Object Oriented Languages

Method Composition

The level of support for method composition varies between languages

- The `super` keyword in Smalltalk allows a method to call the next most specific method

Object Oriented Languages

Method Composition

The level of support for method composition varies between languages

- The `super` keyword in Smalltalk allows a method to call the next most specific method
- `call-next-method` in Lisp and `super()` in Python are similar

Object Oriented Languages

Method Composition

The level of support for method composition varies between languages

- The `super` keyword in Smalltalk allows a method to call the next most specific method
- `call-next-method` in Lisp and `super()` in Python are similar
- many languages only have composition in constructors

Object Oriented Languages

Method Composition

- Common Lisp also has *before*, *after* and *around* composition: they call it method *combination*. These add a method to a generic function that runs before, or after, or instead of the existing method

Object Oriented Languages

Method Composition

- Common Lisp also has *before*, *after* and *around* composition: they call it method *combination*. These add a method to a generic function that runs before, or after, or instead of the existing method
- You can use `call-next-method` to get at the original method from an around method

Object Oriented Languages

Method Composition

- Common Lisp also has *before*, *after* and *around* composition: they call it method *combination*. These add a method to a generic function that runs before, or after, or instead of the existing method
- You can use `call-next-method` to get at the original method from an around method
- Some languages allow arbitrary user-defined method composition: we shall talk about *metaobject protocols* soon

Object Oriented Languages

Method Composition

This is another big reason is why methods are different from functions: with method composition, methods need to know about other applicable methods, while functions live in isolation

Object Oriented Languages

Multiple Inheritance

Next there is another question to tackle: method (and attribute) selection when we have multiple inheritance in the class hierarchy

Object Oriented Languages

Multiple Inheritance

Next there is another question to tackle: method (and attribute) selection when we have multiple inheritance in the class hierarchy

This applies to both single and multiple dispatch method calls

Object Oriented Languages

Multiple Inheritance

Next there is another question to tackle: method (and attribute) selection when we have multiple inheritance in the class hierarchy

This applies to both single and multiple dispatch method calls

Take care here: MI is classes having multiple parents, while multiple dispatch is choosing a method using multiple arguments

Object Oriented Languages

Multiple Inheritance

Next there is another question to tackle: method (and attribute) selection when we have multiple inheritance in the class hierarchy

This applies to both single and multiple dispatch method calls

Take care here: MI is classes having multiple parents, while multiple dispatch is choosing a method using multiple arguments

Of course, we can have multiple dispatch with SI, and single dispatch with MI, and multiple dispatch with MI

Object Oriented Languages

Multiple Inheritance

In single inheritance with single dispatch the job is easy: if the class of the current object has a method defined, use it; else recurse to the parent class

Object Oriented Languages

Multiple Inheritance

In single inheritance with single dispatch the job is easy: if the class of the current object has a method defined, use it; else recurse to the parent class

But with MI you can inherit behaviour or structure from more than one parent

Object Oriented Languages

Multiple Inheritance

In single inheritance with single dispatch the job is easy: if the class of the current object has a method defined, use it; else recurse to the parent class

But with MI you can inherit behaviour or structure from more than one parent

When you have more than one parent, how do you choose which superclass to inherit from?

Object Oriented Languages

Multiple Inheritance

In single inheritance with single dispatch the job is easy: if the class of the current object has a method defined, use it; else recurse to the parent class

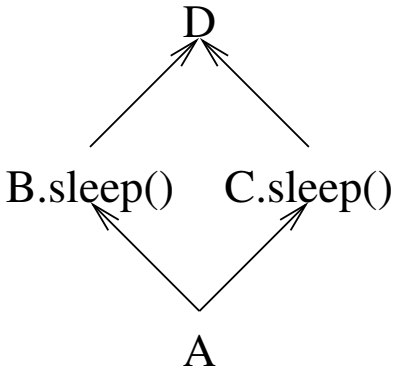
But with MI you can inherit behaviour or structure from more than one parent

When you have more than one parent, how do you choose which superclass to inherit from?

