Object Oriented Languages

Method Dispatch

For simple OO systems selection of the correct method to apply in a given circumstance (*method dispatch*) is fairly easy

For simple OO systems selection of the correct method to apply in a given circumstance (*method dispatch*) is fairly easy

To be definite in the following we shall assume a class centred system

Object Oriented Languages

Method Dispatch

For simple OO systems selection of the correct method to apply in a given circumstance (*method dispatch*) is fairly easy

To be definite in the following we shall assume a class centred system

For `obj.meth()` we (i.e., the lookup mechanism in the compiler or interpreter) look in the object's class to see if there is an applicable method; if not we look to the class's superclass

Object Oriented Languages
Method Dispatch

For simple OO systems selection of the correct method to apply in a given circumstance (*method dispatch*) is fairly easy

To be definite in the following we shall assume a class centred system

For `obj.meth()` we (i.e., the lookup mechanism in the compiler or interpreter) look in the object's class to see if there is an applicable method; if not we look to the class's superclass

Repeat until we find an applicable method, or we run out of superclasses, when we report "no applicable method"

In this context “applicable” means “of the given name” and suitable arguments

In this context “applicable” means “of the given name” and suitable arguments

So `obj.foo(42)` looks for methods with the name `foo` that can be applied to an integer

Object Oriented Languages

Aside

In some languages, mostly those with static type hierarchies, e.g., Java and C++, the method can be determined and chosen at *compile* time as the class hierarchy is determined and fixed at compile time

Object Oriented Languages

Aside

In some languages, mostly those with static type hierarchies, e.g., Java and C++, the method can be determined and chosen at *compile* time as the class hierarchy is determined and fixed at compile time

Meaning no lookup overhead at runtime: the method has already been selected by the compiler and is directly called with no more ado

Object Oriented Languages

Aside

In some languages, mostly those with static type hierarchies, e.g., Java and C++, the method can be determined and chosen at *compile* time as the class hierarchy is determined and fixed at compile time

Meaning no lookup overhead at runtime: the method has already been selected by the compiler and is directly called with no more ado

Other languages, mostly those with dynamic types, e.g., JavaScript and Lisp, the method can only be chosen at *runtime* as the class or object relationships may change during the running of the program

Object Oriented Languages

Aside

So calling a method in such languages has the extra overhead of determining which is the correct method before before we can execute it

Object Oriented Languages

Aside

So calling a method in such languages has the extra overhead of determining which is the correct method before before we can execute it

Meaning it’s a bit slower to run

Object Oriented Languages

Aside

So calling a method in such languages has the extra overhead of determining which is the correct method before before we can execute it

Meaning it’s a bit slower to run

Good compilers/interpreters have many tricks to reduce the dispatch overhead, e.g., method caching

Object Oriented Languages

Aside

So calling a method in such languages has the extra overhead of determining which is the correct method before before we can execute it

Meaning it’s a bit slower to run

Good compilers/interpreters have many tricks to reduce the dispatch overhead, e.g., method caching

This is trading dynamic behaviour against speed of execution

If there are generic functions or multiple inheritance we have to work a bit harder

Object Oriented Languages

Method Dispatch

If there are generic functions or multiple inheritance we have to work a bit harder

Essentially we make a list of all the applicable methods from the arguments' classes and their superclasses, sort them into some useful order, then use the first in the list

Object Oriented Languages

Method Dispatch

If there are generic functions or multiple inheritance we have to work a bit harder

Essentially we make a list of all the applicable methods from the arguments' classes and their superclasses, sort them into some useful order, then use the first in the list

In principle easy, but a lot of detail in reality

Object Oriented Languages

Method Dispatch

This needs various bits of infrastructure to work

This needs various bits of infrastructure to work

We need to know all the superclasses of the class of an object: the *class precedence list* of an argument in a method call is a list of all the superclasses starting with the class of the argument

Object Oriented Languages

Method Dispatch

This needs various bits of infrastructure to work

We need to know all the superclasses of the class of an object: the *class precedence list* of an argument in a method call is a list of all the superclasses starting with the class of the argument

