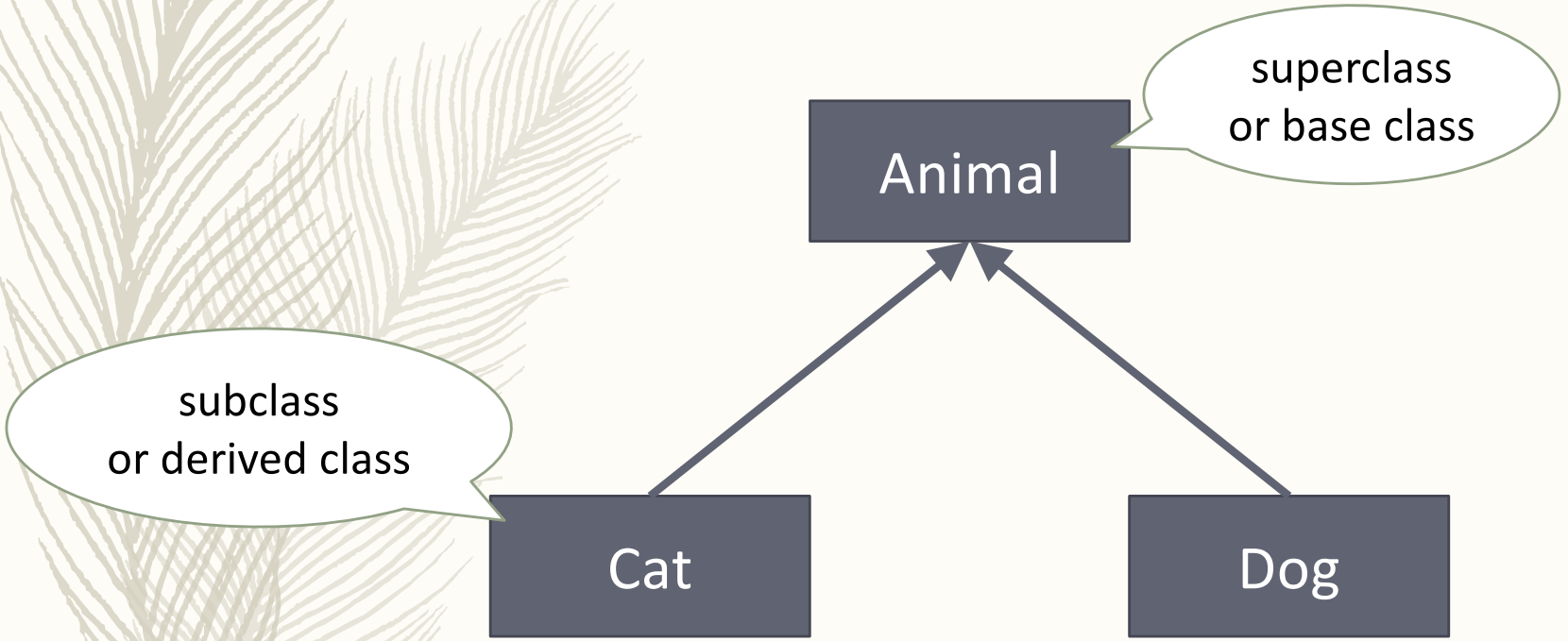
Programming Language Design

12/2 Delegation, Polymorphism, and Multimethods

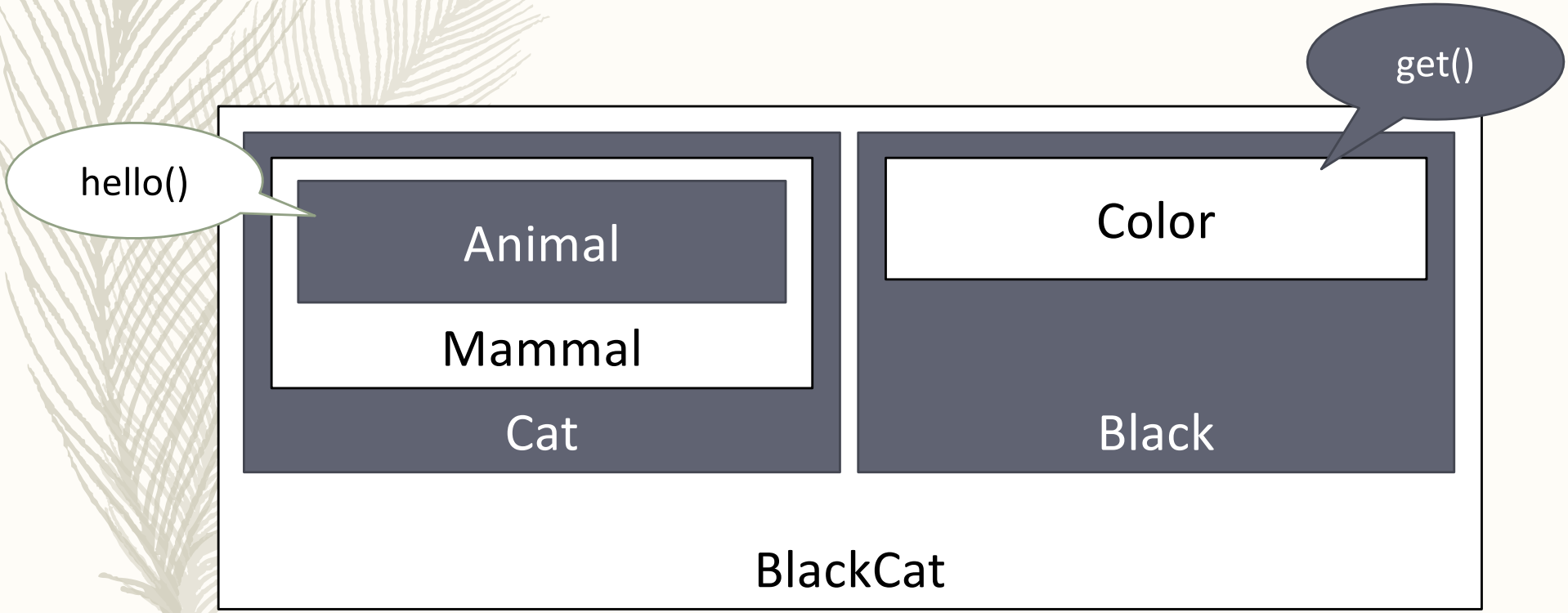
