

Figure 1.5: Polymorphism

Applications of Object Oriented Programming

Main application areas of OOP are:

- User interface design such as windows, menu.
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation systems etc.

1. Client-Server Systems

Object-oriented Client-Server Systems provide the IT infrastructure, creating object-oriented Client-Server Internet (OCSI) applications. Here, infrastructure refers to operating systems, networks, and hardware. OSCI consist of three major technologies:

- The Client Server
- Object-Oriented Programming
- The Internet

2. Object-Oriented Databases

They are also called Object Database Management Systems (ODBMS). These databases store objects instead of data, such as real numbers and integers. Objects consist of the following:

Attributes: Attributes are data that defines the traits of an object. This data can be as simple as integers and real numbers. It can also be a reference to a complex object.

Methods: They define the behavior and are also called functions or procedures.

3. Object Oriented Databases

These databases try to maintain a direct correspondence between the real-world and database objects in order to let the object retain their identity and integrity. They can then be identified and operated upon.

4. Real-Time System Design

Real time systems inherit complexities that makes difficult to build them. Object-oriented techniques make it easier to handle those complexities. These techniques present ways of dealing with these complexities by providing an integrated framework which includes schedulability analysis and behavioral specifications.

5. Simulation And Modelling System

It's difficult to model complex systems due to the varying specification of variables. These are prevalent in medicine and in other areas of natural science, such as ecology, zoology, and agronomic systems. Simulating complex systems requires modelling and understanding interactions explicitly. Object-oriented Programming provides an alternative approach for simplifying these complex modelling systems.

6. Hypertext And Hypermedia

OOP also helps in laying out a framework for Hypertext. Basically, hypertext is similar to regular text as it can be stored, searched, and edited easily. The only difference is that hypertext is text with pointers to other text as well. Hypermedia, on the other hand, is a superset of hypertext. Documents having hypermedia, not only contain links to other pieces of text and information, but also to numerous other forms of media, ranging from images to sound.

7. Neural Networking And Parallel Programming

It addresses the problem of prediction and approximation of complex time-varying systems. Firstly, the entire time-varying process is split into several time intervals or slots. Then, neural networks are developed in a particular time interval to disperse the load of various networks. OOP simplifies the entire process by simplifying the approximation and prediction ability of networks.

8. Office Automation Systems

These include formal as well as informal electronic systems primarily concerned with information sharing and communication to and from people inside as well as outside the organization. Some examples are Email, Word processing, web calendars, Desktop publishing

9. CIM/CAD/CAM Systems

OOP can also be used in manufacturing and design applications as it allows people to reduce the effort involved. For instance, it can be used while designing blueprints, flowcharts, etc. OOP makes it possible for the designers and engineers to produce these flowcharts and blueprints accurately.

10. AI Expert Systems

These are computer applications which are developed to solve complex problems pertaining to a specific domain, which is at a level far beyond the reach of a human brain. It has the following **characteristics**: Reliable, Highly responsive, Understandable, High-performance

1.1.8

Java Buzzwords or Features of Java

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as *java buzzwords*.

A list of most important features of Java language is given below.

1. Simple
2. Object-Oriented
3. Platform independent
4. Secured
5. Robust
6. Architecture neutral
7. Portable
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed

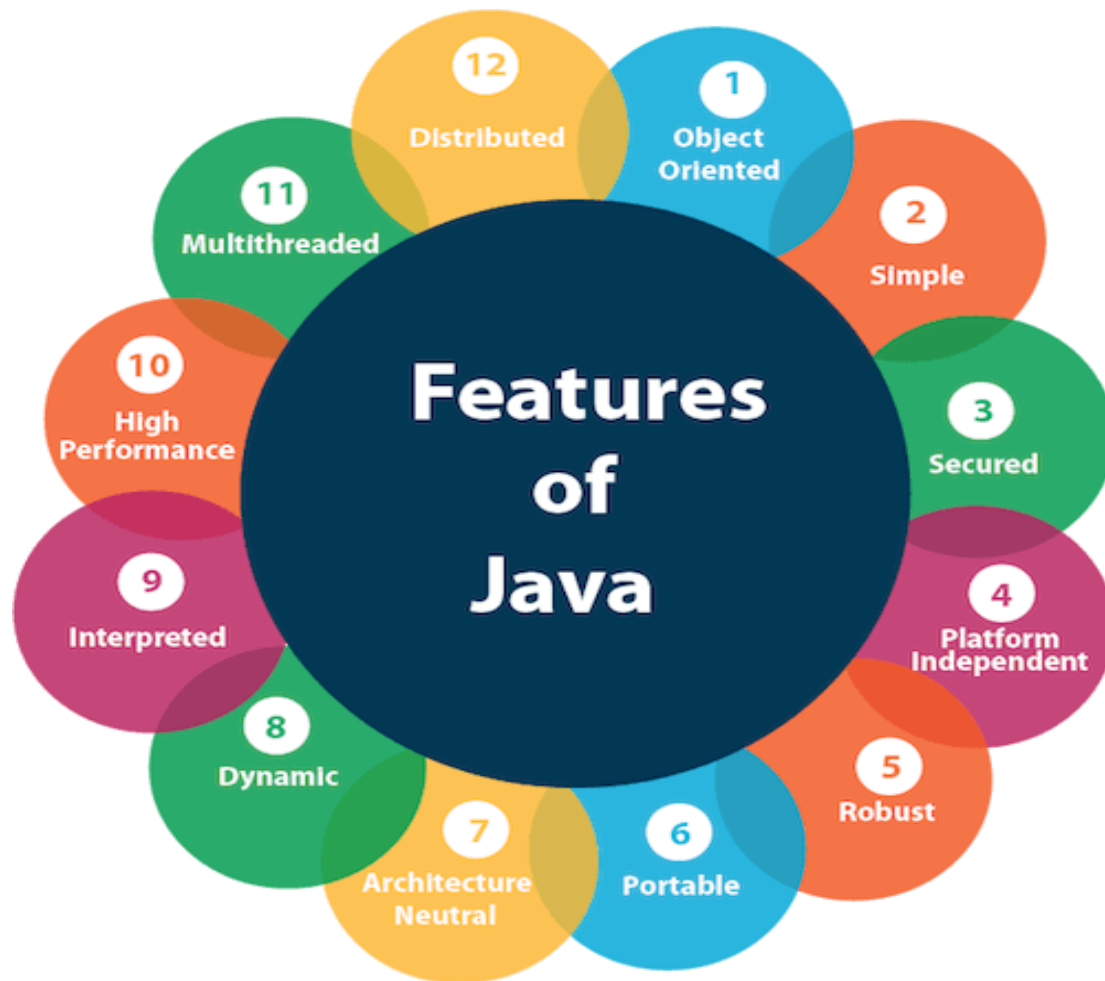


Figure 1.6: Features of Java

Simple:

Java language is simple because:

- 1) Syntax is based on C++ (so easier for programmers to learn it after C++).
- 2) Removed many confusing and/or rarely-used features
Ex.; explicit pointers, operator overloading etc.
- 3) No need to remove unreferenced objects ,to do this Automatic Garbage Collection in java.

Object-oriented :

Object-Oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior. Object-oriented programming (OOPs) is a methodology that simplify software development and maintenance by providing some principles. They are :

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Platform Independence :

Java provides software-based platform. The Java platform differs from most other platforms in the sense that it's a software-based platform that runs on top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

Java code can be run on multiple platforms e.g.Windows, Linux, SunSolaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. *This bytecode is a platform independent code because it can be run on multiple platforms i.e. **Write Once and Run Anywhere(WORA)**.*

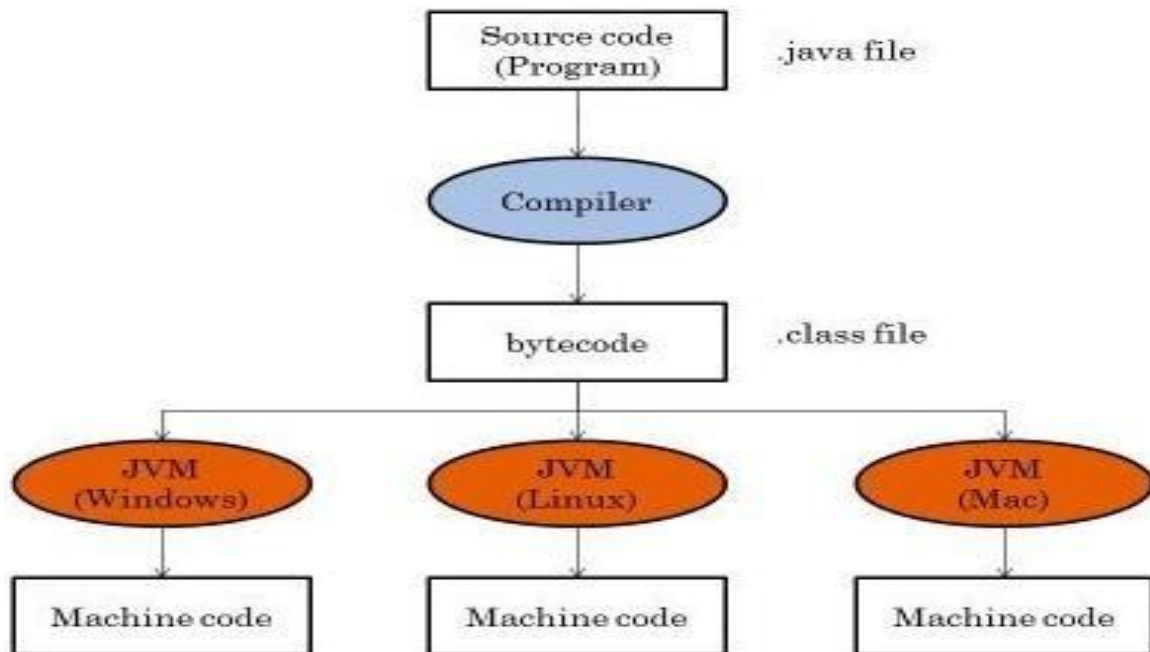


Figure 1.7: platform independency

Secured:

Java is secured because

1. There is no explicit pointers.
2. Programs run inside virtual machine sandbox.

Robust :

Robust simply means strong. Java uses strong memory management. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust.

Architecture-neutral:

There are no implementation dependent features Ex.;size of primitive types is set

Portable :

We may carry the java bytecode to any platform.

High-performance:

Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

Distributed :

We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

Multi-threaded:

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-

threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.

1.1.9 An Overview of Java

Java is a **Programming Language** and a **Platform**.

Platform : Any hardware or software environment in which a program runs, known as a platform. Since Java has its own Runtime Environment (JRE) and API, it is called platform.

History of Java

Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc. It was best suited for internet programming. Later, Java technology as incorporated by Netscape.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

Firstly, it was called "**Greentalk**" by James Gosling and file extension as .gt. After that, it was called **Oak** and was developed as a part of the Green project.

Oak is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania etc. In 1995, Oak was renamed as "**Java**" because it was already a trademark by Oak Technologies. Java is an island of Indonesia where first coffee was produced (called java coffee).

Note: Java is just a name not an acronym.

Java Version History

There are many java versions that has been released.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)
10. Java SE 8 (18th March, 2014)

Applications using Java: There are mainly 4 type of applications that can be created using java

a. Standalone Application:

It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.

b. Web Application:

An application that runs on the server side and creates dynamic page, is called web application. Currently, Servlet, JSP, struts, JSF etc. technologies are used for creating web applications in java.

c. Enterprise Application:

An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

d. Mobile Application:

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

JDK, JRE and JVM

Java Development Kit(JDK)

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools.

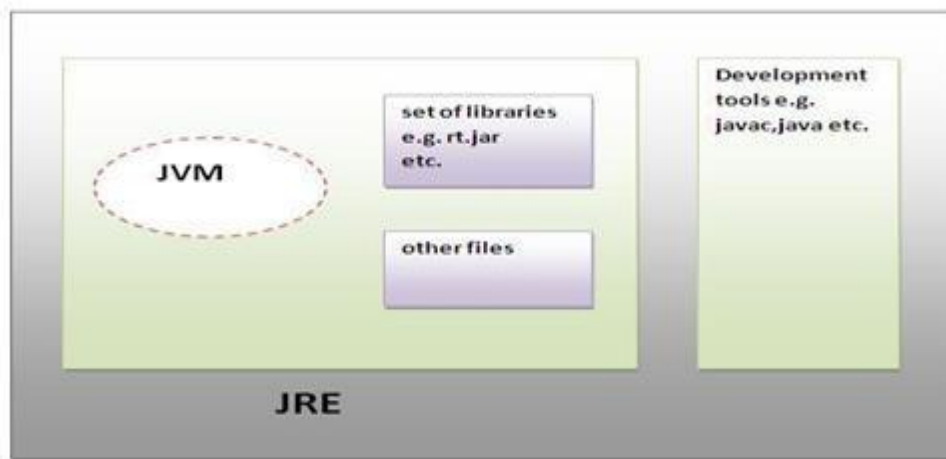


Figure 1.8: Java Development Kit (JDK)

Java Runtime Environment (JRE) :

JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime. Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



Figure 1.9: Java Runtime Environment (JRE)

Java Virtual Machine (JVM)

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a java code. JVM is a part of JRE(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

JVM is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed . JVMs are available for many hardware and software platforms.

