L34

Monday, January 31, 2022 8:32 PM

Method overloading and Method overriding, Super keyword, Polymorphism, Object Class

Saurav Hathi

https://www.youtube.com/channel/UCp6MFWao5vWRnyRCxBsKnfw

Method Overloading

Q1. Method overloading means the ability to have multiple methods with same name, which vary in their parameters. For example:

```
public void concatenate(String text, int num) {
        return text + num;
}
public void concatenate(String text, boolean flag) {
        return text + flag;
}
public void concatenate(String text, char ch) {
        return text + ch;
}
```

In the above code, concatenate method is overloaded **three** times.

Note: In the above example, the variation in the parameters list can be by their count or type or both.

Select all the correct statements given below regarding methods present in the String class.

[**Hint:** You can explore the methods present in the String class by clicking on String and scrolling down until you reach a section heading named **Method Summary**. In that you will see table containing a list of all methods in the String class.]

- [x] `charAt` method is not overloaded.
- [x] `indexOf` method is overloaded **4 times**.
- [x] The `String` constructor is overloaded **15 times** .
- [ ] `valueOf` method is overloaded **8 times**.

Q2. Write a Java program with a class name Addition with the methods add(int, int), add(int, float), add(float, float) and add(float, double, double) to add values of different argument types.

Write the **main(String[])** method within the class and assume that it will always receive a total of **6** command line arguments at least, such that the first **2** are **int**, next **2** are **float** and the last **2** are of type **double**.

If the **main()** is provided with arguments : **1, 2, 1.5f, 2.5f, 1.0, 2.0** then the program should print the output as:

```
Sum of 1 and 2 : 3
Sum of 1.5 and 2.5 : 4.0
Sum of 2 and 2.5 : 4.5
Sum of 1.5, 1.0 and 2.0 : 4.5
```

```java

package q11266;

public class Addition {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        float d = Float.parseFloat(args[2]);
        float e = Float.parseFloat(args[3]);
        double h = Double.parseDouble(args[4]);
        double i = Double.parseDouble(args[5]);
        Addition obj = new Addition();
        obj.add(a, b);
        obj.add(d, e);
        obj.add(b, e);
        obj.add(d, h, i);
    }

    void add(int a, int b) {
        System.out.println("Sum of " + a + " and " + b + " : " + (a + b));
    }

    void add(int c, float d) {
        System.out.println("Sum of " + c + " and " + d + " : " + (c + d));
    }

    void add(float e, float f) {
        System.out.println("Sum of " + e + " and " + f + " : " + (e + f));
    }

    void add(float g, double h, double i) {
        System.out.println("Sum of " + g + ", " + h + " and " + i + " : " + (g + h + i));
    }
}

```

Q3. Write a class Box which contains the data members **width, height** and **depth** all of type **double**.

Write the implementation for the below **3** *overloaded constructors* in the class Box :

- **Box()** - default constructor which initializes all the members with **-1**
- **Box(length)** - parameterized constructor with one argument and initialize all the members with the value in **length**

the members with the corresponding arguments

- **Box(width, height, depth)** - parameterized constructor with three arguments and initialize

Write a method public double volume() in the class Box to find out the **volume** of the given box.

Write the **main** method within the Box class and assume that it will receive either **zero** arguments, or **one** argument or **three** arguments.

For example, if the **main()** method is passed **zero** arguments then the program should print the output as:
Volume of Box() is : -1.0
Similarly, if the **main()** method is passed **one** argument : **2.34**, then the program should print the output as:
Volume of Box(2.34) is : 12.812903999999998
then the program should print the output as: Likewise, if the **main()** method is passed **three** arguments : **2.34, 3.45, 1.59**, then the program should print the output as:
Volume of Box(2.34, 3.45, 1.59) is : 12.836070000000001

```java
package q11267;
public class Box {
    double width;
    double height;
    double depth;
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    Box() {
        width = -1;
        height = -1;
        depth = -1;
    }
    Box(double length) {
        width = length;
        height = length;
        depth = length;
    }
    public double volume() {
        return width * height * depth;
    }
    public void display() {
        if (width == -1 && height == -1 && depth == -1) {
            System.out.println("Volume of Box() is : " + volume());
        } else if (width != -1 && width == height) {
            System.out.println("Volume of Box(" + height + ") is : " + volume());
        } else {
            System.out.println("Volume of Box(" + width + ", " + height + ", " + depth + ") is : " + volume());
        }
    }
    public static void main(String[] args)
    {
```

```
        double[] arr = new double[3];
        for (int i = 0; i < args.length; i++) {
            arr[i] = Double.parseDouble(args[i]);
        }
        if (arr[0] == 0) {
            Box b = new Box();
            b.display();
        } else if (arr[1] == 0) {
            Box b = new Box(arr[0]);
            b.display();
        } else {
            Box b = new Box(arr[0], arr[1], arr[2]);
            b.display();
        }
    }
}
```

Q4. Write a Java program with a class name OverloadArea with overload methods area(float) and area(float, float) to find area of **square** and **rectangle**.

