CSC 143

More About Inheritance & Interfaces

Overview

- An assortment of topics related to inheritance
  - Class Object
  - toString
  - instanceof
  - Overloading and overriding
  - Abstract and concrete classes
  - Inheritance vs composition: which to use?
  - Abstract classes vs interfaces

Inheritance Reviewed

- A class can be defined as an extension another one
  - Inherits all behavior and state from base (super-) class
    But only has direct access to public or protected methods/variables
- Use to factor common behavior/state into classes that can be extended/specialized as needed
- Useful design technique: find a class that is close to what you want, then extend it and override methods that aren't quite what you need

Class Object

- In Java's class model, every class directly or indirectly extends Object, even if not explicitly declared
    class Foo{ … }   has the same meaning as   class Foo extends Object{ … }
- Class Object
  - is the root of the class hierarchy
  - contains a small number of methods which every class inherites (often overridden with something more suitable)
    toString( ), equals( ), clone( ), …

Aside – toString( )

- Most well-designed classes should override toString( ) to return a meaningful description of an instance
  - Rectangle[height: 10; width: 20; x: 140; y: 300]
  - Color[red: 120; green: 60; blue: 240]
  - (BankAccount: owner=Bill Gates, Balance = beyond your imagination)
- Called automatically whenever the object is used in a context where a String is expected
- Use with System.out for a crude, surprisingly effective debugging tool
  - System.out.println(unusualBankAccount);
  - System.out.println(suspectRectangle);

5

instanceof

- The expression
  - \<object\> instanceof \<classOrInterface\>
  is true if the object is an instance of the given class or interface (or any subclass or subinterface of the one given)
- Use should be rare in well-written code
  - Often overused by inexperienced programmers when method override and dynamic dispatch should be used
  - One common use: checking types of generic objects before casting
    ```
    Object o = aList.get(i);
    if (o instanceof ThingThatCanJump) {
        ThingThatCanJump t = (ThingThatCanJump) o;
        t.jump(veryHigh); ...
    ```

6

Overriding and Overloading

- In spite of the similar names, these are very different
- Overriding: Redefinition of a method in a derived (sub-) class
  - Replaces the method that would otherwise be inherited
    ```
    class One { ... public void doIt(...) { ... } ... }
    class Two extends One { ... public void doIt(...) { ... } ... }
    ```
  - Parameter lists must match exactly (number and types)
  - Method called depends on actual (dynamic) type of the object

7

Overloading

- A class may contain multiple definitions for constructors or methods
  ```
  class Many {
      public Many( ) { ... }
      public Many(int x) { ... }
      public Many(double x, String s) { ... }
      public void another(Many m, String s) { ... }
      public void another(String[ ] names) { ... }
  ```
- Known as *overloading*
- Parameter lists must differ in number or type of parameters or both
- Method calls are resolved automatically depending on number and types of arguments – must be a unique best match

8

Overriding vs Overloading

- Overriding
  - Provides an alternative implementation of an inherited method
- Overloading
  - Provides several implementations of the same method
    These are completely independent of each other
- Mixing the two – potentially confusing – avoid!
  - Pitfall: attempt to override a method, but something is slightly different in the parameter list. Result: new method overloads inherited one, doesn't override; new method doesn't get called when you expect it

9

What is a generic Animal?

- Example: class Animal (base class for Dog and Cat)
  - What noise should a generic Animal make?
  - Answer: doesn't really make sense!
- Purpose of class Animal
  - provide common specification for all Animals
  - provides implementation for some methods
  - intended to be extended, not used directly to create objects

10

Abstract Classes

- Idea: classes or methods may be declared abstract
  - Meaning: meant to be extended; can't create instances
- If a class contains an abstract method, it must be declared abstract
- A class that extends an abstract class can override methods as usual
- A class that provides implementation 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

11

Example: Animals

```
public abstract class Animal {          // abstract class
    // instance variables
    ....
    /** Return the noise an animal makes */
    public abstract String noise( ) ;
}

public class Cat extends Animal {          // concrete class
    /** Return the noise a cat makes */
    public String noise( ) { return "purrr"; }
}
```

12

Using Inheritance

- Java inheritance limitation: a class can only extend one class
- Use of inheritance, with or without abstract classes is only appropriate when the classes are related conceptually
  - Never use inheritance just to reuse code from another class
- Composition is normally appropriate if you want to use code in another class, but the classes are otherwise unrelated

```
class SomeClass {
    private ArrayList localList;  // class used to implement SomeClass
                                  // Does not make sense for SomeClass
                                  // to extend ArrayList
```

13

Abstract Classes vs Interfaces

- Both of these specify a type
- Interface
  - Pure specification, no implementation
- Abstract class
  - Specification plus, optionally, partial or full default implementation
- Which to use?

14

Interfaces

- Advantages
  - More flexible than inheritance; does not tie the implementing class to implementation details of base class
  - Classes can implement many interfaces
  - Can make sense for classes that are not related conceptually to implement the same interface (unrelated Things in a simulation, mouse click listeners in a user interface)
- But …
  - Can't inherit (reuse) a default implementation

15

A Design Strategy

- These rules of thumb seem to provide a nice balance for designing software that can evolve over time
  - (Might be a bit of overkill for some CSC143 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
- We'll see this frequently when we look at the Java libraries

16