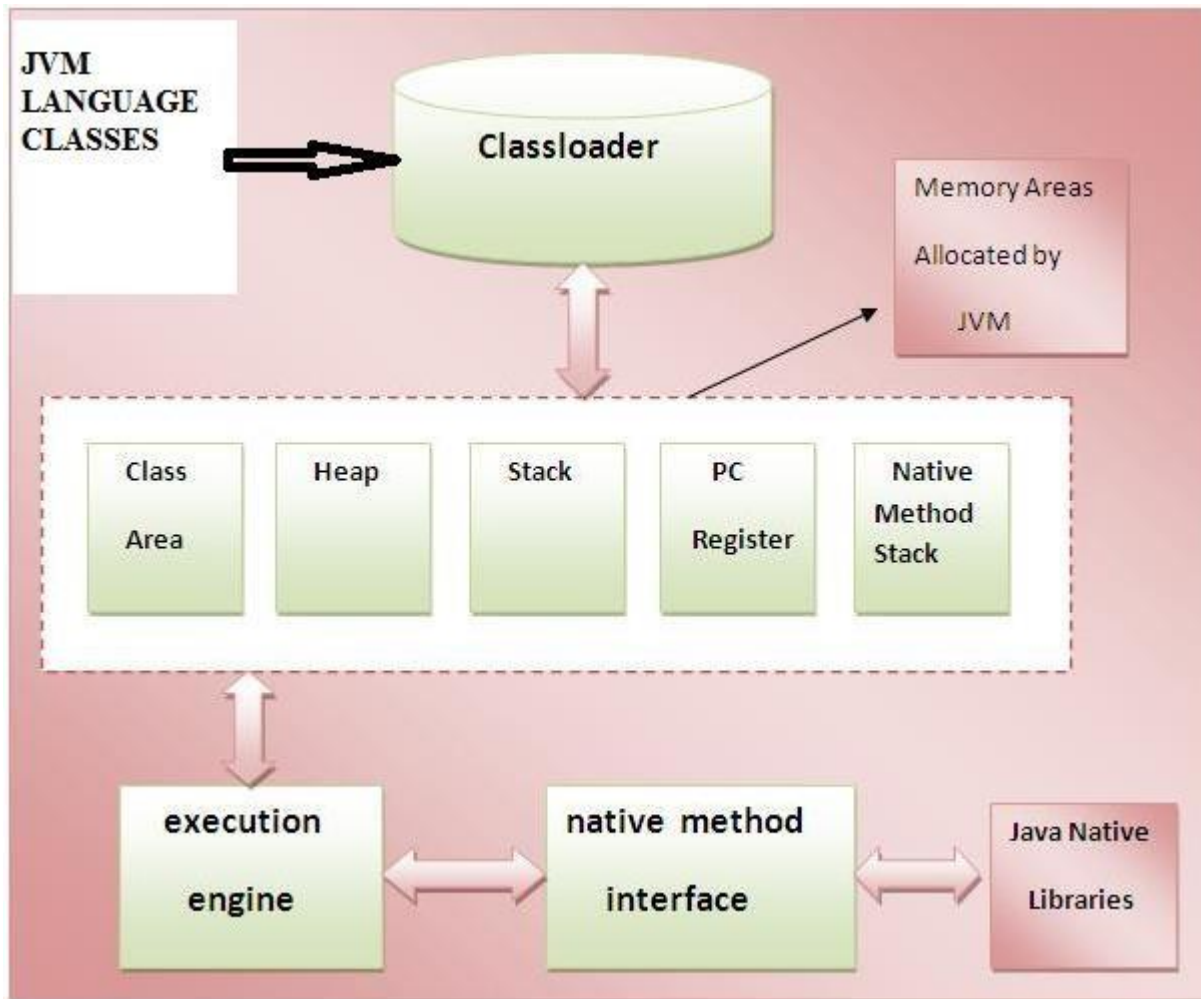
JVM is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, and instance of JVM is created.

NOTE:JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent

Internal Architecture of JVM

When we compile a *.java* file, *.class* files(contains byte-code) with the same class names present in *.java* file are generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.



Class Loader Subsystem

It is mainly responsible for three activities.

- Loading
- Linking
- Initialization

Loading : The Class loader reads the .class file, generate the corresponding binary data and save it in method area.

Linking : Performs verification, preparation, and (optionally) resolution.

- Verification : It ensures the correctness of .class file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception `java.lang.VerifyError`.
- Preparation : JVM allocates memory for class variables and initializing the memory to default values.
- Resolution : It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

Initialization : In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in class hierarchy.

Method area /Class area:In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

Heap area :Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.

Stack area :For every thread, JVM create one *run-time stack* which is stored here. Every block of this stack is called “activation record/stack frame” which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it’s run-time stack will be destroyed by JVM. It is not a shared resource.

PC Registers :Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.

Native method stacks :For every thread, separate native stack is created. It stores native method information.

Writing and Execution of simple Program

1. Open a simple text editor program such as Notepad and type the following content:

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

2. Save the file as HelloWorld.java (note that the extension is .java) under a directory, let's say, C:\Java

3. Compile your first Java program

Now let's compile our first program in the HelloWorld.java file using **javac tool**.

Syntax: javac programname.java

Type the following command to change the current directory to the one where the source file is stored:

```
C:\>cd C:\Java
C:\Java>javac HelloWorld.java
C:\Java>_
```

4. Run your first Java program

It's now ready to run our first Java program. To run java program use following Syntax

Syntax: java programname

That invokes the Java Virtual Machine to run the program called HelloWorld (note that there is no .java or .class extension). You would see the following output:

```
C:\Java>java HelloWorld
Hello world!
C:\Java>
```

Structure of the Java Program

Structure of a java program is the standard format released by Language developer to the Industry programmer.

Sun Micro System has prescribed the following structure for the java programmers for developing java application.

Structure of Java Program	Example
package details class Classname { Datamembers user defined methods; public static void main(string args[]) { block of statements; } }	import java.lang.*; class Demo { int x=10; void display(){ System.out.println("java program structure"); } public static void main(string args[]) { Demo obj=new Demo(); System.out.println("x value is"+obj.x); Obj.display(); } }

A **package** is a collection of classes, interfaces and sub-packages. A sub package contains collection of classes, interfaces and sub-sub packages etc. `java.lang.*`; package is imported by default and this package is known as default package.

Class is keyword used for developing user defined data type and every java program must start with a concept of class.

"**ClassName**" represent a java valid variable name treated as a name of the class each and every class name in java is treated as user-defined data type.

Data member represents either instance or static they will be selected based on the name of the class.

User-defined methods represents either instance or static they are meant for performing the operations either once or each and every time.

Each and every java program starts execution from the `main()` method. And hence `main()` method is known as program driver.

Since `main()` method of java is not returning any value and hence its return type must be void.

Since `main()` method of java executes only once throughout the java program execution and hence its nature must be static.

Since `main()` method must be accessed by every java programmer and hence whose access specifier must be public.

Each and every `main()` method of java must take array of objects of String.

Block of statements represents set of executable statements which are in term calling user-defined methods are containing business-logic.

The file naming convention in the java programming is that which-ever class is containing `main()` method, that class name must be given as a file name with an extension `.java`.

Lexical Issues In Java

A source code of a Java program consists of tokens. Tokens are atomic code elements. Java programs are a collection of Tokens such as white space identifiers, comments, literals, operators, separators, and keywords.

Whitespace:

Java is a free-form language. This means that you do not need to follow any special indentation rules. For example, the Example program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In java, whitespace is a space, tab, or new line.

Identifiers:

Identifiers are used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers or the underscore and dollar sign characters. They must not begin with a number, lest they be confused with a numeric literal. Again, java is case-sensitive, so VALUE is a different identifier the Value. Some examples of valid identifiers are:

AvgTemp count a4 \$test this_is_ok

Invalid variable names include:

2count high-temp Not/ok

Literals:

Using a literal representation of it creates a constant value in java. For example, here are some literals:

100 98.6 'X' "This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments:

As mentioned, there are three types of comments defined by java. You have already seen two: single-line and multiline. The third type is called a documentation comment. This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a `/**` and ends with a `*/`.

Separators

There are few symbols in java that are used as separators. The most commonly used separator in java is the semicolon `;`. Some other separators are Parentheses `()`, Braces `{ }`, Bracket `[]`, Comma `,`, Period `.`.

Java Keywords

There are 49 reserved keywords currently defined in java. These keywords cannot be used as names for a variable, class or method.

1.1.10 Data Types

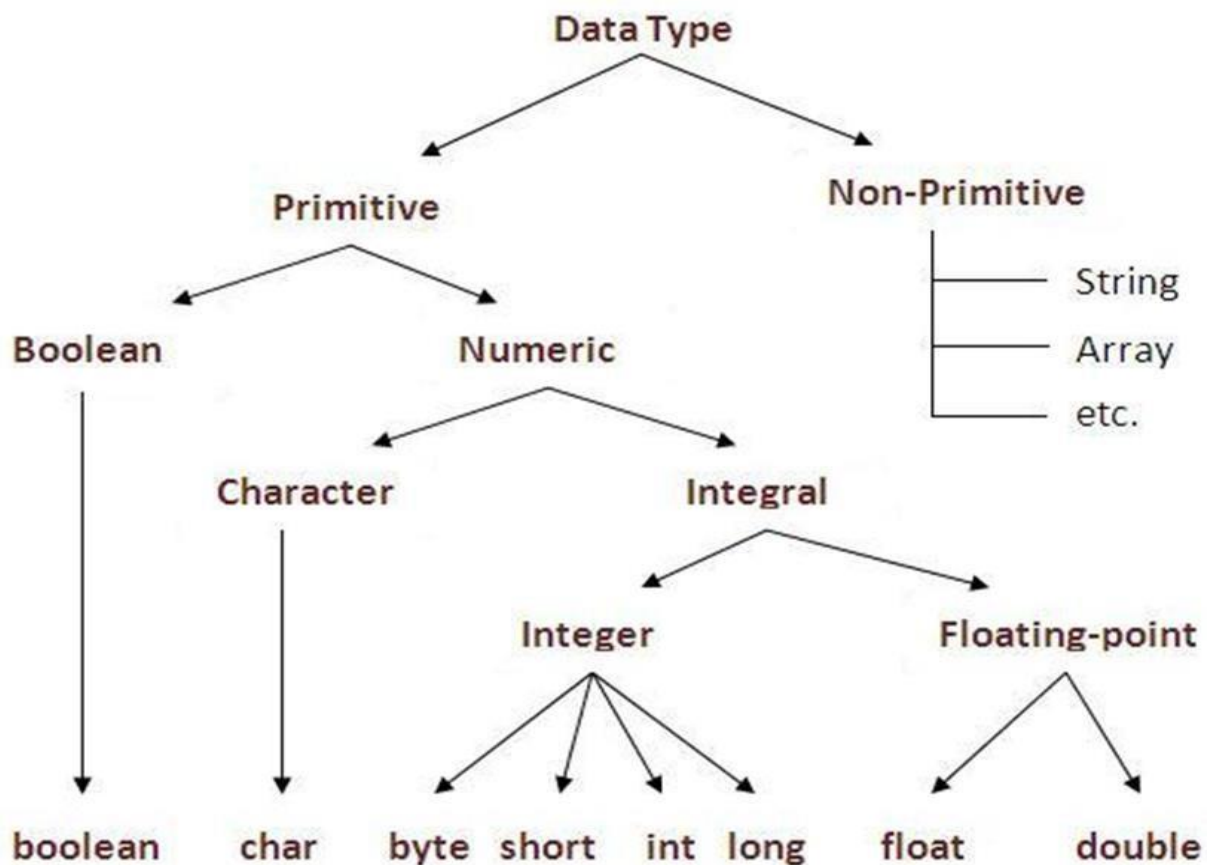
Every variable has a type, every expression has a type and all types are strictly defined. Every assignment should be checked by the compiler by the type compatibility hence java language is considered as strongly typed language. In java, There are two types of Data Types:

■ Primitive Data Types

■ Non-Primitive Data Types

1. Primitive Data Types

Data Types make a variable to store a single value at a time



The Eight Primitive data types in Java are:

The integer types:

- byte
- short
- int
- long

The floating-point types:

- float
- double

Values of class type are references. Strings are references to an instance of class String.

<i>Type</i>	<i>Size(bytes)</i>	<i>Range</i>	<i>Default</i>
byte	1	-128 .. 127	0
short	2	-32768 to 32767	0
Int	4	-2147483648 to 2147483647	0
long	8	-9223372036854775808 to 9223372036854775807	0
float	4	3.4 e38 to 1.4 e-45	0.0
double	8	1.e-308 to 4.9e-324	0.0
boolean	JVM Specific	true or false	FALSE
char	2	0 ..65535	\u0000

2. **Non-Primitive Data Types:** Derived data types are those that are made by using any other data type and can make a variable to store multiple values ,for example, arrays.
3. **User Defined Data Types:** User defined data types are those that user / programmer himself defines. For example, classes, interfaces.

Note : **int a**

Here *a* is a variable of *int* data type.

MyClass obj;

Here *obj* is a variable of data type MyClass and we call them reference variables as they can be used to store the reference to the object of that class.

In java char uses 2 byte in java because java uses unicode system rather than ASCII code system. \u0000 is the lowest range of unicode system.

Unicode System

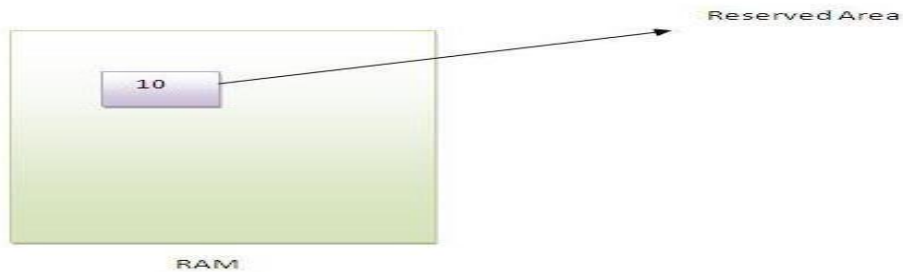
Unicode is a universal international standard character encoding that is capable of representing

most of the world's written languages.

1.1.11 Variables

Introduction of variables:

Variable is name of reserved area allocated in memory



```
int data=10; //Here data is variable
```

A variable has these attributes:

- **Name**
 1. The string of letters in a program used to designate a variable.
 2. A name should start with a letter and consist of letters and/or digits.
 3. Variables in names are case sensitive (capitalization matters).
 4. The naming convention in Java is to start a variable name with a lower case letter.
 5. New words within a name with a name start with a capital letter(example: numberOfCustomers).
- **Value**

The binary data contained in memory. The value is interpreted according to the variable's datatype.
- **Address**

The location in memory where the value is stored.
- **Size**

The number of bytes that the variable occupies in memory.

- **Datatype**
The interpretation of the value.
- **Range**
The minimum and maximum values of the variable. The range of a variable depends on its size. In general, the bigger the size in bytes, the bigger the range.

Declaration of variables:

Syntax:

`Datatype variable_name;`

Example on variable declarations:

```
int maxAge;  
int x, y, selectedIndex;  
char a, b;  
boolean flag;  
double maxVal, massInKilos;
```

Initialization of Variable:

Two types of initializations:

i.Compile time initialization

ii.Runtime initialization

i)Compile time initialization:

Syntax:

`Datatype variable_name=value;`

Example on Variable initialization in declaration:

```
int timeInSeconds = 245;
char a = 'K', b = '$';
boolean flag = true;
double maxVal = 35.875;
```

Dynamic initialization of variable:

Datatype variable_name=Expression;

Example on Dynamic initialization

```
double a =10.0,b=20.0;

double d=Math.sqrt(49)+a+b;
```

ii)Runtime Initialization

In Java, there are three different ways for reading input from the user in the command line environment(console).

- 1.Using Buffered Reader Class
- 2.Using Scanner Class
- 3.Using Console Class

1.Using Buffered Reader Class:

This is the Java classical method to take input, Introduced in JDK1.0. This method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the user in the command line.

Advantages

The input is buffered for efficient reading.

Drawback:

The wrapping code is hard to remember.