This is ordered, typically with the closest classes first, i.e., if `A` is a subclass of `B` which is a subclass of `C`, the list will start `(A B C ...)`

This needs various bits of infrastructure to work

We need to know all the superclasses of the class of an object: the *class precedence list* of an argument in a method call is a list of all the superclasses starting with the class of the argument

This is ordered, typically with the closest classes first, i.e., if `A` is a subclass of `B` which is a subclass of `C`, the list will start `(A B C ...)`

So, for example, an argument of 42 might have CPL `(<fpi> <integer> <number> <object>)`

There will be need to make some choices in the case of multiple inheritance, where there is no clear “closer” class: if `A` inherits directly from both `B` and `C`, do we want `(A B C ...)` or `(A C B ...)`?

There will be need to make some choices in the case of multiple inheritance, where there is no clear “closer” class: if `A` inherits directly from both `B` and `C`, do we want `(A B C ...)` or `(A C B ...)`?

More on this in a moment

Object Oriented Languages

Method Dispatch

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

If we call a generic function on arguments $(a_1, a_2, \ldots)$ 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 $(a_1, a_2, \ldots)$ if it is defined for classes $(A_1, A_2, \ldots)$ where for each i, the class of a_i is a subclass of A_i

Object Oriented Languages

Method Dispatch

If we call a generic function on arguments $(a_1, a_2, \ldots)$ 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 $(a_1, a_2, \ldots)$ if it is defined for classes $(A_1, A_2, \ldots)$ where for each i, the class of a_i is a subclass of A_i

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

If we call a generic function on arguments $(a_1, a_2, \ldots)$ 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 $(a_1, a_2, \ldots)$ if it is defined for classes $(A_1, A_2, \ldots)$ where for each i, the class of a_i is a subclass of A_i

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

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

Object Oriented Languages

Method Dispatch

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 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

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

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 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 alphabetic order in play

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 alphabetic order in play

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

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

Example. Calling a method with arguments `(1 1.0)` of classes `<fpi>` and `<double-float>`

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

Example. Calling a method with arguments `(1 1.0)` of classes `<fpi>` and `<double-float>`

The CPL for the first argument is `(<fpi> <integer> <number> <object>)`

This is the situation for method ordering: each argument position has its own "alphabet", with order specified by the CPL for the object in that position

Example. Calling a method with arguments `(1 1.0)` of classes `<fpi>` and `<double-float>`

The CPL for the first argument is `(<fpi> <integer> <number> <object>)`

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

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 (`<fpi> <object>`) is more specific than one with domain (`<integer> <double-float>`)

Object Oriented Languages

Method Dispatch

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

A method with domain `(<fpi> <object>)` is more specific than one with domain `(<integer> <double-float>)`

Just as “cup” is before “dog”: even though the second argument is very unspecific, 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 `(<fpi> <object>)` is more specific than one with domain `(<integer> <double-float>)`

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

A method with domain `(<float> <float>)` is not applicable unless the language allows automatic coercion of types: a huge extra complication

So the way to choose a method for a given set of arguments is

1. find all the applicable methods
2. find the CPLs for each argument
3. sort the methods in decreasing order of specificity according to the CPLs of the arguments
4. take the first (most specific) in the list

Object Oriented Languages

Method Dispatch

So the way to choose a method for a given set of arguments is

1. find all the applicable methods
2. find the CPLs for each argument
3. sort the methods in decreasing order of specificity according to the CPLs of the arguments
4. take the first (most specific) in the list

The sorted method list is useful for when we want to be more inventive on using methods

So the way to choose a method for a given set of arguments is

1. find all the applicable methods
2. find the CPLs for each argument
3. sort the methods in decreasing order of specificity according to the CPLs of the arguments
4. take the first (most specific) in the list

The sorted method list is useful for when we want to be more inventive on using methods

Note this reduces to what we expect in an object-receiver language that has only a single object dispatch on

