Difference Between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP)

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, 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**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

PRINCIPLES OF OOPS.

1. Encapsulation:

Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition. Typically, only the object's own methods can directly inspect or manipulate its fields.

Encapsulation is the hiding of data implementation by restricting access to accessors and mutators.

An accessor is a method that is used to ask an object about itself. In OOP, these are usually in the form of properties, which have a get method, which is an accessor method. However, accessor methods are not restricted to

properties and can be any public method that gives information about the state of the object.

A ***Mutator*** is public method that is used to modify the state of an object, while hiding the implementation of exactly how the data gets modified. It's the set method that lets the caller modify the member data behind the scenes.

Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state. This type of data protection and implementation protection is called Encapsulation.

A benefit of encapsulation is that it can reduce system complexity.

2. Abstraction

Data abstraction and encapuslation are closely tied together, because a simple definition of data abstraction is the development of classes, objects, types in terms of their interfaces and functionality, instead of their implementation details. Abstraction denotes a model, a view, or some other focused representation for an actual item.

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer." — G. Booch

In short, data abstraction is nothing more than the implementation of an object that contains the same essential properties and actions we can find in the original object we are representing.

3. Inheritance

Inheritance is a way to reuse code of existing objects, or to establish a subtype from an existing object, or both, depending upon programming language support. In classical inheritance where objects are defined by classes, classes can inherit attributes and behavior from pre-existing classes called base classes, superclasses, parent classes or ancestor classes. The resulting classes are known as derived classes, subclasses or child classes. The relationships of classes through inheritance gives rise to a hierarchy.

Subclasses and Superclasses A subclass is a modular, derivative class that inherits one or more properties from another class (called the superclass). The properties commonly include class data variables, properties, and methods or functions. The superclass establishes a

common interface and foundational functionality, which specialized subclasses can inherit, modify, and supplement. The software inherited by a subclass is considered reused in the subclass. In some cases, a subclass may customize or redefine a method inherited from the superclass. A superclass method which can be redefined in this way is called a virtual method.

4. Polymorphism

Polymorphism means one name, many forms. Polymorphism manifests itself by having multiple methods all with the same name, but slightly different functionality.

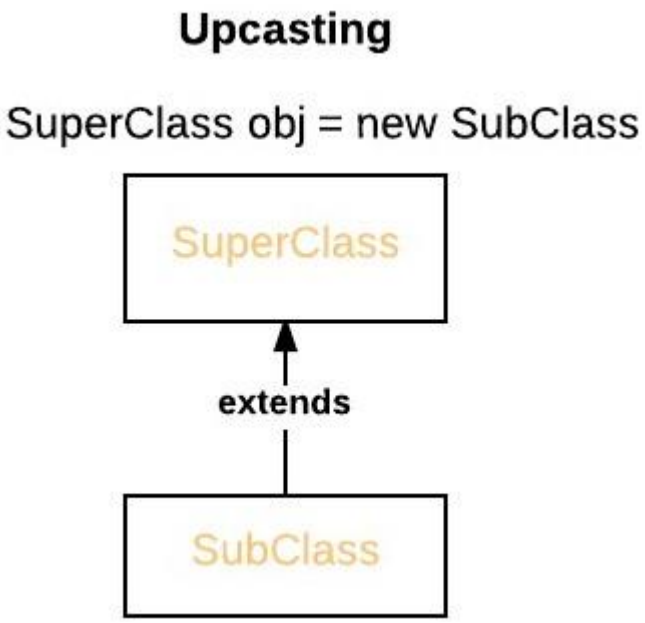
There are 2 basic types of polymorphism. Overridding, also called run-time polymorphism. For method overloading, the compiler determines which method will be executed, and this decision is made when the code gets compiled. Overloading, which is referred to as compile-time polymorphism. Method will be used for method overriding is determined at runtime based on the dynamic type of an object.

Dynamic Method Dispatch or Runtime Polymorphism in Java

Prerequisite: Overriding in java, Inheritance

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.



Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

```
// A Java program to illustrate Dynamic Method
// Dispatch using hierarchical inheritance
class A
{
    void m1()
    {
        System.out.println("Inside A's m1 method");
    }
}

class B extends A
{
    // overriding m1()
    void m1()
    {
        System.out.println("Inside B's m1 method");
    }
}

class C extends A
{
    // overriding m1()
    void m1()
    {
        System.out.println("Inside C's m1 method");
    }
}

// Driver class
class Dispatch
{
    public static void main(String args[])
    {
        // object of type A
        A a = new A();
```

```
        // object of type B
        B b = new B();

        // object of type C
        C c = new C();

        // obtain a reference of type A
        A ref;

        // ref refers to an A object
        ref = a;

        // calling A's version of m1()
        ref.m1();

        // now ref refers to a B object
        ref = b;

        // calling B's version of m1()
        ref.m1();

        // now ref refers to a C object
        ref = c;

        // calling C's version of m1()
        ref.m1();
    }
}
```

Output:

```
Inside A's m1 method

Inside B's m1 method

Inside C's m1 method
```

Explanation :

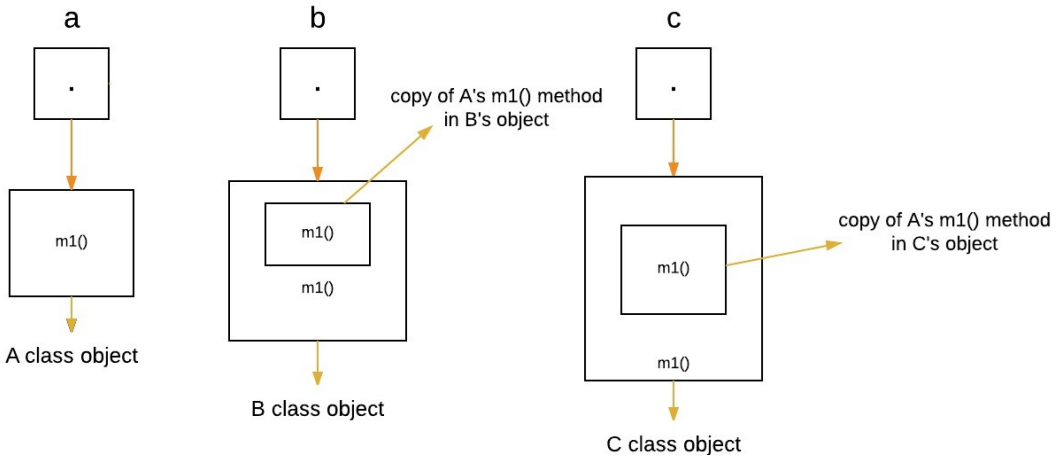
The above program creates one superclass called A and it's two subclasses B and C. These subclasses overrides m1( ) method.