More generally: for method composition we need an order on *all* the superclasses

Object Oriented Languages

Multiple Inheritance

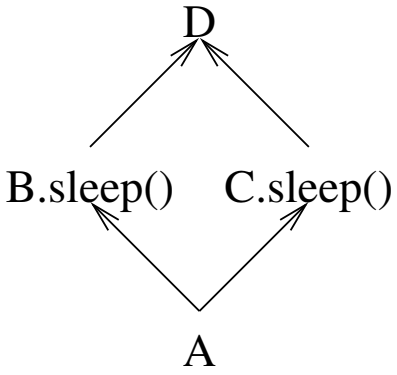


Inheritance diamond

Suppose a method `sleep` is defined in both `B` and `C`, but not `A`

Object Oriented Languages

Multiple Inheritance



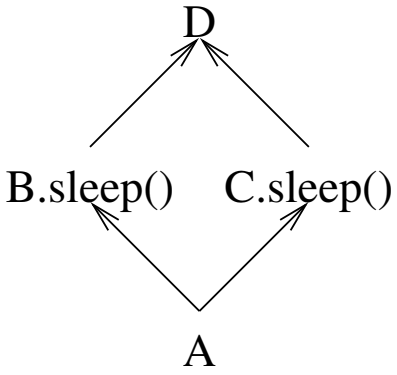
Inheritance diamond

Suppose a method `sleep` is defined in both `B` and `C`, but not `A`

If `sleep` is called with argument in class `A` should it use the method from `B` or `C`?

Object Oriented Languages

Multiple Inheritance



Inheritance diamond

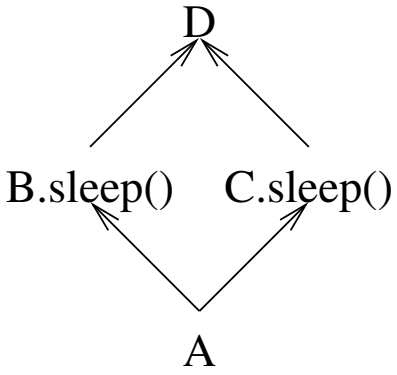
Suppose a method `sleep` is defined in both `B` and `C`, but not `A`

If `sleep` is called with argument in class `A` should it use the method from `B` or `C`?

`B`, perhaps, as that is on the left, and we read left-to-right?

Object Oriented Languages

Multiple Inheritance



Inheritance diamond

Suppose a method `sleep` is defined in both `B` and `C`, but not `A`

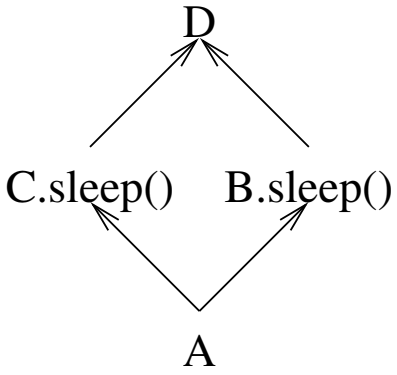
If `sleep` is called with argument in class `A` should it use the method from `B` or `C`?

`B`, perhaps, as that is on the left, and we read left-to-right?

But other people read right-to-left, and what if we had happened to draw the same hierarchy in a different way?

Object Oriented Languages

Multiple Inheritance



Inheritance diamond reversed

Suppose a method `sleep` is defined in both `B` and `C`, but not `A`

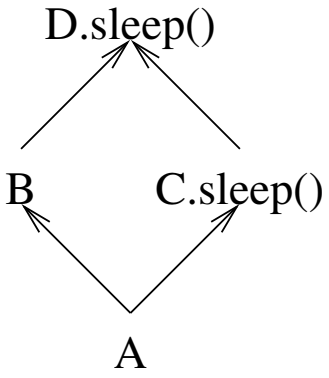
If `sleep` is called with argument in class `A` should it use the method from `B` or `C`?

`B`, perhaps, as that is on the left, and we read left-to-right?

But other people read right-to-left, and what if we had happened to draw the same hierarchy in a different way?

Object Oriented Languages

Multiple Inheritance

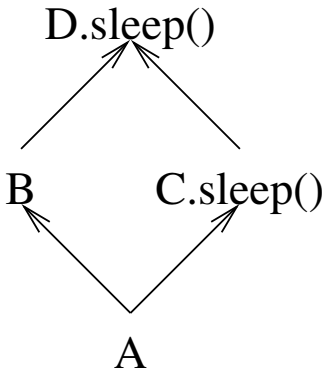


Inheriting from different “levels”

