COP 3330, Spring 2013

Interfaces

Instructor : Arup Ghosh
02-15-13

School of Electrical Engineering and Computer Science
University of Central Florida

Agenda

- For the next few weeks, we're going to do inheritance and related concepts
  - Interfaces (inheritance-lite)
  - Inheritance

Assignment 3

- The purpose is to test your ability to use classes and objects properly.

- The application uses five classes
  - Two of them are already written for you
  - You must write the other three

- The public members of each class are listed and described in the documentation provided with the assignment.
  - See the folder ‘doc’ in the zip file on Webcourses.
  - The file index.html will describe everything.

Basic guidelines

- Code up `Name` and `Homework` according to the specifications
- Don't touch the code in `Files` and `HomeworkQueue`!
  - Put them in your project folder
  - Read the documentation to figure out how to use them
- Allocation is a client for the remaining four classes.
- It will create objects from them and use those objects to solve the problem.
  - This is where all the I/O and other action happens

Sample I/O

- Sample I/O files are provided, as usual.
- Feel free to make your own I/O and use it for testing purposes.
- Try and follow it exactly
  - Some clever use of the `toString()` methods will make it fairly easy.

Interfaces

- NOT the same as GUIs, just a confusing name
- In common parlance, interface = a way for two systems to interact
- In OO terms, an interface is a way to neatly deal with objects of different classes but similar functionality
- Why? Because they *look similar* from the outside.
  - i.e., they present the same interface

Interfaces

- Interfaces specify methods that must be implemented in classes that implement the interface
- Example: The `Comparable` interface requires that a `compareTo` method be implemented
- Example: The `Iterable` interface requires that the `iterator` method be implemented

Interfaces

- You can define your own interfaces
- To create your own interface, use the keyword `interface` instead of `class`
- Interfaces cannot specify implementations for methods, only their prototypes

Noisy things

- Suppose you're writing a massive virtual reality world, and you have objects for everything.
- When moving through the world, the user should hear ambient sounds.
- But lots of very different things make sounds:
  - Animals
  - The wind
  - Playing children
  - Traffic
  - …

Common functionality

- Let's say that every class that has noisy objects defines a `makeNoise()` method
- They obviously work differently
  - The noise made by a `Dog` is different from that made by a `Car`
- When we add a new entity to the world (i.e., make a new class for it) we can give it a `makeNoise()` method if it is Noisy.
- As per encapsulation, every class is responsible for its own noise-making

Motivation

- Suppose you want to write a function that takes a noisy object and its distance from the user, and plays a sound accordingly.

- For example:

```
void playNoise(Dog noisyDog, double dist) {
    Sounds.setVolumeFromDist(dist);
    noisyDog.makeNoise();
}

void playNoise(Child noisyKid, double dist) {
    Sounds.setVolumeFromDist(dist);
    noisyKid.makeNoise();
}
```

Motivation

- Annoyingly, those two functions are exactly identical.
- Only difference is the type of the object being passed to them.
- Feels very redundant.
- Worse, we have to keep remembering to add a new method every time we make a new class that also makes noise.

Motivation II

- Suppose you want to write a function that takes the noisy objects surrounding the user and plays ambient noises accordingly.
- Naturally we want to pass the noisy objects in an array.
- But how? The types are all different.
  - If I make a `Dog[]` array, I can't put objects of `Car` or `Child` into it.
- Same problem all over again

A neat solution

- In an ideal world, all noisy objects would share the same class `Noisy`.
  - Then we can just write one function, and store all such objects in a `Noisy[]` array.

- Sadly, this is not possible. The objects have other functions beyond making noise.

- But, what if we can *treat them* as the same type?

The Noisy interface

```
public interface Noisy {
      public void makeNoise();
}
```

- This represents a contract between all objects that make noise.
- They all agree to define a public `makeNoise()` function.
- In exchange, they can be treated as having the type `Noisy`.

