Overriding Methods & Class Hierarchies

November 15, 2006

Quick Review of Last Lecture

The protected Modifier

• Visibility modifiers affect the way that class members can be used in a child class
• Variables and methods declared with private visibility cannot be referenced by name in a child class
• They can be referenced in the child class if they are declared with public visibility -- but public variables violate the principle of encapsulation
• There is a third visibility modifier that helps in inheritance situations: protected

Appendix E

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class and Interfaces</th>
<th>Methods and variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>default (non-modifier)</td>
<td>visible in its package</td>
<td>visible to any class in the same package as the class.</td>
</tr>
<tr>
<td>public</td>
<td>visible anywhere</td>
<td>visible anywhere</td>
</tr>
<tr>
<td>protected</td>
<td>visible to the enclosing class and all its subclasses</td>
<td>visible to any class in the same package as the class</td>
</tr>
<tr>
<td>private</td>
<td>visible to the enclosing class</td>
<td>not visible to any other class</td>
</tr>
</tbody>
</table>

The protected Modifier

• The protected modifier allows a child class to reference a variable or method directly in the child class
• It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility
• A protected variable is visible to any class in the same package as the parent class
• The details of all Java modifiers are discussed in Appendix E
• Protected variables and methods can be shown with a # symbol preceding them in UML diagrams

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class and Interfaces</th>
<th>Methods and variables</th>
<th>Method</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>accessible to all classes</td>
<td>accessible to all classes</td>
<td>abstract</td>
<td>no</td>
</tr>
<tr>
<td>final</td>
<td>accessible to all classes</td>
<td>accessible to all classes</td>
<td>final</td>
<td>no</td>
</tr>
<tr>
<td>static</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>native</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>override</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>synchronized</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>transient</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>volatile</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
The super Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent’s constructor to set up the “parent’s part” of the object
- The super reference can be used to refer to the parent class, and often is used to invoke the parent’s constructor

A child’s constructor is responsible for calling the parent’s constructor
- The first line of a child’s constructor should use the super reference to call the parent’s constructor
- The super reference can also be used to reference other variables and methods defined in the parent’s class
Modified Book Example

- See `Words2.java` (page 445)
- See `Book2.java` (page 446)
- See `Dictionary2.java` (page 447)

Chapter 8
Sections 8.1 & 8.2

Overriding Methods

- A child class can **override** the definition of an inherited method in favor of its own

- The new method **must have the same signature** as the parent’s method, but can have a different body

- The type of the object executing the method determines which version of the method is invoked

Overriding

- A method in the parent class can be invoked explicitly using the `super` reference

- If a method is declared with the `final` modifier, it cannot be overridden

- The concept of overriding can be applied to data and is called **shadowing variables**

- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overloading vs. Overriding?

- The compiler determines which method is being invoked by analyzing the parameters

```java
float tryMe(int x) // [signature 1] tryMe: int
{ 
    return x + .375;
}

float tryMe(int x, float y) // [signature 2] tryMe: int, float
{ 
    return x*y;
}
```

Method Overloading

- The compiler determines which method is being invoked by analyzing the parameters
Method Overriding

```java
public class Parent {
    public float tryMe(int x) {
        return x + .375;
    }
}

public class Child extends Parent {
    public float tryMe(int x) {
        return x*2;
    }
}
```

Overloading vs. Overriding

- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature

Overloading vs. Overriding

- Overloading lets you define a similar operation in different ways for different parameters
- Overriding lets you define a similar operation in different ways for different object types

Overriding Example

- See `Messages.java` (page 450)
- See `Thought.java` (page 451)
- See `Advice.java` (page 452)

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a class hierarchy

```
Business
\-- RetailBusiness
    \-- Kmart
\-- ServiceBusiness
    \-- Macys
    \-- Kinkos
```
Class Hierarchies

- Two children of the same parent are called siblings
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its ancestor classes
- There is no single class hierarchy that is appropriate for all situations

Employee Class Hierarchy

- The Object Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- Therefore, the `Object` class is the ultimate root of all class hierarchies

The Object Class

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we define the `toString` method, we are actually overriding an inherited definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object’s class along with some other information
Object – the mother of all objects in Java

boolean equals (Object obj)
Returns true if this object is an alias of the specified object.

