

CSE 142

Abstract Methods and Interfaces

1/10/2003

(c) 2001-3, University of Washington

W-1

Outline for Today

- Review
 - Inheritance – types and implementations
- Today
 - Abstract classes – specification + partial implementation
 - Interfaces revisited

1/10/2003

(c) 2001-3, University of Washington

W-2

A New Application

- Suppose we're designing the classes for a pet store simulation
- Inheritance makes sense – we need particular classes for specific kinds of pets, along with a generic "Pet" class that captures common behavior
- Client code can either deal with specific kinds of pets, or generic pets

```
/** Print information about pet p */
public void printInfo(Pet p) {
    System.out.println("Pet's name is " + p.getName() + " and it weighs " +
        p.getWeight());
}
```

1/10/2003

(c) 2001-3, University of Washington

W-3

Specification of Class Pet

- State (instance variables)
 - Name (string)
 - Weight (double)
- Behavior (methods)
 - constructor
 - getName(), getWeight()
 - eat(String food)
 - speak()

1/10/2003

(c) 2001-3, University of Washington

W-4

Specification of a Particular Pet

- Extend Pet with details for particular kind of Pet

- Example

```
public class Cat extends Pet {  
    /** Construct new Cat */  
    public Cat(...) { ... }  
  
    /** Return a cat noise */  
    public String speak() {  
        return "Woof!";  
    }  
}
```

1/10/2003

(c) 2001-3, University of Washington

W-5

What Noise Does a Generic Pet Make?

- We want all Pets to be able to speak
 - So, we need to define method speak() in class Pet
- But how should we implement speak() in Pet?
 - Question doesn't really make sense
 - We want speak() in Pet to ensure it is part of the interface of all Pet objects
 - But there really isn't a sensible generic implementation
 - Classes that extend Pet are expected to provide something appropriate

1/10/2003

(c) 2001-3, University of Washington

W-6

Abstract Methods

- Idea: allow a method specification to be declared in a class *without* an implementation
- Syntax: add the word "abstract" to the method declaration; replace the body with a ";"
- A class with an abstract method is itself abstract and must be declared to be so

```
/** Representation of a generic Pet */  
public abstract class Pet {  
    ...  
    /** Return the noise this pet makes */  
    public abstract String speak();  
    ...  
}
```

1/10/2003

(c) 2001-3, University of Washington

W-7

Implications of Abstract Classes and Methods

- Instances of abstract classes can't be created
 - Abstract methods imply incomplete class implementation
- To be useful an abstract class must be extended
 - And implementations of abstract methods must be provided if instances are to be created
- Abstract classes define new types that can have partial implementations
- The partial implementation is inherited in extended classes, as usual

1/10/2003

(c) 2001-3, University of Washington

W-8

Interfaces Revisited

- An abstract class may have
 - some methods *with* implementations
 - some methods *without* implementation
- Sometimes we only want to define/describe a new type without providing any implementation at all
- Two choices in Java
 - Define an abstract class with only abstract methods
 - Define an *interface* – pure specification with no implementation (which of these to choose? More in a moment. But first...)

1/10/2003

(c) 2001–3, University of Washington

W-9

Interface Definition

- Much like a class definition, but no method bodies and no state (except for static final constants)
- Everything is implicitly abstract, so “abstract” doesn’t need to be written

```
/** Interface filter. All the filters implement this interface */
public interface Filter {
    /** Modify the image according to the filter algorithm */
    void filter(PixelImage theImage);
}
```

1/10/2003

(c) 2001–3, University of Washington

W-10

Using Interfaces

- Any class can implement any interfaces that make sense
- Syntax

```
public class <classname> implements <interfacename> {
    ...
}
```
- A class that implements an interface must provide implementations of all methods declared in the interface
 - No code is inherited from an interface
- An interface defines a new type
- Any class that implements an interface has the interface type (in addition to any others it defines or inherits)

1/10/2003

(c) 2001–3, University of Washington

W-11

Uses for Interfaces

- Allows a class to implement several different types
 - Can implement as many interfaces in a single class as makes sense for the application
- Allows otherwise unrelated classes to have common behavior
 - Example: BankAccounts and Strings are very dissimilar, except for having a *compareTo* method
 - Example: objects in a simulation can all implement behavior that allows them to participate in the simulation, even if they have nothing else in common
 - Simulation engine only needs to know about the common interface

1/10/2003

(c) 2001–3, University of Washington

W-12

Classes, Interfaces, Types, and Inheritance

- Classes and interfaces each define new types
- Classes can *extend* one other class and implement many interfaces
 - The new class has all of the types and members of the inherited class and implemented interfaces
- Interfaces can extend other interfaces
 - The new interface contains everything in the original interface plus anything new, and has all of the types involved
- Which do we use where?

1/10/2003

(c) 2000-3, University of Washington

W-13

Defining New Classes and Types: Guidelines

- For small, one-use classes, pick whatever is simplest
- For more widely used designs, a convention that's fairly common is the following
 - Define important new types as interfaces
 - If possible, provide a default implementation in a class that implements as much of the interface as possible
 - Client code can either
 - Implement* the interface – meaning provide implementations of all the methods
 - Extend* the default implementation, inheriting what makes sense, and overriding or implementing anything that needs to be customized
 - Maximum flexibility while also allowing code reuse when possible

1/10/2003

(c) 2000-3, University of Washington

W-14