Overriding, Hiding, and Applets

CS151

Chris Pollett

Oct. 5, 2005.

Outline

- Overriding Methods
- Subtypes for Interfaces
- Single versus Multiple Inheritance
- Hiding Fields and Class Methods
- More on Applets

Overriding Methods

- Overriding is the introduction of an instance method (i.e., non-static method) in a subclass that has the same name, signature, and return type of a method in the superclass.
- The implementation in the subclass replaces the method in the parent class.
- This differs from overloading where one has methods with the same name in the same class with different signatures.
- Here we are talking about subclasses with a method with both the same name and signature as the parent class.

Example Overriding/Not Overriding

```
class B { public void m(){…}}
class C extends B {public void m(){…}}
//Above is an example of overriding

class B { public void m(){…}}
class C extends B {public void m(int a){…}}
/* Might think above is overloading. Not legal in
   Java; legal in C++ (if modified to C++ syntax)*/
```

Final Methods

- A method that is declared final in a parent class cannot be overriden in a subclass.
- This is done to prevent people from overriding methods by accident that should not be overriden.
- You might want to prevent someone from overriding a method that has to work with other classes in a delicate way. (i.e., the method has to honor some convention or contract.)
- Another reason to use a final method is that it allows the JVM to better optimize your code sometimes.

Invoking Overriden Methods

- Suppose I define a subclass of Point called ColoredPoint. I want two ColoredPoint's to be equal if they are equal as Point's and they have the same Color. So it would be useful to call Point's equals() method when defining ColoredPoint's equals() method.
- To do this one can define in ColoredPoint:

```
public boolean equals(Object other)
{
    if (other == null || !(other instanceof ColoredPoint))
      return false;
    ColoredPoint p = (ColoredPoint)other;
    return (super.equals(p) && color.equals(p.color));
}
```

Restrictions in Java

- Sometimes when we create a subclass of a parent class we want to “forget” some of the methods of the parent class.
- The subclass in this case is called a *restriction* of the parent class.
- For example, you might have a class Polygon and you might want to create a subclass Rectangle which does not support the addVertex method.
- To do this in Java you could either override addVertex in one of two ways:

  (1) public void addVertex(Point p){} //i.e., make it do nothing

  (2) public void addVertex(point p) throws MethodNotSupported

  {throw new MethodNotSupported(“addVertex”);} // give an exception

Subtypes for Interfaces

- Interfaces also define types.
- Both extension and implementation are subtypes in this case.
- A complete definition of the subtype relation can now be given:
  - if C2 extends C1 then C2 is a subtype of C1
  - if I2 extends I1 then I2 is a subtype of I1.
  - if C implements I, then C is a subtype of I.
  - Every interface I is a subtype of Object (will also hold for classes by previous)
  - For every type T, T[] where T is a primitive or reference type, T is a subtype of Object.
  - If T1 is a subtype of T2, then T1[] is a subtype of T2[].

Single versus Multiple Inheritance

- One use of interfaces is to allow a class to implement multiple roles.
- For instance, one might have an interface Student which is implemented by having a getGPA() method and an interface Employee which is implement by having a getSalary() method. Then a class StudentEmployee could implement both these interfaces.
- This is an example of *multiple inheritance*.
- Unlike C++, Java only supports multiple inheritances for interfaces.
- An example headache caused by true multiple inheritances, is if have a diamond shape multiple inheritance relationship. For example, both Employee and Student might be derived from Person which has a **name** field. Then StudentEmployee might end up with two **name** fields, one via Employee and one via Student. So your computer language needs a mechanism to resolve this issue.

Faking True Multiple Inheritance in Java.

- Can use *delegation*:

```
public class EmployeeImpl implements Employee
        {public float getSalary(){/*code for salary */}}
public class StudentImpl implements Student
        {public float getGPA(){/*code for salary */}}
public class StudentEmployee implements Student,Employee
        {
           public StudentEmployee()
           {studentImpl = new StudentImpl();
            employeeImpl = new EmployeeImpl();
           }
           public float getGPA() {return studentImpl.getGPA();}
           public float getSalary() {return employeeImpl.getSalary();}
           protected StudentImpl studentImpl;
           protected EmployeeImpl employeeImpl;
        }
```

Name Collisions among Interfaces

Suppose class A implements X, Y and there is a methods m() in both interface X and interface Y.

- If the two m()'s have different signatures they are considered to be overloaded.
- If they have the same signatures and return type, they are considered to be the same method.
- If they have the same signature but different return types a compile error will occur.
- If they have the same signature and return types but different throws lists, then the union of the exceptions listed is assumed.

Marker Interfaces

- Marker interfaces are empty interfaces that declare no methods or constants.
- They are used to mark that a class has some set of properties.
- For instance, the Cloneable interface is used to indicate a class whose objects can be cloned.

Hiding Fields and Class Methods

- Hiding refers to the introduction of a field or a class method in a subclass with the same name as a field or a class method in a parent class.
- For example, suppose Point has a static method getDescription() which is hidden by ColoredPoint.
- Then:

```
ColoredPoint p = new ColoredPoint(10, 0, Color.blue);
System.out.println(p.getDescription()); /* calls child
    method*/
Point p2 = p;
System.out.println(p2.getDescription()); /* calls parent
    method. Note: this is determined at compile time!*/
```

More on Applets

- An idiom is a common object-oriented coding pattern which is specific to some programming language.
- For example, one might have a Java class that run as both an applet and application:

```
public MyClass extends JApplet
{   public void init() {/* do stuff */}
    public void paint(Graphics g){/* do stuff */}
     public static void main(String args[])
     {
       MyClass myApplet = new MyClass(); // could write own constructor
       JFrame myFrame = new JFrame("MyFrame");
       Container c = myFrame.getContentPane();
       c.add(myApplet);
       myFrame. setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
       myFrame.setSize(100,100); myFrame.setLocation(100,100);
       myApplet.init();
       myFrame.setVisible(true);
       myApplet.start();
     }
}
```

Still More on Applets

- Reading values from param tags

  Between <applet …> and </applet> (or <object..>) we can have:

  <param name="some name" value="some value"> tags. Within our applet we can find out these values using getParameter("some name");

- Drawing String in paint

```
public void paint(Graphics g)
{
    Font font = new Font("Sans-serif",Font.BOLD, 24);
    g.setFont(font);
    String text="hello";
    FontMetrics fm = g.getFontMetrics();
    int length = fm.stringWidth(text);
    //use length to figure out x,y of where to draw then
    g.drawString(text,x,y);
}
```

- Methods java.awt.Graphics class has many other useful methods: fillRect(),drawRect(),drawOval, fillOval, drawLine, etc see online docs.

Reading files in Applets

- An applet is not allowed to read or write files on the client machine.
- It is allowed to read files from the server host.
- For instance, can do:

  URL url = new URL(getDocumentBase(), filename);

  BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()));

  To read from such a file. In our homework the file will be a Jar so want to use JarInputStream or JarURLConnection.