CSE 143 Java

Abstract Classes and Frameworks

Reading: Ch. 15

What is a Generic Animal?

- Purpose of class Animal (base class for Dog and Cat)
  - provide common specification for all Animals
  - define some instance variables
  - provides implementation for some methods
    getName(), getSpecies(), getNumberOfLegs(), etc.
- A few puzzlers...
  - What noise should a generic Animal make?
    Answer: class Animal doesn't have enough information to know!
  - Are there really any objects of type Animal?
    Really, we have a Dog, or Cat, or Dragonfly, or etc.
- Animal exists to be extended, not used directly to create instances

Abstract Classes

- Main idea: methods may be declared **abstract**, and left unimplemented

```
public abstract myMethod( ) ;
```

- If a class contains an abstract method, it must be declared as an **abstract** class with the ***abstract*** keyword

```
public abstract class MyClass {...}
```

- Compare and contrast:
  - Interface
  - Abstract class
  - Concrete class

Abstract vs Concrete

- Cannot instantiate an abstract class (no `new`)
  - Like an interface
- A class that extends an abstract class can override methods (including abstract methods) as usual
- A class that provides implementations for all abstract methods it inherits is said to be **concrete**
  - If a class inherits an abstract method and doesn't override it, it is still abstract
  - An error message is reported if a non-abstract class doesn't implement all inherited abstract methods

Example: Animals as an Abstract Class

```
public abstract class Animal {                  // abstract class
   private int numLegs;

   public int getNumLegs( ) {
      return this.numLegs;
   }

   public abstract String noise( ) ;
}

public class Cat extends Animal {               // concrete subclass
   public String noise( ) { return "purrr"; }
}
```


Comparing Abstract Classes and Interfaces

- Both of these specify a type
- Interface
  - Pure method 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

Pro Abstract Classes

- 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

Pro Interfaces

- 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

Abstract Classes and Frameworks

- Abstract classes are a key component of good OO programming
  - A good place to **factor out** declarations and code that are common to several classes, even if the common code is incomplete
- Support the development of good frameworks
  - Can write a bunch of useful code in abstract classes
  - Let clients write application-specific concrete subclasses with little effort
- Design strategy:
  - Build a bunch of examples in some domain (e.g. a bunch of games)
  - Create abstract classes to capture repeating patterns

Framework Example

- Example: a framework for Dungeon games

```
abstract class MovingThing implements Actor { ... }
    // keeps track of location, perhaps a list of Shapes as appearance
abstract class Character extends MovingThing { ... }
    // keeps track of score, provides default implementations of motion,
    // being captured, etc.
    // clients implement their own concrete subclasses of Character,
    // providing their own visual appearance and customizing behavior as desired
abstract class Monster extends MovingThing { ... }
    // adds chasing & capturing default behavior
    // clients implement their own concrete subclasses of Monster,
    // providing their own visual appearance and customizing behavior as desired
```


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 (e.g. if they already have another superclass)
- We'll see this pattern frequently when we look at the UWCSE and Java libraries

Question for Next Time: If I Had Designed Java...

- The word *abstract* is vague and misleading at best
- If you designed the successor for Java...
  - What word would you use to mark an abstract **method**?
  - What word would you use to mark an abstract **class**?