Programming Language Design

11/18 Interface, Mixin, and Trait

What is object?

- Here we don't discuss how to model and solve your problem with objects
  - Leave them to dedicated course such as software engineering

- Contain a piece of code

  --- From the viewpoint of programmers

  - Can hold state at runtime
  - Can define its behavior

What is class?

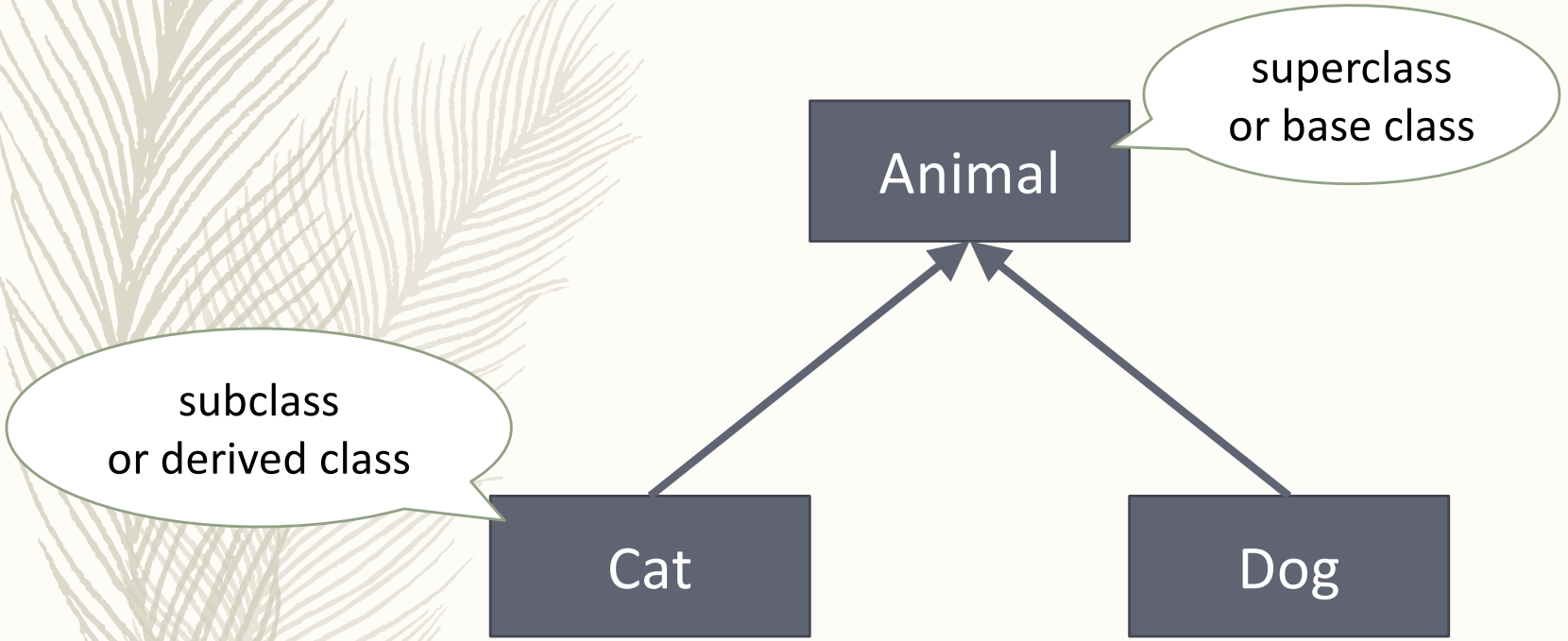
- The template for similar objects
  - They have the same properties
- Different objects of the same class may have different state
  - But share the same behavior
- A way to define such a template
  - Named class-based OOP
  - We will see prototype-based OOP in next lecture

Why we need class and class hierarchy?

- Code reuse!
- Class
  - Reusable: every object share the code for defining the properties
  - But flexible: every object can have different states
- Class hierarchy
  - Common properties among different subclasses are further gathered in their superclass

Recall: Inheritance

– A-kind-of; is-a relation



Recall: Polymorphism in OOP

- The objects of subclasses can behavior differently
  - according to the definition of their subclasses

- For example,

```
Animal c = new Cat();
c.hello();     // meow~
Animal d = new Dog();
d.hello();     // bark!
```

Recall: How hello() is invoked?

- The method hello() is defined in the class Animal
- However, we need to invoke the proper implementation
  - According to its actual type
  - "meow~" for Cats, "bark!" for Dogs

Apparent type: Animal
Actual type: Cat

```
Animal c = new Cat();
c.hello();    // meow~
```

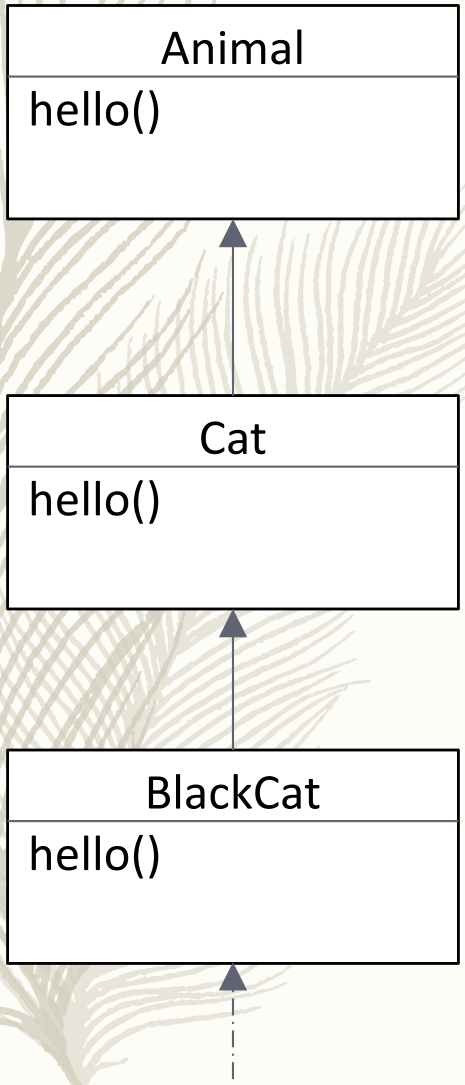
Apparent type: Animal
Actual type: Dog

```
Animal d = new Dog();
d.hello();    // bark!
```

Recall: Dynamic dispatch

- Dispatch the message to proper method implementation
  - According to the actual type at runtime
- The lookup of the implementation for a method call is performed at runtime
  - Since the actual type cannot be known until the message is sent
  - Flexible but expensive

Recall: Dynamic dispatch (cont.)



- `Animal c;`
  `:`
  `c.hello();`

- How to find out the proper implementation for hello()?
  - Check the actual type
  - Check if the class have an implementation for it
    - *If YES, we found*
    - *If NO, check its superclass*
- The inheritance chain might be very long
  - The cost of lookup is expensive

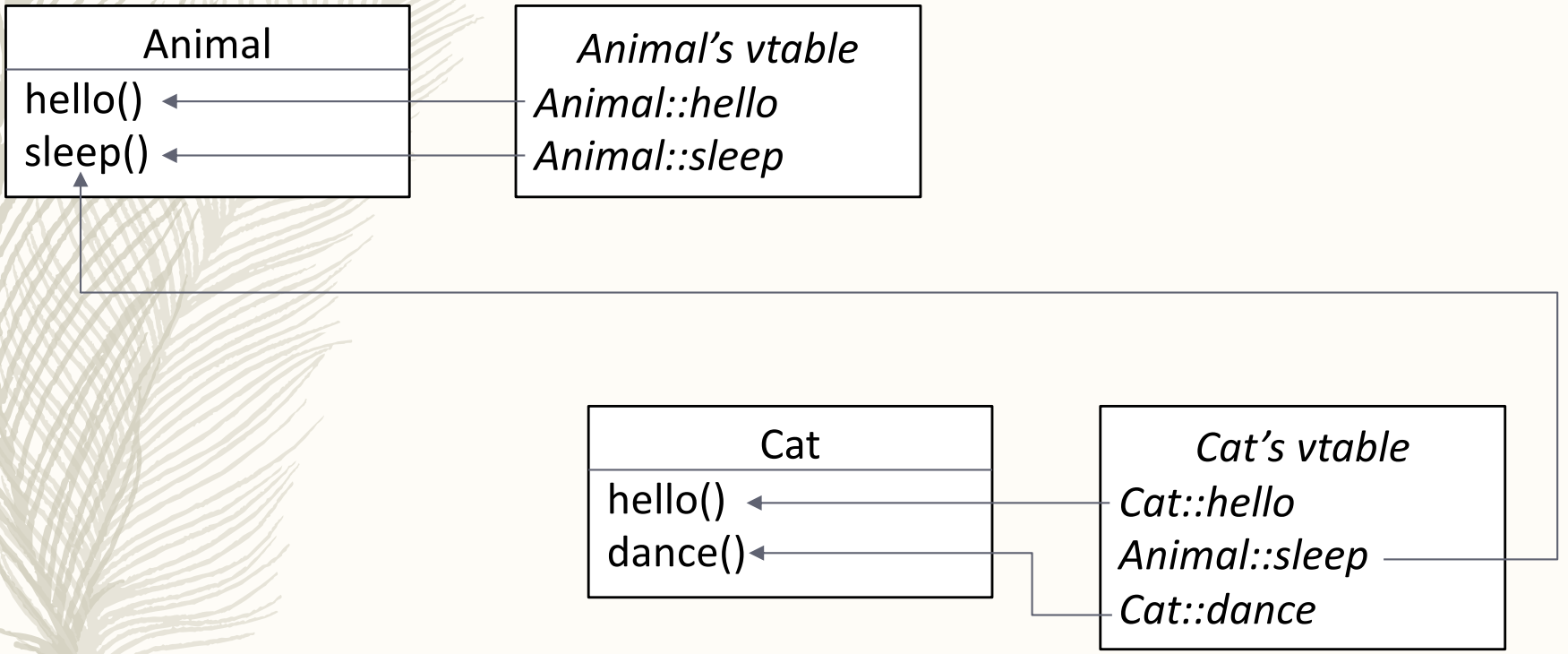
Recall: When is it used?

