**Achieving Polymorphism**

**Method Overloading**

Method overloading can be achieved by defining multiple methods in the same class with the method name being same but the parameters are different.

Eg:

class ArithmeticUtils  
{  
    int add(int a, int b)    // LINE A  
    {  
        return a + b;  
    }  
  
    int add(int a, int b, int c) // LINE B  
    {  
        return a + b + c;  
    }  
  
    double add(double a, double b, double c) // LINE C  
    {  
        return a + b + c;  
    }  
  
    int multiply(int a, int b)  
    {  
        return a \* b;  
    }  
  
    int multiply(int a, int b, int c)  
    {  
        return a \* b \* c;  
    }  
}

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;  
}

**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);  
    }  
}

**Method Overriding**

Method overriding is a powerful concept supported by Java, which helps in changing the behavior of the method implemented in the super-class.

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.");  
    }  
}

One more 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;  
    }  
}

|  |  |
| --- | --- |
| **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. |

**Super Keyword in Java to call super class constructor**

The other use of super keyword is to call the super-class constructors from the sub-class constructor.

class IceCreamPricesWithConstructors  
{  
    public static void main(String arg[])  
    {  
        IceCream ic = new IceCream("Pista", 2);  
        System.out.println(ic.numberOfScoops + " scoops of " + ic.flavor + " flavor price is : " + ic.getPrice());  
          
        FruitSaladWithIceCream fs = new FruitSaladWithIceCream("Chocolate", 1, 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("Vanila", 1, 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;  
      
    IceCream(String flavor, int numberOfScoops)  
    {  
        this.flavor = flavor;  
        this.numberOfScoops = numberOfScoops;  
    }  
      
    double getPrice()  
    {  
        double pricePerScoop = 35.0;  
        return numberOfScoops \* pricePerScoop;  
    }  
}  
  
class FruitSaladWithIceCream extends IceCream  
{  
    int gramsOfFruitSalad;  
  
    FruitSaladWithIceCream(String flavor, int numberOfScoops, int gramsOfFruitSalad)  
    {  
        super(flavor, numberOfScoops); // LINE A  
        this.gramsOfFruitSalad = gramsOfFruitSalad; // LINE B  
    }  
  
    double getPrice()  
    {  
        double iceCreamPrice = super.getPrice();   
        double pricePerGram = 0.75;  
  
        return gramsOfFruitSalad \* pricePerGram + iceCreamPrice;  
    }  
}  
  
class KhubaniKaMeetaWithIceCream extends IceCream  
{  
    int gramsOfKhubaniKaMeeta;  
  
    KhubaniKaMeetaWithIceCream(String flavor, int numberOfScoops, int gramsOfKhubaniKaMeeta)  
    {  
        super(flavor, numberOfScoops);  
        this.gramsOfKhubaniKaMeeta = gramsOfKhubaniKaMeeta;  
    }  
  
    double getPrice()  
    {  
        double iceCreamPrice = super.getPrice();  
        double pricePerGram = 1.25;  
  
        return gramsOfKhubaniKaMeeta \* pricePerGram + iceCreamPrice;  
    }  
}

**Inheritance and constructors**

class MultiLevelDefaultConstructors  
{  
    public static void main(String arg[])  
    {  
        System.out.println("---------------");  
        A a = new A();  
        System.out.println("---------------");  
        B b = new B();  
        System.out.println("---------------");  
        C c = new C();  
        System.out.println("---------------");  
      
    }  
}  
  
class A  
{  
    A()    // LINE A  
    {  
        System.out.println("Created A");  
    }  
}  
  
class B extends A  
{  
    B()  
    {  
        System.out.println("Created B");  
    }  
}  
  
class C extends B  
{  
    C()  
    {  
        System.out.println("Created C");  
    }  
}

**Multi Level Constructors**

class MultiLevelConstructors  
{  
    public static void main(String arg[])  
    {  
        System.out.println("---------------");  
        X x = new X(10);  
        System.out.println("---------------");  
        Y y = new Y(11, 21);  
        System.out.println("---------------");  
        Z z = new Z(12, 22, 32);  
        System.out.println("---------------");  
      
    }  
}  
  
class X  
{  
    int i;  
    X(int i)  
    {  
        this.i = i;  
        System.out.println("Created X");  
    }  
}  
  
class Y extends X  
{  
    int j;  
    Y(int i, int j)  
    {  
        super(i);  
        this.j = j;  
        System.out.println("Created Y");  
    }  
}  
  
class Z extends Y  
{  
    int k;  
    Z(int i, int j, int k)  
    {  
        super(i, j);  
        this.k = k;  
        System.out.println("Created Z");  
    }  
}

**Run time polymorphism**

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.

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");  
    }  
}

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.

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.");  
    }  
}

**B ref = new A();**

* The above line of code gives a Compilation Error. We can assign a sub class object to super class reference but we can not assign a super class object to a sub class reference.