CSE 143

Interfaces

Reading: Ch. 15.1.3

A Problem – Object Model for a Simulation

- Suppose we are designing the classes for a simulation game like the Sims, or Sim City
- We might want to model
  - People (office workers, police/firemen, politicians, ...)
  - Pets (cats, dogs, ferrets, lizards, ...)
  - Vehicles (cars, trucks, buses, ...)
  - Physical objects (buildings, streets, traffic lights, ...)
- Object model – use inheritance
  - Base classes for People, Pets, Vehicles, PhysicalThings, ...
  - Extended classes for specific kinds of things (Cat extends Pet, Dog extends Pet, Truck extends Vehicle...)

Making it Tick

- A time-based simulation has some sort of clock that ticks regularly
- On each tick, every object in the simulation needs to, for instance, update its state, maybe redraw itself, ...
- We would like to write methods in the simulation engine that can work with any object in the simulation

```
/** update the state of simulation object thing for one clock tick */
public void updateState(???  thing) {
    thing.tick( );
    thing.redraw( );
}
```

- Question: What is the type of parameter *thing* in this method?

Type Compatibility

- We want to be able to write something like

```
public void updateState(SimThing thing) { ... }
```

  where "SimThing" is a type that is compatible with Cats, Cars, People, Buildings. How?
- Could create an additional superclass SimThing and have People, Pets, Vehicles, PhysicalThings, ..., all extend it, but:
  - People, Pets, etc. don't have a real "is-a" relationship
  - What if we wanted to have other polymorphic methods that, for example, only apply to breathing things?
  - Deep inheritance hierarchies are brittle, hard to modify

Solution – Interfaces

- We want a way to create a type SimThing independently of the simulation actor class hierarchies, then tag each of those classes so they can be treated as SimThings
- Solution: create a Java ***interface*** to define type SimThing
- Declare that the appropriate classes ***implement*** this interface

SimThing Interface

- Interface declaration

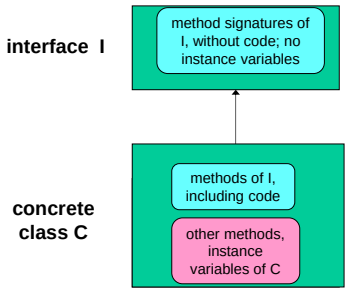
```
/** Interface for all objects involved in the simulation */
public interface SimThing {
    public void tick( );
    public void redraw( );
}
```

- Class declaration using the interface

```
/** Base class for all Pets in the simulation */
public class Pet implements SimThing {
    /** tick method for Pets */
    public void tick( ) { ... }
    /** redraw method for Pets */
    public void redraw( ) { ... }
    ...
}
```

Interfaces and Implements

- 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 (but may include "final" constants)
- A concrete class that implements an interface
  - Contains "implements *InterfaceName*" in the class declaration
  - Must provide implementations (either directly or inherited from a superclass) of all methods declared in the interface
- An abstract class can also implement an interface
  - Can optionally have implementations of some or all interface methods



Interfaces and Extends

- Both describe an "is-a" relation
- If B *implements* interface A, then B inherits the (abstract) method signatures in A
- If B *extends* class A, then B inherits everything in A, which can include method code and instance variables as well as abstract method signatures
- Sometimes people distinguish "interface inheritance" from "code" or "class inheritance"
  - Specification vs implementation
  - Informally, "inheritance" is sometimes used to talk about the superclass/subclass "extends" relation only

Classes, Interfaces, and Inheritance

- A class
  - Extends exactly one other class (which defaults to Object if "extends …" does not appear in the class definition)
  - Implements zero or more interfaces (no limit)
- Interfaces can also extend other interfaces (superinterfaces)
  Interface ScaryThing extends SimThing { … }
  - Mostly found in larger libraries and systems
  - A concrete class implementing an extended interface must implement all methods in that interface and (transitively) all interfaces that it extends

What is the Type of an Object?

- Every interface or class declaration defines a new ***type***
- An instance of a class named *Example* has all of these types:
  - The named class (*Example*)
  - Every superclass that *Example* extends directly or indirectly (including Object)
  - Every interface (including superinterfaces) that Example implements
- The instance can be used anywhere one of its types is appropriate
  - As variables, as parameters and arguments, as return values

Benefits of Interfaces

- May be hard to see in small systems, but in large ones…
- Better model of application domain
  - Avoids inappropriate uses of inheritance to get polymorphism
- More flexibility in system design
  - Can isolate functionality in separate interfaces – better cohesion, less tendency to create monster "kitchen sink" interfaces or classes
  - Allows multiple abstractions to be mixed and matched as needed

Interfaces vs Abstract Classes

- Both of these specify a type
- Interface
  - Pure specification
  - No method implementation (code), no instance variables, no constructors
- Abstract class
  - Method specification plus, optionally:
    - Partial or full default method implementation
    - Instance variables
    - Constructors (called from subclasses using super)
- Which to use?

Abstract Classes vs. Interfaces

Abstract Class Advantages

- Can include instance variables
- Can include a default (partial or complete) implementation, as a starter for concrete subclasses
- Wider range of modifiers and other details (static, etc.)
- Can include constructors, which subclasses can invoke with *super*
- Interfaces with many method specifications are tedious to implement (implementations can't be inherited)

Interface Advantages

- A class can extend *at most one* superclass (abstract or not)
- By contrast, a class (and an interface) can implement any number of super-interfaces
- Helps keep state and behavior separate
- Provides fewer constraints on algorithms and data structures

A Design Strategy

- These rules of thumb seem to provide a nice balance for designing software that can evolve over time:
  (Might be overkill for CSE 143 projects)
  - Any major type should be defined in an interface
  - If it makes sense, provide a default implementation of the interface with a class – can be abstract or concrete
  - Client code can choose to either extend the default implementation, overriding methods that need to be changed, or implement the interface directly (the later is required if the class explicitly extends a superclass)
- This pattern occurs frequently in the standard Java libraries