- In C++
  - Default is static
  - But dynamic dispatch is used when using pointer to call the function f that is virtual in that class c
  -
```
C* p;
   :
p->f();    // f is declared as virtual in C
```

- In Java
  - Default is using dynamic dispatch
  - But is static if the method m is declared as final

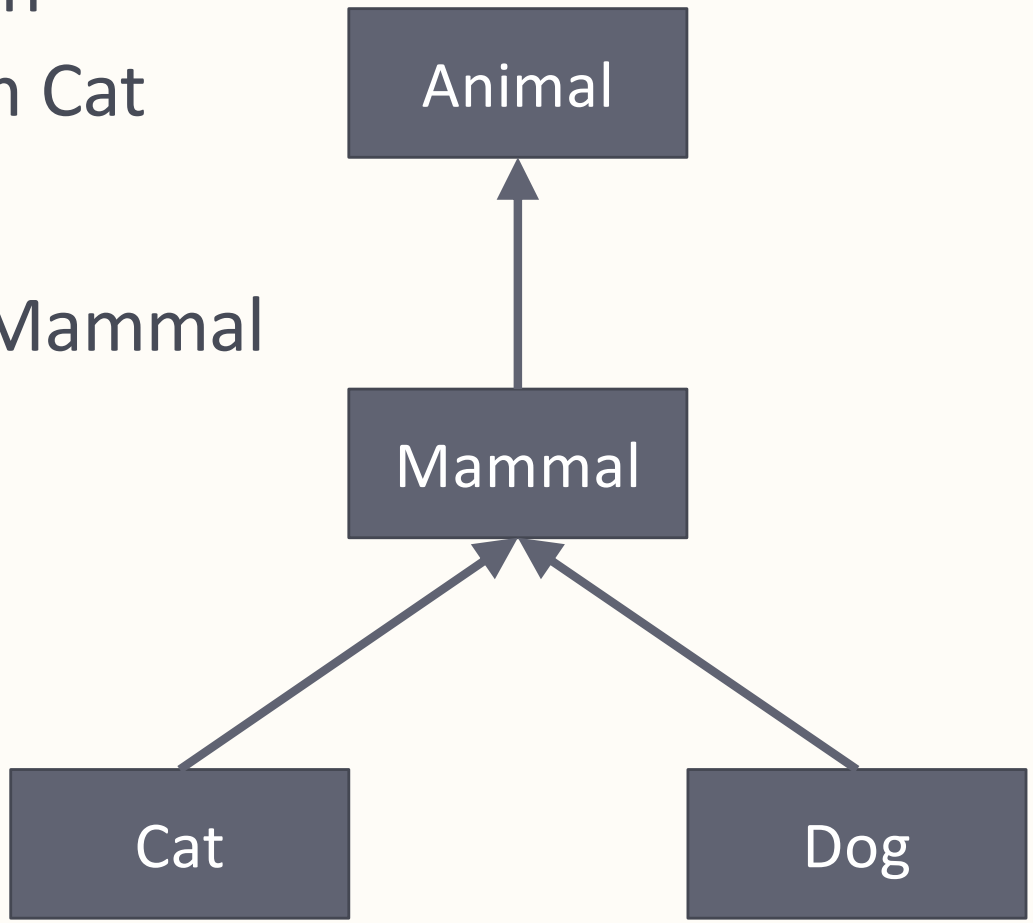
Recall: Virtual table in C++

- One virtual table (vtable) per class
  - All object instances of the class share the same vtable
  - Each entry is a function pointer to the most derived function



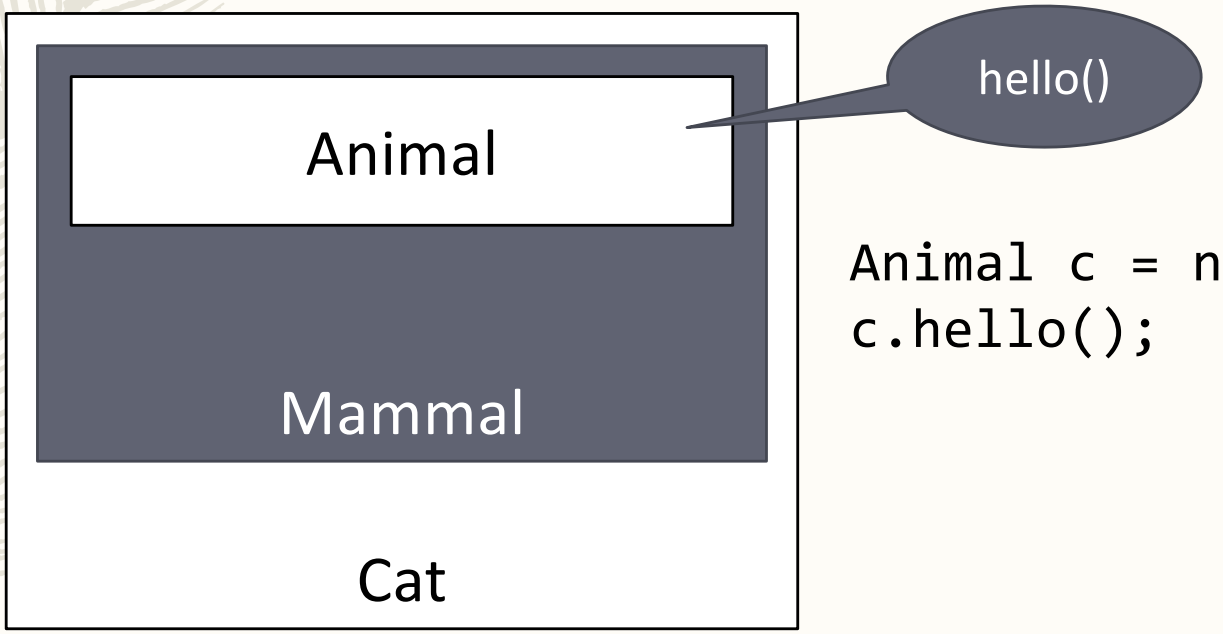
Recall: Inheritance (cont.)

- Extract common properties from Cat and Dog
- Put them into Mammal



Inheritance and object instance

- We can simply think
  - A Cat object instance contains a Mammal object instance
  - A mammal object instance contains an Animal object instance
- So we can invoke hello() on the instance of Animal




```
Animal c = new Cat();
c.hello();
```

