Unit-1 :

Object Oriented Thinking

A way of viewing world:

It means how we handle the real world situations through OOP and how we could make the computer closely model the techniques.

Eg: Raju wished to send some flowers to his friend for his birthday & he was living some miles away. He went to local florist and said the kind of flowers. He want to send to his friend’s address. And florist assured those flowers will be sent automatically.

Agents and Communities:

To solve my food delivery problem, I used a solution by finding an appropriate agent (Zomato) and pass a message containing my request. It is the responsibility of the agent (Zomato) to satisfy my request. Here, the agent uses some method to do this. I do not need to know the method that the agent has used to solve my request. This is usually hidden from me.

So, in object-oriented programming, problem-solving is the solution to our problem which requires the help of many individuals in the community. We may describe agents and communities as follows.

An object-oriented program is structured as a community of interacting agents, called objects. Where each object provides a service (data and methods) that is used by other members of the community.

In our example, the online food delivery system is a community in which the agents are zomato and set of hotels. Each hotel provides a variety of services that can be used by other members like zomato, myself, and my family in the community.

Messages & Methods:

When a message is passed to an agent (or object) that is capable of performing an action, then that action will be initiated in OOP. An object which receives the message sent is called ‘receiver’. When a receiver accepts a message, it means that the receiver has accepted the responsibility of processing the requested action. It then performs a method as a response to the message in order to fulfill the request.

Responsibilities:

A fundamental concept in OOP is to describe behavior in terms of responsibilities. A Request to perform an action denotes the desired result. An object can use any technique that helps in obtaining the desired result and this process will not have any interference from other object. The abstraction level will be increased when a problem is evaluated in terms of responsibilities. The objects will thus become independent from each other which will help in solving complex problems. An Object has a collection of responsibilities related with it which is termed as ‘protocol’

The Operation of a traditional program depends on how it acts on data structures. Where as an OOP operates by requesting data structure to perform a service.

Classes & Instances:

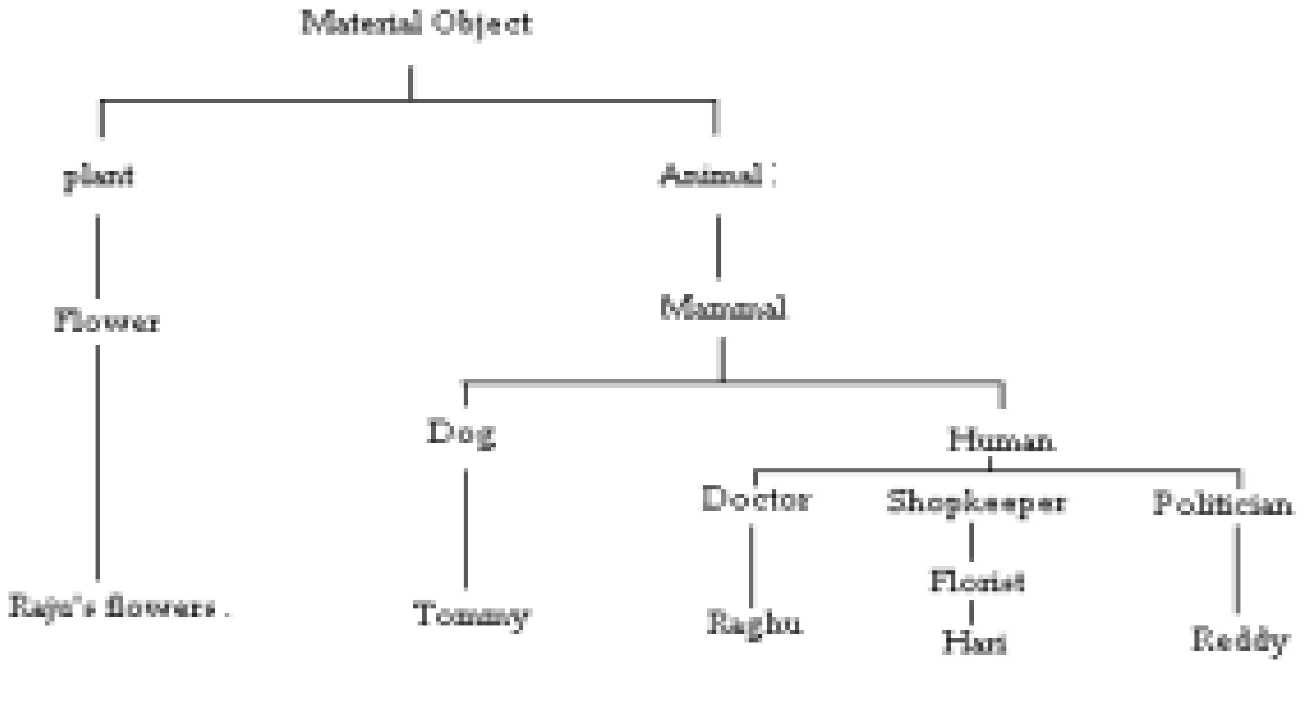
A Receiver’s class determines which method is to be invoked by the object in response to a message. When similar messages are requested then all the objects of a class will invoke the same method.

All objects are said to be the instances of a class.

For e.g., If ‘flower’ is a class then Rose is its instance

Class Hierarchies (Inheritance):

It is possible to organize classes in the form of a structure that corresponds to hierarchical inheritance. All the child classes can inherit the properties of their parent classes. A parent class that does not have any direct instances is called an abstract class. It is used in the creation of subclasses.



Class hierarchy for Different kinds of Material

Let ‘Hari’ be a florist, but florist more specific form of shot keeper. Additionally, a shop keeper is a human and a human is definitely a mammal. But a mammal is an animal & animal is material object.

All these categories along with their relationships can be represented using a graphical technique shown in figure. Each category is regarded as a class. The classes at the top of the tree are said to be more abstract classes and the classes at the bottom of the tree are said to be more specific classes.

Inheritance is a principle, according to which knowledge of a category (or class) which is more general can also be applied to a category which is more specific.

Method Binding:

When the method is super class have same name that of the method in sub class, then the subclass method overridden the super-class method. The program will find out a class to which the reference is actually pointing and that class method will be binded.

E.g.

```
class parent

{              void print()

{      System.out.println("From Parent");

}

}

class  child extends parent

{

void print()

{  System.out.println("From Child");

}

}

class  Bind

{

Public static void main(String arg[])

{     child ob=new child();

ob.print();

}

}
```

```
o/p:       From Child
```

The child's object 'ob' will point to child class print() method thus that method will be binded.

Overriding:

When the name and type of the method in a subclass is same as that of a method in its super class. Then it is said that the method present in subclass overrides the method present in super class. Calling an overridden method from a subclass will always point to the type of that method as defined by the subclass, where as the type of method defined by super class is hidden.

