CMSC 202

Inheritance II

Inherited Constructors?

An Employee constructor cannot be used to create HourlyEmployee objects. Why not?

We must implement a specialized constructor for HourlyEmployees. But how can the HourlyEmployee constructor initialize the private instance variables in the Employee class since it doesn’t have direct access?

The **super** Constructor

- A derived class uses a constructor from the base class to initialize all the data inherited from the base class
  - In order to invoke a constructor from the base class, it uses a special syntax:

```
public DerivedClass(int p1, int p2, double p3)
{
  super(p1, p2);
  derivedClassInstanceVariable = p3;
}
```

  - In the above example, `super(p1, p2);` is a call to the base class constructor

The **`super`** Constructor

- A call to the base class constructor can never use the name of the base class, but uses the keyword **`super`** instead
- A call to **`super`** must always be the first action taken in a constructor definition
- An instance variable cannot be used as an argument to **`super.`** **Why not?**

The **super** Constructor

- If a derived class constructor does not include an invocation of **super**, then the no-argument constructor of the base class will automatically be invoked
  - This can result in an error if the base class has not defined a no-argument constructor
- Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, an explicit call to **super** should almost always be used.

HourlyEmployee Constructor

```
public class HourlyEmployee extends Employee
{
   private double wageRate;
   private double hours;    // for the month

   // the no-argument constructor invokes
   // the Employee (super) no-argument constructor
   // to initialize the Employee instance variables
   // then initializes the HourlyEmployee instance variables

   public HourlyEmployee( )
     {
        super( );
        wageRate = 0;
        hours = 0;
     }
```

HourlyEmployee Constructor

```
// the alternative HourlyEmployee constructor invokes an
// appropriate Employee (super) constructor to initialize
// the Employee instance variables (name and date), and then
// initializes the HourlyEmployee rate and hours

public HourlyEmployee(String theName, Date theDate,
                    double theWageRate, double theHours)
  {
      super(theName, theDate);
      if ((theWageRate >= 0) && (theHours >= 0))
      {
          wageRate = theWageRate;
          hours = theHours;
      }
      else
      {
           System.exit(0);
      }
  }
```

Review of Rules For Constructors

- Constructors can chain to other constructors:
  - in own class, by invoking **`this(…)`**;
  - in parent class, by invoking **`super(…)`**;
- If there is an explicit call to **`this(…)`** or **`super(…)`**, it must be the very first statement in the body
  - It must come even before any local variable declarations
- You can have call to either this() or super(), but not both
- If you don’t have explicit call to this() or super(), an implicit call to a no-arg super() is implicitly inserted

Review of Rules For Constructors

- If your class has no explicit constructor, Java automatically provides a no-arg constructor for you

- Implied by above rules:
  At least one constructor will be called at each class level up the inheritance hierarchy, all the way to the top (Object)

Access to a Redefined Base Method

- Within the definition of a method of a derived class, the base class version of an overridden method of the base class can still be invoked
  - Simply preface the method name with super and a dot

```
// HourlyEmployee’s toString( ) might be
public String toString( )
{
  return (super.toString() + "$" + getRate( ));
}
```

- However, using an object of the derived class outside of its class definition, there is no way to invoke the base class version of an overridden method

You Cannot Use Multiple `supers`

- It is only valid to use `super` to invoke a method from a direct parent
  - Repeating `super` will not invoke a method from some other ancestor class
- For example, if the `Employee` class were derived from the class `Person`, and the `HourlyEmployee` class were derived form the class `Employee` , it would not be possible to invoke the `toString` method of the `Person` class within a method of the `HourlyEmployee` class

```
super.super.toString() // ILLEGAL!
```

You Cannot Use Multiple `supers`

- Why this restriction (i.e., no super.super.method())?
- Because Java enforces strict encapsulation
- Each class has *complete* control over its interface
    - A client using class X (either as local variable, or instance variable for composition) can only access class X's *public* instance variables and methods
    - Even a derived class is a "client" of sorts, with the base class presenting a controlled interface to classes extending it
        - But access controls can be defined differently for clients versus derived classes (e.g., `protected` visibility modifier)
- Strictly layered management style: no "skip-levels" allowed (going to your boss's boss)

An Object of a Derived Class Has More than One Type

- An object of a derived class has the type of the derived class, and it also has the type of the base class

- More generally, an object of a derived class has the type of every one of its ancestor classes
  - Therefore, an object of a derived class can be assigned to a variable of any ancestor type

An Object of a Derived Class Has More than One Type

- An object of a derived class can be plugged in as a parameter in place of any of its ancestor classes
- In fact, a derived class object can be used anyplace that an object of any of its ancestor types can be used
- Note, however, that this relationship does not go the other way
  - An ancestor type can never be used in place of one of its derived types

Base/Derived Class Summary

Assume that class D (Derived) is derived from class B (Base).

1. Every object of type D **is a** B, but not vice versa.

2. D is a more specialized version of B.

3. **Anywhere an object of type B can be used, an object of type D can be used just as well**, but not vice versa.

