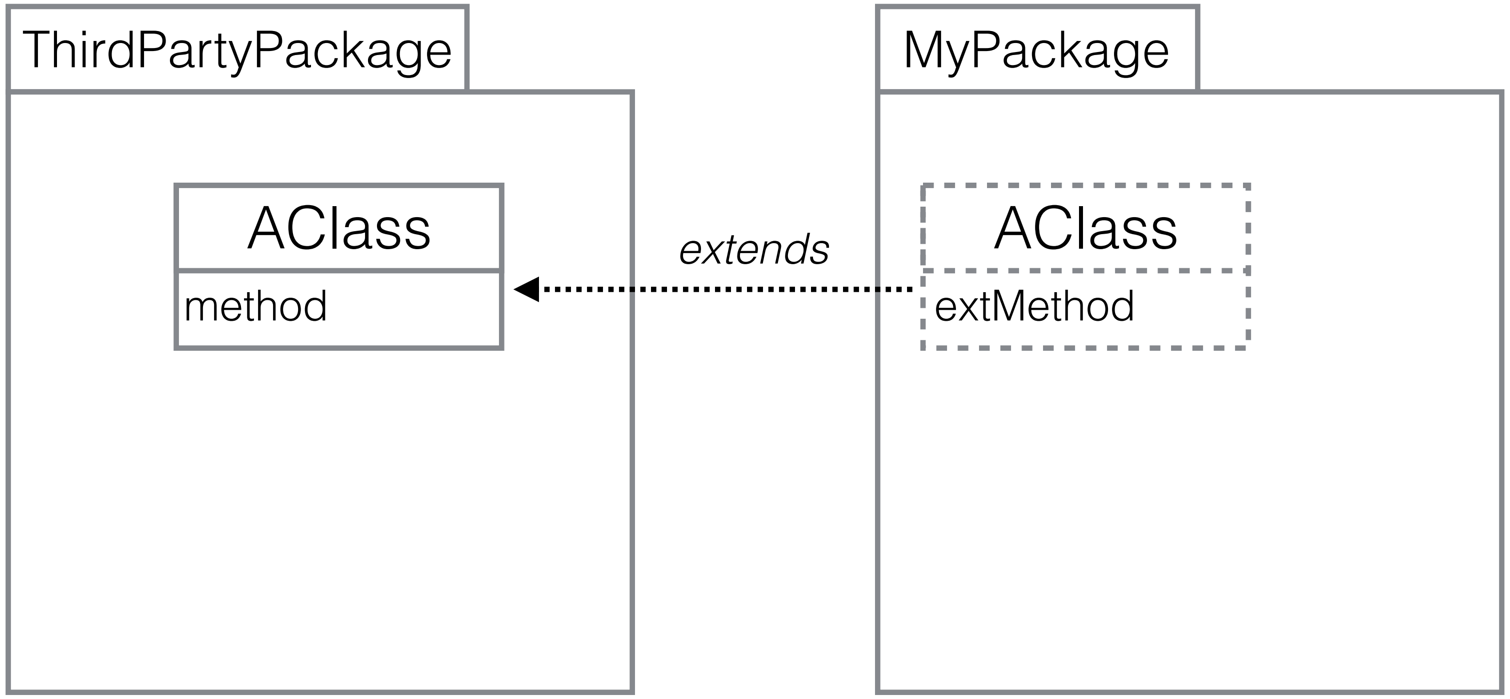
Scoped Selectors

About local extension methods and method visibility

Camille Teruel

Extension methods

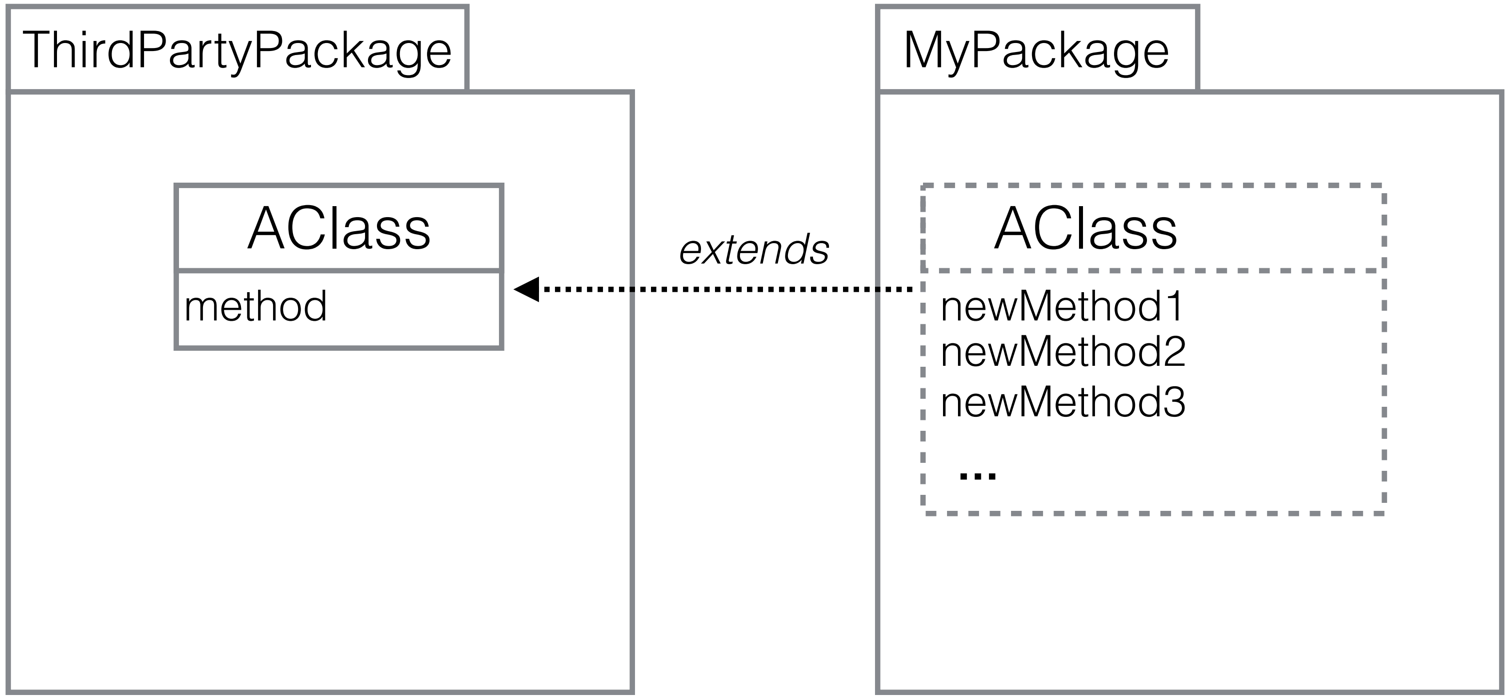
Extension methods



Add methods to classes you don’t own

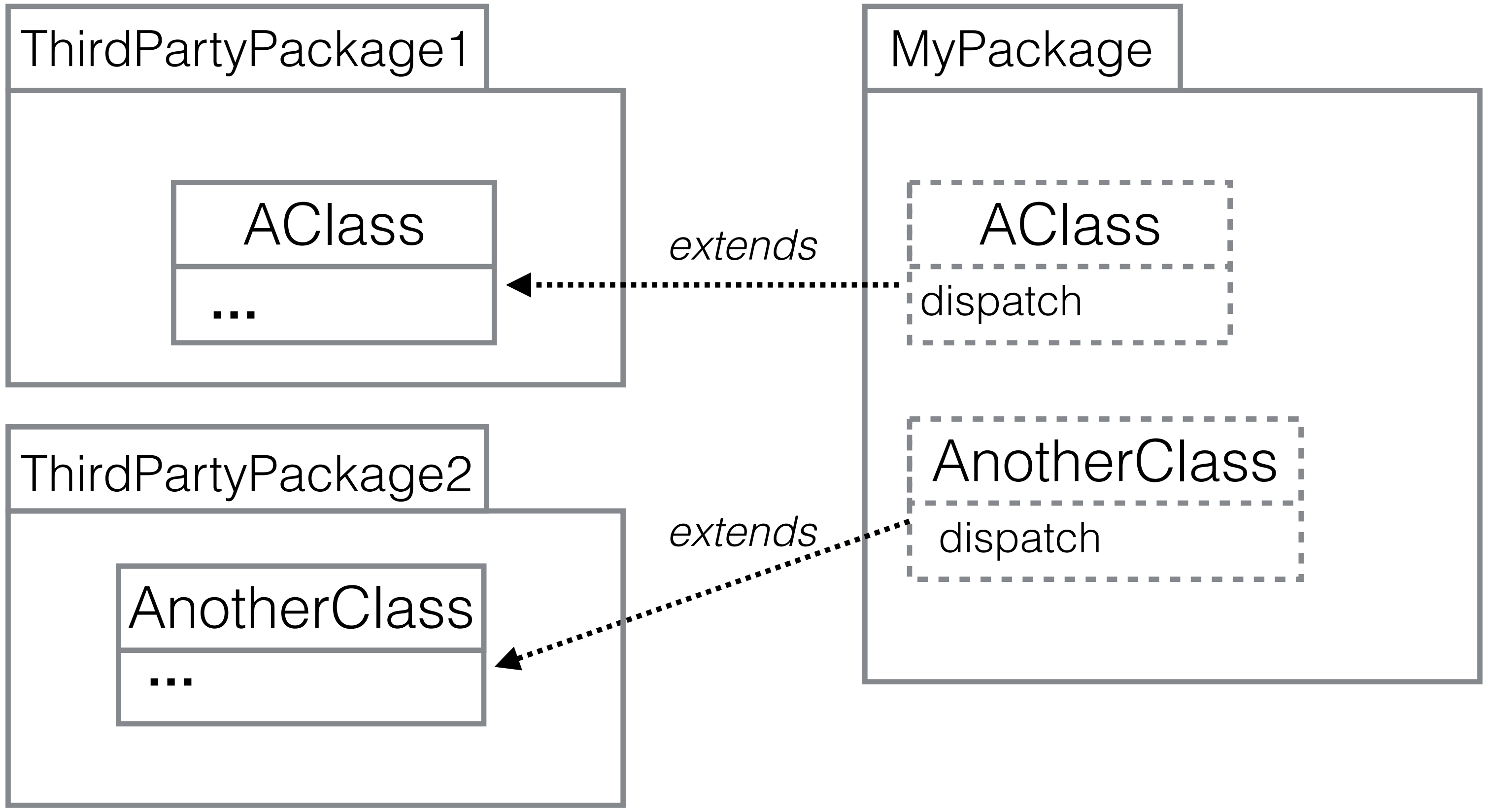
Sometimes a good alternative to subclassing
(no conversion needed)