Example program on BufferedReader Class;

```
import java.io.*;
class Readdemo
```

```

{
public static void main(String args[]) throws IOException
{
int x;String str;
System.out.println("enter x value");
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
x=Integer.parseInt(br.readLine());
System.out.println("x value is"+x);
System.out.println("enter ur name");
str=br.readLine();
System.out.println(" ur name is"+str);
}
}

```

Note: To read other types, we use functions like Integer.parseInt(), Double.parseDouble(). To read multiple values, we use split().

2. Using Scanner Class

This is probably the most preferred method to take input. The main purpose of the Scanner class is to parse primitive types and strings using regular expressions, however it is also can be used to read input from the user in the command line.

Advantages:

Convenient methods for parsing primitives (nextInt(), nextFloat(), ...) from the tokenized input. Regular expressions can be used to find tokens.

Drawback:

The reading methods are not synchronized

```

class GetInputFromUser
{
    public static void main(String args[])
    {
        // Using Scanner for Getting Input from User
        Scanner in = new Scanner(System.in);
        System.out.println("enter a string");
        String s = in.nextLine();

        System.out.println("enter a integer");
        int a = in.nextInt();

        System.out.println("enter a float");
        float b = in.nextFloat();
        System.out.println("You entered string "+s);
        System.out.println("You entered integer "+a);
        System.out.println("You entered float "+b);
    }
}

```

```
}
```

Output:

```
enter a string
BVRITH
enter a integer
12
enter a float
3.4
```

```
You entered string
BVRITHYou entered
integer 12
You entered float 3.4
```

3. Using Console Class

It has been becoming a preferred way for reading user's input from the command line. In addition, it can be used for reading password-like input without echoing the characters entered by the user; the format string syntax can also be used (like `System.out.printf()`).

Advantages:

Reading password without echoing the entered characters.
Reading methods are synchronized.
Format string syntax can be used.

Drawback:

Does not work in non-interactive environment (such as in an IDE).

// Java Program to demonstrate Console Methods

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

```
//Obtaining a reference to the console.
Console con = System.console();

// Checking If there is no console available, then exit.
if(con == null)
{
    System.out.print("No console available");
    return;
}
// Using Console to input data from user
System.out.println("enter name");
String name =con.readLine();
System.out.println(name);

//to read password and then display it
System.out.println("Enter the password: ");
char[] ch=con.readPassword();

//converting char array into string
String pass = String.valueOf(ch);
System.out.println("Password is: " + pass);
}
}
```

Output:

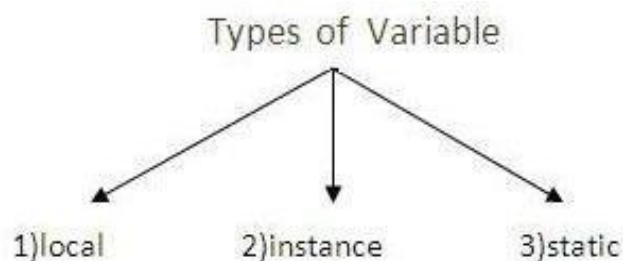
```
Enter your name: abc
Here is your name: abc
Enter the password:
Password is: xzzzz
```

Note: System.console() returns null in an online IDE

Types of Variable

There are three types of variables in java

- Local Variable
- Instance Variable
- Static Variable



Local Variable

A variable that is declared inside the method is called Local variable.

Instance Variable

A variable that is declared inside the class but outside the method is called instance variable .

It is not declared as static.

Static variable

A Variable that is declared as static is called static variable. It cannot be local.

Example to understand the types of variables

```
class A
{
    int data=50;//instance variable
    static int m=100;//static variable
    void method()
    {
        int n=90; //local variable
    }
} //end of class
```

Scope and life time of a variable

The scope determines what objects are visible to other parts of program and also determine Lifetime of the those objects.

In java,scope is defined by the class and by the method.The instance variables and static variables are declared within the class and out of any method ,so this types variables are available through out the class

The local variables are declared inside the method. So these variables are available within the block in which they are declared.

Type conversion and casting

The process of converting one data type to another data type is called as **Casting**

Types of castings:

There are two types of castings

1. Explicit type conversion (or) Narrowing conversion
2. Implicit type conversion (or) Widening conversion

1. Casting Incompatible Types or explicit type conversion or narrowing conversion

Casting a larger data type into a smaller data type may result in a loss of data.

This kind of conversion is sometimes called a **narrowing conversion**, since you are explicitly making the value narrower.

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion.

It has this general form:

type variable1=(target-type) variable2;

Example:

```
double m=50.00;
```

```
int n=(int)m;
```

2. Java's Automatic Conversions or implicit type conversion or widening conversion

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- a. The two types are compatible.
- b. The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place.

Example:

```
int i=30;

double x;

x=i;
```

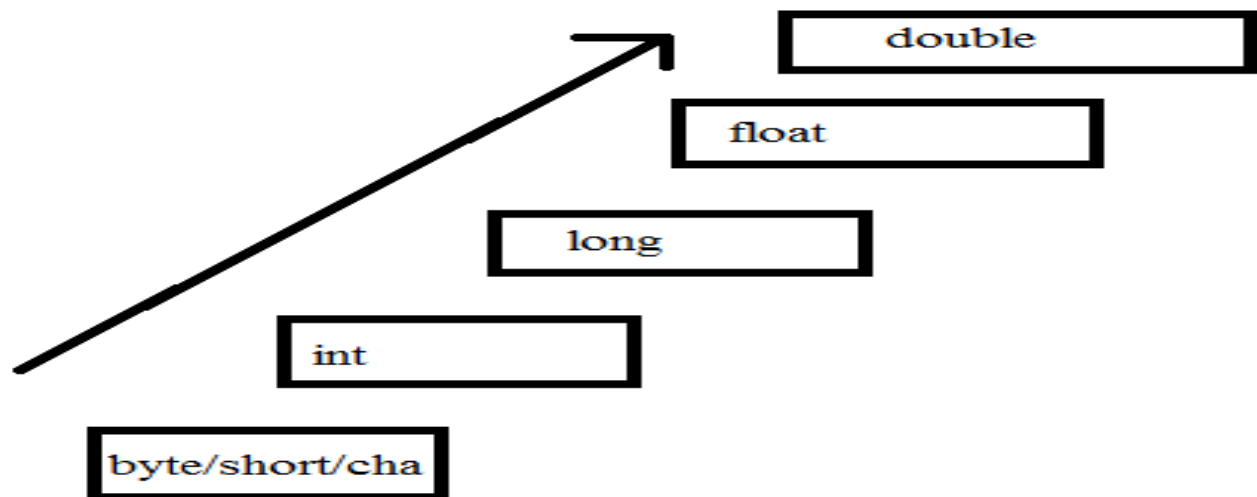
The double type is always large enough to hold all valid int values, so no explicit cast statement is required.

```
class Typecastdemo
{
    public static void main(String args[])
    {
        double f=3.141,x;
        int i,j=30;
        i=(int)f;//explicit conversion
        x=j;//implicit conversion
        System.out.println("i value is"+i);
        System.out.println("x value    is"+x);
    }
}
```

Output: i value is 3

x value is 30.00

Type promotion rules



The sequence of promotion rules that are applied while implicit type conversion is as follows: All byte, short and char are automatically converted to int; then

1. if one of the operands is long, the other will be converted to long and the result will be long
2. else, if one of the operands is float, the other will be converted to float and the result will be float
3. else, if one of the operands is double, the other will be converted to double and the result will be

.The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However the following changes are introduced during the final assignment.

Float to int causes truncation of the fractional part. Double to float causes rounding of digits.

Long int to int causes dropping of the excess higher order bits.

1.1.12 Arrays

Introduction

Definition of Array

Normally, array is a collection of similar type of elements that have contiguous memory location.

- ✓ **Java array** is an object the contains elements of similar data type.
- ✓ It is a data structure where we store similar elements.
- ✓ We can store only fixed set of elements in a java array.
- ✓ Array in java is index based, first element of the array is stored at 0 index.
- ✓ An array is an indexed collection of fixed number of homogeneous data elements.

Types of Array in java

There are two types of array.

1. Single Dimensional Array
2. Multidimensional Array.

Single Dimensional Array:

Single dimensional array declaration:

Syntax:

```
Datatype arrayname[ ];  
Or  
Datatype[ ] arrayname;
```

Example :

```
int a[ ];
```

```
int[ ] a; //recommended to use because name is clearly separated from the type int
[ ]a;
```

At the time of declaration we can't specify the size otherwise we will get compile time error.

Example :

```
int[ ] a;//valid
```

```
int[5] a;//invalid
```

Creation of Array construction:

Every array in java is an object hence we can create by using **new** operator.

Syntax for Creating:

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

Example :

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

we can also combine declaration and creation into one step

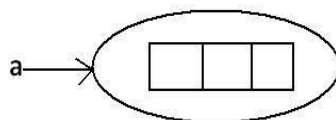
Syntax

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

Example:

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

Diagram:



For every array type corresponding classes are available but these classes are part of java language and not available to the programmer level.

Rules for creating an Array

Rule 1 :

- ❖ At the time of array creation compulsory we should specify the size otherwise we will get compile time error.

Example :

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

```
int[] a=new int[];//C.E:array dimension missing
```

Rule 2:

- ❖ It is legal to have an array with size zero in java.

Example :

```
int[] a=new int[0];
```

```
System.out.println(a.length);//0
```

Rule 3 :

- ❖ If we are taking array size with -ve int value then we will get runtime exception saying NegativeArraySizeException.

Example :

```
int[] a=new int[-3];//R.E:NegativeArraySizeException
```

Rule 4:

- ❖ The allowed data types to specify array size are byte, short, char, int. By mistake if we are using any other type we will get compile time error.

Example :

```
int[] a=new int['a'];//(valid)
```

```
byte b=10;
```

```
int[] a=new int[b];//(valid)
```

```
short s=20;
```

```
int[] a=new int[s];//(valid)
```

```
int[] a=new int[10l];//C.E:possible loss of precision//(invalid)
```

```
int[] a=new int[10.5];//C.E:possible loss of precision//(invalid)
```

Rule 5 :

❖ The maximum allowed array size in java is maximum value of int size [2147483647].

Example :

```
int[] a1=new int[2147483647];(valid)
```

```
int[] a2=new int[2147483648];//C.E:integer number too large: 2147483648(invalid)
```

Array initialization:

Syntax for initialization of Single Dimensional Array:

```
arrayname[subscript]=value;
```

Example:

```
int[] a=new int[4];
```

```
a[0]=10;
```

```
a[1]=20;
```

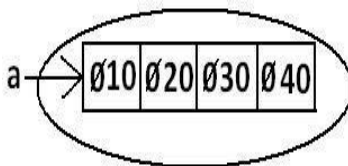
```
a[2]=30;
```

```
a[3]=40;
```

```
a[4]=50;//R.E:ArrayIndexOutOfBoundsException: 4
```

```
a[-4]=60;//R.E:ArrayIndexOutOfBoundsException: -4
```

Diagram:

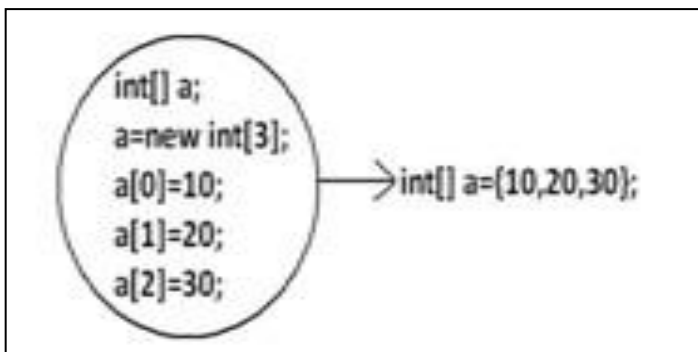


Note:if we are trying to access array element with out of range index we will get RuntimeException saying `ArrayIndexOutOfBoundsException`

Declaration construction and initialization of an array in a single line:

- We can perform declaration construction and initialization of an array in a single line.

Example:



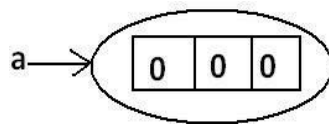
```
int [] a={ 10,20,30};//valid
```

```
String[] s={"abc","def","jog","lmn"};(valid)
```


Whenever we are creating an array every element is initialized with default value automatically.

Example 1:

```
int[] a=new int[3];  
System.out.println(a); //  
System.out.println(a[0]);
```

Diagram :**Array Length:***length Vs length():***length:**

- It is the final variable applicable only for arrays.
- It represents the size of the array.

length() method:

- It is a final method applicable for String objects.
- It returns the no of characters present in the String.

Example:

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

```
System.out.println(x.length);//3
```

```
System.out.println(x.length());//C.E: cannot find symbol
```

Example:

```
String s="bhaskar";
```

```
System.out.println(s.length);//C.E:cannot find symbol
```

```
System.out.println(s.length());//7
```

Example on Single Dimensional Array :

```
class Testarray1
```

```
{
```

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

```
        int a[]={ 33,3,4,5 };//declaration, instantiation and initialization
```

```
        //printing array
```

```
        for(int i=0;i<a.length;i++)//length is the property of array
```

```
            System.out.println(a[i]);
```

```
    }
```

```
}
```

Output :

```
33
```

```
3
```

```
4
```

```
5
```

Array and Functions (or) Passing single dimensional array to a function

Passing Array to method in java. We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get minimum number of an array using method.

```

class Testarray2 {
    static void min(int arr[]) {
        int min=arr[0];
        for(int i=1;i<arr.length;i++)
            if(min>arr[i])
                min=arr[i];
        System.out.println(min);
    }

    public static void main(String args[])
    {
        int a[]={ 33,3,4,5 };
        min(a);    //passing array to method
    }
}

```

Output: 3

2.Multidimensional Array

Two dimensional array

- ❖ In java multidimensional arrays are implemented as array of arrays approach but not matrix form.
- ❖ The main advantage of this approach is to improve memory utilization.