(Adapted from: ***Effective C++***, 2nd edition, pg. 155)

`Protected` Access

- If a method or instance variable is modified by `protected` (rather than `public` or `private`), then it can be accessed *by name*
  - Inside its own class definition
  - Inside any class derived from it
  - In the definition of any class in the same package
- The `protected` modifier provides very weak protection compared to the `private` modifier
  - It allows direct access to any programmer who defines a suitable derived class
  - Therefore, instance variables should normally **not** be marked `protected`

“`Package`” Access

- If a method or instance variable has no visibility modifier (`public` `private`, or `protected`), it is said to have “package access”, and it can be accessed *by name*
  - Inside its own class definition
  - In the definition of any class in the same package
  - *BUT NOT inside any class derived from it*
- So, the implicit “package” access provides slightly stronger protection than the `protected` modifier, but is still very weak compared to the `private` modifier
  - By design, it is used when a set of classes closely cooperate to create a unified interface
  - By default, it is used by novice programmers to get started without worrying about visibility modifiers or packages

Tip: Static Variables Are Inherited

- Static variables in a base class are inherited by any of its derived classes
- The modifiers **`public`**, **`private`**, and **`protected`** have the same meaning for static variables as they do for instance variables

The Class `Object`

- In Java, every class is a descendent of the class ***`Object`***
  - `Object` is the root of the entire Java class hierarchy
  - Every class has `Object` as its ancestor
  - Every object of every class is of type `Object`, as well as being of the type of its own class (and also all classes in between)
- If a class is defined that is not explicitly a derived class of another class, it is by default a derived class of the class `Object`

The Class **`Object`**

- The class **`Object`** is in the package **`java.lang`** which is always imported automatically
- Having an **`Object`** class enables methods to be written with a parameter of type **`Object`**
  - A parameter of type **`Object`** can be replaced by an object of any class whatsoever
  - For example, some library methods accept an argument of type **`Object`** so they can be used with an argument that is an object of any class
  - Recall the ArrayList class (an old form of it) we studied earlier: the store and retrieve methods were declared to work on instances of type **`Object`**

The Class **`Object`**

- The class `Object` has some methods that every Java class inherits
  - For example, the `equals` and `toString` methods

- Every object inherits these methods from some ancestor class
  - Either the class `Object` itself, or a class that itself inherited these methods (ultimately) from the class `Object`

- However, these inherited methods should be overridden with definitions more appropriate to a given class
  - Some Java library classes assume that every class has its own version of such methods

The Right Way to Define **`equals`**

- Since the **`equals`** method is always inherited from the class **`Object`**, methods like the following simply overload it:

```
public boolean equals(Employee otherEmployee)
   { . . . }
```

- However, this method should be **overridden**, not just overloaded:

```
public boolean equals(Object otherObject)
   { . . . }
```

Why **equals()** Must be Overridden

Imagine we have:

```
public class Point {
  public int x, y;
  … // Stuff here like constructors, etc.
  public boolean equals(Point otherPt) {
    return (x == otherPt.x && y == otherPt.y);
  }
}
public class Point3D extends Point {
  public int z;
  public boolean equals(Point3D otherPt) {
    return (x == otherPt.x && y == otherPt.y && z == otherPt.z);
  }
}
  …
  Point pt2d = new Point(1.0, 2.0);
  Point3D pt3d = new Point3D(1.0, 2.0, 3.0);
  if (pt3D.equals(pt2D))
    System.out.println(“pt2d and pt3D equal”);
```

What will it print out?

The Right Way to Define **`equals`**

- The overridden version of **`equals`** must meet the following conditions
  - The parameter **`otherObject`** of type **`Object`** must be type cast to the given class (e.g., **`Employee)`**
  - However, the new method should only do this if **`otherObject`** really is an object of that class, and if **`otherObject`** is not equal to **`null`**
  - Finally, it should compare each of the instance variables of both objects

A Better **equals** Method for the Class **Employee**

```
public boolean equals(Object otherObject)
{
  if(otherObject == null)
    return false;
  else if(getClass( ) != otherObject.getClass( ))
    return false;
  else
  {
    Employee otherEmployee = (Employee)otherObject;
    return (name.equals(otherEmployee.name) &&
      hireDate.equals(otherEmployee.hireDate));
  }
}
```

The `getClass()` Method

- Every object inherits the same `getClass()` method from the `Object` class
  - This method is marked `final`, so it cannot be overridden
- An invocation of `getClass()` on an object returns a representation *only* of the class that was used with `new` to create the object
  - The results of any two such invocations can be compared with `==` or `!=` to determine whether or not they represent the exact same class

```
(object1.getClass() == object2.getClass())
```

Basic Class Hierarchy Design

- How many levels of classes should we create?
  - Two extremes:
    - MovableThing -> A1981BlueMiataWithBlackVinylTop vs.
    - Vehicle->Car->Car2Door->Convertible2Door->Miata->BlueMiata->…
    - or something in between, perhaps? Yes…
- Create intermediate classes where you do—or might later—want to make a distinction that splits the tree
- It is easier to create than take away intermediate classes.
- What to put at a given level?
  - Maximize abstracting out common elements
  - But, think about future splits, and what is appropriate at given level