## **Object-Orientation**



COMP 524: Programming Language Concepts Björn B. Brandenburg

The University of North Carolina at Chapel Hill

Based in part on slides and notes by S. Olivier, A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts.

Thursday, April 15, 2010

## What is OO?

### **Conceptual model.**

- Objects: opaque entities that have an identity, state, and behavior.
- Objects communicate by sending messages to each other.

### Metaphors.

- ➡Orchestra model.
  - Lot's of experts that can do one task well.
  - One conductor that coordinates overall problem solution.
- Service provider model.
  - An object provides (exactly) one service.
  - May rely on sub-contractors.



## What is OO?

### **Conceptual model.**

- Objects: opaque entities that have an identity, state, and behavior.
- Objects communicate by sending messages to each other.

### Metaphors.

- ➡Orchestra model.
  - Lot's of experts that can do one task well.

OO is a natural fit for problem decomposition: humans tend to think in terms of "objects" that "do" "things". OO recognizes this and supports this way of thinking.

**UNC Chapel Hill** 

## Benefits of OO

### Key features.

- → Encapsulation, information hiding. Reduces complexity, conceptual load, likelihood of errors.
- → Inheritance.
  - Increases productivity and code reuse.
- Abstraction, clean interfaces.
  - Improves code reuse, separation of concerns.
  - Enables large teams to develop in parallel.
- Sub-type **polymorphism**.
  - Code reuse.
- → Decoupling.
  - Code reuse.

**UNC Chapel Hill** 



## Benefits of OO

### Key features.

- ➡ Encapsulation, information hiding. Reduces complexity, conceptual load, likelihood of errors.
- → Inheritance.
  - Increases productivity and code reuse.
- → Abstraction, clean interfaces.
  - Improves code reuse, separation of concerns.
  - Enables large teams to develop in parallel.

➡ Sut Co → Dec

Co

OO has succeeded in practice because it makes individual developers and teams as a whole more productive (compared to procedural languages).

**UNC Chapel Hill** 

## Two Flavors of OO

### Focus on OO Concepts.

Pioneered by Smalltalk.

- Adopted by Ruby, Python, Javascript, etc.
- → Very dynamic.
  - Late binding.
  - Dynamic type checking.
  - Objects of the same class can differ in structure.

### Focus on Implementation.

- ➡ Pioneered by Simula 67. Adopted by C++, Java, C#, Eiffel, etc.
- Composite types.
- Some components are functions.
- All objects of one class must have same structure (memory layout).
- Optional early-binding.

**UNC Chapel Hill** 



## Two Flavors of OO

### Focus on OO Concepts.

- Pioneered by Smalltalk.
  - Adopted by Ruby, Python, Javascript, etc.
- → Very dynamic.
  - Late binding.
  - Dynamic type checking.
  - Objects of the same class can differ in structure.

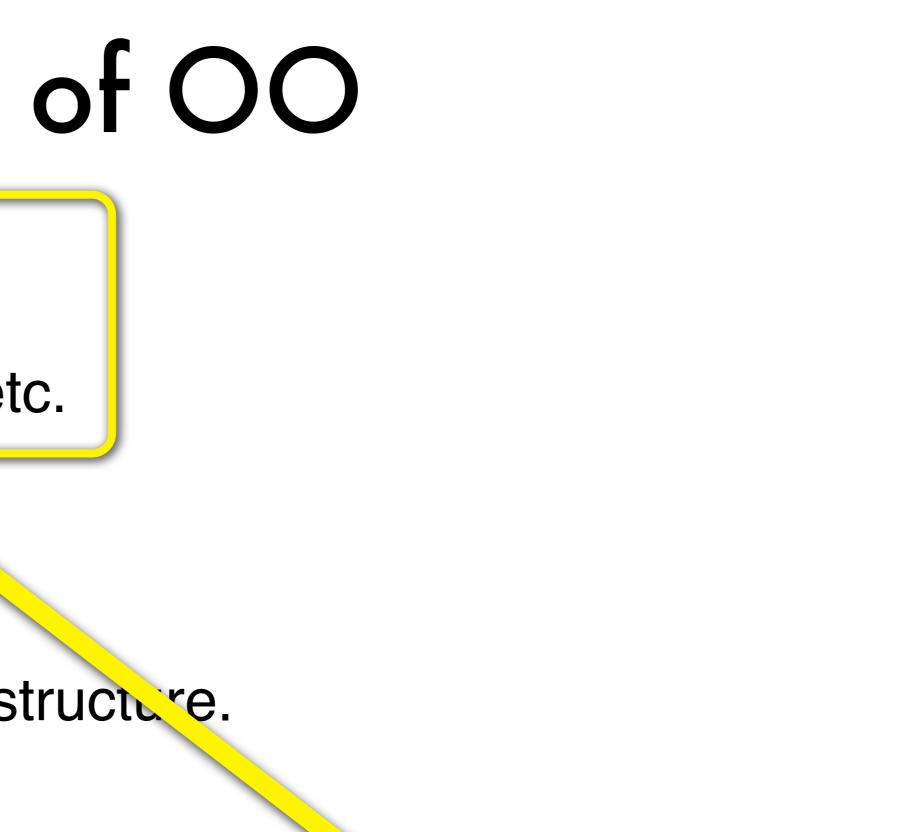
### Focus on Im

- ➡ Pioneered Adopted
- Composite

### Some com

## Pure object orientation: everything is an object (even numbers, functions, etc).

- All objects of one class must have same structure (memory layout).
- Optional early-binding.



## Model and Implementation

### Upon receipt of a message (= method call),

an object may <u>change state</u> (= update its attributes),

### <u>collaborate with other objects</u> (= call methods of other objects),

and finally <u>reply</u> (= **return value**).



