

CSC 143 Java

Object and Class Relationships: Interfaces

Reading: Ch. 9 (on Java interfaces)

(c) University of Washington

02-1

Relationships Between Real Things

- Man walks dog
- Dog strains at leash
- Dog wears collar
- Man wears hat
- Girl feeds dog
- Girl watches dog
- Dog eats food
- Man holds briefcase
- Dog bites man



(c) University of Washington

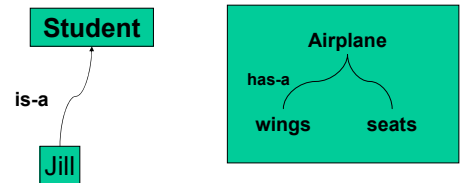
02-2

Common Relationship Patterns

- A few types of relationships occur extremely often
 - **IS-A**: Jill is a student (and an employee and a sister and a skier and
 - **HAS-A**: An airplane has seats (and lights and wings and engines and...
- These are so important and common that programming languages have special features to model them
 - Some of these you know (maybe without knowing you know)
 - Some of them we'll learn about in this course, starting now, with **inheritance**.

(c) University of Washington

02-3



(c) University of Washington

02-4

Reminder: State vs Behavior



- State
 - has blue hair
 - wearing glasses
 - wearing blue shoes
 - is hopping mad
- Behavior
 - clenches fist
 - raises arm
 - hops up and down
 - screams

(c) University of Washington

02-5

Inheritance and Interfaces

- Inheritance is the way that many OO languages model the **IS-A** relationship
 - Interfaces (in Java) is one special form of inheritance
- Inheritance is one of the last missing pieces in our knowledge of Java fundamentals
- A Java **Interface** declares a set of method signatures
 - i.e., says what behavior exists
 - Does not say how the behavior is implemented i.e., does not give code for the methods
 - Does not describe any state

(c) University of Washington

02-6



interface I

method signatures of
I, without code; no
instance variables

concrete
class C

methods of I,
including code

other methods,
instance variables
of C

(c) University of Washington

02-7

A Domain to Model: Geometric Shapes

- Say we want to write programs that manipulate geometric shapes and produce graphical output
- This application domain (the world to model) has:
 - Shapes:
 - Rectangles, Squares
 - Ovals, Circles, Arcs
 - Polygons, Lines, Triangles
 - Images
 - Text
 - Windows
- Let's build a computer model!

(c) University of Washington

02-8

Typical Low-Level Design Process (1)

- Step 1: think up a class for each kind of "thing" to model
 - GWindow
 - Rectangle (no Square)
 - Oval (no Circle), Arc
 - Polygon, Line, Triangle
 - ImageShape
 - TextShape
- Step 2: identify the state/properties of each thing
 - Each shape has an x/y position & width & height
 - Most shapes have a color
 - Most shapes have a filled/unfilled flag
 - Each kind of shape has its own particular properties

(c) University of Washington

02-9

Process (2)

- Step 3: identify the actions (behaviors) that each kind of thing can do
 - Each shape can add itself to a window:
 - s.addTo(w)
 - Each shape can remove itself from its window:
 - s.removeFromWindow()
 - Each shape can move
 - s.moveTo(x, y)
 - s.moveBy(deltaX, deltaY)
 - Most shapes can have its color changed, or its size changed, or ...
 - s.setColor(c)
 - s.resize(newWidth, newHeight)
 - ...

(c) University of Washington

02-10

Key Observation

Many kinds of shapes share common properties and actions

- How can we take advantage of this?
- It would be nice not to have to define things over and over.
- Yet there are differences between the shapes, too.

(c) University of Washington

02-11

A Solution: Interfaces

- Declare common *behaviors* in a Java **interface**

```
public interface Shape {  
    public int getX();  
    public void addTo(GWindow w);  
    ...  
}
```

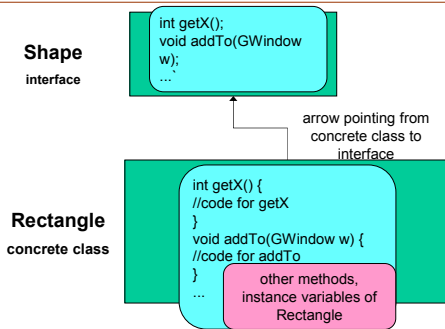
- Create a concrete class for each type of thing that implements this interface
- Annotate the class definition with "implements shape"

```
public class Rectangle implements Shape {  
    public int getX() { ... }  
    ...  
}
```

(c) University of Washington

02-12

Interface vs. Concrete Class



(c) University of Washington

02-13

Implementing Interfaces

- If a class declaration says "implements I..."
 - It **MUST** implement every single method of the interface
 - It cannot change anything about the method interfaces
- A class can implement more than one interface
 - When might this be useful? (Hint: think "modeling")
- A class that implements an interface is completely free to add other methods, instance variables, etc.

(c) University of Washington

02-14

Two Benefits of Interfaces

- The benefits are real, but may be hard to see until you've used the concept in several programs
1. Better model of application domain
Humans talk about "shape"s as a general group; the computer model should, too
 2. Can write code that works on any concrete object that implements the interface (e.g., on any Shape)
Each interface introduces a new **type**
Can declare variables, arguments, results, etc. of that type

(c) University of Washington

02-15

Using Interfaces as Types

- Each interface introduces a new **type**
- An object of a concrete class that implements an interface has **two types, effectively**
 - The concrete type (=the type of the class)
 - The interface type (because of the IS_A relationship between the class and the interface)
- Such an object can be used in any situation where one or the other type is appropriate
 - As variables (Shape s = new Rectangle();)
 - As arguments and parameters (e.g. in GWindow, public void add(Shape s))
 - As return types, e.g.
public Shape getShape() { return new Rectangle(); }

(c) University of Washington

02-16

Some Domains for Examples

- Another set of domains to model: animations & simulations
- Example domains, and the things in those domains:
 - Financial simulation: bank accounts, customers, investors
 - Planetary simulation: suns, planets, moons, spaceships, asteroids
 - Fantasy game: characters, monsters, weapons, walls
- Can have a visual representation of the simulation, using graphical shapes & windows
- Let's build some computer models!

(c) University of Washington

02-17

An Example: A Planetary Simulation



- Model the motion of celestial bodies

- Requirements: left a bit vague for this example
- Step 1: make classes for each kind of thing
- Step 2: identify the state/properties of each thing
- Step 3: identify the actions that each kind of thing can do
- Step 4: if there are classes with many common behaviors, considering making an interface out of the common part

(c) University of Washington

02-18

An Example: A Planetary Simulation

- Step 1: make classes for each kind of thing
 - Sun, Planet, Spaceship
 - Universe containing it all
- Step 2: identify the state/properties of each thing
 - Location, speed, mass
 - List of things in the universe
- Step 3: identify the actions that each kind of thing can do
 - Compute force exerted by other things;
update position & velocity based on forces;
display itself on a window
 - Tell each thing in universe to update itself based on all other things;
react to keyboard & mouse inputs

Frameworks

- When a recurring pattern of classes is identified, it can be extracted into a **framework**
 - Often use interfaces in place of particular classes (e.g. CelestialBody)
- Clients then build their models by extending the framework
 - Making instances of framework classes (e.g. Universe)
 - Making application-specific classes that implement framework interfaces (e.g. CelestialBody)
 - Making new application-specific classes
- **Libraries** are simple kinds of frameworks
 - Don't have interfaces for clients to implement