1. Inside the main() method in Dispatch class, initially objects of type A, B, and C are declared.

```
2.  A a = new A(); // object of type A

3.  B b = new B(); // object of type B

4.  C c = new C(); // object of type C
```

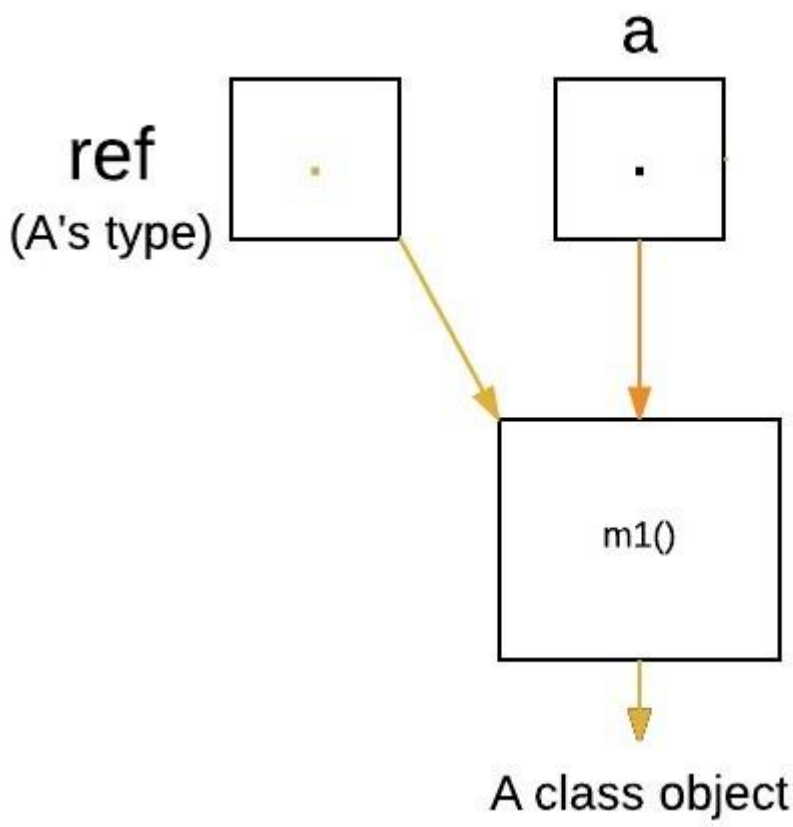


5. Now a reference of type A, called ref, is also declared, initially it will point to null.

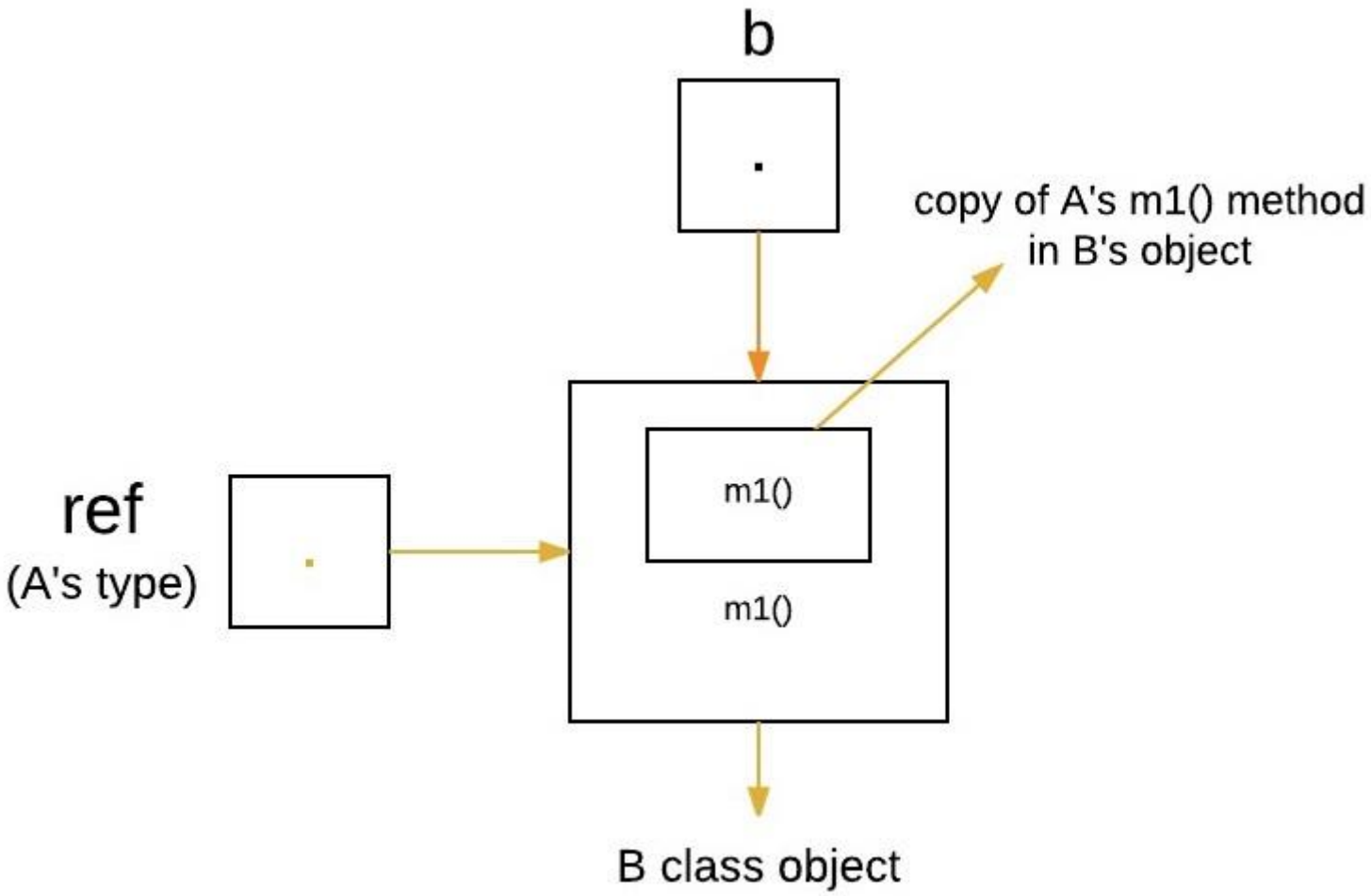
```
6. A ref; // obtain a reference of type A
```