UNC Chapel Hill Thursday, April 15, 2010

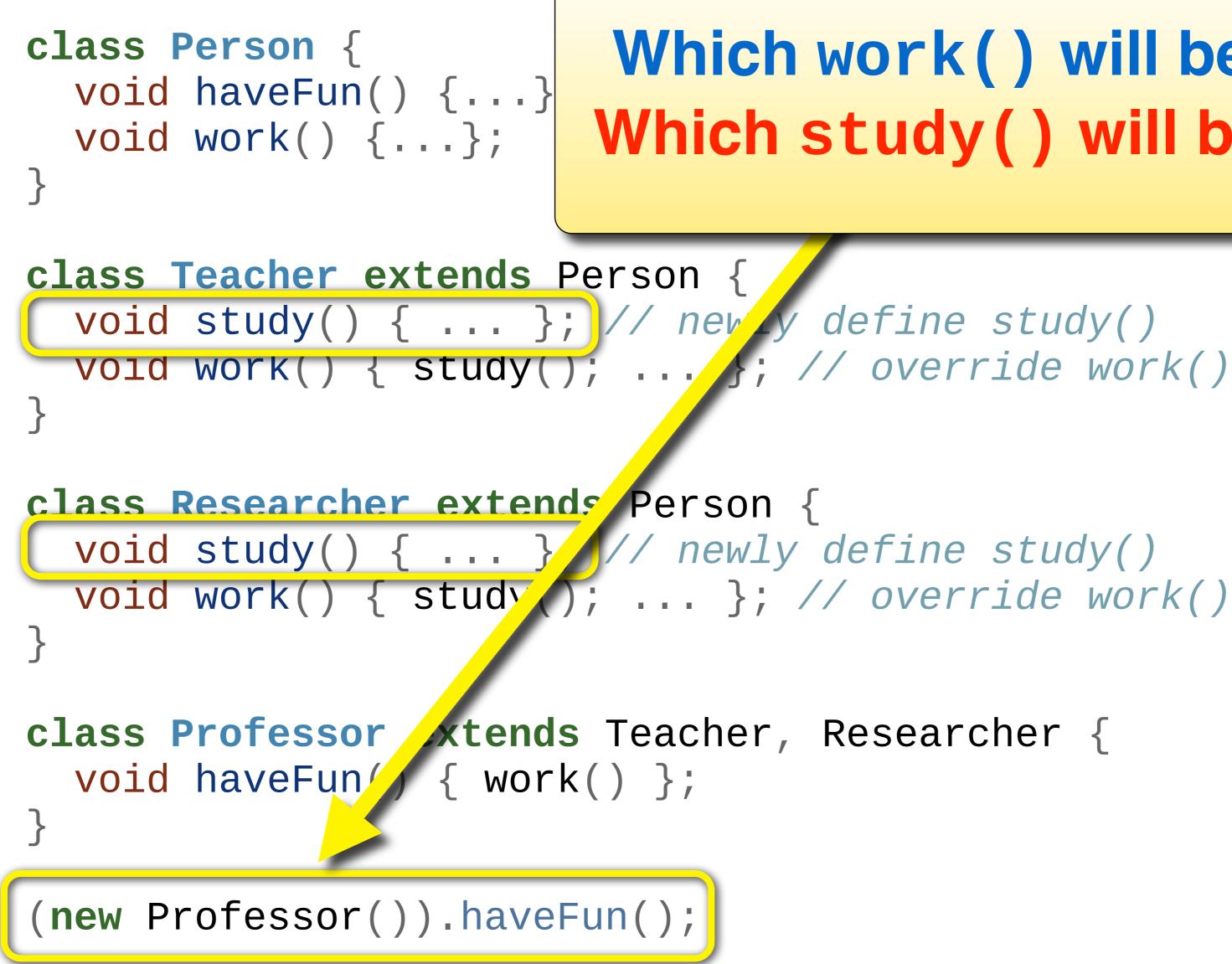
## Multiple Inheritance

```
class Person {
 void haveFun() {...};
  void work() {...};
class Teacher extends Person {
 void study() { ... }; // newly define study()
 void work() { study(); ... }; // override work()
}
class Researcher extends Person {
  void study() { ... }; // newly define study()
 void work() { study(); ... }; // override work()
}
class Professor extends Teacher, Researcher {
  void haveFun() { work() };
(new Professor()).haveFun();
```

**UNC Chapel Hill** 



## Multiple Inheritance



**UNC Chapel Hill** 

Brandenburg – Spring 2010

COMP 524: Programming Language Concepts

## Which work() will be called? Which study() will be called?



## Mix-in Inheritance

### **Restricted** alternative to multiple inheritance. Linear "true" inheritance: only single base class. →Can mix-in traits with a class. •e.g., Java interfaces.

### Interfaces + delegation.

→Pure interfaces: lot's of repeated code. Java's interfaces do not include default implementation.

Better alternative: provide a default class; delegate to member object.

UNC Chapel Hill

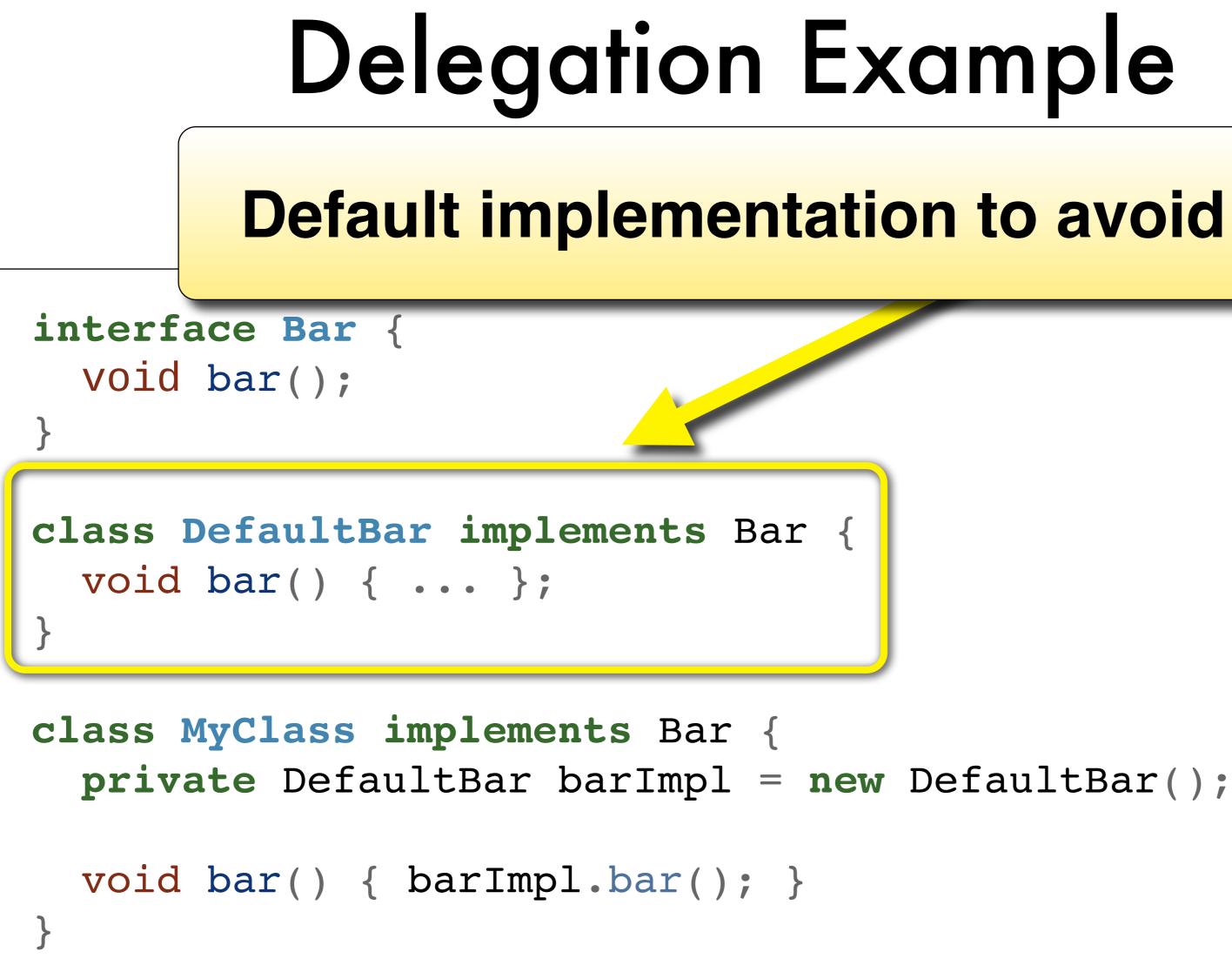


## **Delegation Example**

```
interface Bar {
 void bar();
}
class DefaultBar implements Bar {
 void bar() { ... };
}
class MyClass implements Bar {
  private DefaultBar barImpl = new DefaultBar();
 void bar() { barImpl.bar(); }
ר
```