E.g. (above 'method binding' example)

Exceptions:

Exception is a error condition that occurs in the program execution. There is an object called 'Exception' object that holds error information. This information includes the type and state of the program when the error occurred.

E.g. Stack overflow, Memory error etc

Summary of OOP concepts (proposed by Alan kay):

1. Everything is an
2. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending & receiving *messages*. A message is a request for an action bundled with whatever arguments may be necessary to complete the task.
3. Each object has its own *memory*, which consists of other objects.
4. Every Object is an *instance* of class. A class simply represents a grouping of similar objects, such as integers or lists.
5. The class is the repository for *behavior* associated with an object. That is all objects that are instances of same class can perform the same actions.
6. Classes are organized into a singly rooted tree structure, called *inheritance hierarchy*.

Object Oriented Programming is a paradigm that provides many concepts such as **inheritance**, **data binding**, **polymorphism** etc.

Simula is considered as the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as truly object-oriented programming language.

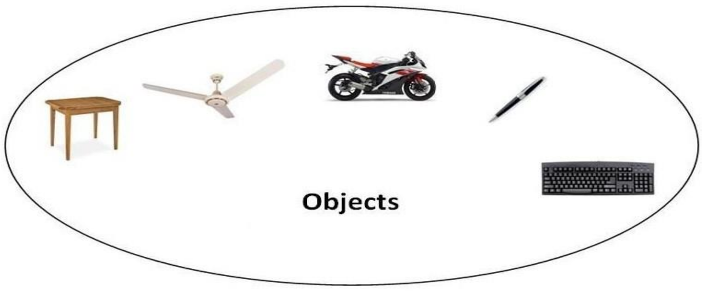
Smalltalk is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object

- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation



Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In java, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

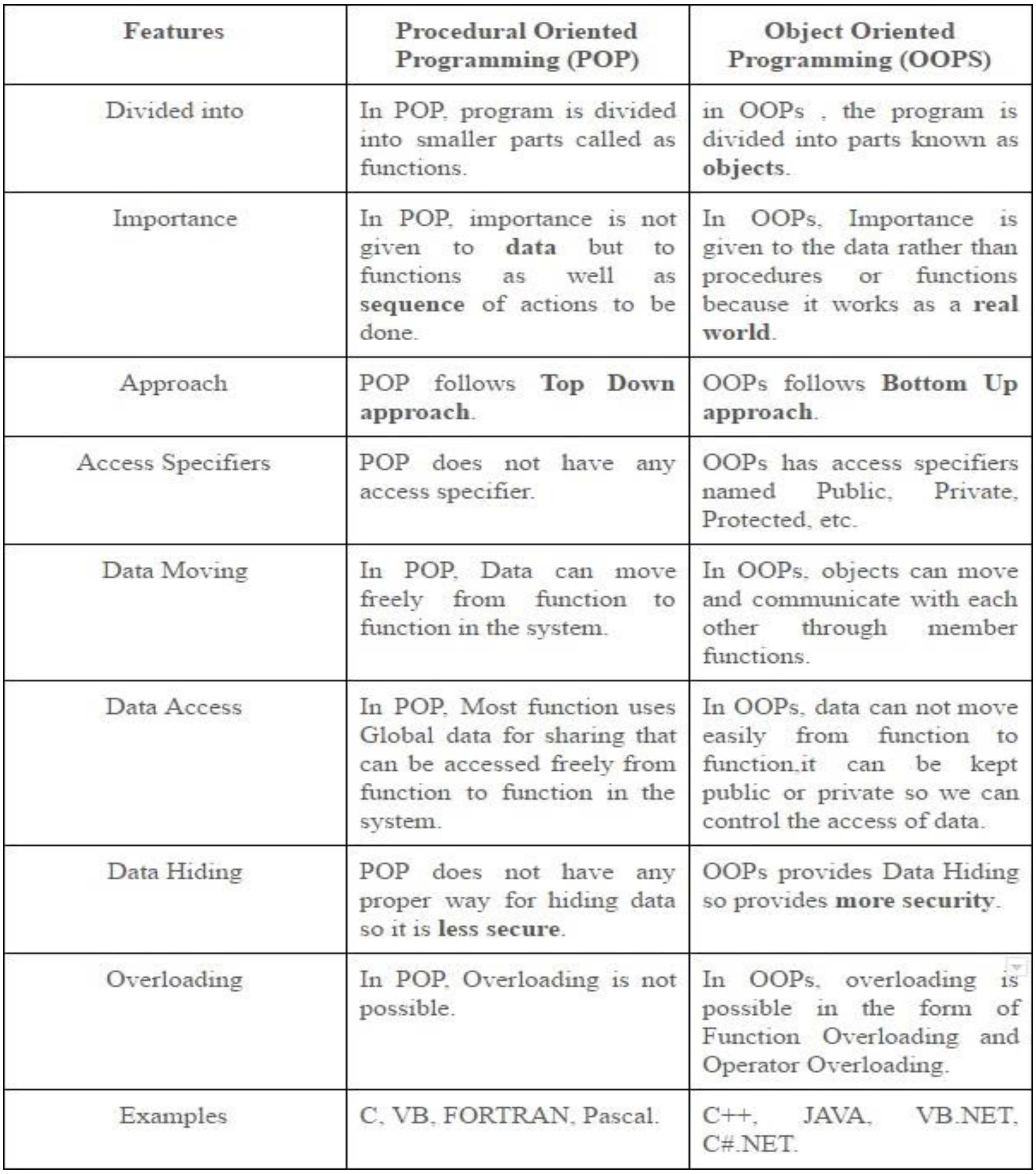
A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Benefits of Inheritance

- One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses. Where equivalent code exists in two related classes, the hierarchy can usually be refactored to move the common code up to a mutual superclass. This also tends to result in a better organization of code and smaller, simpler compilation units.
    - Inheritance can also make application code more flexible to change because classes that inherit from a common superclass can be used interchangeably. If the return type of a method is superclass
    - **Reusability** - facility to use public methods of base class without rewriting the same.
    - **Extensibility** - extending the base class logic as per business logic of the derived class.

- **Data hiding -** base class can decide to keep some data private so that it cannot be altered by the derived class

Procedural and object oriented programming paradigms



Features	Procedural Oriented Programming (POP)	Object Oriented Programming (OOPS)
Divided into	In POP, program is divided into smaller parts called as functions.	in OOPs , the program is divided into parts known as **objects**.
Importance	In POP, importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOPs, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOPs follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOPs has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOPs, objects can move and communicate with each other through member functions.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOPs, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOPs provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOPs, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	C, VB, FORTRAN, Pascal.	C++, JAVA, VB.NET, C#.NET.

Java Programming- History of Java

The history of java starts from Green Team. 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.

For the green team members, it was an advance concept at that time. But, it was 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.