This dispatch calculation will be done either at compile time (for a fixed class hierarchy) meaning no run-time overhead

Object Oriented Languages

Method Dispatch

This dispatch calculation will be done either at compile time (for a fixed class hierarchy) meaning no run-time overhead

Or at run-time, meaning some considerable execution overhead

This dispatch calculation will be done either at compile time (for a fixed class hierarchy) meaning no run-time overhead

Or at run-time, meaning some considerable execution overhead

If clever tricks are not employed

Object Oriented Languages

Method Dispatch

This dispatch calculation will be done either at compile time (for a fixed class hierarchy) meaning no run-time overhead

Or at run-time, meaning some considerable execution overhead

If clever tricks are not employed

For example, a lot of effort has been put into JavaScript on precisely this point

Now, we usually want more specific methods to override (aka *specialise*) less specific methods, but sometimes we want *method composition*

Now, we usually want more specific methods to override (aka *specialise*) less specific methods, but sometimes we want *method composition*

Suppose we have a Java class `B` that extends (is derived from; is a subclass of) `A`

Object Oriented Languages

Method Composition

Now, we usually want more specific methods to override (aka *specialise*) less specific methods, but sometimes we want *method composition*

Suppose we have a Java class `B` that extends (is derived from; is a subclass of) `A`

When making an instance of `B`, a constructor method for `B` does not *replace* (override) the constructor method for `A`, but *both* are called: first `A`'s then `B`'s

Object Oriented Languages

Method Composition

Now, we usually want more specific methods to override (aka *specialise*) less specific methods, but sometimes we want *method composition*

Suppose we have a Java class `B` that extends (is derived from; is a subclass of) `A`

When making an instance of `B`, a constructor method for `B` does not *replace* (override) the constructor method for `A`, but *both* are called: first `A`'s then `B`'s

In this case, a more specific method does not override a less specific one, but is *composed* with it

Object Oriented Languages

Method Composition

Similarly C++ has *destructors* that get called when an object is deleted, and they are called in the *opposite order*, `B`'s then `A`'s

Similarly C++ has *destructors* that get called when an object is deleted, and they are called in the *opposite order*, `B`'s then `A`'s

In these cases the composition is to run both methods, in an appropriate order

Object Oriented Languages

Method Composition

Other languages allow other kinds of composition for general methods

Object Oriented Languages

Method Composition

Other languages allow other kinds of composition for general methods

- The `super` keyword in Smalltalk and Java allows a method to call the next most specific method: this is why we need the complete sorted applicable method list

Object Oriented Languages

Method Composition

Other languages allow other kinds of composition for general methods

- The `super` keyword in Smalltalk and Java allows a method to call the next most specific method: this is why we need the complete sorted applicable method list
- `call-next-method` in Lisp is similar

Object Oriented Languages

Method Composition

Other languages allow other kinds of composition for general methods

- The `super` keyword in Smalltalk and Java allows a method to call the next most specific method: this is why we need the complete sorted applicable method list
- `call-next-method` in Lisp is similar
- 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

Other languages allow other kinds of composition for general methods

- The `super` keyword in Smalltalk and Java allows a method to call the next most specific method: this is why we need the complete sorted applicable method list
- `call-next-method` in Lisp is similar
- 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
- Some languages allow arbitrary user-defined method composition: we shall talk about *metaobject protocols* soon

Object Oriented Languages

Method Composition

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

We have yet to tackle one more question: method selection when we have multiple inheritance in the class hierarchy

We have yet to tackle one more question: method 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

We have yet to tackle one more question: method selection when we have multiple inheritance in the class hierarchy

This applies to both single and multiple dispatch method calls

The basic idea of MI is that you can inherit behaviour or structure from more than one parent

We have yet to tackle one more question: method selection when we have multiple inheritance in the class hierarchy

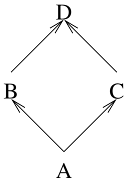
This applies to both single and multiple dispatch method calls

The basic idea of MI is that you can inherit behaviour or structure from more than one parent

But when you have more than one parent, how do you order the superclasses when determining the CPL?

