**Class ,Object and Reference :**

* a **class** is like the blueprint for a house. Using this blueprint, you can build as many houses as you like.
* each house you build (or instantiate, in OO lingo) is an **object**, also known as an **instance**.
* each house also has an address, of course. If you want to tell someone where the house is, you give them a card with the address written on it. That card is the object's **reference**.
* If you want to visit the house, you look at the address written on the card. This is called **dereferencing**.

You can copy that reference as much as you like, but there's just one house -- you're just copying the card that has the address on it, not the house itself.

In Java, you can not access objects directly, you can only use references. Java does not copy or assign objects to each other. But you can copy and assign references to variables so they refer to the same object. Java methods are always pass-by-value, but the value could be an object's reference. So, if I have:

** What is OOPS?**

Object Oriented Programming System is the programming technique to write programs based on the real world objects. The states and behaviors of an object are represented as the member variables and methods. In OOPS programming programs are organized around objects and data rather than actions and logic.

** What are the advantages of OOPS concepts?**

Major advantages of OOPS programming are;

1. **Simplicity**: OOPS programming objects model real world objects, so the complexity is reduced and the program structure is clear.
2. **Modularity**: Each object forms a separate entity whose internal workings are decoupled from other parts of the system.
3. **Modifiability**: It is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.
4. **Extensibility**: Adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.
5. **Maintainability**: Objects can be maintained separately, making locating and fixing problems easier.
6. **Reusability**: Objects can be reused in different programs.

** What is the difference between Procedural programming and OOPS?**

1. Procedural language is based on functions but object oriented language is based on real world objects.
2. Procedural language gives importance on the sequence of function execution but object oriented language gives importance on states and behaviors of the objects.
3. Procedural language exposes the data to the entire program but object oriented language encapsulates the data.
4. Procedural language follows top down programming paradigm but object oriented language follows bottom up programming paradigm.
5. Procedural language is complex in nature so it is difficult to modify, extend and maintain but object oriented language is less complex in nature so it is easier to modify, extend and maintain.
6. Procedural language provides less scope of code reuse but object oriented language provides more scope of code reuse.

** What are the core concepts of OOPS?**

OOPS core concepts are;

1.  Abstraction
2. Encapsulation
3. Polymorphism
4. Inheritance
5. Composition
6. Association
7. Aggregation

** What is Abstraction?**

Abstraction is an OOPS concept to construct the structure of the real world objects. During this construction only the general states and behaviors are taken and more specific states and behaviors are left aside for the implementers.

** What is Encapsulation?**

Encapsulation is an OOPS concept to create and define the permissions and restrictions of an object and its member variables and methods. A very simple example to explain the concept is to make the member variables of a class private and providing public getter and setter methods. Java provides four types of access level modifiers: public, protected, no modifier and private.

** What is the difference between Abstraction and Encapsulation?**

1. “Program to interfaces, not implementations” is the principle for Abstraction and “Encapsulate what varies” is the OO principle for Encapsulation.
2. Abstraction provides a general structure of a class and leaves the details for the implementers. Encapsulation is to create and define the permissions and restrictions of an object and its member variables and methods.
3. Abstraction is implemented in Java using interface and abstract class while Encapsulation is implemented using four types of access level modifiers: public, protected, no modifier and private.

** What is Polymorphism?**

Polymorphism is the occurrence of something in various forms. Java supports various forms of polymorphism like polymorphic reference variables, polymorphic method, polymorphic return types and polymorphic argument types.

** What is Inheritance?**

A subclass can inherit the states and behaviors of it’s super class is known as inheritance.

** What is multiple inheritance?**

A child class inheriting states and behaviors from multiple parent classes is known as multiple inheritance.

** What is the diamond problem in inheritance?**

In case of multiple inheritance, suppose class A has two subclasses B and C, and a class D has two super classes B and C.If a method present in A is overridden by both B and C but not by D then from which class D will inherit that method B or C? This problem is known as diamond problem.

