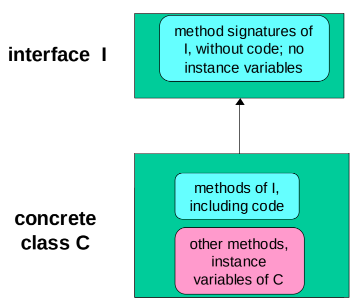
CSE 143 Java

More About Interfaces

Reading: Ch. 15.1.3

Interfaces -- Review

- 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
- PS: An abstract class can also implement an interface
  - Can optionally have implementations of the interface methods



Uses For Interfaces

- We have already seen Java interfaces as a form of software specification
- Boss says "you implement these methods or else!"
- Java language checks that all the methods do in fact get implemented in the concrete classes
- Interfaces have other uses as well

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, monsters...)
  - Pets (cats, dogs, ferrets, lizards, ...)
  - Vehicles (cars, trucks, buses, ...)
  - Physical objects (buildings, traffic lights, carnival rides ...)
- Object model – use inheritance
  - Base classes for People, Pets, Vehicles, PhysicalThings, ...
  - Extended classes for specific kinds of things (Cat extends Pet, Dog extends Pet, ...)

Making it Tick

- "Time-based" simulation work like a movie:
  - On each "frame", the picture of the world is updated a little bit
  - implies 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, ...
- There is a driver or "engine" that drives the simulation
  - Sort of like the movie camera
  - The engine knows of all the objects, but doesn't know how to update them or draw them
  - On each tick, tells every object to update and redraw itself
  - Each object knows how to update itself and how to draw itself

The Engine's Dilemma

- 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( );
}
```

- The same method should work for cars, pets, monsters, ferris wheels, trees, etc.
- Question: What is the type of parameter *thing* in this method?
- Footnote: this is an example of a polymorphic method

Type Compatibility

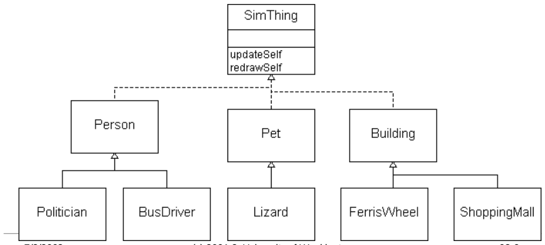
- We want to be able to write something like

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

  where "SimThing" is a type that is compatible with Cats, Cars, People, Buildings.
- The engine only needs to keep track of what objects exist
- The individual objects are responsible for carrying of the actions

Solution

- Take the common behavior and specify it in an interface
- Make all objects in the simulation implement that interface



SimThing Interface

- Interface declaration

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

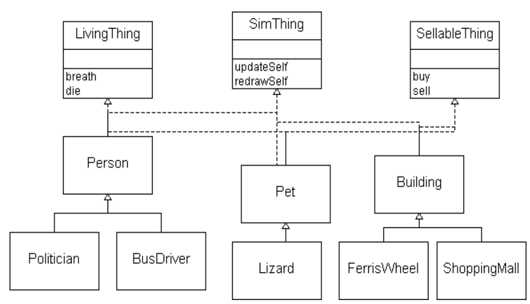
- Class declaration using the interface

```
/** 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( ) { … }
    …
}
```


Why Not...

- Why not make SimThing a class or abstract class?
- Answer: In complex models, things do not always fit into neat hierarchies
  - We might want to specify common behavior for all LivingThings (People, Pets)
  - We might identify behavior for all items which can be bought and sold (Pets, Buildings)
- Class hierarchy won't work
  - A class can extend only one class
  - You can only define one set of common behavior
- Interfaces to the rescue!

SimCity with Three High-Level Interfaces



Implements vs. 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
- Sometimes people distinguish "interface inheritance" from "code" or "class inheritance"
- Informally, "inheritance" is sometimes used to talk about the superclass/subclass "extends" relation only

Classes, Interfaces, and Types

- A class can
  - Extend *exactly* one other class
    implicitly Object if "extends ..." is not included in the class definition
  - Implement *zero or more* interfaces -- no limit!
  - Historical footnote: C++ allows multiple inheritance of classes
- Interfaces can also extend other interfaces (superinterfaces)
  - Mostly found in larger libraries and systems
  - A concrete class implementing an extended interface must implement all methods in all superinterfaces
- Every interface or class declaration creates a new ***type***

What is the Type of an Object?

- An object (instance of a class) can have many types
- An instance of class busDriver has all of these types:
  - The named class (BusDriver)
  - Every superclass that BusDriver extends (Person, Object)
  - Every interface (including superinterfaces) that BusDriver implements (SimThing, LivingThing)
- 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 use 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

Abstract Classes vs. Interfaces

Abstract Class

- 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

Interface

- A class can extend *at most one* superclass (abstract or not), but multiple interfaces
- 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

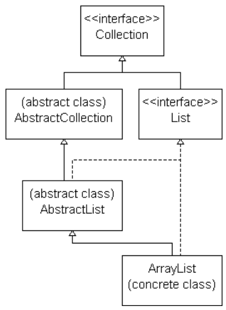
7/2/2003 03-17

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 CSE 143 projects)
  - Any major type should be defined in an interface
  - If it makes sense, provide a default implementation of the interface – 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 complete interface directly (particularly if they already have another superclass)
- This pattern occurs frequently in the standard Java libraries

7/2/2003 03-18

ArrayList: Partial Lineage



7/2/2003 03-19