Or suppose `sleep` is only defined in `D` and `C`

Object Oriented Languages

Multiple Inheritance



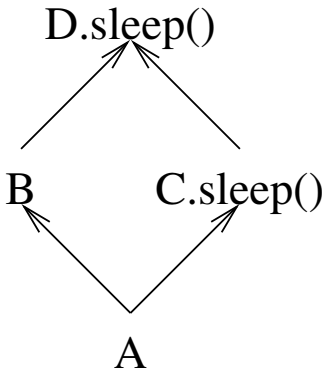
Inheriting from different “levels”

Or suppose `sleep` is only defined in `D` and `C`

Going up the hierarchy in a depth-first search on the left we get to `D` first; going up on the right we get to `C` first

Object Oriented Languages

Multiple Inheritance



Inheriting from different “levels”

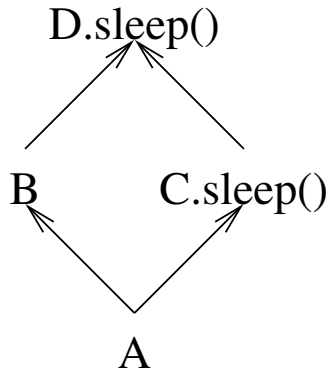
Or suppose `sleep` is only defined in `D` and `C`

Going up the hierarchy in a depth-first search on the left we get to `D` first; going up on the right we get to `C` first

Doing a breadth-first search, we find `C` first

Object Oriented Languages

Multiple Inheritance



Inheriting from different “levels”

Or suppose `sleep` is only defined in `D` and `C`

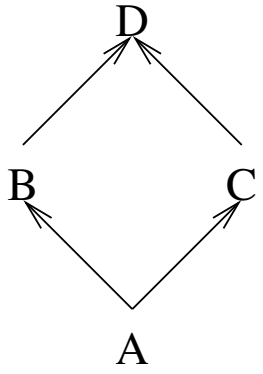
Going up the hierarchy in a depth-first search on the left we get to `D` first; going up on the right we get to `C` first

Doing a breadth-first search, we find `C` first

What should `A` do?

Object Oriented Languages

Multiple Inheritance

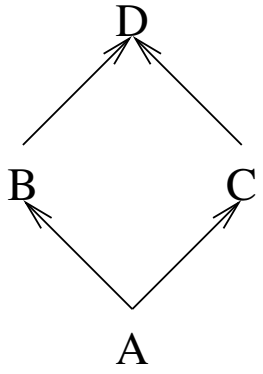


Inheritance diamond

This is called the *diamond problem*

Object Oriented Languages

Multiple Inheritance



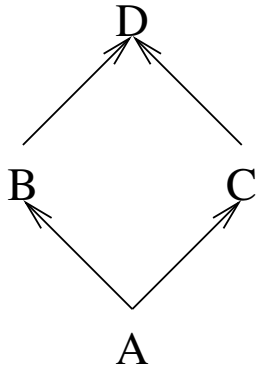
Inheritance diamond

This is called the *diamond problem*

When there is more than one candidate method to call, how does the compiler (or interpreter) choose which one?

Object Oriented Languages

Multiple Inheritance



Inheritance diamond

This is called the *diamond problem*

When there is more than one candidate method to call, how does the compiler (or interpreter) choose which one?

But, remember, the programmer also has to understand how a method is chosen

Object Oriented Languages

Multiple Inheritance

There have been many attempts to address this question

Object Oriented Languages

Multiple Inheritance

There have been many attempts to address this question

Every MI language needs a way of choosing, or forcing the programmer to choose

Object Oriented Languages

Multiple Inheritance

There have been many attempts to address this question

Every MI language needs a way of choosing, or forcing the programmer to choose

Some languages force you to disambiguate yourself, e.g., `D::sleep()`

Object Oriented Languages

Multiple Inheritance

There have been many attempts to address this question

Every MI language needs a way of choosing, or forcing the programmer to choose

Some languages force you to disambiguate yourself, e.g., `D::sleep()`

While many languages have a built-in algorithm to choose for you (see linearisation, below)

Object Oriented Languages

Multiple Inheritance

But does this built-in algorithmic choice reflect the expectations of the programmer?

Object Oriented Languages

Multiple Inheritance

But does this built-in algorithmic choice reflect the expectations of the programmer?

Usually yes in simple cases, but what about more complex hierarchies?

Object Oriented Languages

Multiple Inheritance

But does this built-in algorithmic choice reflect the expectations of the programmer?

Usually yes in simple cases, but what about more complex hierarchies?