Recall: Inheritance

– A-kind-of; is-a relation



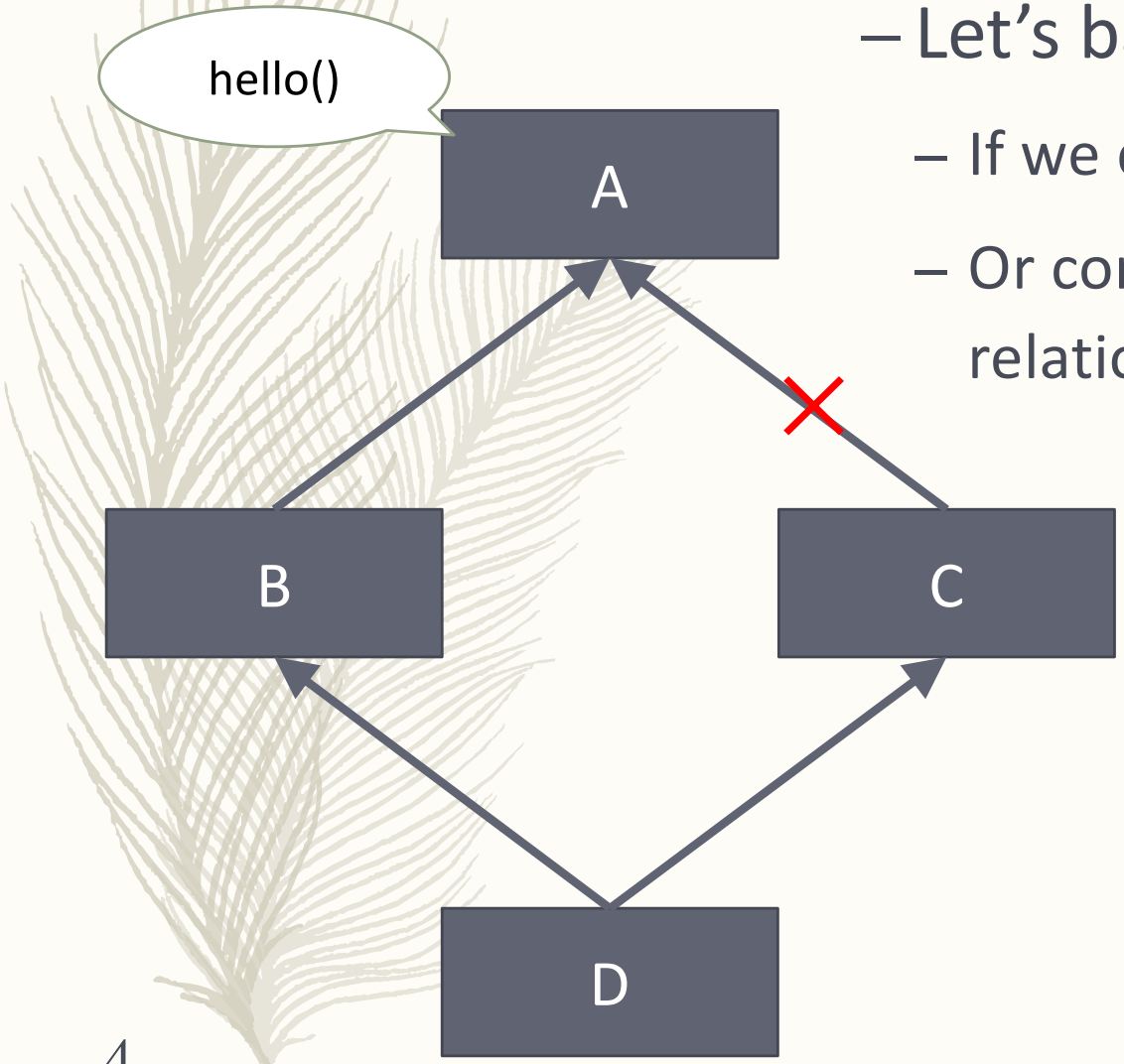
Recall: Multiple inheritance (cont.)