** Why Java does not support multiple inheritance?**

Java was designed to be a simple language and multiple inheritance introduces complexities like diamond problem. Inheriting states or behaviors from two different type of classes is a case which in reality very rare and it can be achieved easily through an object association.

** What is Static Binding and Dynamic Binding?**

Static or early binding is resolved at compile time. Method overloading is an example of static binding.

Dynamic or late or virtual binding is resolved at run time. Method overriding is an example of dynamic binding.

** What is the meaning of “IS-A” and “HAS-A” relationship?**

“IS-A” relationship implies inheritance. A sub class object is said to have “IS-A” relationship with the super class or interface. If class A extends B then A “IS-A” B. It is transitive, that is, if class A extends B and class B extends C then A “IS-A” C. The “instanceof” operator in java determines the “IS-A” relationship.

When a class A has a member reference variable of type B then A “HAS-A” B. It is also known as Aggregation.

** What is Association?**

Association is a relationship between two objects with multiplicity.

** What is Aggregation?**

Aggregation is also known as “HAS-A” relationship. When class Car has a member reference variable of type Wheel then the relationship between the classes Car and Wheel is known as Aggregation. Aggregation can be understood as “whole to its parts” relationship.

Car is the whole and Wheel is part. Wheel can exist without the Car. Aggregation is a weak association.

** What is Composition?**

Composition is a special form of Aggregation where the part cannot exist without the whole. Composition is a strong Association. Composition relationship is represented like aggregation with one difference that the diamond shape is filled.

** What is Dependency?**

When one class depends on another because it uses that at some point in time then this relationship is known as Dependency. One class depends on another if the independent class is a parameter variable or local variable of a method of the dependent class. A Dependency is drawn as a dotted line from the dependent class to the independent class with an open arrowhead pointing to the independent class.

** What is the difference between Association and Dependency?**

The main difference between Association and Dependency is in case of Association one class has an attribute or member variable of the other class type but in case of Dependency a method takes an argument of the other class type or a method has a local variable of the other class type.

** What is a Class?**

A class is the specification or template of an object.

** What is an Object?**

Object is instance of class.

**1.What is polymorphism and what are the types of it?**

Single task can be done in different way.

1)Method overloading(compile time polymorphism),

2)method overriding(run time polymorphism)

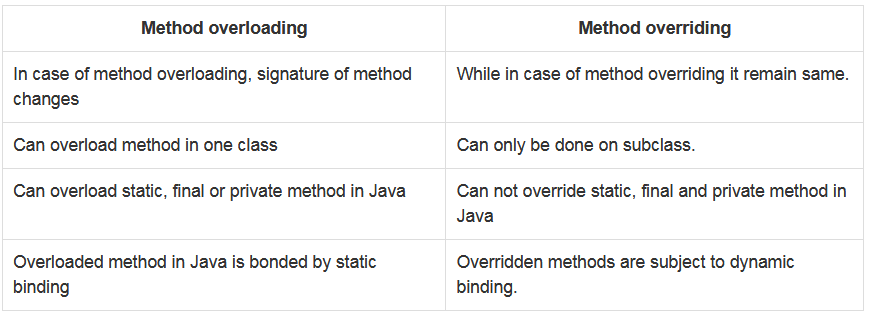
**2.What is method overriding?**

Specific implementation of a method for child class.

**3.What is method overloading?**

If a class have multiple methods by same name but different parameters, it is known as Method Overloading.

**4.Difference between method overloading and overriding?**



**5.What is static and dynamic binding?**

static binding type of object is determined at compile time whereas in dynamic binding type of object is determined at run time.

**6.Why method overloading is not possible by changing the return type in java?**

**7.can we overload main() method?**

Yes,  we can have many main() methods in a class by overloading the main method.