The fact there are many linearisation algorithms tells us something!

Object Oriented Languages

Multiple Inheritance

For example, in simple cases, Common Lisp makes a choice by looking at how the classes were defined

Object Oriented Languages

Multiple Inheritance

If the definition was

```
(defclass D () ...)
(defclass B (D) ...)
(defclass C (D) ...)
(defclass A (B C) ...)
```

it might order the diamond of superclasses of `A` as `(A B C D)`.

Object Oriented Languages

Multiple Inheritance

If the definition was

```
(defclass D () ...)
(defclass B (D) ...)
(defclass C (D) ...)
(defclass A (B C) ...)
```

it might order the diamond of superclasses of `A` as `(A B C D)`.

This is a *linearisation* of the superclasses

Object Oriented Languages

Multiple Inheritance

If the definition was

```
(defclass D () ...)
(defclass B (D) ...)
(defclass C (D) ...)
(defclass A (B C) ...)
```

it might order the diamond of superclasses of `A` as `(A B C D)`.

This is a *linearisation* of the superclasses

And the resulting order `(A B C D)` is called the *class precedence list* (CPL) for `A`

Object Oriented Languages

Multiple Inheritance

If the definition was

```
(defclass D () ...)
(defclass B (D) ...)
(defclass C (D) ...)
(defclass A (B C) ...)
```

it might order the diamond of superclasses of `A` as `(A B C D)`.

This is a *linearisation* of the superclasses

And the resulting order `(A B C D)` is called the *class precedence list* (CPL) for `A`

Thus — for this order — a method defined in `B` is preferred over one defined in `C`

Object Oriented Languages

Multiple Inheritance

If the definition was

```
(defclass D () ...)
(defclass B (D) ...)
(defclass C (D) ...)
(defclass A (B C) ...)
```

it might order the diamond of superclasses of `A` as `(A B C D)`.

This is a *linearisation* of the superclasses

And the resulting order `(A B C D)` is called the *class precedence list* (CPL) for `A`

Thus — for this order — a method defined in `B` is preferred over one defined in `C`

And similarly for `B` vs. `D`

Object Oriented Languages

Multiple Inheritance

On the other hand, if we happened to define

```
(defclass D () ...)
(defclass B (D) ...)
(defclass C (D) ...)
(defclass A (C B) ...)
```

Common Lisp would create a CPL of `(A C B D)`

Object Oriented Languages

Multiple Inheritance

On the other hand, if we happened to define

```
(defclass D () ...)
(defclass B (D) ...)
(defclass C (D) ...)
(defclass A (C B) ...)
```

Common Lisp would create a CPL of `(A C B D)`

This makes the resolution of `B` versus `C` consistent with the (perhaps unconscious) choice of the programmer

Object Oriented Languages

Multiple Inheritance

On the other hand, if we happened to define

```
(defclass D () ...)
(defclass B (D) ...)
(defclass C (D) ...)
(defclass A (C B) ...)
```

Common Lisp would create a CPL of `(A C B D)`

This makes the resolution of `B` versus `C` consistent with the (perhaps unconscious) choice of the programmer

Remember this is a tiny example: in reality the code will be much more complicated

Object Oriented Languages

Multiple Inheritance

A class precedence list helps the language decide which method to use

Object Oriented Languages

Multiple Inheritance

A class precedence list helps the language decide which method to use

It will give us the *method resolution order* (MRO)

Object Oriented Languages

Multiple Inheritance

A class precedence list helps the language decide which method to use

It will give us the *method resolution order* (MRO)

Namely the ordering of the applicable methods so we (a) can pick the right method and (b) have an ordered list of methods for method composition

Object Oriented Languages

Multiple Inheritance

In object-receiver languages, usually the method chosen is the earliest found following the CPL

Object Oriented Languages

Multiple Inheritance

In object-receiver languages, usually the method chosen is the earliest found following the CPL

Thus if the CPL is `(A B C D)` and both `B` and `C` define `sleep`, then pick the method from `B`

Object Oriented Languages

Multiple Inheritance

In object-receiver languages, usually the method chosen is the earliest found following the CPL

Thus if the CPL is `(A B C D)` and both `B` and `C` define `sleep`, then pick the method from `B`

If just `D` and `C` define `sleep`, then pick the method from `C`

Object Oriented Languages

Multiple Inheritance

Or method*s* if we have method composition

Object Oriented Languages

Multiple Inheritance

Or method*s* if we have method composition

Again, this is why we need the whole CPL, not just a single class

Object Oriented Languages

Multiple Inheritance

Or method*s* if we have method composition

Again, this is why we need the whole CPL, not just a single class

It's not the whole story if we have multiple method dispatch as we have extra complication over multiple argument classes

Object Oriented Languages

Multiple Inheritance

Or method*s* if we have method composition

Again, this is why we need the whole CPL, not just a single class

It's not the whole story if we have multiple method dispatch as we have extra complication over multiple argument classes

With object receiver, the MRO is just the CPL; with multimethods calculating the MRO is harder (coming soon!)