Extension methods



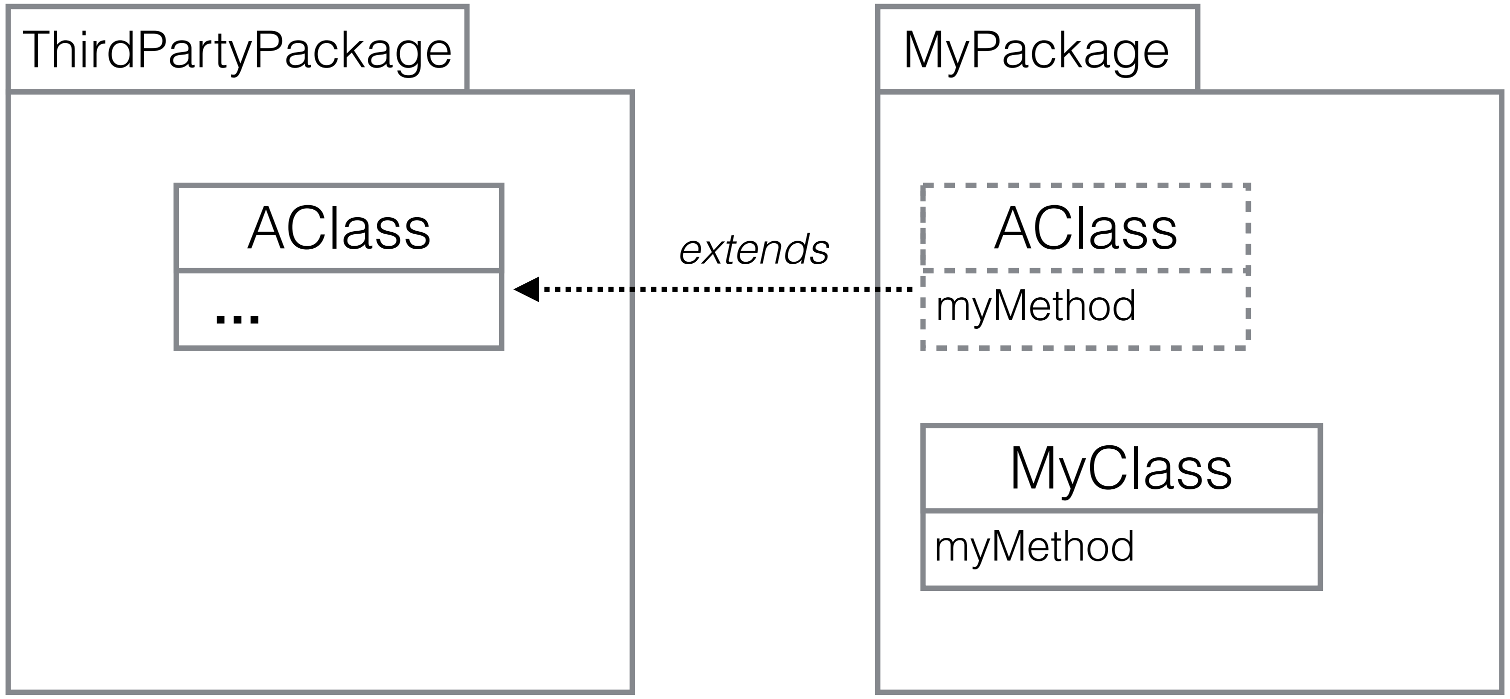
Add new functionalities to classes you don’t own

Extension methods



To dispatch on classes you don’t own

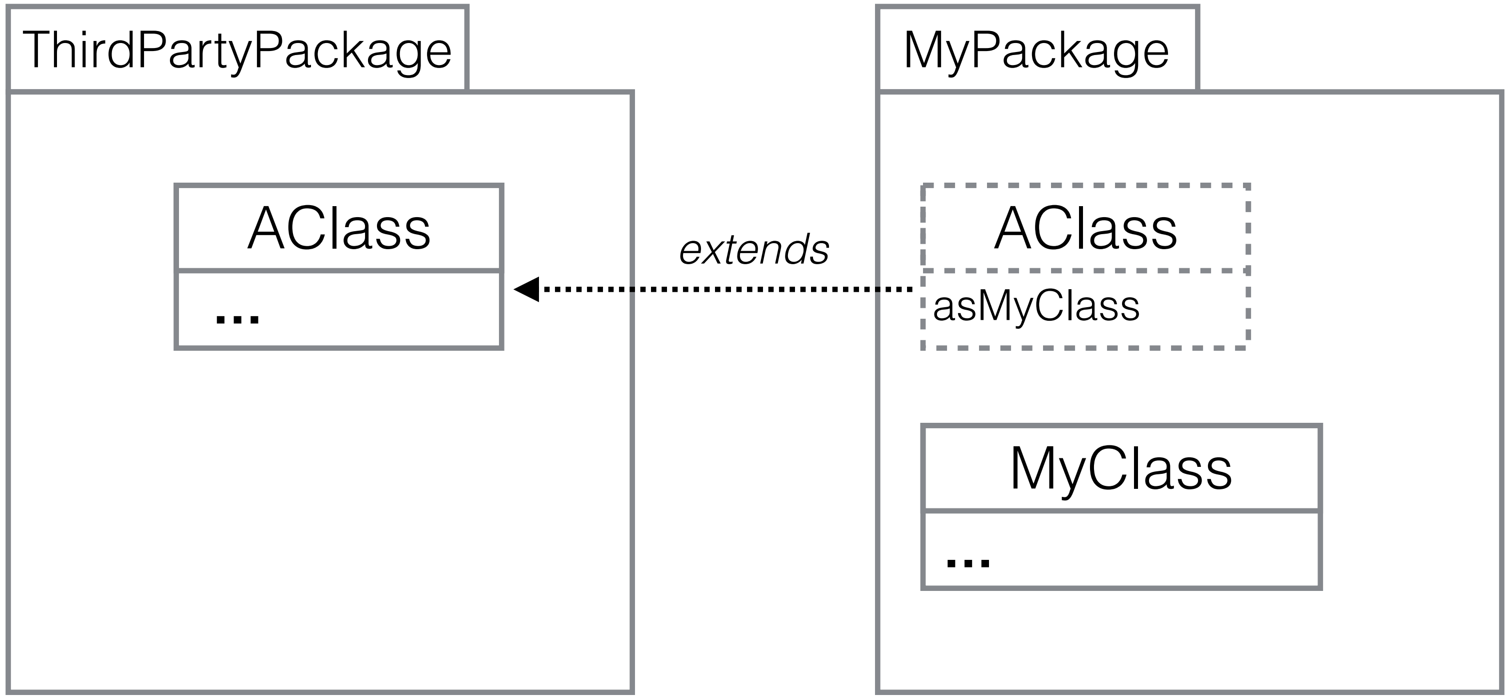
Extension methods



To make classes you don’t own polymorph to yours

(alternativ to Adapter Pattern)

Extension methods



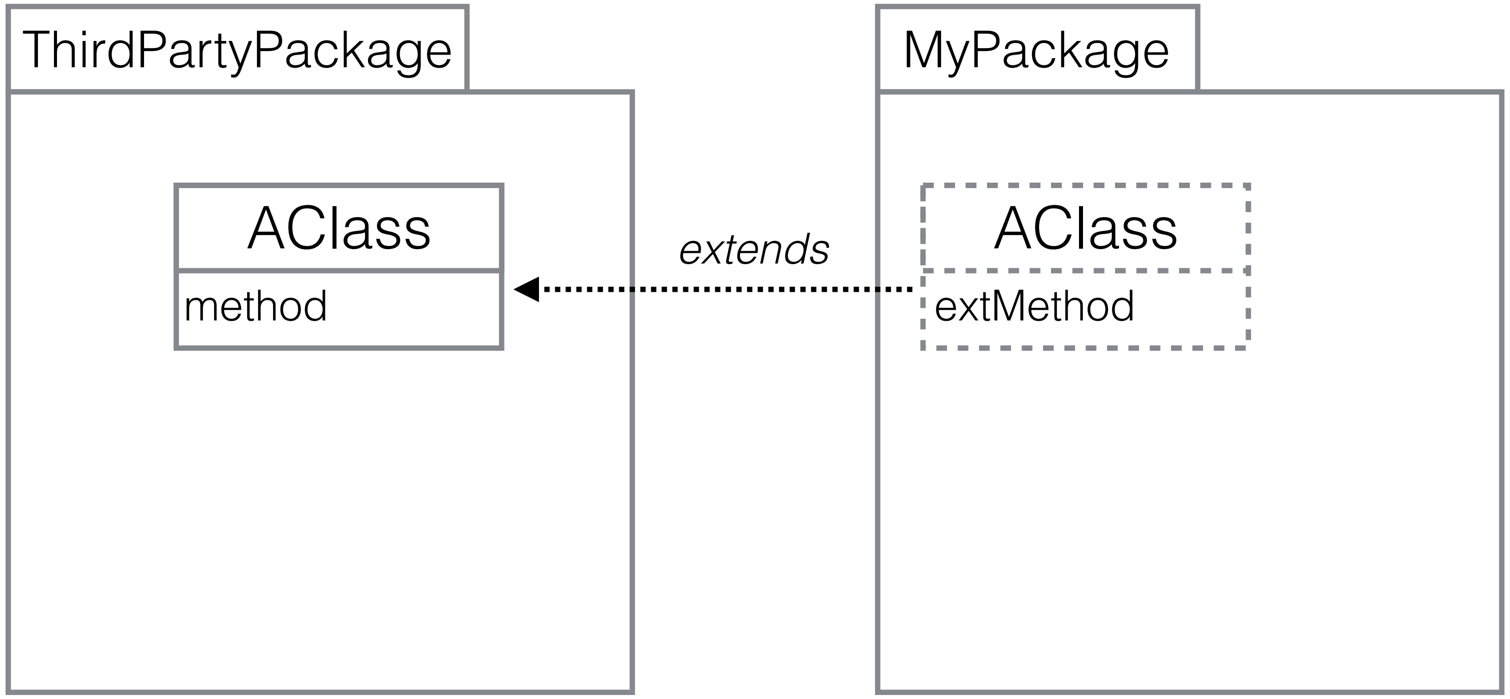
As syntactic sugar

(Smalltalk specific: unary>binary>keyword)

```
… (MyClass from: AClass new) …
    … AClass new asMyClass …
```

