Lecture 36: Polymorphism

Last time:
1. Method overriding  
2. Shadowing  
3. Access issues in inheritance

Today:
1. Project #7 assigned  
2. Object  
3. Polymorphism and abstract methods  
4. Upcasting / downcasting
Project #7 Assigned!

- Project due Sunday, 4/29 at 11 pm
- Project is closed
  - You must complete the project by yourself
  - Assistance can only be provided by teaching assistants (TAs) and instructors
  - You must not look at other students' code
- FINISH!
Recap

- Inheritance occurs when one class *(derived class, subclass)* is defined from another class *(base / parent class, superclass)*.
- To derive a class D from a base class B, use:

```java
public class D extends B { ... }
```
- Derived class inherits all instance variables, methods from base class. It can also define new instance variables, methods
- In derived-class constructor, `super( ... )` can be used to invoke constructor from base class
- Derived class can explicitly refer to entities from base class using `super`, e.g. `super.toString()`
- **Polymorphism**: object in derived class can be used anywhere base class is expected *(a Student “is a” Person!)*
- Derived class can **override** base-class methods (and variables)
  - `final` can be used to disallow overriding
  - Java uses `late binding` to determine which version of method to use
- **protected** modifier exposes declarations to subclasses (and package)
Object

- Recall: inheritance induces “is-a” hierarchy on classes
  - Undergrad “is-a” Student
  - Student “is-a” Person
  - etc.
- Person “is-a” ….?
- Person “is-a”(n) **Object**
More on Object

- Special class at top of class inheritance hierarchy
- Defined in `java.lang` (so available in every program)
- Every class is derived (either directly or indirectly) from `Object`
  - If a class is not derived from anything, it is automatically derived from `Object`
  - e.g.
    ```java
    public class Foo { ...}
    ```
    is equivalent to
    ```java
    public class Foo extends Object {...}
    ```
Structure of Object

- No instance variables
- A number of methods, including:
  - toString()
  - equals (Object o)
    Note: parameter to equals has type Object, so any object can be an argument
- These methods can (and usually should) be overridden
- Class hierarchy:

Your classes are all here

- Student
  - Undergrad
  - GradStudent
- Faculty
  - Instructor
  - Professor

Defined in java.lang.

The classes from the Java class library are here.

- Number
  - Integer
  - Float
  - Double
- String
- Boolean
Polymorphism and Inheritance

- Recall
  - Objects in derived classes can be used wherever objects in base classes are needed
    
    \texttt{e.g. Person \textit{p} = new Student ();}
  
  - Java uses late binding to determine which version of overridden method to use

- This means reference variables like \texttt{p} are **polymorphic**
  
  - Object referred to by \texttt{p} may be from any class derived from \texttt{Person}
  
  - An object is not modified when assigned to \texttt{p}; it retains its original form (\texttt{e.g. Student in above case})
Polymorphism and Arrays

- Example: Create an array of various university people and print
  
  ```java
  Person[ ] list = new Person[3];
  list[0] = new Person("Col. Mustard","000-00-0000");
  list[1] = new Student("Ms. Scarlet","111-11-1111", 1998, 3.2);
  list[2] = new Faculty("Prof. Plum","222-22-2222", 1981);
  for (int i = 0; i < list.length; i++)
    System.out.println( list[i].toString() );
  ```

- What type is list[i]?
  - It can be a reference to any object that is derived from Person
  - Late binding ensures the appropriate toString() will be called
An Inheritance Example

- Goal: picture-drawing program
  - Picture is array of different shapes
  - Program should draw each shape on screen
- Desiderata
  - Shapes to handle initially are circles, rectangles
  - Want to leave door open for new shapes in future
  - Drawing shapes depends on kind of shape (drawing a rectangle different from drawing a circle)
Class Design

- Use inheritance hierarchy for shapes
- Base class (Shape) includes
  - Color field (as in Lecture #34)
  - `drawMe()` method for drawing shape
- Derived classes will override `drawMe()`
Picture Drawing

- **Picture**: array `shapes` of type `Shape[]`
- **To draw picture**, invoke `drawMe()` for each shape:
  ```java
  Shape[] shapes = new Shape[...];
  shapes[0] = new Circle( ... );
  shapes[1] = new Rectangle( ... );
...
  for ( int i = 0; i < shapes.length; i++ )
      shapes[i].drawMe();
  ```
- **Note importance of late binding for this strategy!**

Bonnie Dorr (Adapted from Rance Cleaveland)
Java Code

- **Assume:** class `Point` storing x, y coordinates
- **Code layout**

```java
public class Shape {
    private int color;          // Color of shape
    ...
    public void drawMe () { ... }
}

public class Circle extends Shape {
    private double radius;     // Radius of circle
    private Point center;      // Center of circle
    ...
    public void drawMe () { ... }  // Draws a circle
}

public class Rectangle extends Shape {
    private Point upperLeft;   // Upper left corner
    private Point lowerRight;  // Lower right corner
    ...
    public void drawMe () { ... }  // Draws a rectangle
}
```
Abstract Methods