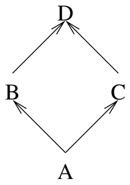
Object Oriented Languages

Multiple Inheritance



Object Oriented Languages

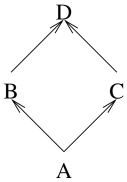
Multiple Inheritance



A method is called with argument in class `A`: should the CPL be `(A B C D)` or `(A C B D)`?

Object Oriented Languages

Multiple Inheritance

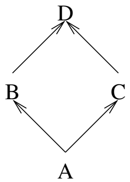


A method is called with argument in class `A`: should the CPL be `(A B C D)` or `(A C B D)`?

Or something else entirely?

Object Oriented Languages

Multiple Inheritance



A method is called with argument in class `A`: should the CPL be `(A B C D)` or `(A C B D)`?

Or something else entirely?

In simple cases, the choice can be made by looking at how the classes were defined

If the definition was

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

the CPL might be `(A B C D)`

On the other hand, for

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

the CPL might be `(A C B D)`

If the definition was

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

the CPL might be `(A B C D)`

On the other hand, for

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

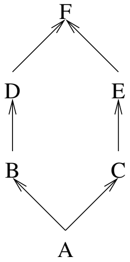
the CPL might be `(A C B D)`

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

But what about `D` and `E` in

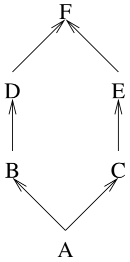
```
(defclass F () ...)
(defclass E (F) ...)
(defclass D (F) ...)
(defclass B (D) ...)
(defclass C (E) ...)
(defclass A (B C) ...)
```

Object Oriented Languages



There is no disambiguating `defclass` to guide us

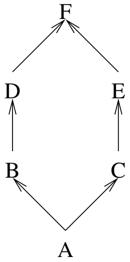
Object Oriented Languages



There is no disambiguating `defclass` to guide us

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

Object Oriented Languages

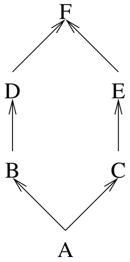


There is no disambiguating `defclass` to guide us

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

Or not

Object Oriented Languages



There is no disambiguating `defclass` to guide us

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

The class definitions do not help here, so we need a little more help

Object Oriented Languages

The class definitions do not help here, so we need a little more help

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

Varieties of Lisp made different choices, of course

Object Oriented Languages

Multiple Inheritance

Varieties of Lisp made different choices, of course

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

Varieties of Lisp 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

Varieties of Lisp 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

Varieties of Lisp made different choices, of course

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

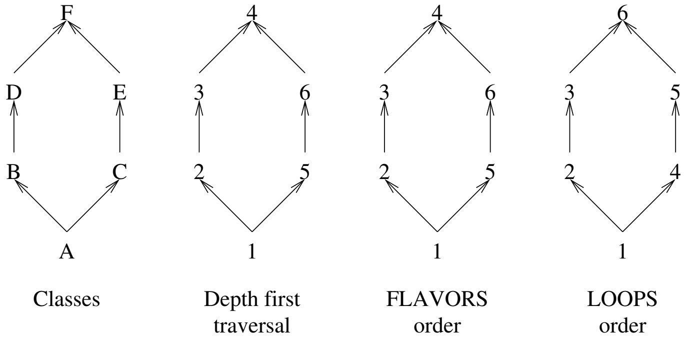
The traversal is `A B D F C E F`, which becomes the CPL `(A B D F C E)`

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



The position of each class in the linearised CPL

Neither are satisfactory, producing *unstable* behaviour in complicated hierarchies: small changes in the class definitions can cause large changes in the CPLs produced

Neither are satisfactory, producing *unstable* behaviour in complicated hierarchies: small changes in the class definitions can cause large changes in the CPLs produced