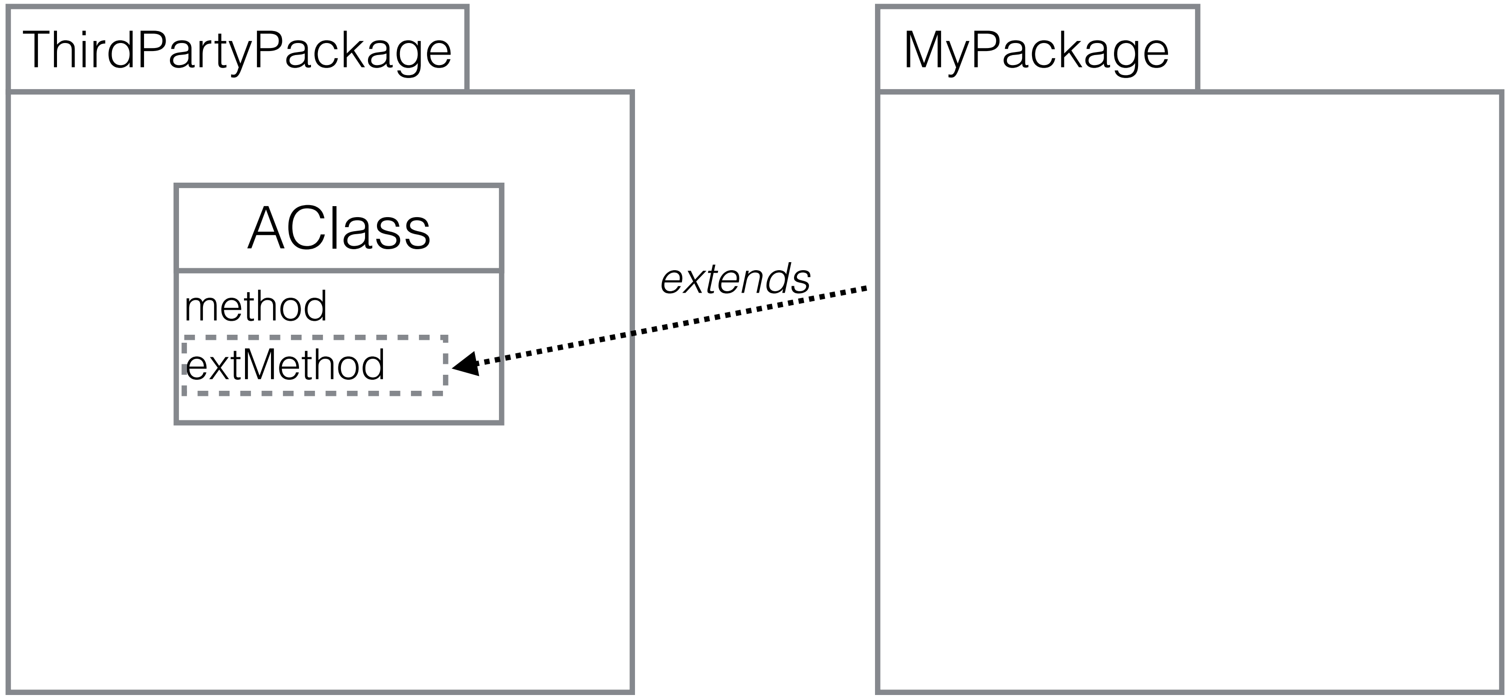
Extension methods: Problems

Problems



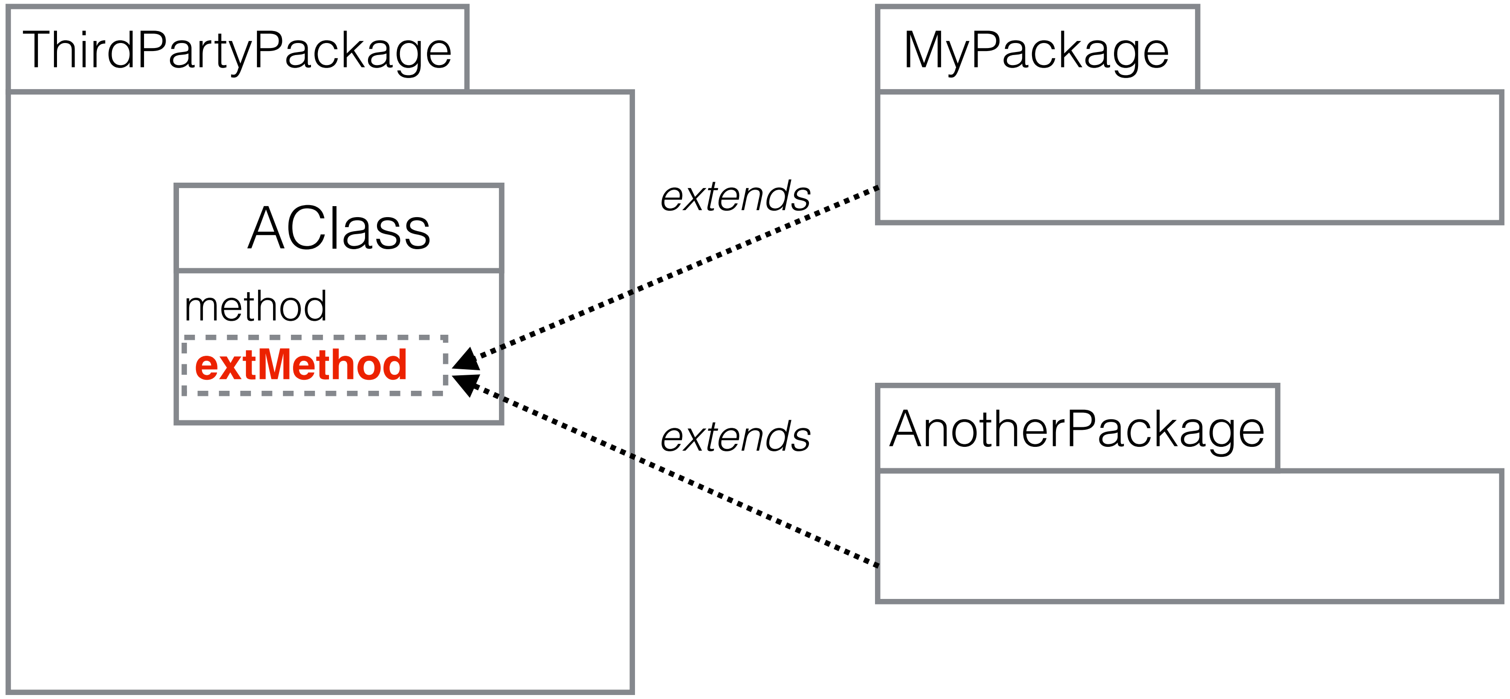
Extension methods are globally visible

Problems



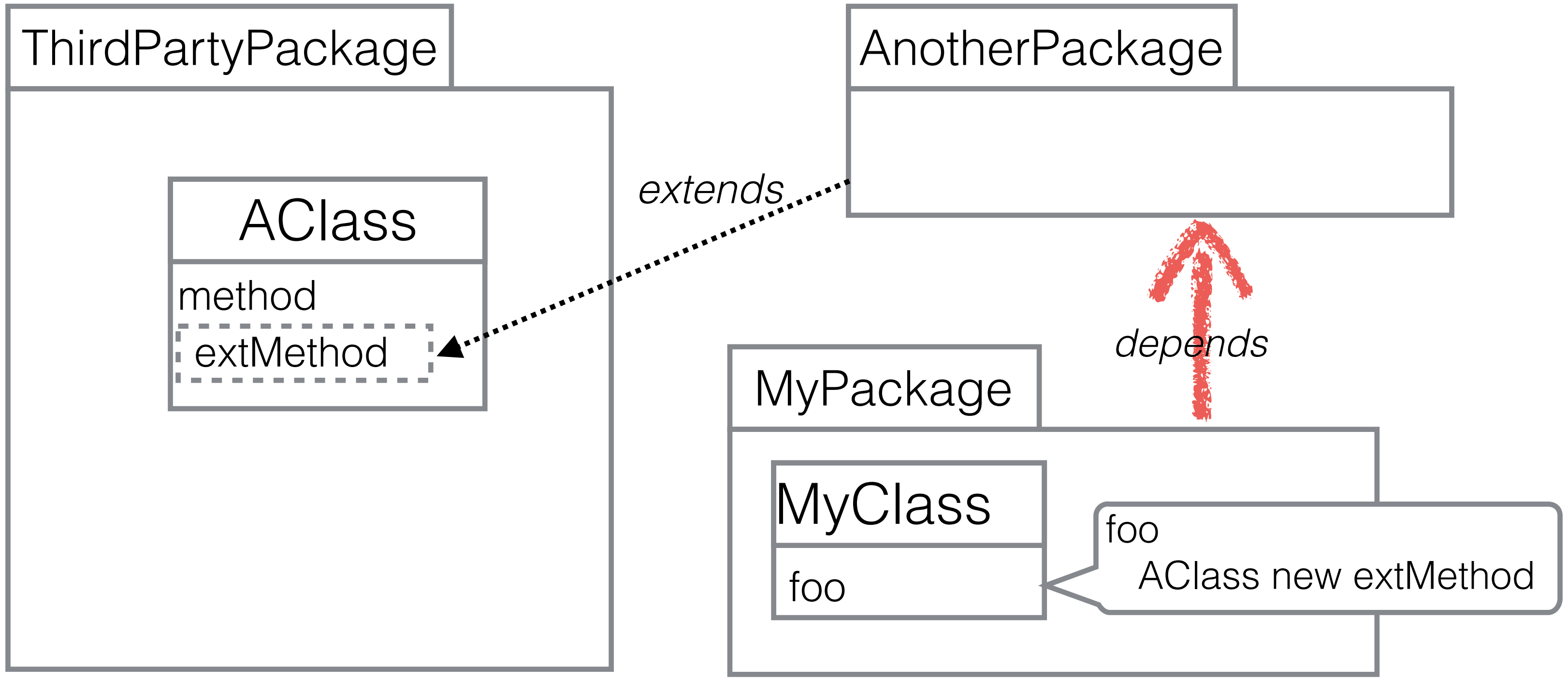
Extension methods are globally visible

Problems



Clash!! Who wins?