7. Now we are assigning a reference to each **type of object** (either A's or B's or C's) to *ref*, one-by-one, and uses that reference to invoke m1( ). As the output shows, the version of m1( ) executed is determined **by the type of object being referred to at the time of the call.**

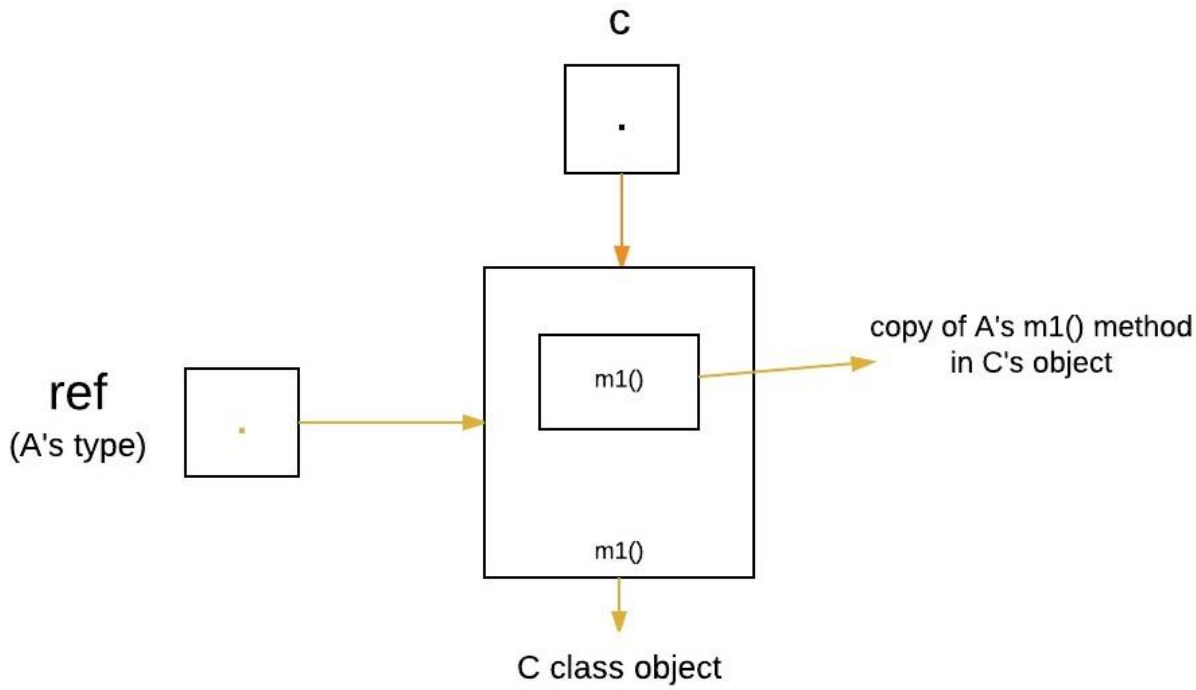
```
8. ref = a; // r refers to an A object
9. ref.m1(); // calling A's version of m1()
```




```
ref = b; // now r refers to a B object
ref.m1(); // calling B's version of m1()
```




```
ref = c; // now r refers to a C object
ref.m1(); // calling C's version of m1()
```



Runtime Polymorphism with Data Members

In Java, we can override methods only, not the variables(data members), so **runtime polymorphism cannot be achieved by data members.** For example :

```
// Java program to illustrate the fact that
// runtime polymorphism cannot be achieved
// by data members

// class A
class A
{
    int x = 10;
}

// class B
class B extends A
{
    int x = 20;
}

// Driver class
public class Test
{
    public static void main(String args[])
    {
        A a = new B(); // object of type B

        // Data member of class A will be accessed
        System.out.println(a.x);
    }
}
```

Output:

```
10
```

Explanation : In above program, both the class A(super class) and B(sub class) have a common variable 'x'. Now we make object of class B, referred by 'a' which is of type of class A. Since variables are not overridden, so the statement "a.x" will **always** refer to data member of super class.

Advantages of Dynamic Method Dispatch

1. Dynamic method dispatch allow Java to support overriding of methods which is central for run-time polymorphism.
2. It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
3. It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

Static vs Dynamic binding

- Static binding is done during compile-time while dynamic binding is done during run-time.
- private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

Difference between object and class

There are many differences between object and class. A list of differences between object and class are given below:

Object	Class
Object is an **instance** of a class.	Class is a **blueprint or template** from which objects are created.
Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a **group of similar objects**.
Object is a **physical** entity.	Class is a **logical** entity.
Object is created through **new keyword** mainly e.g. Student s1=new Student();	Class is declared using **class keyword** e.g. class Student{}
Object is created **many times** as per requirement.	Class is declared **once**.
Object **allocates memory when it is created**.	Class **doesn't allocated memory when it is created**.
There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only **one way to define class** in java using class keyword.