The solution adopted by Common Lisp is more complex, but has a more stable behaviour

Neither are satisfactory, producing *unstable* behaviour in complicated hierarchies: small changes in the class definitions can cause large changes in the CPLs produced

The solution adopted by Common Lisp is more complex, but has a more stable behaviour

It tries to keep the CPL locally consistent, using the order in the class definitions

Neither are satisfactory, producing *unstable* behaviour in complicated hierarchies: small changes in the class definitions can cause large changes in the CPLs produced

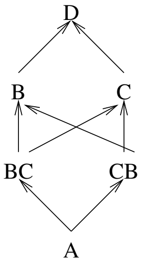
The solution adopted by Common Lisp is more complex, but has a more stable behaviour

It tries to keep the CPL locally consistent, using the order in the class definitions

Exercise. Look it up and try it on the examples above

Exercise. Think about

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



Object Oriented Languages

Multiple Inheritance

Other languages do other things. In C++

```
class D { ... }
class B: public D { ... }
class C: public D { ... }
class A: public B, public C { ... }
```

has the additional peculiarity that class `A` contains *two* copies of `D`, one via `B` and one via `C`

Object Oriented Languages

Multiple Inheritance

Other languages do other things. In C++

```
class D { ... }
class B: public D { ... }
class C: public D { ... }
class A: public B, public C { ... }
```

has the additional peculiarity that class `A` contains *two* copies of `D`, one via `B` and one via `C`

This is because occasionally we want two copies

Object Oriented Languages

Multiple Inheritance

Other languages do other things. In C++

```
class D { ... }
class B: public D { ... }
class C: public D { ... }
class A: public B, public C { ... }
```

has the additional peculiarity that class `A` contains *two* copies of `D`, one via `B` and one via `C`

This is because occasionally we want two copies

An `IOstream` inherits from both `Istream` and `Ostream`, which both inherit from `Stream`: we might want separate file pointers for input and output

Object Oriented Languages

Multiple Inheritance

If we want only a single copy, C++ requires *virtual inheritance*

```
class B: public virtual D { ... }
class C: public virtual D { ... }
class A: public B, public C { ... }
```

If we want only a single copy, C++ requires *virtual inheritance*

```
class B: public virtual D { ... }
class C: public virtual D { ... }
class A: public B, public C { ... }
```

Now the single copy of `D` is inherited by `A`

If we want only a single copy, C++ requires *virtual inheritance*

```
class B: public virtual D { ... }
class C: public virtual D { ... }
class A: public B, public C { ... }
```

Now the single copy of `D` is inherited by `A`

This is the most common usage, so should have been the default!

If we want only a single copy, C++ requires *virtual inheritance*

```
class B: public virtual D { ... }
class C: public virtual D { ... }
class A: public B, public C { ... }
```

Now the single copy of `D` is inherited by `A`

This is the most common usage, so should have been the default!

Exercise. Find out how C++ (and other MI languages) address the CPL linearisation issue

Java avoids the complexities of MI by only supporting *interfaces*

Java avoids the complexities of MI by only supporting *interfaces*

This is a very limited version of MI where no behaviour is inherited, but just requires the class to implement the required behaviour somehow

Java avoids the complexities of MI by only supporting *interfaces*

This is a very limited version of MI where no behaviour is inherited, but just requires the class to implement the required behaviour somehow

An interface is a list of method names, but no implementation, i.e., no code to go with the names

So, for example the interface (*not* class) `Istream` might name methods like `read` and `get_file_position`

So, for example the interface (*not* class) `Istream` might name methods like `read` and `get_file_position`

And the interface `Ostream` might name methods like `write` and `get_file_position`

So, for example the interface (*not* class) `Istream` might name methods like `read` and `get_file_position`

And the interface `Ostream` might name methods like `write` and `get_file_position`

And then we might have

```
class IOstream extends Stream
               implements Istream, Ostream {
...
}
```

Object Oriented Languages

Multiple Inheritance

So, for example the interface (*not* class) `Istream` might name methods like `read` and `get_file_position`