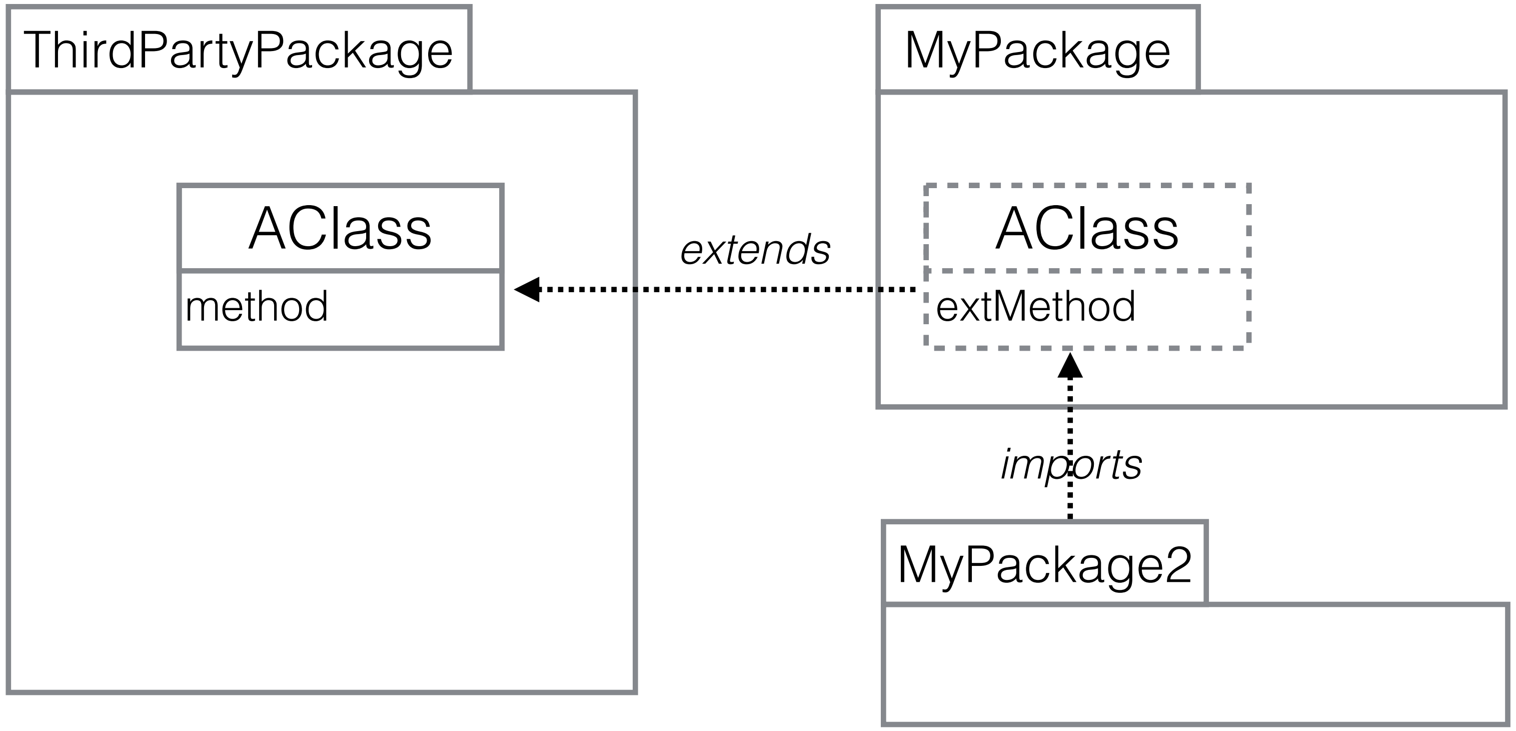
Problems



Sneaky dependencies

Local Extension methods

Local extension methods



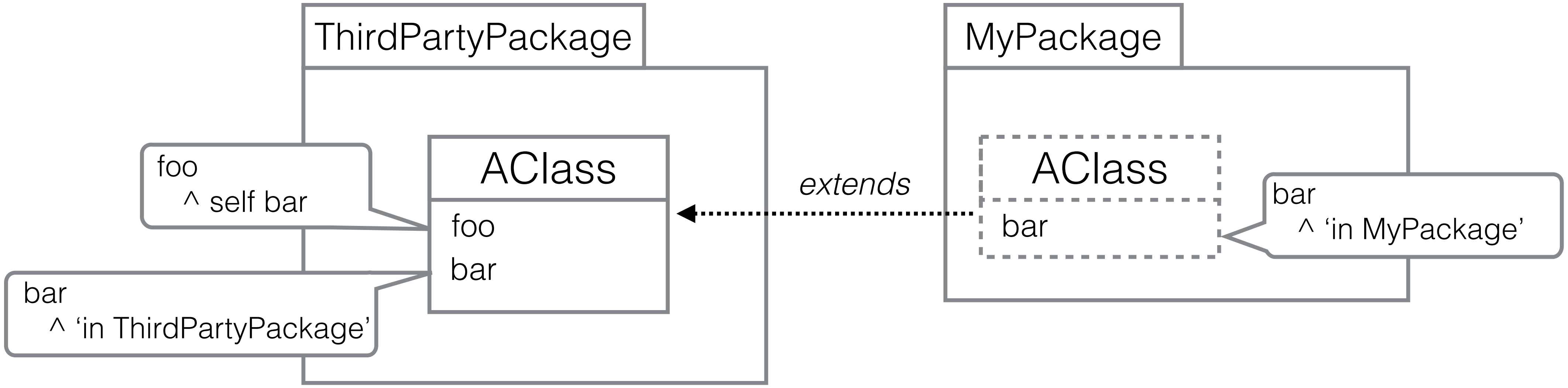
Visible only from packages that declare or import it

Local extension methods

If extension methods are local,
can I override a method locally?
What does that mean?

Local Rebinding

Local Rebinding

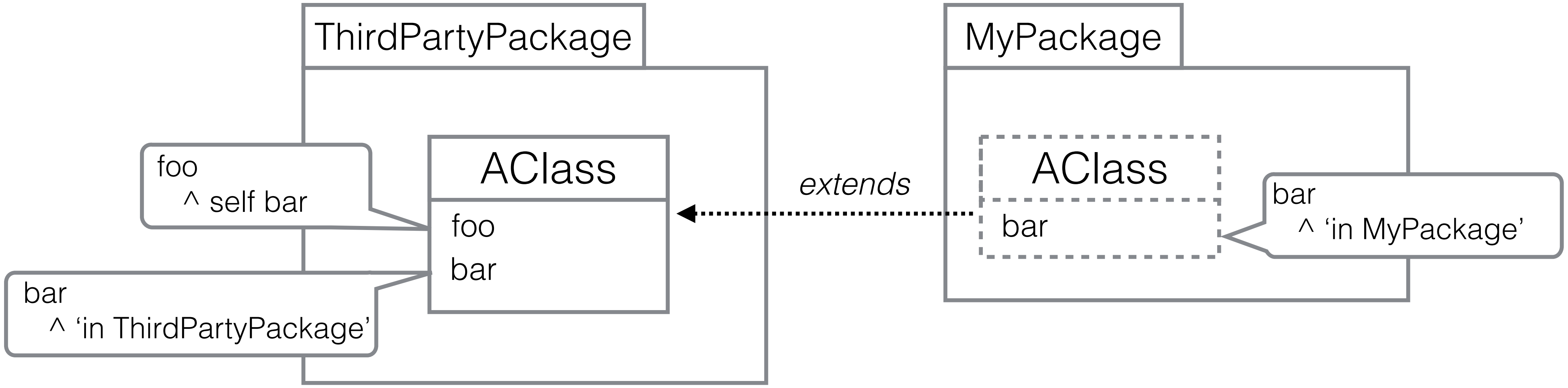


In a class of MyPackage:

```
AClass new foo
```

➞ **?**

Local Rebinding



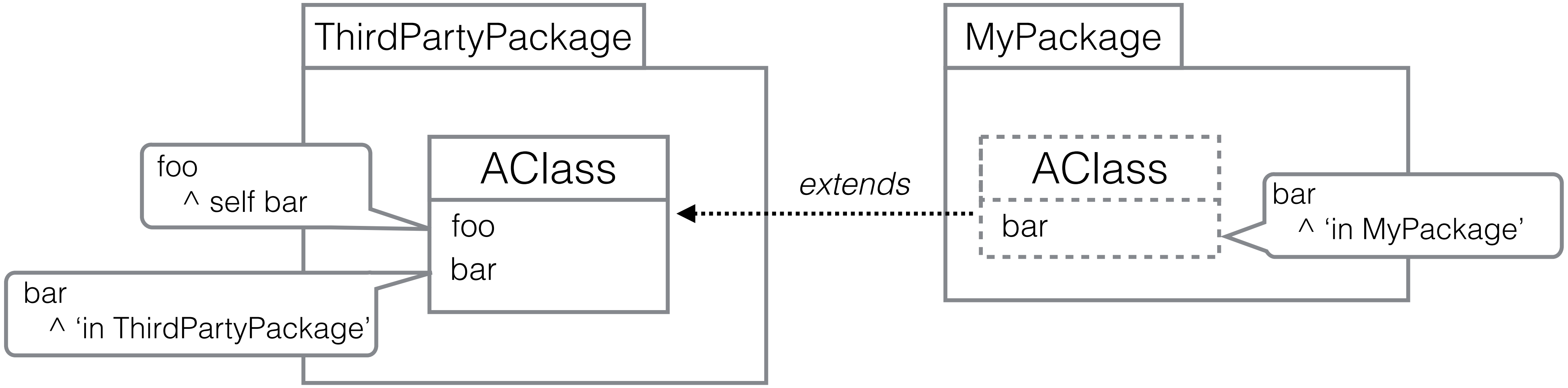
In a class of MyPackage:

```
AClass new foo
```

