# [Polymorphism In Java](http://javaconceptoftheday.com/polymorphism-in-java/)

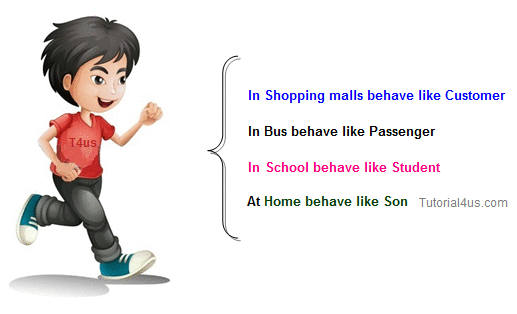
In Greek, Poly means many and morph means shapes or forms. So, polymorphism refers to any entity which takes many forms.

Or  
Polymorphism means defining one entity with multiple behaviors.

Polymorphism in java refers to any entity whether it is an operator or a constructor or any method which takes many forms or can be used for multiple tasks either while compiling or while running a java program.

### Real life example of polymorphism

Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person present in different-different behaviors.



The main Advantage of Polymorphism is flexibility, because using one entity we can perform multiple operations

**Eg:**

Student, Pavan kalyan, println(),...

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

### Example

Let us look at an example.

*public interface Vegetarian{}*

*public class Animal{}*

*public class Deer extends Animal implements Vegetarian{}*

Now, the Deer class is considered to be polymorphic since this has multiple inheritances. Following are true for the above examples −

* ***A Deer IS-A Animal***
* ***A Deer IS-A Vegetarian***
* ***A Deer IS-A Deer***
* ***A Deer IS-A Object***

*When* we apply the reference variable facts to a Deer object reference, the following declarations are legal −

### Example :

*Deer d = new Deer();*

*Animal a = d;*

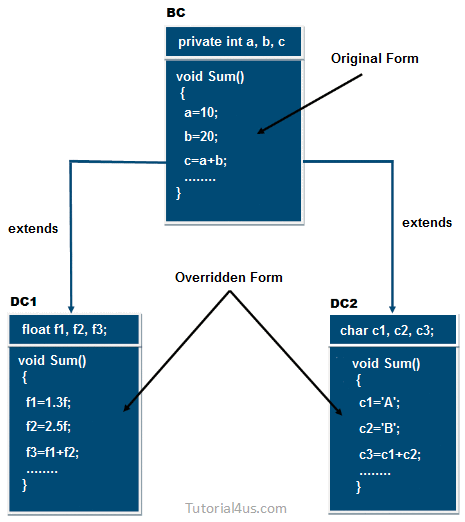
*Vegetarian v = d;*

*Object o = d;*

All the reference variables d, a, v, o refer to the same Deer object in the heap.

### Let us consider the following diagram

Here original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.



In the above diagram the sum method which is present in BC class is called original form and the sum() method which are present in DC1 and DC2 are called overridden form hence Sum() method is originally available in only one form and it is further implemented in multiple forms. Hence Sum() method is one of the polymorphism method.

**In Java we have following 2 types of poly morphisms.**

1. **Compile time polymorphism**

If polymorphism is decided at the time of compilation by java compiler then it is called as compile time polymorphism.  
By using the concept of method overloading we can achieve this compile time polymorphism.

2. **Runtime polymorphism**

If polymorphism is decided at the time of execution time by JVM then it is called as run **time polymorphism**.

By using the concept of method overriding we can achieve this runtime poly morphism.

**In other words,**

**There are two types of polymorphism in Java.**

**1) Static Polymorphism / Compile time Polymorphism      
2) Dynamic Polymorphism / run time polymorphism**

## ****1) Static Polymorphism****

The process of binding the overloaded method within object at compile time is known as **Static polymorphism** due to static polymorphism utilization of resources (main memory space) is poor because for each and every overloaded method a memory space is created at compile time when it binds with an object. In C++ environment the above problem can be solve by using dynamic polymorphism by implementing with virtual and pure virtual function so most of the C++ developer in real worlds follows only dynamic polymorphism.

**Operator Overloading, Constructor Overloading and**[**method overloading**](http://javaconceptoftheday.com/method-overloading-in-java/) are best examples of static polymorphism. Because, they show polymorphism during compilation.

In static polymorphism, object used is determined during compilation itself. So, it is called “**static binding or Early Binding”**.

**Operator Overloading :**

For example, Operator ‘+’ can be used to add two numbers and also can be used to concatenate two strings. It is called operator overloading. ‘+’ is the only operator in java which is used for operator overloading.

**Constructor Overloading :**

We can include multiple constructors in a class. This is called **constructor overloading**. Through [constructor overloading](http://javaconceptoftheday.com/constructors-in-java/), we can create objects to the class in multiple ways. This shows the polymorphism.

**Method Overloading/ Method Over writing:**

***Method Overloading=Method name is same + Signature is different.***

If two or more method in a class have same name but different parameters, it is known as method overloading.

Method overloading is one of the ways through which java supports polymorphism. Method overloading can be done by changing number of arguments or by changing the data type of arguments. If two or more method have same name and same parameter list **but differs in return type are not** said to be overloaded method

We can have different forms of same method in the same class. This is called [**method overloading**](http://javaconceptoftheday.com/method-overloading-in-java/). Through method overloading, we can perform different tasks through different forms of the same method. This shows the polymorphism.

Method overloading means declaring multiple methods with same method name but having different method signature.

In method overloading while writing the method signature we have to follow following 3 Rules

1. Method name must be same for all methods

2. List of parameters must be different like different type of parameters, different number of parameters, and different order of parameters.

3. Return type is not considered in method overloading, it means we never decide method overloading with return type.



In [casting](http://javaconceptoftheday.com/type-casting-in-java/), we have seen super class reference variable can refer to objects of its sub-class. This also shows polymorphism. For example,

|  |
| --- |
| *class A{*  *//Some Statements*  *}*  *class B extends A{*  *//Some Statements*  *}*  *class C extends B{*  *//Some Statements*  *}*  *public class D{*  *public static void main(String[] args){*  *A a = new A();* ***//A-Class variable refers to A-Class object***  *a = new B();* ***//A-Class variable refers to B-Class object***  *a = new C();* ***//A-Class variable refers to C-Class object***  *}*  *}* |

In the above example, ‘a’ is Class A-type reference variable which can be used to refer objects of A-type, B-type or C-type. Because, B-type and C-type are sub class of A-type. This also shows the polymorphism.

**Note:**

* We implement the concept of static polymorphism using method overloading.
* But where as we would be implementing the concept of dynamic polymorphism through method overriding.

**//wap to demo on method overloading**

(using different types of parameters)

*class Demo{*

*void show(int a){*

*System.out.println("int parameter a="+a);*

*}*

*void show(double a){*

*System.out.println("double parameter a="+a);*

*}*

*void show(String a){*

*System.out.println("String parameter a="+a);*

*}*

*void show(boolean a){*

*System.out.println("boolean parameter a="+a);*

*}*

*}*

*class Polymorphism1{*

*public static void main(String ars[]){*

*Demo d = new Demo();*

*d.show(20.3);*

*d.show(true);*

*d.show("abc");*

*d.show(10f);*

*d.show(10);*

*d.show(new String("abc"));*

*d.show('A');*

*}*

*}*

## ****2) Dynamic Polymorphism****

In dynamic polymorphism, method of the program binds with an object at runtime the advantage of dynamic polymorphism is allocating the memory space for the method (either for overloaded method or for override method) at run time.

Out of multiple functions with the same name and with the same data types of the parameters which function has to get executed would be decided dynamically at the run time based on the Runtime parameters because of which we call this polymorphism as Dynamic /Runtime polymorhism.

Thc concept of defining multiple functions with the same name and with the same datatypes of the parameters associated with the same object is known as Dynamic Polymorphism.

[**Method Overriding**](http://javaconceptoftheday.com/method-overriding-java/) is the best example of dynamic polymorphism. It is also called **dynamic binding or late binding**, because type of the object used will be determined at run time only.

**Consider the following example,**

*class SuperClass{*

*void methodOfSuperClass(){*

*System.out.println("From Super Class");*

*}*

*}*

*class SubClass extends SuperClass{*

***//Super Class Method Overrided***

*void methodOfSuperClass(){*

*System.out.println("From Sub Class");*

*}*

*}*

*public class D{*

*static void util(SuperClass superclass){*

*superclass.methodOfSuperClass();*

***//For each execution of this method, different objects will be passed to it.***

***//which Object will be used is determined during run time only.***

***//This shows dynamic polymorphism.***

*}*

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

*SuperClass superclass1 = new SuperClass();*

*SubClass subclass = new SubClass();*

*SuperClass superclass2 = new SubClass();*

*util(superclass1);* ***//SuperClass object is passes to util()***

*util(subclass);* ***//SubClass object is passed to util()***

*util(superclass2);* ***//SubClass object is passed to util()***

*}*

*}*

**Conclusion**

The advantage of dynamic polymorphism is effective utilization of the resources, so java always use dynamic polymorphism. Java does not support static polymorphism because of its limitation.

**Java method overloading**

**Method Overloading :**

If a class have multiple methods by same name but different 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.

**Suppose** you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behaviour of the method because its name differs. So, we perform method overloading to figure out the program quickly.

**Advantage of method overloading**

Method overloading **increases the readability of the program**.

1. In Java, we can define number of methods in a class with the same name.
2. Defining two or more methods with the same name in a class is called **method overloading**.
3. Compile determine which **method to execute automatically**.

**Example:**

***class****Calculation{*

***void****sum(****int****a,****int****b){*

*System.out.println(a+b);*

*}*

***void****sum(****int****a,****int****b,****int****c){*

*System.out.println(a+b+c);*

*}*

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

*Calculation obj=****new****Calculation();*

*obj.sum(10,10,10);*

*obj.sum(20,20);*

*}*

*}*

**Output**:

30

40

1. The return type has no effect on the signature of a method.

**What is signature of the method ?**

Compiler checks **method signature** for duplicate methods or for method overloading. Method signature consist of three things,

**1) Method Name**

**2) Number Of Arguments**

**3) Types of arguments.**

If these three things are same for any two methods in a class, then compiler gives **duplicate method error.**

1. The name of a method along with the **types and sequence of the parameter**s is called **signature of the method**
2. **Signature of the method** should be unique.

### Signature of the method includes –

1. **Number of Parameteres**
2. **Different Parameters**
3. **Sequence of the parameters**

## Compiler first checks method name. If it is same, then it checks number of arguments. If methods differs in number of arguments, then it does not check types of argument. It treats as methods are overloaded. If number of arguments are same then compiler checks types of arguments. If types of arguments are also same, then compiler will give duplicate method error. If types of arguments are not same, then compiler will treat them as methods are overloaded.

## Some Examples of the Method Overloading :

*int display(int num1,int num2);*

Or

*int display(double num1,int num2);*

Or

*void display(double num1,double num2);*

For method overloading to be successful, method name must be same and methods must have different number of arguments or different types of arguments. If method names are different, then those methods will be treated as two different methods.

|  |
| --- |
| *public class MethodOverloading{*  *void methodOverloaded(){*  ***//No argument method***  *}*  *void methodOverloaded(int i){*  ***//One argument is passed***  *}*  *void methodOverloaded(double d){*  ***//One argument is passed but type of argument is different***  *}*  *void methodOverloaded(int i, double d){*  ***//Two argument method***  ***//Method signature of this method is methodOverloaded(int, double)***  *}*  *void methodOverloaded(double d, int i){*  ***//It is also two argument method but type of arguments changes***  ***//Method signature of this method is methodOverloaded(double, int)***  *}*  *void methodOverloaded(double d1, int i1){*  ***//It has same method signature methodOverloaded(double, int) as of above method***  ***//So, it is a Duplicate method, You will get compile time error here***  *}*  *void differentMethod(){*  ***//Different method***  *}*  *}* |

Overloaded methods may have same return types or different return types. It does not effect method overloading.

|  |
| --- |
| *public class MethodOverloading{*  *void methodOverloaded(){*  *//No argument method, return type is void*  *}*  *int methodOverloaded(int i){*  *//Returns int type*  *return i;*  *}*  *int methodOverloaded(double d){*  *//Same return type as of above method*  *return 0;*  *}*  *void methodOverloaded(double d){*  *//Duplicate method because it has same method signature as of above method*  *}*  *}* |

**Important Note :**

If two methods have same signature and different return types, then those methods will not be treated as two different methods or methods overloaded. For duplication, compiler checks only method signature not return types. If method signature is same, straight away it gives duplicate method error.

Overloaded methods may have same access modifiers or different access modifiers. It also does not effect method overloading.

|  |
| --- |
| *public class MethodOverloading{*  *private void methodOverloaded(){*  *//No argument, private method*  *}*  *private int methodOverloaded(int i){*  *//One argument private method*  *return i;*  *}*  *protected int methodOverloaded(double d){*  *//Protected Method*  *return 0;*  *}*  *public void methodOverloaded(int i, double d){*  *//Public Method*  *}*  *}* |

Overloaded methods may be static or non-static. This also does not effect method overloading.

|  |
| --- |
| *public class MethodOverloading{*  *private static void methodOverloaded(){*  *//No argument, private static method*  *}*  *private int methodOverloaded(int i){*  *//One argument private non-static method*  *return i;*  *}*  *static int methodOverloaded(double d){*  *//static Method*  *return 0;*  *}*    *public void methodOverloaded(int i, double d){*  *//Public non-static Method*  *}*  *}* |

From the above examples, it is clear that compiler will check only method signature for method overloading or for duplicate methods. It does not check return types, access modifiers and static or non-static.

**Example:**

***class****Calculation2{*

***void****sum(****int****a,****int****b){*

*System.out.println(a+b);*

*}*

***void****sum(****double****a,****double****b){*

*System.out.println(a+b);*

*}*

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

*Calculation2 obj=****new****Calculation2();*

*obj.sum(10.5,10.5);*

*obj.sum(20,20);*

*}*

*}*

**Output**:

21.0

40

**Example:**

**Observe the following functions of C-lang.**

The above two functions of the same name **display()**raises compilation error in C-lang as C cannot differentiate the same functions depending on their parameters.

But C++ and Java can differentiate the same named functions depending on their parameters. **The same method declared, in the same class, a number of times, but with different parameters is known as "method overloading".**With method overloading, the same method called at different times gives different outputs (functionalities).