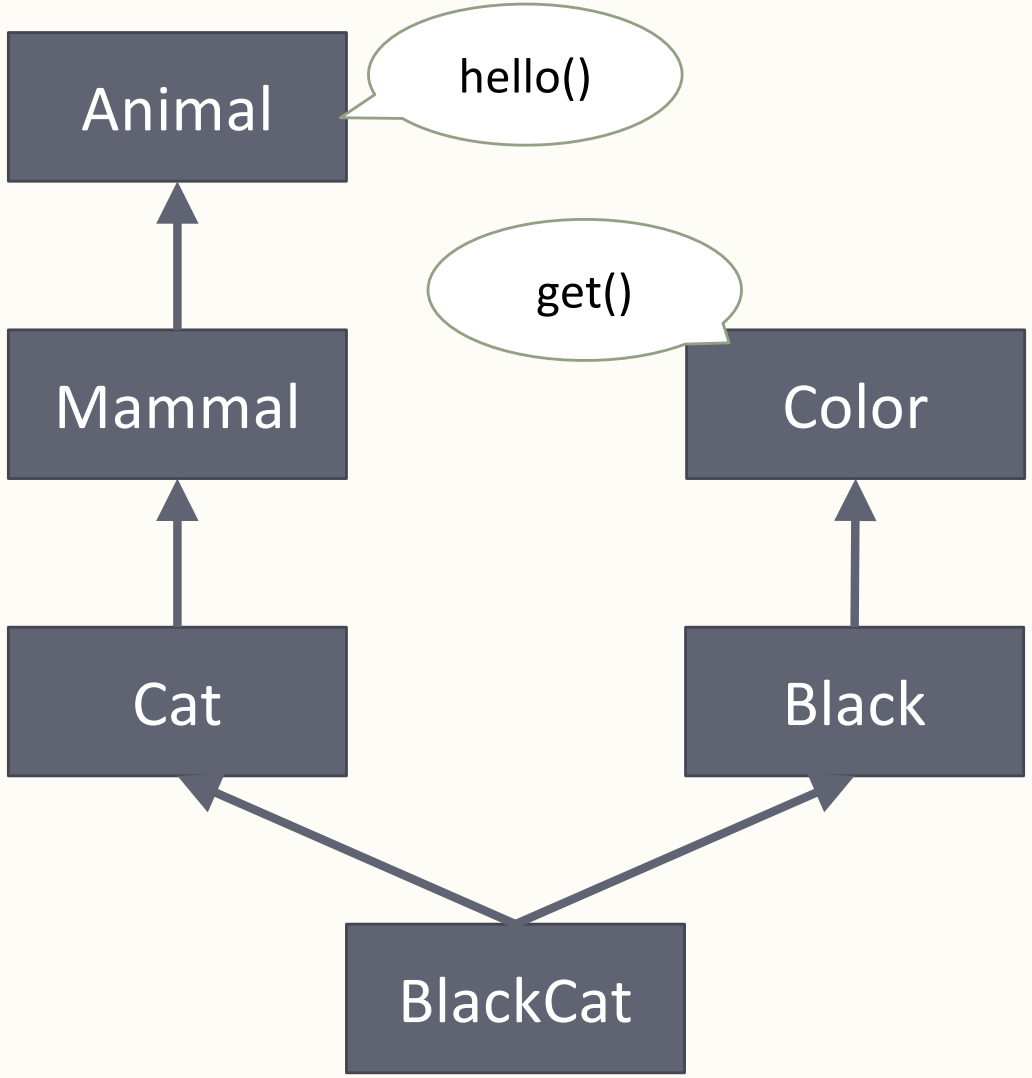
Multiple inheritance

- Why we need it?
  - Inherit properties from multiple classes

- For example,
  - BlackCat inherit from Cat and Black
  - Bat inherit from Mammal and WingedAnimal
  - Smartphone inherit Phone, Computer, and Camera

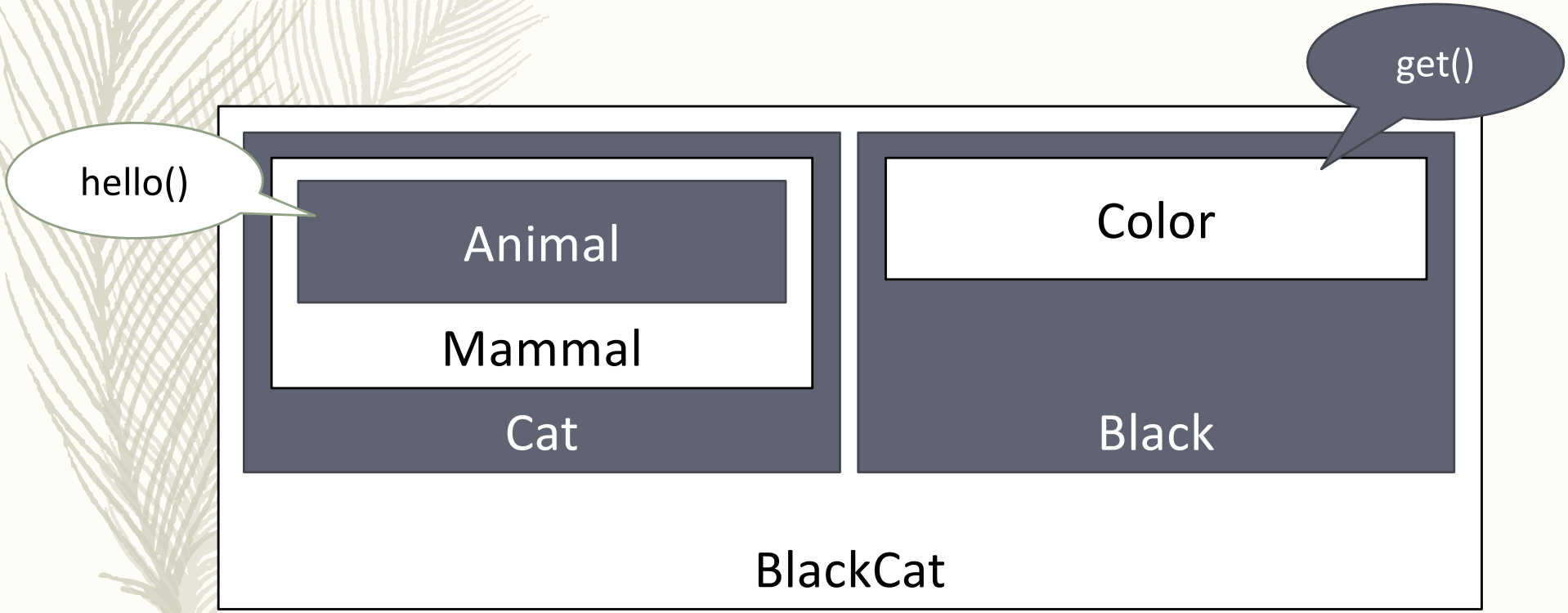
Multiple inheritance (cont.)

- Take BlackCat as an example
- We have a hello() in the instance of Animal
- And a get() in the instance of Color



Multiple inheritance (cont.)

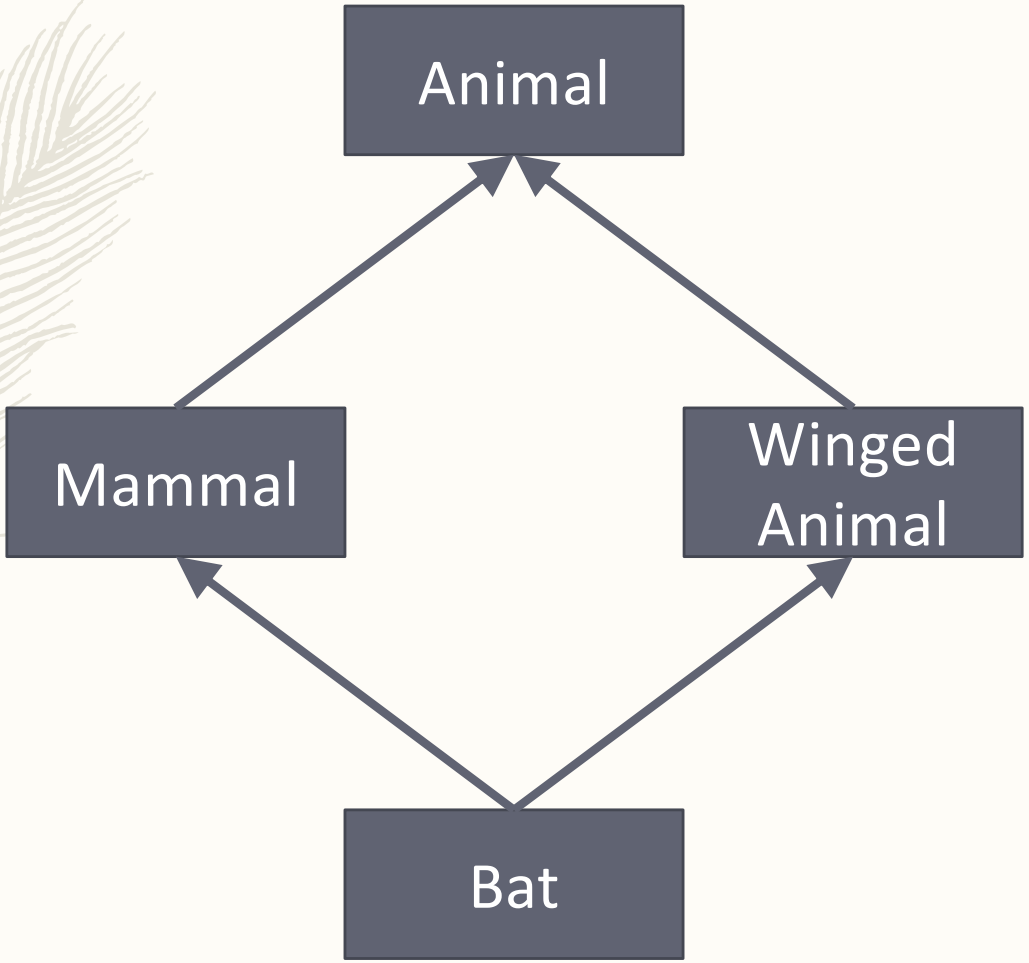
```
BlackCat b = new BlackCat();
b.hello();  /* the one inherit from Mammal,
               which inherit from Animal */
b.get();    // the one inherit from Color
```