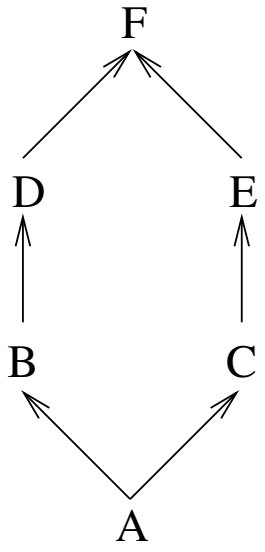
Object Oriented Languages

Multiple Inheritance

Computing a good CPL is not straightforward: what about `D` and `E` in

```
(defclass F () ...)
(defclass E (F) ...)
(defclass D (F) ...)
(defclass B (D) ...)
(defclass C (E) ...)
(defclass A (B C) ...)
```

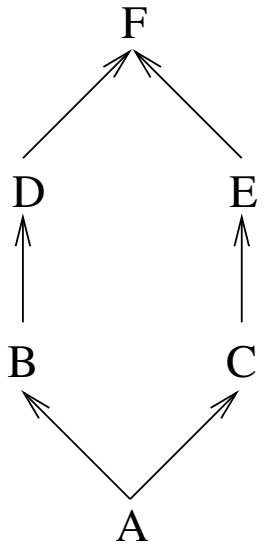
Object Oriented Languages



MI graph with no disambiguating definition

There is no disambiguating `defclass` to guide us to order `D` and `E`

Object Oriented Languages

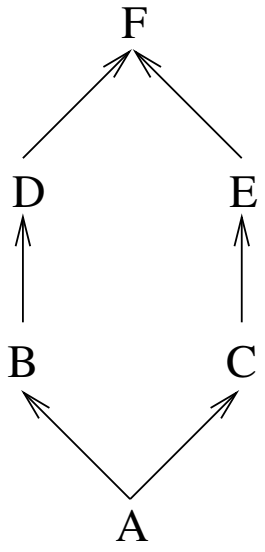


MI graph with no disambiguating definition

There is no disambiguating `defclass` to guide us to order `D` and `E`

We might want `D` before `E` as `B` is before `C`

Object Oriented Languages



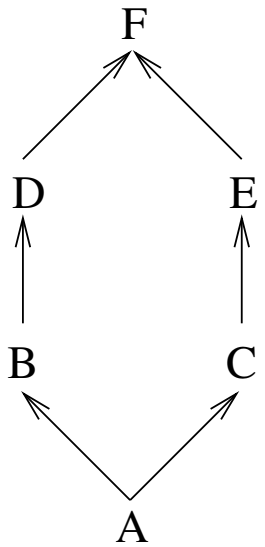
MI graph with no disambiguating definition

There is no disambiguating `defclass` to guide us to order `D` and `E`

We might want `D` before `E` as `B` is before `C`

Or not

Object Oriented Languages



MI graph with no disambiguating definition

There is no disambiguating `defclass` to guide us to order `D` and `E`

We might want `D` before `E` as `B` is before `C`

Or not

And do we want `D` before or after `C`?