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



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

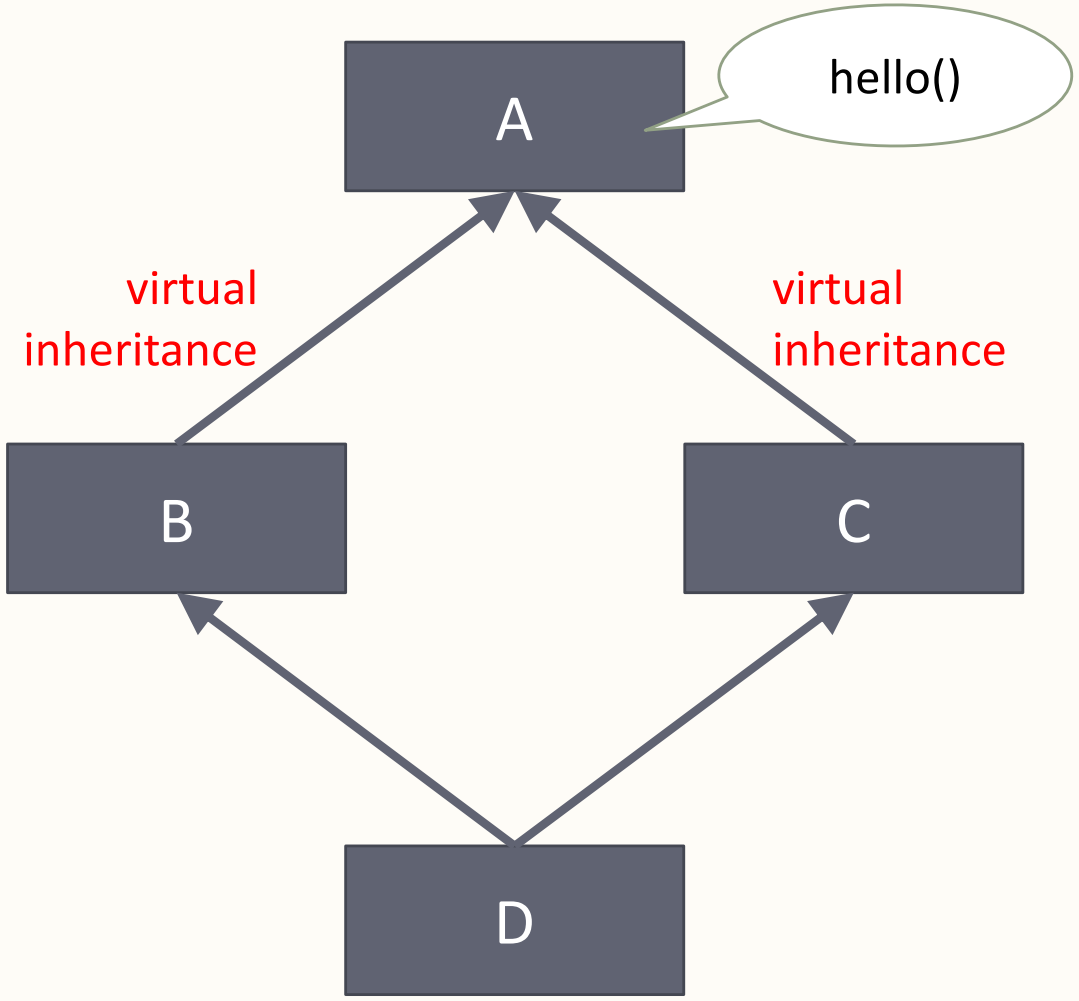
Recall: Still allow multiple inheritance?



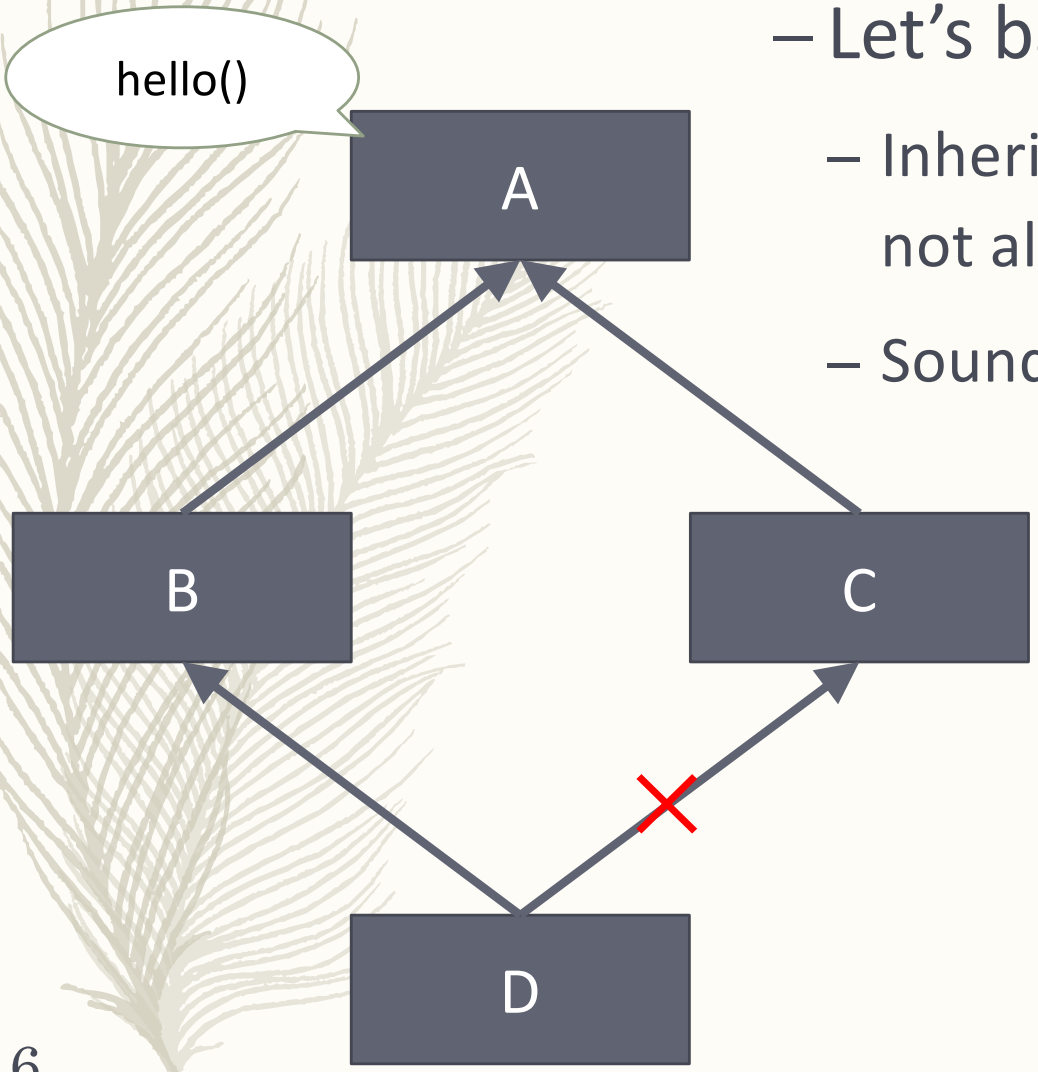
- Let's back to diamond problem
  - If we can break the relation?
  - Or convert to another type of relation?

