applet viewer**. On the web browser we will be able to see the entire web page containing the **applet**. But in the **Java appletviewer** we only see the applet output on the **appletviewer.**

```
appletviewer Example.html
```

Run the **.html** file rather than .java or .class.

3.32.8.1 Applet tag in full form

The Applet tag simply creates the space in required size and then displays the output of an applet in that space. The Applet tag includes several attributes, that are very useful to the programmer to design the applets of the web page.

```
<APPLET
CODE=Appletfilename.class
[ALT=alternate-text]
[NAME=applete-instance-name]
WIDTH=specify the number in pixels
HEIGHT=specify the number in pixels
[ALIGN=alignment]
[VSPACE=pixels]
[HSPACE=pixels]
>
[<PARAM NAME=namel VALUE=valuel>]
[<PARAM NAME=name2 VALUE=value2>]
[Text to be displaced in the absence of java]
</APPLET>
```

In the above syntax **CODE, WIDTH, HEIGHT,** these all are must in the applet, the others are optional.

Table of Attributes in APPLET Tag

Attribute	Used to
CODE=Appletfilename.class	Specify the name of the applet class to be load. This attribute must be specified.
NAME=Appletinstance_name	The applet name optionally be specified, so it is easy to refer other applet on the page. This is optional.

WIDTH=Pixels
HEIGHT=Pixels
	These are used to specify the height and width of the space on the HTML file, that will reserved for applet.

Attribute	Used to
ALIGN=alignment	This is used to specify, the alignment attributes of the page such as TOP, BOTTOM, LEFT, RIGHT, MIDDLE, ABS MIDDLE, ABS BOTTOM, TEXT TOP AND BASELINE. This is optional.
VSPACE=Pixel	Used to specify, when vertical alignments is made. This is optional.
HSPACE=Pixel	Used to specify when horizontal alignments is made. This is optional.

The user must sure, the following things while adding an applet into HTML document.

1. Add an applet tag at appropriate place in web page.
2. Specify the name of the **.class** file of the applet.
3. Specify the required space to display in applet using **WIDTH** and **HEIGHT** attributes.
4. Specify, if any user defined parameters using **<PARAM>** tag.
5. Close the applet tag declaration using **</APPLET>** tag.

3.32.9 Passing parameters to applets

The user defined parameter can also be passed to an applet, using **<PARAM>** tag.

Syntax:

```
<APPLET.....>
<PARAM (NAME(or)COLOR) VALUE=....>
</APPLET>
```

Where,
Name or **colour** attributes is one of the colour and a value attribute.

Eg:

```
<APPLET....>
<PARAM=color VALUE="Blue">
</APPLET>
<APPLET...>
<PARAM NAME=text VALUE="JAVA is wonderful">
```

To receive the parameter use the **init( )** method in the applet to get hold of the parameters defined in the **<PARAM>** tags. This is done using the **getparameter( )** method, which takes one string argument representing the name of the parameter and returns a string containing the value of the parameter.

Eg: Program to illustrate the parameter passing

```
import java.awt.*;
import java.applet.*;
public class parampass extends Applet
{
        string s;
        public void init( )
        {
                s=getParameter("string");
                if(s=null)
                s="is excellent";
                s="Java"+s;
        }
        public void paint (Graphics g)
        {
                g.drawString(str,10,1000);
        }
}
```

Now create an HTML file, that is used to hold the applet as follows:

```
<HTML>
<HEAD>
<TITLE>java Applets</TITLE>
<HEAD>
<BODY>
<APPLET CODE=parampass.class
        WIDTH=400
        HEIGHT=200>
```

```
<PARAM NAME="string"
        VALUE="Applet">
<\APPLET>
</BODY>
</HTML>
```

Save the file with class name and **.html** as extension and execute the file using **appletviewer**.

Eg: **appletviewer parampass.html**

3.32.10 Displaying numeric values

To display the numeric values in an applet first convert the numeric value into string then, using the **drawString( )** method of **Graphics** class, we can easily display the numeric value in an applet. The conversion takes place using the **valueOf( )** method of **String** class.

Eg:

```
import java.awt.*;
import java.applet.*;
public class displaynum extends Applet
{
public void paint(Graphics g)
{
        int a=20;b=30;
        int addition=a+b;
        string sti="addition is"+String.valueOf (addition)
        f.drawString(str,100,1000);
}
}
```

This will displays the sum of the two numbers and displayed the result as

Addition is 50.

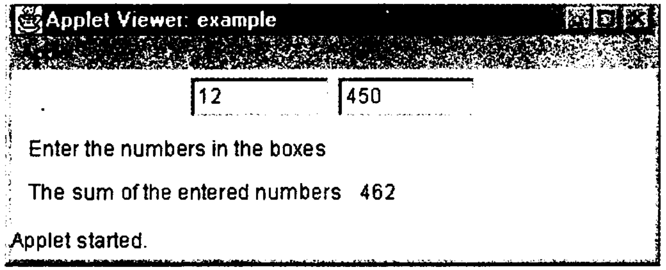
3.32.11 Giving Input to the Applet

In order to get the input from the user, we should create the text field using the **TextField** class. Then, create the objects in the **TextField** class. If you retrieve the entered text it is in string format, convert the string data according to requirements. Because Applet works on the graphical environment, therefore it treats the inputs as string.

