

Java (Core Java)

Polymorphism

Today's Topics

- Polymorphism
 - What is Polymorphism
 - Compile time - Method Overloading
 - Overloading vs. Overriding
 - Covariant Return
 - Run time Polymorphism - Up Casting and Dynamic Method Dispatch
 - instanceof

Polymorphism – What does it mean

- Polymorphism means “multiple forms”
- Polymorphism is implemented through two ways:
 - Method Overloading (usually within same class)
 - Method overriding (across inheritance hierarchies)

Method Overloading - example

```
public class AreaOfTriangle {  
    // Equilateral  
    public double areaOfTriangle(int side) {  
        return Math.pow(side, 2) * Math.pow(3, 0.5) / 4;  
    }  
  
    // Right Angled  
    public double areaOfTriangle(int base, int height) {  
        return 0.5 * base * height;  
    }  
  
    // Scalene  
    public double areaOfTriangle(int side1, int side2, int side3) {  
        double semiPerimeter = 0.5 * (side1 + side2 + side3);  
        return Math.pow(semiPerimeter * (semiPerimeter - side1) * (semiPerimeter - side2) * (semiPerimeter - side3),  
            0.5);  
    }  
  
    public static void main(String[] args) {  
        AreaOfTriangle au = new AreaOfTriangle();  
        System.out.println("Area of equilateral triangle of side 5 is : " + au.areaOfTriangle(5));  
        System.out.println("Area of right angled triangle of base 3 and height 4 is : " + au.areaOfTriangle(3, 4));  
        System.out.println("Area of a triangle of sides 3, 4, 5 is : " + au.areaOfTriangle(5, 5, 5));  
    }  
}
```

Overloading cannot be done by changing return type

```
public class SumOfTwoNumbers {  
  
    int add (int... inputValues) { // see how varargs is used  
        int sum = 0;  
        for (int i : inputValues) {  
            sum += i;  
        }  
        return sum;  
    }  
  
    float add (float... inputValues) {  
        float sum = 0;  
        for (float i : inputValues) {  
            sum += i;  
        }  
        return sum;  
    }  
  
    /* double add (float... inputValues) {  
        double sum = 0;  
        for (float i : inputValues) {  
            sum += i;  
        }  
        return sum;  
    } */  
  
    public static void main (String [] args) {  
        SumOfTwoNumbers sotn = new SumOfTwoNumbers();  
        System.out.println("Sum of 1, 2 is : " + sotn.add(1,2));  
        System.out.println("Sum of 1, 2, 3, 4, 5, 6 is : " + sotn.add(new int [] {1, 2, 3, 4, 5, 6}));  
        System.out.println("Sum of 1.1, 2.1, 3.6, 4, 5.9, 6.87 is : " + sotn.add(1.1f, 2.1f, 3.6f, 4, 5.9f, 6.87f));  
    }  
}
```

What do you think will happen if we uncomment this?

Comparison between Overloading and Overriding

Overloading

Happens typically in same class

Have to have same name

Needs to have different types or number of arguments

Return type cannot be different

Overriding

Happens in classes across inheritance hierarchy

Have to have same name

Needs to have same number and type of argument

Return type of overriding method can be subclass of the return type of overridden method (**covariant return**)

Overriding through concept of Covariant Return

```
public class Animal {
    String color;

    String printMessageFromAnimal() {
        return "Animal Says Hello!!";
    }

    // To be overridden in sub class
    Animal fetchThisAnimal() {
        System.out.println("Returning the Animal");
        return this;
    }

    public Animal() {
        System.out.println("\n\nAn animal is created");
    }

    public void eat() {
        System.out.println("Eating");
    }
}
```

```
public class Horse extends Animal{

    // Simple Overriding
    String printMessageFromAnimal() {
        return "Horse Says Hello!!";
    }

    // Covariant return - Horse is sub class of Animal,
    // and hence this overriding is allowed
    Horse fetchThisAnimal() {
        System.out.println("Returning the Horse");
        return this;
    }
}
```

