The Evolution of Interfaces

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/5448 — Lecture 30 — 12/10/09

Credit where Credit is Due

- Some of the material for this lecture is taken from “Programming in Scala” by Martin Odersky, Lex Spoon, and Bill Venners
  - as such some of this material is copyright © 2007, 2008 Odersky, Spoon and Venners
- In addition, some material is taken from “Ruby For Rails” by David Black
  - as such some of this material is copyright © 2006 Manning Publications

Goals for this Lecture

- Examine mechanisms in more recent OO languages for evolving the concept of “interface”, providing flexibility in specifying the types of an application
    - Go (briefly)
    - Clojure (briefly)
    - Scala
    - Ruby
- Wrap up the semester

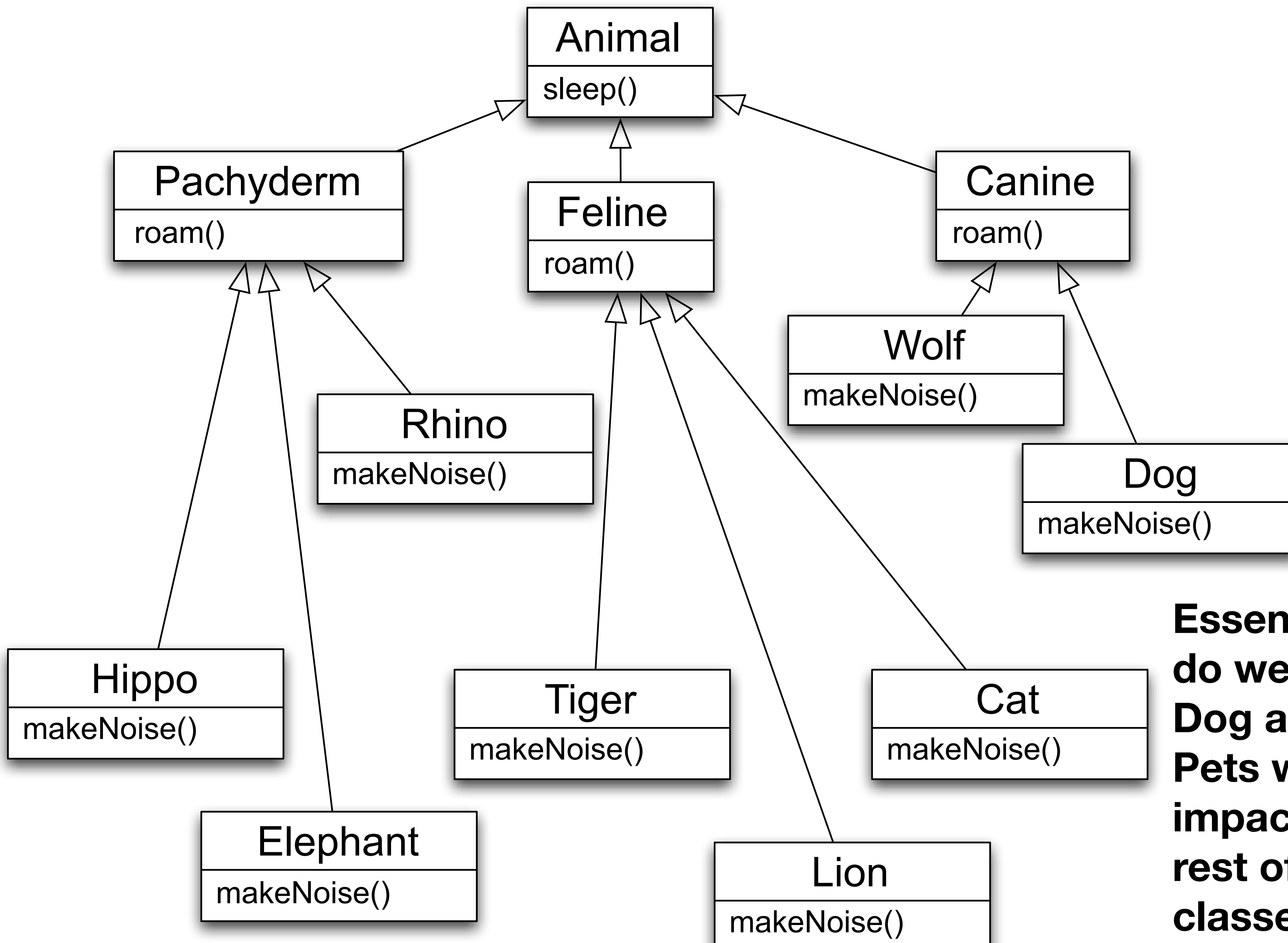
Review: (Lecture 3) Relationships: Interfaces

