CMPE 160.01: Introduction to Object Oriented Programming

PS9: Inheritance and Polymorphism

29/04/2022

1 Overloading and Overriding

Difference between overloading and overriding is subtle, so it is possible to mix two terms. Simply, overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass.

@Override annotation can be used to avoid mixing two. When we use the annotation while we are not overriding a method, we will see an error. That way we can understand that we are doing something wrong.

Below code shows an example of overriding and overloading together, also @Override annotation will be shown:

```
public class OverridingVsOverloading {

   public static void main(String[] args) {
      A a = new A();
      a.p(10);
      a.p(10.0);
   }
}

class B {
   public void p(double i) {
      System.out.println(i * 2);
   }
}

class A extends B {
   // This method overrides the method in B
   @Override
   public void p(double i) {
      System.out.println("Overriding.." + i);
   }
```

```
   // This method overloads the method in B
    public void p(int i) {
    System.out.println("Overloading.." + i);
    }
}
```

2 Polymorphism - Dynamic Binding

We will see an example of dynamic binding by seeing following terms at first:

- Actual Type

- Declared Type

The type that declares a variable is called the variable's **declared type**. A variable of a reference type (not a primitive type like int, float etc.) can hold a null value or a reference to an instance of the declared type. The instance may be created using the constructor of the declared type or its subtype (one of the subtypes, or sub-subtypes). The **actual type** of the variable is the actual class for the object referenced by the variable.

An example can be seen below:

```
public class DynamicBindingDemo {

   public static void main(String[] args) {

      // Declared type for o = 'Object'
      // Actual type for o = 'Person'
      Object o = new Person(); //Polymorphic Call
      Person s = new Student();

      method(o);
      method(s);
   }

   public static void method(Object x) { //Dynamic Binding
      System.out.println(x.toString());
   }
}
```

3 Typecasting and 'instanceof' Operator

We will see an example code that shows implicit and explicit casting, and usage of instanceof operator. Aim is to let compiler know that an object belonging to a superclass is also an instance of a subclass:

```
public static void main(String[] args) {
      // Create and initialize two objects
      Object object1 = new Teacher();
      Object object2 = new Student();

      // Display circle and rectangle
      displayObject(object1);
      displayObject(object2);
   }

   /** A method for displaying an object */
   public static void displayObject(Object object) {
      if (object instanceof Teacher) {
         System.out.println("This is a teacher " +
               ((Teacher)object).toString());
      }
      else if (object instanceof Student) {
         System.out.println("This is a student " +
               ((Student)object).toString());
      }
   }
```

4 Custom Stack Class

As an exercise, we will write two custom stack classes, the first one will contain an ArrayList. We will follow the given UML diagram to make our implementation for the first stack class, called **MyStack**:

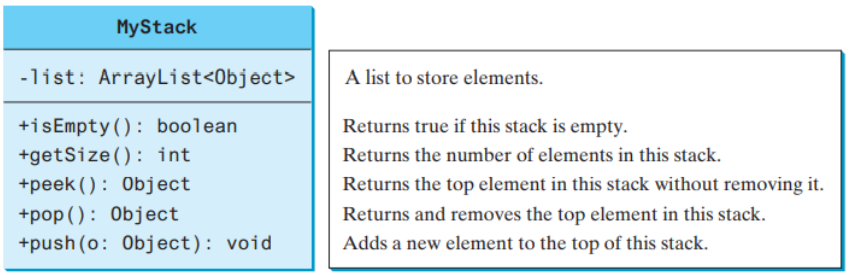

Figure 1: UML diagram for Stack class

Here, we can see that MyStack class contains an ArrayList. This relationship is called as **composition**. It is a 'has-a' relationship. It is different than extending a class, which is an 'is-a' relationship.

For the second implementation, we will create a new class called **MySecondStack**. This time, it will not contain an ArrayList, but it will extend ArrayList class. Implement the same functions as MyStack. (Hint: use 'super' keyword)