CSE 142

Interfaces

Overview

- Defining new types of things – interfaces
- Goal: Learn what it means to implement a specification given in an interface
  - Details of defining new interfaces covered in CSE143
- Reading
  - Niño & Hosch, sec. 15.1.2
    (somewhat technical and beyond what we need, but useful reference)

Goal: Animations

- We'd like to be able to build *animations*: graphical scenes that move on their own.
- What tools do we need to construct animations?
  - Graphical shapes and windows: GWindow, Rectangle, House, etc.
  - Lists of active shapes: ArrayList
  - A repeating loop to tell each active shape in the list to do its action: while, Iterator
- Let's try it!

The Core

- A class that runs the animation:

```
public class Stage { ...
```

- An ArrayList of active objects:

```
private ArrayList actors;
```

- A loop, iterating over each element in the list, telling each actor to perform one step of its action:

```
public void animateOneRound( ) {
    Iterator iter = actors.iterator( );
    while (iter.hasNext( )) {
        Object actor = iter.next( );
        actor.doAction( );    // tell this actor to perform one step
    }
}
```

- But this doesn't work! (Why not?)

The Problem

- We want to be able to send a message doAction to each actor.
- But Object doesn't understand doAction.
- We need to cast the actor to the right type.
- But what is that type?
  - Different animations will have different kinds of actors, e.g., Asteroid vs. JigglingHouse.
  - Some animations can have more than one kind of actor, e.g., JigglingHouse and RisingSun.
- How can we accommodate lots of different kinds of actors? Without rewriting Stage for each animation?

The Solution: Interfaces

- Can define an *interface* for actors, and then declare that each different kind of actor *implements* this interface.
- An interface is like a class, but no instance variables, no constructors, and no method bodies.

```
public interface Actor {
    /** Every Actor can perform some action. */
    public void doAction(Stage stage);
    /** Every Actor can add itself to and remove itself from a GWindow. */
    public void addTo(GWindow w);
    public void removeFromWindow( );
}
```

- Interfaces are regular types: we can declare variables of an interface type, and we can cast to an interface type.
  - We cannot instantiate an interface type, though.

The Core, Revisited

- Now we can rewrite the core using the Actor interface:

```
public void animateOneRound( ) {
    Iterator iter = actors.iterator( );
    while (iter.hasNext( )) {
        Actor actor = (Actor) iter.next( );
        actor.doAction( );   // tell this actor to perform one step
    }
}
```

- We can write other Stage methods using actors.

```
public void addActor(Actor actor) {
    this.actors.add(actor);          // add the actor to the actors list
    actor.addTo(this.window);    // tell the actor to display itself on the screen
}
```


Implementing Interfaces

- To create instances of an interface, the interface must be *implemented* by one or more classes.
- Each class implementing an interface must include methods with bodies for all the methods declared in the interface. The class can have additional methods, too.

```
public class Asteroid implements Actor {
    ... // instance variables, constructors, etc.
    public void doAction(Stage stage) {
        ... /* the code to move according to gravity */  }
    public void addTo(GWindow w) { ... /* the code */ }
    public void removeFromWindow( ) { ... /* the code */ }
    ... // more methods
}
```


Objects of Interface Type

- Because Asteroid implements Actor, we can create instances of the Asteroid class, and then use them wherever an object of type Actor is required
  - An Asteroid *is*-an Actor.

```
Stage asteroidField = new Stage(...);
Asteroid aNewAsteroid = new Asteroid(...);

// add the new asteroid to the stage:
asteroidField.addActor(aNewAsteroid);
```

(recall declaration of addActor: void addActor(**Actor** actor) {...})

Another Implementation of Actor

```
public class JigglingHouse implements Actor {
    private House house;  // the House shape
    private Random random;  // a random number generator
    public JigglingHouse(House h) {
        this.house = h;  this.random = new Random( );  }
    public void addTo(GWindow w) {  this.house.addTo(w);  }
    public void removeFromWindow( ) {  this.house.removeFromWindow( );  }
    public void doAction(Stage stage) {
        int deltaX = this.random.nextInt(21) - 10;  // a number between -10..+10
        int deltaY = this.random.nextInt(21) - 10;
        this.house.moveBy(deltaX, deltaY);
    }
}
```


Another Interface: Shape

- To let GWindow display lots of different kinds of shapes, we built an interface Shape.

```
public interface Shape {
    ...
    // every shape must know how to paint itself onto a Java canvas object
    public void paint(java.awt.Graphics canvas);
}
```

- A GWindow keeps an ArrayList of Shape objects, and sends each one the paint message when the GWindow needs to be (re)drawn.
  - Done behind the scenes – you don't have to deal with this
- Rectangle, Oval, Line, etc. all implement the Shape interface.

Another Interface: Iterator

- Iterator is actually an interface, not a class.
- ArrayList provides a class that implements the Iterator interface, but that knows about the insides of ArrayList.
- Other collections in Java provide their own classes that implement the Iterator interface, but that know about how they're implemented.
- The users of collections don't have to know anything about how they're implemented! They only have to know how to get an iterator and then how to send hasNext( ) and next( ) messages to the iterator.