Recall: 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



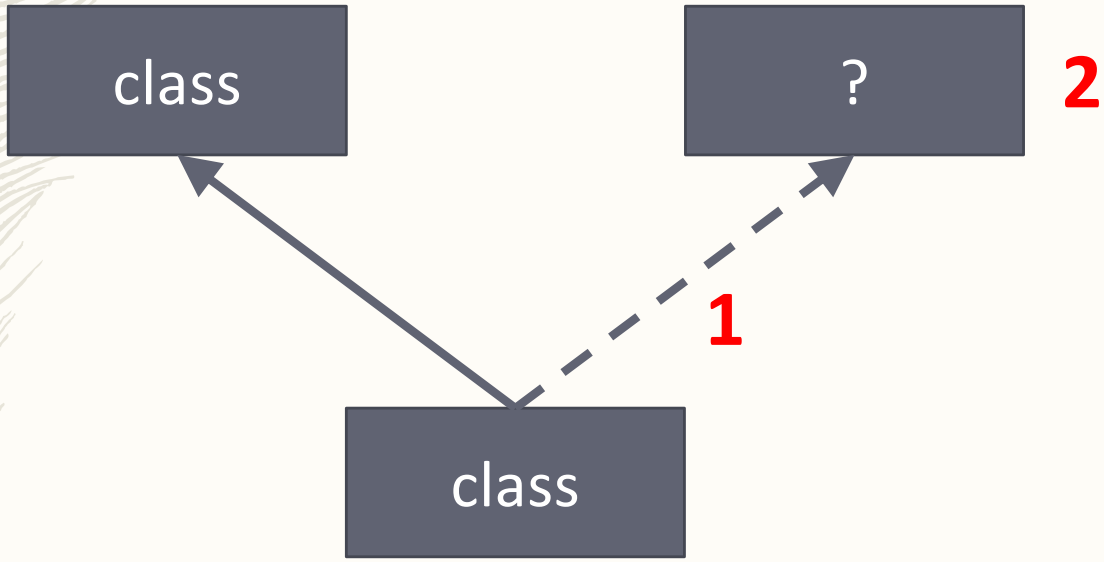
Recall: Single inheritance



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

Recall: 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*



Recall: The difference between them

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

Classes are always there!

- You might notice that in the given example
  - A superclass is used along with mixins
  - Even though mixins can also have states??
    - *can be mixed in to create a new object*
- Why we still mix in a class with a mixin?
  - Not just mix in a mixin with a mixin??

Do we really need classes?

- The examples we saw are written in class-based OO languages
  - Classes are their way to create objects
- Of course mixins can be used without classes
  - Unlike traits, they can hold states
- Language support
  - Different languages might support them in a various way
  - But the basic ideas are the same

Another style of OOP

- Write object-oriented programs with classes
  - Class-based OOP
  - E.g. Smalltalk, C++, Java, C#, Python
- Without classes?!
  - Prototype-based OOP
- Introduced by Self
  - Followed by Common Lisp, JavaScript, etc.

Reference

- David Ungar and Randall B. Smith. Self: The Power of Simplicity. In OOPSLA '87, 1987.
- *Randall B. Smith and David Ungar. Programming as an Experience: The Inspiration for Self. In ECOOP'95, 1995.*

The Self language

- Simplicity
  - Based on prototypes, slots, and behavior
  - Dynamically typed as Smalltalk
- Simpler relationships
  - Class-based
    - *is-a, instance of*
    - *kind-of, subclass of*
  - Self
    - *inherits from*

Everything is an object

- In Smalltalk
  - Everything is an object
    - *Containing a pointer to its class, which describes its format and holds its behavior*
- In Self
  - Everything is also an object, but
    - *No such a class pointer used in Smalltalk*

How it works without class?

- Message sending
  - Class-based
    - Search the method for that message
    - If not found
      - continues via the class pointer
  - Self
    - Search the slot for that message
    - If not found
      - continues via a parent pointer

How it works without class? (cont.)