Write the **main** method within the class and assume that it will receive a total of **2** command line arguments of type **float**.

If the **main()** is provided with arguments : **1.34, 1.98** then the program should print the output as:
Area of square for side in meters 1.34 : 1.7956
Area of rectangle for length and breadth in meters 1.34, 1.98 : 2.6532001

```
package q11268;
public class OverloadArea {
    // Write the overload methods
    Float area(float x){
        return x*x;
        }
    Float area(float x, float y){
        return x*y;

    }

    public static void main (String[] args) {
        // Write the code
        float f1 = Float.parseFloat(args[0]);
        float f2 = Float.parseFloat(args[1]);
        OverloadArea obj = new OverloadArea();
        System.out.println("Area of square for side in meters " + f1 + " : " + obj.area(f1)); // Fill in the missing code
        System.out.println("Area of rectangle for length and breadth in meters " + f1 + ", " + f2 + " : " + obj.area(f1,f2)); // Fill in the missing code
}
}
```

Method Overriding

Q1. In implementation inheritance, when a class B inherits from a class A, the subclass B can modify the implementation present i n its superclass A. For example:

```
class A {
        public int aValue = 2;
        public int getAValue() {
                return aValue;
        }
}
class B extends A {
        public int bValue = 3;
        public int getBValue() {
                return bValue;
        }
        public int getAValue() {  // this method overrides the implementation in class A return 2 * aValue; // returning  double of value stored in aValue }
}
```

Then the below will print **4** and not 2:

```
B b = new B();
System.out.println(b.getAValue());
```

Note: In the above code, the method getAValue() in class B is overriding the method with same name present in its superclass A.

Whenever a method is overridden in the subclass, the new method implementation will be called when the method is invoked on t he instance of subclass.

See and retype the below code to understand the above concept.

In the below code, you will also notice that even a2.getAValue() produces **4**. This is because reference a2 actually points to an instance of class B, which has the overridden method.

```java
package q11269;
public class InheritanceExample {
    public static void main(String[] args) {
        A a1 = new A();
        System.out.println("a1.getAValue() : " + a1.getAValue());
        B b = new B();
        System.out.println("b.getBValue() : " + b.getBValue());
        System.out.println("b.getAValue() : " + b.getAValue());
        A a2 = b;
        System.out.println("a2.getAValue() : " + a2.getAValue());
    }
}
class A {
    public int aValue = 2;
    public int getAValue() {
        return aValue;
    }
}
class B extends A {
    public int bValue = 3;
    public int getBValue() {
        return bValue;
    }
    public int getAValue() {
        return 2 * aValue;
    }
}
```

Q2. **Method overloading** is a feature which allows multiple methods with same name and different parameters in the same class.

Method overriding is a feature where we specialize (or modify) a method behaviour which is already present in the superclass. This takes places only when we have a class extending another class.

Select all the correct statements from the below code:

```
class A {
        public void printMe(int number) { // statement 1 System.out.println(number);
        }
        public void printMe(boolean flag) { // statement 2 System.out.println(flag);
        }
        public void printMe(int number, boolean flag) { // statement 3 System.out.println(number + " : " + flag);
        }
}
class B extends A {
        public void printMe(int number) { // statement 4 System.out.println("The double of " + number + " is : " + (number * 2));
        }
}
```

- [x] Statements 1, 2 and 3 in class A contain overloaded versions of method `printMe`.

  Correct.

- [ ] Statements 1, 2 and 3 in class A contain overridden versions of method `printMe`.

  No, when we have multiple methods with same name and different parameters in the same class it is called overloading and not overriding.

- [ ] The method in statement 4, overrides the method declared in statement 2.

  No, a method overrides a superclass method only when it has the same method name and parameter types in the subclass.

- [x] The method in statement 4, overrides the method declared in statement 1.

  Correct.

- [ ] The method declaration in statement 3 has more parameters than the methods declared in statements 1 and 2, hence it will not be considered as a overloaded version of `printMe`.

  No, for overloading the order of parameter (types) list should be different, meaning, the parameter list can vary in count or type or both.

Q3. Write a Java program to achieve concept of Method Overriding

Assume there is a class called Bank with method calculateInterest(float principal, int time).