*public class Animal{*

*public void eat(){* ***// I***

*System.out.println("All animals eat");*

*}*

*public void eat(int x){* ***// II***

*System.out.println("Whale eats more than " + x + " tons of food a day");*

*}*

*public void eat(String str){* ***// III***

*System.out.println("Parrots eat " + str);*

*}*

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

*Animal a1 = new Animal();*

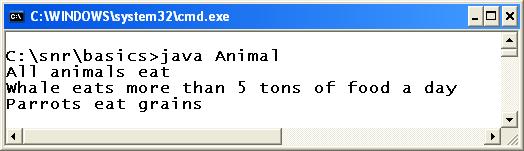
*a1.eat();* ***// calls I***

*a1.eat(5);* ***// calls II***

*a1.eat("grains");* ***// calls III***

*}*

*}*



**Live Example :**

***package*** *com.pritesh.programs;*

***class*** *Rectangle {*

*double length;*

*double breadth;*

*void area(int length, int width) {*

*int areaOfRectangle = length \* width;*

***System****.out.println("Area of Rectangle : " + areaOfRectangle);*

*}*

*void area(double length, double width) {*

*double areaOfRectangle = length \* width;*

***System****.out.println("Area of Rectangle : " + areaOfRectangle);*

*}*

*}*

***class*** *RectangleDemo {*

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

***Rectangle*** *r1 =* ***new******Rectangle****();*

*r1.area(10, 20);*

*r1.area(10.50, 20.50);*

*}*

*}*

**Explanation** :

1. We have defined 2 methods with same name and different type of parameters.
2. When both integer parameters are supplied to the method then it will execute method with all integer parameters and if we supply both floating point numbers as parameter then method with floating numbers will be executed.

**Return type in Overloading (of Java Methods Method Overloading)**

See the following two method declarations.

*public void eat()  
public int eat()*

The above two**eat()**methods differ in their return type but not in parameters. Compiler cannot judge which is to be called depending on the return types; judges only on the parameter list. The above statements raise a compilation error. **The return type may or may not be the same in method overloading.**

**Static Binding**

Compiler decides at compile time itself which overloaded method is to be called. That is, method binding is done at compile time itself. This is known as "**static binding**". At runtime, simply methods are called. Java also supports [dynamic binding](http://way2java.com/oops-concepts/dynamic-polymorphism/) which leads to [dynamic polymorphism](http://way2java.com/oops-concepts/dynamic-polymorphism/).

###### **Static Polymorphism**

In Greek, "**poly**" means "many" and "**morphism**" means "forms". Polymorphism means many forms of the same method – the same method, called at different times, gives different types of output. The three **eat()**methods print different functionalities of the nature of animals like whale and parrot. Because the polymorphism is achieved statically, at compile time, it is known as "**static polymorphism**". Java achieves **static polymorphism** with **method overloading.**

**The 3 rules based on which the concept of static polymorphism would be working:**

* Whenever JVM finds a function call, JVM always searches for the function defined to accept the same data type value of what we are passing at the argument.  
  Thus highest priority would be always given to the function defined to accept the same data type value.

As long as there is a function defined to accept the same data type value what we are passing as an argument matches.

* When there is no function defined to accept the same data type value, then it considers an alternate function.  
  Alternate function is that function, for whose parameter the value what we are passing as an argument matches. But the data type of the parameter and the data type of the parameter and the data type of the value, what we are passing would be same.
* When there is no function defined to accept the same data type value and if there are multiple alternate functions, then (JVM) it considers the priorities of the data types of the parameters with respect to the data types of the values what we are passing as arguments and accordingly the corresponding function would be executed.

**Disadvantages / limitations of static polymorphism:**

Whenever there is no function defined to accept the same data type and if there are multiple alternate functions, then there is every chance of the value, what we are passing as an argument matches to the multiple alternate function parameters with the same priority which leads to confusions and a compilation error.

**Does overloading supports dynamic polymorphism ?**

A. No. Not possible. Possible through method overriding only that too in combination with object assignment.

**Note**: Java differentiates the overloaded method by method name and its parameter list.

**Example 1:**

*class Calculation{  
void sum(int a,double b){  
 System.out.println(a+b);  
}  
void sum( double a, int b){  
 System.out.println(a+b);  
}*

*public static void main(String args[]){  
Calculation obj=new Calculation();  
obj.sum(10,10);  
obj.sum(20,20);*

*}  
}*

***Error:***

***Calculation.java:15: reference to sum is ambiguous, both method sum(int,double) in Calculation and method sum(double,int) in Calculation match  
obj.sum(10,10);  
^  
Calculation.java:16: reference to sum is ambiguous, both method sum(int,double) in Calculation and method sum(double,int) in Calculation match  
obj.sum(20,20);***

***Example 2:***

class Calculation{  
void sum(int a,double b){  
 System.out.println(a+b);  
 }  
void sum( double a, int b){  
 System.out.println(a+b);  
 }

public static void main(String args[]){  
 Calculation obj=new Calculation();  
 obj.sum(10,(double)10);  
 obj.sum((double)20,20);

}  
}

***Ans.*** *void sum(int a,double b) —- A  
void sum( double a, int b) —- B*

*obj.sum(10,10);*

*Think sharp. 10, 10 can go to A or B. The first parameter 10 can be int in A or can be converted to double in B. The same case with second parameter 10 also. This is the ambiguity. Ambiguity in implicit casting.*

*In the second example, you are converting explicitly and thereby no ambiguity to compiler.*

**Java Overload Main Method**

**Can we overload main ?**

* 1. **YES**

**Method overloading** is nothing but using the same method in the same class a number of times with different parameters. Just like any other method, the **main() method also can be overloaded**.

**Following program llustrates.**

**Program with main() overloading**

*public class Demo{*

*public static void main(){*

*System.out.println("Hello World");*

*}*

*public static void main(int x){*

*System.out.println(x);*

*}*

*public static void main(String args){* ***// observe, no [ ]***

***// it is simply a string and not array***

*System.out.println(args);*

*}*

*public static void main(String args[])*

*{*

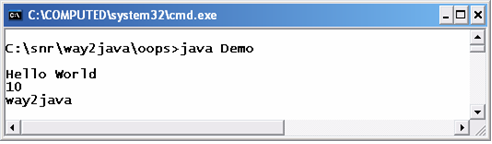
*main();*

*main(10);*

*main("way2java");*

*}*

*}*



The JVM **does not find ambiguity**to call which main() method is to be called. It knows clearly, it requires such main() that includes the parameter String args[]. Other main() methods are treated as **user-defined**. As all the **main()**methods are static in the above code, they are called without the help of an object.

**Rules**

1. We can overload only instance methods or only static methods or both at a time.

**//method overloading with different number of parameters**

*class Demo{*

*static void show(int a){*

*System.out.println("method with 1 parameters");*

*}*

*static void show(int a,int b){*

*System.out.println("method with 2 parameters");*

*}*

*void show(int a,int b,double c){*

*System.out.println("method with 3 parameters");*

*}*

*void show(){*

*System.out.println("method with 0 parameters");*

*}*

*}*

*class Polymorphism2{*

*public static void main(String ars[]){*

*Demo d = new Demo();*

*d.show();*

*d.show(10);*

*d.show(10,20,30.2);*

*d.show(10,20);*

*}*

*}*

2. We can overload methods with in the same class and we can also overload methods with in 2 different classes which are having IS-A relation.

**//method overloading with different order of parameters**

*class A{*

*void show(boolean a,int b){*

*System.out.println("method with boolean,int parameters");*

*}*

*}*

*class B extends A{*

*void show(int a,boolean c){*

*System.out.println("method with int,boolean parameters");*

*}*

*}*

*class Polymorphism3{*

*public static void main(String ars[]){*

*B b= new B();*

*b.show(10,true);*

*b.show(true,10);*

*}*

*}*

3. We can also overload main() method but JVM always invokes only main() method which takes String[] as parameter, if we want to access other oveloaded main() methods then we have to call explicitly.

*class Polymorphism4{*

*public static void main(String ars[]){*

*System.out.println("main() method with String[] as parameters");*

*Polymorphism4.main(true);*

*Polymorphism4.main(90);*

*int iarr[] = {45,45};*

*Polymorphism4.main(iarr);*

*double darr[]= {10.0,11.0};*

*Polymorphism4.main(darr);*

*}*

*public static void main(int ars[]){*

*System.out.println("main() method with int[] as parameters");*

*}*

*public static void main(double ars[]){*

*System.out.println("main() method with double[] as parameters");*

*}*

*public static void main(int a){*

*System.out.println("main()method with int as parameters");*

*}*

*public static void main(boolean a){*

*System.out.println("main()method with boolean as parameters");*

*}*

*}*

#### Different ways of Method overloading

There are two different ways of method overloading

#### Method overloading by changing data type of Arguments

***Example :***

class Calculate{

void sum (int a, int b){

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

}

void sum (float a, float b){

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

}

Public static void main (String[] args){

Calculate cal = new Calculate();

cal.sum (8,5); **//sum(int a, int b) is method is called**.

cal.sum (4.6, 3.8); **//sum(float a, float b) is called**.

}

}

**Output :**

Sum is 13

Sum is 8.4

You can see that sum() method is overloaded two times. The first takes two integer arguments, the second takes two float arguments.

#### Method overloading by changing no. of argument.

***Example* :**

class Area{

void find(int l, int b){

System.out.println("Area is"+(l\*b)) ;

}

void find(int l, int b,int h){

System.out.println("Area is"+(l\*b\*h));

}

public static void main (String[] args){

Area ar = new Area();

ar.find(8,5); **//find(int l, int b) is method is called**.

ar.find(4,6,2); **//find(int l, int b,int h) is called**.

}

}

**Output :**

Area is 40

Area is 48

In this example the find() method is overloaded twice. The first takes two arguments to calculate area, and the second takes three arguments to calculate area.

When an overloaded method is called java look for match between the arguments to call the method and the method's parameters. This match need not always be exact, sometime when exact match is not found, Java automatic type conversion plays a vital role.

#### Example of Method overloading with type promotion.

class Area{

void find(long l,long b){

System.out.println("Area is"+(l\*b)) ;

}

void find(int l, int b,int h){

System.out.println("Area is"+(l\*b\*h));

}

public static void main (String[] args){

Area ar = new Area();

ar.find(8,5);   
**//automatic type conversion from find(int,int) to find(long,long)** .

ar.find(2,4,6)   
**//find(int l, int b,int h) is called**.

}

}

**Output :**

Area is 40

Area is 48

**Why should we go for method overloading ?**

When we want to maintain the flexibility in our application like using one method performing sveral operations then we can use this method overloading.

**Note:**

Here the add method is overloaded once with two int parameters (LINE A), once with three int parameters (LINE B) and once with three double parameters (LINE C). We can define more methods with the same name add as long as the parameters are different. The following methods can be defined.

*double add(int a, double b){  
    return a + b;  
}  
double add(int a, double b, int c, double d){  
    return a + b + c + d;  
}*

But having a method with same parameters and different return type is not allowed. The below methods are not allowed, since the parameters are same and only the return type is different.

*int add(int a, int b){  
    return a + b;  
}  
double add(int a, int b) // NOT ALLOWED  
{  
    return a + b;  
}*

Similarly, the parameter names are of no importance and having two methods with same parameter type and order, but with different parameter names are not allowed.

*int add(int a, int b){  
    return a + b;  
}  
int add(int c, int d) // NOT ALLOWED  
{  
    return c + d;  
}*

**Java method overriding**

**Method Overriding= Method Heading Is Same + Method Body Is Different.**

If a method is declared in the parent class and method with same name and parameter list is written inside the subclass then it is called **method overriding**.

**Or**

In other words, if subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as **method overriding.**

**Or**

If we want to achieve the run time polymorphism then we have to use **method overriding.**

**Or**

The process of redefining original method multiple times for performing multiple operations is called Method Overriding.

When a class extends its super class, all or some members of super class are inherited to sub class. When a inherited super class method is modified in the sub class, then we call it as method is overrided. Through method overriding, we can modify super class method according to requirements of sub class.

Subclass can make use of super class methods straight away by virtue of [inheritnace](http://way2java.com/oops-concepts/inheritance/). If the subclass does not satisfy about the functionality (output) of the super class method, the subclass can rewrite with its own functionality with the same method name. This concept is known as "**method overriding**", supported by OOP languages like C++/Java. In method overriding, super class and subclass have the method with the same signature – same parameters and same return type.

### Let us discuss **rules of method overriding** with the following example.

*class Aves{*

*public void nature(){*

*System.out.println("Aves fly");*

*}*

*}*

*public class Vulture extends Aves{*

*public void nature(){*

*System.out.println("Flies very high altitudes");*

*}*

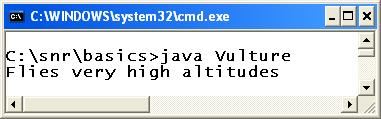
*public static void main(String args[]){*

*Vulture v1 = new Vulture();*

*v1.nature();*

*}*

*}*



In the above program, **Aves**and **Vulture**classes have the same method **nature()** with their own functionalities (outputs).

**If the super class and subclass have the same method, the subclass calls its own method**. By this rule, the subclass object, **v1**, calls its own overridden method.

**Method Overriding in java is most useful features of java. Through inheritance we can reuse already existed code and through method overriding we can modify that reused code according to our requirements. This can be best explained with example.**

|  |
| --- |
| *class SuperClass{*  *void methodOfSuperClass(){*  *System.out.println("From Super Class");*  *}*  *}*  *class SubClass extends SuperClass{*  *void methodOfSuperClass(){*  ***//SuperClass method is overrided***  ***//We can keep any tasks here according to our requirements.***  *System.out.println("From Sub Class");*  *}*  *}*  *public class MethodOverriding{*  *public static void main(String[] args){*  *SuperClass superclass = new SuperClass();*  *superclass.methodOfSuperClass();****//Output : From Super Class***  *SubClass subclass = new SubClass();*  *subclass.methodOfSuperClass();****//Output : From Sub Class***  *}*  *}* |

**Let’s discuss rules to be followed while overriding a method.**

* **Name of the overrided method** must be same as in the super class. You can’t change name of the method in subclass.
* **Return Type Of Overrided Method :**

The return type of the overrided method must be compatible with super class method. If super class method has primitive data type as its return type, then overrided method must have same return type in sub class also. If super class method has derived or user defined data type as its return type, then return type of sub class method must be of same type or its sub class. For example,

|  |
| --- |
| *class SuperClass{*  *void firstMethodOfSuperClass(){*  *System.out.println("From Super Class");*  *}*  *double secondMethodOfSuperClass(){*  *return 0.0;*  *}*  *Object thirdMethodOfSuperClass(){*  *return new Object();*  *}*  *}*  *class SubClass extends SuperClass{*  *int firstMethodOfSuperClass(){*  ***//Compile time error, return type must be void not int***  *}*  *void secondMethodOfSuperClass(){*  ***//Complie time error, return type must be double not void***  *}*  *String thirdMethodOfSuperClass(){*  ***//No Compile time error,***  ***//return type is compatible with super class method, because***  ***//String is sub class of Object class***  *return new String();*  *}*  *}* |

**Note:**

In software development, if we develop any common method for most of the programmers then they can use the common method but they should not change the definition of method and common methods must be final. In other words final methods cant be overridden.

**Note:**

A overloaded method may or maynot be final.