Difference between Encapsulation and Abstraction in OOPS

Difference between Encapsulation and Abstraction in OOPS

Abstraction and Encapsulation are two important Object Oriented Programming (OOPS) concepts. Encapsulation and Abstraction both are interrelated terms.

Real Life Difference Between Encapsulation and Abstraction

Encapsulate means to hide. Encapsulation is also called data hiding.You can think Encapsulation like a capsule (medicine tablet) which hides medicine inside it. Encapsulation is wrapping, just hiding properties and methods. Encapsulation is used for hide the code and data in a single unit to protect the data from the outside the world. Class is the best example of encapsulation.

Abstraction refers to showing only the necessary details to the intended user. As the name suggests, abstraction is the "abstract form of anything". We use abstraction in programming languages to make abstract class. Abstract class represents abstract view of methods and properties of class.

Implementation Difference Between Encapsulation and Abstraction

1. Abstraction is implemented using interface and abstract class while Encapsulation is implemented using private and protected access modifier.

2. OOPS makes use of encapsulation to enforce the integrity of a type (i.e. to make sure data is used in an appropriate manner) by preventing programmers from accessing data in a non-intended manner. Through encapsulation, only a predetermined group of functions can access the data. The collective term for datatypes and operations (methods) bundled together with access restrictions (public/private, etc.) is a class.

3. Example of Encapsulation

```
Class Encapsulation
{
   private int marks;

   public int Marks
  {
     get { return marks; }
     set { marks = value;}
   }
```

```
}
```

4. Example of Abstraction

```
abstract class Abstraction
{
    public abstract void doAbstraction();
}

public class AbstractionImpl: Abstraction
{
    public void doAbstraction()
   {
       //Implement it
   }
}
```

Comparison Chart

BASIS FOR COMPARISON	INHERITANCE	POLYMORPHISM
Basic	Inheritance is creating a new class using the properties of the already existing class.	Polymorphism is basically a common interface for multiple form.

BASIS FOR COMPARISON	INHERITANCE	POLYMORPHISM
Implementation	Inheritance is basically implemented on classes.	Polymorphism is basically implemented on function/methods.
Use	To support the concept of reusability in OOP and reduces the length of code.	Allows object to decide which form of the function to be invoked when, at compile time(overloading) as well as run time(overriding).
Forms	Inheritance may be a single inheritance, multiple inheritance, multilevel inheritance, hierarchical inheritance and hybrid inheritance.	Polymorphism may be a compile time polymorphism (overloading) or run-time polymorphism (overriding).
Example	The class 'table' can inherit the feature of the class 'furniture', as a 'table' is a 'furniture'.	The class 'study_table' can also have function 'set_color()' and a class 'Dining_table' can also have function 'set_color()' so, which

BASIS FOR COMPARISON	INHERITANCE	POLYMORPHISM
		form of the set_color() function to invoke can be decided at both, compile time and run time.

ByteCode is an intermediate code that the java compiler creates after compiling the java source code. This ByteCode is interpreted by the JRE(Java Runtime Environment) to run the underlined program.

Q. Why is JAVA PLATFROM INDEPENDENT?

JAVA is platform independent because once the java source code is written and compiled in a machine, it can be executed in any other type of machine provided the latter contains a JVM (JAVA Virtual Machine) for that machine. JAVA gives this feature by compiling the java source code into intermediate special language called bytecode which can be interpreted in any machine by using the JVM for that machine.

Machine code or **machine language** is a set of instructions executed directly by a computer's central processing unit (CPU).

Bytecode is platform-independent, bytecodes compiled by a compiler running in windows will still run in linux/unix/mac. Machine code is platform-specific, if it is compiled in windows, it will run ONLY in windows.

Portability- Unlike C,C++, there are no "**implementation-dependent**" aspects of the specification. The sizes of the primitive data types are specified, as is the behaviour of the arithmetic on them.

• Good Features of Java (JAMES GOSLING) → GOKUDEV.

① Simple → Programming in Java is simple in the sense that it needs very less training for the C/C++ programmers since its syntax is almost same as that of C/C++. Java omits many rarely used, poorly understood, confusing features of C/C++

② Object oriented → Java is a purely object oriented language.

③ Network Saavy → Java has an extensive library of routines for coping with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the Net via URLS with the same ease as when accessing a local file system.

④ ROBUST → Java puts a lot of emphasis on early checking for possible problems, later, dynamic (runtime) checking, and eliminating situations that are error-prone. The single biggest difference between JAVA and C/C++ is that JAVA has a pointer model that eliminates the possibility of overwriting memory and corrupting data.

⑤ SECURE → Java is intended to be used in networked / distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems.

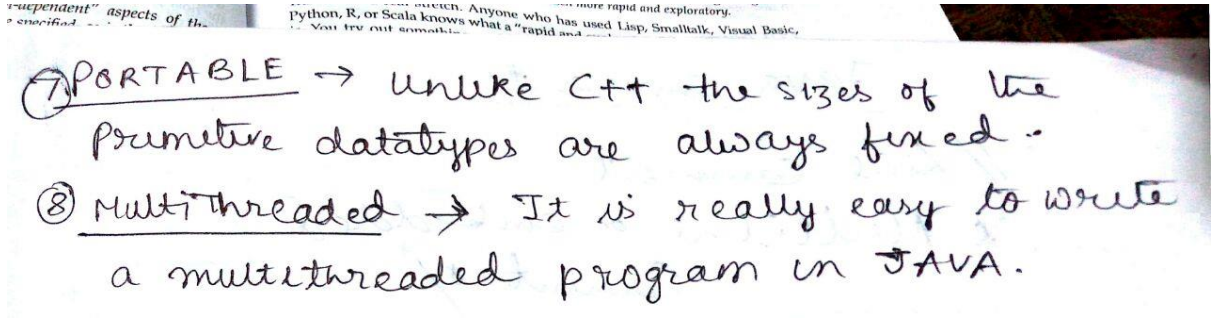
* ⑥ ARCHITECTURE NEUTRAL → The compiled code of JAVA is executable on many processors, given the presence of JAVA runtime system (JRS). The JAVA compiler does this by generating byte code instructions (Targetted to the JAVA Virtual Machine (JVM)) which have nothing to do with a particular computer architecture. Rather they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly.

