OVERRIDING

P p = **new** C();

p.m1();*//will call child objects method`*

**class** P {

**public void** m1(){}

}

**class** C **extends** P{

**public void** m1(){}

}

In overriding, method resolution always takes care by JVM based on runtime object and hence overriding is also considered as runtime polymorphism or dynamic polymorphism or late binding.

Rules for overriding:

Method names, and argument types must be matched. Method signatures must be same.

Return types must be same, but this rule applicable until 1.4 version only. For 1.5 version onwards we can take co-variant return types.

According to this child class method return type need not be same as parent method return type. Its child type also allowed.

**class** P {

**public** Object m1(){ **return null**;}

}

**class** C **extends** P{

@Override

**public** String m1(){**return null**;}

}

Above code is invalid in 1.4v. But it works well in 1.5v.

This above rule is not applicable to primitive types.

Rule 2:

Parent class private methods not available to the child, and hence overriding concept is not applicable for private methods.

Rule 3:

Final method is not available to the child and hence overriding concept is not possible.

We will compile time error.

Rule 4:

Parent class abstract methods, we should override in child class to provide implementation

**abstract class** P {

**public abstract** Object m1();

}

**class** C **extends** P{

@Override

**public** String m1(){**return null**;}

}

Rule 5:

We can override non-abstract method as abstract.

**class** P {

**public** Object m1() { **return null**;}

}

**abstract class** C **extends** P{

@Override

**abstract public** String m1();

}

Advantage: We can stop the availability of parent method implementation to next level child classes.

Rule 6:

In overriding

final to non-final to not possible

non-final to final is possible

The following modifiers wont keep any restrictions

synchronized(up and down change is possible), native, strictfp.

Rule 7:

While overriding we can’t reduce the scope

**class** P {

**public** Object m1() { **return null**;}

}

**class** C **extends** P{

Object m1() { **return null**;} //invalid

}

attempting to assign weaker access privileges error we will get.

But opposite is possible

**class** P {

Object m1() { **return null**;}

}

**class** C **extends** P{

**public** Object m1() { **return null**;} //valid

}

Rule 8:

If child class throws any checked exception compulsary parent class method should throw the same checked exception or it parent. Otherwise we will get compile time error. But there are not restriction for un-checked exception.

Valid:

1. public void m1() throws Exception

public void m1()

2. public void m1() throws Exception

public void m1() throws IOException

3. public void m1() throws IOException

public void m1() throws FileNotFoundException, EOFException

4. public void m1() throws IOException

public void m1() throw AE, NPE

Invalid:

1. public void m1()

public void m1() throws Exception

2. public void m1() throws IOException

public void m1() throws Exception

3. public void m1() throws IOException

public void m1() throws EOFException, InterruptedException

Our overriding should not affect the outside callers.

Rule 9:

Overriding with respect to static methods

**class** P {

**public static void** m1() {}

}

**class** C **extends** P{

**public void** m1() { **return null**;} //can’t override static method

}

We can’t override a static method non static, otherwise we will get compile-time error.

**class** P {

**public void** m1() {}

}

**class** C **extends** P{

**public static void** m1() { }

}

Similarly we can’t override a non-static method as static method.

**class** P {

**public static void** m1() {}

}

**class** C **extends** P{

**public static void** m1() { }

}

No issues :)

If both parent and child class methods are static then we wont get compile time error. It seems overriding concept applicable for static method but it is not overriding and it is method hiding

Method hiding:

Method resolution always takes care by compiler – I got it

Rule 10:

Overriding with respect Var-args methods

We can override var-arg method with another var-arg method only.If we are trying to override with normal method, then it is not overriding it is overloading.

**class** P {

**public void** m1(**int**... i) {}

}

**class** C **extends** P{

**public void** m1(**int** i) { } *// this is not at all method overriding this is overloading*

}

P p = **new** C();

p.m1(); //parent method will be called

Rule 11:

Overriding with respect Variables methods:

Variable resolution always takes care by compiler based on reference type irrespective of whether the variable is static or instance(Overriding concept applicable only for methods not for variable).

**class** P {

**int x** = 888;

}

**class** C **extends** P{

**int x** = 999;

}

P p = **new** C();

System.***out***.println(p.**x**); //888

Difference b/w overloading and overriding

|  |  |  |
| --- | --- | --- |
| Property | Overloading | Overriding |
| Method name | Must be same | Must be same |
| Argument types | Must be different(atleast order) | Must be same(including order) |
| Method signature | Must be different(because of different arguments type) | Must be same |
| Return type | No restriction | Must be same until 1.4. From 1.5V onwards co-variants are allowed |
| Private, static, final methods | Can be overloaded | Cannot be overriden |
| Access modifiers | No strictions | The scope of access modifier cannot be reduced but we can increase the scope |
| Throws clause | No Restriction | If child class method throws any checked exception, compulsary parent class method should throws same checked exception or it parent. But no restrictions for un-checked exception. |
| Method resolution | Compiler based on Reference type | By JVM based on runtime object |
| It is also know as | Compile time polymorphism or Static polymorphism or early binding. | Runtime Polymorphism or late binding. |

Conclusion:

In overloading we have to check only method names(must be same) and argument types must be different.

We are not required to check remaining return types, access modifiers etc.

But in overriding everything we need to check method names, argument types, return types, access modifiers, throws keyword etc.

Consider the following method in parent

*public void m1(int x) throws IOException*

*In the child class which of following method we can take*

*public void m1(int i) – overiding is possible*

*public static int m1(long l) – overloading is possible*

*public static void m1(int i) – overriding not possible*

*public void m1(int i) throws Exception – overiding not possible*

*public static abstract void m1(double d) – static and abstract illegal combination*