- A class can indicate that it implements an interface
  - An interface is a type of class definition in which only **method signatures** are defined
- A class **implementing** an interface provides method bodies for each defined method signature in that interface
  - This allows a class to play different roles, each role providing a different set of services
    - These roles are then independent of the class's inheritance relationships
- Other classes can then access a class via its interface
  - This is indicated via a "ball and socket" notation

Reminder (Lecture 4): Interface Example

- Consider modifying the Animal hierarchy to provide operations related to pets (e.g. play() or takeForWalk())
  - We have several options, all with pros and cons
    - add Pet-related methods to Animal
    - add abstract Pet methods to Animal
    - add Pet methods only in the classes they belong (no explicit contract)
    - make a separate Pet superclass and have pets inherit from both Pet and Animal
    - make a Pet interface and have only pets implement it
      - This often makes the most sense although it hinders code reuse
      - Variation: create Pet interface, but then create Pet helper class that is then composed internally and Pet's delegate if they want the default behavior

Reminder (Lecture 4): Animals (With Inheritance)



Essentially, how do we make Dog and Cat be Pets without impacting the rest of the classes?

Considering the alternatives (I)

- Consider modifying the Animal hierarchy to provide operations related to pets (e.g. play() or takeForWalk())
  - We have several options, all with pros and cons
    - **add Pet-related methods to Animal**
      - This approach is sub-optimal because non-Pet classes receive Pet behaviors via inheritance; you would be forced to override those behaviors to raise an exception for non-Pets.

Considering the alternatives (II)

- Consider modifying the Animal hierarchy to provide operations related to pets (e.g. play() or takeForWalk())
  - We have several options, all with pros and cons
    - **add abstract Pet methods to Animal**
      - even worse than previous solution!
        - every subclass gets Pet methods AND has to implement them
          - with the former method, you at least could take advantage of code reuse

Considering the alternatives (III)

- Consider modifying the Animal hierarchy to provide operations related to pets (e.g. play() or takeForWalk())
  - We have several options, all with pros and cons
    - **add Pet methods only in the classes they belong (no explicit contract)**
      - With this approach (at least in Java and languages similar to it), you lose the advantage of having a type called Pet
        - Instead, your code just has to know that Dog IS-A Pet and that it can invoke Pet operations on it. It also had to know that Dogs and Cats can be treated similarly via their shared Pet methods
        - But you would get no support for the type system!
          - You can’t do this: **Pet p = new Dog();**
          - You can’t do this: **Pet[] p = [new Dog(), new Cat(), new Dog()];**

Considering the alternatives (IV)

- Consider modifying the Animal hierarchy to provide operations related to pets (e.g. play() or takeForWalk())
  - We have several options, all with pros and cons
    - **make a separate Pet superclass and have pets inherit from both Pet and Animal**
      - Multiple Inheritance: enough said

Considering the alternatives (V)

- Consider modifying the Animal hierarchy to provide operations related to pets (e.g. play() or takeForWalk())
  - We have several options, all with pros and cons
    - **make a Pet interface and have only pets implement it**
      - This often makes the most sense although it hinders code reuse
      - Variation: create Pet interface, but then create Pet helper class that is then composed internally and Pet's delegate if they want the default behavior

The landscape is evolving...

- New language features are offering additional alternatives to the ones above
  - or, in one case, removing the cons associated with one of the alternatives

- Consider the use of “interface” in the Go programming language...

The basics of Go (I)

- No explicit support for objects, no type inheritance, no generics, etc.
- To get object-like support, first define a struct

```
type File struct {
    fd      int;    // file descriptor number
    name    string; // file name at Open time
}
```

- and create a factory:

```
func newFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    return &File{fd, name}
}
```

The basics of Go (II)

- To use the factory

```
var Stdin = newFile(0, “/dev/stdin”);
```

- The type of Stdin is *File. To define a method that operates on Files do:

```
func (file *File) Close() os.Error {…}

func (file *File) Read(b []byte) (ret int, err os.Error) {…}

func (file *File) Write(b []byte) (ret int, err os.Error) {…}
```

- The syntax therefore is:

```
    func <receiver>? <name>(<params>) (<return types>) <body>
```

- On to interfaces...