**UNC Chapel Hill** Thursday, April 15, 2010

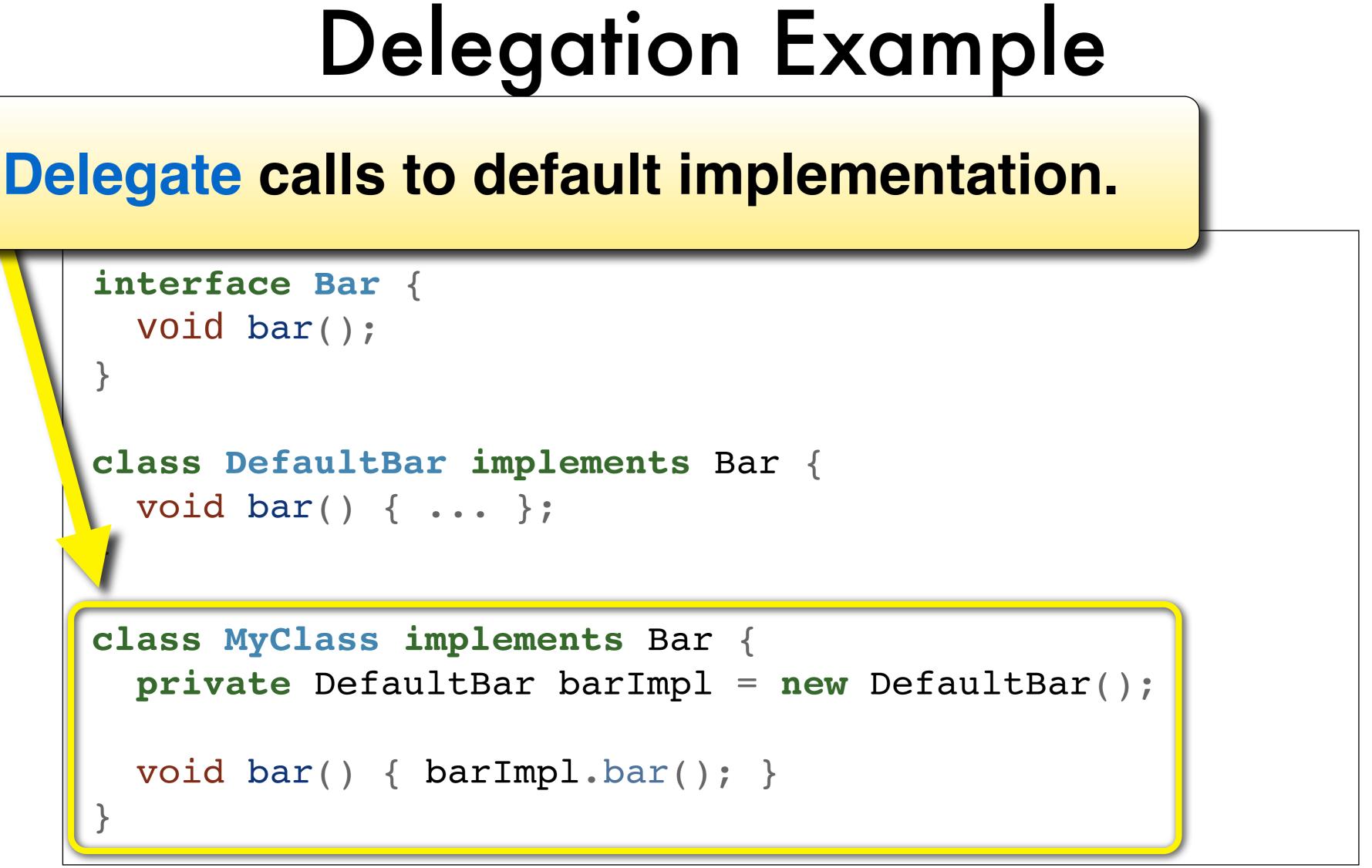






### **Default implementation to avoid repetition.**









## **Delegation Example**

```
interface Bar {
 void bar();
}
class DefaultBar implements Bar {
 void bar() { ... };
}
class MyClass implements Bar {
 private DefaultBar barImpl = new DefaultBar();
 void bar() { barImpl.bar(); }
}
```

## **C**# provides explicit delegate syntax



## **Delegation Example**

interface Bar {
 VOid bar();

## Scala's traits allow default implementations as part of the interface definition:

trait Similarity {
 def isSimilar(x: Any): Boolean
 def isNotSimilar(x: Any): Boolean = !isSimilar(x)
}
From: http://www.scala-lang.org/node/126

UNC Chapel Hill

Thursday, April 15, 2010



## Early vs. Late Binding

## Early Binding.

- → Static name resolution.
- Compiler determines at compile time which code will be called.
- →As efficient as a regular procedure call.

## Late Binding.

- →Name is resolved at runtime.
- Requires dynamic method dispatch. Incurs (small) overhead.





class A { void aFun() { ... }; class B extend A { void aFun() {...}; A obj = new B();obj.aFun();