Object Oriented Languages

This is the basic problem with MI: it is not clear, and different programmers may have different expectations of what should happen

Object Oriented Languages

This is the basic problem with MI: it is not clear, and different programmers may have different expectations of what should happen

The class definitions do not help in this example, so we need a little more help

Object Oriented Languages

This is the basic problem with MI: it is not clear, and different programmers may have different expectations of what should happen

The class definitions do not help in this example, so we need a little more help

We have two dimensions: left-right and up-down, and different people have different ideas (or different requirements) on which should be used to resolve the order

Object Oriented Languages

Multiple Inheritance

These ideas were first explored in Lisp, and different people made different choices, of course

Object Oriented Languages

Multiple Inheritance

These ideas were first explored in Lisp, and different people made different choices, of course

FLAVORS: do a depth-first traversal of the graph, keep the leftmost of any duplicates

Object Oriented Languages

Multiple Inheritance

These ideas were first explored in Lisp, and different people made different choices, of course

FLAVORS: do a depth-first traversal of the graph, keep the leftmost of any duplicates

The traversal is `A B D F C E F`, which becomes the CPL `(A B D F C E)`

Object Oriented Languages

Multiple Inheritance

These ideas were first explored in Lisp, and different people made different choices, of course

FLAVORS: do a depth-first traversal of the graph, keep the leftmost of any duplicates

The traversal is `A B D F C E F`, which becomes the CPL `(A B D F C E)`

LOOPS: do a depth-first traversal of the graph, keep the rightmost of any duplicates

Object Oriented Languages

Multiple Inheritance

These ideas were first explored in Lisp, and different people made different choices, of course

FLAVORS: do a depth-first traversal of the graph, keep the leftmost of any duplicates

The traversal is `A B D F C E F`, which becomes the CPL `(A B D F C E)`

LOOPS: do a depth-first traversal of the graph, keep the rightmost of any duplicates

The same traversal becomes the CPL `(A B D C E F)`

Object Oriented Languages

Multiple Inheritance

Neither are satisfactory algorithms

Object Oriented Languages

Multiple Inheritance

Neither are satisfactory algorithms

For example, FLAVORS has `F` before `C` in the CPL for `A` even though `C` is a subclass of `F`

Object Oriented Languages

Multiple Inheritance

Neither are satisfactory algorithms

For example, FLAVORS has `F` before `C` in the CPL for `A` even though `C` is a subclass of `F`

And both produce undesirable behaviour in complicated hierarchies

Object Oriented Languages

Multiple Inheritance

For example, if `S` has CPL with `T` before `U`, we might hope that a subclass `R` of `S` also has a consistent CPL with `T` and `U` in the same order

Object Oriented Languages

Multiple Inheritance

For example, if `S` has CPL with `T` before `U`, we might hope that a subclass `R` of `S` also has a consistent CPL with `T` and `U` in the same order

If the CPL for `S` is `(S ... T ... U ...)`, the CPL for `R` would be `(R ... T ... U ...)`

Object Oriented Languages

Multiple Inheritance

For example, if `S` has CPL with `T` before `U`, we might hope that a subclass `R` of `S` also has a consistent CPL with `T` and `U` in the same order

If the CPL for `S` is `(S ... T ... U ...)`, the CPL for `R` would be `(R ... T ... U ...)`

This would be a *monotonic* CPL: the CPL of a class is consistent with the CPL of its parents

Object Oriented Languages

Multiple Inheritance

Being monotonic is a desirable property as it agrees with intuition of the programmer on how inheritance should happen

Object Oriented Languages

Multiple Inheritance

Being monotonic is a desirable property as it agrees with intuition of the programmer on how inheritance should happen

But many linearisation algorithms don't guarantee that: they might give non-monotonic CPLs

Object Oriented Languages

Multiple Inheritance

All of LOOPS, FLAVORS and the more complex algorithm actually used by Common Lisp can produce non-monotonic CPLs, even on quite small examples

Object Oriented Languages

Multiple Inheritance

All of LOOPS, FLAVORS and the more complex algorithm actually used by Common Lisp can produce non-monotonic CPLs, even on quite small examples

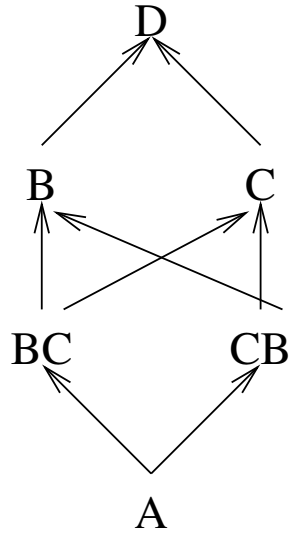
Exercise Look up the CLOS algorithm and find some non-monotonic examples

Object Oriented Languages

Multiple Inheritance

Exercise Think about

```
(defclass D () ...)
(defclass B (D) ...)
(defclass C (D) ...)
(defclass BC (B C) ...)
(defclass CB (C B) ...)
(defclass A (BC CB) ...)
```