*final float SI(float p, float t, float r){*

*float si=(p\*t\*r)/100;*

*return si;*

*}*

**Visibility Of Overrided method :**

You can keep same visibility or increase the visibility of overrided method but you can’t reduce the visibility of overrided methods in the subclass. For example, default method can be overided as default or protected or public method but not as private.For example,

|  |
| --- |
| *class SuperClass{*  *protected void methodOfSuperClass(){*  *System.out.println("From Super Class");*  *}*  *}*  *class SubClass extends SuperClass{*  *private void methodOfSuperClass(){*  ***//Compile time error, can't reduce visibility of overrided method***  ***//here, visibility must be protected or public but not private or default***  *}*  *}* |

**Note : Visibility goes on decreasing from public to protected to default to private members.**

**Arguments Of Overrided Methods :**

For method to be properly overrided, You must not change arguments of method in subclass. If you change the number of arguments or types of arguments of overrided method in the subclass, then method will be overloaded not overrided.

*class SuperClass{*

*void methodOfSuperClass(){*

*System.out.println("From Super Class");*

*}*

*}*

*class SubClass extends SuperClass{*

***//This class will have two methodOfSuperClass() methods.***

***//one is from super class which takes no argument***

***//and one is below method which takes one argument***

*void methodOfSuperClass(int i){*

*System.out.println(i);*

*}*

*}*

*public class MethodOverloading{*

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

*SuperClass superclass = new SuperClass();*

*superclass.methodOfSuperClass();* ***//Output : From Super Class***

*SubClass subclass = new SubClass();*

*subclass.methodOfSuperClass();          /****/Output : From Super Class***

*subclass.methodOfSuperClass(10);* ***// Output : 10***

*}*

*}*

The method overloading and method overriding sounds similar; but they differ completely.

##### **Method Overloading vs Method Overriding**

|  |  |
| --- | --- |
| METHOD OVERLOADING | METHOD OVERRIDING |
| 1. Happens in the same class | Happens among two classes – a super class and a subclass |
| 2. No inheritance | Happens in inheritance only |
| 3. One method does not hide another | Subclass method hides super class method |
| 4. Should have different parameters | Should have same parameters |
| 5. Return type is not bothered | Should have the same return type |
| 6. Leads to [static binding and static polymorphism](http://way2java.com/oops-concepts/using-methods-and-method-overloading/) | Leads to [dynamic binding](http://way2java.com/oops-concepts/dynamic-polymorphism/), [dynamic polymorphism](http://way2java.com/oops-concepts/dynamic-polymorphism/) and[dynamic method dispatch](http://way2java.com/oops-concepts/dynamic-polymorphism/) |

**Java Static Inheritance**

Here, one rule to remember in inheritance is "**static members of a class cannot be overridden**".

**Why the**[**static**](http://way2java.com/oops-concepts/static-keyword-%e2%80%93-philosophy/)**members cannot be overridden?**

Generally, any member (variable or method) when overridden by the subclass, the subclass can call the super class member with "[**super**](http://way2java.com/oops-concepts/member-hiding-super-keyword/)" keyword. But "**static member**" cannot be called as with static members "[**this**](http://way2java.com/oops-concepts/using-this-keyword/)" reference cannot be used because static members can be called without the help of an object. When static members are overridden, the compiler does not raise error as compiler treats both are different belonging to their own class.

The above explanation will be clearer with the following program.

*class Test{*

*public static void display(){*

*System.out.println("Hello 1");*

*}*

*}*

*public class Demo extends Test{*

*public static void display(){*

***// super.display(); // raises compilation error***

*System.out.println("Hello 2");*

*}*

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

*display();*

*Test.display();*

*}*

*}*

The **display()** method is **static** in the super class **Test** and also in subclass **Demo**. Compiler does not treat it as method overriding as it feels each display() method belong to it's own class.

*// super.display();*

"**super**" cannot be used with static members and raises compilation error.

*display();  
Test.display();*

The **display()** declared in Demo class calls its own and **Test.display()** calls Test class display() method. No ambiguity in understanding by the compiler.

**Final conclusion:**Static members of a class cannot be overridden and "**super**" keyword cannot be used with static members (say, to call super class static method from subclass static method).

**Note:** [Private members](http://way2java.com/packages/access-specifiers-accessibility-permissions-restrictions/) of a class cannot be overridden. But public, protected and default members can be overridden.

### 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

Method overriding means declaring 2 methods with same method signature in 2 different classes which are having IS-A relation.

While Method overriding and writing the method signature we must follow following rules.

1. Method name must be same

2. List of parameters must be same

3. Return type must be same

***// wap to demo on method overriding***

*class Parent{*

*void show(int a){*

*System.out.println("Parent class show() method a="+a);*

*}*

*}*

*class Child extends Parent{*

*int x=10;*

***/\*overriding the show() method\*/***

*void show(int a){*

*System.out.println("Child class show() method a="+a);*

*}*

***/\*display() method which is not overriden\*/***

*void display(int a){*

*System.out.println("Child class show() method a="+a);*

*}*

*}*

*class Polymorphism5{*

*public static void main(String ars[]){*

*Parent p;*

*p = new Parent();*

*p.show(10);* ***//Parent class show() method a=10***

*p = new Child();*

*p.show(10);****//Child class show() method a=10***

***//p.display(10); // invalid***

***//System.out.println(p.x) //invalid***

*}*

*}*

In method overriding the method invocation will be linked with either child class method definition or parent class method definition based on the object that reference variable contains, It means that objects are available at run time and this kind of linking or binding can be decided at run time hence it is called as run time polymorphism or dynamic polymorphism.

This linking is decided at run time after the object is created so it is called as late binding.

**Note:**

A class reference variable can hold its object or all its child class objects.

*Object o1 = new String();*

*Object o2 = new Integer();*

*Object o3 = new Student();*

*String s = new Integer();* ***X- invalid (Here String and Itneger classes are not having IS-A relation)***

If parent class reference variable holds child class object then we can access every member of Parent class and we can access only overridden mehods of Child class, but we cannot access original methods and variables of Child class to resolve this problem we have to convert the child object from parent reference variable to child class reference variable(down casting)

**Why should we go for Method overriding ?**

Whenever child class don't want to use definition written by the Parent class and want to use its own logic then we have to use method overriding it means we have to override the same method with new definition inside the child class.

**Rules for Method Overriding :**

1. In Method overriding we must declare 2 methods in 2 different classes which are having Is-A relation But we never do Method overriding within the same class.

2. We can override 2 instance methods (method overriding), we can override 2 static methods (method overhiding) but we cant override 1 instance method and 1 static method.

***//wap to demo on method overriding***

*class Parent{*

*static void show(){*

*System.out.println("Parent Class show() method");*

*}*

*}*

*class Child extends Parent {*

*static void show(){*

*System.out.println("Child Class show() method");*

*}*

*}*

*class Polymorphism6{*

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

*Parent p1;*

*p1 = new Child();*

*p1.show();* ***//Parent Class show() method***

*p1 = new Parent();*

*p1.show();* ***//Parent Class show() method***

*p1 = null;*

*p1.show();* ***//Parent Class show() method***

*Child c = new Child();*

*c.show();*  ***//Child Class show() method***

*}*

*}*

1. Method Must have Same Name as that of Method Declared in Parent Class.
2. Method Must have Same Parameter List as that of Method Declared in Parent Class
3. IS-A relation should be maintained in order to Override Method.

### Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

***class*** *Vehicle{*

***void*** *run(){*

*System.out.println("Vehicle is running");*

*}*

*}*

***class*** *Bike* ***extends*** *Vehicle{*

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

*Bike obj =* ***new*** *Bike();*

*obj.run();*

*}*

*}*

**Output**:Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

### Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is 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

**Why we need to override method ?**

**Consider the following Parent Class –**

*class Human {*

***void*** *talk() {*

*System.out.println("Talking Human");*

*}*

*}*

**Consider following child class –**

*class Male extends Human {*

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

*Male m1 = new Male();*

*m1.talk();*

*}*

*}*

We want to provide the different implementation to talk() method for Male and Human class. Using override we can provide different implementations for talk() in Parent Class and in Child Class.

### Vehicle.java

***package*** *com.c4learn.inheritance;*

***public******class*** *Vehicle {*

***public*** *void vehicleMethod() {*

***System****.out.println("Method in Vehicle.");*

*}*

*}*

### TwoWheeler.java

***package*** *com.c4learn.inheritance;*

***public******class*** *TwoWheeler* ***extends*** *Vehicle {*

***public*** *void vehicleMethod() {*

***System****.out.println("Method" + " in TwoWheeler.");*

*}*

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

*TwoWheeler myBike =* ***new*** *TwoWheeler();*

*Vehicle myVehicle =* ***new*** *Vehicle();*

*myVehicle.vehicleMethod();*

*myBike.vehicleMethod();*

*}*

*}*

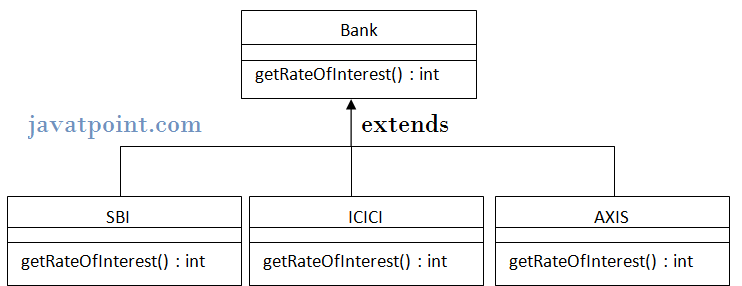
**Output of the Program :**

*Method in Vehicle.*

*Method in TwoWheeler.*

### Real example of Java Method Overriding

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.



**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

**Java method overriding rules**

**1. Same Argument List :**

The argument list should be exactly the same as that of the overridden method.

*package com.c4learn.inheritance;*

*public class InheritanceRules {*

*public* ***int*** *calculate(****int*** *num1,****int*** *num2) {*

***return*** *num1+num2;*

*}*

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

*baseClass b1 = new baseClass();*

***int*** *result = b1.calculate(10, 10);*

*System.out.println("Result : " + result);*

*}*

*}*

*class baseClass extends InheritanceRules {*

*public* ***int*** *calculate(****int*** *num1,****int*** *num2) {*

***return*** *num1\*num2;*

*}*

*}*

In the above example, method is having the same set of parameter list –

*public int calculate(int num1,int num2)*

2. **The return type :**

Return type should be the same or a subtype of the return type declared in the Original Overridden method in the Parent Class.

**Consider the following method from Parent Class –**

*public* ***int*** *calculate(****int*** *num1,****int*** *num2) {*

***return*** *num1+num2;*

*}*

then we cannot change the return type of the method in the overriden method.

3. **Modification of access level :**

Access level cannot be lowered down or restricted than the overridden method’s access level.

Suppose currently method in the Parent class is having the access level public, then we cannot override the method with access level as private and protected.

Below is the table which summarized the which are the allowed ways of modifying the access level of overridden method –

| ***Access Level in Parent*** | ***Access Level in Child*** | ***Allowed ?*** |
| --- | --- | --- |
| Public | Public | Allowed |
| Public | Private | Not Allowed |
| Public | Protected | Not Allowed |
| Protected | Public | Allowed |
| Protected | Protected | Allowed |
| Protected | Private | Not Allowed |

**Following method will throw compile time error –**

*package com.c4learn.inheritance;*

*public class InheritanceRules {*

*public* ***int*** *calculate(****int*** *num1,****int*** *num2) {*

***return*** *num1+num2;*

*}*

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

*baseClass b1 = new baseClass();*

***int*** *result = b1.calculate(10, 10);*

*System.out.println("Result : " + result);*

*}*

*}*

*class baseClass extends InheritanceRules {*

*private* ***int*** *calculate(****int*** *num1,****int*** *num2) {*

***return*** *num1\*num2;*

*}*

*}*

because in the parent class, Method has Access level public while we have made access level as private in the child class.

4. **Cannot Override Final Methods :**

We cannot override the final method declared in the Parent Class/Super Class. Consider the following method is written inside the Parent Class –

*final public* ***int*** *calculate(****int*** *num1,****int*** *num2) {*

***return*** *num1+num2;*

*}*

5. **Cannot Override Static Methods but Re-declaration is possible :**

Consider the following method declared inside the Parent Class –

*public* ***static******int*** *calculate(****int*** *num1,****int*** *num2) {*

***return*** *num1+num2;*

*}*

then child class cannot override static method from parent class but it can redeclare it just by changing the method body like this –

*public* ***static******int*** *calculate(****int*** *num1,****int*** *num2) {*

***return*** *num1\*num2;*

*}*

however we have to keep the method static in the child class, we cannot make it non-static in the child class, So following method declaration inside **child class will throw error** –

*public* ***int*** *calculate(****int*** *num1,****int*** *num2) { //without static*

***return*** *num1\*num2;*

*}*

6. **Rule for Methods that Cannot be Inherited :**

If a method cannot be inherited, then it cannot be overridden by the child class. We cannot inherite the private methods of the parent class in the child class. So[private methods cannot be overriden](http://c4learn.com/javaprogramming/instanceof-keyword-check-whether-a-class-is-instance-of-another-class/) by the childe class. However redeclaration of private method is possible in the child method –

*public class InheritanceRules {*

*private* ***int*** *calculate(****int*** *num1,****int*** *num2) {*

***return*** *num1+num2;*

*}*

*}*

*class baseClass extends InheritanceRules {*

*private* ***int*** *calculate(****int*** *num1,****int*** *num2) {*

***return*** *num1\*num2;*

*}*

*}*

7. **Constructors cannot be overridden.**

### *Can we override static method?*

No, static method cannot be overridden. It can be proved by runtime polymorphism.

### *Why we cannot override static method?*

because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

### *Can we override java main method?*

No, because main is a static method.

### *Java access modifiers with method overriding*

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

***class*** *A{*

***protected******void*** *msg(){*

*System.out.println("Hello java");*

*}*

*}*

***class*** *Simple* ***extends*** *A{*

***void*** *msg(){*

*System.out.println("Hello java");*

*}*

***//C.E:Cannot reduce visibility of the inherited method from A***

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

*Simple obj=****new*** *Simple();*

*obj.msg();*

*}*

*}*

**Example:**

***class*** *A{*

***void*** *msg(){*

*System.out.println("Hello java");*

*}*

*}*

***class*** *Simple* ***extends*** *A{*

***void*** *msg(){*

*System.out.println("Hello java");*

*}*

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

*Simple obj=****new*** *Simple();*

*obj.msg();*

*}*

*}***Output:** Hello java

**Constructor overloading**

Constructor overloading means declaring multiple constructors with different method signature.

Main advantage of constructor overloading is flexibility like providing different ways to create an object for a class.

