

Class 13: Inheritance and Interfaces

Introduction to Computation and Problem Solving

**Prof. Steven R. Lerman
and
Dr. V. Judson Harward**

More on Abstract Classes

- **Classes can be very general at the top of a class hierarchy.**
 - For example, MIT could have a class **Person**, from which **Employees, Students, Visitors, etc.** inherit
 - **Person** is too abstract a class for MIT to ever create an instance of it in an application, but it can hold **name, address, status, etc.** that is in common to all the subclasses
 - We can make **Person** an abstract class: **Person** objects cannot be created, but subclass objects, such as **Student**, can be
- **Classes can be concrete or abstract**

Abstract Classes, p.2

- Another example (leading to graphics in the next lectures)
 - Shape class in a graphics system
 - Shapes are too general to draw; we only know how to draw specific shapes like circles or rectangles
 - Shape abstract class can define a common set of methods that all shapes must implement, so the graphics system can count on certain things being available in every concrete class
 - Shape abstract class can implement some methods that every subclass must use, for consistency: e.g., object ID, object type

3

Shape class

```
abstract class Shape {
    public abstract void draw();
    // Drawing function must be implemented in each
    // derived class but no default is possible: abstract

    public void error(String message);
    // Error function must be implemented in each derived
    // class and a default is available: non-abstract method

    public final int objectID();
    // Object ID function: each derived class must have one
    // and must use this implementation: final method

    ...};

class Square extends Shape {...};
class Circle extends Shape {...};
```

4

Abstract method

- **Shape** is an abstract class (keyword)
- **Shape** has an abstract method **draw()**
 - No objects of type **Shape** can be created
 - **draw()** must be redeclared by any concrete (non-abstract) class that inherits it
 - There is no definition of **draw()** in **Shape**
- This says that all **Shapes** must be drawable, but the **Shape** class has no idea of how to draw specific shapes

5

Non-abstract method

- **Shape** has a non-abstract method **error()**
 - Every derived class must have an error method
 - Each derived class may handle errors as it wishes:
 - It may define its own error function using this interface (method arguments/return value)
 - It may use the super class implementation as a default
 - This can be dangerous: if new derived classes are added and programmers fail to redefine non-abstract methods, the default will be invoked but may do the wrong thing

6

Final method

- Shape has a final method **objectID**
 - Final method is invariant across derived classes
 - Behavior is not supposed to change, no matter how specialized the derived class becomes
- Super classes should have a mix of methods
 - Don't make all abstract super class methods abstract. Take a stand!

7

Preventing Inheritance

- To prevent someone from inheriting from your class, declare it final:

```
final class Grad extends Student { ...
```

 - This would not allow `SpecGrad` to be built
 - (Class can have abstract, final or no keyword)
- You can also prevent individual methods from being overridden (redefined) in subclasses by declaring them final

```
final void getData() { ...
```

 - This would not allow a subclass to redefine `getData()`

8

Interface Definition

Interface is a specification for a set of methods a class must implement

- Interfaces specify but do not implement methods
- A class that implements the interface must implement all its methods (or be abstract)
- You may then invoke methods on this class that rely on the interface. Examples:
 - If your class implements the `Comparable` interface, you can put objects of your class into arrays and use the `Arrays.sort()` method
 - You will use interfaces frequently in Swing (GUI)

9

Interface Details

Interfaces are like abstract classes but:

- A subclass can only inherit from only one superclass
- Multiple interfaces can be implemented in a single class (e.g., `Comparable` and `Cloneable`) in your class
- Instances of interfaces cannot be instantiated

```
Comparable list= new Comparable();           // Error
```

- You can declare objects to be of type interface

```
Comparable name;                             // OK
```

- They can be names for objects of a class that implements the interface:

```
Comparable name= new Name();                 // OK
```

- Interfaces may contain methods and constants

```
public interface Rotatable {  
    void rotate(double theta);           // List reqd methods  
    double MAX_ROTATE= 360; }           // Implicitly final  
// Methods and fields default to be public
```

10

Interfaces and inheritance

- **Interfaces can be inherited**
 - **Scholarship program eligibility**

```
public interface Eligible {
    boolean IsEligible(double age, double income); }
```
 - **Scholarship program actual selection**

```
public interface Selected extends Eligible {
    boolean IsSelected(double gpa, double terms); }
```
 - **Our student program could have a scholarship selection class that operates on objects of Student classes that implement one or both of these interfaces**
 - **Undergrad, Grad, SpecGrad** classes from Lecture 12 would all need to implement these methods