→ `'in MyPackage'`

Local rebinding

Local Rebinding



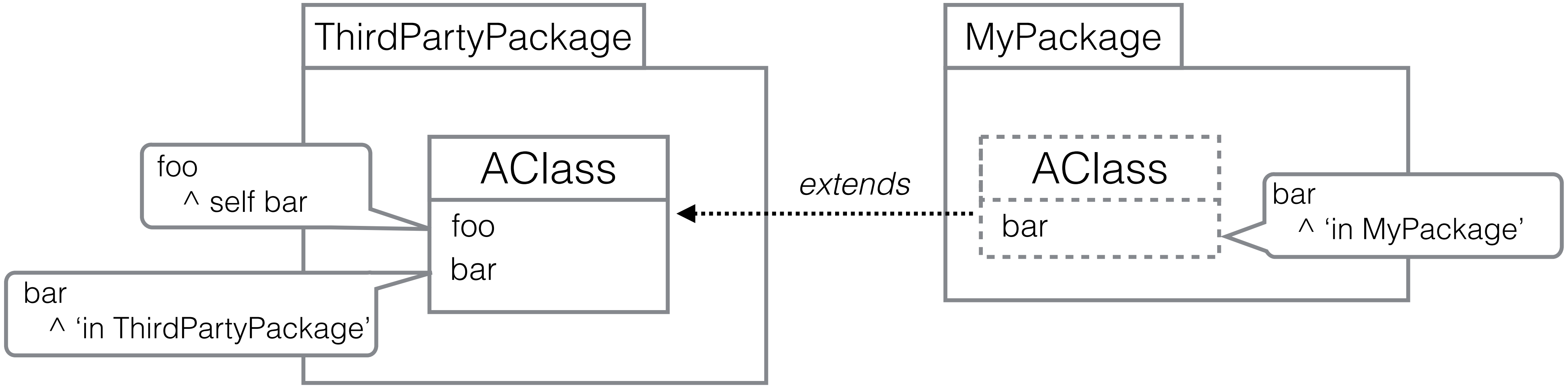
In a class of MyPackage:

```
AClass new foo
```

→ `'in ThirdPartyPackage'`

No Local rebinding

Local Rebinding



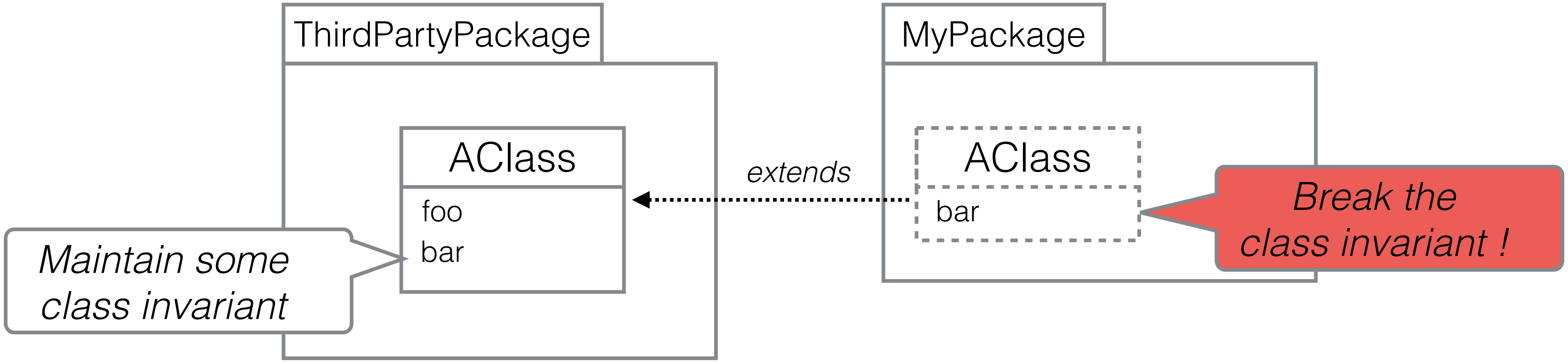
In the context of MyPackage:

```
AClass new foo
```

→ *Which one's your favorite?*

Local Rebinding: Problems

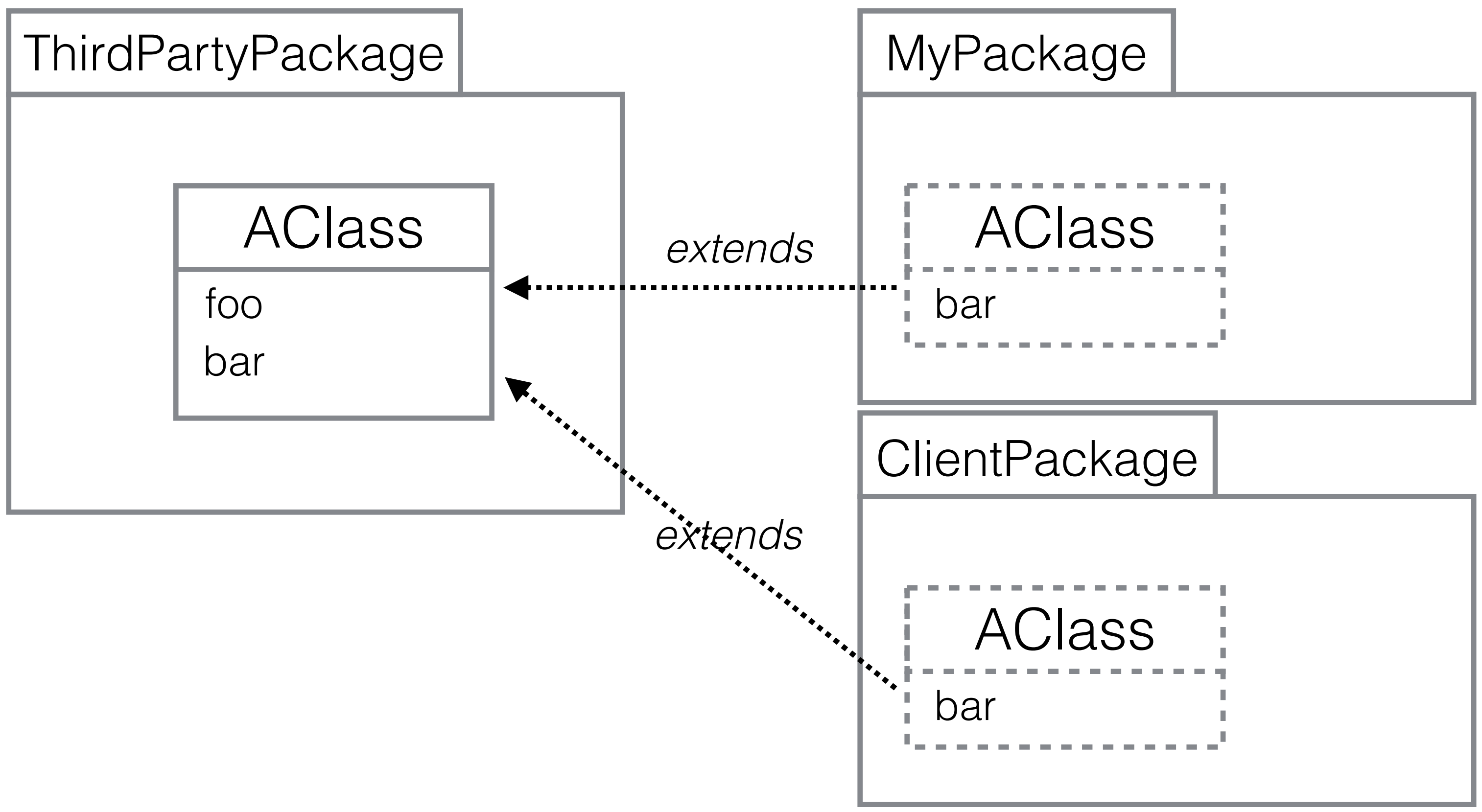
Problems:
Class Encapsulation



You can corrupt a class behavior

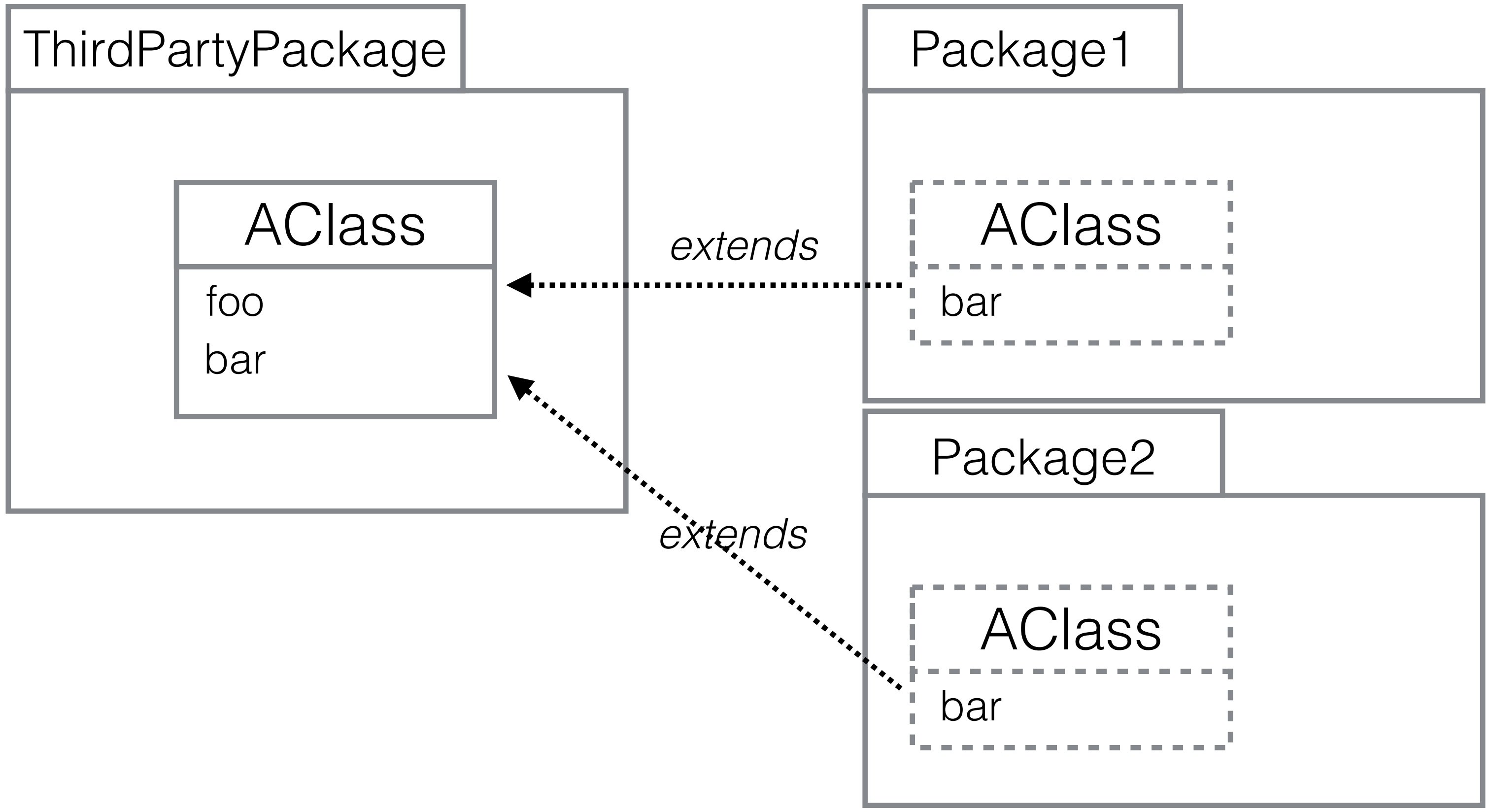
(accidentally or not)