**//wap to demo on constructor overloading**

*class Cube{*

*int l,b,h;*

*Cube(){*

*l=b=h=5;*

*}*

*Cube(int x){*

*l=b=h=x;*

*}*

*Cube(int l,int b,int h){*

*this.l=l;*

*this.b=b;*

*this.h=h;*

*}*

*void area(){*

*System.out.println("Area of Cuboid: "+(l\*b\*h));*

*}*

*}*

*class Polymorphism7{*

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

*Cube c1 = new Cube();*

*c1.area();*

*Cube c2 = new Cube(7);*

*c2.area();*

*Cube c3 = new Cube(2,5,4);*

*c3.area();*

*}*

*}*

**Note:**

1. Constructor overloading must be done with in the same class

2. When we write parameter constructor then it is recommended to write 0 parametrized constructor as well.

# ExceptionHandling with MethodOverriding in Java

|  |
| --- |
| There are many rules if we talk about methodoverriding with exception handling.  **The Rules are as follows:**   * **If the superclass method does not declare an exception**   + If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception. * **If the superclass method declares an exception**   + If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception. |

### If the super class method does not declare an exception

#### Rule: If the super class method does not declare an exception, subclass overridden method cannot declare the checked exception.

***import****java.io.\*;*

***class****Parent{*

***void****msg(){*

*System.out.println("parent");*

*}*

*}*

***class****TestExceptionChild****extends****Parent{*

***void****msg()****throws****IOException{*

*System.out.println("TestExceptionChild");*

*}*

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

*Parent p=****new****TestExceptionChild();*

*p.msg();*

*}*

*}*

**Output:** Compile Time Error

#### 2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

***import****java.io.\*;*

***class****Parent{*

***void****msg(){*

*System.out.println("parent");*

*}*

*}*

***class****TestExceptionChild1****extends****Parent{*

***void****msg()****throws****ArithmeticException{*

*System.out.println("child");*

*}*

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

*Parent p=****new****TestExceptionChild1();*

*p.msg();*

*}*

*}*

**Output:** child

### If the superclass method declares an exception

#### 1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

### Example in case subclass overridden method declares parent exception

***import****java.io.\*;*

***class****Parent{*

***void****msg()****throws****ArithmeticException{*

*System.out.println("parent");*

*}*

*}*

***class****TestExceptionChild2****extends****Parent{*

***void****msg()****throws****Exception{*

*System.out.println("child");*

*}*

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

*Parent p=****new****TestExceptionChild2();*

***try****{*

*p.msg();*

*}****catch****(Exception e){}*

*}*

*}*

**Output:** Compile Time Error

### Example in case subclass overridden method declares same exception

***import****java.io.\*;*

***class****Parent{*

***void****msg()****throws****Exception{*

*System.out.println("parent");*

*}*

*}*

***class****TestExceptionChild3****extends****Parent{*

***void****msg()****throws****Exception{*

*System.out.println("child");*

*}*

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

*Parent p=****new****TestExceptionChild3();*

***try****{*

*p.msg();*

*}****catch****(Exception e){*

*}*

*}*

*}*

Output: child

### Example in case subclass overridden method declares subclass exception

***import****java.io.\*;*

***class****Parent{*

***void****msg()****throws****Exception{*

*System.out.println("parent");*

*}*

*}*

***class****TestExceptionChild4****extends****Parent{*

***void****msg()****throws****ArithmeticException{*

*System.out.println("child");*

*}*

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

*Parent p=****new****TestExceptionChild4();*

***try****{*

*p.msg();*

*}****catch****(Exception e){*

*}*

*}*

*}*

**Output:** child

### Example in case subclass overridden method declares no exception

***import****java.io.\*;*

***class****Parent{*

***void****msg()****throws****Exception{*

*System.out.println("parent");*

*}*

*}*

***class****TestExceptionChild5****extends****Parent{*

***void****msg(){*

*System.out.println("child");*

*}*

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

*Parent p=****new****TestExceptionChild5();*

***try****{*

*p.msg();*

*}****catch****(Exception e){}*

*}*

*}*

**Output**: child

**Java Dynamic Binding Dynamic Polymorphism**

Static binding and [static polymorphism](http://way2java.com/oops-concepts/using-methods-and-method-overloading/) is achieved through method [overloading](http://way2java.com/oops-concepts/using-methods-and-method-overloading/). Now let us go for dynamic polymorphism. Observe the following code.

**Program of Java Dynamic Binding Dynamic Polymorphism**

*class Bird{*

*public void eat(){ // I*

*System.out.println("All birds eat");*

*}*

*}*

*class Peacock extends Bird{*

*public void eat(){ // II*

*System.out.println("Peacock eats grains");*

*}*

*}*

*class Vulture extends Bird{*

*public void eat(){ // III*

*System.out.println("Vulture eats flesh");*

*}*

*}*

*class Crane extends Bird{*

*public void eat(){ // IV*

*System.out.println("Crane eats fish");*

*}*

*}*

*public class DynamicPolyDemo{*

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

*Bird b1 = new Bird();*

*b1.eat();* ***// calls I***

*Peacock p1 = new Peacock();*

*b1 = p1;* *b1.eat();* ***// calls II***

*Vulture v1 = new Vulture();*

*b1 = v1;*

*b1.eat();* ***// calls III***

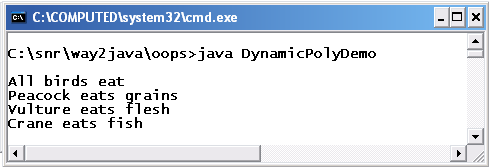
*Crane c1 = new Crane();*

*b1 = c1;*

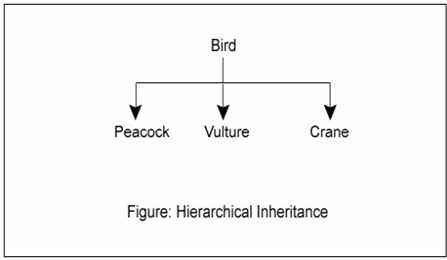
*b1.eat();* ***// calls IV***

*}*

*}*



In the above code, **Bird** is inherited by three classes – **Peacock**, **Vulture** and **Crane**. It is an example of [**hierarchical inheritance**](http://way2java.com/oops-concepts/types-of-inheritance/).

[](http://way2java.com/wp-content/uploads/2011/05/ss55.bmp)

Subclasses **Peacock**, **Vulture** and **Crane** overrides the **eat()**method of Bird.

*b1 = p1;*

*b1.eat();*

In the above statement, subclass Peacock object **p1** is assigned to super class Bird object**b1**. Earlier, in [Object Casting](http://way2java.com/casting-operations/object-casting/), we know **if a subclass object is assigned to a super class object, the super class object will call subclass overridden method**. As per this rule, **b1** will call **p1 eat()**method. Infact, the **b1** contains the reference of **p1**.

*b1 = v1;*

*b1.eat();*

Now, the Bird object **b1** is assigned with the subclass Vulture object **v1**. Now the earlier **p1**reference is replaced by **v1**. Now **b1** points to **v1** location. Now **b1** calls **v1** eat method.

*b1 = c1;*

*b1.eat();*

Again, the **v1** reference in the super class object**b1** is replaced by Crane object **c1**. Now, b1 calls Crane's **eat()** method.

Observe the code, the statement **b1.eat()** prints 4 different outputs by calling 4 different eat() methods. The same method when called different times giving different values is known a**polymorphism**("poly" means **many**and "morphism" means **forms**).

###### **Java Dynamic Binding**

**Dynamic binding** always says create an object of base class but do not create the object of derived classes. Dynamic binding principal is always used for executing polymorphic applications.

The process of binding appropriate versions (overridden method) of derived classes which are inherited from base class with base class object is known as dynamic binding.

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

*Less memory space*

*Less execution time*

*More performance*

Which **eat()**method is to be called is decided at runtime because in every statement the address of the earlier subclass object is replaced. This address replacement happens at runtime (and not at compile time). Which overridden method is to be called is decided (or binded) dynamically at runtime. This feature is known as **dynamic binding**. Because[polymorphism](http://way2java.com/oops-concepts/oops-concepts-introduction/) is achieved dynamically at runtime, this is known as **dynamic polymorphism**.

Dynamic binding is also called as **dynamic method dispatch** as which overridden method is to be dispatched for execution is known at runtime.

###### **Rules followed to achieve dynamic polymorphism**

**1. There must be method overriding.   
2. Subclass object must be assigned to a super class object.**

The rule followed is "**when a subclass object is assigned to a super class object, the super object will call subclass overridden method**".

We know earlier, [static polymorphism](http://way2java.com/oops-concepts/using-methods-and-method-overloading/) which is achieved through method overloading.

###### **Dynamic polymorphism through Interfaces**

The same **Bird** program, in the earlier example, can be modified to suit with interfaces. Now declare **Bird as an interface**. We know that [reference variables](http://way2java.com/oops-concepts/reference-variables-anonymous-objects/) can be created with abstract classes and interfaces (but not objects).

*interface Bird{*

*public abstract void eat(); // I*

*}*

*class Peacock implements Bird{*

*public void eat(){ // II*

*System.out.println("Peacock eats grains");*

*}*

*}*

*class Vulture implements Bird{*

*public void eat(){ // III*

*System.out.println("Vulture eats flesh");*

*}*

*}*

*class Crane implements Bird{*

*public void eat(){ // IV*

*System.out.println("Crane eats fish");*

*}*

*}*

*public class DynamicPolyDemo{*

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

*Bird b1;* ***// reference variable of interface***

*Peacock p1 = new Peacock();*

*b1 = p1;*

*b1.eat();* ***// calls II***

*Vulture v1 = new Vulture();*

*b1 = v1;*

*b1.eat(); //* ***calls III***

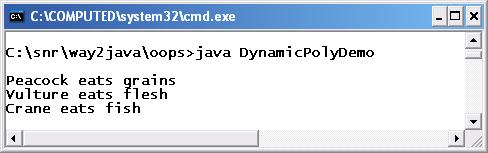
*Crane c1 = new Crane();*

*b1 = c1;*

*b1.eat();* ***// calls IV***

*}*

*}*



Here**Bird is interface**and not a concrete class (in the earlier program it is concrete class).

*Bird b1;*

**b1** is reference variable and not object.

*b1 = p1;*

*b1.eat();*

Subclass Peacock object is assigned to the reference variable **b1** of interface Bird. Here, super class is an interface and eat() is an abstract method. When a subclass object is assigned to a super class reference variable, the reference variable calls subclass overridden method. Now,**b1** calls subclass Peacock's eat() method.

The addresses of subclass objects, **p1**, **v1** and **c1**, in the super class reference variable **b1** are replaced in every statement dynamically at runtime. Every time when**b1.eat(**) method is called, it prints different outputs. This is known as polymorphism. Because [polymorphism](http://way2java.com/oops-concepts/oops-concepts-introduction/) is achieved at runtime dynamically, this is known as **dynamic polymorphism**. Here,**dynamic polymorphism is achieved through interfaces**.

**Question:**

public class Snake{  
 public void swim(){  
 Console.WriteLine(“Swims with the whole body”);  
 }  
}

public class Duck : Snake{  
 public void swim(){  
 Console.WriteLine(“Swims with legs”);  
 }  
}

public class Fish : Duck{  
 public void swim(){  
 Console.WriteLine(“Swims with fins”);  
 }  
}  
public class Animal:Fish{  
 static void Main(string[] args){  
 Snake sn = new Fish ();  
 sn.swim();  
 }  
}

The above call to sn.swim is not calling Fish’s swim() method but rather calling its own method (Snake’s swim)

**Ans:**

Go for hierarchical inheritance where each class extends Snake. You have done multilevel inheritance. Polymorphism works with hierarchical. Your program works with Fish extends Snake.

**Question:**

*Suppose a program..*

*class A{  
 void msg(){  
 sop(“hello A”);  
 }  
}*

*class B extends A{  
 void msg(){  
 sop(“hello B”);  
 }  
}*

*class C extends A{  
 void msg(){  
 sop(“hello C”);  
 }  
}*

*class dispatch{  
 psvm(String a[]){*

*A a=new A();  
 A b=new B();  
 A c=new C();  
 a.msg();  
 b.msg();  
 c.msg();  
 }  
}*

Now plz explain what is the advantage of using polymorphism..we can also make 3 refrence of type A, B and C..then wht is the point of making three refrence of only one type tht of Parent class type..wht is the advantage..plz explain me i m so confuse

**Ans:**

Polymorphism is useful to know what subclass method is to be called at runtime.

**What is the need of dynamic method dispatch?**

* 1. The same method call gets different outputs

**Polymorphism is when we have same method name in same/different classes , do it depends upon by which object we are calling the function from main class ?**

1. It depends on which subclass object is assigned to superclass object.

**What’s the significance of having dynamic polymorphism in Java? What is the exact real life usage for this feature? In this case any way we are creating subclass object & then assigning that to a superclass reference variable. Instead of calling the method through the reference variable of super class we can invoke the same using respective subclass variable itself. I am sure when this feature is there they would have thought for good real life fit for this.**

A.

1. With one class (super class) object, we can get the result required by the user.  
2. User may pass any object he likes.  
3. If you call subclass object, each subclass may have different methods.  
4. When you extend super class, the sub classes will have uniform methods.  
5. The same method of super class, the sub class may fill with its own code.

**How can reference variable of Runnable interface call getClass() of java.lang.Object though getclass Method is not present in Runnable interface?**

A.

The rule followed is “when a subclass object is assigned to a super class object, the super object will call subclass overridden method”.

**Runnable r= new Thread();  
System.out.println(“the class name of r is “+r.getClass());**

Could you explain how r is able to call getClass() as it is not in Runnable interface.

**Program for separate method overloading and method overriding by using constructor..**

First of all constructors cannot be overridden but can be overloaded. You wanted method overloading using constructors. Following is the program.

*public class Demo{  
public Demo(){  
 display();  
 display(10);  
 display(10, 20);  
}  
public void display(){  
 System.out.println(“Hello 1”);  
}  
public void display(int x){  
 System.out.println(x);  
}  
public void display(int x, int y){  
 System.out.println(x\*y);  
}  
public static void main(String args[]){  
 new Demo();  
}  
}*

**Is Java Pass by reference or pass by value:**

For any method both primitive data types and non-primitive data types can be passed as parameters. i.e. primitive data types like int, char, double can be passed as well as non-primitive data types like Student, Book, Bike can be passed as parameters. Passing primitive data types is usually referred as *Pass By Value*, where as passing non-primitive data types is *Pass By Reference*.

*class PassByValue{  
    public static void main(String s[]){  
        int i = 5;  
        System.out.println("i before increment : " + i);  
        increment(i);  
        System.out.println("i after increment : " + i);  
    }  
      
    public static void increment(int i){  
        System.out.println("i before increment in method : " + i);  
        i++;  
        System.out.println("i after increment in method : " + i);  
    }  
}*

**OUTPUT**

*i before increment : 5  
i before increment in method : 5  
i after increment in method : 6  
i after increment : 5*

As we found in the previous program, when the parameter is passed by value, the changed value is not reflected in the calling method. The below program shows how we can pass by reference and get the changed values reflected in the calling method.