And the interface `Ostream` might name methods like `write` and `get_file_position`

And then we might have

```
class IOstream extends Stream
                implements Istream, Ostream {
...
}
```

The class `IOstream` must implement (directly or inherited from `Stream`) all the methods mentioned in the definitions of interfaces `Istream` and `Ostream`

So, for this example, `IOstream` must implement `read`, `write` and `get_file_position`

Object Oriented Languages

Multiple Inheritance

So, for this example, `IOstream` must implement `read`, `write` and `get_file_position`

A class can derive from multiple interfaces, but not multiple classes

So, for this example, `IOstream` must implement `read`, `write` and `get_file_position`

A class can derive from multiple interfaces, but not multiple classes

There is no possibility of inheriting multiple methods of the same name, as the class can still only inherit a method from at most one parent class—and nothing from the interfaces

So, for this example, `IOstream` must implement `read`, `write` and `get_file_position`

A class can derive from multiple interfaces, but not multiple classes

There is no possibility of inheriting multiple methods of the same name, as the class can still only inherit a method from at most one parent class—and nothing from the interfaces

There is no problem with being told more than once that a class needs to implement a method of a given name

An interface is more like a list of requirements of a class than inheriting things

An interface is more like a list of requirements of a class than inheriting things

Interfaces provide all of the MI functionality that most people need

An interface is more like a list of requirements of a class than inheriting things

Interfaces provide all of the MI functionality that most people need

Exercise. Go (Golang) also has interfaces. Read about them

Object Oriented Languages

Multiple Inheritance

Lisp and other languages like Perl and Python have a similar concept called a *mixin*

Object Oriented Languages

Multiple Inheritance

Lisp and other languages like Perl and Python have a similar concept called a *mixin*

The name was taken in analogy with an ice-cream shop where you can buy mixin flavours for your ice-cream

Lisp and other languages like Perl and Python have a similar concept called a *mixin*

The name was taken in analogy with an ice-cream shop where you can buy mixin flavours for your ice-cream

A mixin defines only behaviour (methods), not attributes (slots), that is to be mixed into another class

Lisp and other languages like Perl and Python have a similar concept called a *mixin*

The name was taken in analogy with an ice-cream shop where you can buy mixin flavours for your ice-cream

A mixin defines only behaviour (methods), not attributes (slots), that is to be mixed into another class

They are abstract in the Java sense that you can't make direct instances of them

Lisp and other languages like Perl and Python have a similar concept called a *mixin*

The name was taken in analogy with an ice-cream shop where you can buy mixin flavours for your ice-cream

A mixin defines only behaviour (methods), not attributes (slots), that is to be mixed into another class

They are abstract in the Java sense that you can't make direct instances of them

Mixins, unlike interfaces, *can* implement methods

In Common Lisp's CLOS mixins are just a style of programming: it supports full MI, too

In Common Lisp's CLOS mixins are just a style of programming: it supports full MI, too

Other languages have explicit mixin mechanisms

In Common Lisp's CLOS mixins are just a style of programming: it supports full MI, too

Other languages have explicit mixin mechanisms

Exercise. Compare traits and mixins

Some people say MI is too complex, hard to implement properly and produces unexpected results, so you should not have it or use it

Some people say MI is too complex, hard to implement properly and produces unexpected results, so you should not have it or use it

If you want multiple behaviours, you can use SI with *class composition*

Some people say MI is too complex, hard to implement properly and produces unexpected results, so you should not have it or use it

If you want multiple behaviours, you can use SI with *class composition*

An `IOstream` should be a new, independent class, *containing* instances of `Istream` and `Ostream`

Object Oriented Languages

Multiple Inheritance

Some people say MI is too complex, hard to implement properly and produces unexpected results, so you should not have it or use it

If you want multiple behaviours, you can use SI with *class composition*

An `IOstream` should be a new, independent class, *containing* instances of `Istream` and `Ostream`