11

Interface example

```
import java.util.*;

public class Interfacel {
    public static void main(String[] args) {
        Student[] list= new Student[4];
        list[0]= new Student("Mary", 6);
        list[1]= new Student("Joan", 4);
        list[2]= new Student("Anna", 8);
        list[3]= new Student("John", 2);
        Arrays.sort(list);
        for (int i= 0; i < list.length; i++) {
            Student s= list[i];
            System.out.println(s.getName() + " "
                + s.getTerms());
        }
    }
} // Look up Comparable interface in Javadoc
```

12

Interface example, p.2

```
class Student implements Comparable { // Req'd by Arrays.sort
    public Student(String n, int t) {
        name= n; terms= t;    }
    public String getName() { return name; }
    public int getTerms() { return terms; }
    public int compareTo(Object b) { // Req'd by Comparable
        Student two= (Student) b;
        if (terms < two.terms)
            return -1;
        else if (terms > two.terms)
            return 1;
        else
            return 0;    }
    private String name;
    private int terms;
}
```

13

Inheritance- Key Points

- **Inheritance allows a programmer to extend objects that the programmer did not write**
 - Access restrictions still hold for the super class
 - If the base class changes private data or members, the subclasses should be unaffected
 - Protected members in superclass allow direct access by subclasses
 - Subclass has all data (private, protected and public) of the super class. Each object has all this data.
 - Sub class can use only public and protected methods and data of the super class, not private methods or data
 - **All Java objects inherit implicitly from class Object**
 - Java libraries, Java documentation use Object frequently

14

Fun with animals

```
class Bird {  
    public void fly();    // Birds can fly  
    ... };
```

15

Fun with animals

```
class Bird {  
    public void fly();    // Birds can fly  
    ... };
```

```
class Penguin extends Bird {    // Penguins are birds  
    ... };
```

16

Fun with animals

```
class Bird {
    public void fly();      // Birds can fly
    ... };

class Penguin extends Bird {      // Penguins are birds
    ... };

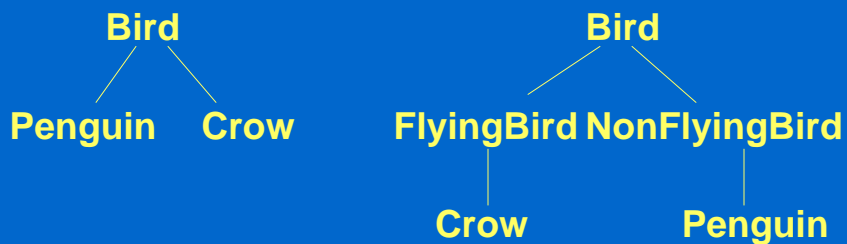
// Problems:
// If super class fly() is final, Penguins must fly

// If super class fly() is abstract or non-abstract,
// Penguin's fly() can print an error, etc. It's clumsy

// With inheritance, every subclass has every method and
// data field in the superclass. You can never drop
// anything. This is a design challenge in real system.
```

17

Possible solutions

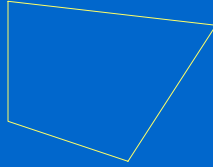


Decision depends on use of system:
If you're studying beaks, difference between flying and not flying may not matter

18

More issues

Quadrilateral



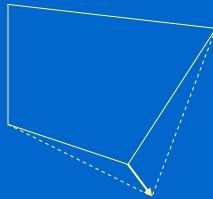
Rectangle



19

More issues

Quadrilateral



`MoveCorner()`

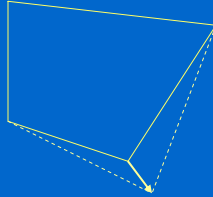
Rectangle



20

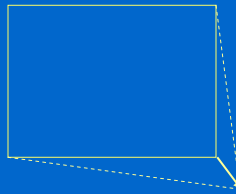
More issues

Quadrilateral



MoveCorner()

Rectangle

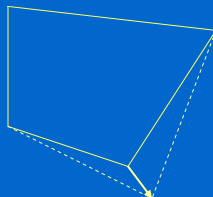


MoveCorner()

21

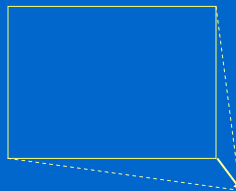
More issues

Quadrilateral



MoveCorner()

Rectangle



MoveCorner()

Must override the MoveCorner() method in subclasses to move multiple corners to preserve the correct shape

22

Lab Exercise(1)

A. Download `Electronics.zip` :

- In your web browser go to the Course web site and download the zip file under the heading Lecture13, JavaFiles into a new folder.
- Unzip `Electronics.zip`. You should have the following files:
 - `Electronic.java`
 - `CellPhone.java`
 - `Computer.java`
 - `Laptop.java`

23

Lab Exercise(2)

- B. Create a new project in Eclipse called `Electronics` and add all the unzipped files to your project.
- C. Figure 1 is the class diagram of the Electronics project. It shows the relationship between the classes, and it shows all their data fields.
 - `Electronic`, the parent class of `Computer` and `Cellphone`, is an abstract class.
 - `Computer` is the parent class of `Laptop`, but is not abstract. This means that you can instantiate an object of type `Computer` (which will represent in our case a desktop computer).