*class PassByReference{  
    public static void main(String args[]){  
        Student sMain = new Student();  
        sMain.name = "Jagan";  
        sMain.marks = 93;  
        sMain.section = 'B';  
      
        System.out.println("Marks before incrementing : " + sMain.marks);  
        incrementMarks(sMain);  
        System.out.println("Marks after incrementing : " + sMain.marks);  
    }  
    public static void incrementMarks(Student sMethod){  
        System.out.println("Marks before incrementing in the method: " + sMethod.marks);  
        sMethod.marks = sMethod.marks + 1;  
        System.out.println("Marks after incrementing in the method: " + sMethod.marks);  
    }  
}  
class Student{  
    String name;  
    int marks;  
    char section;  
}*

**OUTPUT**

*Marks before incrementing : 93  
Marks before incrementing in the method: 93  
Marks after incrementing in the method: 94  
Marks after incrementing : 94*

Please note that only when the values in the non primitive data type are changed using the dot(.) operator, the value is reflected in the calling method. If the reference itself is overwritten as shown below (LINE A), then the changes will not be reflected in the main method. This is because, as discussed in [Java Objects References](http://java.meritcampus.com/core-java-topics/java-objects-references), when the reference itself is overwritten, the object it points to is changed. But the reference in the main method still points to the old object and any changes done to the new object will not impact the old object.

*public static void incrementMarks(Student sMethod){  
    System.out.println("Marks before incrementing in the method: " + sMethod.marks);  
    sMethod = new Student(); // LINE A  
    sMethod.name = "Murugan";  
    sMethod.marks = 75;  
    sMethod.section = 'A';  
    System.out.println("Marks after incrementing in the method: " + sMethod.marks);  
}*

**Method Overriding Example:**

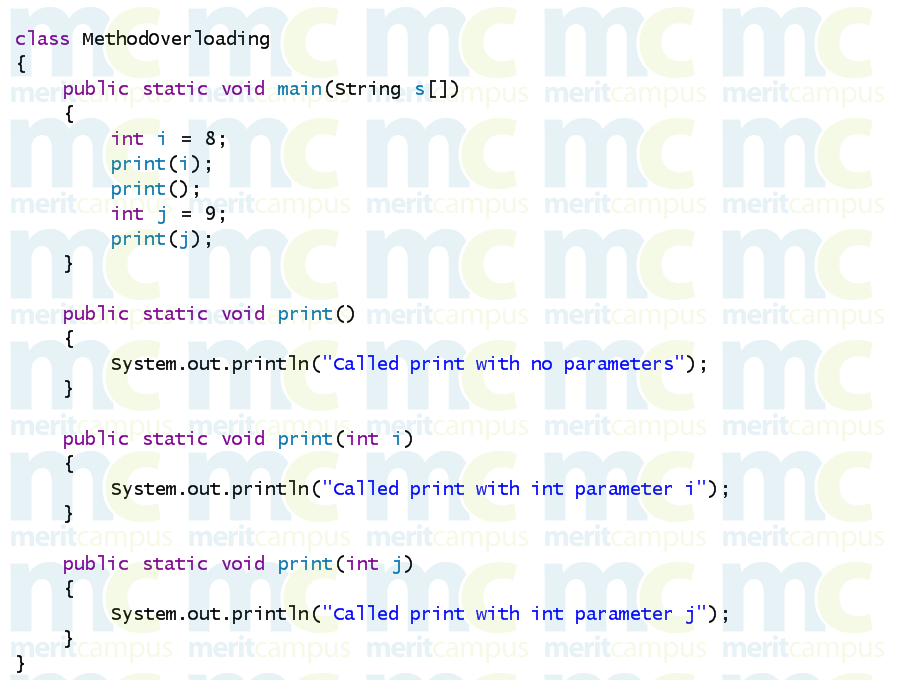
*class IceCreamPricesWithOverriding{  
    public static void main(String arg[]){  
        IceCream ic = new IceCream();  
        ic.flavor = "Pista";  
        ic.numberOfScoops = 2;  
         System.out.println(ic.numberOfScoops + " scoops of " + ic.flavor + " flavor price is : " + ic.getPrice());  
   
       FruitSaladWithIceCream fs = new FruitSaladWithIceCream();  
        fs.flavor = "Chocolate";  
        fs.numberOfScoops = 1;  
        fs.gramsOfFruitSalad = 50;  
          
        System.out.print(fs.gramsOfFruitSalad + " grams of fruit salad and ");  
         System.out.println(fs.numberOfScoops + " scoops of " + fs.flavor + " flavor price is : " + fs.getPrice());  
          
        KhubaniKaMeetaWithIceCream kkm = new KhubaniKaMeetaWithIceCream();  
        kkm.flavor = "Vanila";  
        kkm.numberOfScoops = 1;  
        kkm.gramsOfKhubaniKaMeeta = 75;  
          
        System.out.print(kkm.gramsOfKhubaniKaMeeta + " grams of khubani ka meeta and ");  
        System.out.println(kkm.numberOfScoops + " scoops of " + kkm.flavor + " flavor price is : " + kkm.getPrice());*

*}  
}  
class IceCream{  
    String flavor;  
    int numberOfScoops;  
    double getPrice(){  
        double pricePerScoop = 35.0;  
        return numberOfScoops \* pricePerScoop;  
    }  
}  
class FruitSaladWithIceCream extends IceCream{  
    int gramsOfFruitSalad;  
    double getPrice(){  
        double iceCreamPrice = super.getPrice(); // LINE A  
        double pricePerGram = 0.75;  
        return gramsOfFruitSalad \* pricePerGram + iceCreamPrice;  
    }  
}  
class KhubaniKaMeetaWithIceCream extends IceCream{  
    int gramsOfKhubaniKaMeeta;  
    double getPrice(){  
        double iceCreamPrice = super.getPrice(); // LINE B  
        double pricePerGram = 1.25;  
        return gramsOfKhubaniKaMeeta \* pricePerGram + iceCreamPrice;  
    }  
}*

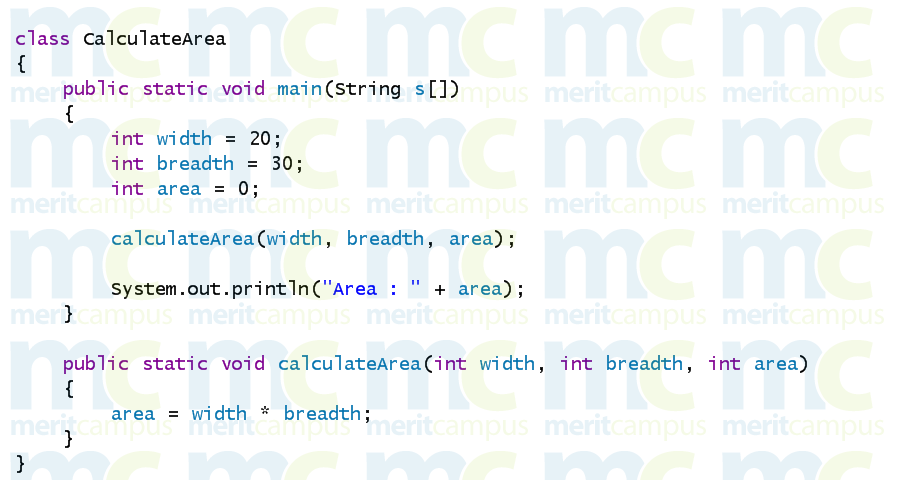
**OUTPUT**

2 scoops of Pista flavor price is : 70.0  
50 grams of fruit salad and 1 scoops of Chocolate flavor price is : 72.5  
75 grams of khubani ka meeta and 1 scoops of Vanila flavor price is : 128.75

**What will be the output of the following program?**



What will be the output of the following program.



What will be the output of the following program?



**How a method can be Overridden in different ways ?**

*class MultiLevelMethodOverriding{  
    public static void main(String arg[]){  
        System.out.println("----------------------------");  
        A a = new A();  
        a.print();  
        System.out.println("----------------------------");  
        B b = new B();  
        b.print();  
        System.out.println("----------------------------");  
        C c = new C();  
        c.print();  
        System.out.println("----------------------------");  
        D d = new D();  
        d.print();  
        System.out.println("----------------------------");  
        E e = new E();  
        e.print();  
        System.out.println("----------------------------");      
    }  
}  
class A{  
    void print(){  
        System.out.println("In class A");  
    }  
}  
class B extends A{  
    void print() {  
        super.print();  
        System.out.println("In class B");  
    }  
}  
class C extends B{  
    void print(){  
        super.print();  
        System.out.println("In class C");  
    }  
}  
class D extends A{  
    void print() {  
        System.out.println("In class D, printing before super class A");  
        super.print();  
    }  
}  
class E extends A{  
    void print(){  
        System.out.println("In class E, not calling the super class method at all.");  
    }  
}*

**OUTPUT**

*----------------------------  
In class A  
----------------------------  
In class A  
In class B  
----------------------------  
In class A  
In class B  
In class C  
----------------------------  
In class D, printing before super class A  
In class A  
----------------------------  
In class E, not calling the super class method at all.  
----------------------------*

**The differences between method overloading and method overriding are:**

|  |  |
| --- | --- |
| Method Overloading | Method Overriding |
| This is compile time polymorphism. i.e. which method needs to be called is decided at the compile time itself. | This is run time polymorphism. i.e.  which method needs to be called will  be decided only at run time. |
| The method signature is not same. The method name is same, but the parameter types and the return type are different. | The method signature is exactly the same.  The method name, the parameter types and  the return type have to be same. |
| We can implement this concept with only one class. | We need at least two classes, one extending  from the other to implement this concept. i.e.  We need inheritance, to override a method. |

**Note:**

When methods have same name, but different parameter list and different return type, then it is known as **Overloading**.

**Runtime Polymorphism**

Polymorphism is one of three *object oriented concepts*. The other two are abstraction and inheritance. Run time polymorphism is a way to decide which method to call depending upon the state (variables) of the program.

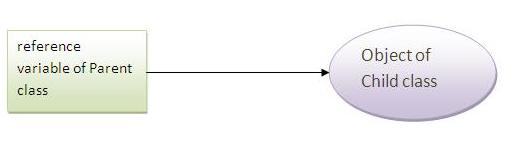
**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

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

Let's first understand the upcasting before Runtime Polymorphism.

### Upcasting

When reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



**class** A{}

**class** B **extends** A{}

--------------

A a=**new** B();//upcasting

### Example of Java Runtime Polymorphism

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

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

***class****Bike{*

***void****run(){System.out.println("running");}*

*}*

***class****Splender****extends****Bike{*

***void****run(){System.out.println("running safely with 60km");}*

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

*Bike b =****new****Splender();//upcasting*

*b.run();*

*}*

*}*

**Output**: running safely with 60km.

**The following program demonstrates the run time polymorphism.**

*class RunTimePolymorphism{  
    public static void main(String arg[]){  
        int input = 24;  
        A aref;   
          
        if(input < 10){  
            aref = new A();  
        }else if(input < 30){  
            aref = new B();  
        }else{  
            aref = new C();  
        }  
        aref.print(); // LINE A  
    }  
}  
class A{  
    void print(){  
        System.out.println("class A method called.");  
    }  
}  
class B extends A{  
    void print(){  
        System.out.println("class B method called.");  
    }  
}  
class C extends A{  
    void print(){  
        System.out.println("class C method called.");  
    }  
}*

**OUTPUT**

class B method called.

### Java Runtime Polymorphism with data member

|  |
| --- |
| Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members. |
| In the example given below, both the classes have a datamember speedlimit, we are accessing the datamember by the reference variable of Parent class which refers to the subclass object. Since we are accessing the datamember which is not overridden, hence it will access the datamember of Parent class always. |

#### Rule: Runtime polymorphism can't be achieved by data members.

***class****Bike{*

***int****speedlimit=90;*

*}*

***class****Honda3****extends****Bike{*

***int****speedlimit=150;*

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

*Bike obj=****new****Honda3();*

*System.out.println(obj.speedlimit);//90*

*}*

***Output:****90*

## Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

***class****Animal{*

***void****eat(){System.out.println("eating");}*

*}*

***class****Dog****extends****Animal{*

***void****eat(){System.out.println("eating fruits");}*

*}*

***class****BabyDog****extends****Dog{*

***void****eat(){System.out.println("drinking milk");}*

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

*Animal a1,a2,a3;*

*a1=****new****Animal();*

*a2=****new****Dog();*

*a3=****new****BabyDog();*

*a1.eat();*

*a2.eat();*

*a3.eat();*

*}*

*}*

**Output:** eating

eating fruits

drinking Milk

**Example:**

**class** Animal{

**void** eat(){System.out.println("animal is eating...");}

}

**class** Dog **extends** Animal{

**void** eat(){System.out.println("dog is eating...");}

}

**class** BabyDog1 **extends** Dog{

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

Animal a=**new** BabyDog1();

a.eat();

}}

**Output: Dog is eating**

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

**Dynamic method dispatch:**

The advantage of overridden methods is that we can achieve run-time polymorphism. We can call the same method, but get different outputs depending upon the type of object.