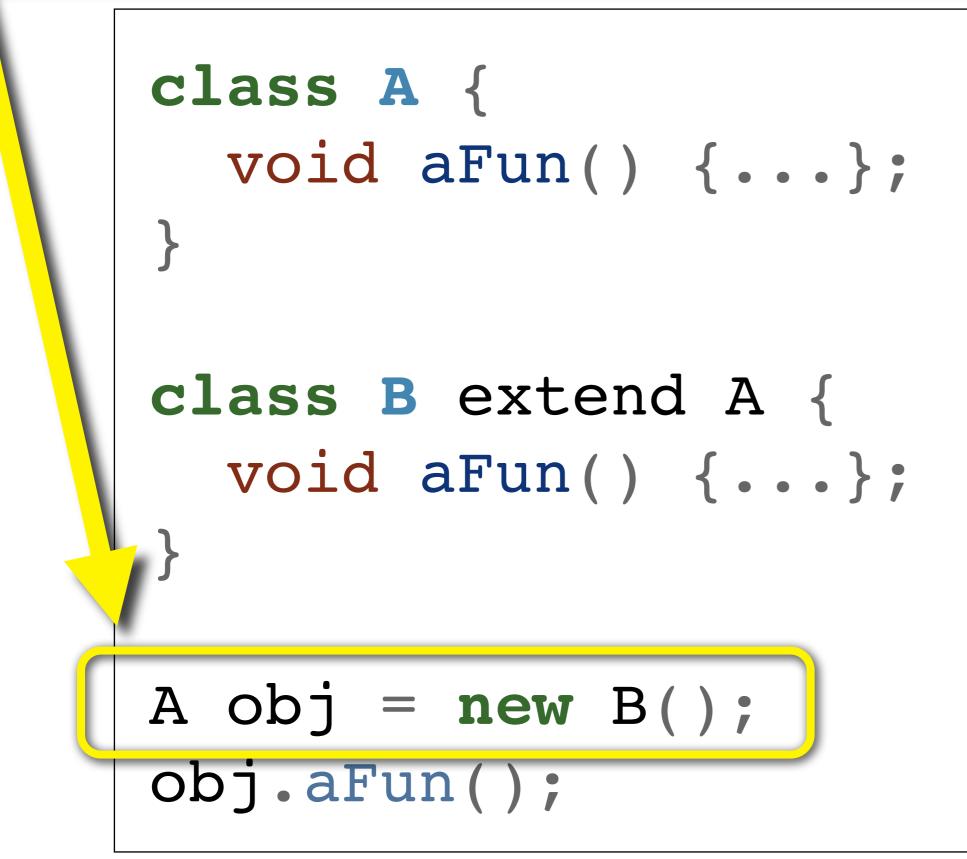


Brandenburg – Spring 2010



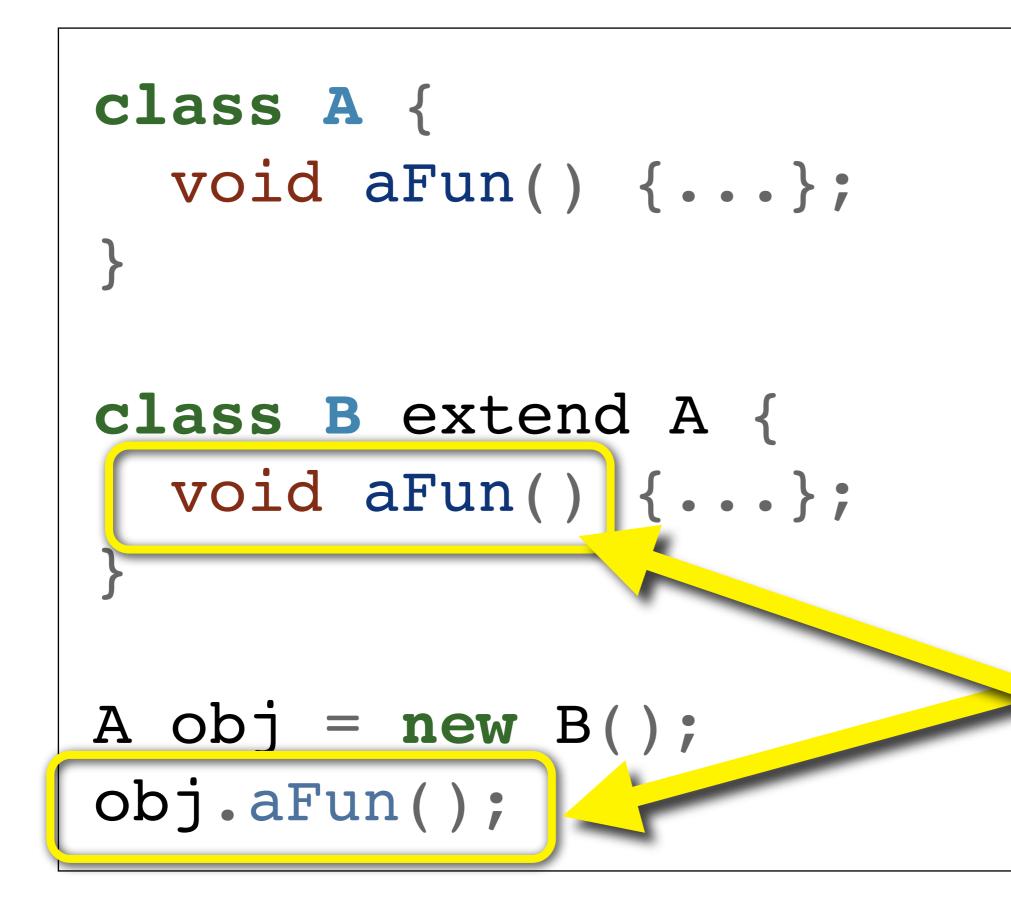


## Super-class reference type.











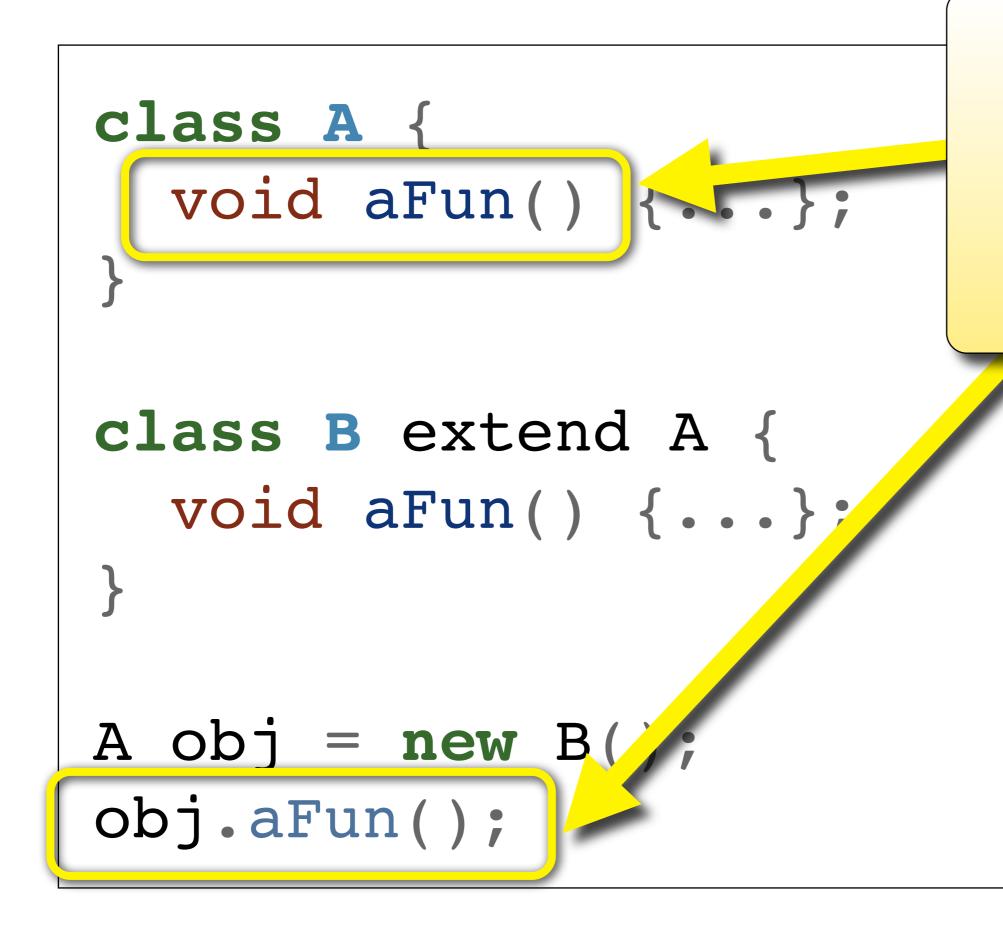
Brandenburg – Spring 2010

Thursday, April 15, 2010

COMP 524: Programming Language Concepts

## Late binding: **B.aFun()** is called.







Brandenburg – Spring 2010

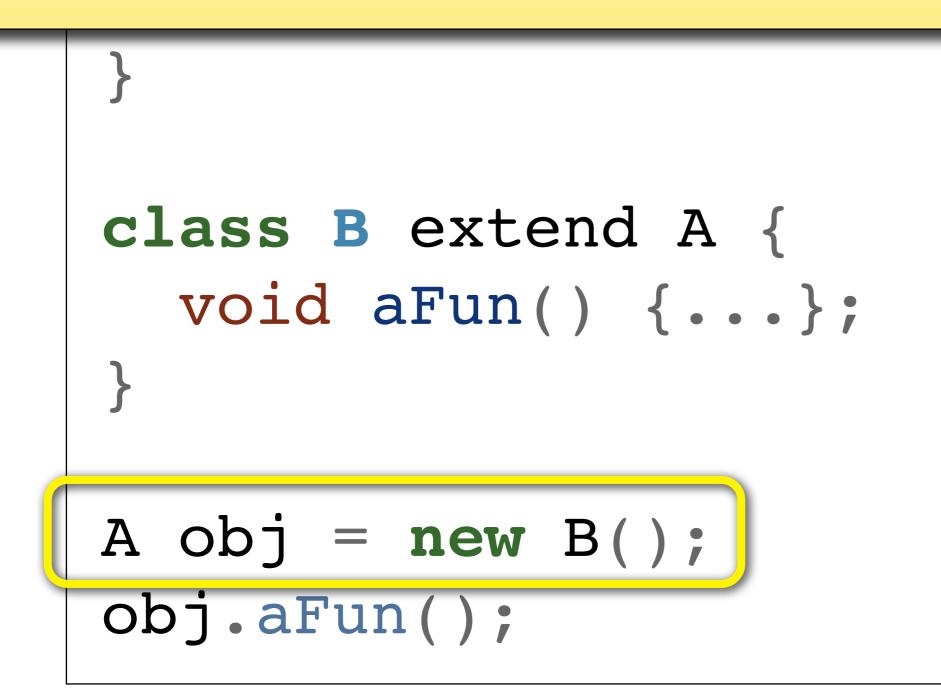
Thursday, April 15, 2010

COMP 524: Programming Language Concepts