Two dimensional array declaration:

```
Datatype arrayname[ ][ ];
Or
Datatype[ ][ ] arrayname;
```

Alternate two dimensional array Declaration Syntax

```
int [ ][ ]a;
```

```
int [ ][ ]a;
```

```
int a[ ][ ];
```

All are valid statements

```
int [ ] [ ]a;
```

```
int [ ] a[ ];
```

```
int [ ]a[ ];
```

Example :

```
int[] []a,[]b;
  |    |
  |    |invalid
  |    |valid
```

Creation of Two- Dimensional array:

syntax:

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

Example:

```
int a[ ][ ];
```

```
a=new int[3][2];
```

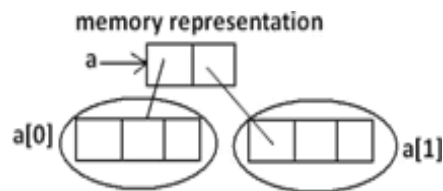
we can also combine declaration and creation into one step

Syntax

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

Example1:

```
int[][] a=new int[2][3]; base size
```



Variable Size Array:

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

```
a[0]=new int[2];
```

```
a[1]=new int[4];
```

```
a[2]=new int[3];
```

Two- Dimensional array Array initialization :

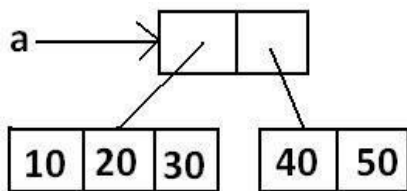
Syntax for initialization of Two- Dimensional Array:

```
datatype [subscript][subscript] arrayname ={list of value};
```

Example:

```
int[][] a={{ 10,20,30},{40,50}};
```

Diagram:



If we want to use this short cut compulsory we should perform declaration, construction and initialization in a single line. If we are trying to divide into multiple lines then we will get compile time error.

Two Dimensional array length

Length variable applicable only for arrays where as length() method is applicable for String objects.

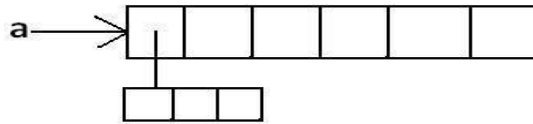
Example :

```
int[][] a=new int[6][3];
```

```
System.out.println(a.length);//6
```

```
System.out.println(a[0].length);//3
```

Diagram:



Examples on Two Dimensional array

Example 1: program to declare,create,initialize and access the elements of 2-D array

class array2d

```
{

    public static void main(String args[]) {

        //declaring and initializing 2D array

        int arr[][]={{ 1,2,3},{2,4,5},{4,4,5}};

        //printing 2D array

        for(int i=0;i<3;i++) {

            for(int j=0;j<3;j++) {

                System.out.print(arr[i][j]+" ");

            }

            System.out.println();

        }

    }

}
```

Output:1 2 3

2 4 5

4 4 5

Example 2 : program for Addition of two matrix

```
class twodarrayaddition
{
    public static void main(String args[]) {
        //creating two matrices
        int a[][]={{ 1,3,4},{3,4,5}};
        int b[][]={{ 1,3,4},{3,4,5}};
        //creating another matrix to store the sum of two matrices
        int c[][]=new int[2][3];
        //adding and printing addition of 2 matrices
        for(int i=0;i<2;i++){
            for(int j=0;j<3;j++){
                c[i][j]=a[i][j]+b[i][j];
                System.out.print(c[i][j]+" ");
            }
            System.out.println();//new line
        }
    }
}
```

Output : 2 6 8
6 8 10

Alternate Three dimensional array declaration Syntax:

<code>int[][][] a;</code>	}	All are valid statements
<code>int [][][]a;</code>		
<code>int a[][][];</code>		
<code>int [] a[][];</code>		
<code>int [][]a[];</code>		

Anonymous Arrays:

- ✚ Sometimes we can create an array without name such type of nameless arrays are called **anonymous arrays**.
- ✚ The main objective of anonymous arrays is “**just for instant use**”.
- ✚ We can create anonymous array as follows.

```
new int[] { 10,20,30,40 }; (valid)
```

```
new int[][] { { 10,20 }, { 30,40 } }; (valid)
```

At the time of anonymous array creation we can't specify the size otherwise we will get compile time error.

Example :

```
new int[3] { 10,20,30,40 }; //C.E: ';' expected (invalid)
```

Advantage of Java Array

- **Code Optimization** : It makes the code optimized, we can retrieve or sort the data easily.
- **Random access** : We can get any data located at any index position.
- **Readability** : we can represent multiple values with the same name so that readability of the code will be improved

Disadvantage of Java Array

- Fixed in size that is once we created an array there is no chance of increasing or decreasing the size based on our requirement that is to use arrays concept compulsory we should know the size in advance which may not possible always.
- We can resolve this problem by using collections.

1.1.13 Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups –

- ❖ Arithmetic Operators
- ❖ Relational Operators
- ❖ Bitwise Operators
- ❖ Logical Operators
- ❖ Assignment Operators
- ❖ Misc Operators

The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators –

Assume integer variable A holds 10 and variable B holds 20, then –

Show Examples

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

The Relational Operators

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then –Show Examples

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.

!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators

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

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

```

a = 0011 1100
b = 0000 1101
-----
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

```

The following table lists the bitwise operators –

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

Show Examples

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
(bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators

The following table lists the logical operators –

Assume Boolean variables A holds true and variable B holds false, then –

Show Examples

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
(logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

The Assignment Operators

Following are the assignment operators supported by Java language –

Show Examples

=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A

-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	$C -= A$ is equivalent to $C = C - A$
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	$C *= A$ is equivalent to $C = C * A$
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	$C /= A$ is equivalent to $C = C / A$
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	$C \% = A$ is equivalent to $C = C \% A$
<<=	Left shift AND assignment operator.	$C <<= 2$ is same as $C = C << 2$
>>=	Right shift AND assignment operator.	$C >>= 2$ is same as $C = C >> 2$
&=	Bitwise AND assignment operator.	$C \&= 2$ is same as $C = C \& 2$
^=	bitwise exclusive OR and assignment operator.	$C \wedge= 2$ is same as $C = C \wedge 2$
=	bitwise inclusive OR and assignment operator.	$C = 2$ is same as $C = C 2$

Miscellaneous Operators

Miscellaneous operators includes

- *Conditional Operator (? :)*
- *instanceof Operator*

There are few other operators supported by Java Language.

Conditional Operator (? :)

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as –

```
variable x = (expression) ? value if true : value if false
```

Following is an example –

Example

```
public class Test {  
    public static void main(String args[]) {  
        int a, b;  
        a = 10;  
        b = (a == 1) ? 20 : 30;  
        System.out.println("Value of b is : " + b);  
        b = (a == 10) ? 20 : 30;  
        System.out.println("Value of b is : " + b);  
    }  
}
```

This will produce the following result –

Output

```
Value of b is : 30  
Value of b is : 20
```

instanceof Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as –

```
( Object reference variable ) instanceof (class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example –

Example

```
public class Test {
    public static void main(String args[]) {
        String name = "James";
        // following will return true since name is type of String
        boolean result = name instanceof String;
        System.out.println( result );
    }
}
```

This will produce the following result –

Output

```
true
```

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example –

Example

```
class Vehicle { }

public class Car extends Vehicle {
    public static void main(String args[]) {
        Vehicle a = new Car();
    }
}
```

```

        boolean result = a instanceof Car;
        System.out.println( result );
    }
}

```

This will produce the following result –

Output

```
true
```

Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator –

For example, $x = 7 + 3 * 2$; here x is assigned 13, not 20 because operator $*$ has higher precedence than $+$, so it first gets multiplied with $3 * 2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	$()$ $[]$ $.$ (dot operator)	Left to right
Unary	$++$ $--$ $!$ \sim	Right to left

Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left

Assignment, Arithmetic, and Unary Operators

The Simple Assignment Operator

One of the most common operators that you'll encounter is the simple assignment operator "=".

```
int cadence = 0;
```

```
int speed = 0;
```

```
int gear = 1;
```

The Arithmetic Operators

```
class ArithmeticDemo {  
    public static void main (String[] args)    {  
        int result = 1 + 2;  
        // result is now 3  
        System.out.println("1 + 2 = " + result);  
        int original_result = result;  
        result = result - 1;  
        // result is now 2  
        System.out.println(original_result + " - 1 = " + result);  
        original_result = result;  
        result = result * 2;  
        // result is now 4  
        System.out.println(original_result + " * 2 = " + result);  
    }  
}
```

```
        original_result = result;

        result = result / 2;

        // result is now 2

        System.out.println(original_result + " / 2 = " + result);

        original_result = result;

        result = result + 8;

        // result is now 10

        System.out.println(original_result + " + 8 = " + result);

        original_result = result;

        result = result % 7;

        // result is now 3

        System.out.println(original_result + " % 7 = " + result);

    }

}
```

This program prints the following:

$$1 + 2 = 3$$

$$3 - 1 = 2$$

$$2 * 2 = 4$$

$$4 / 2 = 2$$

$$2 + 8 = 10$$

$$10 \% 7 = 3$$

You can also combine the arithmetic operators with the simple assignment operator to create *compound assignments*. For example, `x+=1;` and `x=x+1;` both increment the value of `x` by 1.

The `+` operator can also be used for concatenating (joining) two strings together, as shown in the following `ConcatDemo` program:

```
classConcatDemo
{
    public static void main(String[] args)
    {
        String firstString = "This is";
        String secondString = " a concatenated string.";
        String thirdString = firstString+secondString;
        System.out.println(thirdString);
    }
}
```

The Unary Operators

The unary operators require only one operand; they perform various operations such as incrementing / decrementing a value by one, negating an expression, or inverting the value of a boolean.

Operator	Description
+	Unary plus operator; indicates positive value (numbers are positive without this, however)
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

The following program, `UnaryDemo`, tests the unary operators:

```
ClassUnaryDemo {  
  
    public static void main(String[] args) {  
  
        int result = +1;  
  
        // result is now 1  
  
        System.out.println(result);  
  
        result--;  
  
        // result is now 0  
  
        System.out.println(result);  
  
        result++;  
  
        // result is now 1  
  
        System.out.println(result);  
  
        result = -result;  
  
        // result is now -1  
  
        System.out.println(result);  
  
        boolean success = false;  
  
        // false  
  
        System.out.println(success);  
  
        // true  
  
        System.out.println(!success);  
  
    }  
}
```


The increment/decrement operators can be applied before (prefix) or after (postfix) the operand. The code `result++`; and `++result`; will both end in `result` being incremented by one. The only difference is that the prefix version (`++result`) evaluates to the incremented value, whereas the postfix version (`result++`) evaluates to the original value. If you are just performing a simple increment/decrement, it doesn't really matter which version you choose. But if you use this operator in part of a larger expression, the one that you choose may make a significant difference.

The following program, `PrePostDemo`, illustrates the prefix/postfix unary increment operator:

```
classPrePostDemo {  
  
    public static void main(String[] args) {  
  
        int i = 3;  
  
        i++;  
  
        // prints 4  
  
        System.out.println(i);  
  
        ++i;  
  
        // prints 5  
  
        System.out.println(i);  
  
        // prints 6  
  
        System.out.println(++i);  
  
        // prints 6  
  
        System.out.println(i++);  
  
        // prints 7  
  
        System.out.println(i);  
  
    }  
}
```

```
}
```

Equality, Relational, and Conditional Operators

The Equality and Relational Operators

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use "==" , not "=", when testing if two primitive values are equal.

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

```
classComparisonDemo {
    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        if(value1 == value2)
            System.out.println("value1 == value2");
        if(value1 != value2)
            System.out.println("value1 != value2");
        if(value1 > value2)
            System.out.println("value1 > value2");
        if(value1 < value2)
            System.out.println("value1 < value2");
        if(value1 <= value2)
            System.out.println("value1 <= value2");
    }
}
```

The Conditional Operators

The `&&` and `||` operators perform *Conditional-AND* and *Conditional-OR* operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

Operator	Description
<code>&&</code>	Conditional-AND
<code> </code>	Conditional-OR

```
class ConditionalDemo1 {  
  
    public static void main(String[] args){  
  
        int value1 = 1;  
  
        int value2 = 2;  
  
        if((value1 == 1) && (value2 == 2))  
  
            System.out.println("value1 is 1 AND value2 is 2");  
  
        if((value1 == 1) || (value2 == 1))  
  
            System.out.println("value1 is 1 OR value2 is 1");  
  
    }  
  
}
```

The following program, `ConditionalDemo2`, tests the `?:` operator:

```
class ConditionalDemo2 {  
  
    public static void main(String[] args){  
  
        int value1 = 1;  
  
        int value2 = 2;  
  
        int result;  
  
        booleansomeCondition = true;  
  
        result = someCondition ? value1 : value2;  
  
        System.out.println(result);  
  
    }  
  
}
```

Because `someCondition` is true, this program prints "1" to the screen. Use the `?:` operator instead of an `if-then-else` statement if it makes your code more readable; for example, when the expressions are compact and without side-effects (such as assignments).

Bitwise and Bit Shift Operators

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used. Therefore, their coverage is brief; the intent is to simply make you aware that these operators exist.

The unary bitwise complement operator `~` inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0". For example, a `byte` contains 8 bits; applying this operator to a value whose bit pattern is "00000000" would change its pattern to "11111111".

The signed left shift operator `<<` shifts a bit pattern to the left, and the signed right shift operator `>>` shifts a bit pattern to the right. The bit pattern is given by the left-hand operand, and the number

of positions to shift by the right-hand operand. The unsigned right shift operator ">>>" shifts a zero into the leftmost position, while the leftmost position after ">>" depends on sign extension.

The bitwise `&` operator performs a bitwise AND operation.

The bitwise `^` operator performs a bitwise exclusive OR operation.

The bitwise `|` operator performs a bitwise inclusive OR operation.

The following program, `BitDemo`, uses the bitwise AND operator to print the number "2" to standard output.

```
class BitDemo {  
  
    public static void main(String[] args) {  
  
        int bitmask = 0x000F;  
  
        int val = 0x2222;  
  
        // prints "2"  
  
        System.out.println(val & bitmask);  
  
    }  
  
}
```

1.1.14 Expressions

Expressions, Statements, and Blocks

Definition: An *expression* is a series of variables, operators, and method calls (constructed according to the syntax of the language) that evaluates to a single value. As discussed in the previous section, operators return a value, so the use of an operator is an expression. This partial listing of the MaxVariablesDemo program shows some of the program's expressions in boldface:

```
...