String toString ()
Returns a string representation of this object.

Object clone ()
Creates and returns a copy of this object.

Object.java

• In fact, Object has more methods as can be seen from the source file.

• java/lang/Object.java

The Object Class

• The equals method of the Object class returns true if two references are aliases
• We can override equals in any class to define equality in some more appropriate way
• As we’ve seen, the String class defines the equals method to return true if two String objects contain the same characters
• The designers of the String class have overridden the equals method inherited from Object in favor of a more useful version

AWT Class Hierarchy

Exceptions Class Hierarchy
Abstract Classes

- An abstract class is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier abstract on the class header to declare a class as abstract:

```java
public abstract class Product {
    // contents
}
```

Abstract Classes

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
- An abstract method cannot be defined as final or static
- The use of abstract classes is an important element of software design – it allows us to establish common elements in a hierarchy that are too generic to instantiate

Abstract Classes

- An abstract class often contains abstract methods with no definitions (like an interface)
- Unlike an interface, the abstract modifier must be applied to each abstract method
- Also, an abstract class typically contains non-abstract methods with full definitions
- A class declared as abstract does not have to contain abstract methods -- simply declaring it as abstract makes it so

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- That is, one interface can be derived from another interface
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the ancestor and child interfaces
- Note that class hierarchies and interface hierarchies are distinct (they do not overlap)

This example shows how multiple inheritance can be faked in java

![Diagram](http://www.vsj.co.uk/pix/articleimages/may05/javathread3.jpg)
Visibility Revisited

- It’s important to understand one subtle issue related to inheritance and visibility
- All variables and methods of a parent class, even private members, are inherited by its children
- As we’ve mentioned, private members cannot be referenced by name in the child class
- However, private members inherited by child classes exist and can be referenced indirectly

Visibility Revisited

- Because the parent can refer to the private member, the child can reference it indirectly using its parent’s methods
- The super reference can be used to refer to the parent class, even if no object of the parent exists

Example

- See FoodAnalyzer.java (page 459)
- See FoodItem.java (page 460)
- See Pizza.java (page 461)

You can use jGrasp to draw diagram like this one

Designing for Inheritance

- As we’ve discussed, taking the time to create a good software design reaps long-term benefits
- Inheritance issues are an important part of an object-oriented design
- Properly designed inheritance relationships can contribute greatly to the elegance, maintainability, and reuse of the software
- Let’s summarize some of the issues regarding inheritance that relate to a good software design
Inheritance Design Issues

- Every derivation should be an is-a relationship
- Think about the potential future of a class hierarchy, and design classes to be reusable and flexible
- Find common characteristics of classes and push them as high in the class hierarchy as appropriate
- Override methods as appropriate to tailor or change the functionality of a child
- Add new variables to children, but don’t redefine (shadow) inherited variables

Inheritance Design Issues

- Allow each class to manage its own data; use the `super` reference to invoke the parent’s constructor to set up its data
- Even if there are no current uses for them, override general methods such as `toString` and `equals` with appropriate definitions
- Use abstract classes to represent general concepts that lower classes have in common
- Use visibility modifiers carefully to provide needed access without violating encapsulation

Restricting Inheritance

- The `final` modifier can be used to curtail inheritance
- If the `final` modifier is applied to a method, then that method cannot be overridden in any descendent classes
- If the `final` modifier is applied to an entire class, then that class cannot be used to derive any children at all
  - Thus, an abstract class cannot be declared as final
- These are key design decisions, establishing that a method or class should be used as is

THE END