Create sub-classes of Bank with names SBI, ICICI and AXIS and override the calculateInterest(float principal, int time) method.

Create a constant of type float called INTEREST_RATE in classes SBI, ICICI and AXIS with values 10.8, 11.6 and 12.3 respectively.

Use the formula **(principal * INTEREST_RATE * time) / 100** to calculate the **interest** for given **principal** and **time** and return the value as float in the **overriden method**.

For example, if the **two** arguments passed to the **main** method are 1000 and 5, (principal and time) below is the expected output:

```
SBI rate of interest = 540.0
```

ICICI rate of interest = 580.0
AXIS rate of interest = 615.0

```java
package q11271;
class Bank {
    public float calculateInterest(float principal, int time) {
        return 0;
        }

}
class SBI{
    public static final float INTEREST_RATE = 10.8f;
    public float calculateInterest(float principal, int time) {
        return (principal * INTEREST_RATE * time) / 100;
        }

}
class ICICI{
    public static final float INTEREST_RATE = 11.6f;
    public Float calculateInterest(float i, int j) {
        return (i * INTEREST_RATE * j) / 100;
        }

}
class AXIS{
    public static final float INTEREST_RATE = 12.3f;
    public Float calculateInterest(float i, int j) {
        return (i * INTEREST_RATE * j) / 100;
        }

}
public class TestOverriding {
    public static void main(String[] args) {
        SBI sbi = new SBI();
        ICICI icici = new ICICI();
        AXIS axis = new AXIS();
        float principal = Float.parseFloat(args[0]);
        int time = Integer.parseInt(args[1]);
        System.out.println("SBI rate of interest = " + sbi.calculateInterest(principal, time));
        System.out.println("ICICI rate of interest = " + icici.calculateInterest(principal, time));
        System.out.println("AXIS rate of interest = " + axis.calculateInterest(principal, time));
        }S
}
```

Super keyword and its usage

Q1. The super keyword in **java** is used to refer/access either a member field, method or a constructor present in the super class hierarchy of the current class where the super keyword is used.

The **super** keyword can be used to :

- access the member fields of parent class when both parent and child class have member fields with same name
- explicitly call the default or parameterized constructors of parent class
- access the method of parent class when child class has overridden that method

When both the child and parent class contain variables with same names, we can access the member field of parent class inside the child class by using the syntax:
super.variableName
When an **instance** of subclass is created, the **new** keyword invokes the **constructor** of child class. This invocation of constructor in the subclass **implicitly** invokes the constructor of the **parent** class, if we do not explicitly write code to invoke the constructor of super class using **super**.

Hence, we should always remember that when we create an object of child class, the **parent** class constructor is executed first and then the **child** class constructor is executed.

Supposing we do not write code to call the super class constructor in the child class constructor, then the compiler implicitly adds super() (this invokes the **no argument constructor** of parent class) as the first statement in the constructor of **child** class. If the parent class does not have a default constructor (constructor with no parameters), then the programmer should explicitly call the parameterized constructor else the compiler will flag and error saying there is no default constructor in the super class.

Below is the syntax for calling the **parameterized** constructor of a **parent** class :
super(parameter-list);
The call to the **parent** class constructor must be the first statement inside the **child** class constructor.

The **super** keyword can also be used to invoke parent class method. It is used in case of **method overriding**.

In other words **super** keyword is used when the base class method name and the derived class method name is same.

The syntax of calling a method of the super class using **super** keyword is:
super.methodName();
See and retype the below code to understand the usage of super keyword.

```java
package q11272;
class SuperClass {
    int num;
    public SuperClass(int value) {
        num = value;
    }
    public void printHello() {
        System.out.println("Hello from SuperClass");
    }
}
class SubClass extends SuperClass {
    int num;
    public SubClass(int value) {
        super(value);
        num = value + 5;
        System.out.println("SuperClass number = " + super.num);
        System.out.println("SubClass number = " + num);
    }
    public void printHello() {
        super.printHello();
        System.out.println("Hello from SubClass");
    }
}
public class SuperKeyword {
    public static void main(String[] args) {
        SubClass obj = new SubClass(10);
        obj.printHello();
    }
}
```

Q2. Write a Java program to illustrate the usage of **super** keyword.

Create a class called Animal with the below members:

- a constructor which prints **Animal is created**
- a method called **eat()** which will print **Eating something** and returns nothing.

Create another class called Dog which is derived from the class Animal, and has the below members:

- a constructor which calls **super()** and then prints **Dog is created**
- a method **eat()** which will print **Eating bread** and returns nothing
- a method **bark()** which will print **Barking** and returns nothing
- a method **work()** which will call **eat()** of the **superclass** first and then the **eat()** method in the current class, followed by the **bark()** method in the current class.