Not inheriting

Object Oriented Languages

Multiple Inheritance

```
class IOStream: public Istream, public Ostream { ... }
```

becomes

```
class IOStream: { public: Istream i; Ostream o; ... }
```

Object Oriented Languages

Multiple Inheritance

```
class IOStream: public Istream, public Ostream { ... }
```

becomes

```
class IOStream: { public: Istream i; Ostream o; ... }
```

And we need to write `str.i.ptr` or `str.o.ptr` as appropriate to get the stream pointers

```
class IOStream: public Istream, public Ostream { ... }
```

becomes

```
class IOStream: { public: Istream i; Ostream o; ... }
```

And we need to write `str.i.ptr` or `str.o.ptr` as appropriate to get the stream pointers

This can be used by SI languages, too, such as Java

Object Oriented Languages

Multiple Inheritance

```
class IOStream: public Istream, public Ostream { ... }
```

becomes

```
class IOStream: { public: Istream i; Ostream o; ... }
```

And we need to write `str.i.ptr` or `str.o.ptr` as appropriate to get the stream pointers

This can be used by SI languages, too, such as Java

Exercise. See *anonymous structures* in C11 and similar languages that help a little in this regard by allowing unambiguous abbreviations of nested structure accesses

We lose the convenience of inheritance and automatic method selection, but such people argue the inheritance is too problematic to use anyway

We lose the convenience of inheritance and automatic method selection, but such people argue the inheritance is too problematic to use anyway

So this becomes much more like prototyping OO

Object Oriented Languages

Inheritance

We can now see the options if we have a language without inheritance:

We can now see the options if we have a language without inheritance:

- use differential inheritance (clone and add or change behaviour)

We can now see the options if we have a language without inheritance:

- use differential inheritance (clone and add or change behaviour)
- composition

We can now see the options if we have a language without inheritance:

- use differential inheritance (clone and add or change behaviour)
- composition
- live with it

With all these variants of OO, why should we be content with just one kind of OO within a language?

With all these variants of OO, why should we be content with just one kind of OO within a language?

Just because a language designer has said this language should have that kind of OO, should we be stuck with it?

With all these variants of OO, why should we be content with just one kind of OO within a language?

Just because a language designer has said this language should have that kind of OO, should we be stuck with it?

A *metaobject protocol* (MOP) is a means by which we describe what kind of object protocol we want

The metaobject protocol exposes the internal mechanisms of how objects are structured, how methods are chosen, how properties are inherited (if we have inheritance) and so on

The metaobject protocol exposes the internal mechanisms of how objects are structured, how methods are chosen, how properties are inherited (if we have inheritance) and so on

Early experiments with MOPs in Simula were developed in Smalltalk and led to CLOS: the *Common Lisp Object System*, a fully reflective object system

Object Oriented Languages

Metaobject Protocols

The metaobject protocol exposes the internal mechanisms of how objects are structured, how methods are chosen, how properties are inherited (if we have inheritance) and so on

Early experiments with MOPs in Simula were developed in Smalltalk and led to CLOS: the *Common Lisp Object System*, a fully reflective object system

Recall: *reflective* means a system can look at itself and even change itself

And the best way to describe an OO system?

And the best way to describe an OO system?

Using itself!

And the best way to describe an OO system?

Using itself!

In CLOS (as in other MOP languages) there are

- classes that describe the structure and behaviour of classes
- methods that describe how objects should be created and initialised
- methods that describe how methods are looked up
- methods that describe how methods should be inherited or overridden or combined
- and so on for all aspects of an OO system

Exercise. Think about the bootstrap problem of a MOP

We shall take examples from Telos, the EuLisp Object System, as it is much simpler than CLOS

We shall take examples from Telos, the EuLisp Object System, as it is much simpler than CLOS

There are standard classes and methods that describe standard structure, inheritance, method selection and so on

We shall take examples from Telos, the EuLisp Object System, as it is much simpler than CLOS

There are standard classes and methods that describe standard structure, inheritance, method selection and so on