- Object creation
  - Class-based
    - *Executing a "plan"*
  - Self
    - *Cloning (copying) an example*
    - *No need to create a class even for only one object instance*
    - *No metaclass, no metametaclass, …*

Named slots

- A Self object contains named slots
  - Which may store either state or behavior
  - No distinction between accessing a variable and sending a message

Named slots (cont.)

- No variables?!
  - To access “x” variable on an object
    - *Send a message “x” to it*
    - *Find the “x” slot*
    - *Evaluate the object found therein*
      - The result is the number since the “x” slot contains a number
  - To change the contents of the “x”
    - *Send a message ”x:” with a number as the argument*
    - *The slot named “x:” containing the assignment primitive*

Delegation

- Self replaces inheritance with delegation
  - A message to an object can be delegated to another object
  - Allow runtime function invocation on a specific object instance
  - JavaScript follows these concepts

Object creation in JavaScript

– Create a prototype

```
function Animal(name) {
    this.name = name;
}
```

– Create an object based on the prototype

```
var x = new Animal("x");
```

Add field or method

- Add a field to an object

```
x.hello = "hello";
```

- Add a method to an object

```
x.hello = function() {
    print(name + ": hello");
};
```

Add to the prototype

– Add a field to a prototype

```
Animal.prototype.weight = 10;
```

– Add a method to a prototype

```
Animal.prototype.weight = function() {
    return 10;
};
```

Function object

– A method on an object is actually a function object

```
function Animal(name) {
    this.name = name;
    this.hello = function() {
        print(name + ": hello");
    };
}
var x = new Animal("x");
```

Modify prototype affects its objects

– Since the message “weight” cannot be found on x and then passed to its prototype!

```
function Animal(name) {
}
var x = new Animal("x");
print(x.weight);      // undefined

Animal.prototype.weight = 10;
var y = new Animal("y");
print(y.weight);      // 10

print(x.weight);      // 10
```

Call a function object

```
function Animal(name) {
    this.name = name;
    this.hello = function() {
        print(name + ": hello");
    };
}
function Bird() {
    this.fly = function(where) {
        print(this.name + " fly " + where);
    };
}
var x = new Animal("x");
var y = new Bird("y");
y.fly.call(x, "in the sky");    // "x fly in the sky"
```

- "call" can be used to invoke a method with "this"!

Delegate a method call

```
function Animal(name) {
    this.name = name;
    this.hello = function() {
        print(name + ": hello");
    };
}
function Bird() {
    this.fly = function(where) {
        print(this.name + " fly " + where);
    };
}
var x = new Animal("x");
var y = new Bird("y");
x.hello = y.fly;
x.hello("in the sky");    // "x fly in the sky"
```

- It is also possible to delegate a method call to another method

What is polymorphism?

- polymorphous (also polymorphic)
  - *"having many forms, styles, etc during the stages of development"*

    --- Longman Dictionary of Contemporary English, 5th edition

  - *"having or passing through many stages of development"*

    --- Oxford Advanced Learner's Dictionary, 9th edition

Ad-hoc polymorphism

- An example: the polymorphic + operator
  - When calling + on two operands, dispatch to concrete implementation
    - *Based on the types of the operands*

```
1 + 2 = 3
1.0 + 2.0 = 3.0
"1" + "2" = "12"
```

- Also known as overloading

Parametric polymorphism

- An example: drop the first element in a List
  - When calling drop on a List, the precise input type never comes into play
    - *Acts uniformly on a range of types*

```
def drop[A](l: List[A]): List[A] = l.tail

val result1 = drop(List(1, 2, 3))
val result2 = drop(List("one", "two", "three"))
```

- Also known as generics

Ad-hoc vs. Parametric

- Ad-hoc polymorphism
  - Associate a single function symbol with a set of implementation
  - Apparent, synthetic
- Parametric polymorphism
  - A single generic function which works for a variety of types
  - “true” polymorphism

Subtyping polymorphism

- If S is a subtype of T
  - S <: T
  - Any term of type S can be safely used in a context where a term of type T is expected
- S >: T pronounced “S is a supertype of T”
- Class-based OOP provides subtype polymorphism in the form of subclassing

Subtype? Subclass?

"We are using the words 'subtype' and 'supertype' here to emphasize that now we are talking about a semantic distinction."

"By contrast, 'subclass' and 'superclass' are simply linguistic concepts in programming languages that allow programs to be built in a particular way. They can be used to implement subtypes, but also, as mentioned above, in other ways."

Reference

- *Barbara Liskov. Keynote Address - Data Abstraction and Hierarchy. In OOPSLA'87, 1987.*

Nominal typing

- S is a subtype of T if and only if it is declared
- Declarative
  - Explicitly express design intention

- S is not a subtype of T if it is not declared
  - Even if S can do anything T can do
- The one you see in class-based OO languages

Reference

- *Donna Malayeri and Jonathan Aldrich. Integrating Nominal and Structural Subtyping. In ECOOP'08, 2008.*

Check it according to declaration

```
class Duck {
public:
    std::string quack() { return "Quack!"; }
};

class Person {
public:
    std::string quack() { return "I am not a duck..."; }
};

void callQuack(Duck* duck) {
    std::cout << duck->quack() << std::endl;
}

int main() {
    callQuack(new Duck());
    callQuack(new Person());    // compilation error!!
}
```

- *candidate function not viable: no known conversion from 'Person *' to 'Duck *' for 1st argument*

Duck typing

– “In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with.”