A problematic MI graph

Object Oriented Languages

Multiple Inheritance

Many languages have moved to a linearisation algorithm called C3

Object Oriented Languages

Multiple Inheritance

Many languages have moved to a linearisation algorithm called C3

It is fairly easy to implement and is monotonic

Object Oriented Languages

Multiple Inheritance

Many languages have moved to a linearisation algorithm called C3

It is fairly easy to implement and is monotonic

Together with a few other desirable properties

Object Oriented Languages

Multiple Inheritance

Many languages have moved to a linearisation algorithm called C3

It is fairly easy to implement and is monotonic

Together with a few other desirable properties

It is now used in Python, Perl and several other MI languages

Object Oriented Languages

Multiple Inheritance

Many languages have moved to a linearisation algorithm called C3

It is fairly easy to implement and is monotonic

Together with a few other desirable properties

It is now used in Python, Perl and several other MI languages

Exercise Read about C3 linearization

Object Oriented Languages

Multiple Inheritance

Some examples of “typical” MI hierarchies, from “A Monotonic Superclass Linearization for Dylan”, Barrett et al., 1996:

	classes	MI joins
LispWorks	507	70
CLIM	842	184
database	38	4
emulator	571	205
proprietary	665	124
Watson	673	114

Feet

- Dylan: tries to shoot you in the foot like Scheme while enviously watching Java eat its lunch

Object Oriented Languages

Method Dispatch

We now know enough to talk about how to pick methods, that is, determine the method resolution order (MRO) for multimethods

Object Oriented Languages

Method Dispatch

We now know enough to talk about how to pick methods, that is, determine the method resolution order (MRO) for multimethods

We need to find the right method to call given a bunch of arguments

Object Oriented Languages

Method Dispatch

We now know enough to talk about how to pick methods, that is, determine the method resolution order (MRO) for multimethods

We need to find the right method to call given a bunch of arguments

This needs various bits of infrastructure to work

Object Oriented Languages

Method Dispatch

We need to know all the superclasses of the classes of the objects involved, thus we need to compute their CPLs using your favourite linearisation algorithm

Object Oriented Languages

Method Dispatch

We need to know all the superclasses of the classes of the objects involved, thus we need to compute their CPLs using your favourite linearisation algorithm

For example, the arguments `(4.0 99)`

Object Oriented Languages

Method Dispatch

We need to know all the superclasses of the classes of the objects involved, thus we need to compute their CPLs using your favourite linearisation algorithm

For example, the arguments `(4.0 99)`

The argument of 4.0 might have CPL
`(double float number object)`

Object Oriented Languages

Method Dispatch

We need to know all the superclasses of the classes of the objects involved, thus we need to compute their CPLs using your favourite linearisation algorithm

For example, the arguments `(4.0 99)`

The argument of 4.0 might have CPL
`(double float number object)`

While the argument of 99 might have CPL
`(int integer number object)`

Object Oriented Languages

Method Dispatch

If we call a generic function on arguments $(a_1, a_2, \ldots, a_n)$ we first need to find those methods on the GF that it makes sense to consider

Object Oriented Languages

Method Dispatch

If we call a generic function on arguments $(a_1, a_2, \ldots, a_n)$ we first need to find those methods on the GF that it makes sense to consider

A method is *applicable* to a call with arguments $(a_1, a_2, \ldots, a_n)$ if it is defined for classes $(A_1, A_2, \ldots, A_n)$ where for each i, the class of a_i is a subclass of A_i

Object Oriented Languages

Method Dispatch

So a method with *domain* `(integer number)` is applicable to a call with arguments `(23 42)` as these arguments have classes `(int int)`

Object Oriented Languages

Method Dispatch

So a method with *domain* `(integer number)` is applicable to a call with arguments `(23 42)` as these arguments have classes `(int int)`

Here, `int` is a subclass of `integer`, and `int` is a subclass of `number`

Object Oriented Languages

Method Dispatch

So a method with *domain* `(integer number)` is applicable to a call with arguments `(23 42)` as these arguments have classes `(int int)`

Here, `int` is a subclass of `integer`, and `int` is a subclass of `number`

But not applicable to a call with arguments `(4.0 99)` as `4.0` has class `double` which is not a subclass of `integer`

Object Oriented Languages

Method Dispatch

So a method with *domain* `(integer number)` is applicable to a call with arguments `(23 42)` as these arguments have classes `(int int)`