– Here we simply regard it as a container without thinking about the order

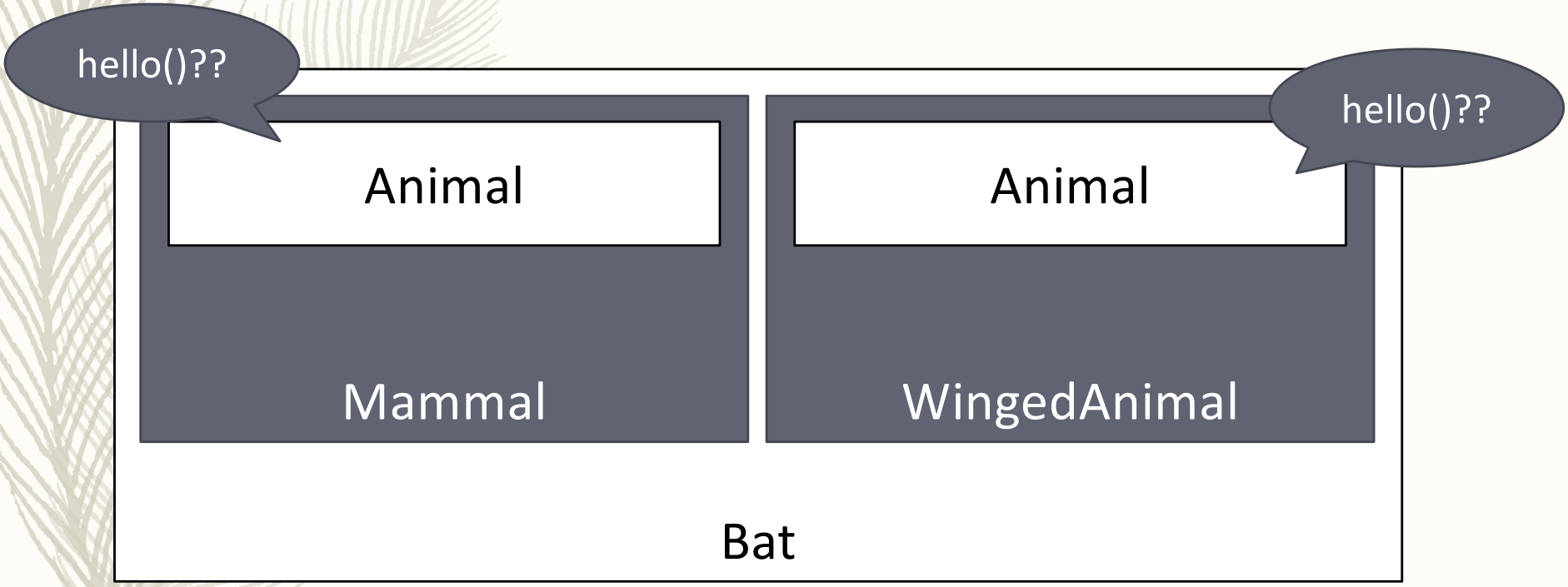
How about this case?

- Both Mammal and WingedAnimal inherit from Animal
  - There are two instances of Animal!



There are two hello() in a Bat object

```
Animal b = new Bat();
b.hello();  /* invoke the one inherit from
               Mammal or Winged Animal?? */
```



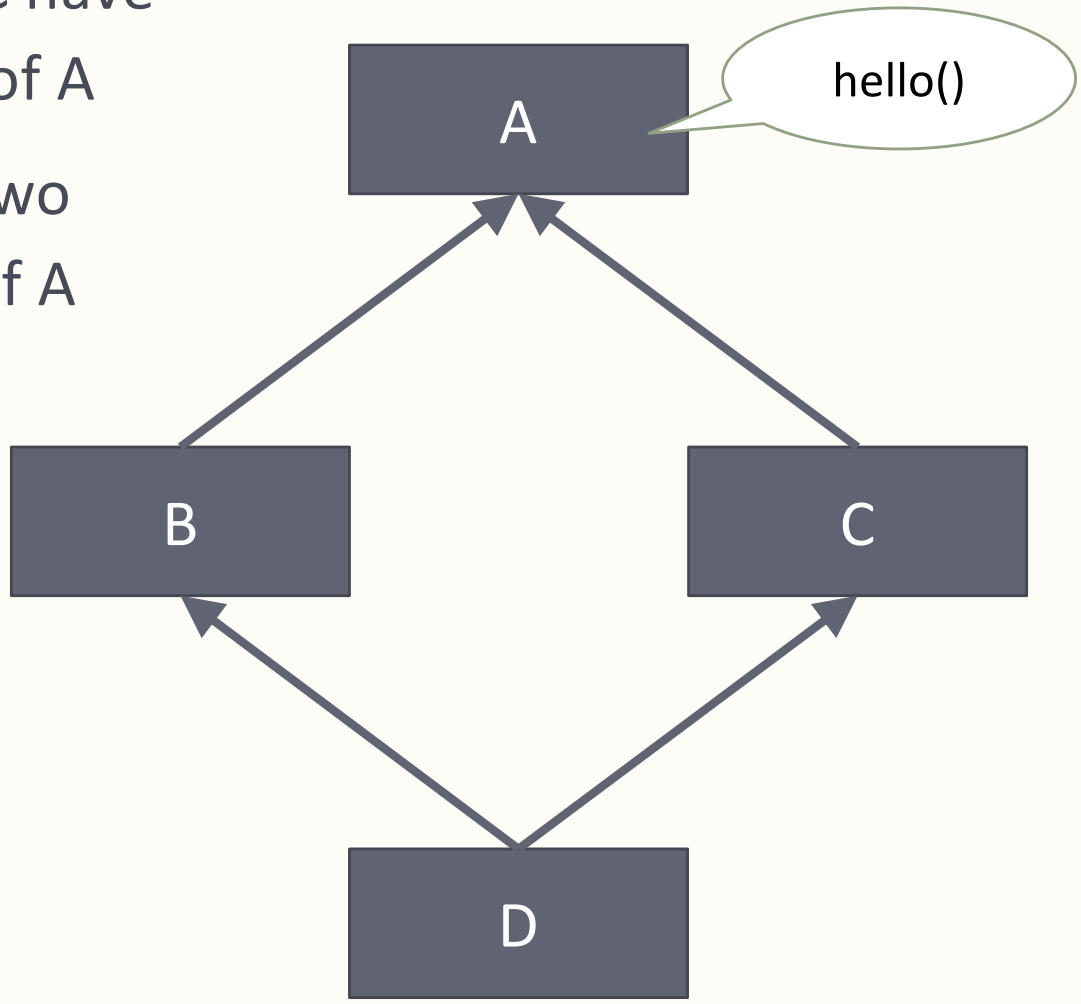
this and super

- In OOP we need pointers to the object itself
  - Usually named “this” or “self”
  - E.g. this.hello()
- Pointer to its superclass
  - Usually named “super”
  - E.g. super.hello()
- Pointers to the constructor of superclass
  - E.g. super()

Diamond problem

- Both B and C have an instance of A

→ so D have two instances of A



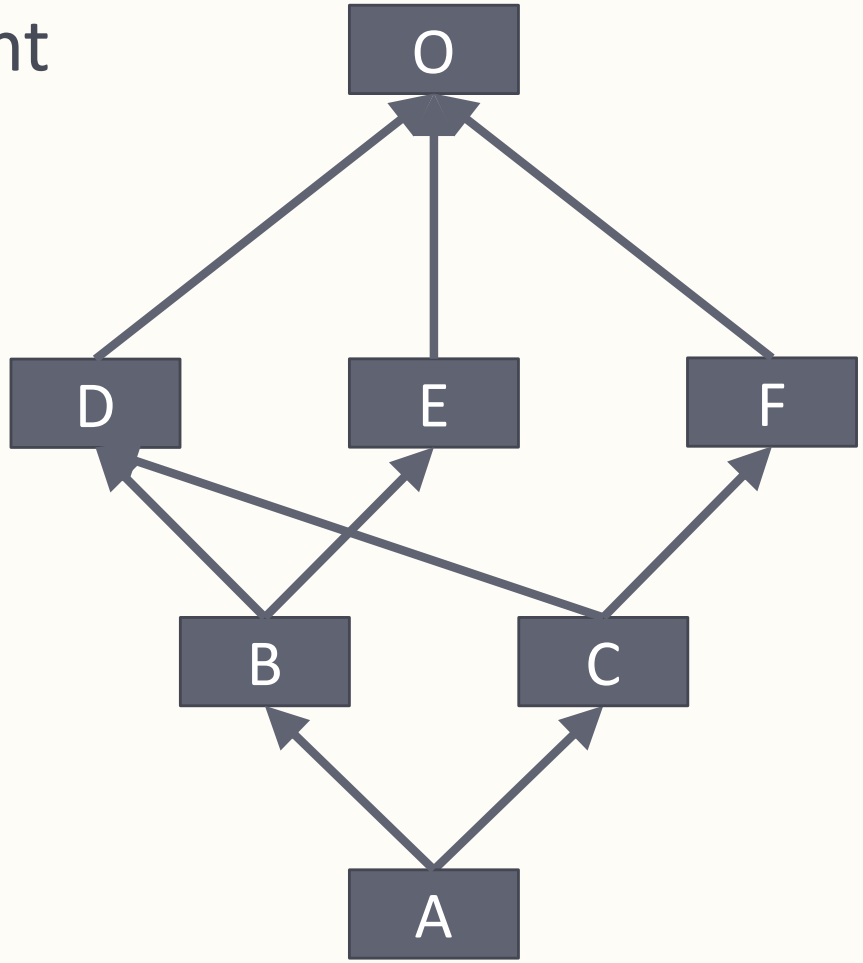
Cause compilation error in C++

```
class A {
public:
    void hello() { cout << "A:hello!\n"; }
};

class B: public A {};

class C: public A {};

class D: public B, public C {};

int main() {
    A* d = new D();
    d->hello();
    return 0;
}
```

```
error: non-static member 'hello' found in
multiple base-class subobjects of type 'A':
  class D -> class B -> class A
  class D -> class C -> class A
    d->hello();
      ^
note: member found by ambiguous name
lookup
    void  hello() { cout << "A:hello!\n";
}
```

