Inheritance material part 2 – Examples of polymorphism; The Object Class

This document has some examples of polymorphism and a short discussion of the object class. Note that there is plenty of repetition in terms of concepts already mentioned. Hopefully, this will help to reinforce some of the more salient points regarding inheritance and polymorphism (and not annoy some of you too much!)

Also, in terms of exercise 4 on Inheritance, I'd advise you to not look at the first four pages of this document (you can still skip to the rest without any real impact) until you've attempted/completed the exercise.

Let's look at a simple, practical example of polymorphism.

```
 1 import java.util.*;
 2
 3 public class PolymorphismTester
 4 {
 5    public static void main(String[] args)
 6    {
 7       ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();
 8
 9       //because a superclass reference can refer to a subclass object
10       //we can store ANY type of BankAccount in our list
11       accounts.add(new BankAccount(100));
12       accounts.add(new SavingsAccount(10, 200));
13       accounts.add(new CheckingAccount(300));
14
15
16       //Now, let's add some money to each account
17       for(BankAccount currAccount: accounts)
18       {
19          currAccount.deposit(50);
20       }
21
22       //Check that they all updated
23       for(BankAccount currAccount: accounts)
24       {
25          System.out.println("Balance: " + currAccount.getBalance());
26       }
27
28    }
29 }
```

What we've said about polymorphism is that the correct overridden method will be invoked at run-time depending on what object type is being referenced.

In other words, when we invoke `deposit()` for a CheckingAccount object it should automatically call the overridden deposit() method (remember that the code in CheckingAccount should also increment the transactionCount). This means that polymorphism should be taking place at line 19.

Question: How do I verify that it - *polymorphism* - is taking place?

Answer: Print out the transactionCount value for any CheckingAccount objects (provide a getMethod() if necessary).

Problem: As already stated, the type of the reference – BankAccount – means that I can only call methods declared in the BankAccount class *EVEN THOUGH* the reference could be pointing at a CheckingAccount object – i.e. even if the CheckingAccount class has a getTransactionCount() method I cannot call it[1].

Solution:

1. I check to see if the reference is actually referring to a CheckingAccount object.
2. If it is, I need to set **another** reference of type CheckingAccount to point to the same object. Now I can call the extra/new methods declared in the subclass.

So, how do I actually do the two things outlined above.

For (1) I use a special boolean operator called `instanceof` which can tell me if I'm referring to a CheckingAccount object.

(2) is a little bit messy. I assign the BankAccount reference to a new CheckingAccount reference (so that they both point at the same object). Unfortunately, if I try to do this directly the left-hand-side and the right-hand-side of the assignment operation are of different types and many of us already know that most (***strongly typed***) high-level languages are very picky about that sort of thing.

Therefore I have to "cast" the BankAccount reference to a CheckingAccount reference to tell the compiler that it is actually a CheckingAccount reference.

Let's see this idea in operation:

[1] Of course, this rule is there for good reason. It would be very dangerous to try and call a method that may/may-not be available in a subclass.

```
//Check that they all updated
for(BankAccount currAccount: accounts)
{
  System.out.print("Balance: " + currAccount.getBalance());

  //Check if current reference is referencing a CheckingAccount object
  if (currAccount instanceof CheckingAccount)
  {
    //make another reference (temp) point to the object - the difference is
    //that is has the correct type
    CheckingAccount temp = (CheckingAccount)currAccount;

    System.out.print(" Transaction Count : " + temp.getTransactionCount());

    //Note that I could have done it in one statement but
    //the above mechanism is less likely to confuse
    //System.out.print(" Transaction Count : " +
                    ((CheckingAccount)currAccount).getTransactionCount());
   }
   //Add in a carraige return for formatting
   System.out.println();

}
```

This will result in the following output:

```
Balance: 150.0
Balance: 250.0
Balance: 350.0 Transaction Count : 1
```

And from that output, we can verify that polymorphism has indeed taken place: The same method call in the loop has caused different behaviour for different types of objects.