## **Early binding:** A.aFun() is called.



Late binding: type of the object determines the method. Early binding: type of the reference determines the method.

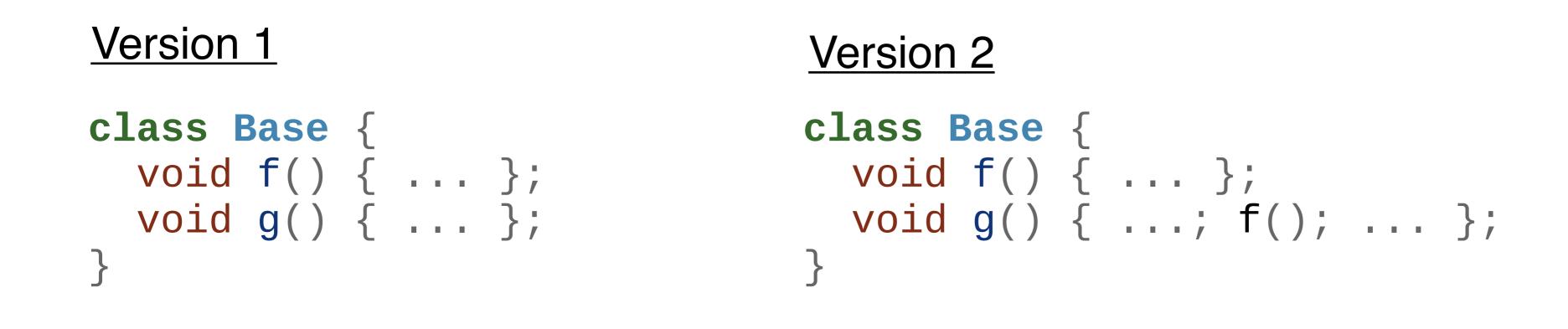




Thursday, April 15, 2010



## Fragile Base Classes apparently correct changes to a base class that break subclasses



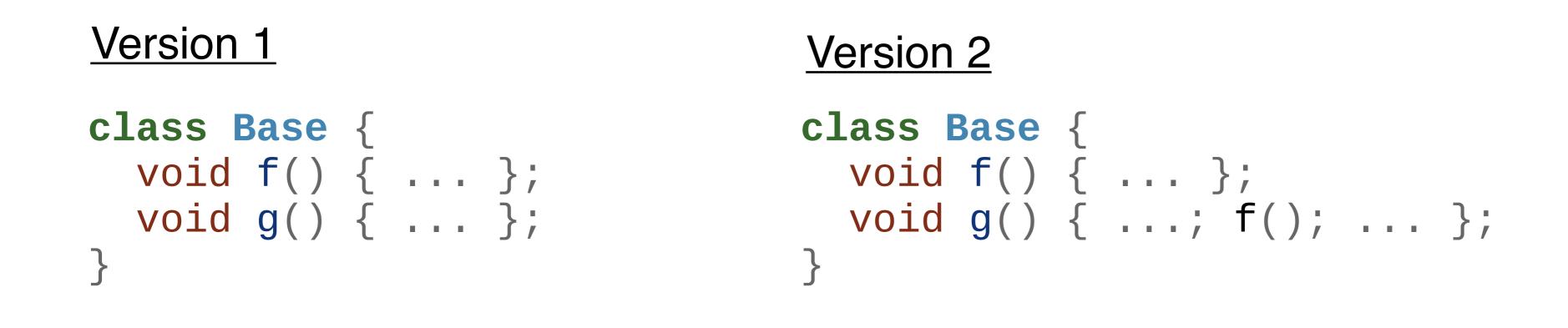
### <u>Client</u> class Child extends Base { void f() { ....; g(); .... }; }