The main motto of JAVA is (Write once, Run everywhere).

* JIT COMPILER

Just in Time Compiler

JRE also contains a JIT compiler that compiles certain modules or methods of JAVA program (byte code) as a whole when it is first occured and that compiled code can be executed directly on the next occurences.



(7) PORTABLE → Unlike C++ the sizes of the primitive datatypes are always fixed.

(8) Multi Threaded → It is really easy to write a multithreaded program in JAVA.

Interface

Interface is something like an abstract class that can only contain (not definition) some method declarations and some final variables. ~~public abstract~~ or

If a class implements an interface (by using the 'implements' keyword then it must ~~that~~ define (implement) all the method of the interface)

Comparing abstract class and interface

(1) ~~An~~ cannot instantiate

(2) ~~Both the~~ can create a variable of the type. that contains the reference of an instance of a class implementing the interface

(3) unlike ~~abstract class~~ inheritance, a class can implement more than one interface

(4) An interface can extend one or more interfaces

(5) The method declarations are always public, so the public keyword is not required

ation-depend...

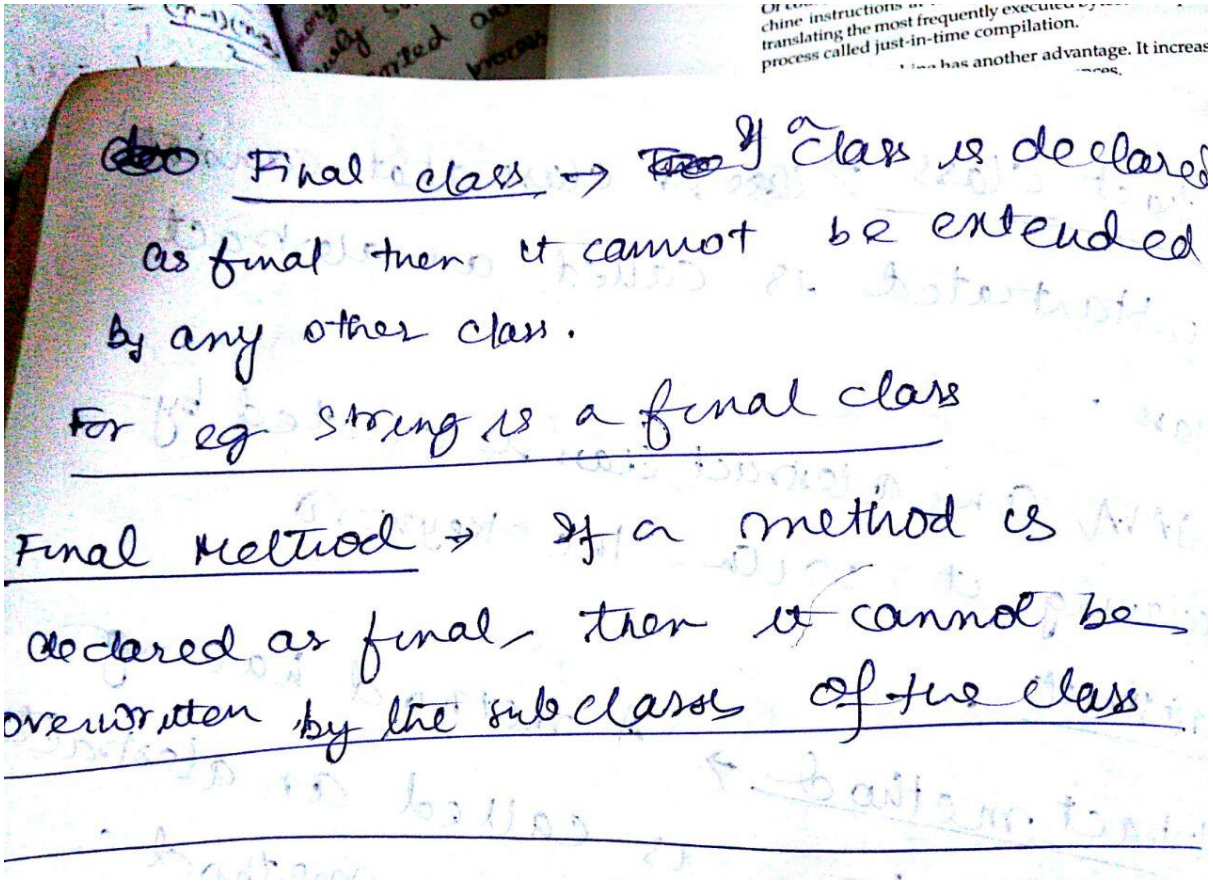
This seems a real stretch. Anyone who has used ... Python, R, or Scala knows what a ...

...ues directly on any machine to which ... linking is a more incremental and lightweight process can be much more rapid and exploratory.

... Visual Basic.

(6) Unlike abstract class, an interface can contain only method declarations and final variables, whereas an abstract class can contain no abstract method at all.

(7) The instance of keyword operator works as same as in case of inheritance



Final class → If a class is declared as final then it cannot be extended by any other class.

For eg String is a final class

Final Method → If a method is declared as final then it cannot be overwritten by the subclasses of the class

Abstract class → A class that cannot be instantiated is called an abstract class.

In JAVA, an abstract class is created by declaring it with the keyword abstract.

Abstract method → A method having no definition is called an abstract method. In JAVA, such a method is declared with a keyword abstract.

A class having atleast one abstract method must be declared as an abstract class.