**The following program can be used to demonstrate this.**

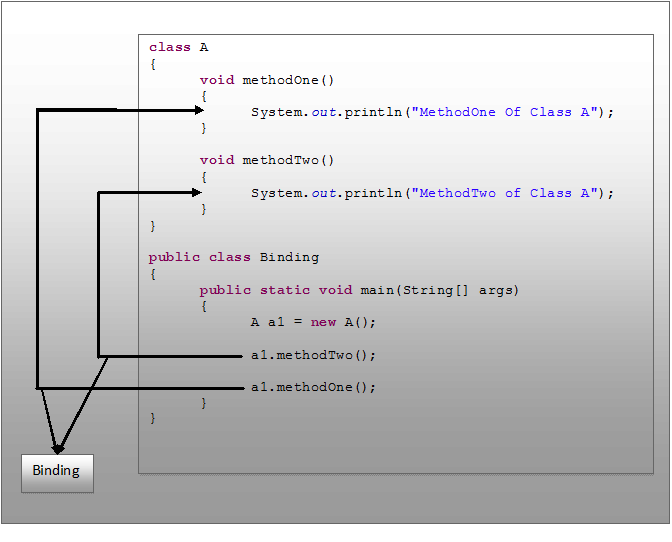
*class CallOverriddenMethods{  
    public static void main(String arg[]){  
        A a = new A();  
        a.print();  
        B b = new B();  
        b.print();  
        C c = new C();  
        c.print();  
        System.out.println("-----------------");  
        A a1 = b; // LINE A  
        System.out.println("After assigning B's object to A's reference and calling the print method on A's reference");  
        a1.print();  
        System.out.println("-----------------");  
        B b1 = c; // LINE B  
        System.out.println("After assigning C's object to B's reference and calling the print method on B's reference");  
        b1.print();  
        System.out.println("-----------------");  
        A a2 = c; // LINE C  
        System.out.println("After assigning C's object to A's reference and calling the print method on C's reference");  
        a2.print();  
        System.out.println("-----------------");      
    }  
}  
class A{  
    void print(){  
        System.out.println("Print method in class A called");  
    }  
}  
class B extends A{  
    void print(){  
        System.out.println("Print method in class B called");  
    }  
}  
class C extends B{  
    void print(){  
        System.out.println("Print method in class C called");  
    }  
}*

**OUTPUT**

*Print method in class A called  
Print method in class B called  
Print method in class C called  
-----------------  
After assigning B's object to A's reference and calling the print method on A's reference  
Print method in class B called  
-----------------  
After assigning C's object to B's reference and calling the print method on B's reference  
Print method in class C called  
-----------------  
After assigning C's object to A's reference and calling the print method on C's reference  
Print method in class C called  
-----------------*

# [Difference Between Static Binding And Dynamic Binding In Java](http://javaconceptoftheday.com/static-binding-and-dynamic-binding-in-java/)

**Binding** refers to the link between method call and method definition. This picture clearly shows what is binding.



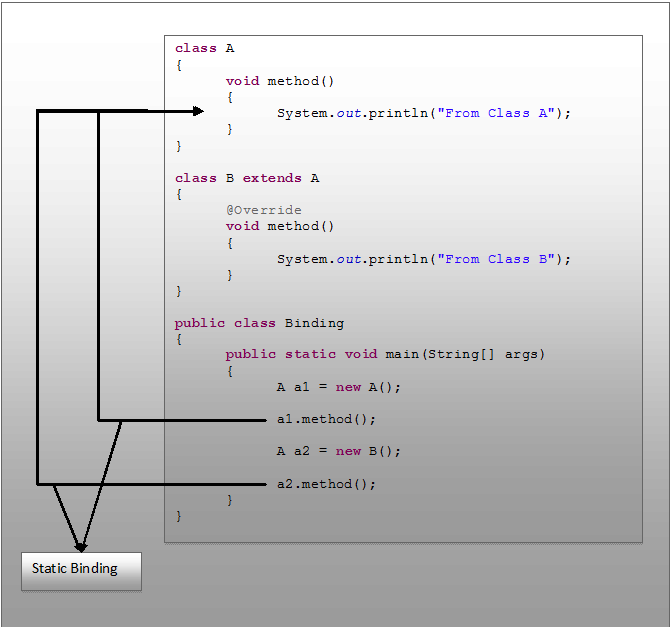
In this picture, **“a1.methodOne()”** call is binding to corresponding **methodOne()** definition and **“a1.methodTwo()”** call is binding to corresponding **methodTwo()** definition.

For every method call there should be proper method definition. This is a rule in java. If compiler does not see the proper method definition for every method call, it throws error.

Now, come to static binding and dynamic binding in java.

**Static Binding In Java :**

**Static binding** is a binding which happens during **compilation**. It is also called **early binding** because binding happens before a program actually runs.

Static binding can be demonstrated like in the below picture.  


In this picture, **‘a1’** is a reference variable of type Class A pointing to object of class A.  **‘a2’** is also reference variable of type class A but pointing to object of Class B.

During compilation, while binding, compiler does not check the type of object to which a particular reference variable is pointing. It just checks the type of reference variable through which a method is called and checks whether there exist a method definition for it in that type.

For example, for **“a1.method()”** method call in the above picture, compiler checks whether there exist method definition for **method()** in Class A. Because ‘**a1′** is Class A type. Similarly, for **“a2.method()”** method call, it checks whether there exist method definition for **method()**in Class A. Because ‘**a2′** is also Class A type. It does not check to which object, **‘a1’** and **‘a2’** are pointing. This type of binding is called **static binding**.

If there is any private, final or static method in a class, there is static binding.

### Example of static binding

***class****Dog{*

***private******void****eat(){System.out.println("dog is eating...");}*

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

*Dog d1=****new****Dog();*

*d1.eat();*

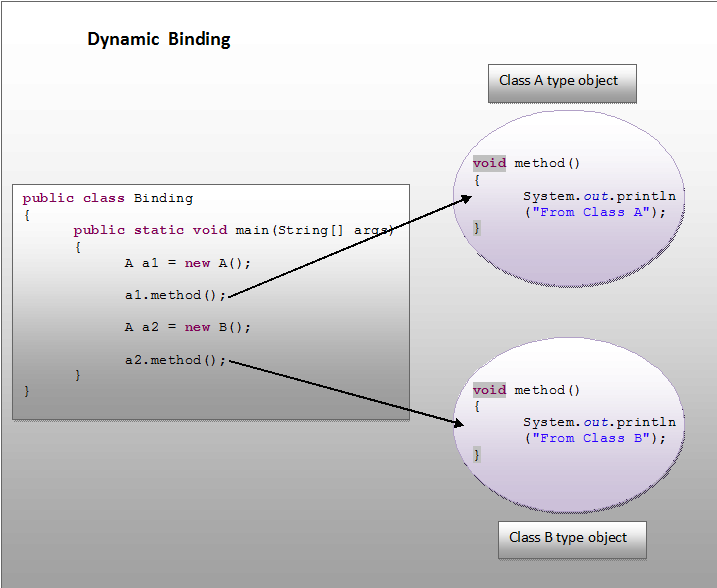
*}*

*}*

**Dynamic Binding In Java :**

**Dynamic binding** is a binding which happens during **run time**. It is also called **late binding** because binding happens when program actually is running.

During run time actual objects are used for binding. For example, for **“a1.method()”** call in the above picture, **method()** of actual object to which **‘a1’** is pointing will be called. For **“a2.method()”** call, **method()** of actual object to which **‘a2’** is pointing will be called. This type of binding is called dynamic binding.

The dynamic binding of above example can be demonstrated like below.  


### Example of dynamic binding

***class****Animal{*

***void****eat(){*

*System.out.println("animal is eating...");*

*}*

*}*

***class****Dog****extends****Animal{*

***void****eat(){*

*System.out.println("dog is eating...");*

*}*

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

*Animal a=****new****Dog();*

*a.eat();*

*}*

*}*

**Differences Between Static Binding And Dynamic Binding In Java :**

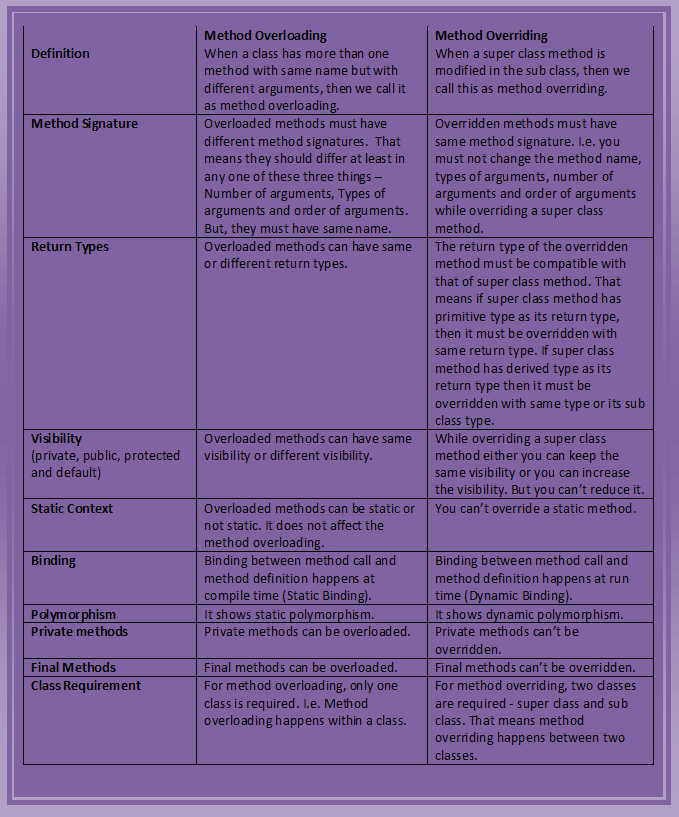
The above findings can be summarized like below.

|  |  |
| --- | --- |
| **Static Binding** | **Dynamic Binding** |
| It is a binding that happens at compile time. | It is a binding that happens at run time. |
| Actual object is not used for binding. | Actual object is used for binding. |
| It is also called early binding because binding happens during compilation. | It is also called late binding because binding happens at run time. |
| Method overloading is the best example of static binding. | Method overriding is the best example of dynamic binding. |
| Private, static and final methods show static binding. Because, they can not be overridden. | Other than private, static and final methods show dynamic binding. Because, they can be overridden. |

# [Difference Between Method Overloading And Method Overriding In Java](http://javaconceptoftheday.com/difference-between-method-overloading-and-method-overriding-in-java/)

Method overloading and method overriding are two important java concepts which allows java programmer to define the methods with same name but different behavior. Both method overloading and method overriding shows polymorphism. It is also one of the most asked java interview question for freshers. In this article, I have tried to list out the differences between method overloading and method overriding in java.

|  |  |
| --- | --- |
| **Method Overloading** | **Method Overriding** |
| **Definition** | When a class has more than one method with same name but with different arguments, then we call it as method overloading. | When a super class method is modified in the sub class, then we call this as method overriding. |
| **Method Signature** | Overloaded methods must have different method signatures.  That means they should differ at least in any one of these three things – Number of arguments, Types of arguments and order of arguments. But, they must have same name. | Overridden methods must have same method signature. I.e. you must not change the method name, types of arguments, number of arguments and order of arguments while overriding a super class method. |
| **Return Types** | Overloaded methods can have same or different return types. | The return type of the overridden method must be compatible with that of super class method. That means if super class method has primitive type as its return type, then it must be overridden with same return type. If super class method has derived type as its return type then it must be overridden with same type or its sub class type. |
| **Visibility**(private, public, protected and default) | Overloaded methods can have same visibility or different visibility. | While overriding a super class method either you can keep the same visibility or you can increase the visibility. But you can’t reduce it. |
| **Static Context** | Overloaded methods can be static or not static. It does not affect the method overloading. | You can’t override a static method. |
| **Binding** | Binding between method call and method definition happens at compile time (Static Binding). | Binding between method call and method definition happens at run time (Dynamic Binding). |
| **Polymorphism** | It shows static polymorphism. | It shows dynamic polymorphism. |
| **Private methods** | Private methods can be overloaded. | Private methods can’t be overridden. |
| **Final Methods** | Final methods can be overloaded. | Final methods can’t be overridden. |
| **Class Requirement** | For method overloading, only one class is required. I.e. Method overloading happens within a class. | For method overriding, two classes are required – super class and sub class. That means method overriding happens between two classes. |



**Method Overloading Example :**

|  |
| --- |
| *public class MainClass{*  *static String concateString(String s1, String s2){*  *return s1+s2;*  *}*  *static String concateString(String s1, String s2, String s3){*  *return s1+s2+s3;*  *}*  *static String concateString(String s1, String s2, String s3, String s4){*  *return s1+s2+s3+s4;*  *}*  *public static void main(String[] args){*  *concateString("ONE", "TWO");*  *concateString("ONE", "TWO", "THREE");*  *concateString("ONE", "TWO", "THREE", "FOUR");*  *}*  *}* |

**Method Overriding Example :**

*class SuperClass{*

*void SuperClassMethod(){*

*System.out.println("SUPER CLASS METHOD");*

*}*

*}*

*class SubClass extends SuperClass{*

*@Override*

*void SuperClassMethod(){*

*System.out.println("SUPER CLASS METHOD IS OVERRIDDEN");*

*}*

*}*

# [Interview Questions On Method Overloading](http://javaconceptoftheday.com/important-java-interview-questions-on-method-overloading/)

**1) What is method overloading?**

When a class has more than one method with same name but different parameters, then we call those methods are overloaded. Overloaded methods will have same name but different number of arguments or different types of arguments.

**2) What is method signature? What are the things it consist of?**

Method signature is used by the compiler to differentiate the methods. Method signature consist of three things.

a) Method name

b) Number of arguments

c) Types of arguments

**3) Can we declare one overloaded method as static and another one as non-static?**

Yes. Overloaded methods can be either static or non static.

**4) How do compiler differentiate overloaded methods from duplicate methods?**

Compiler uses method signature to check whether the method is overloaded or duplicated. Duplicate methods will have same method signatures i.e same name, same number of arguments and same types of arguments. Overloaded methods will also have same name but differ in number of arguments or else types of arguments.

**5) Is it possible to have two methods in a class with same method signature but different return types?**

No, compiler will give duplicate method error. Compiler checks only method signature for duplication not the return types. If two methods have same method signature, straight away it gives compile time error.

**6) In “MyClass” , there is a method called “myMethod” with four different overloaded forms. All four different forms have different visibility ( private, protected, public and default). Is “myMethod” properly overloaded?**

Yes. Compiler checks only method signature for overloading of methods not the visibility of methods.

**7) Can overloaded methods be synchronized?**

Yes. Overloaded methods can be synchronized.

**8) Can we overload main() method?**

Yes, we can overload main() method. A class can have any number of main() methods but execution starts from **public static void main(String[] args)** only.

**9) Can we declare overloaded methods as final?**

Yes, we can declare overloaded methods as final.

**10) In the below class, is constructor overloaded or is method overloaded?**

|  |  |
| --- | --- |
|  | *public class A{*  *public A(){*  *//-----> (1)*  *}*  *void A(){*  *//-----> (2)*  *}*  *}* |

None of them. It is neither constructor overloaded nor method overloaded. First one is a constructor and second one is a method.

**11) Overloading is the best example of dynamic binding. True or false?**

False. Overloading is the best example for static binding. (Click [here](http://javaconceptoftheday.com/difference-between-static-binding-and-dynamic-binding/) to see what is static binding and what is dynamic binding)

**12) Can overloaded method be overrided?**

Yes, we can override a method which is overloaded in super class.

# [Interview Questions On Method Overriding](http://javaconceptoftheday.com/java-interview-questions-on-method-overriding/)

**1) What is method overriding?**

Modifying a super class method in the sub class is called method overriding. Using method overriding, we can change super class method according to the requirements of sub class.

**2) What are the rules to be followed while overriding a method?**

There are 5 main rules you should kept in mind while overriding a method. They are,

a) Name of the method must be same as that of super class method.

b) Return type of overridden method must be compatible with the method being overridden. i.e if a method has primitive type as it’s return type then it must be overridden with primitive type only and if a method has derived type as it’s return type then it must be overridden with same type or it’s sub class types.

c) You must not reduce the visibility of a method while overriding.

d) You must not change parameter list of a method while overriding.

e) You cannot increase the scope of exceptions while overriding a method with throws clause.

**3) Can we override static methods?**

No, Static methods cannot be overridden. If we try to override them they will be hidden in the sub class.