Interfaces in Go

- Interfaces are special types in Go that define method signatures

```
type reader interface {
    Read(b []byte) (ret int, err os.Error);
}
```

- This defines a type name called “reader” and this type name can now be used anywhere a type name can appear in Go:
  - as a receiver, as a parameter, as a return type
- What’s more, an “object” (struct + methods) does not have to declare that it implements an interface: Instead, if it has all the methods defined by the interface it automatically matches!
  - We can pass a *File to ANY method that accepts a **reader** as a parameter
  - We can invoke any method on *File that says its receiver is a **reader**

Back to Pets

- What this means to our previous example is that Go has eliminated some of the cons associated with this alternative
    - **add Pet methods only in the classes they belong (no explicit contract)**
- We can now define a Pet interface that specifies method signatures associated with Pets
- We can then define methods for Dog and Cat that match the ones in that interface
    - We can then put Dogs and Cats in Pet collections and we can create a new pet variable that points at a Dog or a Cat
- We get the benefits of interfaces but with no need for a class to specify that it implements that interface, the compiler simply takes care of it
    - **Demo** (Note: What con is still present in this approach?)

Clojure Destructuring (I)

- Go's approach to interfaces is similar to what is called "duck typing"
  - If it walks like a duck and quacks like a duck, it's probably a duck
  - If an object responds to one or two methods of Duck, it's probably a Duck
- A similar feature (although in reverse) can be seen in clojure

```
(defstruct author :first-name :last-name)

(def erikson (struct author "Steven" "Erikson"))

(defn greet-author-1 [author]
  (println "Hello," (:first-name author)))

(greet-author-1 erikson)

(defn greet-author-2 [{fname :first-name}]
  (println "Hello," fname))

(greet-author-2 {:last-name "Vinge" :first-name "Vernor"})

```

Clojure Destructuring (II)

- In greet-author-1, the definition states that a single argument should be passed in, but it doesn't say what that argument should be
    - This is common in all languages that use dynamic typing; you can let anything be passed in and won't find out until run-time whether it will work or not (this is a feature not a bug!)
- In greet-author-2, the definition states that a single argument should be passed in, further more it states that it should be a map, and that map should include the key :first-name
    - [{fname :first-name}]
- In essence, this defines an "interface" that says you can pass any map to me at all (or anything that acts like a map) as long as that map has a :first-name key
    - with this information, the run-time system can be a little smarter and warn you if you pass a non-map to this function.

Scala Traits

- Scala (Scalable Language) has a feature called “traits” that provide flexibility in how Scala applications define their type hierarchies
- First some basics
  - In Scala, the top most type is Any which implements ==, !=, equals, hashCode, and toString
    - It has two subclasses, AnyValue and AnyRef
      - Under AnyValue are the “primitive” classes, such as Int, Float, …
      - Under AnyRef are “reference” classes, such as String, List, etc.
  - Scala has two “bottom types”: Null and Nothing
    - Null is a subclass of all “reference” classes
      - It allows you to say things like: **var myList : List = null;**
    - Nothing is the “bottom most type” of Scala, it is a subclass of all other types; it has no values and it is used to handle abnormal termination

Nothing type in Scala

- For instance, Scala has a method that looks like this
  - def error(m: String): Nothing = throw new RuntimeException(m)
- The return type is Nothing because this method throws an exception and will likely cause the program to terminate
- Because Nothing is a subclass of all other types, you can write code like this
  - def divide(x: Int, y: Int) : Int =
    - if (y != 0) x / y else error("can't divide by zero")
- The true branch has an expression that evaluates to Int
- The false branch has an expression that evaluates to Nothing
  - but since Nothing is a subtype of Int, the type of the "if" statement is Int, as required by the return type of the method
- As you can see, Scala's type system already provides some interesting features; now let's look at traits

Traits (I)

- Scala Traits are “interfaces on steroids”
  - They can be used like Java interfaces and simply define a set of method signatures; they define a type that can then be referenced and other classes can declare that they implement that type
- But
  - Unlike Java interfaces, traits can define instance variables and method bodies, when a class extends the trait it gains access to these definitions, enabling code reuse
  - Traits are therefore designed to be mixed into different parts of the class hierarchy

