Inheritance, Method Overloading, Interfaces, Abstract Classes

Game Design Experience
Professor Jim Whitehead
January 28, 2009
Announcements

• Homework #1 back Friday
  ▶ Was unable to connect with TAs yesterday

• Work breakdown and schedule
  ▶ Due next Monday
  ▶ Team assignment
  ▶ Goals:
    • Develop an understanding of work needed to be performed to complete project, and time required
    • Discuss breakdown of tasks among partners
Classification

- Classification is the act of assigning *things* to *categories of things*.
  - The *is-a* relationship

- Examples:
  - A Volkswagen Jetta *is a* (kind of) car.
  - A hot dog *is a* (kind of) food.

- Classification is at the heart of object-oriented modeling.
  - An object-oriented class represents a category
    - Class car, Class food
  - Object instances (in OO) are instances of categories
Developing classes

• The process of taking a set of real world objects and developing its associated category is called abstraction.

• Example:
  ▶ If I give you the set of food items:
    • Cheese whizz, hot dog, corn chips, bran flakes, chicken
  ▶ The act by which you create the category *food* and call these all examples of food, is *abstraction*.

• The process of abstraction is used to determine what classes should be in your software
Abstraction and Classification is tricky

- Consider the following items. Which ones do you consider to be furniture?
  - Sofa
  - Recliner
  - Television
  - Stove
  - Counter-top microwave oven
  - Fixed line telephone

- Qualities of categories
  - Fuzzy boundaries
  - Some category members are better examples of the category than others
  - Categories often best understood in contrast to other categories
    - Dividing line between furniture and kitchen appliances
  - *Women, Fire, Dangerous Things*, George Lakoff
    - Book on categorization
Hierarchies of Categories

• Sometimes there are situations where you have multiple levels of categories
  ► Category1 is a (kind of) Category2 is a (kind of) Category3
  ► Example
    • A cell phone is a (kind of) telephone
    • A wired phone is a (kind of) telephone
    • An iPhone is a (kind of) cell phone
    • Telephone is an abstract category

• In software, may want to represent things like *telephone*, but never have direct instances
  ► Only want instances of sub-categories
  ► Example: want to represent telephones, but do not want instances of telephone. Only want instances of cell phone, or wired phone.
  ► In software, we would like telephone to provide details about all telephones…
    • All phones have a telephone number, can make and receive calls
  ► …but not to give cell phone or wired-phone specific information
    • Cell phones have text messaging, wired phones do not
Representing Abstract Categories

• Two ways to represent abstract categories
  ► Interface
    • Class properties
    • Method names and parameter lists, **but no method implementations of any methods.**
  ► Abstract Class
    • Class variables
    • Can have a mixture of:
      – Methods that are fully implemented
        » Might (or might not) be overridden in subclasses
      – Methods that only have a name and a parameter list
        » Abstract method
        » Must be implemented in subclasses
Interface

• Describes a set of methods, properties that must be implemented by all subclasses
  ▶ But, no implementation details are given
  ▶ Subclasses can implement the methods and properties as they see fit
    • So long as they match the types of the properties, method return values, and method parameters

• Describes the external boundary of class
  ▶ What other classes need to know to use its features
  ▶ Provides no implementation detail
Defining an interface

[attributes] [access-modifiers] interface identifier [:base-interface(s)]
{   interface-body }

Simple example:

interface ITelephone
{
    public string PhoneNum {get; set;}  // Phone number property, can read and write

    void display_phone_num();          // Output phone number
}

• Naming convention:
   ► Put capital “I” in front of interface name
Using an Interface

interface ITelephone
{
    public string PhoneNum {get; set;}   // Phone number property, can read and write

    void display_phone_num();             // Output phone number
}

public cellphone : ITelephone
{
    public string PhoneNum {get; set;}  // Use automatic property to implement interface property

    public void display_phone_num()
    {
        System.Console.WriteLine("cell phone number is {0}", PhoneNum);
    }
}

• Syntactically, looks like inheritance
• Must implement all aspects of interface
Abstract Classes

• Sometimes you want to:
  ► Define the information passing interface for a class
    • Can do this with an interface
  ► And provide implementations for some simple methods that all subclasses are likely to use, while only providing interfaces for some other methods which subclasses must implement
    • Cannot use an interface for this
    • Define an abstract class instead

• An abstract class looks like a regular class, except
  ► Some methods are marked **abstract** and some are marked or **virtual**
Abstract vs Virtual Methods

- **An abstract method**
  - Only the name, return type, and parameter list is provided
  - There is no implementation – subclasses must implement
  - Must use `override` keyword in implementing method

- **A virtual method**
  - The entire method is implemented
    - Have name, return type, parameter list, and implementing statements
  - Use of virtual keyword signifies that subclasses are welcome to override
  - Subclasses may provide new implementation, but are not required to do so
    - If they do, must use `override` keyword
class Telephone
{
    public string PhoneNum {get; set;}   // Phone number property, can read and write

    virtual void display_phone_num()             // Output phone number, might be overridden by subclasses
    {
        System.Console.WriteLine(“Telephone number is {0}”, PhoneNum);
    }

    abstract bool call(string num_to_call);    // Abstract class, must be implemented by subclasses
}

public cellphone : Telephone
{
    public override void display_phone_num()       // Overrides implementation in Telephone
    {
        System.Console.WriteLine(“Cell phone number is {0}”, PhoneNum);
    }

    public override bool call(string num_to_call)         // Implements abstract method given in Telephone
    {
        ... // implementation of calling logic
    }
}
Given: class B is a subclass of class A

- class B: A { … }

Methods in B can override (re-implement) any method in A

- Do not need to use override keyword for this, just do it
- Only need to use override when method in A is marked virtual, or abstract
- Acts as a way of forcing programmer to focus on the intention of the person developing the parent class (A)
Interface vs Abstract Class

• Interface
  ► Pro
    • A class can inherit multiple interfaces
    • Interfaces can inherit (specialize) other interfaces
      – Can have rich hierarchies of interfaces
    • Can create containers with interfaces
      – Hold instances of any kind of subclass
    • Can create references to interfaces
      – Can refer to instances of any kind of subclass
  ► Con
    • Cannot provide any implementation, even for simple, generic methods


**Interface vs Abstract Class (cont’d)**

- **Abstract class**
  - **Pro**
    - Can provide implementation of some methods, while leaving others to be implemented by subclasses
    - Allows deferring some, but not all, implementation decisions to subclass
    - Useful when there needs to be a fixed call order among methods
    - Can implement method that defines call order, but leave implementation of called methods to subclasses
    - *Template Method pattern*
  - **Con**
    - Subclass can only inherit one Abstract class
    - Abstract class must be top of inheritance hierarchy
    - Parent class may make some implementation decisions subclass cannot easily change
*C# code tends to prefer interfaces over abstract classes*

- Interfaces are commonly used in C# code

*For game code*

- Can define interfaces for specific roles of objects in game
  - ICollidable (for objects that can collide with one another)
    - Player, Enemy, Bullets can all have distinct implementations, but still inherit from ICollidable
  - Then, use List<ICollidable> to hold all collidable objects
    - One list can hold objects of type Player, Enemy, and Bullet, even though their implementations are very different
  - Collision detection then only uses methods and properties in ICollidable interface
Homework

• Read:
  ► Chapter 5 (Inheritance and Polymorphism)
  ► Chapter 8 (Interfaces)
  ► In *Programming C# 3.0*

• Start work on work breakdown