Although a abstract class cannot be instantiated, a reference variable of such a class can be created which can store the reference of an instance of a class that extends the abstract class. Such a class that extends an abstract class must override all the abstract methods, if any, of its superclass or else it must also be declared as an abstract class

If a derived class's constructor does not call any of this superclass constructor, then no argument constructor of its superclass gets called automatically.

If the derived class's constructor wants to call one of its superclass constructor then it calls that by using the super keyword.

and that call must be the very statement in the constructor. Parameterised constructor should be called by super, doesn't gets called automatically

Instance OF

instance of is a boolean Keyword-operator. (both keyword and operator). returns boolean value.

It is used as

v instanceof C

The above expression returns true if the reference variable v contains the reference of an instance of a class C or of any of the derived classes of class C; otherwise returns false.

④ <default> ⇒ when nothing is specified

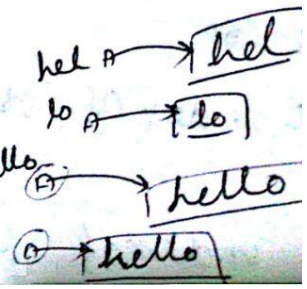
this keyword → This keyword can only be used within a non static method and it acts as a reference to the invoking instance.

In JAVA, unlike local variables all the member variables, if not initialised, get their default values (0 for numeric variable, false for boolean, null for reference variables).

GARBAGE COLLECTION IN JAVA.

The Java Runtime system has a special subsystem called Garbage Collector (GC) that periodically deallocates all the memory chunks that have been allocated but are not being referenced by any variable. So the Java programmer does not have to worry about any memory deallocation.

```
String hel = "hel";
String lo = "lo";
```



Java virtual machine (JVM)

A Java virtual machine (JVM), an implementation of the Java Virtual Machine Specification, interprets compiled Java binary code (called bytecode) for a computer's processor (or "hardware platform") so that it can perform a Java program's instructions. Java was designed to allow application programs to be built that could be run on any platform without having to be rewritten or recompiled by the programmer for each separate platform. A Java virtual machine makes this possible because it is aware of the specific instruction lengths and other particularities of the platform.

1)JVM converts the byte code into its equalent excutable code.
2)It loads the excuteable code into memory(RAM).
3)Excutes this code through local OS.
4)Deletes the excutable code from RAM

JAVA DOES NOT SUPPORT DESTRUCTORS- Java does not support destructor in the way that c++ provides. C++ has direct destrcors that we can write in our code and it will clear object once object goes out of scope. But that is not case in Java, you can't simply predict when object is going out of scope and when to destroy object, to do this java has concept called Garbage Collection. Garbage Collector takes care of clearing up memory allocated to object. It has finalize method which is entirely dicretion of Garbage Collector. So as in c++ 'delete' or 'free()' is used to release allocated memory in java that work is done by finalize and Garbage Collector.So since Java is Garbage Collector language it does not have destructor.

Java Variables

Java Variables or variables in any other programming language are containers, which hold some value. Because Java is a strongly typed language, so every variable must be declared and initialized or assigned before it is used. A variable, in the simplest way, is declared by placing a valid type followed by the variable name. And so, don't forget to place a semicolon at the end of every statement because it completes the statement. Throughout the tutorial we will use terms 'variable' and 'field' interchangeably as they refer to the same thing and there should be no confusion regarding that. Following statements demonstrate variable declaration in Java

```
byte age; //contains -128 to 127
long population;
float temperature;
double salary;
```

Java Variables - Naming Rules and Conventions

While naming Java variables we have to adhere to some naming rules. Such as:

1. Variable names are case-sensitive.
2. Variable name can be an unlimited-length sequence of Unicode letters and digits.
3. Variable name must begin with either a letter or the dollar sign "$", or the underscore character "_".
4. Characters except the dollar sign "$" and the underscore character "_" for example, '+' or '?' or '@' are invalid for a variable name.
5. No white space is permitted in variable names.
6. Variable name we choose must not be a keyword or reserved word.

Additional to above mentioned naming rules Java developers also adhere to some naming conventions for Java variables but these conventions are totally optional. So it is your wish if you follow them or not. However, following naming conventions is a great practice and it makes your program more readable and beautiful. Most prevalent naming conventions are as follows:

1. Always begin your Java variable names with a letter, not "$" or "_"
2. The dollar sign character, by convention, is never used at all in Java variables. There may be situations where auto-generated names will contain the dollar sign, but your variable names should always avoid using it
3. It's technically legal to begin your variable's name with "_" but this practice is discouraged
4. Choosing full words instead of cryptic abbreviations for variable names are always considered good, it makes your code easier to read and understand
5. If chosen Java variable name consists of only one word, use all lower-case letters to spell it. If it consists of more than one word, capitalize the first letter of each subsequent word. The names `firstChoice` and `reverseGear` are good examples of this convention. For constant valued variables, such as `static final int NUM_OF_CHOICES = 7`, the convention changes slightly, capitalizing all letter and separating subsequent words with the underscore character. By convention, the underscore character is not used elsewhere.

As it has been said, above conventions are not rules therefore they can be compromised when needed.

Java Variable Types

Java variables are categorized in different types depending upon where are they declared? For example, in a class or in a method's body. Java defines following four types of variables.

Local Variables

As name suggests, a local Java variable is scoped to the block, which is between the opening and closing braces. Local variables are declared within a block or constructor or method's body. Local variables are created when control enters into the block or constructor of method's body and are destroyed once the control exits from the body of method or constructor or block.

Local variables are only visible to the methods or blocks or constructors in which they are declared; they are not accessible from the rest of the class. There is no designated access modifier for local variables. Also, there is no default value for local variables, therefore, local variables must be initialized or assigned some value before they are used first time.

