COP 3330, Spring 2013

Code Examples: Interfaces

Instructor : Arup Ghosh
02-20-13

School of Electrical Engineering and Computer Science
University of Central Florida

Recap

- An interface declares (i.e., no function bodies) zero or more methods that must be defined (i.e., with function bodies) in any class that implements the interface.
  - This is done with the `interface` keyword

- The interface name is a type, and is used for polymorphism.
  - Any class implementing the interface can also act as if it has another type – the interface type.
  - E.g., if `Dog` implements `Noisy`, then a `Dog` object is both a `Dog` and a `Noisy`.

Recap- Why use interface?

- To achieve fully abstraction.
- To simulate multiple inheritance.
- Can be used to achieve loose coupling.

Today

- Code for the Noisy example from last time.

- Next day - Comparable interface.
  - Custom sorting using `Collections.sort()`

Noisy example

- This is basically a bunch of toy classes pretending to implement a virtual world.

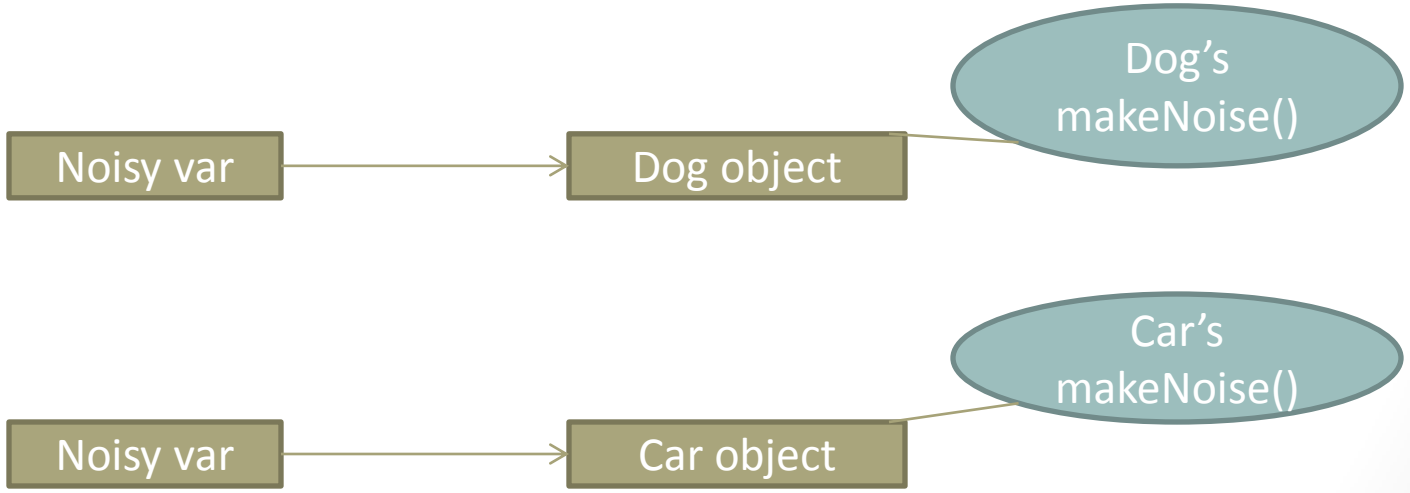
- I'll just put print statements in most methods, since we're just illustrating a concept.

- Focus on:
  - Creating the interface `Noisy`
  - Implementing it in the classes `Dog, Car and Child`.
  - The polymorphic usage of `Noisy` variables.

Dynamic binding

- Notice how calling `makeNoise()` went straight to the right implementation?

- This is an example of *dynamic binding.*



Dynamic binding

- The compiler only sees a Noisy variable.
  - It doesn't know the *real type* of the object it refers to.
- Normally, when you put a function call in your code, the compiler links it to the right function in the right class.
  - How? For normal cases, looks at object type and matches the name to a method in that class.
- This is called *static binding* (no relation to the static keyword) or compile-time binding.
- But for cases with interfaces, real type information is not available at compile time.

Dynamic binding

- For polymorphic method calls, the method is bound *at runtime.*
- During execution, the JVM examines the object and figures out its true type.
- Then it directs the method call to the right class.
- Thus, binding is *dynamic* – at runtime.
- We'll see this again with inheritance.