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

2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling and file extension was .gt.

4) After that, it was called Oak and was developed as a part of the Green project.

Java Version History

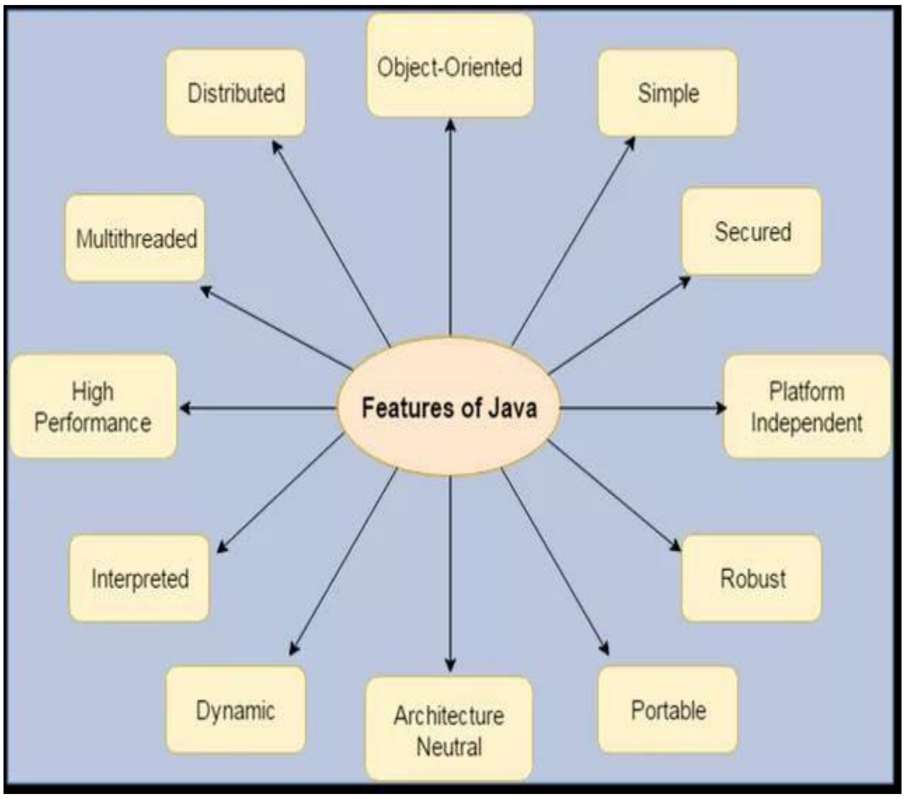
There are many java versions that has been released. Current stable release of Java is Java SE 8.

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)

Features of Java(buzzwords)

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

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



Java Comments

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

Types of Java Comments

There are 3 types of comments in java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

```
1.      //This is single line comment
```

Example:

```
public class CommentExample1 {
public static void main(String[] args) {
   int i=10;//Here, i is a variable
   System.out.println(i);
}
}
```

Output:

```
10
```

Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

```
/*
This
is
multi line
comment
*/
```

Example:

```
public class CommentExample2 {
public static void main(String[] args) {
/* Let's declare and
 print variable in java. */
   int i=10;
   System.out.println(i);
} }
```

Output:

```
10
```

Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool**.

Syntax:

```
/**
This
is
documentation
comment
*/
```

Example:

```
/** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*/
public class Calculator {
/** The add() method returns addition of given numbers.*/
public static int add(int a, int b){return a+b;}
/** The sub() method returns subtraction of given numbers.*/
public static int sub(int a, int b){return a-b;}
}
```

Compile it by javac tool:

```
javac Calculator.java
```

Create Documentation API by javadoc tool:

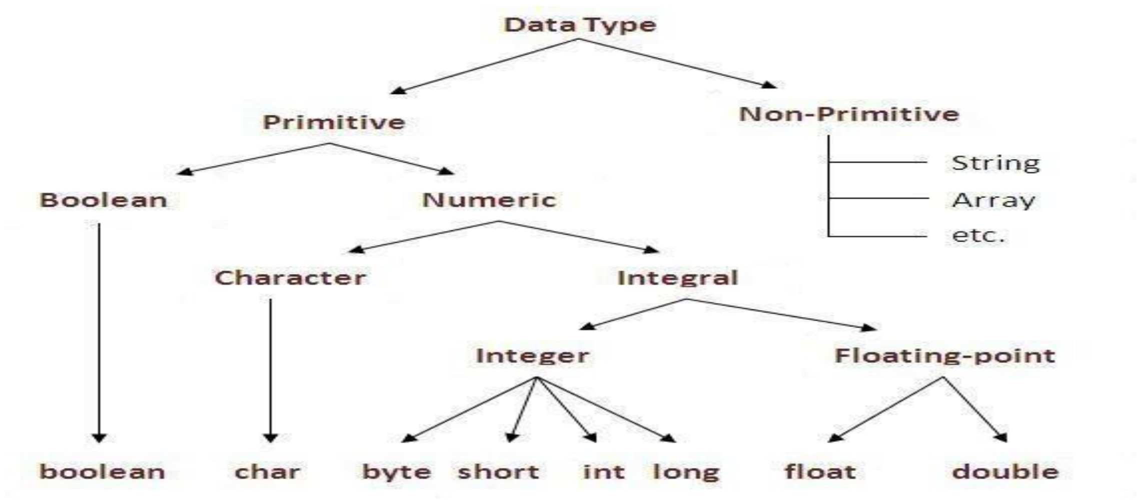
```
javadoc Calculator.java
```

Now, there will be HTML files created for your Calculator class in the current directory. Open the HTML files and see the explanation of Calculator class provided through documentation comment.

Data Types

Data types represent the different values to be stored in the variable. In java, there are two types of data types:

- Primitive data types
- Non-primitive data types



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Java Variable Example: Add Two Numbers

```
class Simple{
public static void main(String[] args){
int   a=10;
int   b=10;
int c=a+b;
System.out.println(c);
}}
```

Output:20

Variables and Data Types in Java

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in java: primitive and non-primitive.

Types of Variable

There are three types of variables in java:

- local variable
- instance variable
- static variable

1) Local Variable

A variable which is declared inside the method is called local variable.

2) Instance Variable

A variable which is declared inside the class but outside the method, is called instance variable . It is not declared as static.

3) Static variable

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

We will have detailed learning of these variables in next chapters.

Example to understand the types of variables in java

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

Constants in Java

A constant is a variable which cannot have its value changed after declaration. It uses the **'final' keyword.**

Syntax

modifier **final** dataType variableName = value; *//global constant*

modifier **static final** dataType variableName = value; *//constant within a c*

Scope and Life Time of Variables

The scope of a variable defines the section of the code in which the variable is visible. As a general rule, variables that are defined within a block are not accessible outside that block. The lifetime of a variable refers to how long the variable exists before it is destroyed. Destroying variables refers to deallocating the memory that was allotted to the variables when declaring it. We have written a few classes till now. You might have observed that not all variables are the same. The ones declared in the body of a method were different from those that were declared in the class itself. There are three types of variables: instance variables, formal parameters or local variables and local variables.