Here, `int` is a subclass of `integer`, and `int` is a subclass of `number`

But not applicable to a call with arguments `(4.0 99)` as `4.0` has class `double` which is not a subclass of `integer`

Nor a call `(4 "hello")`, even though 4, with class `int` is a subclass of `integer`, we see that `string` is not a subclass of `number`

Object Oriented Languages

Method Dispatch

Next, a method with domain $(A_1, A_2, \ldots, A_n)$ is *more specific* than a method with domain $(B_1, B_2, \ldots, B_n)$ for the arguments $(a_1, a_2, \ldots, a_n)$ if

1. they are both applicable to $(a_1, a_2, \ldots, a_n)$ and
2. there is an k with $A_i = B_i$ for $i < k$, but
3. A_k appears before B_k in the CPL for argument a_k

Object Oriented Languages

Method Dispatch

In simpler terms, one method is more specific than another if the class in the first place they differ is more specific

Object Oriented Languages

Method Dispatch

In simpler terms, one method is more specific than another if the class in the first place they differ is more specific

This is a kind of alphabetical ordering, where the alphabet is specified by the CPL

Object Oriented Languages

Method Dispatch

In simpler terms, one method is more specific than another if the class in the first place they differ is more specific

This is a kind of alphabetical ordering, where the alphabet is specified by the CPL

In a normal alphabetic order, we put “can” before “cat” as this is determined by the first place the words differ: namely “n” comes before “t”

Object Oriented Languages

Method Dispatch

In simpler terms, one method is more specific than another if the class in the first place they differ is more specific

This is a kind of alphabetical ordering, where the alphabet is specified by the CPL

In a normal alphabetic order, we put “can” before “cat” as this is determined by the first place the words differ: namely “n” comes before “t”

We naturally extend to, say, “cat1” before “cat3” as “1” comes before “3”. But now there is more than one order in play

Object Oriented Languages

Method Dispatch

In simpler terms, one method is more specific than another if the class in the first place they differ is more specific

This is a kind of alphabetical ordering, where the alphabet is specified by the CPL

In a normal alphabetic order, we put “can” before “cat” as this is determined by the first place the words differ: namely “n” comes before “t”

We naturally extend to, say, “cat1” before “cat3” as “1” comes before “3”. But now there is more than one order in play

Or even “c♡9” before “c♣1” if “♡” is before “♣”. Each character position has its own alphabet

Object Oriented Languages

Method Dispatch

This is the situation for method ordering: each argument position has its own “alphabetic order”, with the order being the CPL for the object in that position

Object Oriented Languages

Method Dispatch

This is the situation for method ordering: each argument position has its own “alphabetic order”, with the order being the CPL for the object in that position

Example. Calling a method with arguments `(1 1.0)` of classes `int` and `double`

Object Oriented Languages

Method Dispatch

This is the situation for method ordering: each argument position has its own “alphabetic order”, with the order being the CPL for the object in that position

Example. Calling a method with arguments `(1 1.0)` of classes `int` and `double`

The CPL for the first argument is `(int integer number object)`

Object Oriented Languages

Method Dispatch

This is the situation for method ordering: each argument position has its own “alphabetic order”, with the order being the CPL for the object in that position

Example. Calling a method with arguments `(1 1.0)` of classes `int` and `double`

The CPL for the first argument is `(int integer number object)`

The CPL for the second argument is `(double float number object)`

Object Oriented Languages

Method Dispatch

A method with domain `(integer float)` is more specific than one with domain `(integer number)`

Object Oriented Languages

Method Dispatch

A method with domain `(integer float)` is more specific than one with domain `(integer number)`

A method with domain `(int object)` is more specific than one with domain `(integer double)`

Object Oriented Languages

Method Dispatch

A method with domain `(integer float)` is more specific than one with domain `(integer number)`

A method with domain `(int object)` is more specific than one with domain `(integer double)`

Just as “cup” is before “dog”: even though the second argument is very late in the alphabet, the first argument prevails

Object Oriented Languages

Method Dispatch

A method with domain `(integer float)` is more specific than one with domain `(integer number)`

A method with domain `(int object)` is more specific than one with domain `(integer double)`

Just as “cup” is before “dog”: even though the second argument is very late in the alphabet, the first argument prevails

A method with domain `(float float)` is not applicable for those arguments unless the language allows automatic coercion of types: a huge extra complication that we shall ignore