OOJ Lecture 9
Inheritance and Polymorphism

- Reading: Savitch, Chapter 7
- Reference: Big Java, Horstman, Chapter 9

Objectives

- To understand the concept of polymorphism and Dynamic binding

Polymorphism

- It denotes the principle that behaviour can vary depending on the actual type of an object

Polymorphism

- An interface variable holds a reference to an object of a class that realises the interface:

```
Measurable x;          //no type yet
x = new BankAccount(10000);
x = new Coin(0.1, "dime");
```

- You can never construct an interface! An interface can only be assigned by a reference

```
x = new Measurable(); // ERROR
```

- You can call any of the interface methods:

```
double m = x.getMeasure();
```

- Which method is called?

Polymorphism

- Depends on the actual object.
- If x refers to a bank account, then it calls
  `BankAccount.getMeasure`
- If x refers to a coin, then it calls
  `Coin.getMeasure`
- Polymorphism (Greek - many shapes):
  - The type of the object determines the method that is called
  - Referred to as *late binding*.
  - Resolved at runtime

Polymorphism & Overloading

- Polymorphism is different from overloading, but is closely related to overriding.
- Method overloading:
  - More that one method in a class with the same name but different parameters
  - Overloading is resolved by the compiler (early binding)

Overloading Example

```
class Rabbit
{ ..
  int run(int duration, boolean zigzag){ ... }
  // method 1
  char  run(int duration)  { ... }
  // method 2
}
```

- The two methods must have different sets of parameters.

- When a call comes, which method does the compiler choose?
  - It chooses the one matching the parameters.
    E.g. bunny.run(7) calls method 2,
    bugs.run(5,true) calls method 1.
    If a call matches two methods, program error.

Polymorphism Example

```
class Animals
{
  int useless = 0;
  void wish()
  {
     System.out.println("I want to go home");
  }
  void speech()
  {
     System.out.print("Thank you.”);
     wish();          //call wish()
  }
}
```

Polymorphism Example - cont'd

```
class Rabbit extends Animals
{
  void wish()
  {
     System.out.println("I want a carrot");
                                        //override wish()
  }
}
class Turtle extends Animals
{
  void wish()
  {
     System.out.println("I want a shrimp");
                                      //override  wish()
  }
}
interface Speeches {
     void wish();
     void speech();
}
```

Polymorphism Example - cont'd

```
class Animals implements Speeches  {
  int useless = 0;
  public void wish() {
      System.out.println("I want to go home");
  }
  public void speech() {
      System.out.print("Thank you.");
      wish();  //call wish()
  }
}

class Rabbit extends Animals {
  public void wish() {
      System.out.println("I want a carrot,");
      // override wish()
  }
}

class Turtle extends Animals {
  public void wish() {
      System.out.println("I want a shrimp,");
  }
}
```

Polymorphism Example - cont'd

```
class AnimalWishes
{
   public static void main(String args[])
   {   Animals[] animals = new Animals[3];
       animals[0] = new Turtle();
       animals[1] = new Rabbit();
       animals[2] = new Turtle();
       for (int i = 0; i < animals.length;i++)
                                    animals[i].speech();
   }
}
```

What is the output?

> **javac Animals.java AnimalWishes.java**
> **java AnimalWishes**
> **Thank you.I want a shrimp**
> **Thank you.I want a carrot**
> **Thank you.I want a shrimp**

- In Java polymorphism means: when the superclass is legal, the subclass is also legal.

Another Polymorphism Example

```
//Reference: Horstman, Chapter 9
class BankAccount{ …
   void transfer(BankAccount other,  double amount)
   {  withdraw(amount);
      other.deposit(amount);
   }
…}
class test {
   public static void main(String args[])
   { …
      BankAccount momsAccount =
                              new BankAccount();
      CheckingAccount HarryCheck=
                             new CheckingAccount(10);
      momsAccount.transfer(harryCheck, 100);
      …
   }
}
```

An Example of Polymorphism

- Generic method:
  ```
  public void transfer(BankAccount other,
                       double amount)
  ```
  Works with any kind of bank account (plain, checking, savings)
- Subclass object reference converted to superclass reference *other*
  ```
  momsAccount.transfer(harrysChecking,1000);
  ```
- Note polymorphism:
  `other.deposit(amount)` calls `CheckingAccount.deposit` (and charges transaction fee), not BankAccount.deposit

Polymorphism

- Why not just declare parameter as `Object`?
  - `Object` class doesn't have `deposit` method

How do Programs Know Where to Go Next?