Instance variables

Instance variables are those that are defined within a class itself and not in any method or constructor of the class. They are known as instance variables because every instance of the class (object) contains a copy of these variables. The scope of instance variables is determined by the access specifier that is applied to these variables. We have already seen about it earlier. The lifetime of these variables is the same as the lifetime of the object to which it belongs. Object once created do not exist for ever. They are destroyed by the garbage collector of Java when there are no more reference to that object. We shall see about Java's automatic garbage collector later on.

Argument variables

These are the variables that are defined in the header oaf constructor or a method. The scope of these variables is the method or constructor in which they are defined. The lifetime is limited to the time for which the method keeps executing. Once the method finishes execution, these variables are destroyed.

Local variables

A local variable is the one that is declared within a method or a constructor (not in the header). The scope and lifetime are limited to the method itself.

One important distinction between these three types of variables is that access specifiers can be applied to instance variables only and not to argument or local variables.

In addition to the local variables defined in a method, we also have variables that are defined in bocks life an if block and an else block. The scope and is the same as that of the block itself.

Arrays

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables:

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

```
dataType[] arrayRefVar; // preferred way.
or
dataType arrayRefVar[]; // works but not preferred way.
```

Note: The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example:

The following code snippets are examples of this syntax:

```
double[] myList;        // preferred way.
or
double myList[];        // works but not preferred way.
```

Creating Arrays:

You can create an array by using the new operator with the following syntax:

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things:

- It creates an array using new dataType[arraySize];
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively you can create arrays as follows:

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example:

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:

```
double[] myList = new double[10];
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.

Processing Arrays:

When processing array elements, we often use either for loop or for each loop because all of the elements in an array are of the same type and the size of the array is known.

Example:

Here is a complete example of showing how to create, initialize and process arrays:

```
public class TestArray
{
  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};
           // Print all the array elements
    for (int i = 0; i < myList.length; i++) {
      System.out.println(myList[i] + " ");
    }
    // Summing all elements
    double total = 0;
    for (int i = 0; i < myList.length; i++) {
      total += myList[i];
    }
    System.out.println("Total is " + total);
    // Finding the largest element
    double max = myList[0];
    for (int i = 1; i < myList.length; i++) {
      if (myList[i] > max) max = myList[i];
    }
    System.out.println("Max is " + max);
  }
}
```

This would produce the following result:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

```
public class TestArray {
public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};
 // Print all the array elements
    for (double element: myList) {
      System.out.println(element);
}}}
```

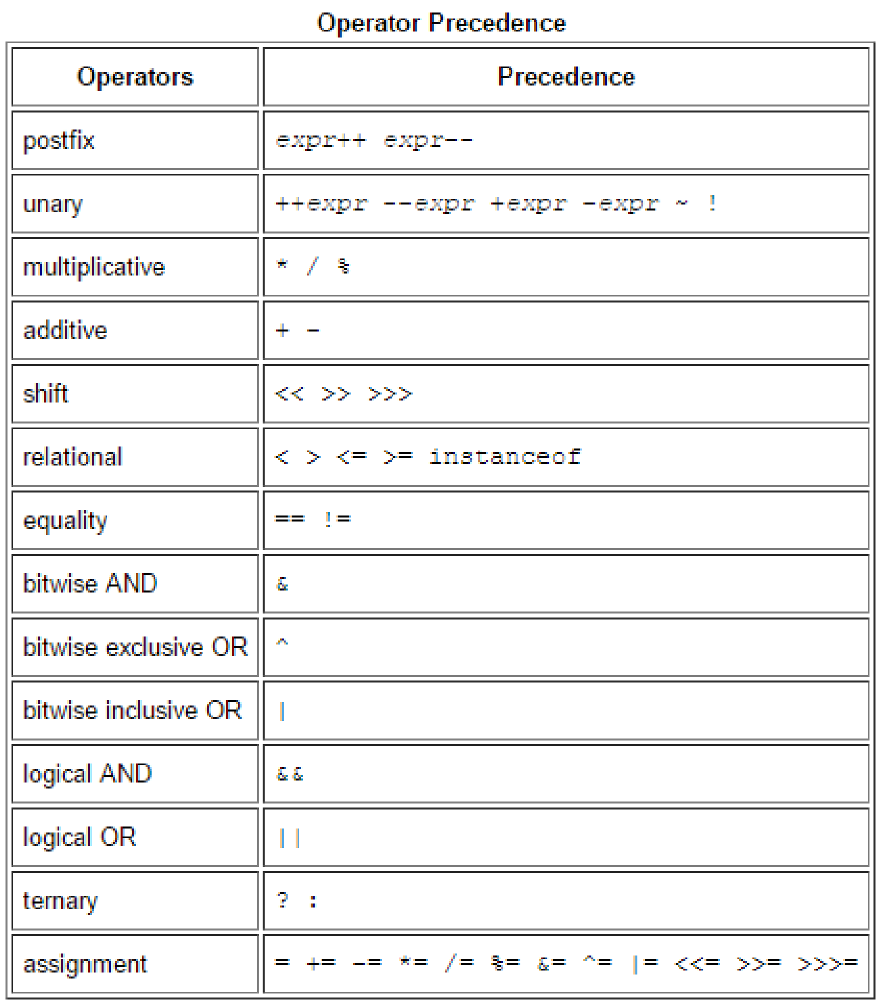
Operators in java

Operator in java is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in java which are given below:

- Unary Operator,
- Arithmetic Operator,
- shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Operators Hierarchy



Operator Precedence

Operators	Precedence
postfix	`expr++ expr--`
unary	`++expr --expr +expr -expr ~ !`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
ternary	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Expressions

Expressions are essential building blocks of any Java program, usually created to produce a new value, although sometimes an expression simply assigns a value to a variable. Expressions are built using values, variables, operators and method calls.

Types of Expressions

While an expression frequently produces a result, it doesn't always. There are three types of expressions in Java:

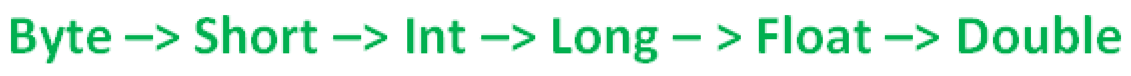
- Those that produce a value, i.e. the result of (1 + 1)
- Those that assign a variable, for example (v = 10)
- Those that have no result but might have a "side effect" because an expression can include a wide range of elements such as method invocations or increment operators that modify the state (i.e. memory) of a program.

Java Type casting and Type conversion

Widening or Automatic Type Conversion
Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign value of a smaller data type to a bigger data type.

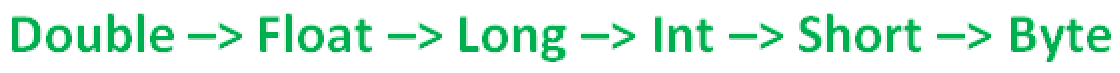
For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.


Byte –> Short –> Int –> Long – > Float –> Double

Widening or Automatic Conversion

Narrowing or Explicit Conversion
If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.


Double –> Float –> Long –> Int –> Short –> Byte

- Narrowing or Explicit Conversion

Java Enum

Enum in java is a data type that contains fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The java enum constants are static and final implicitly. It is available from JDK 1.5.

Java Enums can be thought of as classes that have fixed set of constants.

Simple example of java enum