// other primitive types

char aChar = 'S';

boolean aBoolean = true;

// display them all

System.out.println("The largest byte value is " + largestByte);

...

if (Character.isUpperCase(aChar)) {

    ... }
```

Each expression performs an operation and returns a value, as shown in the following table.

Expression	Action	Value Returned
aChar = 'S'	Assign the character 'S' to the character variable aChar	The value of aChar after the assignment ('S')
"The largest byte value is " + largestByte	Concatenate the string "The largest byte value is " and the value of largestByte converted to a string	The resulting string: The largest byte value is 127
Character.isUpperCase(aChar)	Call the method isUpperCase	The return value of the method: true

The data type of the value returned by an expression depends on the elements used in the expression. The expression `aChar = 'S'` returns a character because the assignment operator returns a value of the same data type as its operands and `aChar` and `'S'` are characters. As you see from the other expressions, an expression can return a boolean value, a string, and so on.

The Java programming language allows you to construct compound expressions and statements from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other. Here's an example of a compound expression:

$$x * y * z$$

. For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

$$x + y / 100 \quad // \text{ambiguous}$$

You can specify exactly how you want an expression to be evaluated, using balanced parentheses—(and). For example, to make the previous expression unambiguous, you could write:

$$(x + y) / 100 \quad // \text{unambiguous, recommended}$$

If you don't explicitly indicate the order in which you want the operations in a compound expression to be performed, the order is determined by the *precedence* assigned to the operators in use within the expression. Operators with a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Thus, the two following statements are equivalent:

$$x + y / 100$$

$$x + (y / 100) \quad // \text{unambiguous, recommended}$$

When writing compound expressions, you should be explicit and indicate with parentheses which operators should be evaluated first. This practice will make your code easier to read and to maintain.

1.1.15. Control Statements

A control statement works as a determiner for deciding the next task of the other statements whether to execute or not. An 'If' statement decides whether to execute a statement or which statement has to execute first between the two. In Java, the control statements are divided into three categories which are selection statements, iteration statements, and jump statements. A program can execute from top to bottom but if we use a control statement. We can set order for executing a program based on values and logic, see below table .

» Decision Making in Java

- Simple if Statement
- if...else Statement
- Nested if statement
- if...else if...else statement
- Switch statement

» Looping Statements in Java

- While
- Do...while
- For
- For-Each Loop

» Branching Statements in Java

- Break
- Continue
- Return

Decision Making in Java

Decision making statements are statements which decides what to execute and when. They are similar to decision making in real time. Control flow statements control the flow of a program's execution. Here flow of execution will be based on state of a program. We have 4 decision making statements available in Java.

Simple if Statement :

Simple if statement is the basic of decision-making statements in Java. It decides if certain amount of code should be executed based on the condition.

Syntax:

if (condition)

```
{  
    Statement 1; //if condition becomes true then this will be executed  
}
```

Statement 2; //this will be executed irrespective of condition becomes true or false

Example:

```
class ifTest {  
  
    public static void main(String args[]) {  
  
        int x = 5;  
  
        if (x > 10)  
  
            System.out.println("Inside If");  
  
            System.out.println("After if statement");  
  
        }  
  
    }  
}
```

Output:

After if statement

if...else Statement :

In if...else statement, if condition is true then statements in if block will be executed but if it comes out as false then else block will be executed.

Syntax:

```
if (condition) {  
    Statemen 1; //if condition becomes true then this will be executed  
}
```

Example:

```
class ifelseTest {  
    public static void main(String args[]) {  
        int x = 9;  
        if (x > 10)  
            System.out.println("i is greater than 10");  
        else  
            System.out.println("i is less than 10");  
        System.out.println("After if else statement");  
    }  
}
```

Output:

i is less than 10

After if else statement

Nested if statement :

Nested if statement is if inside an if block. It is same as normal if...else statement but they are written inside another if...else statement.

Syntax:

```
if (condition1)
{
    Statemen 1; //executed when condition1 is true
    if (condition2)
    {
        Statement 2; //executed when condition2 is true
    }
    else
    {
        Statement 3; //executed when condition2 is false
    }
}
```

Example:

```
class nestedifTest {

    public static void main(String args[]) {

        int x = 25;

        if (x > 10)    {

            if (x%2==0)
```

```

        System.out.println("i is greater than 10 and even number");

        else

        System.out.println("i is greater than 10 and odd number");

    }

    else {

        System.out.println("i is less than 10");

    }

    System.out.println("After nested if statement");

}

}

```

Output:

i is greater than 10 and odd number
 After nested if statement

if...else statement :

if...else if statements will be used when we need to compare the value with more than 2 conditions. They are executed from top to bottom approach. As soon as the code finds the matching condition, that block will be executed. But if no condition is matching then the last else statement will be executed.

Syntax:

```

if (condition1) {
    Statemen 1; //if condition1 becomes true then this will be executed
}
else if (condition2) {
    Statement 2; // if condition2 becomes true then this will be executed
}
....
....
else {
    Statement 3; //executed when no matching condition found
}

```

Example:

```
class ifelseifTest {  
  
    public static void main(String args[]) {  
  
        int x = 2;  
  
        if (x > 10) {  
  
            System.out.println("i is greater than 10");  
  
        }  
  
        else if (x < 10)  
  
            System.out.println("i is less than 10");  
  
        }  
  
        else {  
  
            System.out.println("i is 10");  
  
        }  
  
        System.out.println("After if else if ladder statement");  
  
    }  
  
}
```

Output:

i is less than 10

After if else if ladder statement

Switch statement :

Java switch statement compares the value and executes one of the case blocks based on the condition. It is same as if...else if ladder. Below are some points to consider while working with switch statements:

- » case value must be of the same type as expression used in switch statement
- » case value must be a constant or literal. It doesn't allow variables
- » case values should be unique. If it is duplicate, then program will give compile time error

Let us understand it through one example.

```
class switchDemo{  
  
    public static void main(String args[]) {  
  
        int i=2;  
  
        switch(i){  
  
            case 0:  
  
                System.out.println("i is 0");  
  
                break;  
  
            case 1:  
  
                System.out.println("i is 1");  
  
                break;  
  
            case 2:  
  
                System.out.println("i is 2");  
  
                break;  
  
            case 3:
```

```
        System.out.println("i is 3");

        break;

        case 4:

        System.out.println("i is 4");

        break;

        default:

        System.out.println("i is not in the list");

        break;

    }

}

}
```

Looping Statements in Java :

Looping statements are the statements which executes a block of code repeatedly until some condition meet to the criteria. Loops can be considered as repeating if statements. There are 3 types of loops available in Java.

While :

While loops are simplest kind of loop. It checks and evaluates the condition and if it is true then executes the body of loop. This is repeated until the condition becomes false. Condition in while loop must be given as a Boolean expression. If int or string is used instead, compile will give the error.

Syntax:

```
Initialization;
    while (condition)
    {
        statement1;
        increment/decrement;
    }
```

Example:

```
class whileLoopTest {  
  
    public static void main(String args[]) {  
  
        int j = 1;  
  
        while (j <= 10)    {  
  
            System.out.println(j);  
  
            j = j+2;  
  
        }  
  
    }  
  
}
```

Output:

1 3 5 7 9

Do...while :

Do...while works same as while loop. It has only one difference that in do...while, condition is checked after the execution of the loop body. That is why this loop is considered as exit control loop. In do...while loop, body of loop will be executed at least once before checking the condition

Syntax:

```
do  
{  
    statement1;  
}while(condition);
```


Example:

```
class dowhileLoopTest {  
  
    public static void main(String args[]) {  
  
        int j = 10;  
  
        do {  
  
            System.out.println(j);  
  
            j = j+1;  
  
        } while (j <= 10)  
  
    }  
  
}
```

Output: 10

For Statement :

It is the most common and widely used loop in Java. It is the easiest way to construct a loop structure in code as initialization of a variable, a condition and increment/decrement are declared only in a single line of code. It is easy to debug structure in Java.

Syntax:

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

Example:

```
class forLoopTest
{
    public static void main(String args[])
    {
        for (int j = 1; j <= 5; j++)
            System.out.println(j);
    }
}
```

Output:

```
1
2
3
4
5
```

For-Each Loop :

For-Each loop is used to traverse through elements in an array. It is easier to use because we don't have to increment the value. It returns the elements from the array or collection one by one.

Example:

```
class foreachDemo {
    public static void main(String args[]) {
        int a[] = {10,15,20,25,30};
        for (int i : a) {
```

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

Output:

```
10
15
20
25
30
```

Branching Statements in Java :

Branching statements jump from one statement to another and transfer the execution flow. There are 3 branching statements in Java.

Break :

Break statement is used to terminate the execution and bypass the remaining code in loop. It is mostly used in loop to stop the execution and comes out of loop. When there are nested loops then break will terminate the innermost loop.

Example:

```
class breakTest {

    public static void main(String args[]) {

        for (int j = 0; j < 5; j++) {

            // come out of loop when i is 4.
```

```
        if (j == 4)
            break;

        System.out.println(j);
    }

    System.out.println("After loop");
}
}
```

Output:

0

1

2

3

4

After loop

Continue :

Continue statement works same as break but the difference is it only comes out of loop for that iteration and continue to execute the code for next iterations. So it only bypasses the current iteration.

Example:

```
class continueTest {

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

```

for (int j = 0; j < 10; j++) {

    // If the number is odd then bypass and continue with next value

    if (j%2 != 0)

        continue;

    // only even numbers will be printed

    System.out.print(j + " ");

}

}

}

```

Output:

0 2 4 6 8

ASIS FOR COMPARISON	BREAK	CONTINUE
Task	It terminates the execution of remaining iteration of the loop.	It terminates only the current iteration of the loop.
Control after break/continue	'break' resumes the control of the program to the end of loop enclosing that 'break'.	'continue' resumes the control of the program to the next iteration of that loop enclosing 'continue'.

Causes	It causes early termination of loop.	It causes early execution of the next iteration.
Continuation	'break' stops the continuation of loop.	'continue' do not stops the continuation of loop, it only stops the current iteration.
Other uses	'break' can be used with 'switch', 'label'.	'continue' can not be executed with 'switch' and 'labels'.

Return :

Return statement is used to transfer the control back to calling method. Compiler will always bypass any sentences after return statement. So, it must be at the end of any method. They can also return a value to the calling method.

1.1.16 Introducing Classes

Class fundamentals

How to create classes by using the following basics:

- » The parts of a class definition
- » Declaring and using instance variables
- » Defining and using methods
- » Creating Java applications, including the main() method and how to pass arguments to a Java program from a command line

Definition :

A class is a sort of template which has attributes and methods. An object is an instance of a class, e.g. Ram is an object of type Person.

A class is defined as follows:

```
class classname {  
    // declare instance variables  
  
    type var1;  
    type var2;  
    // ... type varN;  
  
    // declare methods  
  
    type method1(parameters) {  
        // body of method  
    }  
  
    type method2(parameters) {  
        // body of method  
    }  
  
    // ...  
  
    type methodN(parameters) {  
        // body of method  
    }  
}
```

The classes we have used so far had only one method, **main()**, however not all classes specify a main method. The main method is found in the main class of a program (starting point of program).

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called Box that defines three instance variables: width, height, and depth. Currently, Box does not contain any methods (but some will be added soon).

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

As stated, a class defines a new type of data. In this case, the new data type is called Box. You will use this name to declare objects of type Box. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type Box to come into existence.

Declaration of Object and its initialization

To create an object of the class we use **new** keyword as shown below syntax

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

To actually create a Box object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, mybox will be an instance of Box. Thus, it will have “physical” reality. For the moment, don't worry about the details of this statement.

As mentioned earlier, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every Box object will contain its own copies of the instance variables width, height, and depth. To access these variables, you will use the dot (.) operator. The dot operator links the name of the object with the name of an instance variable.

To set the values of the parameters we use the following syntax:

```
object.member=value;
```

For example, to assign the width variable of mybox the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of width that is contained within the mybox object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object.

Here is a complete program that uses the Box class:

```
/* A program that uses the Box class. Call this file BoxDemo.java */
```

```
class Box {  
    double width;  
    double height;  
    double depth;  
}  
  
// This class declares an object of type Box.  
  
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox = new Box();  
        double vol;  
        // assign values to mybox's instance variables  
        mybox.width = 10;  
        mybox.height = 20;  
        mybox.depth = 15;
```

```

        // compute volume of box

        vol = mybox.width * mybox.height * mybox.depth;

        System.out.println("Volume is " + vol);

    }

}

```

You should call the file that contains this program BoxDemo.java, because the main() method is in the class called BoxDemo, not the class called Box. When you compile this program, you will find that two .class files have been created, one for Box and one for BoxDemo. The Java compiler automatically puts each class into its own .class file. It is not necessary for both the Box and the BoxDemo class to actually be in the same source file. You could put each class in its own file, called Box.java and BoxDemo.java, respectively.

To run this program, you must execute BoxDemo.class. When you do, you will see the following output:

```
Volume is 3000.0
```

As stated earlier, each object has its own copies of the instance variables. This means that if you have two Box objects, each has its own copy of depth, width, and height. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another.

Constructors

- **Constructor** is a **special type of method** that is used to initialize the object.
- Constructor is **invoked at the time of object creation**. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating Constructor

There are basically two rules defined for the constructor.

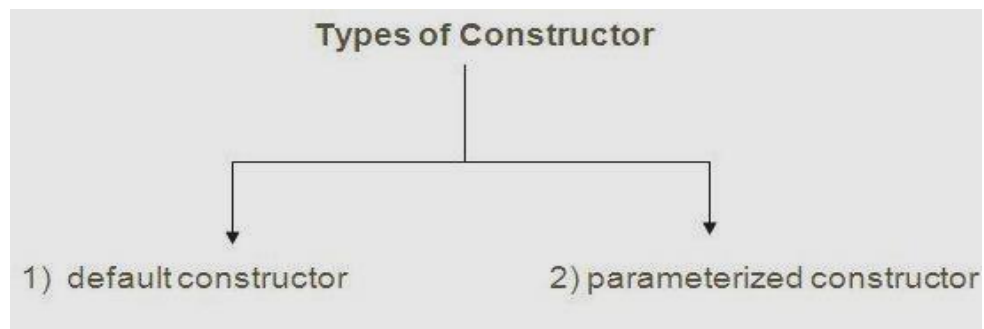
1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of Constructors

There are two types of constructors:

1. default constructor (no-arg constructor)

2. parameterized constructor



1) Default Constructor

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

Syntax of Default Constructor:

```
class_name( )
```

```
{
```

```
Statements;
```

```
}
```

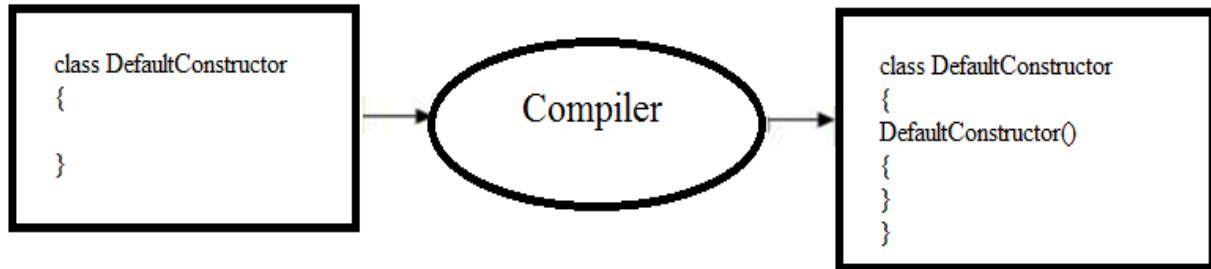
Example of Default Constructor

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

```
class DefaultConstructor
{
DefaultConstructor()
{
System.out.println("hai");
}
public static void main(String args[])
{
DefaultConstructor obj=new DefaultConstructor();
}
}
```

Output: hai

Rule: *If there is no constructor in a class, compiler automatically creates a default constructor.*



Purpose of Default Constructor

Default constructor provides the default values to the object like 0, null etc. depending on the type. **Example of default constructor that displays the default values**

```
class DefaultC
{
int stdid;
String name;
static void display()
{
System.out.println("stdid"+"");
System.out.println("name"+"");
}
public static void main(String args[])
{
DefaultC dc=new DefaultC();
dc.display();
}
}
```

Explanation : In the above class, you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

2. Parameterized constructor

A constructor that has parameters is known as a parameterized constructor.