**4) What happens if we change the arguments of overriding method?**

If we change the arguments of overriding method, then that method will be treated as overloaded not overridden.

**5) Can we override protected method of super class as public method in the sub class?**

Yes. You can increase the visibility of overriding methods but can’t reduce it.

**6) Can we change the return type of overriding method from Number type to Integer type?**

Yes. You can change as Integer is a sub class of Number type.

**7) Can we override a super class method without throws clause as a method with throws clause in the sub class?**

Yes, but only with unchecked type of exceptions.

**8) Can we change an exception of a method with throws clause from SQLException to NumberFormatException while overriding it?**

Yes. Overridden method may throw SQLException or it’s sub class exception or any unchecked type of exceptions.

**9) Can we change an exception of a method with throws clause from unchecked to checked while overriding it?**

No. We can’t change an exception of a method with throws clause from unchecked to checked.

**10) How do you refer super class version of overridden method in the sub class?**

Using super keyword, we can refer super class version of overridden method in the sub class.

**11) Can we override private methods?**

No question of overriding private methods. They are not at all inherited to sub class.

**12) Can we remove throws clause of a method while overriding it?**

Yes. You can remove throws clause of a method while overriding it.

**13) Is it possible to override non-static methods as static?**

No. You can’t override non-static methods as static.

**14) Can we change an exception of a method with throws clause from checked to unchecked while overriding it?**

Yes. We can change an exception from checked to unchecked but reverse is not possible.

**15) Can we change the number of exceptions thrown by a method with throws clause while overriding it?**

Yes, we can change. But, exceptions must be compatible with throws clause in the super class method.

# [Java Practice Questions On Method Overloading And Overriding](http://javaconceptoftheday.com/java-practice-questions-on-method-overloading-and-overriding/)

**1) What will be the output of the following program?**

*class A*

*{*

*}*

*class B extends A*

*{*

*}*

*class C extends B*

*{*

*}*

*public class MainClass {*

*static void overloadedMethod(A a){*

*System.out.println("ONE");*

*}*

*static void overloadedMethod(B b){*

*System.out.println("TWO");*

*}*

*static void overloadedMethod(Object obj){*

*System.out.println("THREE");*

*}*

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

*C c = new C();*

*overloadedMethod(c);*

*}*

*}*

**Answer :**  
TWO

**2) In a class, one method has two overloaded forms. One form is defined as static and another form is defined as non-static. Is that method properly overloaded?**

**Answer :**  
Yes. Compiler checks only method signature to verify whether a particular method is properly overloaded or not. It doesn’t check static or non-static feature of the method.

**3) In the below class, is ‘method’ overloaded or duplicated?**

*public class MainClass*

*{*

*void method(int ... a)*

*{*

*System.out.println(1);*

*}*

*void method(int[] a)*

*{*

*System.out.println(2);*

*}*

*}*

**Answer :**  
Duplicated. Because, var args (int … a) are nothing but the arrays. So here, (int … a) and (int[] a) are the same.

**4) Method signature consists of**

a) Method Name, Return Type and Number Of Arguments  
b) Access Modifier, Method Name and Types Of Arguments  
c) Method Name, Number Of Arguments, Types Of Arguments and Order Of Arguments  
d) Return Type, Access Modifier and Order Of Arguments

**Answer :**  
**c) Method Name, Number Of Arguments, Types Of Arguments and Order Of Arguments**

**5) In the below Class X, is ‘method’ properly overloaded?**

*class X*

*{*

*int method(int i, int d)*

*{*

*return i+d;*

*}*

*static int method(int i, double d)*

*{*

*return (int)(i+d);*

*}*

*double method(double i, int d)*

*{*

*return i+d;*

*}*

*static double method(double i, double d)*

*{*

*return i+d;*

*}*

*}*

**Answer :**  
Yes.

**6) What will be the output of the following program?**

*class X{*

*void method(int a){*

*System.out.println("ONE");*

*}*

*void method(double d){*

*System.out.println("TWO");*

*}*

*}*

*class Y extends X{*

*@Override*

*void method(double d){*

*System.out.println("THREE");*

*}*

*}*

*public class MainClass {*

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

*new Y().method(100);*

*}*

*}*

**Answer :**  
ONE

**7) What will be the outcome of the below program?**

*public class MainClass {*

*double overloadedMethod(double d){*

*return d \*= d;*

*}*

*int overloadedMethod(int i){*

*return overloadedMethod(i \*= i);*

*}*

*float overloadedMethod(float f){*

*return overloadedMethod(f \*= f);*

*}*

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

*MainClass main = new MainClass();*

*System.out.println(main.overloadedMethod(100));*

*}*

*}*

**Answer :**  
It will throw java.lang.StackOverflowError at run time. Because, overloadedMethod(int) keeps calling itself.

**8) Method Overriding shows static polymorphism. True or false?**

**Answer :**  
False. Method Overriding shows dynamic polymorphism.

**9) In a class, One method has 4 overloaded forms. All have different access modifiers (private, default, protected and public). Is that method properly overloaded?**

**Answer :**  
Yes. That method is properly overloaded.

**10) What will be the outcome of the following program?**

*class X{*

*void calculate(int a, int b){*

*System.out.println("Class X");*

*}*

*}*

*class Y extends X{*

*@Override*

*void calculate(int a, int b){*

*System.out.println("Class Y");*

*}*

*}*

*class Z extends Y{*

*@Override*

*void calculate(int a, int b) {*

*System.out.println("Class Z");*

*}*

*}*

*public class MainClass {*

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

*X x = new Y();*

*x.calculate(10, 20);*

*Y y = (Y) x;*

*y.calculate(50, 100);*

*Z z = (Z) y;*

*z.calculate(100, 200);*

*}*

*}*

**Answer :**  
Line 39 (Z z = (Z) y) will throw java.lang.ClassCastException at run time. Because, Y cannot be cast to Z.

**11) Will you find out the error in the below code?**

*class X{*

*static void methodOfX(){*

*System.out.println("Class X");*

*}*

*}*

*class Y extends X{*

*@Override*

*static void methodOfX(){*

*System.out.println("Class X");*

*}*

*}*

**Answer :**  
Can’t override static methods.

**12) What possible types a ‘superClassMethod()’ of below ‘SuperClass’ can return when it is overridden in the sub class?**

|  |
| --- |
| *class SuperClass{*  *Object superClassMethod(){*  *return new Object();*  *}*  *}* |

**Answer :**  
Any type. Because Object class is a super class for all the classes in Java.

**13) Can we override protected method as private?**

**Answer :**  
No. While overriding, visibility of a method can be increased but can not be reduced.

**14) What will be the output of this program?**

*class SuperClass{*

*void superClassMethod(Number n){*

*System.out.println("From Super Class");*

*}*

*}*

*class SubClass extends SuperClass{*

*void superClassMethod(Double d){*

*System.out.println("From Sub Class");*

*}*

*}*

*public class MainClass {*

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

*SubClass sub = new SubClass();*

*sub.superClassMethod(123321);*

*}*

*}*

**Answer :**  
From Super Class

**15) What actually polymorphism means in Java?**

**Answer :**  
Polymorphism in java refers to any entity whether it is an operator or a constructor or any method which takes many forms or can be used for multiple tasks, either while compiling or running a java program.

**16) Does Java support operator overloading?**

**Answer :**  
Java doesn’t support operator overloading. (exception being ‘+’ symbol which is used for both addition of two numbers as well as for concatenation of two strings.)

**17) What will be the output of the below program?**

*class A{*

*public A(int i){*

*System.out.println(1);*

*}*

*public A(){*

*this(10);*

*System.out.println(2);*

*}*

*void A(){*

*A(10);*

*System.out.println(3);*

*}*

*void A(int i){*

*System.out.println(4);*

*}*

*}*

*public class MainClass {*

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

*new A().A();*

*}*

*}*

**Answer :**  
1  
2  
4  
3

**18) Why method overriding is called late binding or dynamic binding?**

**Answer :**  
Because object used for binding will be known only at run time.

**19) What will be the output of the following program?**

*public class MainClass {*

*static void method(Integer i){*

*System.out.println(1);*

*}*

*static void method(Double d){*

*System.out.println(2);*

*}*

*static void method(Number n){*

*System.out.println(4);*

*}*

*static void method(Object o){*

*System.out.println(5);*

*}*

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

*method((short)12);*

*}*

*}*

**Answer :**  
4

**20) How do compiler differentiate overloaded methods from duplicate methods?**

**Answer :**  
Compiler uses method signature to check whether the method is overloaded or duplicated. Duplicate methods will have same method signatures i.e same name, same number of arguments and same types of arguments. Overloaded methods will have same name but differ in number of arguments or in types of arguments.

**21) final methods can be overridden but can’t be overloaded? True or False?**

**Answer :**  
False. final methods can be overloaded but can’t be overridden.

**22) Does the below program shows polymorphism or not?**

*class A{*

*}*

*class B extends A{*

*}*

*class C extends B{*

*}*

*public class MainClass{*

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

*A a = new A();*

*a = new B();*

*a = new C();*

*}*

*}*

**Answer :**  
Yes. Class-A type reference variable is referring to Class-A type object, Class-B type object and Class-C type object. This shows the polymorphism.

**23) What will be the output of the following program?**

*class X{*

*int method(int i){*

*return i \*= i;*

*}*

*}*

*class Y extends X{*

*double method(double d){*

*return d /= d;*

*}*

*}*

*class Z extends Y{*

*float method(float f){*

*return f += f;*

*}*

*}*

*public class MainClass{*

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

*Z z = new Z();*

*System.out.println(z.method(21.12));*

*}*

*}*

**Answer :**  
1.0

**24) What will be the output of the following program?**

*class ClassOne{*

*void method(String s1){*

*method(s1, s1+s1);*

*}*

*void method(String s1, String s2){*

*method(s1, s2, s1+s2);*

*}*

*void method(String s1, String s2, String s3){*

*System.out.println(s1+s2+s3);*

*}*

*}*

*public class MainClass{*

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

*ClassOne one = new ClassOne();*

*one.method("JAVA");*

*}*

*}*

**Answer :**  
JAVAJAVAJAVAJAVAJAVAJAVA

**25) Constructor overloading is also one form of the polymorphism. Yes or No?**

**Answer :**  
Yes. Constructor overloading is also one form of polymorphism.

**26) Is the following program written correctly? If yes, what will be the output?**

*class A{*

*public A(int i){*

*System.out.println(myMethod(i));*

*}*

*int myMethod(int i){*

*return ++i + --i;*

*}*

*}*

*class B extends A{*

*public B(int i, int j){*

*super(i\*j);*

*System.out.println(myMethod(i, j));*

*}*

*int myMethod(int i, int j){*

*return myMethod(i\*j);*

*}*

*}*

*public class MainClass{*

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

*B b = new B(12, 21);*

*}*

*}*

**Answer :**  
505  
505

**27) What will be the output of the below program?**

*class A{*

*void myMethod(Object o, Double D){*

*System.out.println(1);*

*}*

*void myMethod(Integer I, Number N){*

*System.out.println(2);*

*}*

*}*

*class B extends A{*

*void myMethod(Float F, Double D){*

*System.out.println(3);*

*}*

*void myMethod(Double D, Integer I){*

*System.out.println(4);*

*}*

*}*

*public class MainClass{*

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

*B b = new B();*

*b.myMethod(11.11, 0000);*

*b.myMethod(8778, 3223);*

*b.myMethod(2.3\*1.2, 4.1\*1.4);*

*b.myMethod((float)23.56, 21.45);*

*}*

*}*

**Answer :**  
4  
2  
1  
3

**28) In the below example, Class B extends Class A. Which method of Class A is not properly overridden in class B?**

*class A{*

*void methodOne(Double D){*

*}*

*int methodTwo(Integer I){*

*return I;*

*}*

*}*

*class B extends A{*

*@Override*

*void methodOne(double d){*

*}*

*@Override*

*int methodTwo(Integer I){*

*return (int)1.1;*

*}*

*}*

**Answer :**  
methodOne() is not properly overridden. Because, arguments are not compatible.

**29) Can we access super class version of overridden method in the sub class. If yes, how?**

**Answer :**  
Yes. We can access super class version of overridden method in the sub class using **super** keyword.

**31) What will be the output of the following program?**

*class A{*

*static void methodOne(){*

*System.out.println("AAA");*

*}*

*}*

*class B extends A{*

*static void methodOne(){*

*System.out.println("BBB");*

*}*

*}*

*public class MainClass{*

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

*A a = new B();*

*a.methodOne();*

*}*

*}*

**Answer :**  
AAA

**32) In the below class A, ‘myMethod()’ has three different forms. All are throwing different exceptions, but have same signature. Is it OK?**

*class A{*

*void myMethod() throws IOException{*

*System.out.println("ONE");*

*}*

*void myMethod() throws NumberFormatException{*

*System.out.println("TWO");*

*}*

*void myMethod() throws ArrayIndexOutOfBoundsException{*

*System.out.println("THREE");*

*}*

*}*

**Answer :**  
It is not Ok. You will get duplicate method error.

**33) Can you identify the error in below code snippet?**

*class A{*

*void myMethod(){*

*System.out.println("Super Class");*

*}*

*}*

*class B extends A{*

*@Override*

*void myMethod() throws SQLException{*

*System.out.println("Sub Class");*

*}*

*}*

**Answer :**  
SQLException is not compatible with throws clause of super class method. If super class method doesn’t have throws clause, then it can be overridden with only unchecked type of exceptions. SQLException is not an unchecked type of exception.

**34) Can we remove throws clause of a method while overriding it?**

**Answer :**  
Yes, we can remove throws clause of a method while overriding it.

**35) What will be the outcome of the following program?**

*class ABC{*

*void methodABC(){*

*new XYZ().methodXYZ();*

*}*

*}*

*class XYZ extends ABC{*

*void methodXYZ(){*

*methodABC();*

*}*

*}*

*public class MainClass{*

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

*new ABC().methodABC();*

*}*

*}*

**Answer :**  
You will get StackOverflowError.

**36) What will be the output of the below program?**

*class ABC{*

*void methodABC(){*

*System.out.println(111);*

*}*

*void methodABC(int i){*

*System.out.println(222);*

*}*

*}*

*class XYZ extends ABC{*

*@Override*

*void methodABC(int i){*

*System.out.println(333);*

*}*

*@Override*

*void methodABC(){*

*System.out.println(444);*

*}*

*}*

*public class MainClass{*

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

*ABC abc = new XYZ();*

*abc.methodABC(10);*

*abc.methodABC();*

*}*

*}*

**Answer :**  
333  
444

**37) What are the possible access modifiers a protected method can have if it is overridden in the sub class?**

**Answer :**  
protected or public.

**39) In the below example, is “methodOfX()” correctly overridden in the sub classes of Class X?**

*class X{*

*void methodOfX(){*

*System.out.println("Class X");*

*}*

*}*

*class Y extends X{*

*@Override*

*protected void methodOfX(){*

*System.out.println("Class Y");*

*}*

*}*

*class Z extends Y{*

*@Override*

*public void methodOfX(){*

*System.out.println("Class Z");*

*}*

*}*

**Answer :**  
Yes.

**40) What will be the output of the following program?**

*class ABC{*

*String myMethod(String s){*

*return s+s;*

*}*

*}*

*class PQR extends ABC{*

*String myMethod(String s, double d){*

*return myMethod(s+d);*

*}*

*}*

*class XYZ extends PQR{*

*String myMethod(String s, double d, int i){*

*return myMethod(s, d+i);*

*}*

*}*

*public class MainClass{*

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

*XYZ xyz = new XYZ();*

*System.out.println(xyz.myMethod("JAVA", 23.23, 111));*

*}*

*}*

**Answer :**  
JAVA134.23JAVA134.23

**Example of runtime polymorphism.**

In below example we create two class Person an Employee, Employee class extends Person class feature and override walk() method. We are calling the walk() method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime. Here method invocation is determined by the JVM not compiler, So it is known as runtime polymorphism.