Traits (II)

- Mechanics: Traits are defined like classes but with keyword “trait”

```
trait Philosophical {
  def philosophize() {
    println(“I consume memory, therefore I am!”)
  }
}
```

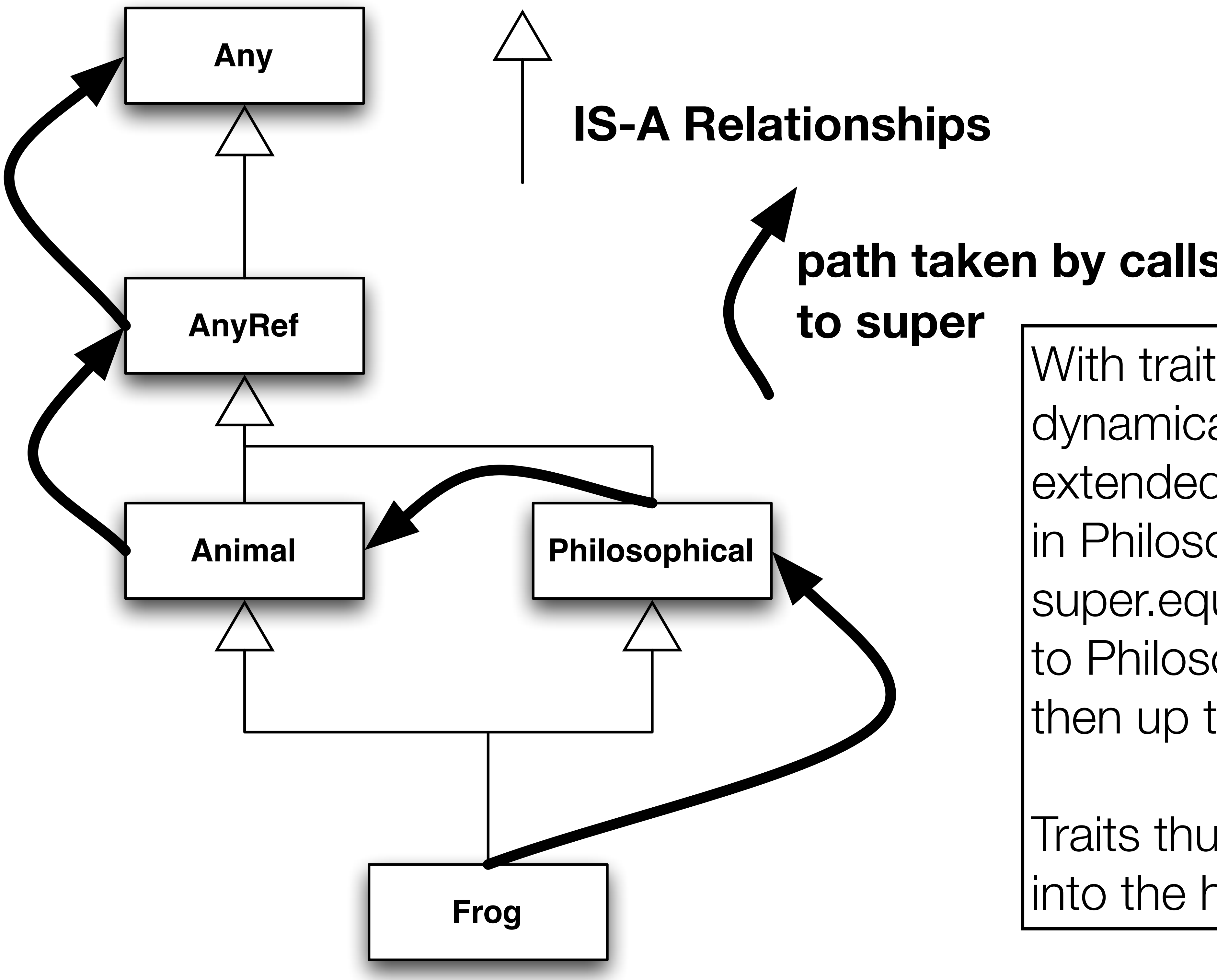
- If a class uses a trait directly, it is mixed in via the extends keyword

```
class Frog extends Philosophical {
  override def toString = “green”
}
```