Usage of Parameterized Constructor

Parameterized constructor is used to provide different values to the distinct objects.

Example on Parameterized constructor :

```
class Paraconstructor
{
int stdid;
String name;
Paraconstructor(int x,String n)
{
stdid=x;
name=n;
}
void display()
{
System.out.println("stdid is"+" "+stdid);
System.out.println("name is"+" "+name);
}
public static void main(String args[])
{
Paraconstructor obj1=new Paraconstructor(5,"abc");
obj1.display();
Paraconstructor obj2=new Paraconstructor(7,"def");
obj2.display();
}
}
```

Output:

stdid is 5

name is abc

stdid is 7

name is def

Difference between constructor and method

Constructor	Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Understanding of This Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the 'this' keyword. this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked. You can use this anywhere a reference to an object of the current class' type is permitted.

To better understand what this refers to, consider the following version of Box():

/ Here, Box uses a parameterized constructor to initialize the dimensions of a box.*/*

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
// compute and return volume
double volume() {
```

```

return width * height * depth;
}
}
class BoxDemo {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1= new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);

double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}

```

The use of `this` is redundant, but perfectly correct. Inside `Box()`, `this` will always refer to the invoking object.

Instance Variable Hiding

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

```

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

```

A word of caution: The use of *this* in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity, and use `this` to overcome the instance variable hiding. It is a matter of taste which approach you adopt.

1.1.17 Methods and Classes

Class Methods in Java

A method is a collection of statements that are grouped together to perform an operation.

Methods describe behavior of an object.

For example, if we have a class Human, then this class should have methods like eating(), walking(), talking(), etc, which in a way describes the behaviour which the object of this class will have.

Syntax:

```
return-type methodName(parameter-list)

{

    //body of method

}
```

Example of a Method:

```
public String getName(String st) {

    String name="StudyTonight";

    name=name+st;

    return name;

}
```

```
public String getName (String st)
```



modifier

return-type

method-name

parameter

Modifier : Modifier are access type of method. We will discuss it in detail later.

Return Type : A method may return value. Data type of value return by a method is declare in method heading.

Method name : Actual name of the method.

Parameter : Value passed to a method.

Method body : collection of statement that defines what method does.

Parameter Vs. Argument in a Method :

While talking about method, it is important to know the difference between two terms parameter and argument.

Parameter is variable defined by a method that receives value when the method is called. Parameter are always local to the method they dont have scope outside the method. While argument is a value that is passed to a method when it is called.

```
public void sum( int x, int y )
{
    System.out.println(x+y);
}
public static void main( String[ ] args )
{
    Test b=new Test( );
    b.sum( 10, 20 );
}
```

Overloading:

Overloading:

Overloading is a machanism by which Java supports polymorphism.

In Java we have 2 types of Overloading namely

i)Method Overloading

ii)Constructor Overloading

i)Method overloading:

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

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.

Example on Method Overloading

```
class Methodoverload
{
void display()
{
System.out.println("display method without arguements");
}
void display(int a)
{
System.out.println("display method with integer"+" "+a);
}
void display(int a,double b)
{
System.out.println("display method with integer"+a+"and double"+b);
}
public static void main(String args[])
{
Methodoverload obj=new Methodoverload();
obj.display();
obj.display(10);
obj.display(20,65.99);
}
}
```

Output:

display method without arguments

display method with integer 10

display method with integer 20 and double 65.99

Constructor Overloading:

In addition to overloading normal methods, you can also overload constructor methods

```
class Coverload
{
    int a;double b;
    Coverload()
    {
        System.out.println("constructor without parameters");
    }
    Coverload(int i)
    {
        a=i;
        System.out.println("constructor with one parameter");
        System.out.println("a value is"+a);
    }
    Coverload(int i,double d)
    {
        a=i;
        b=d;

        System.out.println("constructor with two parameter");
        System.out.println("a value is"+a);
        System.out.println("b value is"+b);
    }
    public static void main(String args[])
    {
        Coverload obj1=new Coverload();
        Coverload obj2=new Coverload(10);
        Coverload obj3=new Coverload(40,29.99);
    }
}
```

Output:

constructor without parameters

constructor with one parameter
a value is 10
constructor with two parameter
a value is 40
b value is 29.99

Recursion:

Recursion:

Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

Example on Recursion:

```
class Recursion
{
    int fact(int n)
    {
        int result;
        if(n==0||n==1)
            return 1;
        else
            result=n*fact(n-1);
        return result;
    }
    public static void main(String args[])
    {
        int ans;
        Recursion obj=new Recursion();
        ans=obj.fact(5);
        System.out.println("factorial of 5 is"+ans);
    }
}
```

Output:factorial of 5 is **120.**

Parameter passing Techniques (or) call-by-value and call-by-reference

call-by-value and call-by-reference :

There are two ways to pass an argument to a method

1. **call-by-value** : In this approach copy of an argument value is pass to a method. Changes made to the argument value inside the method will have no effect on the arguments.
2. **call-by-reference** : In this reference of an argument is pass to a method. Any changes made inside the method will affect the argument value.

Example of call-by-value:

```
public class Test {  
    public void callByValue(int x) {  
        x=100;  
    }  
    public static void main(String[] args) {  
        int x=50;  
        Test t = new Test();  
        t.callByValue(x);//function call  
        System.out.println(x);  
    }  
}
```

Output: 50

Example of call-by-Reference:

```
public class CBRTest {
    int a;
    CBRTest(int i)
    {
        a=i;
    }
    public void callByReference(CBRTest obj)
    {
        obj.a=obj.a+10;
    }
    public static void main(String[] args)
    {
        CBRTest obj = new CBRTest(10);
        System.out.println("before calling a value is"+obj.a);
        obj.callByReference(obj)//function call by passing object
        System.out.println("after calling a value is"+obj.a);
    }
}
```

Output:

before calling a value is 10

after calling a value is 20

NOTE: When a primitive type is passed to a method, it is done by use of call-by-value. Objects are implicitly passed by use of call-by-reference.

Understanding of Static and Final Keyword

Static Keyword:

The static keyword in Java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than an instance of the class.

The static can be:

- i. Variable (also known as a class variable)
- ii. Method (also known as a class method)
- iii. Block

i).Java static variable

If you declare any variable as static, it is known as a static variable.

The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.

The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program memory efficient (i.e., it saves memory).

Understanding the problem without static variable

```
class Student{  
  
    int rollno;  
  
    String name;  
  
    String college="BVRITH";  
  
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

Note:Java static property is shared to all objects

//Java Program to demonstrate the use of static variable

```
class Student{  
  
    int rollno;//instance variable  
  
    String name;  
  
    static String college ="BVRITH";//static variable  
  
}
```

Note:As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value

```
class Counter{  
  
    static int count=0;//will get memory only once and retain its value  
  
    Counter(){  
  
        count++;//incrementing the value of static variable  
  
        System.out.println(count);  
  
    }  
  
    public static void main(String args[]){  
  
        //creating objects  
  
        Counter c1=new Counter();  
  
        Counter c2=new Counter();  
  
        Counter c3=new Counter();  
  
    }  
  
}
```

Output:

1

2

3

ii) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

```
class Calculate{
static int cube(int x){
return x*x*x;
}
public static void main(String args[]){
int result=Calculate.cube(5);
System.out.println(result);
}
}
```

Restrictions for the static method

There are two main restrictions for the static method. They are:

- The static method can not use non static data member or call non-static method directly.
- this and super cannot be used in static context.

iii) Java static block

Is used to initialize the static data member.

It is executed before the main method at the time of classloading.

Example:

```
class Demo{

static{ System.out.println("static block is invoked");}

public static void main(String args[]){

System.out.println("Hello main");
```

```
}
```

```
}
```

Output:

static block is invoked

Hello main

Final Keyword In Java:

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- variable
 - method
 - class
- **if a variable is made as final it cannot change its value**

Example:

```
class Demo
```

```
{
```

```
int a=10;
```

```
final int b=20;
```

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

```
{
```

```
a=a+10;
```

```
System.out.println(a);
```

```
b=b+20;//error because variable 'b' is final
```

```
System.out.println(b);
```

```
}
```

```
}
```

➤ **if a method is made as final it cannot override it.**

Class Demo

{

Final void display()

{

System.out.println("hello");

}

}

Class FinalDemo extends Demo

{

void display() //overriding is not permitted

{

System.out.println("hello");

}

public static void main(String arg[])

{

FinalDemo fd=new FinalDemo();

fd. display();

}

}

If a class is made as final it cannot be extended by another class

Program to demonstrate using final with inheritance

```

final class A
{
}

Class B extends A //error since A is final we can't inherit it properties
{
    void display()
    {
        System.out.println("demo on final");
    }

    public static void main(String args[]){
        B obj=new B();
        Obj.display();
    }
}

```

symbolic constants

Symbolic constants in Java are named constants. Constants may appear repeatedly in number of places in the program. Constant values are assigned to some names at the beginning of the program, then the subsequent use of these names in the program has the effect of caving their defined values to be automatically substituted in appropriate points. The constant is declared as follows:

Syntax : final datatype symbolicname= value;

Eg: final float PI =3.14159;
 final int STRENGTH =100;

Rules :-

1. Symbolic names take the some form as variable names. But they one written in capitals to distance from variable names. This is only convention not a rule.
2. After declaration of symbolic constants they shouldn't be assigned any other value within the program by using an assignment statement.

For eg:- STRENGTH = 200 is illegal

3. They can't be declared inside a method. They should be used only as class data members in the beginning of the class.

```
class SymbolicDemo
{
static final int A=10;

static final double PI=3.141;

public static void main(String args[])
{
//final int A=10;

//final double PI=3.141;

int a=20,sum;

double r=30.49;

sum=A+a;

System.out.println("sum is "+sum);

double area=PI*r*r;

System.out.println("area is "+area);

//A=A+1;//ERROR CANNOT ASSIGN A VALUE FOR FINAL VARIABLE

}

}
```

Introduction of Nested and Inner classes

Nested classes in Java

A class declared inside a class is known as nested class.

We use nested classes to logically group classes and interfaces in one place so that it can be more readable and maintainable code.

Additionally, nested class can access all the members of outer class including private data members and methods but viceverse is not possible

Syntax of Nested class

```
class Outer_class_Name
{
    ...
    class Nested_class_Name
    {
        ...
    }
    ...
}
```

Advantage of nested classes

There are basically three advantages of nested classes. They are

- Nested classes represent a special type of relationship that is it can access all the members (data members and methods) of outer class including private.
- Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.

- Code Optimization: It requires less code to write.

Types of Nested class:

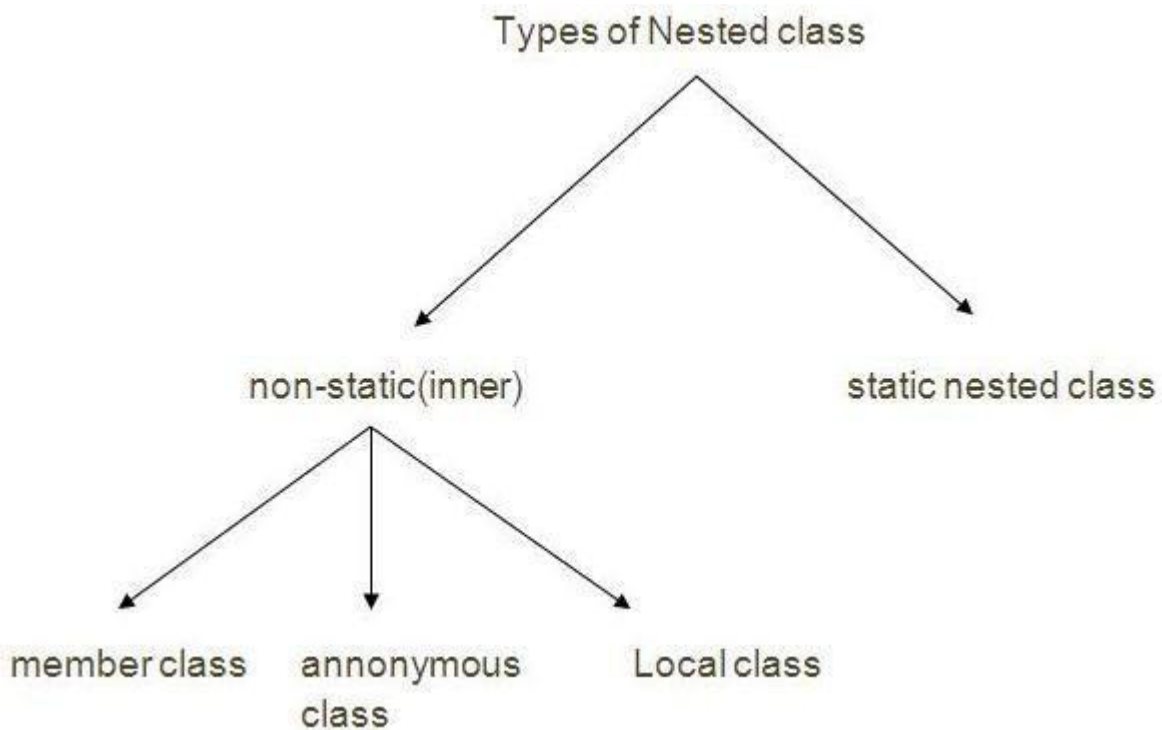
There are two types of nested classes non-static and static nested classes.

The Non-static nested classes are also known as **Inner classes**.

1. Non-static nested class(inner class)

- a) Member inner class
- b) Local inner class
- c) Anonymous inner class

2. Static nested class



1.Member inner class

A class that is declared inside a class but outside a method is known as member inner class.

Invocation of Member Inner class

- ✓ From within the class
- ✓ From outside the class

Example of member inner class :

In this example, we are invoking the method of member inner class from the display method of Outer class.

```

class Outer
{
private int data=20;
class MemberInner
{
void message()
{
System.out.println("private data of outerclass is"+data);
}
}
void display()
{
MemberInner in=new MemberInner();
in.message();
}
public static void main(String args[])
{
Outer out=new Outer();
out.display();
}
}

```

Output:

private data of outerclass is 20

2) Local inner class

A class that is created inside a method is known as local inner class. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Program of local inner class

```
class Outer{
    private int data=30;//instance variable
    void display()
    {
        class LocalInner
        {
            void message()
            {
                System.out.println("private data of outer class"+data);
            }
        }
        LocalInner in=new LocalInner();
        in.message();
    }
    public static void main(String args[]){
        Outer obj=new Outer();
        obj.display();
    }
}
```

Output:

private data of outer class 30

Rules for Local Inner class

- 1) Local inner class cannot be invoked from outside the method.
- 2) Local inner class cannot access non-final local variable.