Not only diamond problem…

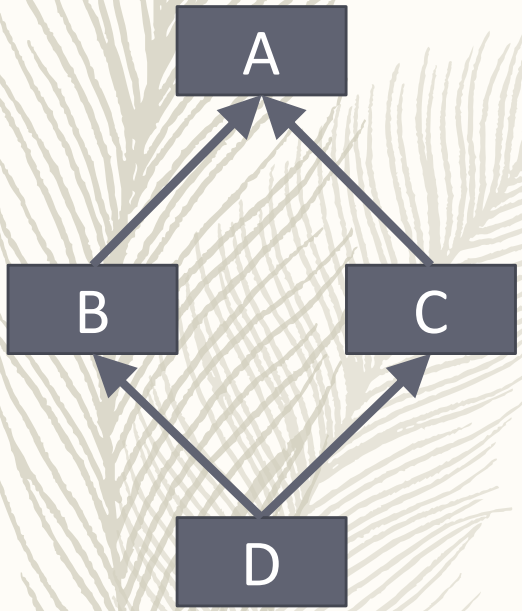
- Multiple inheritance might be very complicated
- Especially when we think about super() call!
  - How to traverse properly?



Reference

- https://www.python.org/download/releases/2.3/mro/

Method resolution order in Python

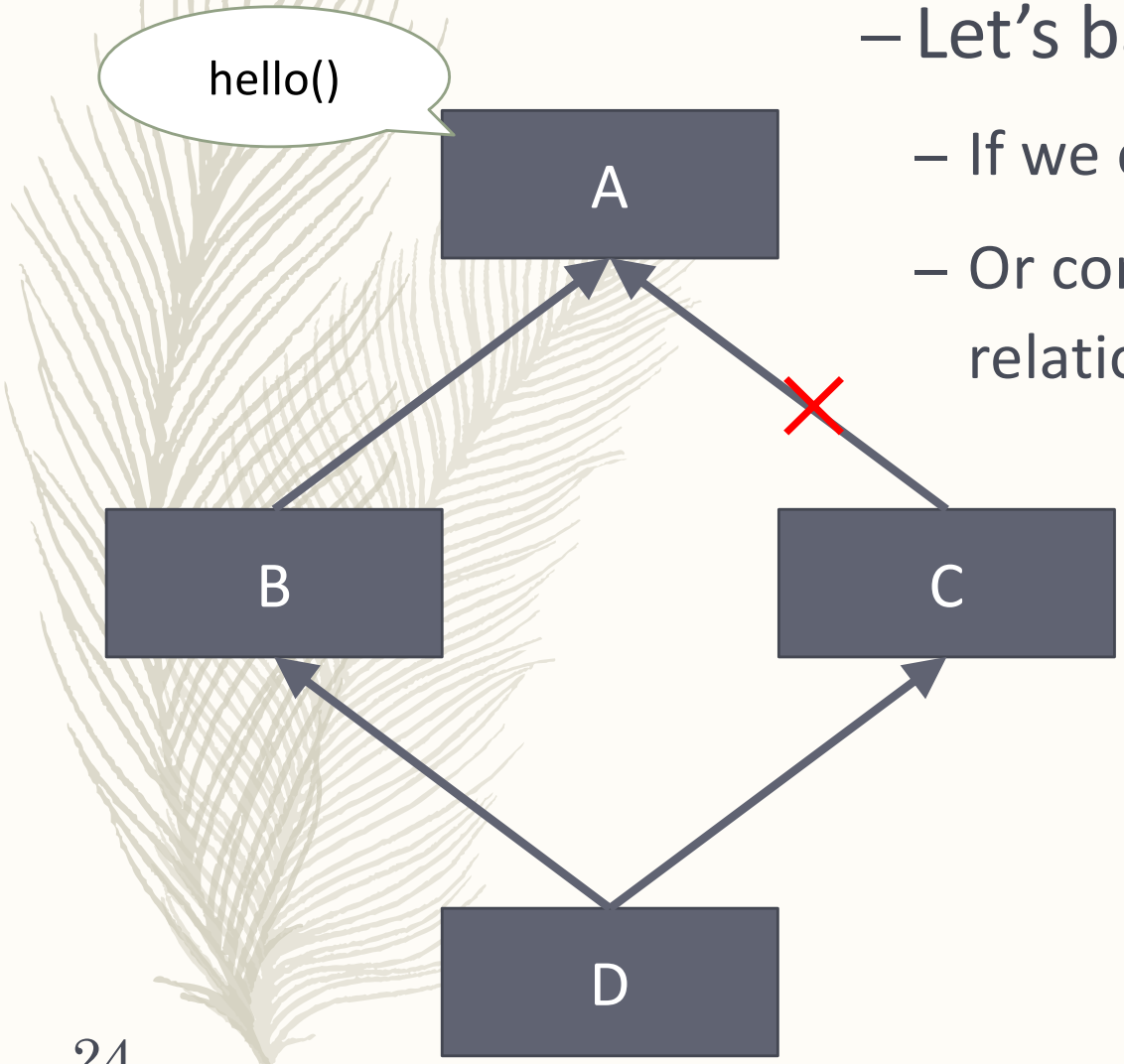


- Method resolution order (MRO)
  - Resolve diamond problem when using multiple inheritance
- Old-style classes
  - Depth-first, left-to-right
  - D, B, A, C, A
- New-style classes (Inherit from object)
  - C3 linearization is adopted in Python 2.3 and newer
  - D, B, C, A, object

Reference

- https://docs.python.org/2/reference/datamodel.html#new-style-and-classic-classes
- https://www.python.org/download/releases/2.3/mro/
- https://www.python.org/doc/newstyle/

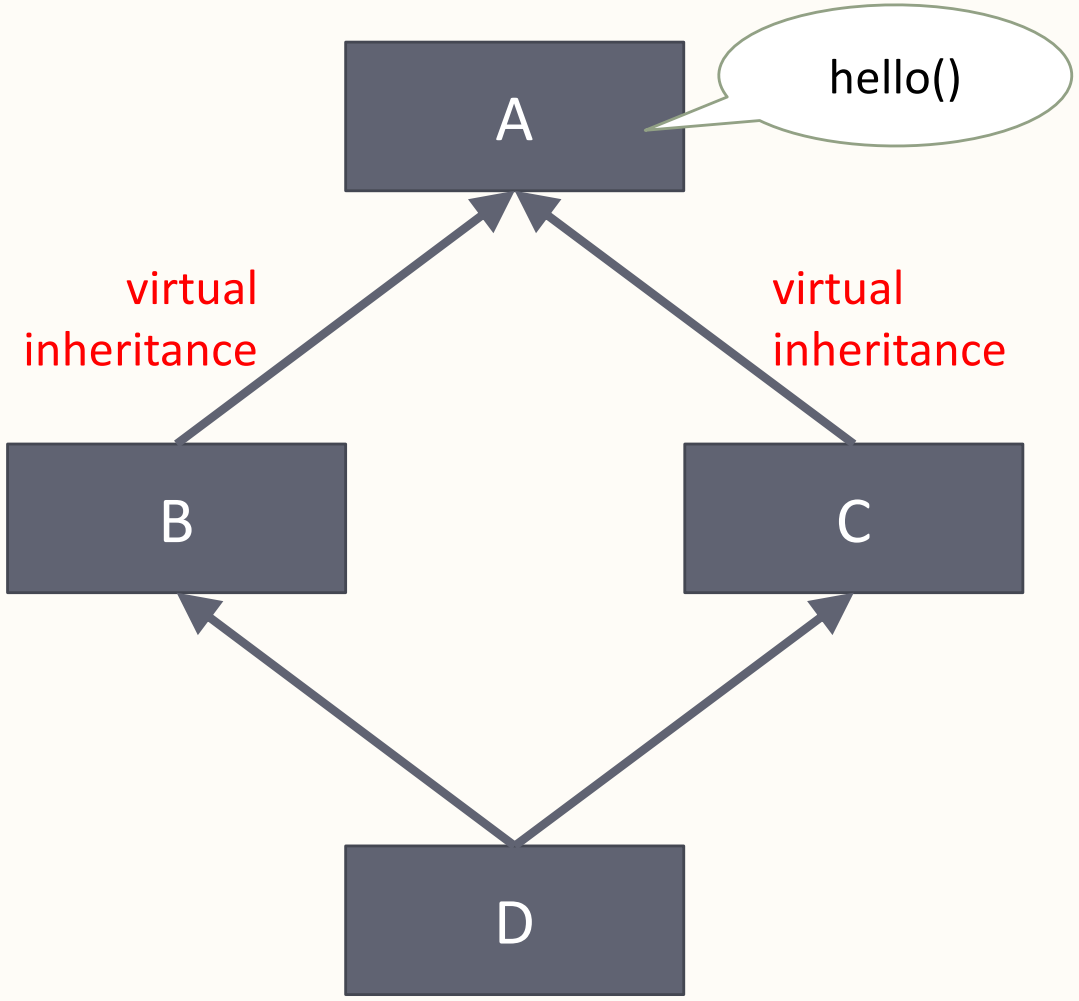
Still allow multiple inheritance?