Remember that run-time polymorphism is possible because of overriding.

Another example of polymorphism using BankAccount

If you look at the BankAccount class you'll see a new method called transfer():

```
/**
   Transfers money from the bank account to another account
   @param amount the amount to transfer
   @param other the other account
*/
public void transfer(double amount, BankAccount other)
{

   this.withdraw(amount);
   other.deposit(amount);
}
```

Due to inheritance, any subclass can call this method AND any subclass can also be passed in as the second parameter. For example:

```
SavingsAccount momsSavings
      = new SavingsAccount(5, 200);

CheckingAccount harrysChecking
      = new CheckingAccount(100);

harrysChecking.transfer(30, momsSavings);
```

In the above example, `this` will refer to harrysChecking and because money is withdrawn from his CheckingAccount the transactionCount will be updated (because polymorphism means that it is the withdraw() method from the CheckingAccount class that is automcatically invoked).

Hopefully, you can appreciate that this sort of thing must be a run-time mechanism.

(Yet) Another example of polymorphism in the context of inheritance.

In the context of inheritance, ***the facility that provides this polymorphic behaviour is the capability of superclass references to actually refer to subclass objects***. This means that if we have a superclass reference referring to a subclass object we can invoke the subclass methods via polymorphism (Caveat:- both the superclass and subclass must have a method(s) with the same signature for this feature to be possible – i.e. you have to override[2] a method/methods in the subclass).

Ok, maybe an example will help to clarify this:- basically what is impressive in the example (on pages 6 and 7) is the `for` loop :-

```
for(Person currPerson : list)
{
   currPerson.printDetails(); //Polymorphism means "one interface
                              //- multiple forms"
   System.out.println();
}
```

This is because the same line of code - `currPerson.printDetails();` actually invokes different versions of `printDetails()` at run-time (depending on what type of object `currPerson` actually refers to).

This is important – the same (line of) code **invokes different behaviour at run-time** (not compile-time). That's what polymorphism is.

Note: the following example is illustrative mainly because class methods rarely print directly to the screen – they usually return information to the calling code as a string, perhaps, and the calling code can then choose to print the information, if it wishes. I've deliberately ignored this notion to better illustrate how polymorphism works.

---

[2] ***Overridden methods*** are methods that are redefined within an **inherited or subclass**. They have the ***same*** signature (i.e. return-type + method-name + parameters (same amount and same types for each parameter)).

```java
public class Person
{
   private String name;
   int age;

   public Person()
   {
   }
   public Person(String inName, int inAge)
   {
      name = inName;
      age = inAge;
   }

   public void printDetails()
   {
      System.out.print("Name: " + name + " Age: " + age);
   }
}
```

```java
public class Student extends Person
{
   int studentId;

   public Student(String inName, int inAge, int inId)
   {

      super(inName, inAge);
      studentId = inId;
   }

   public void printDetails()
   {
      super.printDetails();
      System.out.print(" ID " + studentId);
   }
}
```

```
//This shows inheritance but also provides an example of
//an object-oriented feature called polymorphism
class InheritanceTesterStudentPerson
{
   public static void main(String args[])
   {
      ArrayList<Person> list = new ArrayList<Person>();

      list.add(new Person("Jim Jones", 27));
      list.add(new Person("Mary Smith",30));

      System.out.println("Current List : ");
      for(Person currPerson : list)
      {
         currPerson.printDetails();
         System.out.println();
      }


      //The next bit seems a bit weird - I'm adding a Student object
      //to an ArrayList of type Person.

      //Java allows you to do this - a superclass reference
      //can point to (reference) a subclass object
      list.add(new Student("Mike Murphy", 19, 1234));


      System.out.println("Current List : ");
      //The for loop actually uses polymorphism -
      //What does this mean?
      //It means that the correct printDetails() method
      //will be invoked automatically
      //i.e. if the current object is a Person it will invoke
      //printDetails from the Person class and if the current object
      //is a Student it will invoke printDetails from the Student class
      for(Person currPerson : list)
      {
         currPerson.printDetails(); //Polymorphism means "one interface

                                    //- multiple forms"


         System.out.println();
      }


   }
}
```