Instance Variables or Non-Static Fields

Data members or fields or variables of a class that are declared non-static, but outside a method, constructor or any block are known as *instance variables*. Instance variables are created when an object of a class is created by using `new` keyword. Objects store their states in non-static instance variables. Instance variables are declared private or public or protected or default (no keyword). Instance variables are initialized to some default values by the compiler, in case they are not initialized by the creator of class. Instance variables are destroyed along with the object they have been created for. The following program declares some non-static instance variables:

```
/* Book.java */
class Book
{
        //instance variables
        private String title;
        private String publisher;
        private int numOfPages;

        //zero argument constructor needed when
        //one or more argument constructors are defined
        public Book() {}

        public Book (String title, String publisher, int numOfPages)
        {
                this.title = title;
                this.publisher = publisher;
                this.numOfPages = numOfPages;
        }

        public void printBookDetails()
        {
                System.out.println("Title: " + title);
                System.out.println("Publisher: " + publisher);
                System.out.println("Num of Pages: " + numOfPages);
        }
}
```

Class Variables or Static Fields

Class variables in Java are fields declared with `static` keyword. Modifier `static`informs the compiler that there will be only one copy of such variables created regardless of how many objects of this class are created. Static variables are created when a class is loaded into the memory by the class loader and destroyed when a class is destroyed or unloaded. Visibility of static fields will depend upon access modifiers. Default vales given to the static members will follow the same rule as it is done in instance variables.

Method Parameters

While passing values to member methods we need parameter variables. Parameters are not fields; they are always called variables and used only to copy their values into the formal parameters. Variables or values passed to a method by caller are called actual parameters, and the callee receives these values in formal parameters. Formal parameters are local variables to that particular method.

Java Variables - Dynamic Initialization

Initialization is the process of providing value to a variable at declaration time. A variable is initialized once in its life time. Any attempt of setting a variable's value after its declaration is called assignment. To use a local variable you have to either initialize or assign it before the variable is first used. But for class members, the compulsion is not so strict. If you don't initialize them then compiler takes care of the initialization process and set class members to default values.

Java allows its programmers to initialize a variable at run time also. Initializing a variable at run time is called dynamic initialization. The following piece of code (`DynamicInitializationDemo.java`) demonstrates it.

```
/* DynamicInitializationDemo.java */
public class DynamicInitializationDemo
{
    public static void main(String[] args)
    {
        //dynSqrt will be initialized when Math.sqrt
        //will be executed at run time
        double dynSqrt = Math.sqrt (16);
        System.out.println("sqrt of 16 is : " + dynSqrt);
    }
}

OUTPUT
======
sqrt of 16 is : 4.0
```

Java Variables - Intermediate Type Promotion

In some situations intermediate result of an expression is out of range of the variable to which the final result has to be assigned, while the final result is in variable's range? Let's consider an example where we have to compute average age of three octogenarians. Suppose we store individual's and average age in `byte` variables. The ages of all three are 82, 87, and 89 years respectively. Now to compute the average of their age you first have to add all three ages together and then divide it by 3. But, if you see, as soon as you add all three ages you will get 258 that is beyond the range of `byte` (127 is the largest positive value a `byte` can store), while the final averaged age 86 that is in the range of `byte`.

Java handles this situation very smartly by promoting intermediate results to integer. But it imposes an extra overhead on programmers to cast results back to the resultant type, `byte` in our case. See the following piece of code.

```
/* TypePromotionDemo.java */
public class TypePromotionDemo
{
    public static void main(String[] args)
    {
        byte age1 = 82, age2 = 87, age3 = 89;
        byte avgAge = (byte) ((age1 + age2 + age3) / 3); //cast int to byte
        System.out.println("Average age is : " + avgAge);
    }
}

OUTPUT
======
Average age is : 86
```

In addition to the elevation of `byte` to `int`, Java defines several type promotion rules that apply to expressions. First, all `byte` and `short` values are promoted to `int`, as just described. Then, if one operand is a `long`, the whole expression is promoted to `long`. If one operand is a `float`, the entire expression is promoted to `float`. If any of the operands is `double`, the result is `double`.

Java Variables - Type Conversion and Type Casting

When value of one variable is assigned to another or a literal is assigned to a variable then either automatic type conversion takes place or we have to type cast explicitly. Automatic type conversion is done when either both sides (left and right) are of same type or the left hand (destination) side is larger in size. The latter is called *widening*. For example, assigning `byte` to `short` or `int` to `long`.

If we wish to assign incompatible types then we will have to type cast at our own and according to the problem for example, `int` to `short` conversion will not be done automatically, because `short` is smaller than `int`. In this case we will have to explicitly cast it and this conversion is called *narrowing*.

Java is Statically Typed Language

Note that Java is a *statically-typed* language that means the type of a variable must be known at compile time, as type checking is done during compilation of the code. The main advantage static typing offers is that type checking is done by the compiler before the program is executed. The compiler can therefore deduce whether or not a given variable is capable of performing actions requested of it. For example, the following piece of code will result into compilation error.

```
/* StaticTypedDemo.java */
public class StaticTypedDemo
{
    public static void main(String[] args)
    {
        byte s = 90;
        s = s + 2; // Type mismatch: cannot convert from int to byte
        System.out.println(s);
```

```
    }
}
```

Because, in statement s = s + 2; the intermediate term s + 2 is automatically promoted to `int` and therefore, cannot be assigned to a `byte` without the use of a cast. The same example also illustrates the strong type checking that Java places severe restrictions on the intermixing of operations that is permitted to occur.

ASCII defines 128 characters, which map to the numbers 0–127. Unicode defines (less than) 2^{21}characters, which, similarly, map to numbers 0–2^{21} (though not all numbers are currently assigned, and some are reserved).
Unicode is a superset of ASCII, and the numbers 0–128 have the same meaning in ASCII as they have in Unicode. For example, the number 65 means "Latin capital 'A'".

Because Unicode characters don't generally fit into one 8-bit byte, there are numerous ways of storing Unicode characters in byte sequences, such as UTF-32 and UTF-8.