- Let's back to diamond problem
  - If we can break the relation?
  - Or convert to another type of relation?

Virtual inheritance in C++

- Resolve it by declaring ALL the inheritance relation as virtual
- Note that we must use virtual for B and C, not D

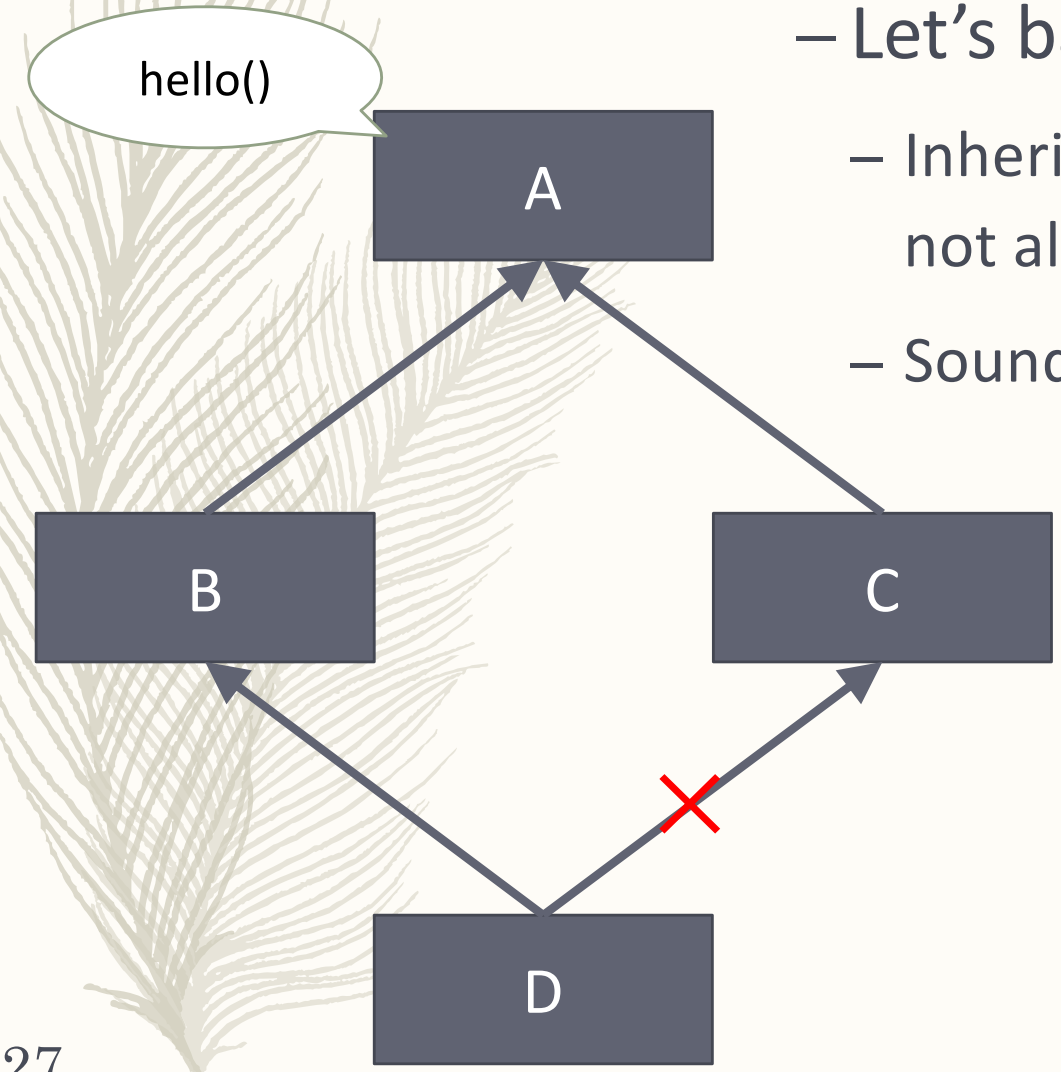


Virtual inheritance in C++ (cont.)

- Only a single instance of A is shared among all virtual inheritance

```
class A {
public:
    void hello() { cout << "A:hello!\n"; }
};

class B: virtual public A {};

class C: virtual public A {};

class D: public B, public C {};

int main() {
    D* d = new D();
    d->hello();
    return 0;
}
```

Single inheritance



- Let's back to diamond problem
  - Inheriting from multiple classes is not allowed
  - Sounds reasonable?

But sometimes we still need…

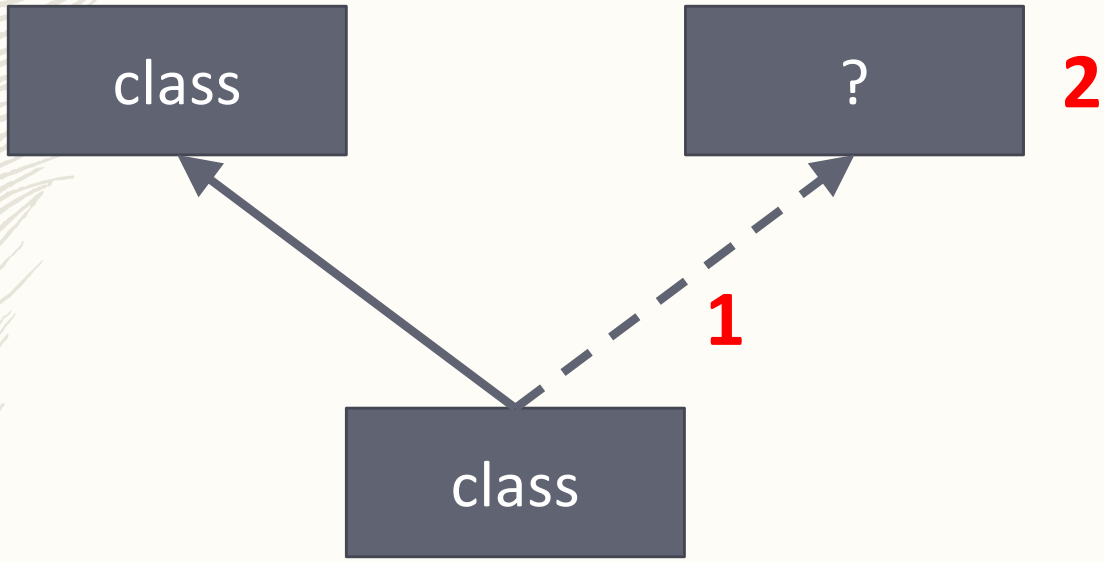
- Some kind of “multiple inheritance”
  - Need to “inherit” behaviors from different things
- To do class composition
  - Compose functionalities from different classes
  - For code reuse
  - To modularize code

Class composition is necessary

- For example,
  - Bat is a Mammal,
    but also has the behavior of WingedAnimal
  - Bird is an Animal, not Mammal,
    but has the behavior of WingedAnimal
  - ImageButton is a Button,
    but has the behavior of Image
  - Smartphone is a Phone,
    but also has the behavior of Computer and Camera

Single inheritance (cont.)