- If a class extends a class AND uses a trait, the trait is mixed in via “with”

```
class Animal
class Frog extends Animal with Philosophical {
  override def toString = “green”
}
```

Relationships



With traits, calls to super are dynamically bound; since Frog extended Animal but then mixed in Philosophical, if it calls super.equals(), the call first goes to Philosophical, then to Animal, then up the tree to Any;

Traits thus interpose themselves into the hierarchy

Two uses of Traits

- Providing rich interfaces via a small number of abstract methods
    - A trait will often define a small number of abstract methods that need to be implemented by a class that uses the trait
        - It will then define a larger number of methods in terms of the abstract methods, providing the class that uses the trait with a “rich interface”
        - trait Ordered for instance defines <, >, <=, >= methods in terms of an abstract “compare” method; a client class implements compare in a way that makes sense for it and then gets the four methods above for free
- Providing stackable modifications
    - Small traits (one or two methods) that provide services that can be combined into a set of classes with a range of different behaviors

```scala
abstract class IntQueue {
  def get() : Int
  def put(x : Int)
}

import scala.collection.mutable.ArrayBuffer
class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}

trait Doubling extends IntQueue {
  abstract override def put(x : Int) {
    super.put(2 * x)
  }
}

trait Incrementing extends IntQueue {
  abstract override def put(x: Int) {
    super.put(x + 1)
  }
}

trait Filtering extends IntQueue {
  abstract override def put(x: Int) {
    if (x >= 0) super.put(x)
  }
}
```

Stackable behavior via Traits

With these definitions, you can create a doubling, filtering IntQueue with the following declaration

val q = (new BasicIntQueue with Doubling with Filtering)

q.put(-1)
q.put(0)
q.put(1)

q.get() ; returns 0
q.get() ; returns 2

The -1 does not appear in the queue because it gets filtered out by the Filtering trait

Back to Pets

- Traits in Scala change this alternative:
  - **make a Pet interface and have only pets implement it**
- to:
  - **make a Pet trait and have only pets extend it**
- By making a Pet trait, you could provide default implementations for each of the Pet methods which individual animals can override if needed
  - You don't lose out on code reuse and you don't have to go the route of creating a helper object that each Pet composes and then delegates to

Ruby Modules

- Ruby has a feature that is similar to Scala traits called modules
  - modules are simply bundles of constants, instance variables and methods
  - modules cannot be instantiated; they have to be mixed into other classes
- However, the class Class is a subclass of class Module
  - so, Classes are simply Modules that can be instantiated
- Method lookup is similar to Scala traits
  - when a method m is invoked on object o, the search goes
    - does o's class have method m?
    - does o's class mix in a module?
      - If yes, does it have method m?
    - does o's superclass have method m?
    - does o's superclass mix in a module? ...

If a class mixes in more than one module, then the search will look at each module in reverse order of how it was included in the class

Example

```ruby
module Stacklike
  attr_reader :stack

  def initialize
    @stack = Array.new
  end

  def add_to_stack(obj)
    @stack.push(obj)
  end

  def take_from_stack
    @stack.pop
  end
end

class Stack
  include Stacklike
end
```

To use this code, you can now say things like

s = Stack.new
s.add_to_stack("a")
puts s.take_from_stack()

Stack is an empty class until it imports the code from the Stacklike module

Back to Pets

- Modules in Ruby change this alternative:
  - **make a Pet interface and have only pets implement it**
- to:
  - **make a Pet module and have only pets include it**
- By making a Pet module, you can provide default implementations for each of the Pet methods, which individual animals can override if needed
  - You don’t lose out on code reuse and you don’t have to go the route of creating a helper object that each Pet composes and then delegates to
- Note: UNLIKE Scala traits, Ruby modules do not have a notion of defining method signatures that are implemented by other classes

Wrapping Up

- What have we learned this semester?
  - Fundamental OO concepts, terminology and notations
  - OO analysis and design techniques
  - OO principles, patterns and life cycles
    - Adaptor, Command, Composite, Decorator, Factory, Flyweight, Iterator, MVC, Observer, Proxy, Singleton, State, Strategy, Template Method
  - UML (class, sequence, activity, state, use case)
  - Refactoring, Test-driven design
- Solid foundation in becoming not just a programmer but a DESIGNER
  - Have a good Winter break!