UNC Chapel Hill

Thursday, April 15, 2010



## Fragile Base Classes apparently correct changes to a base class that break subclasses



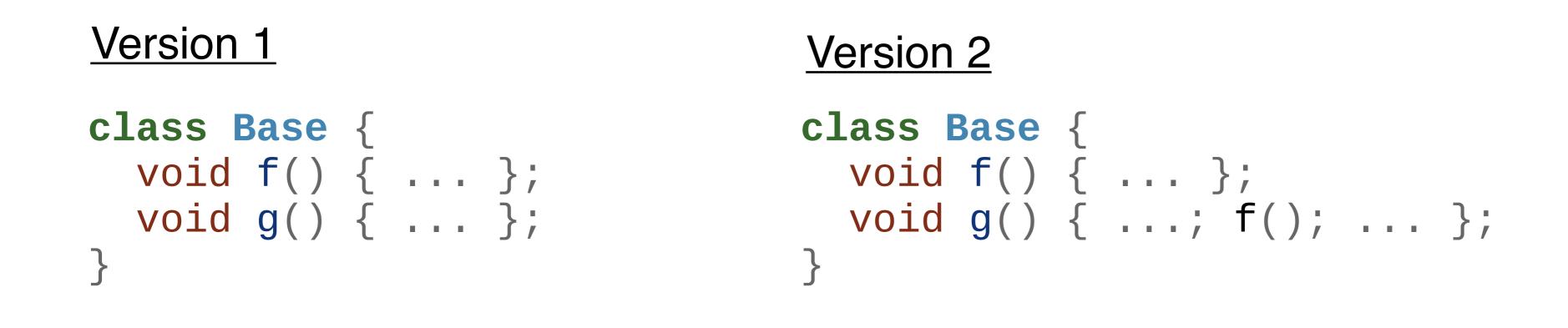
### <u>Client</u> class Child extends Base { void f() { ....; g(); .... }; }

UNC Chapel Hill

Thursday, April 15, 2010



## Fragile Base Classes apparently correct changes to a base class that break subclasses



### <u>Client</u> class Child extends Base { void f() { ....; g(); .... }; }

UNC Chapel Hill

Thursday, April 15, 2010





## Fragile Base Classes

## Large problem in practice.

- Many systems ship with large class libraries. • E.g., Java, C#/.NET, Objective-C.
- Developers can subclass system classes. Every upgrade can break previously-working code!

### Avoidance.

Requires careful class design. Later implementation changes should make very little assumptions.



## Fragile Base Classes

## Large problem in practice.

- Many systems ship with large class libraries. • E.g., Java, C#/.NET, Objective-C.
- Developers can subclass system classes. Every upgrade can break previously-working code!

## Avoidance.

Requires careful class design. Later implementation changes should make very little assumptions.

**Related problem:** binary compatibility vs. separate compilation. Recompilation necessary if base class changes.

## **Class Modification at Runtime** aka "monkey patching"

### **Pure OO: Everything is an object.** →Even classes.

Objects can change state.

In many dynamic languages this can be used to modify classes at runtime.

•E.g., Python, **Ruby**,...

Inheritance vs. modification. Inheritance leaves the superclass unchanged. Direct modification affects all modules using the class. Imagine amending the built-in string class...

UNC Chapel Hill

### class Base(object): def a\_method(self): print "a\_method was called"



UNC Chapel Hill

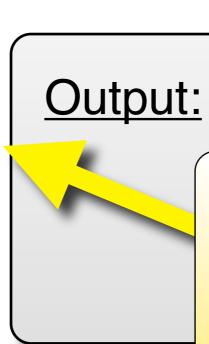
Thursday, April 15, 2010

Brandenburg – Spring 2010

Output:



### class Base(object): def a\_method(self): print "a\_method was called"



**UNC Chapel Hill** 

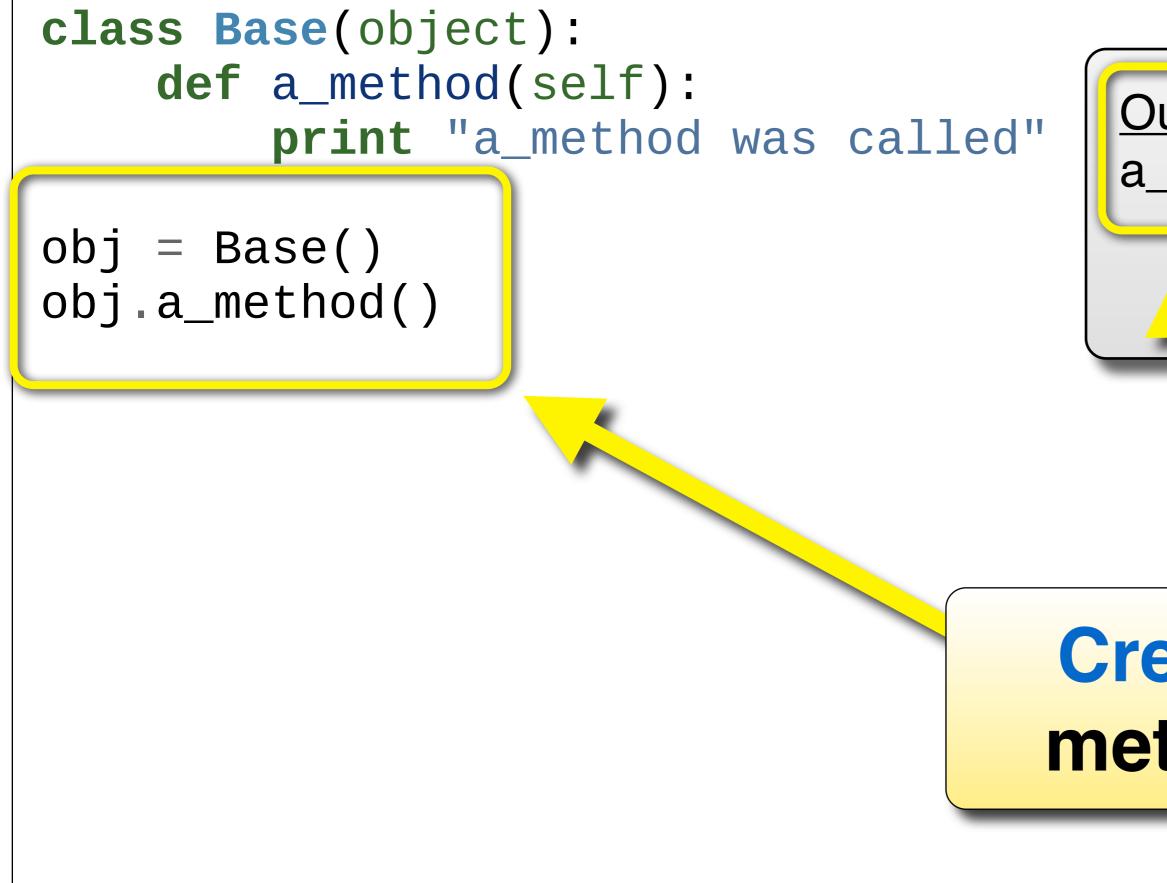
Thursday, April 15, 2010

Brandenburg – Spring 2010

COMP 524: Programming Language Concepts

## **Class definition** with one method.





**UNC Chapel Hill** 

Thursday, April 15, 2010

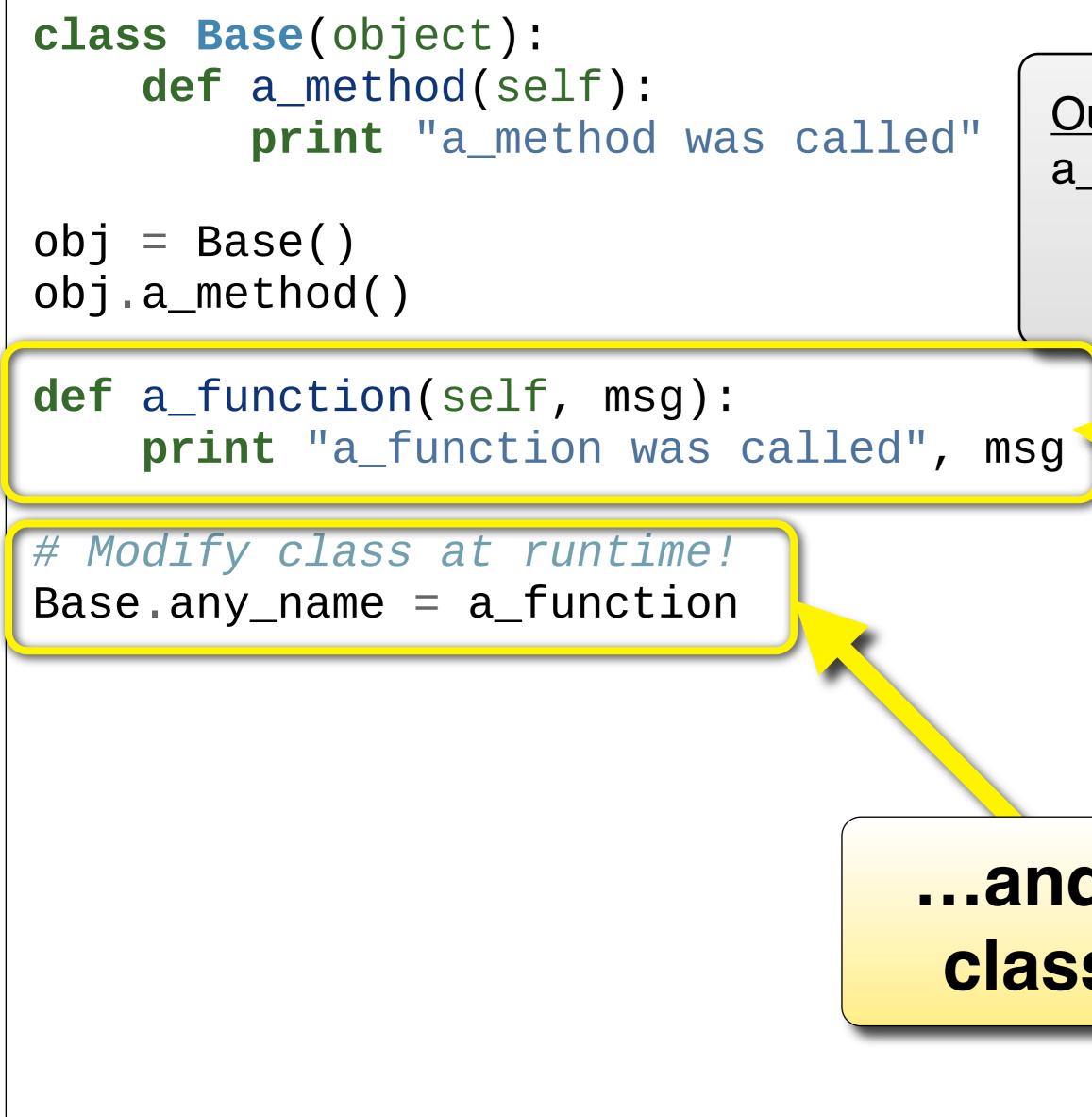
Brandenburg – Spring 2010

COMP 524: Programming Language Concepts

Output: a\_method was called

### **Create instance;** method is called.





**UNC Chapel Hill** 

Thursday, April 15, 2010

Brandenburg – Spring 2010

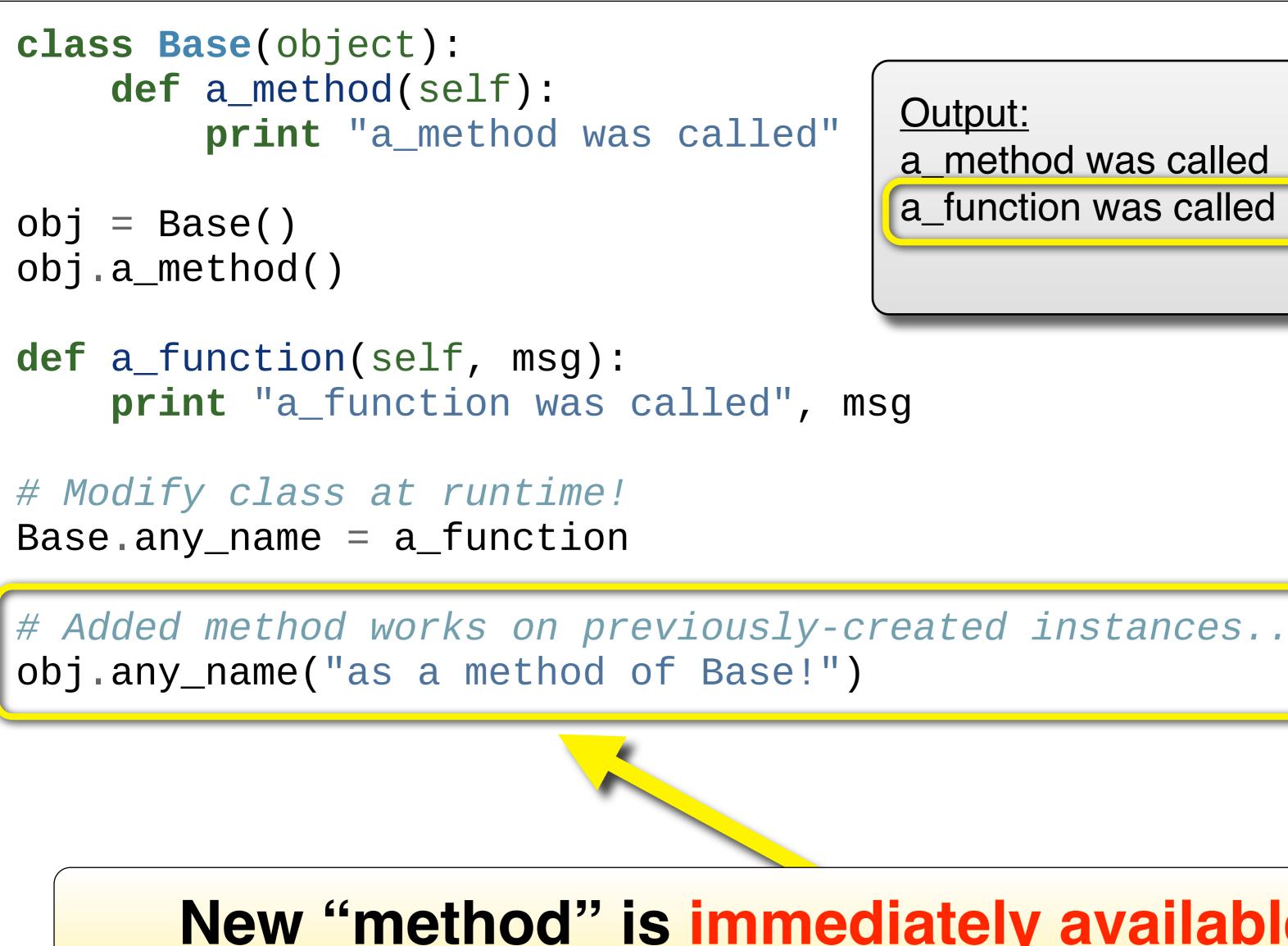
COMP 524: Programming Language Concepts

Output: a method was called

## **Define top-level** function...

### ...and add it to the class at runtime.





## New "method" is immediately available in all instances, as if declared in the class itself.

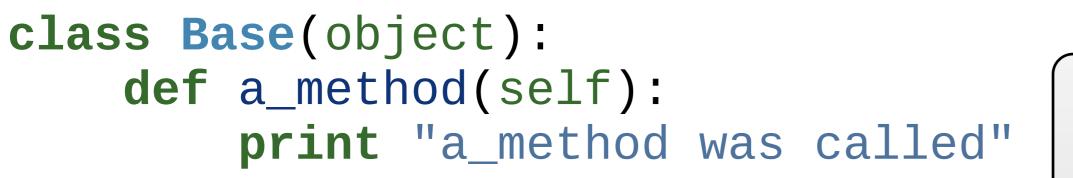
**UNC Chapel Hill** 

Thursday, April 15, 2010

Brandenburg – Spring 2010

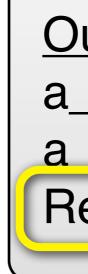
COMP 524: Programming Language Concepts

# <u>Output:</u> a method was called a\_function was called as a method of Base!



```
obj = Base()
```

obj.a\_method()



**def** a\_function(self, msg):  $\sim 11 \circ d^{\parallel}$ 

## **Can also replace (or remove)** previously-declared methods.

# Added method works on previously-created instances. obj.any\_name("as a method of Base!")

def dangerous(self): print "Replacing methods can cause tricky bugs!"

# Replace existing method at runtime!  $Base_a_method = dangerous$ 

```
obj.a_method()
```

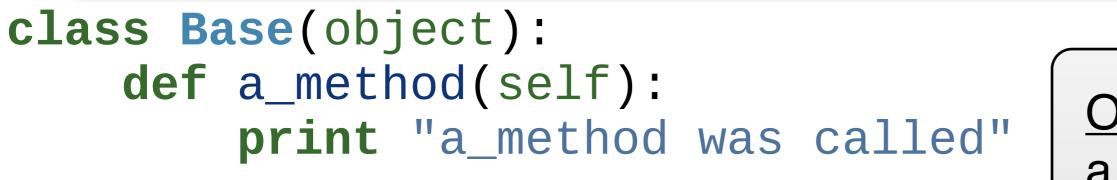
Brandenburg – Spring 2010

<u>Output:</u> a\_method was called a function was called as a method of Base! Replacing methods can cause tricky bugs!



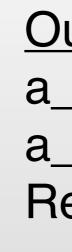


### In Python, some built-in classes that are implemented in C cannot be modified. In Ruby, virtually every class can be modified.



```
obj = Base()
obj.a_method()
```

14: O



**def** a\_function(self, msg): **print** "a\_function was called", msg

# Modify class at runtime! Base\_any\_name = a\_function

# Added method works on previously-created instances.. obj.any\_name("as a method of Base!")

**def** dangerous(self): print "Replacing methods can cause tricky bugs!"

# Replace existing method at runtime!  $Base_a_method = dangerous$ 

```
obj.a_method()
```

Output: a\_method was called a function was called as a method of Base! Replacing methods can cause tricky bugs!

## **Runtime Patches: Discussion**

### Uses.

- Add functionality, e.g., logging, caching, invariant checking,...
- → Fix bugs in third-party module.
- → Add convenience methods.
  - E.g., add a "make a file with this name" method to the string class (this is actually done in the Ruby-based <u>brew</u> package manager).

### **Dangers**.

- Two patches for the same class.