The implements keyword

```
class Dog implements Noisy {
      ...   // All the usual stuff

      public void makeNoise() {
            Sounds.playSound("dog_bark.mp3");
      }
}
```

- Once this is added, the Dog class MUST have the method(s) specified in the interface Noisy.
  - Compiler error if this is not done.

Interfaces and method bodies

- Notice that the interface only specifies a method *signature.*
- But it ends right there – no method body is defined.
  - In fact, you cannot define methods inside an interface.
  - Just method signatures. Prototypes, if you will.
- However, any class that implements an interface must provide a method body.
- Basically it's a rule saying, "You have agreed to have methods that look like this".

Polymorphism

- Now we can treat these objects the same.
- Say we have a `Dog, Child` and `Car` object – called `dog, child` and `car` respectively. All 3 classes implement `Noisy.`
- `Noisy m = child;`
- `Noisy[] noisyThings = new Noisy[] {dog, child, car};`
- I can treat each of these three types as the type `Noisy`!
  - This is polymorphism – these classes have *multiple forms*.

Interfaces: is-a relationship

- Interfaces define an is-a relationship, so an object of a class that implements a particular interface can be viewed as an instance of that interface
- For example (referring to the previous slide) a Child object is-a Noisy
  - This is legitimate code:

```
Noisy x = new Child();
```

Polymorphism

- Polymorphism refers to the ability to call different code with the same method call depending on the type of the object
- Another Example:

```
Point[] p = new Point[3];
p[0] = new Point2D(5,7);
p[1] = new Point3D(1,2,3);
p[2] = new Point4D(5,6,7,8);
for(int i=0; i<p.length;i++)
  System.out.println(p[i].magnitude());
```

Polymorphism

- The term dynamic binding refers to determination of which code to branch to at run-time instead of compile-time

Interfaces

- Interfaces specify methods that must be implemented in classes that implement the interface
- Example: The `Comparable` interface requires that a `compareTo` method be implemented
- Example: The `Iterable` interface requires that the `iterator` method be implemented

Polymorphism in methods

- And now we can write those functions from before.

```
void playNoise(Noisy noisyThing, double dist) {
    Sounds.setVolumeFromDist(dist);
    noisyThing.makeNoise();
}
```

- Now these are all valid, and the same function handles them.

```
playNoise(dog, 3.0);
playNoise(cat, 1.5);
playNoise(car, 20.3);
```

Methods in an interface

- The example interface defined just one method.
- But in general, an interface can define any number of methods (and even no methods at all).
- A class that implements an interface must provide a method body for *ALL* methods in the interface.

Interfaces define types

- The name of the interface (`Noisy` in our example) is a *type*, just like a class name.
- You can't instantiate an object of this type. It has no constructor.
- But you can have `Noisy` variables, which will store an object of any class, as long as it implements `Noisy`.

From the outside…

- If you have a variable of type Noisy, you can only call the method `makeNoise()` on it.
- What if it refers to a Dog object? Dog can have other methods like `wagTail()`, `doTrick()`, `eat()` and so on.
- They're not visible.
  - Why? Because it could just as easily refer to a `Car` object, which would not have these methods.
- All you can access are the methods of `Noisy` – the *common interface*.

Example

```
Noisy thing = dog;              // Polymorphic
Noisy otherThing = child;       // assignment

thing.makeNoise();              // VALID.
thing.wagTail();                // NOT VALID!
otherThing.playGame();          // NOT VALID!
otherThing.makeNoise();         // VALID.
```

Methods from the object's true type are not visible.

Methods from the type of the *containing variable* are visible.

Summary

- Interfaces are used to unify the treatment of different classes that have similar functionality.

- An interface declares zero or more methods (without bodies) that must be present (with bodies) in any class that implements the interface.

- An interface can be treated as a type, in that you can make variables of that type. Such variables can polymorphically hold an object of any type that implements the interface