Inheritance: Abstract Classes and Interfaces

COMP 110
Summer II 2012

Peter Lincoln
7/17/2012
Questions?

- Yesterday:
  - More About Inheritance
  - Lab 7
Today in COMP 110

- Lecture
  - Inheritance: Abstract Classes & Interfaces

- Exercise
  - Inheritance
Lecture Outline

- More About
  - equals Method
  - super Keyword
  - final Keyword

- Abstract Classes

- Interfaces
The equals method

- Implements an equivalence relation

- It is reflexive
  - For any non-null reference value x, x.equals(x) should return true
The equals method

- It is *symmetric*
  - For any non-null reference values \( x \) and \( y \), \( x.equals(y) \) should return true if and only if \( y.equals(x) \) returns true.
The equals method

- It is *transitive*
- For any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true
The equals method

- It is \emph{consistent}

- For any non-null reference values \(x\) and \(y\), multiple invocations of \(x.equals(y)\) consistently return \texttt{true} or consistently return \texttt{false}, provided no information used in equals comparisons on the objects is modified.
The equals method

- For any non-null reference value x, x.equals(null) should return false
This implementation is not symmetric

```java
public boolean equals(Object obj) {
    if ((obj != null) && (obj instanceof Student)) {
        Student otherStudent = (Student) obj;
        return (this.id == otherStudent.id);
    }
    return false;
}
```
Back to the Student Example

Why?
  - The instanceof operator will return true if obj is a subclass of Student

    public boolean equals(Object obj)
    {
        if ((obj != null) && (obj instanceof Student))
        {
            Student otherStudent = (Student) obj;
            return (this.id == otherStudent.id);
        }
        return false;
    }
Back to the Student Example

- Fourth try

```java
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if ((obj == null) || (obj.getClass() != this.getClass()))
        return false;

    Student otherStudent = (Student) obj;
    return (this.id == otherStudent.id);
}
```

- The `getClass` method will return the runtime class of an object, so if `obj` is a subclass of `Student` (in this case), this method will return `false`
The super Keyword

- super is like this, except that it provides explicit access to the parent class
  - Call parent class method implementations of overridden public methods
  - Call parent class public constructors
public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public String toString() {
        return "Name: " + name;
    }
}

public class Student extends Person {
    private int id;

    public Student(String name, int id) {
        super(name);
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public String toString() {
        return super.toString() + "Id: " + id;
    }
}

Call Person’s constructor
Call Person’s toString() method
super & Student Class

```java
public class Person {
    // ...

    public String toString()
    {
        return "Name: " + name;
    }
}

public class Student extends Person {
    // ...

    public String toString()
    {
        return super.toString() + ", Id: "+ id;
    }
}
```

Code:

```java
Student std =
    new Student("Apu", 17375);

System.out.println(std.toString());
```

Outputs:

Name: Apu, Id: 17375
The final Keyword

- We’ve seen this before:
  - `public static final int DAYS_PER_WEEK = 7;`

- `final` indicates that something cannot be changed anymore
The final Keyword: Variables

- Static constants
  - `public static final int DAYS_PER_WEEK = 7;`

- Instance variables that can only be set in the constructor
  - `public final int length; // in arrays`
  - `private final String birthplace;`
The **final** Keyword: Methods

- Indicates that a method cannot be overridden
  - `public final void specialMethod()
    {
      // ...
    }`

- Attempts to override are compiler errors
- The method name can still be overloaded
- Marking methods as `final` is uncommon for simple programs
The `final` Keyword: Classes

- Indicates that a class cannot be used as a base class
  - `public final class SomeSealedClass {
    // ... 
  }

- Attempts to inherit from `final` classes are compiler errors
- Marking classes as `final` is uncommon for simple programs
Implementing Base Class Methods

- The prior examples have all had implementations for the overridden base class methods

- What if no default implementation for a base class method makes sense?

- What if a default implementation should never be used?
The abstract Keyword

- Indicates that a method has no body in this class
  - Must be implemented by a derived class

- Example (from a class representing shapes)
  - `public abstract void drawHere();`

- Syntax
  - `public abstract TypeOrVoid methodName(Argument_List);`
The abstract Keyword

- A class that contains abstract methods must also be marked as abstract

Example

- `public abstract class ShapeBase {
   public abstract void drawHere();
}

Syntax

- `public abstract class ClassName {
   Class_Body...
}
Example: Expression Classes

- Suppose we wanted to represent Java expressions as a hierarchy of classes
  - E.g. \((5 + 7) \times -4\)

- What are some of the basic components in simple Java expressions?
  - Literals
  - Binary Operators
  - Unary Operators
Example: Expression Classes

- Suppose we wanted to represent Java expressions as a hierarchy of classes
  - E.g. \((5 + 7) \times -4\)

- What are some of the components in simple Java expressions?
  - Literals: 5, 7, 4 (or -4)
  - Binary Operators: +, *
  - Unary Operators: -
Example: Expression Classes

- What is a common operation regarding expressions?
  - Evaluation

- Name the base class: Expression
  - Method:
    - `public abstract double evaluate();`
Note that the class name and abstract method entry are italicized.
Additional Expression Support

- Literals
- Binary Operators
  - Addition
  - Subtraction
  - Multiplication
  - Division
- Unary Operators
  - Negation

This is starting to look like a hierarchy
Literals

- Concrete class
  - Non-abstract

- Need to store a literal value

- Add getter/setter

- Implement evaluate

```
Expression

+evaluate() : double

Literal

-value : double
+evaluate() : double
+getValue() : double
+setValue(in value : double) : void
```
Binary Operators

- Are there any common elements among binary operator expressions?
  - Left Operand
  - Right Operand
  - Operator