Problems:
Package Conflicts



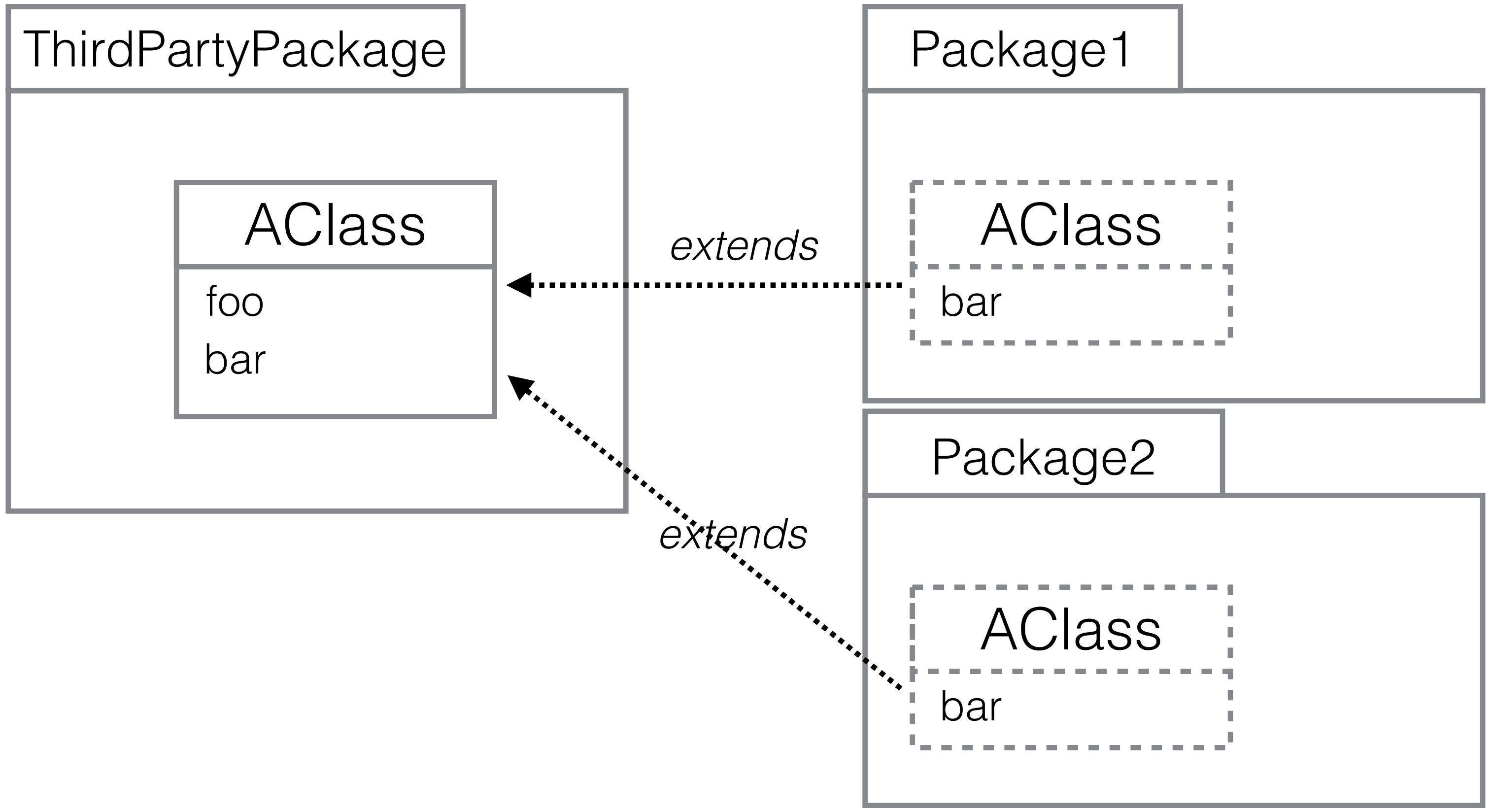
Clients have precedence!
A client can override accidentally
the behavior you expect

Problems:
Package Conflicts



Who wins?

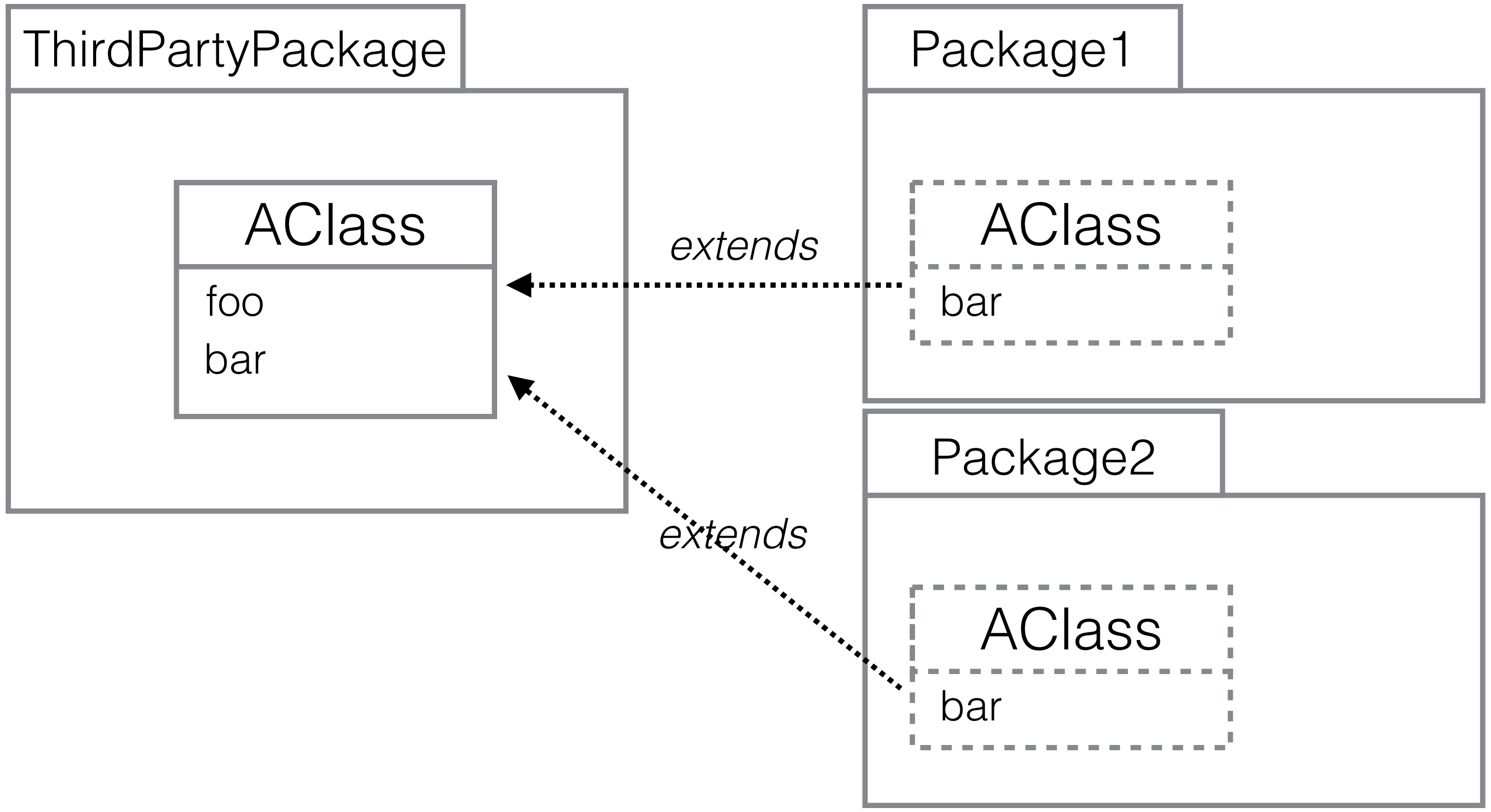
Problems: Package Conflicts



Oldest caller always wins

(depends on the state of the whole call stack)

Problems:
Package Conflicts



You **must know** the implementation of the packages you **transitively** depends on

Problems:
Package Conflicts

Local rebinding breaks the purpose of local extension methods…

Solutions?

- Look back just one context. *Weird…*
- Newest caller wins. *Just invert the problem*
- Let the programmer decides up to where to look back. *How to specify that? Do you want that?*

Problems: Performances

To enable local rebinding you must introspect (thus reify) the whole call stack…

Local Rebinding: Conclusion

- *Seems* more “natural”
- Breaks encapsulation
- Leads to conflicts
  (that local extension methods are supposed to solve)
- Performances

→ I do ***not*** want it,
do you?

