CMSC 202H

Polymorphism

2nd Lecture

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 Dogs

```
public class Animal
{
   public Animal( )
        {System.out.println("Animal"); }
}
public class Dog extends Animal
{
   public Dog( )
        {System.out.println("Dog"); }
}
public class Poodle extends Dog
{
   public Poodle( )
        {System.out.println("Poodle");}
}

public class Collie extends Dog
{
   public Collie( )
        {Sytem.out.println( "Collie");}
}
```

```
public class Cage
{

  public Cage( )
  { System.out.println("Cage");}

}


public class Zoo extends Cage
{

   public Zoo( )
   { System.out.println("Zoo"); }

}
```

DogKennel

```
public class DogKennel extends Zoo
{
   private Poodle harry = new Poodle( );
   private Collie lassie = new Collie( );

   public DogKennel( ) { System.out.println( "DogKennel"); }

   public static void main (String[ ] args)
   {    DogKennel kennel = new DogKennel( );    }
}

//--- Output ---
Cage
Zoo
Animal
Dog
Poodle
Animal
Dog
Collie
DogKennel
```

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

Order of Execution

The stages of instance initialization are:
(numbering not important: ordering is!)

- Stage 1: space allocated on heap
- Stage 2: All instance variables, direct and inherited, initialized to default values
- Stage 3: Constructor for requested class initiated

Order of Execution

- Stage 4: Call to `this(…)` or `super(…)` executed first
  - Recall: *all* constructors must start with either a call to this() or implicit/explicit call to super() (except for class *Object*)
- Stage 5: After call to `super(…)`, instance variable initializers executed next, in lexical order
  - This is why they cannot be used as arguments to `super()`, or to `this()`
- Stage 6: Constructor body executed

Derived Class Copy Constructors

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

```
public Dog( Dog anotherDog )
{
   // super takes care of the base class "stuff"
   super( anotherDog );      // polymorphism

   // code specific to Dogs follows
}
```

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.
  - NB: It is ***not*** a constructor!

- 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 Animal clone()
{
   // call Animal's copy constructor
  return new Animal(this);
}
```

Another example:

```
public Dog clone()
{
   // Dog's copy constructor
  return new Dog(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 Zoo

```
public class Zoo
{
   Animal[ ] animals = new Animal[3];
   public Zoo( )
   {    animals[0] = new Dog( );
        animals[1] = new Cat( );
        animals[2] = new Pig( );
   }
   // incorrect copy constructor – why?
   public Zoo( Zoo z )
   {
        animals = new Animal[3];
        for (int k = 0; k < 3; k++)
              animals[ k ] = new Animal( z.animals[ k ] );
   }
}
```

Pitfall: Limitations of Copy Constructors

- The statement

animals[k] = new Animal(z.animals[k]);

only copies the base class (Animal) part of each animal, 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 animal 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., `Dog, Cat, Pig`).
- **The correct statement is**

animals[k] = z.animals[k].clone( ) ;

Complexities and Issues with clone()

- Object.clone() does shallow field copies
- Option 2: call super.clone() first to get shallow copy, then modify specific reference fields.
- Object.clone() implementation will throw CloneNotSupportedException if class does not explicitly implement Cloneable interface
  - So, you cannot call super.clone() from just any class
- clone() inherits to all children
  - Once declared “public”, cannot be “disabled” in any descendent class

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

- A class that has at least one abstract method is called an *abstract class.*

- 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 has no abstract methods is called a *concrete class.*
- Note: you *are* technically allowed to declare a class **`abstract`** even though it has no abstract methods.

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.

Additional Topics/Questions

- Are constructors inherited?
- What happens when a child redefines an instance variable?
  - Variables do not overload or override: they “hide”
  - What happens if:
    - parent: “public int x”, child: “public String x”
    - parent: “public int x:, child: “private int x”
    - → then: child-of-child: “x = 42”
- Can a child class define a private method with the same signature as an inherited method?

Additional Topics/Questions

- What happens when a parent method calls a method overridden by the child?
- What happens when a parent's method is called?
  - Recall: parent method can be triggered through inheritance, or via super.someMethod()
  - What happens w/call to myOverriddenMethod() in parent?
  - What happens w/call to private method in parent?
    - …when child has same-named private method?
    - …when child has same-named public method?