  - Unpredictable application: "last one wins."
  - Incompatible changes.
- Corresponding source hard to find (maintenance problem). • Eq., if you notice a **bug in a class in module A**, the corresponding code could reside in modules B, C, D, ...
- Fragile updates: changes to the class being patched can render runtime patches in any number of modules incorrect.

**UNC Chapel Hill** Thursday, April 15, 2010

## **Objects without Classes** prototype-based languages

- Some languages avoid classes completely.
- Pioneered by the language Self.
- Gaining in popularity (JavaScript is prototype-based.)

### Concept.

- Everything is an object.
- Objects have a prototype (reference to another object): Messages (i.e., method calls, member references) not handled by an object are redirected to the prototype.
- Objects are created by cloning an existing object, which becomes the prototype.
- Exact details vary between languages.

UNC Chapel Hill

## Prototype Example (JavaScript)

```
function Bar() {
  this.credits = "created by Bar"
}
function Foo() {
  this.credits = "created by Foo"
}
Bar.prototype.get_proto_name = function () { return "I'm a Bar." }
Foo.prototype.get_proto_name = function () { return "I'm a Foo." }
obj1 = new Bar()
obj2 = new Foo()
document.write("<br>--Before--<br>")
document.write("obj1 was " + obj1.credits + ": " + obj1.get_proto_name())
document.write("<br>")
document.write("obj2 was " + obj2.credits + ": " + obj2.get_proto_name())
obj1.__proto__ = Foo.prototype;
obj2.__proto__ = Bar.prototype;
document.write("<br>--After--<br>")
document.write("obj1 was " + obj1.credits + ": " + obj1.get_proto_name())
document.write("<br>")
document.write("obj2 was " + obj2.credits + ": " + obj2.get_proto_name())
```

UNC Chapel Hill

```
Output:
Can change prototype at runtime.
                                           --Before--
Equivalent to changing the "class."
    function Bar() {
                                           --After--
      this.credits = "created by Bar"
    function Foo() {
      this.credits = "created by Foo"
    Bar.prototype.get_proto_name = function () { return "I'm a Bar." }
    Foo.prototype.get_proto_name = function () { return "I'm a Foo." }
    obj1 = new Bar()
    obj2 = new Foo()
    document.write("<br>--Before--<br>")
    document.write("obj1 was " + obj1.credits + ": " + obj1.get_proto_name())
    document.write("<br>")
    document.write("obj2 was " + obj2.credits + ": " + obj2.get_proto_name())
    obj1.__proto__ = Foo.prototype;
    obj2.__proto__ = Bar.prototype;
    document.write("<br>--After--<br>")
    document.write("obj1 was " + obj1.credits + ": " + obj1.get_proto_name())
    document.write("<br>")
    document.write("obj2 was " + obj2.credits + ": " + obj2.get_proto_name())
```

obj1 was created by Bar: I'm a Bar. obj2 was created by Foo: I'm a Foo.

obj1 was created by Bar: I'm a Foo. obj2 was created by Foo: I'm a Bar.