- How to resolve such problems?
  1. Let some links (inheritance) be different
  2. Let some units (classes) be different
     - *Introduce a new thing that is different from class*

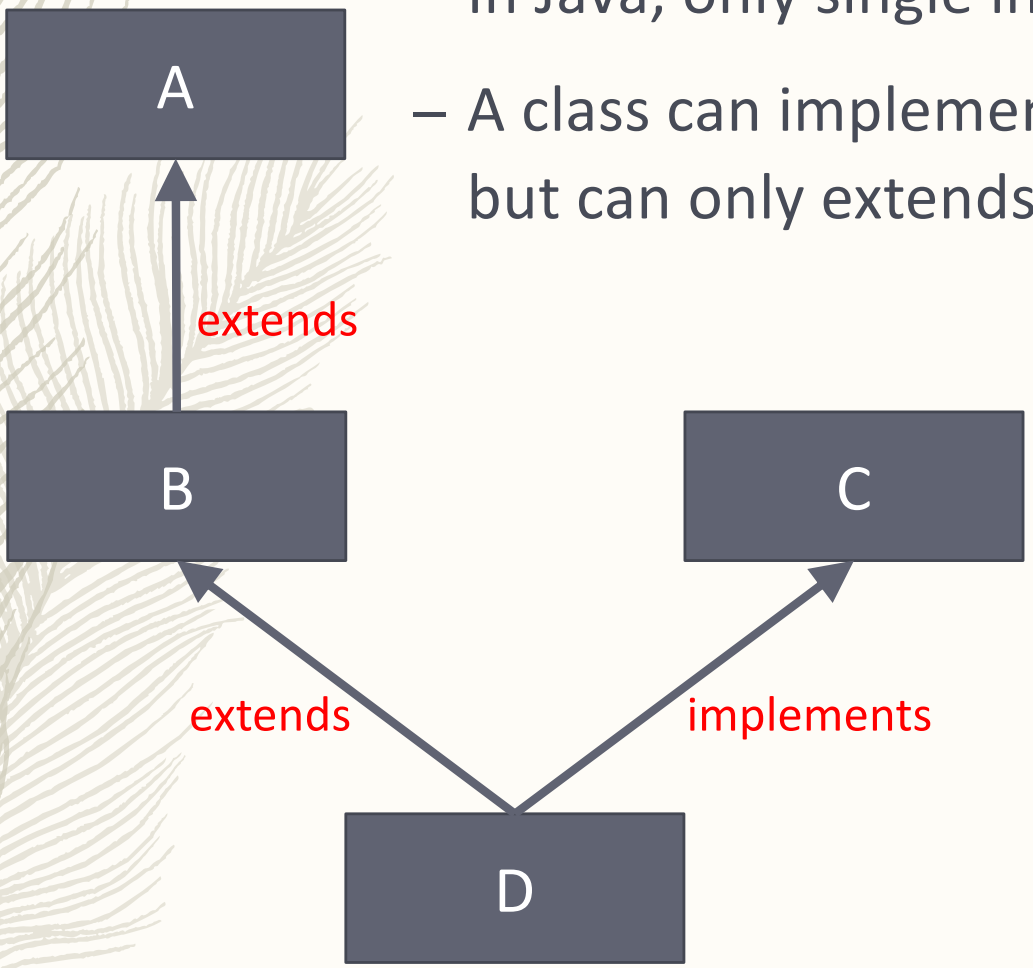


Interface: a very restricted form

- An interface can only define how a behavior should be
  - Only method declaration: no implementation is given
  - Only behavior: no state is held
- An interface can be extended into another interface
  - It will include all methods in it
- A class implements it must implement all methods in it
  - If Bat implements WingedAnimal, we must give the implementation of fly()

Interface in Java

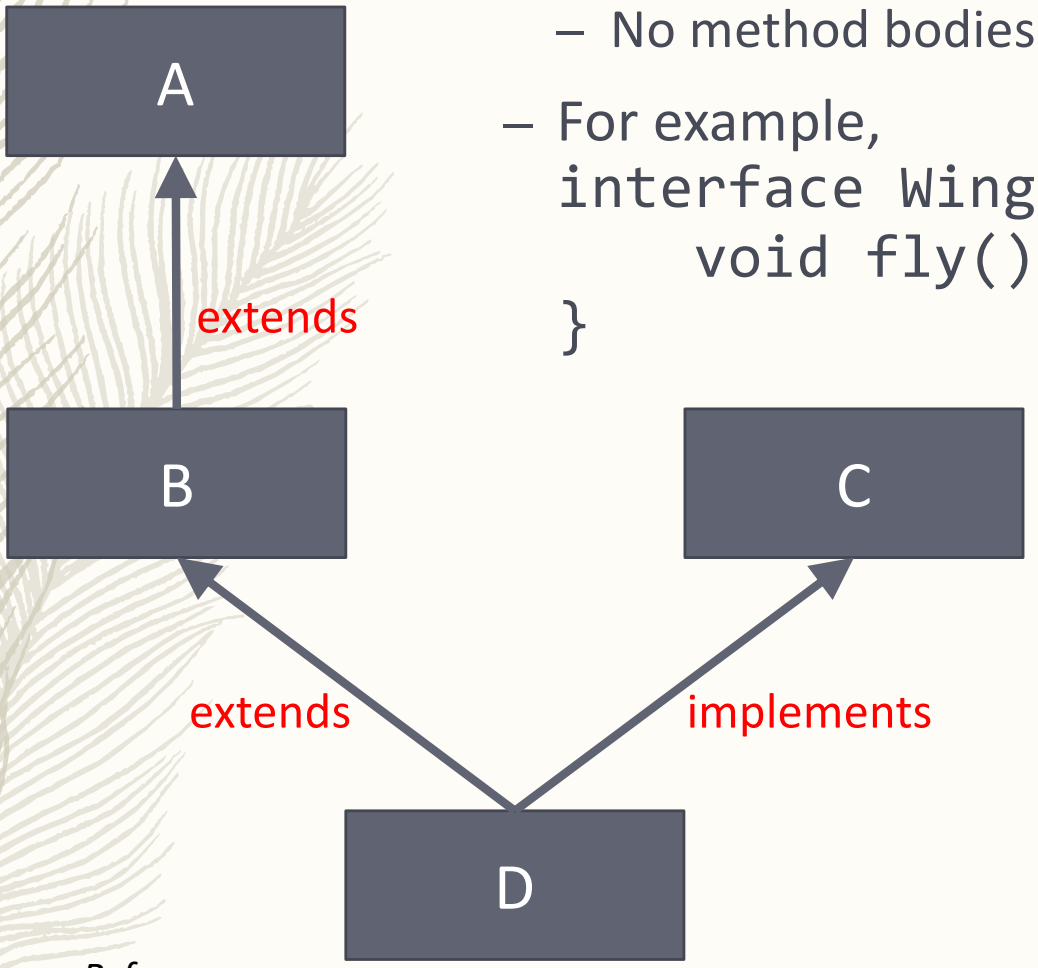
- In Java, only single inheritance is allowed
- A class can implement a lot of interfaces, but can only extends a class



Interface in Java (cont.)

- Interface is a set of method signature
  - No method bodies
- For example,

```
interface WingedAnimal {
    void fly();
}
```



Reference

- https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

Another approach: Mixin

- Can be mixed in to provide its functionality

- Proposed in Flavors
  - An extension to Lisp

- The naming is inspired by a ice cream store
  - Mix ice cream with candy, cookies, etc.

Reference

- *Howard I. Cannon. Flavors: A non-hierarchical approach to object-oriented programming, 1982, Symbolics Inc.*

Different from a class

- A mixin is intended to be mixed in with others
  - Not ”is-a” relation
  - Rarely used as a standalone object
- Have method implementations and might hold states
  - A restricted form of multiple inheritance
- If two mixins contain methods with the same name
  - Resolve by mixing order

Implicitly resolve by ordering

- If we say (M1, M2) are mixed in, and both of them have m()
  - M1::m() will be selected
  - If (M2, M1) are mixed in
    - *M2::m() will be selected*
- Examples in Python
  - `class ThreadingUDPServer(ThreadingMixIn, UDPServer):`
    - *mix-in class comes first, since it overrides a method defined in UDPServer*

Reference

- https://docs.python.org/2/library/socketserver.html

Yet another approach: Trait

- A trait is a set of methods that implement behavior
- A restricted form of mixin
  - Can be regarded as a design between interface and mixin
- You might see the name in different proposals…
  - First introduced in Self and also proposed in some research
    - *Their semantics are more similar to the mixins than traits we discussed here*

Reference

- *Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable Units of Behavior. In ECOOP'03, 2003.*

Different from a mixin

- A trait has full method implementation but cannot hold state
  - Do not specify any state variables
  - Never directly access state variables
- Trait composition is symmetric and flattened
- If two traits contain methods with the same name
  - Programmers must explicitly resolve it

Traits in Scala

- The Bat example in Scala using traits
  - Traits can be “mixed in” statically or dynamically

```
class Animal {
  def hello() = println("hello!")
}
class Mammal extends Animal {
  override def hello() = println("Mammal::hello!")
}
trait WingedAnimal extends Animal {
  def fly() = println("fly")
}
// statically mix in Mammal with WingedAnimal
class Bat extends Mammal with WingedAnimal {
}
// dynamically mix in Mammal with WingedAnimal
val bat = new Mammal() with WingedAnimal
```

Traits in Scala (cont.)

