CSCI 136
Data Structures & Advanced Programming

Jeannie Albrecht

Lecture 3

Feb 12, 2014

Administrative Details

- Lab 1 design doc "due" at beginning of lab
  - Several implementation options
  - I recommend making an array of positions rather than trying to represent the board with the array
  - coins[0] = 1 means first coin is in space 1
- Lab today in TCL 217a (216 is available, too)
  - Lab is due next Monday at noon
    - Submit via turnin (details are in the handout)
- If you want to configure your laptop (PC or Mac) to work on labs, bring laptop to lab or come see me

2

Last Time

- Reviewed command line arguments and use of Scanner for reading input from stdin (System.in)
- Object oriented programming
  - Objects model physical items, concepts, processing
  - Objects have properties and capabilities

3

Today's Outline

- Continue Java refresher
- Discuss interfaces, classes, and inheritance
- Learn about toString() and equals()

4

Implementing Cards

- Think before we code!
- Start general.
  - Build an *interface* that advertises all public features of a card
  - Not an implementation (define methods, **but don't include code**)
- Then get specific.
  - Build specific implementation of a card using our general card interface

5

(Random) Notes about Interfaces

- Interface methods **are always** public
  - Java does not allow non-public methods in interfaces
- Interface instance variables are always **static final**
  - static variables are shared across instances
  - final variables never change
- Most classes contain constructors; interfaces do not!
- Can create interface objects (just like class objects)

6

Start General: CardInterface

- What data do we have to represent?
  - Properties of cards
  - How can we represent these properties?
- What methods do we need?
  - Capabilities of cards
  - Do we need *accessor* and *mutator* methods?

Get Specific: Card

- Now suppose we want to build a specific card object
- We want to use the properties/capabilities defined in our interface
  - That is, we want to *implement* the interface

```
public class Card implements CardInterface {
    . . .
}
```

- Note: Classes do not *need* main methods (although they often contain them)
  - Main method just tells the JRE where to "start"
  - See CardMain.java

PokerHand

- Now that we have implemented CardInterface and Card, how would we implement PokerHand?
- What data structures do we need?
- We need a way to store 5 cards…
  - Can use an array of Card objects!

Array Review

- Syntax for 1-D array:

```
int hand[ ] = new int[5];
```

- Syntax for 2-D array:

```
int hand[ ][ ] = new int[10][15];
```

- Determine size of array?

```
hand.length;  //not .length()!!
```


Class Specialization

- We now know that classes can *implement* one or more interfaces
- Classes can also *extend* other classes
  - Inherit fields and **method bodies**
  - Note: *implements* does not do this!!!!!!!
- Interfaces can extend other interfaces
- By extending other classes/interfaces, we can create specialized classes

Specialization Example

```
class Fish {
  public void swim() { … }
  public void eat() { … }
}

class Shark extends Fish {
  //can use swim() in Fish without implementing it
  //or can optionally override using a specialized swim()
  public void swim() { … }

}
```

- What does the following code do?

```
Fish fish = new Shark();
fish.swim();
```

- What are the benefits of specialization?

Specialization Example

```
class Fish {
  public void swim() { … }
  public void eat() { … }
}

class Shark extends Fish {
  //can use swim() in Fish without implementing it
  //or can optionally override using a specialized swim()
  public void swim() { … }

}
```

- What does the following code do?

```
Fish fish = new Shark();
fish.swim();
```
This calls Shark.swim()

- What are the benefits of specialization?
  - Code reuse and extensibility

13

Specialization Example

```
class Fish {
  public void swim() { … }
  public void eat() { … }
}

class Shark extends Fish {
  //can use swim() in Fish without implementing it
  //or can optionally override using a specialized swim()
  public void swim() { … }
  public void attack() { … }
}
```

- What does the following code do?

```
Fish fish = new Shark();
fish.swim();
fish.eat();
```

- Does this work?

```
fish.attack();
```

14

Specialization Example

```
class Fish {
  public void swim() { … }
  public void eat() { … }
}

class Shark extends Fish {
  //can use swim() in Fish without implementing it
  //or can optionally override using a specialized swim()
  public void swim() { … }
  public void attack() { … }
}
```

- What does the following code do?

```
Fish fish = new Shark();
fish.swim();
fish.eat();
```
Calls Fish.eat()

- Does this work?

```
fish.attack();
```
No, because attack() is not defined in Fish.

15

(See additional examples in FishMain.java)