```
class EnumExample1{
public enum Season { WINTER, SPRING, SUMMER, FALL }

public static void main(String[] args) {
for (Season s : Season.values())
System.out.println(s);
}}
```

Output:

```
WINTER
SPRING
SUMMER
FALL
```

Control Flow Statements

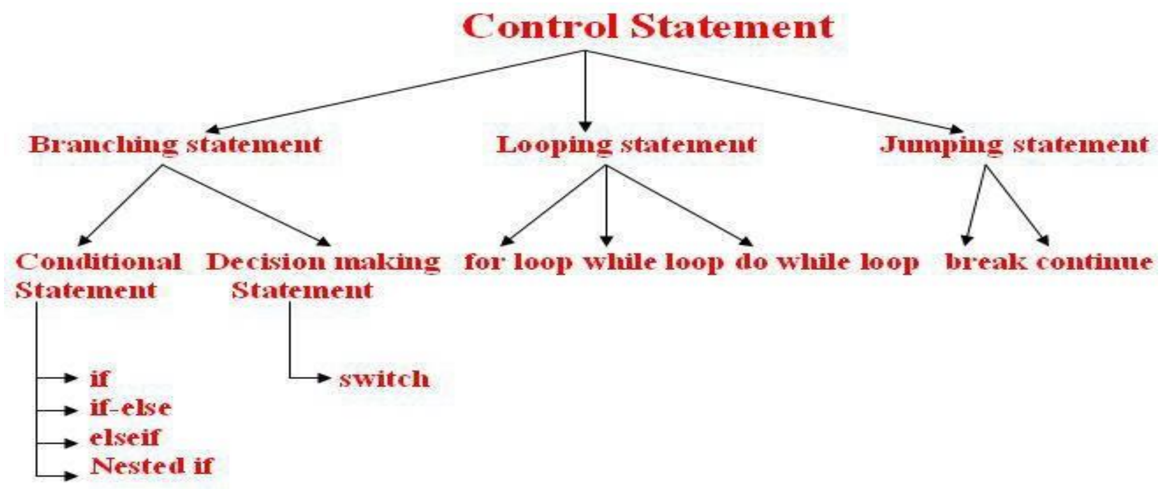
The control flow statements in Java allow you to run or skip blocks of code when special conditions are met.

The "if" Statement

The "if" statement in Java works exactly like in most programming languages. With the help of "if" you can choose to execute a specific block of code when a predefined condition is met. The structure of the "if" statement in Java looks like this:

```
if (condition) {
// execute this code
}
```

The condition is Boolean. Boolean means it may be true or false. For example you may put a mathematical equation as condition. Look at this full example:



Creating a Stand-Alone Java Application

1. Write a main method that runs your program. You can write this method anywhere. In this example, I'll write my main method in a class called Main that has no other methods. **For example:**

```
2.    public class Main
3.    {
4.      public static void main(String[] args)
5.      {
6.        Game.play();
7.      } }
```

8. Make sure your code is compiled, and that you have tested it thoroughly.

9. If you're using Windows, you will need to set your path to include Java, if you haven't done so already. This is a delicate operation. Open Explorer, and look inside C:\ProgramFiles\Java, and you should see some version of the JDK. Open this folder, and then open the bin folder. Select the complete path from the top of the Explorer window, and press Ctrl-C to copy it.

Next, find the "My Computer" icon (on your Start menu or desktop), right-click it, and select properties. Click on the Advanced tab, and then click on the Environment variables button. Look at the variables listed for all users, and click on the Path variable. Do not delete the contents of this variable! Instead, edit the contents by moving the cursor to the right end, entering a semicolon (;), and pressing Ctrl-V to paste the path you copied earlier. Then go ahead and save your changes. (If you have any Cmd windows open, you will need to close them.)

10. If you're using Windows, go to the Start menu and type "cmd" to run a program that brings up a command prompt window. If you're using a Mac or Linux machine, run the Terminal program to bring up a command prompt.

11. In Windows, type dir at the command prompt to list the contents of the current directory. On a Mac or Linux machine, type ls to do this.

12. Now we want to change to the directory/folder that contains your compiled code. Look at the listing of sub-directories within this directory, and identify which one contains your code. Type cd followed by the name of that directory, to change to that directory. For example, to change to a directory called Desktop, you would type:

cd Desktop

To change to the parent directory, type:

cd ..

Every time you change to a new directory, list the contents of that directory to see where to go next. Continue listing and changing directories until you reach the directory that contains your .class files.

13. If you compiled your program using Java 1.6, but plan to run it on a Mac, you'll need to recompile your code from the command line, by typing:

```
javac -target 1.5 *.java
```

14. Now we'll create a single JAR file containing all of the files needed to run your program.

Java Console Class

The Java Console class is be used to get input from console. It provides methods to read texts and passwords.

If you read password using Console class, it will not be displayed to the user.

The java.io.Console class is attached with system console internally. The Console class is introduced since 1.5.

Let's see a simple example to read text from console.

```
1.    String text=System.console().readLine();
2.    System.out.println("Text is: "+text);
```

Java Console Example

```
import java.io.Console;
class ReadStringTest{
public static void main(String args[]){
Console c=System.console();
System.out.println("Enter your name: ");
String n=c.readLine();
System.out.println("Welcome "+n);    }    }
```

Output

```
Enter your name: Nakul Jain
Welcome Nakul Jain
```

Java - Methods

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.**println()** method, for example, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, and apply method abstraction in the program design.

Creating Method

Considering the following example to explain the syntax of a method −

- **public static** − modifier

- **int** − return type
- **methodName** − name of the method
- **a, b** − formal parameters
- **int a, int b** − list of parameters

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

Syntax

```
modifier returnType nameOfMethod (Parameter List) {
   // method body
}
```

The syntax shown above includes −

- **modifier** − It defines the access type of the method and it is optional to use.
- **returnType** − Method may return a value.
- **nameOfMethod** − This is the method name. The method signature consists of the method name and the parameter list.

  Parameter List − The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body** − The method body defines what the method does with the statements.

Call by Value and Call by Reference in Java

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

Example of call by value in java

In case of call by value original value is not changed. Let's take a simple example:

```
class Operation{
 int data=50;
 void change(int data){
 data=data+100;//changes will be in the local variable only
 }
 public static void main(String args[]){
   Operation op=new Operation();
   System.out.println("before change "+op.data);
```

```
    op.change(500);
    System.out.println("after change "+op.data);
  }
}
```

```
Output:before change 50
       after change 50
```

In Java, parameters are always passed by value. For example, following program prints
i = 10, j = 20.