3) Anonymous inner class

A class that have no name is known as anonymous inner class.

Example:**Program of anonymous inner class by abstract class**

abstract class Demo

```

{
abstract void display();
}
class Annomynous
{
public static void main(String args[])
{

Demo obj=new Demo()
{
void display()
{
System.out.println("Demo on annomynous inner class");
}
};
obj.display();
}
}
Output: Demo on annomynous inner class

```

4) static nested class

A static class that is created inside a class is known as static nested class. It cannot access the non-static members outer class.

- It can access static data members of outer class including private.
- static nested class cannot access non-static (instance) data member or method.

Program of static nested class that have instance method

```

class Outer{
    static int data=30;

    static class Inner
    {
        void msg(){System.out.println("data is "+data);}
    }

    public static void main(String args[]){
        Outer.Inner obj=new Outer.Inner();
        obj.msg();
    }
}

```

Output:data is 30

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.

1.1.18 Java String

Introduction to Java string class and String object

- String is a sequence of characters. But in java, string is an object that represents a sequence of characters. The java.lang.String class is used to create string object.

For example:

```
char[] ch={'j','a','v','a'};
```

```
String s=new String(ch);
```

is same as:

```
String s="java";
```

- Java String class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

There are two ways to create String object:

- i. By **String Literal**
- ii. By new keyword

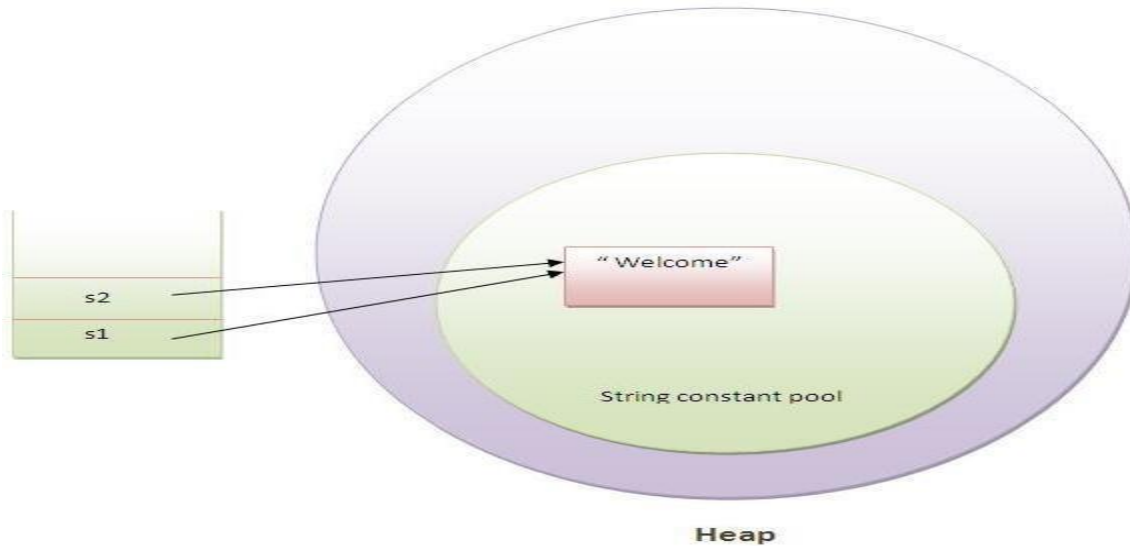
i) String Literal

Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
String s2="Welcome";//will not create new instance
```



ii) By new keyword

String s= new String("Welcome"); //creates two objects and one reference variable

In such case, JVM will create a new string object in normal(non pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in heap(non pool).

Java String Example:

```
public class StringExample
{
    public static void main(String args[])
    {
        String s1="java";//creating string by java string literal
        char ch={'s','t','r','i','n','g','s'};
        String s2=new String(ch);//converting char array to string
        String s3=new String("example");//creating java string by new keyword
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
    }
}
```

String manipulation methods

Java String class method

String objects are immutable, whenever you want to modify a String, you must either copy it into a StringBuffer or StringBuilder, or use one of the following String methods, which will construct a new copy of the string with your modifications complete.

The java.lang.String class provides many useful methods to perform operations on sequence of char values

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns formatted string
4	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
5	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value
6	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string
7	boolean equals(Object another)	checks the equality of string with object

8	<code>boolean isEmpty()</code>	checks if string is empty
9	<code>String concat(String str)</code>	concatinates specified string
10	<code>String replace(char old, char new)</code>	replaces all occurrences of specified char value
11	<code>String replace(CharSequence old, CharSequence new)</code>	replaces all occurrences of specified CharSequence
12	<code>static String equalsIgnoreCase(String another)</code>	compares another string. It doesn't check case.
13	<code>String[] split(String regex)</code>	returns splitted string matching regex
14	<code>String[] split(String regex, int limit)</code>	returns splitted string matching regex and limit
15	<code>int indexOf(int ch)</code>	returns specified char value index
16	<code>int indexOf(int ch, int fromIndex)</code>	returns specified char value index starting with given index
17	<code>int indexOf(String substring)</code>	returns specified substring index
18	<code>int indexOf(String substring, int fromIndex)</code>	returns specified substring index starting with given index
19	<code>String toLowerCase()</code>	returns string in lowercase.

20	String toUpperCase()	returns string in uppercase.
21	String trim()	removes beginning and ending spaces of this string.
22	static String.valueOf(int value)	converts given type into string. It is overloaded.

Program on StringHandling :

```

public class StringhandlingExample
{
    public static void main(String args[])
    {
        String name="JAVA PROGRAMMING";
        String s1="hello";
        String s2="hello";
        String joinstring,stringlower,stringupper;
        char ch=name.charAt(4);//returns the char value at the 4th index
        System.out.println(ch);
        System.out.println(s1.compareTo(s2));//0 because both are equal
        System.out.println(s1.equals(s2));//true because content and case is same
        System.out.println("string length is: "+s1.length());//5 is the length of hello string
        joinstring=string.join("welcome", "to", "javaprogramming");
        System.out.println(joinstring); //welcome to javaprogramming
    }
}

```

```

stringlower=name.toLowerCase( );

System.out.println(stringlower); // javaprogramming

stringupper= s1.toUpperCase();

System.out.println(stringupper); //HELLO

System.out.println(s1.trim()+"javaprogramming");//hellojavaprogramming

//space between s1 contents and javaprogramming is removed

}

}

```

StringBuffer

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

StringBuffer Constructors

Important Constructors of StringBuffer class

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.

public char	charAt(int index)	is used to return the character at the specified position.
public int	length()	is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to `main()`. A command-line argument is the information that directly follows the program's name on the command line when it is executed.

To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a `String` array passed to the `args` parameter of `main()`. The first command-line argument is stored at `args[0]`, the second at `args[1]`, and so on.

Example 1:

The following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
```

```
class CommandLine {  
    public static void main(String args[]) {  
        for(int i=0; i<args.length; i++)  
            System.out.println("args[" + i + "]: " +args[i]);  
    }  
}
```

Running:

```
C:>\ java CommandLine have a nice time
```

output:

```
args[0]:have  
args[1]: a  
args[2]:nice  
args[3]:time
```

REMEMBER All command-line arguments are passed as strings.

Example 2:

Program to add to numbers that are passed through command prompt

```
class CommandLine  
{  
    public static void main(String args[])  
    {  
        int a,b,sum;  
        a=Integer.parseInt(args[0]);  
        b=Integer.parseInt(args[1]);  
        sum=a+b;  
        System.out.println("addition of"+a+"and"+b+"is"+sum);  
    }  
}
```

1. Inheritance Concept

Inheritance is the mechanism of deriving new class from old one, old class is known as *superclass* and *new class* is known as **subclass**. The subclass inherits all of its instances variables and methods defined by the superclass and it also adds its own unique elements. Thus we can say that subclass are specialized version of superclass.

Benefits of Java's Inheritance

1. Reusability of code
2. Code Sharing
3. Consistency in using an interface

1. Inheritance Basics

Definition: The process by which one class acquires the properties (data members) and functionalities (methods) of another class is called *inheritance*.

(or)

Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.

Child Class:

The class that extends the features of another class is known as child class, sub class or derived class.

Parent Class:

The class whose properties and functionalities are used (inherited) by another class is known as parent class, super class or Base class.

Inheritance allows us to reuse of code, it improves reusability in your java application.

The biggest advantage of Inheritance is that the code that is already present in base class need not be rewritten in the child class. This means that the data members(instance variables) and methods of the parent class can be used in the child class. Child class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

Syntax: Inheritance in Java

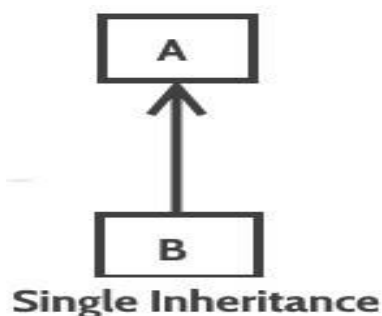
To inherit a class we use extends keyword. Here class XYZ is child class and class ABC is parent class. The class XYZ is inheriting the properties and methods of ABC class.

```
class XYZ extends ABC  
  
{  
  
}
```

Types of inheritance

1. Single Inheritance
2. Multilevel inheritance
3. [Hierarchical inheritance](#)
4. Multiple Inheritance

1.Single Inheritance: refers to a child and parent class relationship where a class extends the another class.



Program to demonstrate Single Level Inheritance:

```

class A
{
int i=10;
void Adisplay()
{
System.out.println("class A display");
}
}
class B extends A
{
int j=20;
void Bdisplay()
{
System.out.println("class B display");
}
public static void main(String args[])
{
B obj=new B();
obj.Adisplay();
obj.Bdisplay();
System.out.println("i value is "+obj.i);
System.out.println("j value is "+obj.j);
}
}

```

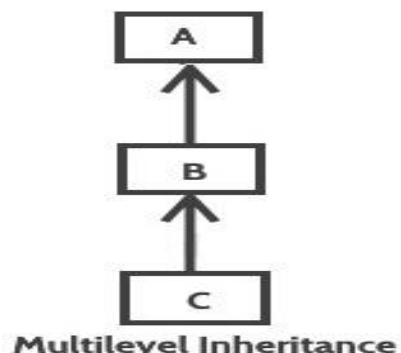
class A display

class B display

i value is 10

j value is 20

2. Multilevel inheritance: refers to a child and parent class relationship where a class extends the child class. For example class C extends class B and class B extends class A.



Program to demonstrate Multi Level Inheritance:

```
class A
{
A()
{
System.out.println("A constructor");
}
void Amethod()
{
System.out.println("method of A");
}
}
class B extends A
{
B()
{
System.out.println("B constructor");
}
void Bmethod()
{
System.out.println("method of B");
}
void welcome()
{
System.out.println("class B-welcome method");
}
}
class C extends B
{
C()
{
System.out.println("C constructor");
}
void welcome()
{
System.out.println("class C- welcome method");
}
}
public static void main(String args[])
{
C oc=new C();
oc.Amethod();
oc.Bmethod();
oc.welcome();
}
```

```
}
}
```

Output:

A constructor

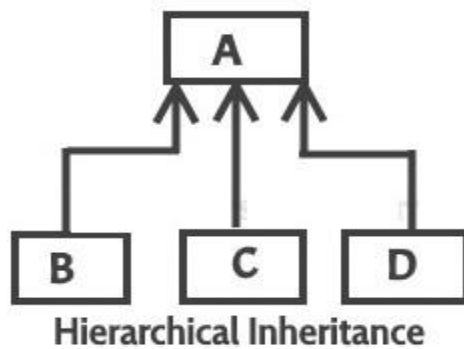
B constructor

C constructor

method of A

method of B

method of A



3. Hierarchical inheritance: refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.

```
class A
{
A()
{
System.out.println("A constructor");
}
void Amethod()
{
System.out.println("method of A class");
}
}
class B extends A
{
```

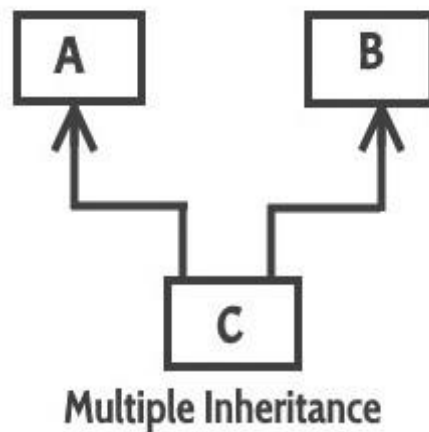


```
B()
{
System.out.println("B constructor");
}
void Bmethod()
{
System.out.println("method of B class");
}
}
class D extends A
{
D()
{
System.out.println("D constructor");
}
void Dmethod()
{
System.out.println(" method of D class");
}
public static void main(String args[])
{
D od=new D();
od.Amethod();
od.Dmethod();
B ob=new B();
ob.Amethod();
ob.Bmethod();
}
}
```

Output:

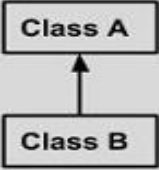
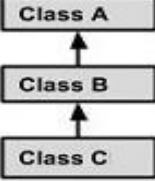
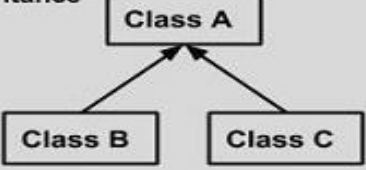
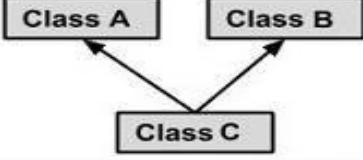
```
A constructor
D constructor
method of A class
method of D class
A constructor
B constructor
method of A class
method of B class
```

4. Multiple Inheritance: refers to the concept of one class extending more than one classes, which means a child class has two parent classes. For example class C extends both classes A and B. Java doesn't support multiple inheritances.



5. Hybrid inheritance: Combination of more than one types of inheritance in a single program. For example class A & B extends class C and another class D extends class A then this is a hybrid inheritance example because it is a combination of single and hierarchical

inheritance.