Scoped selectors

Scoped Selectors

- A mechanisms to make extension methods local
- Without local rebinding
- Without performance cost
- Like Selector Namespaces?
- Also permits to set visibility on methods
  (private, protected, other kind)

Scoped Selectors: Design Survey

How to Import?

What’s the granularity…

- …of importee?
  - Extension method (tedious)
  - Class extension (extension methods for the same class)
  - Extension, *i.e* any set of extension methods (subsumes others)

How to Import?

What’s the granularity…

- …of importer?
  - Package
  - Class
  - Method

Overriding

With subclassing, when you override a method, you can still call the overridden method thanks to `super`.

What about extension methods?

What keyword? (`super` or another)

Extend an extension?

Use case:

- A probability package provides an Extension
- A statistics package *extends* that extension to add new methods.
- Importing this new extension makes both set of methods available

Scoped Selectors: Implementation

Implementation: Idea

- A selector is two-fold:
  - A *verb* that you type down in source code and that denotes a message name
  - A *key* object used to look up methods in method dictionaries

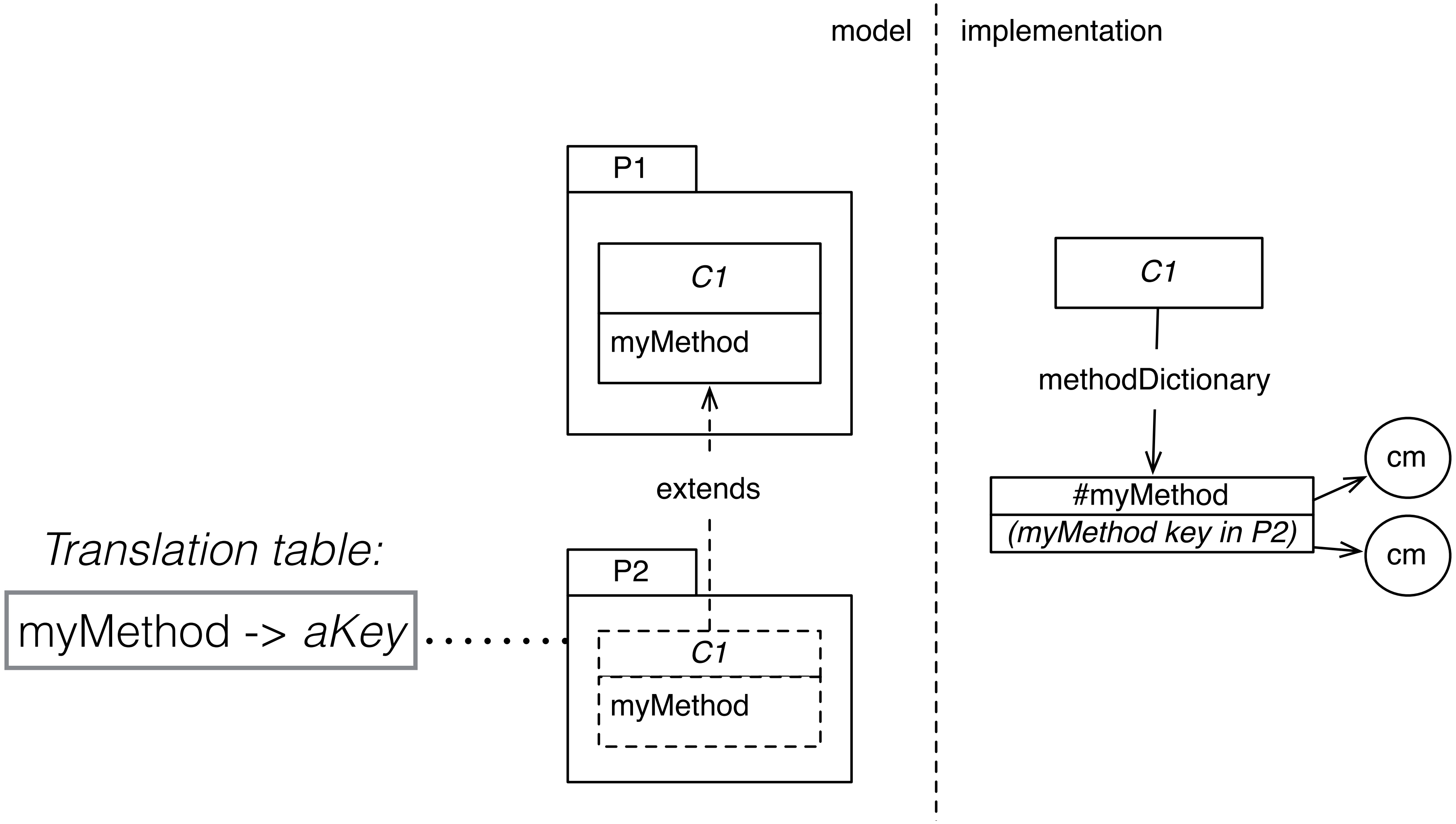
Implementation: Idea

Any object can be used during the lookup!

Lets split the concepts of:

- the name I give to a message (a *verb*)
- the object used during the lookup (a *key*)

Implementation



Want More Details?

Polymorphism

Dammit… I just broke polymorphism…

```
obj message
```

Cannot know the class of `obj` statically…

Polymorphism

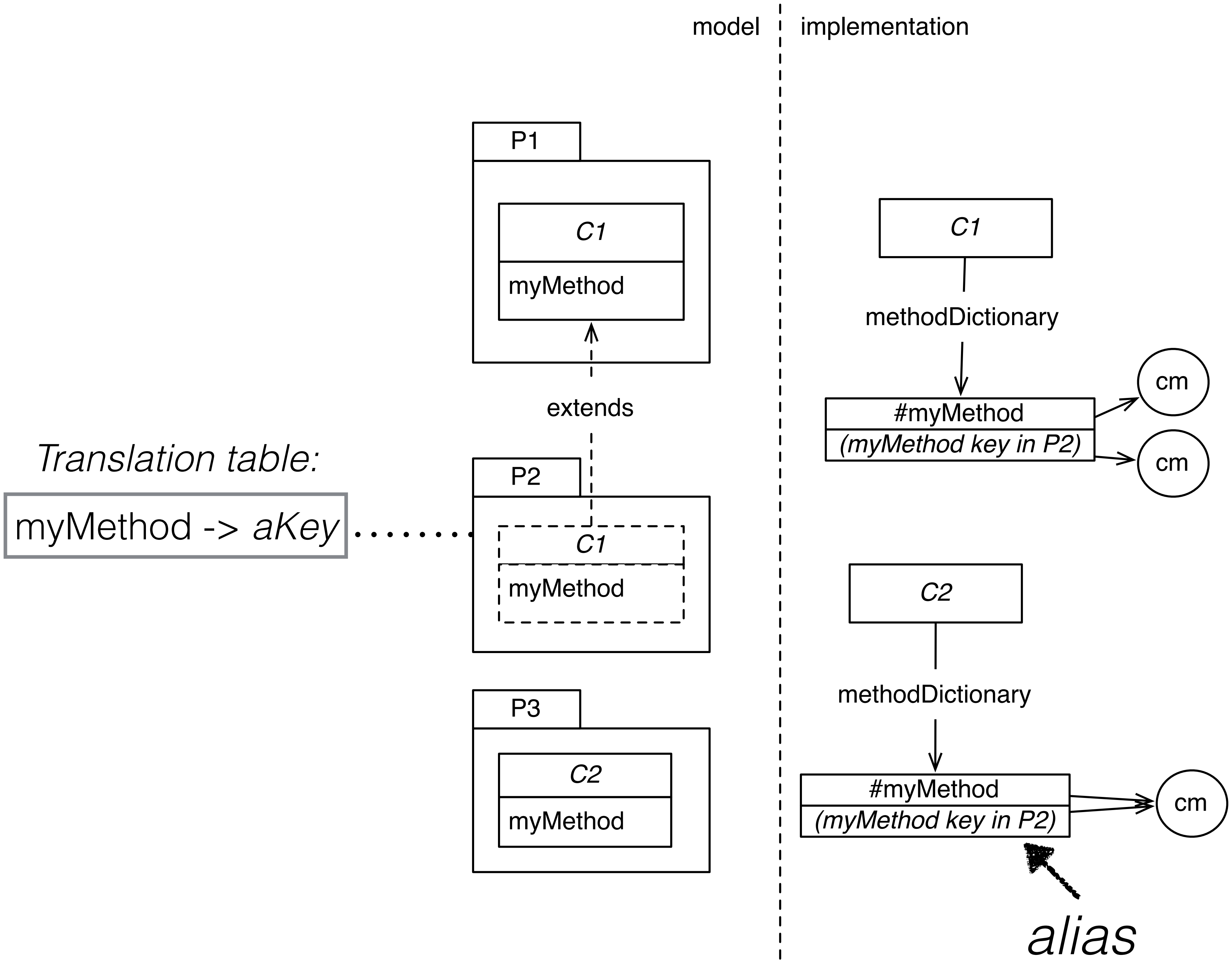
Solution 1: retry the lookup with original selector