```
// Test.java
class Test {
   // swap() doesn't swap i and j
   public static void swap(Integer i, Integer j) {
      Integer temp = new Integer(i);
      i = j;
      j = temp;
   }
   public static void main(String[] args) {
      Integer i = new Integer(10);
      Integer j = new Integer(20);
      swap(i, j);
      System.out.println("i = " + i + ", j = " + j);
      }
}
```

Static Fields and Methods

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 instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

Java static variable

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

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

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

Understanding problem without static variable

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

Example of static variable

```
//Program of static variable
class Student8{
   int rollno;
   String name;
   static String college ="ITS";
   Student8(int r,String n){
   rollno = r;
   name = n;
   }
 void display (){System.out.println(rollno+" "+name+" "+college);}
 public static void main(String args[]){
 Student8 s1 = new Student8(111,"Karan");
 Student8 s2 = new Student8(222,"Aryan");

 s1.display();
 s2.display();
 } }
```

```
Output:111 Karan ITS
       222 Aryan ITS
```

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

Example of static method

```
//Program of changing the common property of all objects(static field).

class Student9{
    int rollno;
    String name;
    static String college = "ITS";
    static void change(){
    college = "BBDIT";
    }
    Student9(int r, String n){
    rollno = r;
    name = n;
```

```
 }
 void display (){System.out.println(rollno+" "+name+" "+college);}
public static void main(String args[]){
Student9.change();
Student9 s1 = new Student9 (111,"Karan");
Student9 s2 = new Student9 (222,"Aryan");
Student9 s3 = new Student9 (333,"Sonoo");
s1.display();
s2.display();
s3.display();
} }
```

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

Java static block

- Is used to initialize the static data member.
- It is executed before main method at the time of class loading.

Example of static block

```
class A2{
  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
```

“this” keyword in java

Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.

4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit 5000
112 sumit 6000
```

Java String

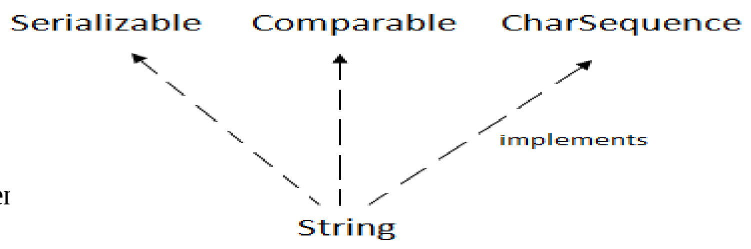
string is basically an object that represents sequence of char values. An array of characters works same as java string. For example:

```
1. char[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=new String(ch);
```

ssame as:

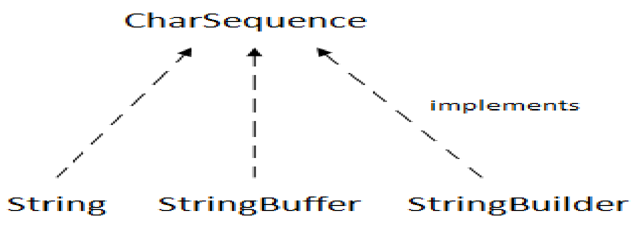
```
1. String s="javatpoint";
```

2. **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.
3. The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



CharSequence Interface

The CharSequence interface is used to represent sequence of characters. It is implemented by String, StringBuffer and StringBuilder classes. It means, we can create string in java by using these 3 classes.



The java String is immutable i.e. it cannot be changed. Whenever we change any string, a new instance is created. For mutable string, you can use StringBuffer and StringBuilder classes.

There are two ways to create String object:

1. By string literal
2. By new keyword

String Literal

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

```
1. 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:

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

By new keyword

```
1. 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);
}}
```

java
strings
example

Immutable String in Java

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

Let's try to understand the immutability concept by the example given below:

```
class Testimmutablestring{
  public static void main(String args[]){
    String s="Sachin";
    s.concat(" Tendulkar");//concat() method appends the string at the end
    System.out.println(s);//will print Sachin because strings are immutable objects
  }  }
Output:Sachin
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Sachin";
   s=s.concat(" Tendulkar");
   System.out.println(s);
 } } Output:Sachin Tendulkar
```

Method Overloading in java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Method Overloading: changing no. of arguments

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

In this example, we are creating static methods so that we don't need to create instance for calling methods.

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

Output:

```
22
33
```

Method Overloading: changing data type of arguments

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

Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

Java Recursion Example 1: Factorial Number

```
public class RecursionExample3 {
  static int factorial(int n){
   if (n == 1)
     return 1;
     else
    return(n * factorial(n-1));
    } }
public    static    void    main(String[]    args)    {
System.out.println("Factorial of 5 is: "+factorial(5));
} }
```

Output:

```
Factorial of 5 is: 120
```

Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){}
```

Simple Example of garbage collection in java

```
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
 public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 } }
```

```
object is garbage collected
object is garbage collected
```

Inheritance in Java

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object. Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

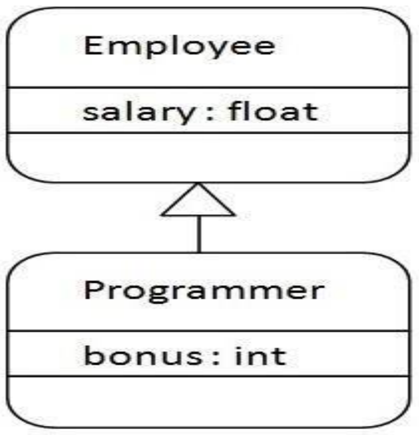
Why use inheritance in java

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

Syntax of Java Inheritance

```
1. class Subclass-name extends Superclass-name
2. {
3.    //methods and fields
4. }
```

