*Polymorphism*

Same name with different forms is the concept of polymorphism.

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Example

We can use same abs() method to calculate the absolute value for int type, long type, float type etc.

Example:

1. abs(int)

2. abs(long)

3. abs(float)

We can achieve polymorphism by using two ways.

1. *Method overloading-*
2. *Method overriding-*
3. *Method overloading-*

It is the same method name with different argument called as Method overloading. There is no need of super and sub class relationship. It is called as early binding, compile time polymorphism or static binding.

Having overloading concept in java reduces complexity of the programming.

**Example-1**

**public** **class** Test {

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

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

}

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

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

}

**void** add(**double** a) {

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

}

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

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

}

}

**public** **class** TestMain {

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

Test test = **new** Test();

test.add(10.5);

test.add(10.5, 11.5);

test.add(2, 4);

test.add(5, 10, 15);

}

}

Output is

10.5

22.0

6

30

*Point to be remember*

1. Method name must be same.
2. Parameter or argument must be different. (sequence of argument, number of argument or datatype should be different)
3. Return type can be anything
4. Access specifier can be anything
5. Exception thrown can be anything

**Why?**

Suppose we got the business requirement from the client in last year

Class Employee {

Void addStudent (String firstname, string lastname, string city) { }

End user is calling the class as below

//End User 1

addStudent (“ram”,”pawar”,”Pune”);

//End User 2

addStudent (“ram”,”deshmukh”,”Mumbai”); }

After that I got the new requirement from the client in current year, to update the pan card details.

What options we have?

1. Modified into the existing method.

2. Create the new method with new parameter.

First way modifying into existing method is not good approach, it will increase the unit testing of it. If we are making the changes into existing method, then how existing user will call the method I mean they need to add one more extra attribute, in future again, you got requirement to add one more attribute so every time user need to change at their side, this is not the good thing.

Second way, create the same method in that class and add the new attribute into it. If client second want pan card details so he can call that method otherwise calls the first method if pan card is not required.

**Example -2**

**public** **class** TestMain {

**public** **void** methodOne(**int** i) {

System.***out***.println("int-arg method");

}

**public** **void** methodOne(**float** f) // overloaded methods

{

System.***out***.println("float-arg method");

}

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

TestMain t = **new** TestMain();

t.methodOne(12);// int-arg method

t.methodOne(10.5f);// float-arg method

// t.methodOne(10.5);//C.E:cannot find symbol

}

}

Output

int-arg method

float-arg method

**Example- 3**

**package** com.poly;

**public** **class** A{

**void** test(Object object) {

System.***out***.println("test- Object");

}

**void** test(String string) {

System.***out***.println("test- String");

}

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

A a = **new** A();

a.test(**new** Object());

a.test("Rahul");

a.test(**new** X());

a.test(**new** String());

}

}

Output :

test- Object

test- String

test- Object

test- String

*Why it is called as compile time polymorphism?*

Because it is decided at compile time which one method should get called that’s why it is called as compile time polymorphism.

In overloading compiler is responsible to perform method resolution (decision) based on the reference type (but not based on run time object). Hence overloading is also considered as compile time polymorphism (or) static polymorphism (or) early biding.

In overloading method resolution is always based on reference type and runtime object won't play any role in overloading.

*2.Overriding*

1. Whatever the Parent has by default available to the Child through inheritance, if the Child is not satisfied with Parent class method implementation then Child is allow to redefine that Parent class method in Child class in its own way this process is called overriding.

2. The Parent class method which is overridden is called overridden method.

3. The Child class method which is overriding is called overriding method.

*Point to be remember*

1. Method name must be same. That is method signature must be same. Method signature means method name with argument (return type and access specifier are not part of method signature).

2) Until 1.4 version the return types must be same but from 1.5 version onwards covariant return types are allowed. // covariant return types we can see upcoming lecture

3) Method parameters must be same.

4.We can extend the method scope in overriding but not reduce the visibility of it.

5. Subclass method's access modifier must be the same or higher than the superclass method access modifier

6.While overriding if the child class method throws any checked exception compulsory the parent class method should throw the same checked exception or its parent otherwise we will get compile time error.

Let see the Example

* In the above program, B is implementing the method m1 () with the same signature as super class A i.e m1 () of class B is overriding m1() of class A.
* If you want to add new features to existing class, then you should not disturb the existing class. You should always write the subclass of that class that is the best practice.

**public** **class** A {

**void** m1() {

System.***out***.println("class - A- m1 () method");

}

}

**public** **class** B **extends** A {

@Override

**void** m1() {

System.out.println("class - B- m1 () method");

}

**void** m2() {

System.out.println("class- B- m2() method");

}

}

**public** **class** TestMain {

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

B b= **new** B();

b.m1();

b.m2();

}

}

Output-

class - B- m1 () method

class- B- m2() method

Example base on Accessibility>>

Subclass method's access modifier must be the same or higher than the superclass method access modifier

|  |  |
| --- | --- |
| superclass | In subclass, we can have access specifier |
| public | public |
| protected | protected, public |
| default | default, protected, public |
| private | We cannot override the private |

Until 1.4 version the return types must be same but from 1.5 version onwards covariant return types are allowed.

According to this Child class method return type need not be same as Parent class method return type its Child types also allowed.

**class** Parent {

**public** Object methodOne() {

**return** **null**;

}

}

**class** Child **extends** Parent {

**public** String methodOne() {

**return** **null**;

}

}

1. Parent class final methods we can't override in the Child class.
2. Parent class non-final methods we can override as final in child class. We can override native methods in the child classes.
3. Private methods are not visible in the Child classes hence overriding concept is not applicable for private methods.
4. We should override Parent class abstract methods in Child classes to provide implementation.
5. While overriding we can't reduce the scope of access modifier.

Example:

**class** Parent {

**public** **void** methodOne() { }

}

**class** Child **extends** Parent {

**protected** **void** methodOne( ) { }

}   
Output:

Compile time error

*Some Live Example for reference*

**Live Example-1**

**class** RBI {

**void** getSimpleIntereset(**float** simpleRate) {

//logic here

}

}

**class** Axis **extends** RBI {

**void** getSimpleIntereset(**float** simpleRate) {

//logic here

}

}

**class** HDFC **extends** RBI {

**void** getSimpleIntereset(**float** simpleRate) {

//logic here

}

}

**Live Example-2**

**class** Policy {

**void** getPremium(Customer customer) {

//logic here

}

}

**class** TermPolicy **extends** Policy {

**void** getPremium(Customer customer) {

//logic here

}

}

**class** RiderProtection **extends** TermPolicy {

**void** getPremium(Customer customer) {

//logic here

}

}