-> Slow, doesn’t leverage the method lookup cache

Polymorphism

Solution 2: make *aliases* in method dictionaries

Polymorphism



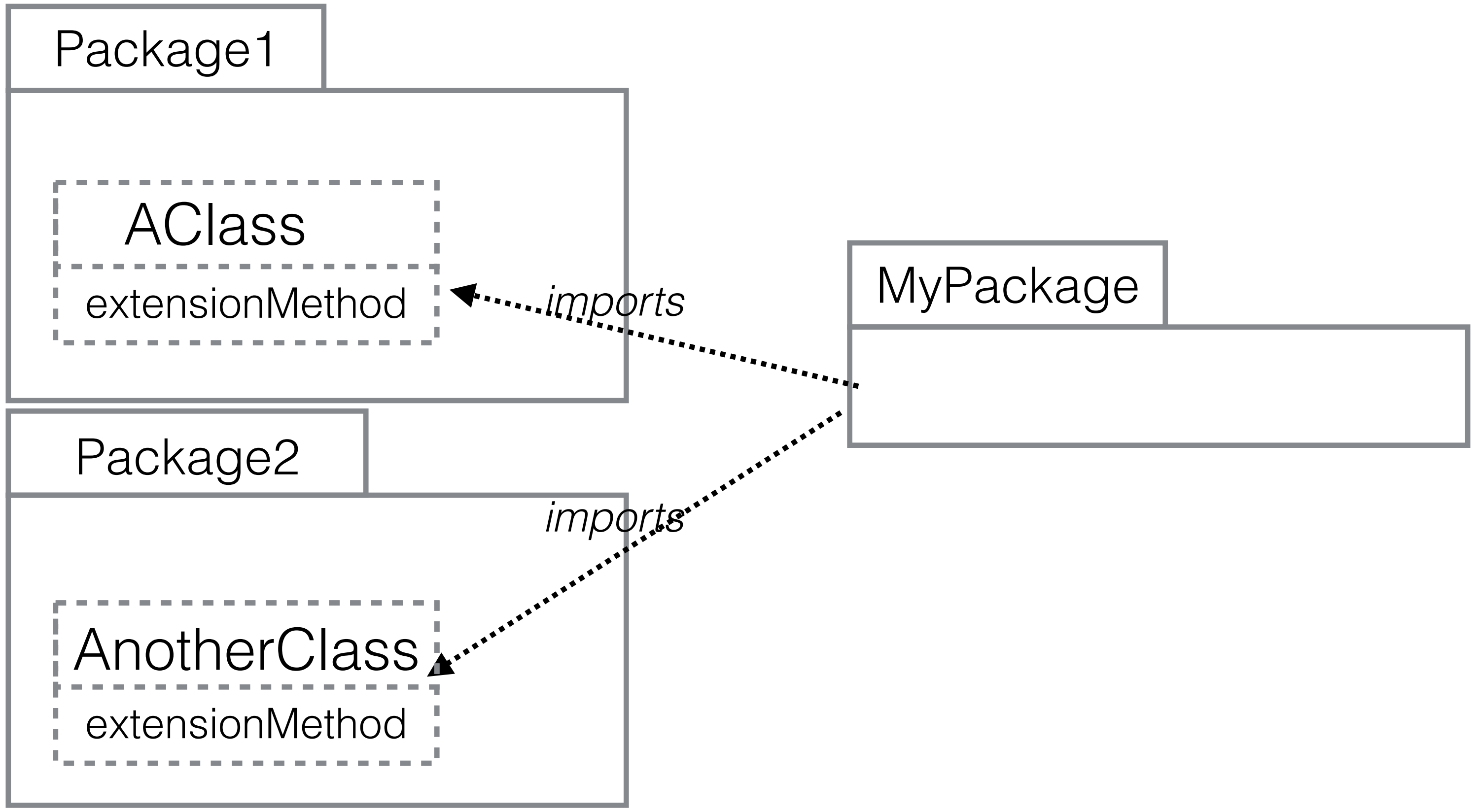
Polymorphism

When do we install aliases?

- Solution 2.1: Eagerly in all classes
  (but maybe some aliases will never be used)
- Solution 2.2: Lazily when a lookup fails

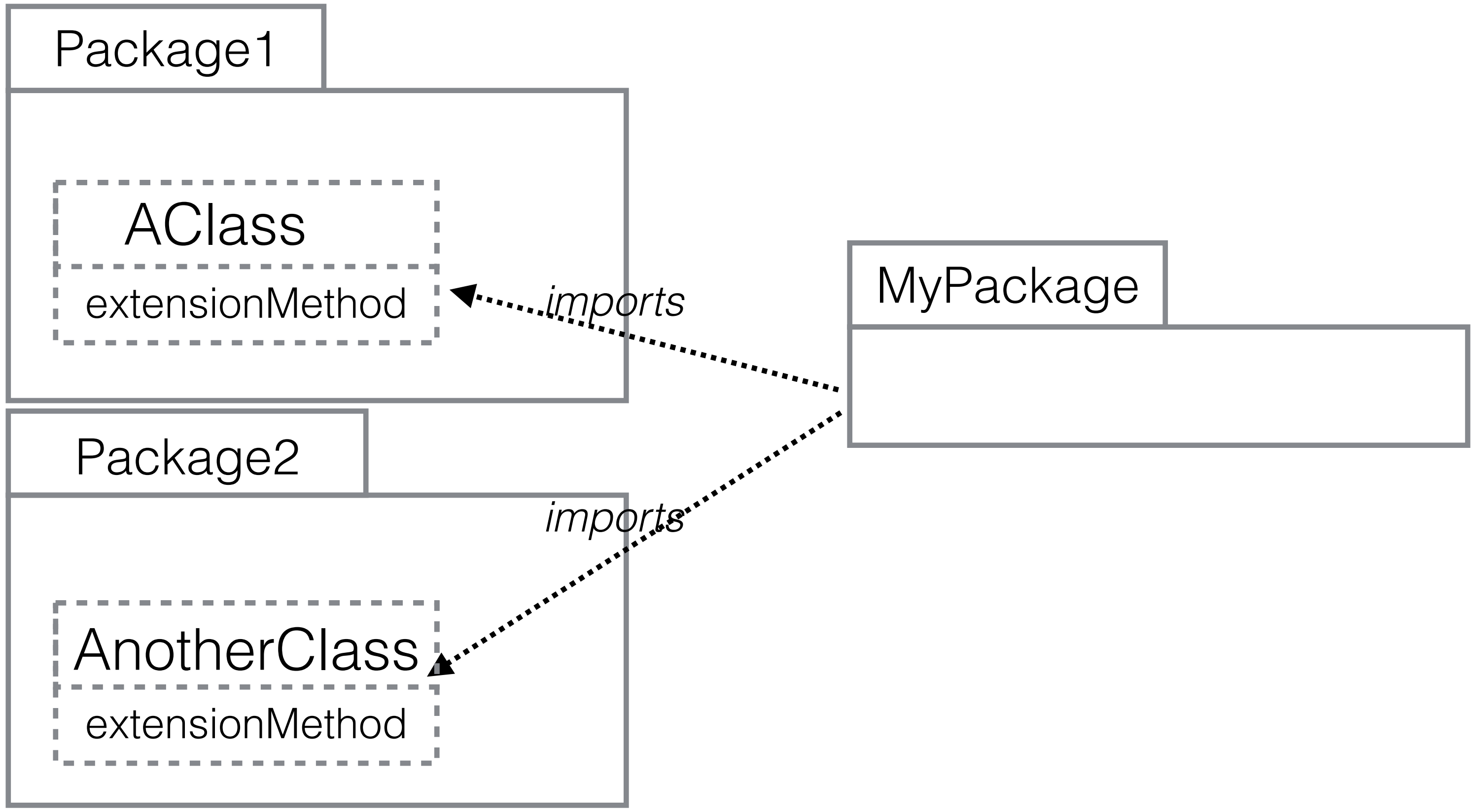
Want More Details?

Implementation: Merging



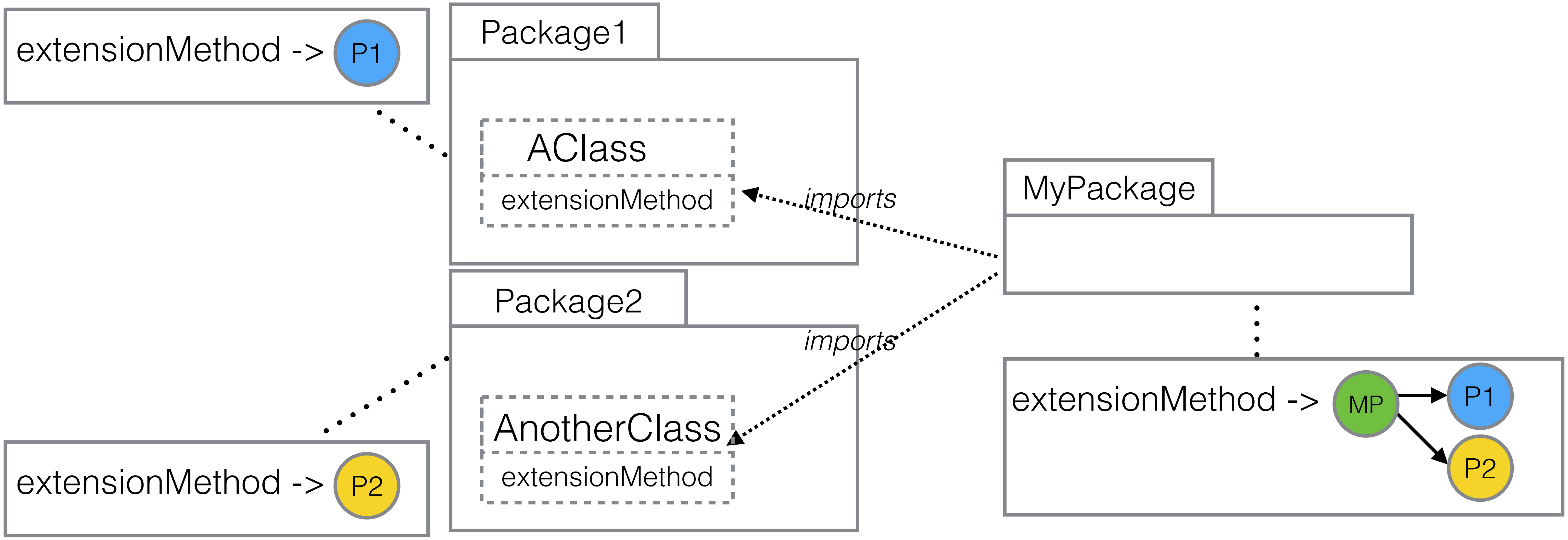
MyPackage imports two different extension methods with the same name

Implementation: Merging



What the key associated with the verb extensionMethod in MyPackage?

Implementation: Merging



When the lookup fails with MP, it ask the receiver if it understands P1 or P2 (no alias)

Method Visibility

Method Visibility

Splitting the concept of *selector* into *verb* and *key* also permits to implement support method visibility:

- private
- protected
- *others*

Method Visibility: Implementation

The implementation is way simpler than with local extension methods (no aliases, no merging)

Distinction between *object-sends* and *self-sends*:

`obj privateMethod` will always fail, even if `obj == self`

`self privateMethod` will succeed, self-sends have more authority

That’s all folks!!