- Programs normally execute in sequence
- Non-sequential execution occurs with:
  - selection (if/if-else/switch) and repetition (while/do-while/for)
    (depending on the test it may not go in sequence)
  - method calls, which jump to the memory location that contains the methods' instructions and return to the calling program when the method had finished execution
- One job of the compiler is to try to figure out the memory addresses for these jumps
- The compiler cannot always know the address
  - sometimes it needs to be determined at run time

Static and Dynamic Binding

- *Binding*: determining the memory addresses for jumps
- *Static*: done at compile time
  - also called *offline* or *early binding*
- *Dynamic*: done at run time
- Compilation is done *offline*
  - it is a separate operation done before running a program
- Binding done at compile time is static, and
- Binding done at run time is dynamic
  - also called *late binding*

Example of Dynamic Binding: General Description

- Subclasses call a method in their superclass which calls a method that is overridden (defined) in each of the subclasses
  - the parent/super class is compiled separately and before the subclasses are even written
  - the compiler cannot possibly know which address to use
  - therefore the address must be determined (bound) at run time

Dynamic Binding: Specific Example

Superclass: `Animal`

- Defines methods: `wish()` & `speech()`
- `Speech()` calls `wish()`

subclasses: `Rabbit` & `Turtle`

- Inherit `speech()`
- Override `wish()`
- If a call is made to speech, then wish() is called – who's wish()?

- The `Animal` class is compiled before `Rabbit` & `Turtle` are even written, so the address of `wish()` in the subclasses cannot be known then
  - it must be determined during run time, i.e. dynamically

Polymorphism

- Using the process of dynamic binding to allow different objects to use different method actions for the same method name
- Polymorphism means different shapes
- Now the term usually refers to use of dynamic binding

Another Example – you may test it

```
            /** Polymorphism.java */
import java.util.Random;

abstract class Animal
{
    public abstract void iAmA();
    public void speak()
    { System.out.println("Burp!"); }
}
class Bird extends Animal
{
    public void iAmA()
    { System.out.println("I am a Bird."); }
    public void speak()
    { System.out.println("Cheep!"); }
}
```

Polymorphism Example - cont'd

```
class Dog extends Animal
{
    public void iAmA()
    { System.out.println("I am a Dog."); }
    public void speak()
    { System.out.println("Bark!"); }
}
class Snake extends Animal
{
    public void iAmA()
    { System.out.println("I am a Snake."); }
    public void speak()
    { System.out.println("Ssssss!"); }
}
class Human extends Animal
{
    public void iAmA()
    { System.out.println("I am a human."); }
}
```

Polymorphism Example - cont'd

```
/** Class to demonstrate polymorphism. */
public class Polymorphism
{
  static Random dice = new Random();
  /** method to randomly catch an animal
                              for our "zoo". */
  static Animal catchAnimal()
  {
     int iRoll = Math.abs(dice.nextInt()%3);
     switch (iRoll)
     {       case 0:
                 return new Bird();
             case 1:
                 return new Dog();
             case 2:
                 return new Snake();
      }
      return null;
  }
```

Polymorphism Example - cont'd

```
/* main method to invoke from JVM.
   We catch a bunch of animals, put them in
   the zoo (an array) and let them speak! */
  public static void main(String args[])
  {
        Animal zoo[] = new Animal[6];
        for (int i=0; i < 6; i++)
            zoo[i] = catchAnimal();

        for (int i=0; i < 6; i++)
        {   zoo[i].iAmA();
            zoo[i].speak();
        }

        human aHuman = new human();
        aHuman.iAmA();
        aHuman.speak();
    }
}
```

Instantiate this, what is output? Try it in practical class.