Note also, that polymorphism allows old-code to execute new code – imagine I have a method in some class (we'll call it FairlyPointlessClass – while the class may be fairly pointless, the concept that it illustrates is not) that does the following :-

```
public class FairlyPointlessClass

{

	public void printObject (Object anObject)

	{

		System.out.println("Printing object details: " +

					anObject.toString());

	}

}
```

ASIDE: The parameter of `printObject()` is of type `Object`. If you know nothing about the `Object` class, then you may want to skip to the next section of the document and then return to the description below.

Think about what this method does (and is capable of doing).

Well, we know:-

1. if we include any object in a System.out then java invokes that object's `toString()` method (and will thus print out whatever string that method returns). I've actually explicitly done this in the example above.
2. All Classes automatically inherit from the `Object` class (which has a `toString()` method and several others)
3. Even though our parameter is of type "reference to an instance of the `Object` class" it can, in fact, also be a reference to an instance of any Class that inherits from the `Object` class – i.e. we can actually pass a reference to ANY object in to this method.
4. If the object that we pass in has overridden the `toString()` method then this is what gets called – so, if someone writes a new class 100 years from now, our method above will work with it and actually invoke code in the new class that it knows nothing about (other than it is contained in a method called `toString()`).

Exercise-

Copy the class above and also write a new class called MyClass that overrides the toString() method (for example get it to return the string "I am a totally brand new object") - you don't need any other instance fields or methods.

Now write a tester class that creates an object of FairlyPointlessClass and an object of MyClass. Invoke the printObject method by passing in the MyClass object (reference).

The Object Class

Object class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived. Therefore, the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

1. Every class in java automatically inherits the Object class, either directly or indirectly.

2. The Object class contains a number of methods that are designed to be overridden – the most important ones for us (at this point in time) are the `toString()` method and the `equals()` method.

3. Because we are overriding, it is vital that we maintain **the exact same signature** in our new classes; for equals this is `public boolean equals(Object obIn)`. Note that the type of the parameter is **`Object`**.

4. Of course, when we invoke the method we will pass in objects of the type that we are dealing with, e.g. `if (personOb1.equals(personOb2))` .
   This is fine since we already know that a superclass reference – obIn – can refer to any subclass objects (basically anything since all classes are subclasses of Object).

5. In the equals method we must cast the parameter to the correct class type so that we can use it properly. Below is an example of a Coin class which represents the name of the coin and the monetary value:

```
//Used to illustrate the use of the overridden equals method
public class Coin
{
   private double value;
   private String name;

   public Coin(double v, String n)
   {
      value = v;
      name = n;
   }

   //When we override, we must maintain the same signature as Object superclass
   //The argument is of type Object BUT it CAN BE ANY SUBCLASS ALSO!!!!
   public boolean equals(Object otherObject)
   {
      //Even though we assume that the argument is of type Coin, we must cast
      //the object so that we can use it
      Coin other;
      //other = (Coin) otherObject; //could do it like this
```

```
      //The code below is actually safer to cast than above - why?
       //Because in theory, the argument could be any object reference (String,
      //BankAccount, etc.)
       if (otherObject instanceof Coin)
       {
          other = (Coin) otherObject;
       }
       else
       {
          return false; // the two objects cannot be equal
       }

       if ((this.value == other.value) && //Note == for primitives
             (this.name.equals(other.name)))
       {
          return true;
       }
       else
       {
          return false;
       }
   }

   public String toString()
   {
      return "[Name: " + name + ", Value: " + value + "]";
   }
}
```