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



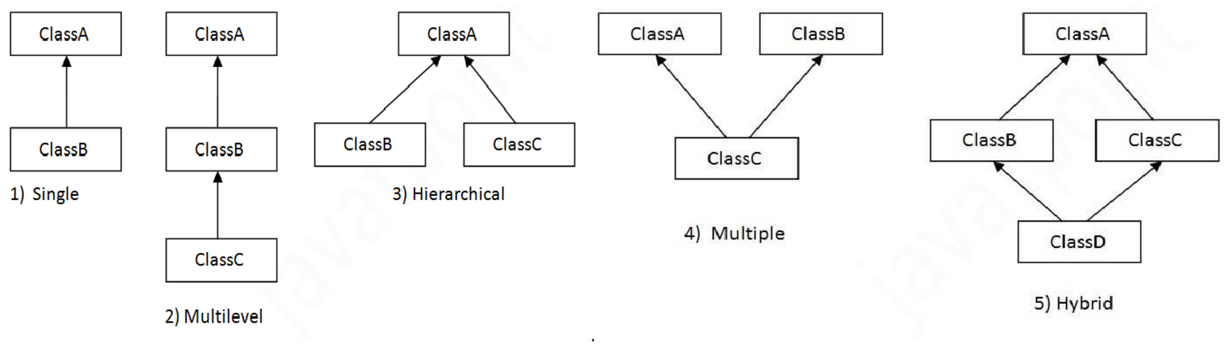
```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
} }
```

OUTPUT:

```
 Programmer salary is:40000.0

Bonus of programmer is:10000
```

Types of inheritance in java



Single Inheritance Example

File: TestInheritance.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

File: TestInheritance2.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
```

```
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
              at();
}}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

File: TestInheritance3.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

Member access and Inheritance

A subclass includes all of the members of its super class but it cannot access those members of the super class that have been declared as private. Attempt to access a private variable would cause compilation error as it causes access violation. The variables declared as private, is only accessible by other members of its own class. Subclass have no access to it.

CONSTUCTORS

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other start-up procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Syntax

Following is the syntax of a constructor

```
class ClassName {
  ClassName() {
  }
}
```

Java allows two types of constructors namely −

- No argument Constructors
- Parameterized Constructors

No argument Constructors

As the name specifies the no argument constructors of Java does not accept any parameters instead, using these constructors the instance variables of a method will be initialized with fixed values for all objects.

Example

```
Public class MyClass {
  Int num;
  MyClass() {
    num = 100;
  }
}
```

You would call constructor to initialize objects as follows

```
public class ConsDemo {
  public static void main(String args[]) {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass();
    System.out.println(t1.num + " " + t2.num);
  }
}
```

This would produce the following result

100 100

Parameterized Constructors

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Example

Here is a simple example that uses a constructor −

```
// A simple constructor.
class MyClass {
  int x;

  // Following is the constructor
  MyClass(int i ) {
    x = i;
  }
}
```

You would call constructor to initialize objects as follows −

```
public class ConsDemo {
  public static void main(String args[]) {
    MyClass t1 = new MyClass( 10 );
    MyClass t2 = new MyClass( 20 );
    System.out.println(t1.x + " " + t2.x);
  }
}
```

This would produce the following result −

```
10 20
```

Constructor Overloading in Java

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
class Student5{
int id;
String name;
int age;
Student5(int i,String n){
id = i;
```

```
        name = n;
        }
        Student5(int i,String n,int a){
        id = i;
        name = n;
        age=a;
        }
        void display(){System.out.println(id+" "+name+" "+age);}

        public static void main(String args[]){
        Student5 s1 = new Student5(111,"Karan");
        Student5 s2 = new Student5(222,"Aryan",25);
        s1.display();
        s2.display();
       }
    }
```

Output:

```
111 Karan 0
222 Aryan 25
```

Creating a Multilevel Inheritance Hierarchy in Java

Inheritance involves an object acquiring the properties and behaviour of another object. So basically, using inheritance can extend the functionality of the class by creating a new class that builds on the previous class by inheriting it.

Multilevel inheritance is when a class inherits a class which inherits another class. An example of this is class C inherits class B and class B in turn inherits class A.

A program that demonstrates a multilevel inheritance hierarchy in Java is given as follows:

Example

```
class A {
  void funcA() {
    System.out.println("This is class A");
  }
}
class B extends A {
  void funcB() {
    System.out.println("This is class B");
  }
}
class C extends B {
  void funcC() {
    System.out.println("This is class C");
  }
}
public class Demo {
  public static void main(String args[]) {
    C obj = new C();
    obj.funcA();
    obj.funcB();
    obj.funcC();
  }
}
```

Output

```
This is class A
This is class B
This is class C
```

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.

super is used to refer immediate parent class instance variable.

```
class Animal{
```

```
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();

d.printColor();
}}
```

Output:

```
black
white
```

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:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

Polymorphism :

The polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.

In java, polymorphism implemented using method overloading and method overriding.

Ad hoc polymorphism :

The ad hoc polymorphism is a technique used to define the same method with different implementations and different arguments. In a java programming language, ad hoc polymorphism carried out with a method overloading concept.

In ad hoc polymorphism the method binding happens at the time of compilation. Ad hoc polymorphism is also known as compile-time polymorphism. Every function call binded with the respective overloaded method based on the arguments.

The ad hoc polymorphism implemented within the class only.

Let's look at the following example java code.

Example

```
import java.util.Arrays;

public class AdHocPolymorphismExample {

        void sorting(int[] list) {
                Arrays.parallelSort(list);
                System.out.println("Integers after sort: " + Arrays.toString(list) );
        }
        void sorting(String[] names) {
                Arrays.parallelSort(names);
                System.out.println("Names after sort: " + Arrays.toString(names) );
        }

        public static void main(String[] args) {

                AdHocPolymorphismExample obj = new AdHocPolymorphismExample();
                int list[] = {2, 3, 1, 5, 4};
                obj.sorting(list);    // Calling with integer array

                String[] names = {"rama", "raja", "shyam", "seeta"};
                obj.sorting(names);           // Calling with String array
        }
}
```

Pure polymorphism :

The pure polymorphism is a technique used to define the same method with the same arguments but different implementations. In a java programming language, pure polymorphism carried out with a method overriding concept.

In pure polymorphism, the method binding happens at run time. Pure polymorphism is also known as run-time polymorphism. Every function call binding with the respective overridden method based on the object reference.

When a child class has a definition for a member function of the parent class, the parent class function is said to be overridden.

The pure polymorphism implemented in the inheritance concept only.

Let's look at the following example java code.

Example

```
class ParentClass{

        int num = 10;

        void showData() {
                System.out.println("Inside ParentClass showData() method");
                System.out.println("num = " + num);
        }

}

class ChildClass extends ParentClass{

        void showData() {
                System.out.println("Inside ChildClass showData() method");
                System.out.println("num = " + num);
        }
}

public class PurePolymorphism {

        public static void main(String[] args) {
                ParentClass obj = new ParentClass();
                obj.showData();

                obj = new ChildClass();
                obj.showData();

        }
}
```

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

Example of method overriding

```
Class Vehicle{
void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
void run(){System.out.println("Bike is running safely");}
public static void main(String args[]){
Bike2 obj = new Bike2();
obj.run();
}
```

Output:Bike is running safely