– A style of type checking, done at runtime

– In JavaScript, Python, Ruby

Reference

- https://groups.google.com/forum/?hl=en#!msg/comp.lang.python/CCs2oJdyuzc/NYjla5HKMOIJ

Check only the behaviors

```
class Duck:
  def quack(self):
    return 'Quack!'

class Person:
  def quack(self):
    return 'I am not a duck...'

def callQuack(duck):
    print(duck.quack())

callQuack(Duck())       // Quack!
callQuack(Person())     // I am not a duck...
```

- It works since the Person object has quack method!

Structural typing

- S is a subtype of T if its methods and fields are a superset of T's methods and fields
- Intrinsic
  - Express type requirements without defining subtyping hierarchy
- "Type-safe duck typing"
- E.g. Scala

Reference

- *Donna Malayeri and Jonathan Aldrich. Integrating Nominal and Structural Subtyping. In ECOOP'08, 2008.*

Check if it works as promised

```
class Duck {
  def quack = "Quack!"
}

class Person {
  def quack = "I am not a duck..."
}

def callQuack(duck: {def quack: String}) {
    println(duck.quack)
}

callQuack(new Duck)
callQuack(new Person)
```

– In the case of duck typing, the message might be sent to an object that does not have such a method!

Liskov substitution principle

"What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T."

Reference

- *Barbara Liskov. Keynote Address - Data Abstraction and Hierarchy. In OOPSLA'87, 1987.*

A classic example of violation

- A Square is a Rectangle?
  - Let's say we want to derive Square from Rectangle…
  - But a Square is not always substitutable for a Rectangle!

```
void test(Rectangle r) {
  r.setWidth(16);
  r.setHeight(10);
  assert r.getWidth() == 16;
  assert r.getHeight() == 10;
}
test(new Square());    // fail!
```

The type of a function

- Think about the subtype of a variable is easy
  - As to the subtype of a function?

- First we think about what the type of a function is
- If it takes S1 and return S2, for example

in Scala

```
def foo(p: S1): S2 = { ... }
```

in Java

```
S2 foo(S1 p) { ... }
```

we say its type is "S1 ⟶ S2"

The subtype of a function

- Think about the subtype of a function

  - To know when it is safe to use a function of one type in a context where a different function type is expected
    - i.e. when we can safely substitute a function with another function

Contravariance

- What is the relation between the types of functions?
- Suppose Cat <: Animal
  - A function foo takes a Cat and returns an Integer
  - A function bar takes an Animal and returns an Integer
- the type of foo >: the type of bar
  - Contravariant

Covariance

- What is the relation between the types of functions?
- Suppose Cat <: Animal
- A function foo takes an Integer and returns an Animal
- A function bar takes an Integer and returns a Cat

- the type of foo <: the type of bar
  - Covariant

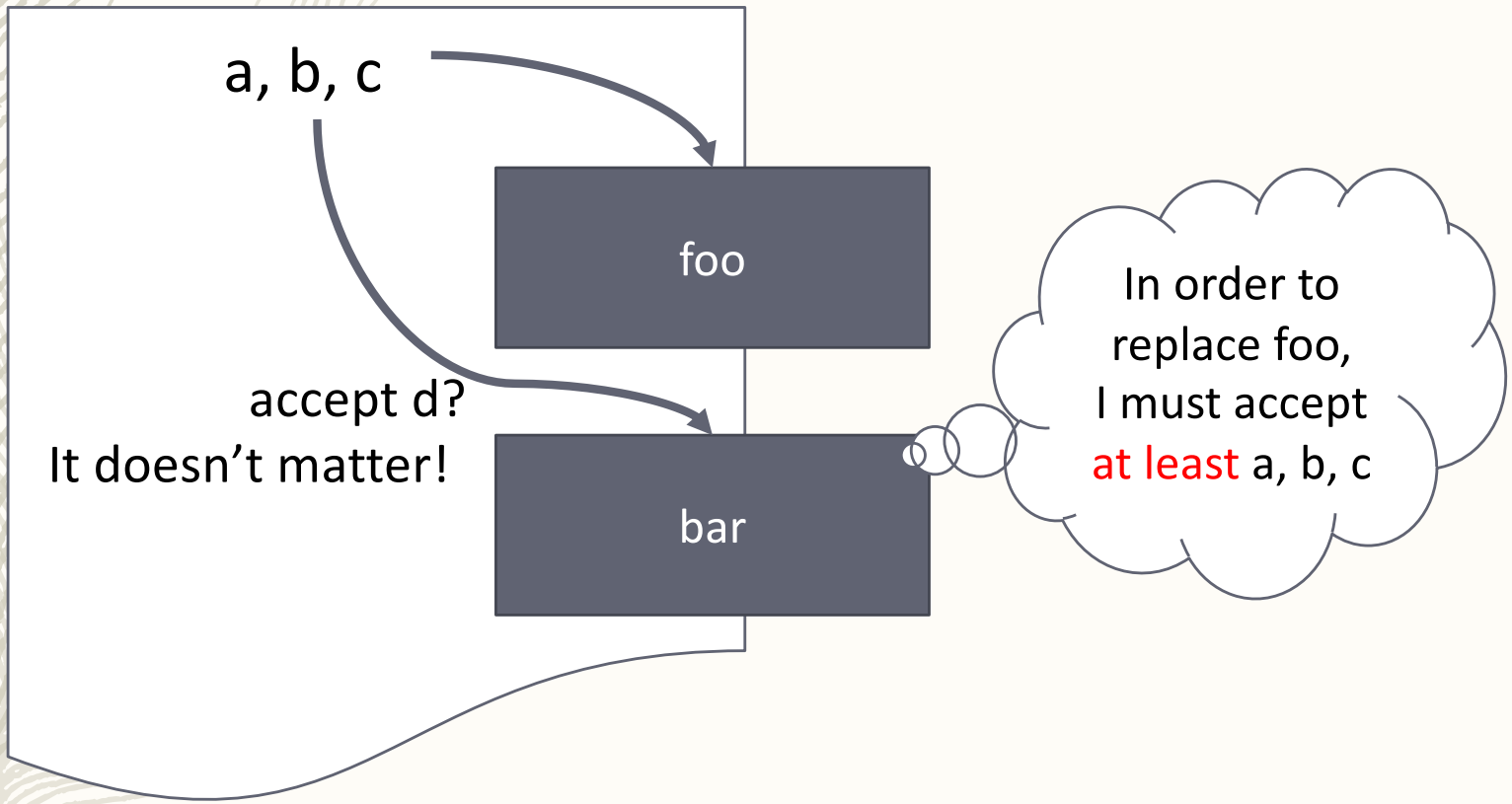
The subtype of a function (cont.)