Eg:

```
import java.applet.*;
import java.awt.*;
/*
<APPLET CODE="example" WIDTH=400 HEIGHT=400>
</APPLET>
*/
public class example extends Applet
{
TextField field1,field2;
public void init( )
{
        field1=new TextField(10);
        field2=new TextField(10);
        add(field1);
        add(field2);
        field1.setText("0");
        field2.setText("0");
}
public void paint(Graphics g)
{
int x=0,y=0,z=0;
String str1,str2,str3;
g.drawString("Enter the numbers in the boxes",10,50);
try
{
str1=field1.getText( );
x=Integer.parseInt(str1);
str2=field2.getText( );
y=Integer.parseInt(str2);
}
catch(Exception e)
{
System.out.println(e);
}
z=x+y;
str3=String.valueOf(z);
g.drawString("The sum of the entered numbers",10,75);
g.drawString(str3,200,75);
}
public boolean action(Event event,Object obj)
{
repaint( );
return true;
}
}
```

OUTPUT



```
12    450

Enter the numbers in the boxes

The sum of the entered numbers   462

Applet started.
```

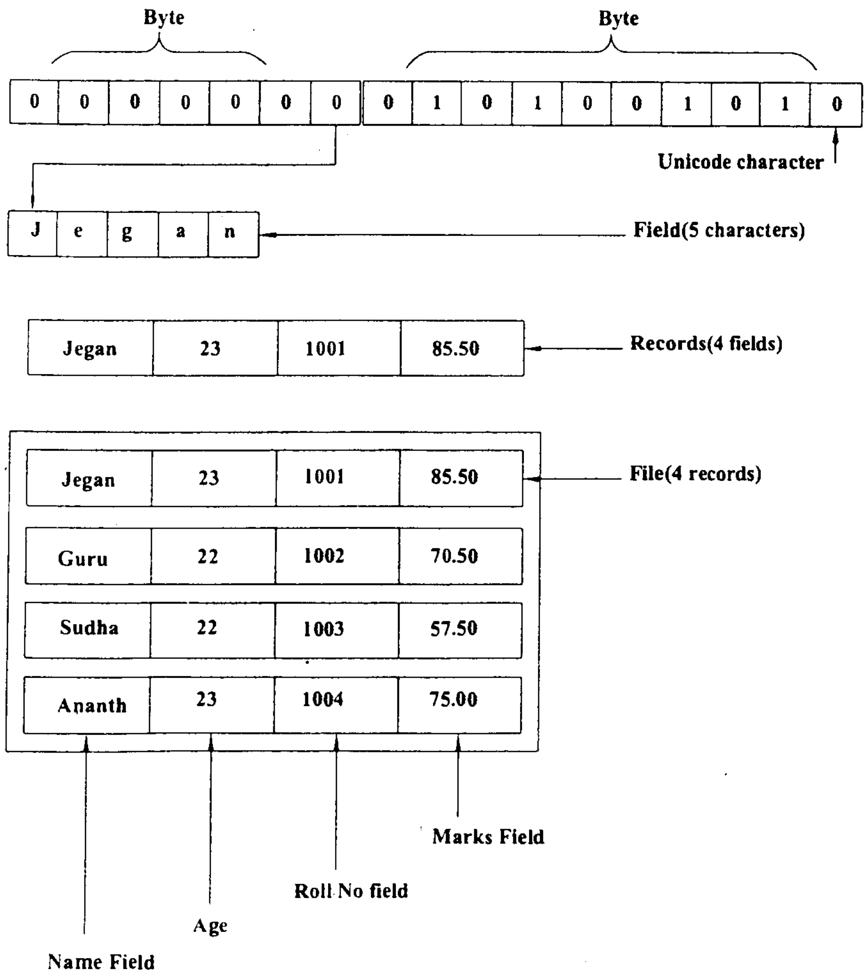
Explanation

The statement **TextField field1,field2**, creates two object **field1,field2** for the **TextField** class. **field1=new TextField(10);** allocate 10 spaces to the **field1**. **add(field1)** it adds the **field1** to the applet field. **field1.setText("Null");** Initialize the contents of the objects to zero. In the **paint** method **Str1=field1.getText( )**, get the string from the **field1** and stores in **Str1**.

3.33 FILES IN JAVA

3.33.1 File

A file is a collection of related records placed together in a particular place (in the memory or disk). A record is a composed of several fields. A field is a group of characters. In Java the characters are unicode characters. Unicode means two bytes.



Data representation in Java files