Write a class ExampleOnSuper with the **main()** method, create an object to Dog which calls the method **work()**.

```java
package q11273;
class Animal {
    public Animal() {
        System.out.println("Animal is created");
    }
    void eat() {
        System.out.println("Eating something");
    }
}
class Dog extends Animal {
    public Dog() {
        super();
        System.out.println("Dog is created");
    }
    void eat() {
        super.eat();
        System.out.println("Eating bread");
    }
    void bark() {
        System.out.println("Barking");
    }
    void work() {
        eat();
        bark();
    }
}
public class Main {
    public static void main(String args[]) {
        Dog d = new Dog();
        d.work();
    }
}
```

Q3. Write a Java program to access the class members using **super** Keyword.

Create a class called SuperClass with the below members:

- declare two member fields **value1** and **value2** of type **int**
- a parameterized constructor with **two** arguments, which assigns two arguments to the members respectively
- a method called **show()** which will print **This is super class show() method** as well as the value of **value1**.

Create another class called SubClass which is derived from the class SuperClass, and has the below members:

- declare two member fields **value3** and **value4** of type **int**
- a parameterized constructor with **four** arguments, which assigns the first two arguments with **SuperClass** members and next two values with **SubClass** members
- a method called **show()** which

1. will print **This is sub class show() method**
2. will call **show()** of **SuperClass**
3. will print **value2** from **SuperClass**
4. will print **value3** of **SubClass**
5. will print **value4** of **SubClass**

Write a class AccessUsingSuper with the **main()** method, create an object to SubClass which calls the method **show()**.

For example, if the input is given as [10, 20, 30, 40] then the output should be:
This is sub class show() method
This is super class show() method
value1 = 10
value2 from super class = 20
value3 = 30
value4 = 40

```java
package q11274;
class SuperClass {
    int value1, value2;
    // Write the code
    SuperClass(int value1, int value2) {

        this.value1 = value1;

        this.value2 = value2;
    }
    void show() {

        System.out.println("This is sub class show() method");
    }
}
class SubClass extends SuperClass {

    int value3, value4;
    // Write the code
    SubClass(int value1, int value2, int value3, int value4) {

        super(value1, value2);

        this.value3 = value3;

        this.value4 = value4;
    }
    void show() {

        super.show();
        System.out.println("This is super class show() method");
        System.out.println("value1 = " + value1);
        System.out.println("value2 from super class = " + value2);
        System.out.println("value3 = " + value3);
        System.out.println("value4 = " + value4);
    }
}

public class AccessUsingSuper {
    public static void main(String[] args) {
        SubClass obj = new SubClass(Integer.parseInt(args[0]), Integer.parseInt(args[1]), Integer.parseInt(args[2]),
        Integer.parseInt(args[3]));
        obj.show();
    }
}
```

Polymorphism

Q1. Let us consider the below example code to understand polymorphism.

First see and retype the below code and then observe the output you receive.

After seeing the output when you read the below text you will understand polymorphism better.

1. You will notice that getName() method is overridden in class B and class C.
2. You will notice that the overridden getName() method in class B first invokes the getName() method present in class A using the super keyword.
3. Similar is the case with class C.
4. Both the overridden methods in B and C append their own information to the value returned by the superclass's getName() method.

Notice that references a, b, c are declared to be of type class A, even though their instances objects are of types A, B and C respectively.

We are able to hold the objects of type B and C in references b, c of type A because both B and C are subclasses of type A.

Here it appears that the method getName() when invoked on references of type A is behaving differently (by producing different output).

The reason for this polymorphic behaviour is that JVM (Java virtual machine) does not call the method declared in the type (c lass) of the reference (which in our case is class A), but it will appropriately look for the implementation in the object (instance) whose address is held by the reference.

Hence, when b.getName() is invoked, since b referrers to an instance of class B, the implementation in class B is executed. Similar is the case with the method call c.getName();.

We can define polymorphism as the ability of a method in a class to behave differently depending on the actual object it is b eing invoked on.

```java
package q11275;
public class PolymorphismExample2 {
    public static void main(String[] args) {
        A a = new A();
        A b = new B();
        C c = new C();
        System.out.println(a.getName());
        System.out.println(b.getName());
        System.out.println(c.getName());
    }
}
class A {
    public String getName() {
        return "A";
    }
}
class B extends A {
    public String getName() {
        return super.getName() + " " + "B";
    }
}
class C extends B {
    public String getName() {
        return super.getName() + " " + "C";
    }
}

```

Q2. In Java, since Object class is the superclass (root class) of every class, instance of any class **is also an instance** of Object class.