- Suppose we have a function of type S1 ⟶ S2
  - And we know T1 >: S1 and T2 <: S2
- Then any functions of type T1 ⟶ T2 can also be viewed as having type S1 ⟶ S2
  - T1 ⟶ T2 <: S1 ⟶ S2
- Suppose we have a function of type S1 ⟶ S2, named foo and a function of type T1 ⟶ T2, named bar…
  - When can we replace foo with bar??

Reference
Benjamin C. Pierce. Types and Programming Languages, The MIT Press.

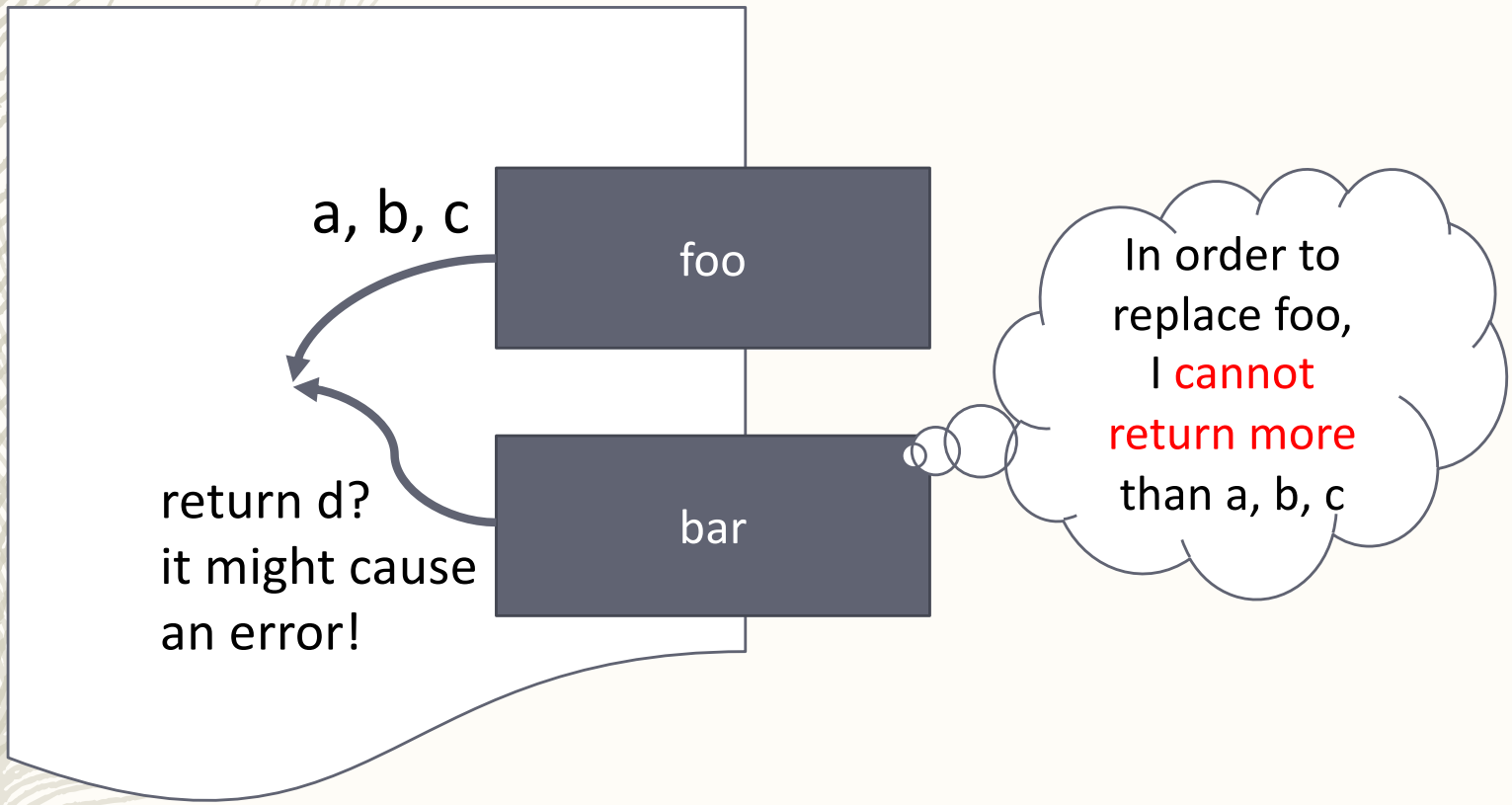
What a subtype accepts must be looser

- Since the type of the elements given to foo is S1
  - bar can only accept the elements of any supertype T1 of S1



What a subtype returns must be stricter

- Since the places using f accept elements of type S2
  - those places also accept elements of any subtype T2 of S2



Covariant and contravariant

- The subtype relation is reversed for the arguments
  - Contravariant
- The subtype relation is the same direction for the return type
  - Covariant

How about the type of arrays?