- Do all binary operators perform the same evaluation?
  - No, need another abstract class
getOperatorString() is declared to help simplify toString()

- abstract

In BinOpExpr:

```java
public String toString()
{
    String left = "(" + this.leftOperand + "\)");
    String right = " (" + this.rightOperand + ")";
    return left + getOperatorString() + right;
}
```
Unary Operators

- Are there any common elements among unary operator expressions?
  - Operand
  - Operator

- Do all unary operators perform the same evaluation?
  - No, need another abstract class
Unary Operators (1\textsuperscript{st} Try)

### Expression

- `evaluate() : double`

### UnOpExpr

- `operand : Expression`
- `getOperand() : Expression`
- `setOperand(in operand : Expression) : void`
- `toString() : String`
- `getOperatorString() : String`

### Negation

- `evaluate() : double`
- `getOperatorString() : String`
Operators: Any Commonalities?

Solution: Add another level of hierarchy
Expression Example

- Eclipse Demo
Adding a New Operator

- Suppose we wanted to add exponential?
  - Where would it fit in?
```java
Expression
+evaluate() : double

OperatorExpr
+getOperatorString() : String

BinOpExpr
-leftOperand : Expression
-rightOperand : Expression
+getLeftOperand() : Expression
+getRightOperand() : Expression
+setLeftOperand(in operand : Expression) : void
+setRightOperand(in operand : Expression) : void
+toString() : String

Literal
-value : double
+evaluate() : double
+getValue() : double
+setValue(in value : double) : void
+toString() : String

UnOpExpr
-operand : Expression
+getOperand() : Expression
+setOperand(in operand : Expression) : void
+toString() : String

Addition
+evaluate() : double
+getOperatorString() : String

Subtraction
+evaluate() : double
+getOperatorString() : String

Multiplication
+evaluate() : double
+getOperatorString() : String

Division
+evaluate() : double
+getOperatorString() : String

Negation
+evaluate() : double
+getOperatorString() : String

Exponential
+evaluate() : double
+getOperatorString() : String
```
Interfaces

- Interfaces are like abstract classes with only abstract methods
  - No defined methods or constructors

- Interfaces are useful
  - When there is no useful default implementation for a method in a base type
  - When you want to indicate multiple *is-a* relationships on a derived class
Interface Examples

/**
 * Indicates that the implementing class is measurable.
 */
public interface Measurable {
    /** Returns the perimeter */
    public double getPerimeter();

    /** Returns the area */
    public double getArea();
}

/**
 * Indicates that the implementing class is nameable.
 */
public interface Nameable {
    /** Returns the name */
    public String getName();
}
public class Rectangle implements Measurable {
    public double width;
    public double height;

    public void setDimensions(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double getArea() {
        return width * height;
    }

    public double getPerimeter() {
        return 2 * (width + height);
    }
}

public class Circle implements Measurable {
    private double radius;

    public void setRadius(double radius) {
        this.radius = radius;
    }

    public double getPerimeter() {
        return 2 * Math.PI * radius;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }
}
What if we added a Square class?

```java
public class Rectangle implements Measurable {
    public double width;
    public double height;

    public void setDimensions(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double getArea() {
        return width * height;
    }

    public double getPerimeter() {
        return 2 * (width + height);
    }
}

public class Square implements Measurable {
    public double width;

    public void setWidth(double width) {
        this.width = width;
    }

    public double getPerimeter() {
        return 4 * width;
    }

    public double getArea() {
        return width * width;
    }
}

Should we apply class inheritance to simplify this?
Are all rectangles squares?
  - No

Are all squares rectangles?
  - Yes

But Rectangle has two instance variables (width, height) to Square’s one (width)
  - How can we attribute Square properly?
Multiple Inheritance

- Java does not support multiple inheritance of classes

- However, Java does support multiple inheritance of interfaces
Rectangle & Square Resolution

- Declare a Rectangular interface
  ```java
  public interface Rectangular {
  }
  ```

- Interfaces do not need to include any method headers
  - Useful only for instanceof operator testing
Rectangular Interface

- We could add methods though
  - public interface Rectangular
    {
      public double getWidth();
      public double getHeight();
    }
Rectangle Class

```java
public class Rectangle implements Measurable, Rectangular {
    public double width;
    public double height;
    // Old methods still present
    public double getWidth() {
        return width;
    }
    public double getHeight() {
        return height;
    }
}
```

Square Class

```java
public class Square implements Measurable, Rectangular {
    public double width;
    // Old methods still present
    public double getWidth() {
        return width;
    }
    public double getHeight() {
        return width;
    }
}
```
The implemented methods from the interfaces are repeated in the class’ blocks
Interface Inheritance

- Interfaces can inherit from other interfaces as well
  - Use the extends keyword
  - Supports multiple inheritance

Examples

- public interface Rectangular extends Geometrical
- public interface Trainable extends Callable, Capable
public class **ClassName**
- The class derives from **Object** and implements no interfaces

public class **ClassName** extends **BaseClassName**
- The class derives from **BaseClassName** and implements no interfaces

public class **ClassName** implements **Interface1**, **Interface2**, ...
- The class derives from **Object** and implements one or more interfaces

public class **ClassName** extends **BaseClassName**
  implements **Interface1**, **Interface2**, ...
- The class derives from **BaseClassName** and implements one or more interfaces
public interface InterfaceName
  The interface inherits from nothing

public interface InterfaceName extends BaseInterface1, BaseInterface2, ...
  The interface inherits from one or more base interfaces
This is the UML diagram for one library in the Avatar project (hierarchy only, no methods)
Questions?
Logistics

- Next:
  - Inheritance Exercise