Polymorphism II

CMSC 202

Topics

- Constructors and polymorphism
- The clone method
- Abstract methods
- Abstract classes

Constructors and Polymorphism

- A constructor for the base class is automatically called during construction of a derived class.
- This call propagates up the inheritance hierarchy until the constructor for every base class is called.
- Why does this make sense?
  - The constructor’s job is to see that the object is completely built.
  - The derived class cannot have access to the base class’ private instance variables.
  - The base class constructor must be called to initialize its instance variables.
  - Therefore, all base class constructors must be called to fully initialize the entire derived class object.

More Vehicles

```java
public class Aircraft extends Vehicle {
   public Aircraft(){
      super();
      System.out.println("Aircraft");
   }
}

public class Automobile extends Vehicle {
   public Automobile(){
      super();
      System.out.println("Automobile");
   }
}
```

```java
public class Watercraft extends Vehicle {
   public Watercraft(){
      super();
      System.out.println("Watercraft");
   }
}

public class Vehicle {
   public Vehicle(){
      System.out.println("Vehicle");
   }
}
```

- In each constructor we are making explicit calls to base class constructors.
  - Which constructor is executed first?

Vehicle Station

```
public class VehicleStation {

   private Vehicle auto = new Automobile();
   private Aircraft aircraft = new Aircraft();

   public VehicleStation(){
      System.out.println("Vehicle Station");
   }

   public static void main(String[] args){
      VehicleStation station = new VehicleStation();
   }
}
```

Output

```
Vehicle
Automobile
Vehicle
Aircraft
Vehicle Station
```

Order of Construction

Note that base class constructors are called implicitly if there is no explicit call

`super( );`

1. Base class constructors, recursively ***from the top*** of the hierarchy
2. Instance variables in order of declaration
3. The body of the derived class constructor

Derived Class Copy Constructors

Derived class copy constructors must make an explicit call to the base class copy constructor.

```
public Aircraft(Aircraft other){
   super(other);  // polymorphism
   // initialize aircraft instance variables
}
```

- This copy constructor will invoke its base class' copy constructor.
  - We can reuse the code we have created to copy Vehicles because an Aircraft is a Vehicle.
  - Java will upcast the other Aircraft to a Vehicle.

```
public Vehicle(Vehicle other){
   // initialize Vehicle instance variables
}
```

A First Look at the `clone` Method

- Every object inherits a method named `clone` from the class `Object`.
  - The method `clone` has no parameters.
  - Its purpose is to return a deep copy of the calling object.

- However, the inherited version of the method was not designed to be used as is.
  - Each class is expected to override it with a more appropriate version.

A First Look at the `clone` Method

- The heading for the `clone` method defined in the `Object` class is:

```
protected Object clone()
```

- The heading for a `clone` method that overrides the `clone` method in the `Object` class can differ somewhat from the heading above.
  - A change to a more permissive access, such as from protected to public, is always allowed when overriding a method definition.
  - Changing the return type from `Object` to the type of the class being cloned is allowed because every class is a descendent class of the class `Object.` This is an example of a ***covariant return type***.

A First Look at the `clone` Method

- If a class has a copy constructor, the **`clone`** method for that class can use it to create the copy returned by the **`clone`** method.

```
public Vehicle clone(){
   // return a copy of this Vehicle
   return new Vehicle(this);
}
```

Another example:

```
public Aircraft clone(){
   // return a copy of this Aircraft
   return new Aircraft(this);
}
```

Pitfall: Limitations of Copy Constructors

- The copy constructor and `clone` method for a class appear to do the same thing.
  - However, there are cases where only a `clone` will work.

Cloning a VehicleStation

```
public class VehicleStation {
    private Vehicle[] vehicles = new Vehicle[3];

    public VehicleStation(){
        vehicles[0] = new Aircraft();
        vehicles[1] = new Automobile();
        vehicles[2] = new Watercraft();
    }

    public VehicleStation(VehicleStation other){
        for(int i = 0; i < vehicles.length; i++){
            vehicles[i] = new Vehicle(other.vehicles[i]);
        }
    }
}
```

Pitfall: Limitations of Copy Constructors

- The statement

  `vehicles[i] = new Vehicle(other.vehicles[i]);`

only copies the base class (Vehicle) part of each Vehicle, not the specific stuff in each derived class.