Single Inheritance  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
Multi Level Inheritance  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A {} public class B extends A {.....} public class C extends B {.....} </pre>
Hierarchical Inheritance  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A {} public class B extends A {.....} public class C extends A {.....} </pre>
Multiple Inheritance  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> public class A {} public class B {.....} public class C extends A,B { } // Java does not support mutiple Inheritance </pre>

1. Member Access

Member Access and Inheritance

An instance variable of a class will be declared private to prevent its unauthorized use or tampering. Inheriting a class *does not* overrule the private access restriction. Thus, even though a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared private. For example, if, as shown here, width and height are made private in TwoDShape, then Triangle will not be able to access them:

```
//private members are not inherited
```

```
//This program does not compile....
```

```
Class PrivateAccess
```

```
{
```

```

private int a=10;

private int b=20;

int c;

void display()

{

c=a+b;

System.out.println(c);

}

}

Class Demo extends PrivateAccess

{

Void sub()

{

c=a-b;//error private member a and b can't be inherited.

}

}

```

Java-Superclass Variable Can Reference a Subclass Object

Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. In the following program, obj is a reference to NewData object, Since NewData is a subclass of Data, it is permissible to assign obj a reference to the NewData object. When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is way obj can't access data3 and data4 even it refers to a NewData object.

Program

```
class Data {  
  
    int data1;  
    int data2;  
}  
  
class NewData extends Data{  
  
    int data3;  
    int data4;  
}  
  
public class Javaapp {  
  
    public static void main(String[] args) {  
  
        Data obj = new NewData();  
        obj.data1 = 50;  
        obj.data2 = 100;  
        System.out.println("obj.data1 = "+obj.data1);  
        System.out.println("obj.data2 = "+obj.data2);  
    }  
}
```

Program Output

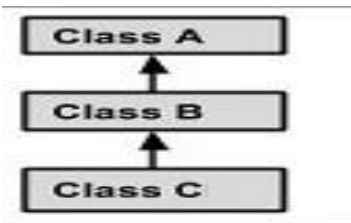
```
obj.data1 = 50  
obj.data2 = 100
```

1. Creating Multilevel Hierarchy

Create multilevel hierarchy

It is nothing but the enhancement of the Simple Inheritance. From the type name, it is pretty much clear that Inheritance is done at 'n' number of levels, where $n > 1$.

In simple inheritance, a subclass or derived class derives the properties from its parent class, but in multilevel inheritance, a subclass is derived from a derived class. One class inherits the only single class. Therefore, in multilevel inheritance, every time ladder increases by one. The lowermost class will have the properties of all the **superclass**.



Program to demonstrate Multi Level Inheritance/hierarchy:

```

class A
{
A()
{
System.out.println("A constructor");
}
void Amethod()
{
System.out.println("method of A");
}
}
class B extends A
{
B()
{
System.out.println("B constructor");
}
void Bmethod()
{
System.out.println("method of B");
}
void welcome()
{
System.out.println("class B-welcome method");
}
}
  
```

```
}  
}  
class C extends B  
{  
    C()  
    {  
        System.out.println("C constructor");  
    }  
    void welcome()  
    {  
        System.out.println("class C- welcome method");  
    }  
    public static void main(String args[])  
    {  
        C oc=new C();  
        oc.Amethod();  
        oc.Bmethod();  
        oc.welcome();  
    }  
}
```

Output:

A constructor

B constructor

C constructor

method of A

method of B

method of A

Note: Multilevel inheritance is not multiple inheritances where one class can inherit more than one class at a time. **Java does not support multiple inheritances.**

1. Super Uses

super keyword in java

The **super** keyword in java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable

Usage of java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

The use of super keyword

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

Example program to demonstrating invoking of parent class instance variable and parent class method

```
class A
{
int data=10;
void display()
{
System.out.println(" A class display");
}
}

class B extends A
{
int data=20;

void display()
{
System.out.println(" B class display");
}

void superdisplay()
{
System.out.println("data in A class is"+super.data);
super.display();
}

public static void main(String args[])
{
B obj=new B();
obj.display();
System.out.println(obj.data);
obj.superdisplay();
}
}
```

Output

```
B class display
20
data in A class is 10
A class display
```

3) *super* is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor.

```
class Parentclass
{
    Parentclass()
    {
        System.out.println("Constructor of parent class");
    }
}
class Subclass extends Parentclass
{
    Subclass()
    {
        //Compile implicitly adds super() here as the first statement of this
        constructor.
        System.out.println("Constructor of child class");
    }
    Subclass(int num)
    {
        // Even though it is a parameterized constructor The compiler still adds the super() here
        System.out.println("arg constructor of child class");
    }
    void display()
    {
        System.out.println("Hello!");
    }
    public static void main(String args[])
    {
        Subclass obj= new Subclass();
        //Calling sub class method
        obj.display();
        Subclass obj2= new Subclass(10);
        obj2.display();
    }
}
```

Output:

Constructor of parent class

Constructor of child class

Hello!

Constructor of parent class

arg constructor of child class

Hello!

1. Using Final with Inheritance

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- variable
- method
- class

if a variable is made as final it cannot change its value

if a method is made as final it cannot override it.

If a class is made as final it cannot be extended by another class

Program to demonstrate using final with inheritance

```
final class A
{
}

Class B extends A //error since A is final we can't inherit it properties
{
    void display()
    {
        System.out.println("demo on final");
    }

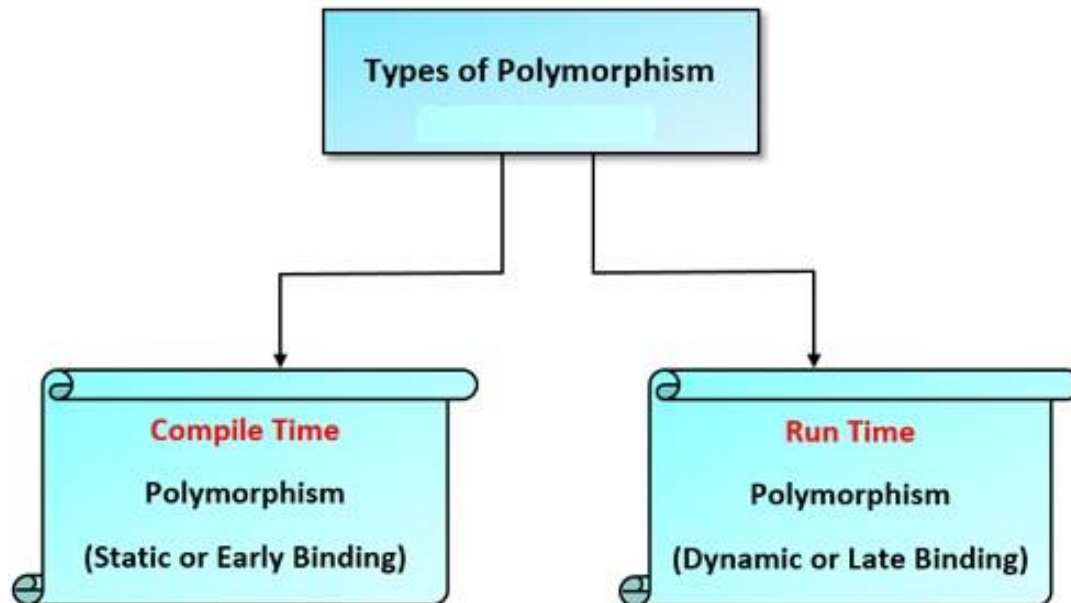
    public static void main(String args[]){
        B obj=new B();
        Obj.display();
    }
}
```

1. Polymorphism-ad hoc polymorphism

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java:

1. Compile-time polymorphism or Ad hoc polymorphism
2. Runtime polymorphism. Or Pure polymorphism



We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polyymorphism.

1. Compile-time polymorphism or Ad hoc polymorphism:

Ad hoc polymorphism is also known as function overloading or operator overloading because a polymorphic function can represent a number of unique and potentially different implementations depending on the type of argument it is applied to.

The term ad hoc in this context is not intended to be pejorative; it refers simply to the fact that this type of polymorphism is not a fundamental feature of the type system.

i)operator overloading:

Java also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands, adds them and when placed between string operands, concatenates them.

In java, Only "+" operator can be overloaded:

Example on Operator overloading

```
// Java program for Operator overloading

class Operatoroverloading {

    void operator(String str1, String str2)
    {
        String s = str1 + str2;
        System.out.println("Concatinated String - " + s);
    }

    void operator(int a, int b)
    {
        int c = a + b;
        System.out.println("Sum = " + c);
    }
}

class Main {
    public static void main(String[] args)
    {
        Operatoroverloading obj = new Operatoroverloading ();
        obj.operator(2, 3);
        obj.operator("BVRIT", "H");
    }
}
```

Output:

Sum = 5

Concatinated String -BVRITH

ii) Method Overloading:

When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments. Overloaded methods are generally used when they conceptually execute the same task but with a slightly different set of parameters.

Example on method overloading

```
class MultiplyFun {  
  
    // Method with 2 parameter  
    static int Multiply(int a, int b)  
    {  
        return a * b;  
    }  
    // Method with 3 parameter  
    static int Multiply(int a, int b,int c)  
    {  
        return a * b*c;  
    }  
  
    // Method with the same name but 2 double parameter  
    static double Multiply(double a, double b)  
    {  
        return a * b;  
    }  
}  
  
class Main {  
    public static void main(String[] args)  
    {  
  
        System.out.println(MultiplyFun.Multiply(2, 4));  
        System.out.println(MultiplyFun.Multiply(2, 4,2));  
        System.out.println(MultiplyFun.Multiply(5.5, 6.3));  
    }  
}
```

Output:

8

16

34.65

Note:

we overload static methods

we cannot overload methods that differ only by static keyword

we overload main() in Java

2. Runtime polymorphism. Or Pure polymorphism

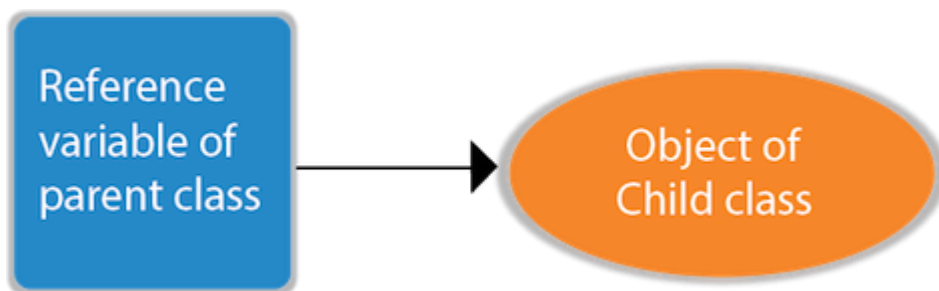
Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.



For example:

```
class A{ }
```

```
class B extends A{ }
```

```
A a=new B();//upcasting .
```

Example of Java Runtime Polymorphism

In this example, we are creating two classes A and B. B class extends A class and overrides its display () method. We are calling the display () by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class A
{
    void display ()
    {
        System.out.println(" A class display ");
    }
}
class B extends A
{
    void display ()
    {
        System.out.println(" B class display ");
    }
    public static void main(String args[])
    {
        A obj = new B();//upcasting
        obj. display ();
    }
}
```

Output:

B class display

Advantages of dynamic binding along with polymorphism with method overriding are.

- Less memory space

- Less execution time
- More performance

Difference between Static & Dynamic Polymorphism

	<i>Static Polymorphism</i>	<i>Dynamic Polymorphism</i>
1	Compile time polymorphism or static polymorphism relates to method overloading.	Run time polymorphism or dynamic polymorphism or dynamic binding relates to method overriding.

Difference between Method overloading and Method overriding

	<i>Method overloading</i>	<i>Method overriding</i>
1	When a class have same method name with different argument, than it is called method overloading.	Method overriding - Method of superclass is overridden in subclass to provide more specific implementation .
2	Method overloading is generally done in same class but can also be done in SubClass .	Method overriding is always done in subClass in java.
3	Both Static and instance method can be overloaded in java.	Only instance methods can be overridden in java. Static methods can't be overridden in java.
4	Main method can also be overloaded in java	Main method can't be overridden in java, because main is static method and static methods can't be overridden in java (as mentioned in above point)
5	private methods can be overloaded in java.	private methods can't be overridden in java, because private methods are not inherited in subClass in java.
6	final methods can be overloaded in java.	final methods can't be overridden in java, because final methods are not inherited in subClass in java.
7	Call to overloaded method is bonded at compile time in java.	Call to overridden method is bonded at runtime in java.
8	Method overloading concept is also known as compile time polymorphism or ad hoc	Method overriding concept is also known as runtime time polymorphism or pure polymorphism or Dynamic binding in java.

	polymorphism or static binding in java.	
--	---------------------------------------------------	--

1. Method Overriding

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable of the current object.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting

```
class A{ }
```

```
class B extends A{ }
```

```
A a=new B();//upcasting
```

```
class A
{
    void display ()
    {
        System.out.println(" A class display ");
    }
}

class B extends A
{
    void display ()
    {
        System.out.println(" B class display ");
    }
    public static void main(String args[])
    {
        A obj = new B();//upcasting
        obj. display ();
    }
}
```


Output:

B class display

1. Abstract Classes

Abstract class in Java

- A class that is declared with ***abstract*** keyword, is known as abstract class in java.
- It can have abstract and non-abstract methods (method with body).
- It needs to be extended and its method implemented.
- It cannot be instantiated.

Example abstract class

abstract class A

```
{  
  
}
```

Abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

Example abstract method

abstract void printStatus();//no body and abstract

Example on abstract class and method(or)Example on multilevel inheritance using abstract

```
abstract class Hello
{
    abstract void hai();
}
abstract class Demo extends Hello
{
}
class Welcome extends Demo
{
    void hai()
    {
        System.out.println("hello");
    }
}
public class AbstractClass {
public static void main(String args[])
{
    //Hello oa=new Hello();//error because for abstract class we can't create object
    //Demo ob=new Demo();//error because for abstract class we can't create object
    Welcome oh=new Welcome();
    oh.hai();
}
}
```

Output:hello

Difference Between Abstract and Concrete class

Abstract Class	Concrete Class
Abstract class contain abstract method(i.e a method without implementation)along with concrete methods	Concrete class can't contain abstract method
For Abstract class object cannot be created	For Concrete class object can be created.
Abstract classes need to be extended in order to make it complete class	Concrete class need not be extended because it is a complete class
Abstract methods of abstract class need to be implemented in subclass/child class	All concrete methods contain implementations.
Example: abstract class demo { abstract hai(); void welcome() { System.out.println("welcome"); }	Example: class demo { void welcome() { System.out.println("welcome"); }

Object Class

There is one special class, Object, defined by Java.

The **Object class** is the parent class of all the classes in java by default.

Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class.

The parent class reference variable can refer the child class object, know as upcasting.

Methods of Object class

The Object class provides many methods. They are as follows:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout) throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout, int nanos) throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait() throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize() throws Throwable	is invoked by the garbage collector before object is being garbage collected.