Example:

1. public class overldmain
2. {
3. public static void main(String[] args)
4. {
5. System.out.println("main(String[] args)");
6. }
8. public static void main(String args1)
9. {
10. System.out.println("main(String arg1)");
11. }
13. public static void main(String arg1, String arg2)
14. {
15. System.out.println("main(String arg1, String arg2)");
16. }
17. }

**8.What is run time polymorphism and compile time polymorphism?**

2. **Compile time polymorphism:**
3. it is nothing but the method overloading in java. In simple terms we can say that a class can have more than one methods with same name but with different number of arguments or different types of arguments or both.
4. **Runtime polymorphism:**
5. Runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time.
6. In this process, an overridden method is called through the reference variable of a super class. The determination of the method to be called is based on the object being referred to by the reference variable.
7. Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to [constructor overloading](https://beginnersbook.com/2013/05/constructor-overloading/) in Java, that allows a class to have more than one constructor having different argument lists.
8. let’s get back to the point, when I say argument list it means the parameters that a method has: For example the argument list of a method add(int a, int b) having two parameters is different from the argument list of the method add(int a, int b, int c) having three parameters.

## Three ways to overload a method

1. In order to overload a method, the argument lists of the methods must differ in either of these:  
   1. Number of parameters.  
   For example: This is a valid case of overloading
2. add(int, int)
3. add(int, int, int)
4. 2. Data type of parameters.  
   For example:

add(int, int)

add(int, float)

3. Sequence of Data type of parameters.  
For example:

add(int, float)

add(float, int)

**Invalid case of method overloading:**  
When I say argument list, I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

int add(int, int)

float add(int, int)

**Method overloading** is an example of [Static Polymorphism](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/). We will discuss [polymorphism](https://beginnersbook.com/2013/03/polymorphism-in-java/) and types of it in a separate tutorial.

**Points to Note:**  
1. Static Polymorphism is also known as compile time binding or early binding.  
2. [Static binding](https://beginnersbook.com/2013/04/java-static-dynamic-binding/) happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

## Method Overloading examples

As discussed in the beginning of this guide, method overloading is done by declaring same method with different parameters. The parameters must be different in either of these: number, sequence or types of parameters (or arguments). Lets see examples of each of these cases.

Argument list is also known as parameter list

### Example 1: Overloading – Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

class DisplayOverloading

{

public void disp(char c)

{

System.out.println(c);

}

public void disp(char c, int num)

{

System.out.println(c + " "+num);

}

}

class Sample

{

public static void main(String args[])

{

DisplayOverloading obj = new DisplayOverloading();

obj.disp('a');

obj.disp('a',10);

}

}

**Output:**

a

a 10

In the above example – method disp() is overloaded based on the number of parameters – We have two methods with the name disp but the parameters they have are different. Both are having different number of parameters.

### Example 2: Overloading – Difference in data type of parameters

In this example, method disp() is overloaded based on the data type of parameters – We have two methods with the name disp(), one with parameter of char type and another method with the parameter of int type.

class DisplayOverloading2

{

public void disp(char c)

{

System.out.println(c);

}

public void disp(int c)

{

System.out.println(c );

}

}

class Sample2

{

public static void main(String args[])

{

DisplayOverloading2 obj = new DisplayOverloading2();

obj.disp('a');

obj.disp(5);

}

}

Output:

a

5

### Example3: Overloading – Sequence of data type of arguments

Here method disp() is overloaded based on sequence of data type of parameters – Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

class DisplayOverloading3

{

public void disp(char c, int num)

{

System.out.println("I’m the first definition of method disp");

}

public void disp(int num, char c)

{

System.out.println("I’m the second definition of method disp" );

}

}

class Sample3

{

public static void main(String args[])

{

DisplayOverloading3 obj = new DisplayOverloading3();

obj.disp('x', 51 );

obj.disp(52, 'y');

}

}

**Output:**

I’m the first definition of method disp

I’m the second definition of method disp

## Method Overloading and Type Promotion

When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

**What it has to do with method overloading?**  
Well, it is very important to understand type promotion else you will think that the program will throw compilation error but in fact that program will run fine because of type promotion.  
Lets take an example to see what I am talking here:

class Demo{

void disp(int a, double b){

System.out.pr intln("Method A");

}

void disp(int a, double b, double c){

System.out.println("Method B");

}

public static void main(String args[]){

Demo obj = new Demo();

