CSC 143 Java

Interfaces

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?

Solution – Interfaces

- We want a way to create a type SimThing independent 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**

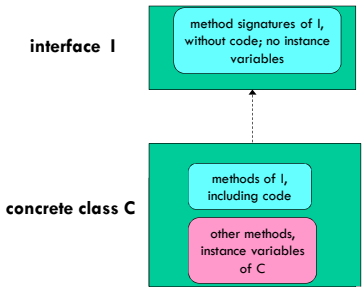
```
/** Interface for all objects involved in the simulation */
public interface SimThing {
    public void tick( );
    public void redraw( );
}
```

- **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



What is the Type of an Object?

- Every interface or class declaration defines a new *type*
- An instance of a class named *Example* conforms to 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 arguments, as return values

Interfaces vis-a-vis Inheritance

- Both describe an "Is-A" relation
- If B *implements* interface A, then B inherits the method signatures from A (must implement them)
- If B *extends* class A, then B inherits everything from A, which can include method code and instance variables
- 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

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 specify 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 can implement any number of 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 some CSC 143 projects)
  - **Any major type should be defined in an interface**
  - **If it makes sense, provide a default implementation of the interface**
  - **Client code can choose to either extend the default implementation, overriding methods that need to be changed, or implement the complete interface directly (needed if the class already has a specified superclass)**
- **This pattern occurs frequently in the standard Java libraries**