- Traits in Scala can be used to do mixin composition
  - Confused about the terms?
    - *Mixins are usually used to describe behavioral composition*
    - *Traits can be regarded as a restricted form of mixins*

- A trait can specify a member variable without assigning value
  - Then a class integrating it must assign it

```
trait WingedAnimal extends Animal {
  val name: String
  override def hello() = println(name + ": hello!")
  def fly() = println("fly")
}
class Bat(val name: String) extends Animal with WingedAnimal {
}
```

Traits in Scala (cont.)

- A trait can use a member variable in the class it extends
  - The state is owned by the class it extends
- Here “extends” means
  - To use the members in that class
  - A class mixed in with the trait must be that class or its subclass
  - Not ”is-a” relation!

```
class Animal(var name: String) {
  def hello() = println("hello!")
}
trait WingedAnimal extends Animal {
  override def hello() {
    name = "WingedAnimal"
    println(name + ": hello!")
  }
}
```

Traits in Scala (cont.)

- A class can “extends” a trait

```
trait Animal {
  def hello() = println("hello!")
}
class Bat extends Animal {
}
```

- or “extends” a class “with” many traits

```
class Animal {
  def hello() = println("hello!")
}
trait Winged {
  def fly() = println("fly")
}
trait Swimming {
  def swim() = println("swim")
}
class Duck extends Animal with Winged with Swimming {
}
```

The difference between them

- Interface
  - only method signatures
- Trait
  - can contain method implementations
- Mixin
  - can contain states
- Class
  - involve in inheritance

The difference between them (cont.)

- Both mixin and trait are used for behavioral composition
  - Non-inheritance composition
  - How to resolve method conflicts and ordering
  - Trait
    - *needs explicitly resolving conflicts*
    - *flattened*
  - Mixin
    - *implicitly resolves conflicts*
    - *might be nested (depend on the design)*

Default methods in Java 8

- However, the interface in Java 8
  - May have method implementations!
- Default methods
  - Give a default implementation for certain methods
- Why?
  - Avoid rewriting all existing classes that implement an interface when we add a new thing to that interface

An example of using default methods

- Suppose we want to add a method “forEach” to the interface “Iterable”
  - Without default methods, all classes implementing Iterable have to be rewritten!
- If we define the implementation of forEach and declare as “default”
  - Any existing class that implements Iterable can benefit from

```
public interface Iterable {
    Iterator iterator();
    default void forEach(Consumer action) {
        for (T t : this) {
            action.accept(t);
        }
    }
}
```

Confuse with abstract class?

- Abstract class may or may not include abstract methods
  - Cannot be instantiated
- Abstract method
  - Only method signature (without method body)
- Different from the interface in Java 8??
  - Have states and visibility control
  - And, of course, involved in inheritance

Reference

- https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

Which one should I use?

- Now we have class, abstract class, mixin, trait, interface
  - they are all kinds of units
  - to do composition for code reuse
  - → You should carefully think about the usage and meaning
- Anyway, use “the real” multiple inheritance might not be a good idea
  - Simpler is better!
  - Complicated and might be surprising, unexpected
  - Use behavioral composition instead

Which one should I use? (cont.)

- Is it a "is-a" relation? → class
  - Not reasonable to have concrete implementations or objects? → abstract class
  - Just a behavior across units?
  - Can it be described without details, like specs? → interface
  - Need concrete behavioral definition?
    - *Need to hold states?* → *mixins*
    - *Explicit resolving and flattened relation is preferred?* → *traits*

Class extension

- All things you see until now can be regarded as
  - A way to reuse your code in a modular way
  - Include inheritance or mixin (in a broader meaning)
- A kind of class extension
  - Extend a class to support certain functionalities
- However, they are “static”, “non-destructive” in some senses

Class extension (cont.)

- When we need to extend a class
  - We may extend it by a subclass
  - We may let it implement interfaces
  - We may mix in it with mixins or traits
- Basically we need subclasses to change the behavior of a class
  - In software development

How to add a new method?

- For example, is we are using a library Time
  - How to add a method such as oneHourAgo()?
  - If we have source file, we can modify and recompile
  - But obviously is not a good way
    - *Not compatible with others*
    - *Should not be applied to all*
    - *Such modification/patch should be separated*

:

How to add a new method? (cont.)

– Otherwise, if the library allows us to subclass it, we can create MyTime

– It might look like (in pseudo-code):

```
#include <Time.h>

class MyTime extends Time
    def oneHourAgo
        self - 3600
    end
end

now = MyTime.new
puts now.oneHourAgo
```

How to add a new method? (cont.)

- However, we can only use MyTime objects
  - oneHourAgo method is added to MyTime rather than Time
  - Existing usage of Time cannot benefit from this modification

```
#include <Time.h>

class MyTime extends Time
    def oneHourAgo
        self - 3600
    end
end

now = MyTime.new
puts now.oneHourAgo
```

Open class in Ruby

- Classes in Ruby are open!
  - Never be closed
- Usually used for patch
  - Fix a bug in a third-party library without directly modifying it
  - Monkey patching

Open class in Ruby (cont.)

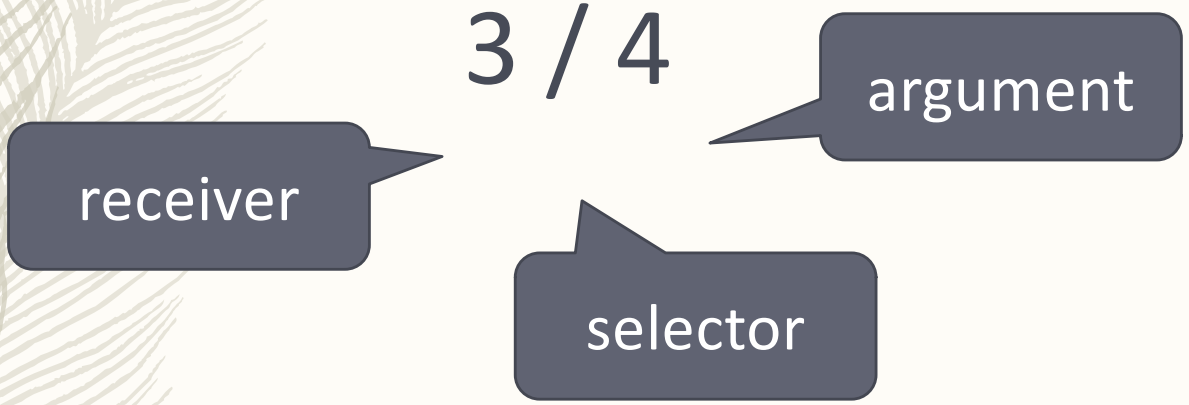
- we can simply add such as method
  - It become valid…
  - Not subclass!

```
class Time
    def oneHourAgo
        self - 3600
    end
end

now = Time.new
puts now.oneHourAgo
```

Another example of open class

- We might do dangerous things…
  - Modify the default behavior of standard library

- Recall: sending a message in Smalltalk



Another example of open class

In Ruby

3 / 4

actually means

3./(4)

where 3 and 4 are Fixnum

```
puts 3 / 4       # => 0

class Fixnum
    def /(n)
        self.to_f / n.to_f
    end
end

puts 3 / 4       # => 0.75
```

Refinements in Ruby

- The scope of open class is global
  - The patch is visible to all clients of the patched class
  - Some modules might work unexpectedly!
- Refinements is proposed to reduce the impact
  - Experimental feature in Ruby 2.0
  - Expected to exist in future versions

Refinements in Ruby 2.0

```
class C
end

module M
  refine C do
    def foo
      puts "C#foo in M"
    end
  end
end

def call_foo(x)
  x.foo
end

using M

x = C.new
x.foo       # prints "C#foo in M"
call_foo(x) # => raises NoMethodError
```

- "Lexical scope"
- Deactivated when control is transferred outside
  - Require or load a file
  - Call a method defined outside

Reference

- https://ruby-doc.org/core-2.1.1/doc/syntax/refinements_rdoc.html

Refinement in Ruby 3.0

```
module StrUtils
  def indent(level)
    ' ' * level + self
  end
end

module M
  refine String do
    import_methods StrUtils
  end
end

using M
"foo".indent(3)
#=> "   foo"

module M
  refine String do
    import_methods Enumerable
    # Can't import method which is not
    # defined with Ruby code: Enumerable#drop
  end
end
```

- Refinement is a class of the self (current context) inside refine statement
  - It allows to import methods from other module
- import_methods
  - copies methods and adds them into the refinement, so the refinement is activated in the imported methods

  → *only methods defined in Ruby code can be imported*

Reference

- https://ruby-doc.org/core-3.1.0/Refinement.html

Next lecture

11/25 **Practice 5**

12/2 Delegation, Polymorphism, and Multimethods