8 - 9

Classes and Objects: A Deeper Look (Part II)

© 1992-2007 Pearson Education, Inc. All rights reserved.

Last time

- intro to exceptions (keyword `throw`)
- overloaded constructors (keyword `this`)
- inheritance (keyword `extends`)
- case study: Time and BinaryTime classes

© 1992-2007 Pearson Education, Inc. All rights reserved.

Objectives

- inheritance and composition examples
- keyword `this` and `super`
- `protected` keyword
- `@Override` annotation
- `java.lang.Object` class
- Examples: Time and Clock classes and inheritance

© 1992-2007 Pearson Education, Inc. All rights reserved.

How to reuse code

- Reuse code by reusing classes
  - Create new class from existing class(es)
  - Absorb existing class data and behaviors
  - Enhance with new capabilities

- Composition (compose using existing classes)
- Inheritance (inherit from existing classes)

© 1992-2007 Pearson Education, Inc. All rights reserved.

Composition

- **Composition**
  - **A class can have references to objects of other classes as members**
  - **Sometimes referred to as a *has-a* relationship**

© 1992-2007 Pearson Education, Inc. All rights reserved.

Inheritance

- **Inheritance**
  - **Subclass extends superclass**
    - **Subclass**
      - **More specialized group of objects**
      - **Behaviors inherited from superclass (can customize)**
      - **Additional behaviors**
  - **Sometimes referred to as a *is-a* relationship**

© 1992-2007 Pearson Education, Inc. All rights reserved.

Inheritance example: JPanel class

- MyPanel class extends `JPanel`

```
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import javax.swing.JPanel;

// myPanel class that extends JPanel and setups up painting canvas
public class MyPanel extends JPanel {

   public void paintComponent( Graphics g ) {
        super.paintComponent(g);
        setBackground(Color.BLACK);

        Font font = new Font("Lucida Grande", Font.PLAIN, 32);
        g.setFont(font);
        g.setColor(Color.BLUE);
    }
}
```

© 1992-2007 Pearson Education, Inc. All rights reserved.

Program: Time / Clock class

© 1992-2007 Pearson Education, Inc. All rights reserved.

Class hierarchy

- Direct superclass
  - Inherited explicitly (one level up hierarchy)
- Indirect superclass
  - Inherited two or more levels up hierarchy
- Single inheritance
  - Inherits from only one superclass
  - Java only allows single inheritance (no multiple parents)

© 1992-2007 Pearson Education, Inc. All rights reserved.

Referring to the current object's members with the `this` reference

- The `this` reference
  - Object can access a reference to itself with keyword `this`
  - Non-`static` methods implicitly use `this` when referring to the object's instance variables and other methods
  - Can be used to access instance variables when they are shadowed by local variables or method parameters

© 1992-2007 Pearson Education, Inc. All rights reserved.

Referring to the superclass object’s members with the `super` reference

- The `super` reference
  - Object can access a reference to its superclass (parent) with keyword `super`
  - When a superclass method is overridden in a subclass, the subclass version often calls the superclass version to do a portion of the work
  - prefix the superclass method name with the keyword `super` and a dot (.) separator when referencing the superclass’s method

© 1992-2007 Pearson Education, Inc. All rights reserved.

Overloaded Constructors

- **Overloaded constructors**
  - **Provide multiple constructor definitions with different signatures**
- **No-argument constructor**
  - **A constructor invoked without arguments**
- **The `this` and `super` reference can be used to invoke another constructor**
  - **Allowed only as the first statement in a constructor's body**

© 1992-2007 Pearson Education, Inc. All rights reserved.

`@Override` Annotation

- Annotations
  - meta-data about program, but not part of the executable
  - like “comments” for the compiler and developer tools
  - used to specify programmer’s intentions
  - at sign character @ starts the annotation
- Example: `@Override` annotation
  - informs the compiler that the element is meant to override an element declared in a superclass
  - prevents errors with misspelled method names

© 1992-2007 Pearson Education, Inc. All rights reserved.

`protected` Members

- `protected` access
  - Intermediate level of access protection between `public` and `private`
  - `protected` members accessible by
    - superclass members
    - subclass members
    - Class members in the same package

© 1992-2007 Pearson Education, Inc. All rights reserved.