For example:
class Person {
}
Person p = new Person();
In the above code, the instance referred by p IS-A Person. Since every class in Java, including Person is a subclass of Object, the statement p IS-A Object is also correct.

Which means every object in Java will have more than one IS-A relationship (One with its own class type and one with Object class type).

Any object which satisfies more than one IS-A relation is called polymorphic. For example, let us write classes A, B and C which override toString() method of Object class. First see and retype the below code and then observe the output you receive.

After seeing the output read the below observations to further understand the polymorphic behaviour.

1. You will notice that System.out.println(a); is same as System.out.println(a.toString());, because the **println** method which takes any object internally calls toString() method on that reference.
2. You will notice that the overridden toString() method in class B first invokes the toString() method present in class A using the super keyword.
3. Similar is the case with class C.
4. Both the overridden methods in B and C append their own information to the value returned by the superclass's toString() method.

Notice that references a, b, c are declared to be of type class Object, since all classes are subtypes of class Object.

For the above stated reason we are able to hold the instances of type A, B and C in references a, b, c of type Object.

When the method toString() is invoked on references of type Object, we see that the actual implementation present in their respective objects is being called to produce the output..

This behaviour is called **virtual method invocation**. Since the delegation to the correct method happens on the instance during runtime, its also called as **runtime polymorphism**.

```java
package q11276;
public class PolymorphismExample2 {
    public static void main(String[] args) {
        Object a = new A();
        Object b = new B();
        Object c = new C();
        System.out.println(a);
        System.out.println(b);
        System.out.println(c.toString());
    }
}
class A {
    public String toString() {
        return "A";
    }
}
class B extends A {
    public String toString() {
        return super.toString() + " " + "B";
    }
}
class C extends B {
    public String toString() {
        return super.toString() + " " + "C";
    }
}

```

Q3. Write a Java program that implements **runtime polymorphism**.

Create a class Animal with one method whoAmI() which will print **I am a generic animal.**

Create another class Dog which extends Animal, which will print **I am a dog.**

Create another class Cow which extends Animal, which will print **I am a cow.**

Create another class Snake which extends Animal, which will print **I am a snake.**

Write a class RuntimePolymorphismDemo with the **main()** method, create objects to all the classes Animal, Dog, Cow, Snake and call whoAmI() with each object.

```java
package q11277;
public class RuntimePolymorphismDemo {
    public static void main(String[] args) {
        Animal ref1 = new Animal();
        Animal ref2 = new Dog();
        Animal ref3 = new Cow();
        Animal ref4 = new Snake();
        ref1.whoAmI();
        ref2.whoAmI();
        ref3.whoAmI();
        ref4.whoAmI();
  }
}

// Write all the classes with methods
class Animal {
    void whoAmI() {
        System.out.println("I am a generic animal");
    }
}
class Dog extends Animal {
    void whoAmI() {
        System.out.println("I am a dog");
    }
}
class Cow extends Animal {
    void whoAmI() {
        System.out.println("I am a cow");
    }
}

class Snake extends Animal {
    void whoAmI() {
        System.out.println("I am a snake");
    }
}
```

Object class

Q1. In Java Object is the root class of all classes. Which means that all the methods of Object class described below are also present in each and every class in Java.

1. clone() - creates and returns a copy of this object. However, for this method to work the class has to implement Cloneable interface.
2. equals(Object obj) - it compares if two references are pointing to the same address. However, you can override this method to provide custom implementation which will verify the contents and not just references.We will learn more about it in the later sections.
3. finalize() - this method is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. Programmers do not call it, but can override and write cleanup code in it.
4. getClass() - this method returns the in-memory runtime representation of the Class object that was loaded and used to create the instance. For example:
   Student st = new Student();
   Class clazz = st.getClass();

   The reference **clazz** points to the Class object which has the details of the Student class loaded in memory.
5. hashCode() - this method returns a hash code value (an integer) for the object. (we will learn more about it in data structures)
6. toString() - this method returns the string representation of the object.

There are many other methods like **notify**, **notifyAll**, and **wait** which are used to synchronize data access and method calls when multiple threads are involved.

See and retype the below code and observe the output to understand the usage of getClass() method.

```java
package q11278;
public class GetClassExample {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        String text = "Ganga";
        System.out.println("a.getClass() : " + a.getClass());
        System.out.println("b.getClass() : " + b.getClass());
        System.out.println("text.getClass() : " + text.getClass());
    }
}
class A {
}
class B {
}
```