- We need to call the copy constructor for the derived class to make an appropriate deep copy, but copy constructors must be called by name, and we don't know what kind of Vehicle is really stored in each element of the array.

- If the `clone` method is used instead of the copy constructor, then (because of polymorphism) a true copy is made, even from objects of a derived class (e.g., Automobile, Aircraft, Watercraft).

- **The correct statement is**

  `vehicles[i] = other.vehicles[i].clone();`

Introduction to Abstract Classes

```
public class Employee
{
   private String name;
   private Date hireDate;

   // constructors, accessors, mutators, equals, toString
}

public class HourlyEmployee extends Employee
{
   private double wageRate;
   private double hours; //for the month
   public double getPay( ) {return wageRate * hours;}

   // constructors, accessors, mutators, equals, toString
}

public class SalariedEmployee extends Employee
{
   private double salary; //annual
   public double getPay() { return salary / 12; }

   // constructors, accessors, mutators, equals, toString
}
```

samePay

- Suppose that we decide that it will often be necessary to determine if two Employees have the same pay.
  - We decide to implement a method named samePay in the Employee class.
  - This method should be able to compare the pays for any kinds of Employees.

```
public boolean samePay(Employee other)
{
  return(this.getPay() == other.getPay());
}
```

Problem with samePay

- The method `samePay` calls `getPay`.
    - While `getPay` is defined for `SalariedEmployees` and `HourlyEmployees`, there is no meaningful implementation of `getPay` for a generic Employee.
    - We can’t implement `getPay` without knowing the type of Employee.

- Solution:
    - Require that classes derived from Employee (who know what type they are) implement a suitable `getPay` method that can then be used from `samePay`.
    - Java provides this capability through the use of ***abstract methods***.

Introduction to Abstract Classes

- An abstract method is like a placeholder for a method that will be fully defined in a descendent class.
  - It postpones the definition of a method.
  - It has a complete method heading to which the modifier `abstract` has been added.
  - It cannot be private.
  - It has no method body, and ends with a semicolon in place of its body.

```
public abstract double getPay();
public abstract void doIt(int count);
```

  - The body of the method is defined in the derived classes.
- The class that contains an abstract method is called an ***abstract class***.

Abstract Class

- If a class has at least one abstract method, it must be declared as an ***abstract class***.
- (Note that a class that has no abstract methods may also be declared as abstract, if desired. For example, a class that contains only instance variables.)
- An abstract class must have the modifier `abstract` included in its class heading.

```
public abstract class Employee
{
  private instanceVariables;
  . . .
  public abstract double getPay();
  . . .
}
```

Abstract Class

- An abstract class can have any number of abstract and/or fully defined methods.

- If a derived class of an abstract class adds to or does not define all of the abstract methods,
  - it is abstract also, and
  - must add **`abstract`** to its modifier.

- A class that is not abstract is called a ***concrete class***.

Abstract Employee Class

```
public abstract class Employee
{
  private String name;
  private Date hireDate;
  public abstract double getPay( );

  // constructors, accessors, mutators, equals, toString

  public boolean samePay(Employee other)
  {
     return(this.getPay() == other.getPay());
  }

}
```

Pitfall: You Cannot Create Instances of an Abstract Class

- An abstract class can only be used to derive more specialized classes.
  - While it may be useful to discuss employees in general, in reality an employee must be a salaried worker or an hourly worker.
- An abstract class constructor cannot be used to create an object of the abstract class.
  - However, a derived class constructor will include an invocation of the abstract class constructor in the form of **`super`**.

An Abstract Class Is a Type

- Although an object of an abstract class cannot be created, it is perfectly fine to have a parameter of an abstract class type.
    - This makes it possible to plug in an object of any of its descendent classes.
- It is also fine to use a variable of an abstract class type, as long is it names objects of its concrete descendent classes only.