```
1. class Bank{
int getRateOfInterest(){return 0;}
}
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}
 class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
} }
```

```
Output:
SBI Rate of Interest: 8
```

```
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

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

```
1. abstract class A{}
```

abstract method

```
1. abstract void printStatus();//no body and abstract
```

Example of abstract class that has abstract method

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely..");}
public static void main(String
 args[]){ Bike obj = new Honda4();
 obj.run();
}
1. }
```

```
running safely..
```

Object class in Java

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

```
2. Object obj=getObject();//we don't know what object will be returned from this method
```

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Forms of Inheritance :

All objects eventually inherit from Object, which provides useful methods such as equals and toString.
In general we want to satisfy *substitutability:* if B is a subclass of A, anywhere we expect an instance of A we can use an instance of B.

Inheritance gets used for a number of purposes in typical object-oriented programming:

specialization -- the subclass is a special case of the parent class (e.g. Frame and CannonWorld)
specification -- the superclass just specifies which methods should be available but doesn't give code. This is supported in java by interfaces and abstract methods.
construction -- the superclass is just used to provide behavior, but instances of the subclass don't really act like the superclass. Violates substitutability. Exmample: defining Stack as a subclass of Vector. This is not clean -- better to define Stack as having a field that holds a vector.
extension -- subclass adds new methods, and perhaps redefines inherited ones as well.
limitation -- the subclass restricts the inherited behavior. Violates substitutability. Example: defining Queue as a subclass of Dequeue.
combination -- multiple inheritance. Provided in part by implementing multiple interfaces.

Inheritance is used in a variety of way and for a variety of different purposes .

- Inheritance for Specialization
- Inheritance for Specification
- Inheritance for Construction
- Inheritance for Extension
- Inheritance for Limitation
- Inheritance for Combination

One or many of these forms may occur in a single case.

Forms of Inheritance (- *Inheritance for Specialization* -)

Most commonly used inheritance and sub classification is for specialization.
Always creates a subtype, and the principles of substitutability is explicitly upheld.
It is the most ideal form of inheritance.

An example of subclassification for specialization is;

```
public class PinBallGame extends Frame {
// body of class
}
```

Specialization

- By far the most common form of inheritance is for specialization.
  - Child class is a specialized form of parent class
  - Principle of substitutability holds

- A good example is the Java hierarchy of Graphical components in the AWT:
  - Component
    - Label
    - Button
    - TextComponent
      - TextArea
      - TextField
    - CheckBox
    - ScrollBar

Forms of Inheritance (- *Inheritance for Specification* -)

This is another most common use of inheritance. Two different mechanisms are provided by Java, ***interface*** and ***abstract***, to make use of *subclassification for specification*. Subtype is formed and substitutability is explicitly upheld.
Mostly, not used for refinement of its parent class, but instead is used for definitions of the properties provided by its parent.

```
class FireButtonListener implements ActionListener {
// body of class
}
class B extends A {
// class A is defined as abstract specification class
}
```

Specification

- The next most common form of inheritance involves specification. The parent class specifies some behavior, but does not implement the behavior
  - Child class implements the behavior
  - Similar to Java interface or abstract class
  - When parent class does not implement actual behavior but merely defines the behavior that will be implemented in child classes
- Example, Java 1.1 Event Listeners:
  ActionListener, MouseListener, and so on specify behavior, but must be subclassed.

Forms of Inheritance (- *Inheritance for Construction* -)

Child class inherits most of its functionality from parent, but may change the name or parameters of methods inherited from parent class to form its interface.
This type of inheritance is also widely used for code reuse purposes. It simplifies the construction of newly formed abstraction but is not a form of subtype, and often violates substitutability.
Example is ***Stack*** class defined in Java libraries.

Construction

- The parent class is used only for its behavior, the child class has no *is-a* relationship to the parent.
  - Child modify the arguments or names of methods
- An example might be subclassing the idea of a *Set* from an existing *List* class.
  - Child class is not a more specialized form of parent class; no substitutability

Forms of Inheritance (- *Inheritance for Extension* -)

Subclassification for extension occurs when a child class only adds new behavior to the parent class and does not modify or alter any of the inherited attributes.
Such subclasses are always subtypes, and substitutability can be used.
Example of this type of inheritance is done in the definition of the class Properties which is an extension of the class HashTable.

Generalization or Extension

- The child class generalizes or extends the parent class by providing more functionality
  - In some sense, opposite of subclassing for specialization
- The child doesn't change anything inherited from the parent, it simply adds new features
  - Often used when we cannot modify existing base parent class
- Example, ColoredWindow inheriting from Window
  - Add additional data fields
  - Override window display methods

Forms of Inheritance (- *Inheritance for Limitation* -)

Subclassification for limitation occurs when the behavior of the subclass is smaller or more restrictive that the behavior of its parent class.
Like subclassification for extension, this form of inheritance occurs most frequently when a programmer is building on a base of existing classes.
Is not a subtype, and substitutability is not proper.

Limitation

- The child class limits some of the behavior of the parent class.
- Example, you have an existing List data type, and you want a Stack
- Inherit from List, but override the methods that allow access to elements other than top so as to produce errors.

Forms of Inheritance (- *Inheritance for Combination* -)
This types of inheritance is known as *multiple inheritance* in Object Oriented Programming.
Although the Java does not permit a subclass to be formed be inheritance from more than one parent class, several approximations to the concept are possible.

Example of this type is Hole class defined as;

```
class Hole extends Ball implements PinBallTarget{
// body of class
}
```

Combination

- Two or more classes that seem to be related, but its not clear who should be the parent and who should be the child.
- Example: Mouse and TouchPad and JoyStick
- Better solution, abstract out common parts to new parent class, and use subclassing for specialization.

Summary of Forms of Inheritance :

- Specialization. The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.
- Specification. The parent class defines behavior that is implemented in the child class but not in the parent class.
- Construction. The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class.
- Generalization. The child class modifies or overrides some of the methods of the parent class.
- Extension. The child class adds new functionality to the parent class, but does not change any inherited behavior.
- Limitation. The child class restricts the use of some of the behavior inherited from the parent class.
- Variance. The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary.
- Combination. The child class inherits features from more than one parent class. This is multiple inheritance and will be the subject of a later chapter.

The Benefits of Inheritance :

- Software Reusability (among projects)
- Increased Reliability (resulting from reuse and sharing of well-tested code)
- Code Sharing (within a project)
- Consistency of Interface (among related objects)
- Software Components
- Rapid Prototyping (quickly assemble from pre-existing components)
- Polymorphism and Frameworks (high-level reusable components)
- Information Hiding

The Costs of Inheritance :

- Execution Speed
- Program Size
- Message-Passing Overhead
- Program Complexity (in overuse of inheritance)