**Example of Polymorphism in Java**

**class** Person

{

**void** walk()

{

System.**out**.println("Can Run....");

}

}

**class** Employee **extends** Person

{

**void** walk()

{

System.**out**.println("Running Fast...");

}

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

{

Person p=**new** Employee(); //upcasting

p.walk();

}

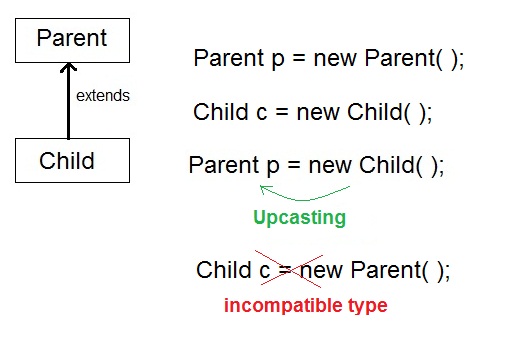
}

**Output**

Running fast...

### Runtime Polymorphism or Dynamic method dispatch

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.



#### Upcasting

When **Parent** class reference variable refers to **Child** class object, it is known as **Upcasting**

#### Example

class **Game**

{

public void type()

{ System.out.println("Indoor & outdoor"); }

}

Class **Cricket** extends **Game**

{

public void type()

{ System.out.println("outdoor game"); }

public static void main(String[] args)

{

Game gm = new Game();

Cricket ck = new Cricket();

gm.type();

ck.type();

gm=ck; **//gm refers to Cricket object**

gm.type(); **//calls Cricket's version of type**

}

}

**Output :**

Indoor & outdoor

Outdoor game

Outdoor game

Notice the last output. This is because of **gm = ck**; Now gm.type() will call Cricket version of type method. Because here gm refers to cricket object.

#### Q. Difference between Static binding and Dynamic binding in java ?

Static binding in Java occurs during compile time while dynamic binding occurs during runtime. Static binding uses type(Class) information for binding while dynamic binding uses instance of class(Object) to resolve calling of method at run-time. Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.

#### Covariant return type

Since Java 5, it is possible to override a method by changing its return type. If subclass override any method by changing the return type of super class method, then the return type of overriden method must be **subtype of return type** declared in original method inside the super class. This is the only way by which method can be overriden by changing its return type.

*Example*:

class Animal

{

Animal myType()

{

return new Animal();

}

}

class Dog extends Animal

{

Dog myType() **//Legal override after Java5 onward**

{

return new Dog();

}

}

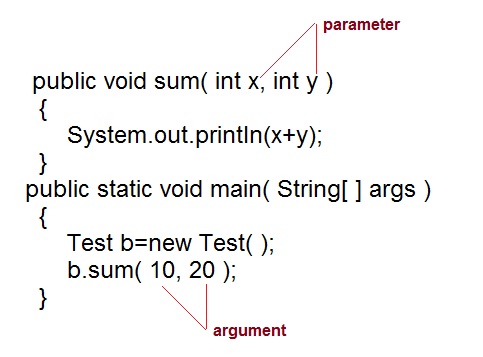
#### Q. Can we Override static method ? Explain with reasons ?

No, we cannot override static method. Because static method is bound to class whereas method overriding is associated with object i.e at runtime.

#### Parameter Vs. Argument

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

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



#### Example of Method overloading with type promotion.

class Area{

void find(long l,long b){

System.out.println("Area is"+(l\*b)) ;

}

void find(int l, int b,int h){

System.out.println("Area is"+(l\*b\*h));

}

public static void main (String[] args){

Area ar = new Area();

ar.find(8,5);   
**//automatic type conversion from find(int,int) to find(long,long)** .

ar.find(2,4,6) **//find(int l, int b,int h) is called**.

}

}

**Output :**

Area is 40

Area is 48

## Virtual Methods

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

### Example

/\* File name : Employee.java \*/

public class Employee {

private String name;

private String address;

private int number;

public Employee(String name, String address, int number) {

System.out.println("Constructing an Employee");

this.name = name;

this.address = address;

this.number = number;

}

public void mailCheck() {

System.out.println("Mailing a check to " + this.name + " " + this.address);

}

public String toString() {

return name + " " + address + " " + number;

}

public String getName() {

return name;

}

public String getAddress() {

return address;

}

public void setAddress(String newAddress) {

address = newAddress;

}

public int getNumber() {

return number;

}

}

Now suppose we extend Employee class as follows −

/\* File name : Salary.java \*/

public class Salary extends Employee {

private double salary; // Annual salary

public Salary(String name, String address, int number, double salary) {

super(name, address, number);

setSalary(salary);

}

public void mailCheck() {

System.out.println("Within mailCheck of Salary class ");

System.out.println("Mailing check to " + getName()

+ " with salary " + salary);

}

public double getSalary() {

return salary;

}

public void setSalary(double newSalary) {

if(newSalary >= 0.0) {

salary = newSalary;

}

}

public double computePay() {

System.out.println("Computing salary pay for " + getName());

return salary/52;

}

}

Now, you study the following program carefully and try to determine its output −

/\* File name : VirtualDemo.java \*/

public class VirtualDemo {

public static void main(String [] args) {

Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);

Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);

System.out.println("Call mailCheck using Salary reference --");

s.mailCheck();

System.out.println("\n Call mailCheck using Employee reference--");

e.mailCheck();

}

}

This will produce the following result −

### Output

Constructing an Employee

Constructing an Employee

Call mailCheck using Salary reference --

Within mailCheck of Salary class

ailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--

Within mailCheck of Salary class

ailing check to John Adams with salary 2400.0

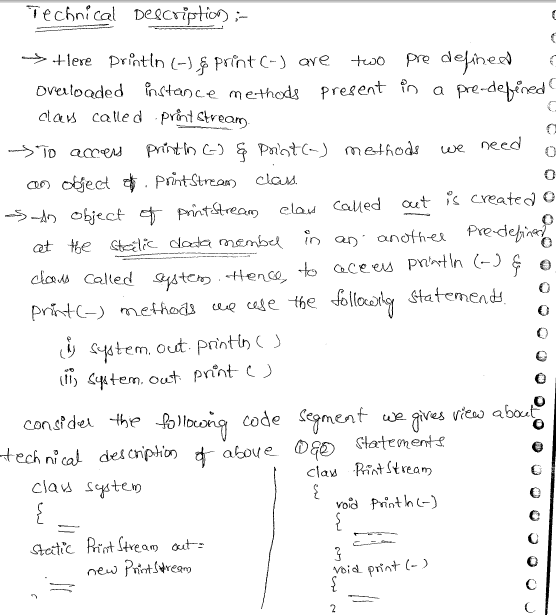
Here, we instantiate two Salary objects. One using a Salary reference **s**, and the other using an Employee reference **e**.

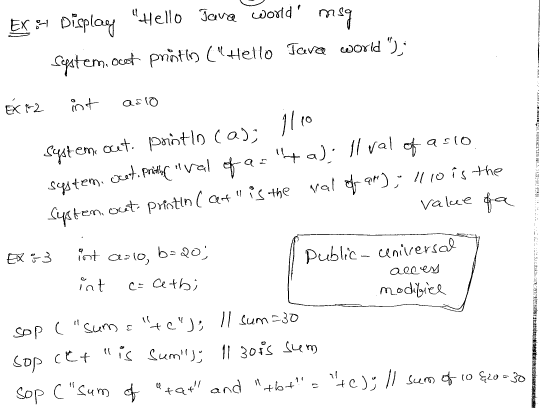
While invoking *s.mailCheck()*, the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.

mailCheck() on **e** is quite different because **e** is an Employee reference. When the compiler sees *e.mailCheck()*, the compiler sees the mailCheck() method in the Employee class.

Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.

This behavior is referred to as virtual method invocation, and these methods are referred to as virtual methods. An overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.





## Static Polymorphism Dynamic Polymorphism

Binding leads to polymorphism or **without binding there is no polymorphism**.

* 1. **Program on Static Polymorphism**

*public class Area{*

*public void calculate(){*  ***// I***

*System.out.println("Nothing to calculate");*

*}*

*public void calculate(int x){*  ***// II***

*System.out.println("Circle Area: " + Math.PI \* Math.pow(x, 2));*

*}*

*public void calculate(double x) {* ***// III***

*System.out.println("Circle Perimeter: " + 2 \* Math.PI \* x);*

*}*

*public void calculate(int x, int y){*  ***// IV***

*System.out.println("Rectangle Area: " +x\*y);*

*}*

*public static void main(String args[])*

*{*

*Area a1 = new Area();*

*a1.calculate();* ***// I***

*a1.calculate(10, 20);*  ***// IV***

*a1.calculate(10.5);*  ***// III***

*a1.calculate(10);*  ***// II***

*}*

*}*

In the above example, **calculate()** method is called four times and each time it gives different outputs. This is called Polymorphism. In an OOPs language, **the same method called different times gives different outputs is known as Polymorphism (poly=many, morphism=form)**. Otherway, many forms of the same method. For example, I say **God is polymorphic**. God is only one but has different forms like Lord Balaji, Goddess Laxmi, Goddess Saraswathi and infact, each one fulfils different wishes like Chilkur Balaji in Hyderabad helps in Visa stamping.

**As this polymorphism is achieved through static binding, it is known as static polymorphism.**

Method overloading leads to static binding and static binding leads to static polymorphism.

**Method overloading –> Static Binding –> Static Polymorphism**

###### **B) Dynamic Polymorphism**

Observe the program given in the same "Static binding and Dynamic binding".

*class Lecturer{*

*public void call(){* ***// I***

*System.out.println("Hello Sir, Good morning");*

*}*

*}*

*class Raju extends Lecturer{*

*public void call(){ // II*

*System.out.println("Hello Raju, improve your programming skills");*

*}*

*}*

*class Prasad extends Lecturer{*

*public void call(){ // III*

*System.out.println("Hello Prasad, improve your English language");*

*}*

*}*

*class Jyostna extends Lecturer{*

*public void call(){ // IV*

*System.out.println("Hello Jyostna, improve your aptitude and reasoning skills");*

*}*

*}*

*public class DynamicBindingExample{*

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

*Lecturer l1 = new Lecturer();*

*l1.call();*  ***// I***

*Raju r1 = new Raju();*

*l1 = r1;*

*l1.call();* ***// II***

*Prasad p1 = new Prasad();*

*l1 = p1;*

*l1.call();* ***// III***

*Jyostna j1 = new Jyostna();*

*l1 = j1;*

*l1.call();* ***// IV***

*}*

*}*

Here also, the same method **call()** called different times gives different outputs of calling**Raju**, **Prasad** and**Jyostna** with different advices. **This is known as polymorphism. As polymorphism is achieved through dynamic binding, it is known as dynamic polymorphism**.

**Principle involved in Dynamic binding**

*l1 = r1;  
l1.call();*

**If a subclass object is assgined to super class object, the super class object will call subclass overridden method.** Observe, when **r1** is assigned to **l1**, super class **l1** will call subclass **r1**call() method.

|  |  |  |
| --- | --- | --- |
| PROPERTY | STATIC POLYMORPHISM | DYNAMIC POLYMORPHISM |
| OCCURRENCE | Occurs at compile-time | Occurs at runtime |
| ACHIEVEMENT | Achieved through static binding | Achieved through dynamic binding |
| REQUIRED METHODS | Method overloading should exist | Method overriding should exist |
| INHERITANCE | Inheritance is not involved | Inheritance is involved |
| CLASSES | Happens in the same class | Happens between two classes |
| OBJECT ASSIGNMENT | Object assignment is not required | Required where a subclass object is assigned to super class object |

**When superclass object refer to subclass object and super class object belong to superclass and can access all super class methods as well as fields i mean it’s purely superclass object .why superclass object can not call subclass methods that are not overriden as well as fields ??**

Else, dynamic polymorphism cannot be achieved.

**why it is called dynamic binding? superclass is pointing subclass that is known at the compile time.**

Because which subclass method is to be called is known at runtime only. At compile time, only syntax is checked.

##### **Object Casting Polymorphism Interview Questions**

1. **What is casting?**  
   Converting one data type (or object) to the other is known as casting. It is required in a programming language as the user entered data may be in different form than required in coding.
2. **How many types of casting exists in Java?**  
   There are two types of casting – [Primitive data type casting](http://way2java.com/casting-operations/data-type-casting-type-conversion/) and [Object casting](http://way2java.com/casting-operations/object-casting/) ([string to data type conversion is not casting](http://way2java.com/java-lang/string-to-int-conversion-in-java/)).
3. **How many types of primitive data type casting exist in Java?**  
   Three types – implicit casting, explicit casting and boolean casting.
4. **How many types of object casting exist in Java?**  
   Two types – implicit casting and explicit casting.
5. **What are data type casting rules?**  
   1. A data type of lower size (occupying less memory) can be assigned to a data type of higher size implicilty (done by JVM automatically).  
   2. But the converiosn of data type of higher size to lower size requires explicit conversion (should be done by programmer himself).
6. **What are**[**object casting rules**](http://way2java.com/casting-operations/object-casting/)**?**  
   1. Object of the same class can be assigned to one another and is done implicilty.  
   2. A subclass object can be assigned to a super class object and is done implicitly.  
   3. A super class object cannot be assigned to a subclass object, and still to do, requires explicit casting.
7. **What is dynamic method dispatch or dynamic binding?**  
   Decision of which method is to be sent for execution at runtime is known as dynamic method dispatch or dynamic binding (remember, Java achieves static binding through method overloading).
8. **How Java achieves dynamic polymorphism?**  
   Java achieves [dynamic polymorphism](http://way2java.com/oops-concepts/dynamic-polymorphism/) through [method overriding](http://way2java.com/oops-concepts/method-overriding/) and [object casting](http://way2java.com/casting-operations/object-casting/).