Unicode vs ASCII

ASCII and Unicode are two character encodings. Basically, they are standards on how to represent difference characters in binary so that they can be written, stored, transmitted, and read in digital media. The main difference between the two is in the way they encode the character and the number of bits that they use for each. ASCII originally used seven bits to encode each character. This was later increased to eight with Extended ASCII to address the apparent inadequacy of the original. In contrast, Unicode uses a variable bit encoding program where you can choose between 32, 16, and 8-bit encodings. Using more bits lets you use more characters at the expense of larger files while fewer bits give you a limited choice but you save a lot of space. Using fewer bits (i.e. UTF-8 or ASCII) would probably be best if you are encoding a large document in English.
One of the main reasons why Unicode was the problem arose from the many non-standard extended ASCII programs. Unless you are

using the prevalent page, which is used by Microsoft and most other software companies, then you are likely to encounter problems with your characters appearing as boxes. Unicode virtually eliminates this problem as all the character code points were standardized.
Another major advantage of Unicode is that at its maximum it can accommodate a huge number of characters. Because of this, Unicode currently contains most written languages and still has room for even more. This includes typical left-to-right scripts like English and even right-to-left scripts like Arabic. Chinese, Japanese, and the many other variants are also represented within Unicode. So Unicode won't be replaced anytime soon.

In order to maintain compatibility with the older ASCII, which was already in widespread use at the time, Unicode was designed in such a way that the first eight bits matched that of the most popular ASCII page. So if you open an ASCII encoded file with Unicode, you still get the correct characters encoded in the file. This facilitated the adoption of Unicode as it lessened the impact of adopting a new encoding standard for those who were already using ASCII.

Summary:

1.ASCII uses an 8-bit encoding while Unicode uses a variable bit encoding.
2.Unicode is standardized while ASCII isn't.
3.Unicode represents most written languages in the world while ASCII does not.
4.ASCII has its equivalent within Unicode

A ***symbolic constant*** is an **"variable"** whose value does not change during the entire lifetime of the program. There are no symbolic constants in java. Final variables serve as symbolic constants. A final variable

declaration is qualified with the reserved word **`final`**. The variable is set to a value in the declaration and cannot be reset. Any such attempt is caught at compile time.

final double PI = 3.14 ;
They are constants because of the 'final' keywords, so they canNOT be reassigned a new value after being declared as final
And they are symbolic , because they have a name

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

Cast Operator:

1. Cast operator is used to convert numeric values from one numeric type to another or to change an object reference to a compatible type.
2. Used to enable conversions that would normally be disallowed by the compiler. a reference of any object can be cast to a reference of type Object
3. a reference to an object can be cast into a reference of type ClassName if the actual class of the object, when it was created, is a subclass of ClassName.
4. a reference to an object can be cast into a reference of type InterfaceName if the class of the object implements Interface, if the object is a subinterface of InterfaceName or if the object is an array type and InterfaceName is the Cloneable interface.

The **difference** is that the **short circuit** operator doesn't evaluate the second operand if the first operand is true, which the **logical** OR without **short circuit**always evaluates both operands.

Static Method: Restrictions.

a. Access only static type data (static type instance variable).
b. Call only static method ,if non-static then compile time error.
c. No need the class object to call the static method.
d. Cant use this and super keyword otherwise compile time error.

Constructors

1. Constructors have the same name as that of the class they belong to.
2. Constructors are executed when an object is declared.
3. Constructors have neither return value nor void.
4. The main function of constructor is to initialize objects and allocate appropriate memory to objects.
5. Though constructors are executed implicitly, they can be invoked explicitly.
6. Constructors can have default values and can be overloaded.
7. The constructor without arguments is called as default constructor.

Destructors

1. Destructors have the same name as that of the class they belong to preceded by ~ (tilde).
2. Similar to constructors, the destructors do not have return type and not even void.
3. Constructors and destructors cannot be inherited, though a derived class can call the constructors and destructors of the base class.
4. Destructors can be virtual, but constructors cannot.
5. Only one destructor can be defined in the destructor. The destructor does not have any arguments.
6. Destructors neither have default values nor can be overloaded.
7. Programmer cannot access addresses of constructors and destructors.
8. TURBO C++ compiler can define constructors and destructors if they have not been explicitly defined. They are also called on many cases without explicit calls in programs. Any constructor or destructor created by the compiler will be public.
9. Constructors and destructors can make implicit calls to operators new and delete if memory allocation/de-allocation is needed for an object.
10. An object with a constructor or destructor cannot be used as a member of a union.

	Method Overloading	Method Overriding
Definition	In Method Overloading, Methods of the same class shares the same name but each method must have different number of parameters or parameters having different types and order.	In Method Overriding, sub class have the same method with same name and exactly the same number and type of parameters and same return type as a super class.
Meaning	Method Overloading means more than one method shares the same name in the class but having different signature.	Method Overriding means method of base class is re-defined in the derived class having same signature.
Behavior	Method Overloading is to “add” or “extend” more to method’s behavior.	Method Overriding is to “Change” existing behavior of method.
	Overloading and Overriding is a kind of polymorphism.Polymorphism means “one name, many forms”.	
Polymorphism	It is a **compile time polymorphism**.	It is a **run time polymorphism**.
Inheritance	It may or may not need **inheritance** in Method Overloading.	It always requires inheritance in Method Overriding.
Signature	In Method Overloading, methods must have **different signature**.	In Method Overriding, methods must have **same signature**.
Relationship of Methods	In Method Overloading, relationship is there between methods of same class.	In Method Overriding, relationship is there between methods of super class and sub class.
Criteria	In Method Overloading, methods have same name different signatures but in the same class.	In Method Overriding, methods have same name and same signature but in the different class.
No. of Classes	Method Overloading does not require more than one class for overloading.	Method Overriding requires at least two classes for overriding.
Example		

```
Class Add

{

   int sum(int a, int b)

   {

     return a + b;

   }

  int sum(int a)

  {

    return a + 10;

   }

}
```

```
Class A  // Super Class

{

  void display(int num)

  {

     print num ;

   }

}

//Class B inherits Class A

Class B //Sub Class

{

  void display(int num)

  {

     print num ;

   }

}
```