- Suppose Cat <: Animal
  - What is the relation between Cat[] and Animal[]?
- Obvious it is not contravariant
  - Cat[] >: Animal[] does not make sense
    - *since not every Animal[] is a Cat[]*
- Covariant?
  - Does Cat[] <: Animal[] make sense?
    - *only when the arrays is immutable!*
    - *Otherwise, an Animal[] may contain a Dog*

In Java, arrays are covariant

- Cat[] is a subtype of Animal[]

- Without covariant, how could we handle arrays?
  - void handleArray(Object[] array1, Object[] array2)

  → only works for Object[]??

- In early versions of Java, Generics is not supported
  - before 5.0

In Java, arrays are covariant (cont.)

- Invariant is safe when the arrays is mutable

```
public class Test {
    public static void main(String[] args) {
        Cat[] cats = new Cat[1];
        Animal[] animals = cats;
        animals[0] = new Dog();    // exception!
    }
}
```

- Looks safe but causes runtime error!
  - java.lang.ArrayStoreException

In Java, generics is invariant

- Generics in Java
  - Let class/interface be parameterized over types

```
ArrayList<String> strs = new ArrayList<String>();
ArrayList<Object> objs = strs;    // error!
```

- error: incompatible types: ArrayList<String> cannot be converted to ArrayList<Object>

```
// if we allow it...
objs.add(new Integer(1));
String str = strs.get(0);    // what is it??
```

In Scala, it depends

- Immutable collections such as List
  - Covariant

```
val list:List[Any] = List[Int](1,2,3)     // ok
```

- Mutable collections such as Array
  - Invariant

```
val arr:Array[Any] = Array[Int](1,2,3)     // error!
```

Back to consider dispath

- How can we dispatch a method call?

```
obj.m(a, b, c);
```

- Subtyping polymorphism
  - *Overriding*
- Ad-hoc polymorphism
  - *Overloading*

- Use them together?

Overloading vs. Overriding

```
obj.m(a, b, c);
```

- Overriding
  - At runtime, late binding
  - Depends on the receiver (i.e. obj)
- Overloading
  - At compile-time, early binding
  - Depends on the types of parameters (i.e. a, b, c)

- Overriding is applied before overloading

Single dispatch

- The polymorphism for dispatching a method call used in languages such as Smalltalk, C++, and Java
  - Single: only the type of the receiver is used for dispatch
  - The types of parameters are not used for dynamic dispatch

An example of single dispatch

```
class Animal {
  void sayHelloTo(Animal a) {
    println("an animal says hello to an animal.");
  }
  void sayHelloTo(Cat c) {
    println("an animal says hello to a cat.");
  }
}
class Cat extends Animal {
  void sayHelloTo(Animal a) {
    println("a cat says hello to an animal.");
  }
  void sayHelloTo(Cat c) {    // this one??
    println("a cat says hello to a cat.");
  }
}
Animal x = new Cat();
Animal y = new Cat();
x.sayHelloTo(y);    // what is the result?
```

An example of single dispatch (cont.)

```
class Animal {
  void sayHelloTo(Animal a) {
    println("an animal says hello to an animal.");
  }
  void sayHelloTo(Cat c) {
    println("an animal says hello to a cat.");
  }
}
class Cat extends Animal {
  void sayHelloTo(Animal a) {
    println("a cat says hello to an animal.");
  }
  void sayHelloTo(Cat c) {
    println("a cat says hello to a cat.");
  }
}
Animal x = new Cat();
Animal y = new Cat();
x.sayHelloTo(y);    // a cat says hello to an animal.
```

Multiple Dispatch

- Dynamically dispatch based on the actual types of the arguments
  - Polymorphic on its arguments
- Multimethods
  - A set of methods supporting multiple dispatch
- Introduced in Common Lisp Object System (CLOS)

Generic Functions

```
(defclass bat ()())
(defclass cat ()())
(defclass dog ()())

(defgeneric talk (a b))

(defmethod talk ((b bat) (c cat))
  ;; for bat and cat
  )
(defmethod talk ((c cat) (d dog))
  ;; for cat and dog
  )
(defmethod talk ((b bat) (d dog))
  ;; for bat and dog
  )
```

- Packages a set of methods and select the right one to invoke

Reference

- *Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An Overview. In ECOOP'87, 1987.*

Generic Functions (cont.)

- Decouple objects and operations upon objects
- Message sending systems works well with unary operations
  - But run into problems with multiary operations
  - For example, sum a sequence of numbers
  - An object accepts a message and performs the addition

Reference

- *Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An Overview. In ECOOP'87, 1987.*

Next lecture

12/9 **Practice 6**

12/16 Aspect-Oriented Programming,
Event-Driven Programming,
and Context-Oriented Programming