/\* I am passing float value as a second argument but

\* it got promoted to the type double, because there

\* wasn't any method having arg list as (int, float)

\*/

obj.disp(100, 20.67f);

}

}

Output:

Method A

As you can see that I have passed the float value while calling the disp() method but it got promoted to the double type as there wasn’t any method with argument list as (int, float)

But this type promotion doesn’t always happen, lets see another example:

class Demo{

void disp(int a, double b){

System.out.println("Method A");

}

void disp(int a, double b, double c){

System.out.println("Method B");

}

void disp(int a, float b){

System.out.println("Method C");

}

public static void main(String args[]){

Demo obj = new Demo();

/\* This time promotion won't happen as there is

\* a method with arg list as (int, float)

\*/

obj.disp(100, 20.67f);

}

}

Output:

Method C

As you see that this time type promotion didn’t happen because there was a method with matching argument type.  
**Type Promotion table:**   
The data type on the left side can be promoted to the any of the data type present in the right side of it.

byte → short → int → long

short → int → long

int → long → float → double

float → double

long → float → double

## Lets see few Valid/invalid cases of method overloading

Case 1:

int mymethod(int a, int b, float c)

int mymethod(int var1, int var2, float var3)

Result: Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types.

Case 2:

int mymethod(int a, int b)

int mymethod(float var1, float var2)

Result: Perfectly fine. Valid case of overloading. Here data types of arguments are different.

Case 3:

int mymethod(int a, int b)

int mymethod(int num)

Result: Perfectly fine. Valid case of overloading. Here number of arguments are different.

Case 4:

float mymethod(int a, float b)

float mymethod(float var1, int var2)

Result: Perfectly fine. Valid case of overloading. Sequence of the data types of parameters are different, first method is having (int, float) and second is having (float, int).

Case 5:

int mymethod(int a, int b)

float mymethod(int var1, int var2)

Result: Compile time error. Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since return type of method doesn’t matter while overloading a method.

Guess the answers before checking it at the end of programs:  
**Question 1 – return type, method name and argument list same.**

class Demo

{

public int myMethod(int num1, int num2)

{

System.out.println("First myMethod of class Demo");

return num1+num2;

}

public int myMethod(int var1, int var2)

{

System.out.println("Second myMethod of class Demo");

return var1-var2;

}

}

class Sample4

{

public static void main(String args[])

{

Demo obj1= new Demo();

obj1.myMethod(10,10);

obj1.myMethod(20,12);

}

}

**Answer:**  
It will throw a compilation error: More than one method with same name and argument list cannot be defined in a same class.

**Question 2 – return type is different. Method name & argument list same.**

class Demo2

{

public double myMethod(int num1, int num2)

{

System.out.println("First myMethod of class Demo");

return num1+num2;

}

public int myMethod(int var1, int var2)

{

System.out.println("Second myMethod of class Demo");

return var1-var2;

}

}

class Sample5

{

public static void main(String args[])

{

Demo2 obj2= new Demo2();

obj2.myMethod(10,10);

obj2.myMethod(20,12);

}

}

**Answer:**  
It will throw a compilation error: More than one method with same name and argument list cannot be given in a class even though their return type is different. **Method return type doesn’t matter in case of overloading.**

# Difference Between Interface and Abstract Class

Last modified on September 7th, 2014 by Joe.

1. Interface should have only abstract methods but abstract class can have abstract methods or concrete methods or combination of both.
2. Interface should declare with abstract keyword and abstract class should declare with abstract keyword
3. Variables declared in a Java interface is by default final. An  abstract class may contain non-final variables.
4. Members of a Java interface are public by default. A Java abstract class can have the class members like private, protected, etc..
5. Java interface should be implemented using keyword “implements”; A Java abstract class should be extended using keyword “extends”.
6. An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
7. A Java class can implement multiple interfaces but it can extend only one abstract class.
8. Interface is absolutely abstract and cannot be instantiated; A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.
9. In comparison with java abstract classes, java interfaces are slow as it requires extra indirection.