Take a look at the class diagram and make sure it is consistent with the java classes you have downloaded.

24

Class Diagram

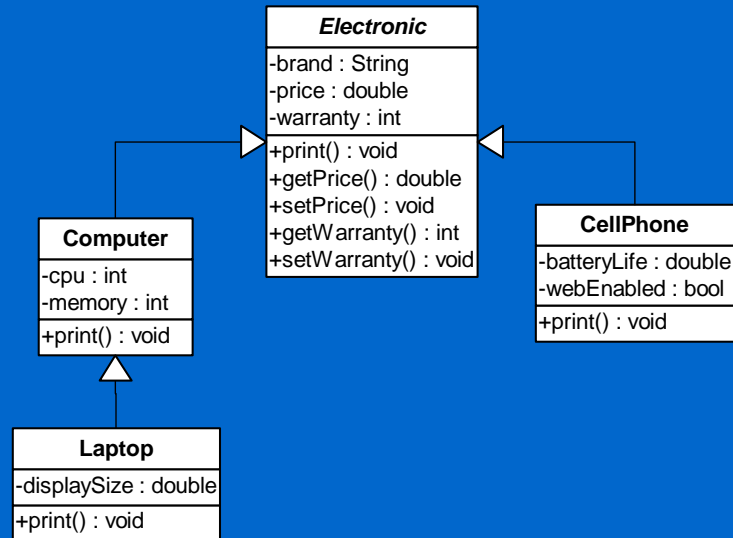


Figure 1

25

Creating an Array

- A. In the `Electronics` project, create a class called `ElectronicMain` with an empty `main()` method.
- B. In the `main()` method, create an array that can hold three objects of type `Electronic`.
- C. Add to this array three objects with the following specifications:
 - Cell phone: Brand: Nokia TS200; Price: \$300; Warranty: 18 months; Battery Life: 3.5Hr; Web Enabled: true;
 - Desktop (Computer): Brand: Dell D2100; Price: 1000; Warranty: 24 months; CPU speed: 2500 MHz; Memory: 512 MB;
 - Laptop: Brand: HP N5170; Price: 1500; Warranty: 24 months; CPU speed: 1500 MHz; Memory: 256 MB; Display Size: 15"
- D. Print out the specifications of all the elements contained in the array.

26

Adding Functionality

- A. We need a method to increase the price of an electronic device by a given percentage. The method should have the following signature:

```
public void increasePrice(int percentage)
```

- This method doesn't behave the same for all electronic devices:
 - When a computer's price is increased by any percentage, its warranty is increased by 12 months.
 - When a cell phone's price is increased by any percentage, its warranty is increased by 6 months.

27

Adding functionality (2)

- B. Add this functionality to the classes:

- Note: `Electronic` class should declare the `increasePrice(int percentage)` method as abstract. The function must therefore be implemented in each of the derived classes.

- C. In the `main()` method, increase the price of all the elements in the array by 5%. Print the all the elements again. The output should be the following:

```
- Brand: Nokia TS200; Price: $315.0; Warranty: 24 months  
  Battery Life: 3.5Hr; Web Enabled: true  
- Brand: Dell D2100; Price: $1050.0; Warranty: 36 months  
  CPU Speed: 1500MHz; Memory Size: 512MB  
- Brand: HP N5170; Price: $1575.0; Warranty: 36 months  
  CPU Speed: 900MHz; Memory Size: 256MB  
  Display Size: 15.0"
```

28

Counting Electronic devices (Optional)

A. We need to keep track of the number of electronic devices we have created. You should add the following data fields in the appropriate classes:

- numberElectronics
- numberCellPhones
- numberComputers
- numberLaptops

These data fields should be initialized to 0.

29

Counting Electronic devices (Optional)

B. Every time a new object is created, the appropriate counters should be incremented by 1. Taking our array example you should have:

- numberElectronics = 3
- numberCellPhones = 1
- numberComputers = 2 (Remember that a Laptop is a Computer)
- numberLaptops = 1

C. In each of the classes, create a method `getCount()` that returns the class counter. You should be able to call `getCount()` without instantiating any object.

D. In the `main()` method, call the `getCount()` function of all the classes (Electronic, CellPhone, Computer and Laptop). Compile and run.

30