Software Engineering Observation 9.1

Methods of a subclass cannot directly access `private` members of their superclass.

© 1992-2007 Pearson Education, Inc. All rights reserved.

`java.lang.Object` Class

- Every class (object) implicitly extends **`Object`**
- Methods defined in the **`Object`** class:
  - **`clone`**
  - **`equals`**
  - **`finalize`**
  - **`getClass`**
  - **`hashCode`**
  - **`notify, notifyAll, wait`**
  - **`toString`**

© 1992-2007 Pearson Education, Inc. All rights reserved.

Method	Description
`clone`	This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it is called. When cloning is required for objects of a class, the class should override method `clone` as a `public` method and should implement interface `Cloneable` (package `java.lang`). The default implementation of this method performs a so-called shallow copy—instance variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden `clone` method's implementation would perform a deep copy that creates a new object for each reference type instance variable. There are many subtleties to overriding method `clone`. You can learn more about cloning in the following article: `java.sun.com/developer/JDCTechTips/2001/tt0306.html`

Object methods that are inherited directly or indirectly by all classes. (Part 1 of 4)

© 1992-2007 Pearson Education, Inc. All rights reserved.

Method	Description
`Equals`	This method compares two objects for equality and returns `true` if they are equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override method `equals` to compare the contents of the two objects. The method's implementation should meet the following requirements: • It should return `false` if the argument is `null`. • It should return `true` if an object is compared to itself, as in `object1.equals( object1 )`. • It should return `true` only if both `object1.equals( object2 )` and `object2.equals( object1 )` would return `true`. • For three objects, if `object1.equals( object2 )` returns `true` and `object2.equals( object3 )` returns `true`, then `object1.equals( object3 )` should also return `true`. • If `equals` is called multiple times with the two objects and the objects do not change, the method should consistently return `true` if the objects are equal and `false` otherwise. A class that overrides `equals` should also override `hashCode` to ensure that equal objects have identical hashcodes. The default `equals` implementation uses operator == to determine whether two references *refer to the same object* in memory. Section 29.3.3 demonstrates class `String`'s `equals` method and differentiates between comparing `String` objects with == and with `equals`.

`Object` methods that are inherited directly or indirectly by all classes. (Part 2 of 4)

© 1992-2007 Pearson Education, Inc. All rights reserved.

Method	Description
`finalize`	This `protected` method (introduced in Section 8.10 and Section 8.11) is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. It is not guaranteed that the garbage collector will reclaim an object, so it cannot be guaranteed that the object's `finalize` method will execute. The method must specify an empty parameter list and must return `void`. The default implementation of this method serves as a placeholder that does nothing.
`getClass`	Every object in Java knows its own type at execution time. Method `getClass` (used in Section 10.5 and Section 21.3) returns an object of class `Class` (package `java.lang`) that contains information about the object's type, such as its class name (returned by `Class` method `getName`). You can learn more about class `Class` in the online API documentation at `java.sun.com/j2se/5.0/docs/api/java/lang/Class.html`.

Object methods that are inherited directly or indirectly by all classes. (Part 3 of 4)

© 1992-2007 Pearson Education, Inc. All rights reserved.

Method	Description
`hashCode`	A hashtable is a data structure (discussed in Section 19.10) that relates one object, called the key, to another object, called the value. When initially inserting a value into a hashtable, the key's `hashCode` method is called. The hashcode value returned is used by the hashtable to determine the location at which to insert the corresponding value. The key's hashcode is also used by the hashtable to locate the key's corresponding value.
`notify`, `notifyAll`, `wait`	Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Chapter 23. In J2SE 5.0, the multithreading model has changed substantially, but these features continue to be supported.
`toString`	This method (introduced in Section 9.4.1) returns a `String` representation of an object. The default implementation of this method returns the package name and class name of the object's class followed by a hexadecimal representation of the value returned by the object's `hashCode` method.

Object methods that are inherited directly or indirectly by all classes. (Part 4 of 4)

© 1992-2007 Pearson Education, Inc. All rights reserved.