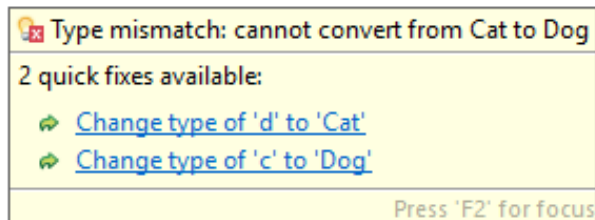
What do you think will happen if we change the return type of the `fetchThisAnimal` to String?

Run Time Polymorphism – Up-casting

- Also called “Dynamic Method Dispatch”
- Which method to call, is decided at run time.
- Implemented through up-casting

```
public class Upcasting {  
    public static void main (String [] args) {  
        Dog d = new Dog();  
        Cat c = new Cat();
```

```
        d = c; // Not allowed since dog is not a cat, and hence cat object cannot be referenced by a dog variable
```



```
        Animal da = new Dog(); // upcasting  
        Animal ca = new Cat(); // upcasting
```

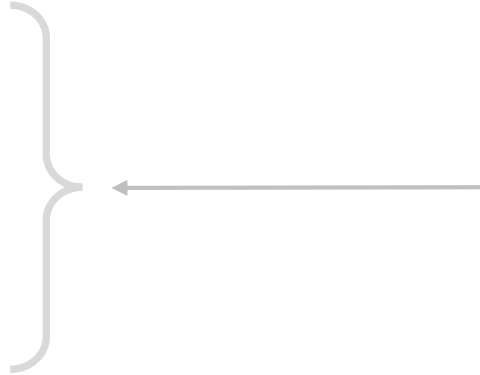
```
        da = ca; // Allowed since both dog and cat are Animals (through inheritance)
```

```
    }  
}
```


Run Time Polymorphism - Example

```
public class TestRunTimePolymorphism {  
    public static void main (String [] args) {  
        Animal a1 = new Animal();  
        a1.eat(); // But we do not know which animal yet..  
  
        // So make the call to find out which animal is to be fed today and then call the eat method  
        Animal a2 = findWhichAnimalToFeed("Monday");  
        a2.eat(); // During compile time we do not know whether a2 is a dog or cat.  
    }  
}
```

```
private static Animal findWhichAnimalToFeed(String dayOfWeek) {  
    switch (dayOfWeek) {  
        case "Sunday":  
        case "Tuesday":  
        case "Thursday":  
        case "Saturday":  
            return new Dog();  
        default:  
            return new Cat();  
    }  
}
```



Which method to call, is decided at run time.

instanceof

```
public class TestInstanceOf {
    public static void main(String[] args) {
        Animal a1 = new Animal();
        a1.eat(); // But we do not know which animal yet..

        System.out.println("Is this an instance of Animal? " + (a1 instanceof Animal));
        System.out.println("Is this an instance of Dog? " + (a1 instanceof Dog));
        System.out.println("Is this an instance of Cat? " + (a1 instanceof Cat));

        // So make the call to find out which animal is to be fed today and then call the eat method
        Animal a2 = findWhichAnimalToFeed("Monday");
        a2.eat(); // During compile time we do not know whether a2 is a dog or cat.

        System.out.println("Is this an instance of Animal? " + (a2 instanceof Animal));
        System.out.println("Is this an instance of Dog? " + (a2 instanceof Dog));
        System.out.println("Is this an instance of Cat? " + (a2 instanceof Cat));

        // Sample usage of instanceof
        // Suppose you need to make the animal make a sound. Because you do not know whether it is a
        // dog or a cat, you cannot decide which one to call

        if (a2 instanceof Dog) {
            ((Dog) a2).bark(); // type casted a2 to Dog first and then invoked the bark method. Why?
        } else if (a2 instanceof Cat) {
            ((Cat) a2).mew();
        }
    }
}
```