- Problem: how to implement `drawMe()` method for `Shape`?
  `Shape` class is for generic shapes … insufficient info to draw
- Could just do something ad hoc in implementation:
  - Raise an exception
  - Do nothing
  - Bad idea!
    - Point of `drawMe()` method is to override it in derived classes
    - If we don’t override ad hoc implementation: run-time errors!
- Could leave out `drawMe()` method from `Shape` …
  but then we lose access to `drawMe()` method in picture drawing
- Better solution: Abstract methods/classes
  - What we want is for the compiler to know that `Shape` is an incomplete class: it declares but does not define some methods (e.g. `drawMe()`)
    that will be implemented in derived classes
  - To do this we use abstract methods
Abstract Methods and Classes

● Abstract methods
  ● Like method declarations in an interface
    ● Prototype info (return type, name, arguments)
    ● No body
  ● Derived classes can provide implementation
  ● Notation
    public abstract void drawMe();

● Abstract classes
  ● Abstract class: class with at least one abstract method
  ● abstract modifier must be used when declaring abstract classes
    public abstract Shape { ... }
  ● Abstract classes are incomplete, because not all methods implemented
    ● new cannot be used on abstract classes
      Shape s = new Shape( ... ); // Illegal! Shape is abstract
    ● Abstract classes can be used as types, however
      Shape s = new Circle( ... );

● Classes that are not abstract are called concrete
public abstract class Shape {  // Abstract class
    private int color;    // Color of shape
    ...
    public abstract void drawMe ();  // Abstract method
}

public class Circle extends Shape {  // Concrete class
    private double radius;  // Radius of circle
    private Point center;  // Center of circle
    ...
    public void drawMe () {  ...  }  // Draws a circle
}

public class Rectangle extends Shape {  // Concrete class
    private Point upperLeft;  // Upper left corner
    private Point lowerRight;  // Lower right corner
    ...
    public void drawMe () {  ...  }  // Draws a rectangle
}
Recap Redux

- Inheritance occurs when one class (derived class, subclass) is defined from another class (base / parent class, superclass).
- To derive a class D from a base class B, use:
  ```java
  public class D extends B { … }
  ```
- Derived class inherits all instance variables, methods from base class. It can also define new instance variables, methods
- In derived-class constructor, `super( … )` can be used to invoke constructor from base class
- Derived class can explicitly refer to entities from base class using `super`, e.g. `super.toString( )`
- Polymorphism: object in derived class can be used anywhere base class is expected (a Student “is a” Person!)
- Derived class can override base-class methods (and variables)
  - `final` can be used to disallow overriding
  - Java uses late binding to determine which version of method to use
- `protected` modifier exposes declarations to subclasses (and package)
- All objects are derived (directly or indirectly) from Object
- Late binding and inheritance allows you to create polymorphic variables
- When a method in a base class is not provided, the method and class are said to be abstract. Abstract methods may be implemented in (concrete) derived classes
Class vs. Type Information

- In Java
  - Every object is in one class (the one it was created from using `new`)
  - Objects may have many types
    - Interfaces
    - Superclasses
- E.g. consider
  
  ```java
  Student bob = new Student();
  Person p = bob;
  
  Class of object pointed to by `bob, p is Student`
  
  Type of object can be `Student, Person, Object, etc.`
Accessing Class / Type Information

- Objects can access their class info at run-time
- **getClass()**
  - Method defined in `Object`
  - Returns representation of object’s class
  - E.g.
    ```java
    Person bob = new Person( ... );
    Person ted = new Student( ... );
    
    if ( bob.getClass() == ted.getClass() )
    // false (ted is really a Student)
    ```
- **instanceof**
  - Java boolean operator (not a method)
  - Returns true if given object “is-a”(n) object of given (class) type
  - E.g.
    ```java
    Student carol = new Student ( ... );
    if (carol instanceof Person) // true, because carol “is-a” Person
    ```
Object Casting

- Recall casting in primitive types
  - Casting: conversion of elements from one type to another
  - Upcasting
    - Every element in source type “is-a” element in destination type
    - Can be done automatically
      
        ```java
double x = 3;  // 3 (int) upcast to double
```
  - Downcasting
    - Elements in source type are not necessarily elements in destination type
    - Must use explicit type conversions to perform this casting
      
        ```java
int x = (int)3.0; // 3.0 explicitly cast to int
```
- Similar notions can be found with object types also
  - Upcasting
    - Casting a reference to a superclass (casting up the inheritance tree)
    - Always done automatically and is always safe
  - Downcasting
    - Casting a reference to a derived class
    - Requires explicit casting operator, which checks type info at run-time
    - May not be legal (depending on the actual object type)
Casting Example

Person bob = new Person( ... );
Student ted = new Student( ... );
Student carol = new Student( ... );
GradStudent alice = new GradStudent( ... );

bob = ted; // Upcasting. OK

carol = bob; // Implicit downcasting.
// Compile error!

carol = (Student) bob; // Explicit downcasting.
// OK: bob holds Student object

alice = (GradStudent) ted; // Explicit downcasting.
// Run-time error; ted not a GradStudent object