These implement OO behaviour as you might expect from other languages

Object Oriented Languages

Metaobject Protocols

We shall take examples from Telos, the EuLisp Object System, as it is much simpler than CLOS

There are standard classes and methods that describe standard structure, inheritance, method selection and so on

These implement OO behaviour as you might expect from other languages

The class `<simple-class>` and its methods describe these standard things

Object Oriented Languages

Metaobject Protocols

We shall take examples from Telos, the EuLisp Object System, as it is much simpler than CLOS

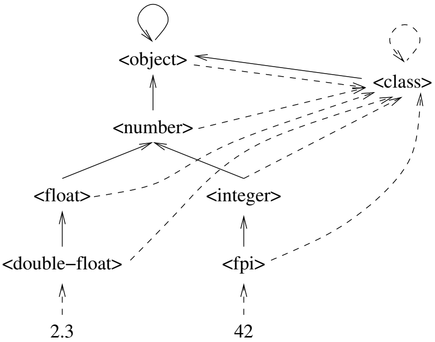
There are standard classes and methods that describe standard structure, inheritance, method selection and so on

These implement OO behaviour as you might expect from other languages

The class `<simple-class>` and its methods describe these standard things

It is a subclass of the topmost (abstract) class `<class>`

Previously we saw:

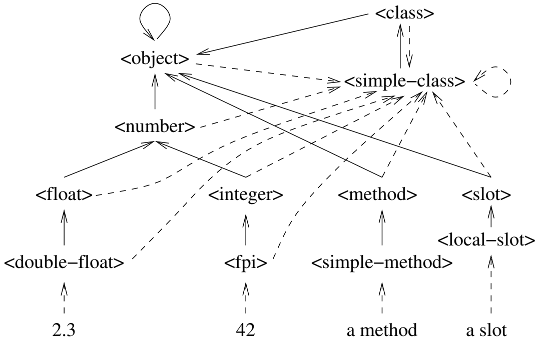


Part of the EuLisp Class Hierarchy (simplified)

Dotted arrow is *instance of*/*member of*/*is a*; solid arrow is *inherits from*/*subclass*/*extends*/*subset*

Object Oriented Languages

Metaobject Protocols



Part of the EuLisp Class Hierarchy (not so simplified)

Dotted arrow is *instance of* or *member of* or *is a*; solid arrow is *inherits from* or *subclass* or *extends*

If we want different behaviours we can create new classes that implement those behaviours

If we want different behaviours we can create new classes that implement those behaviours

Classes are instances of subclasses of the class `<class>`

If we want different behaviours we can create new classes that implement those behaviours

Classes are instances of subclasses of the class `<class>`

Thus the class `<string>` is an instance of `<simple-class>`

If we want different behaviours we can create new classes that implement those behaviours

Classes are instances of subclasses of the class `<class>`

Thus the class `<string>` is an instance of `<simple-class>`

Just as strings are instances of `<string>`

Making a new class:

```
(defclass <myclass> () ... class: <simple-class>)
```

using the `class:` keyword to indicate this is an instance of `simple-class` (the default)

Making a new class:

`(defclass <myclass> () ... class: <simple-class>)`

using the `class:` keyword to indicate this is an instance of `simple-class` (the default)

Or `(defclass <myclass> (<simple-class>) ... class: <simple-class>)`

if you want to inherit `simple-class`'s default structure and behaviour

Then we can add methods to the generic functions that comprise the metaobject protocol to implement our new functionality

Then we can add methods to the generic functions that comprise the metaobject protocol to implement our new functionality

We can now define classes that have this new functionality by

```
(defclass <weirdobject> () ...  class: <myclass>)

(make <weirdobject> ...)
```

Methods: how do we find the right method to apply?

Methods: how do we find the right method to apply?

We need to find the class precedence list for an argument

Methods: how do we find the right method to apply?

We need to find the class precedence list for an argument

So Telos provides a generic function `compute-class-precedence-list`

Methods: how do we find the right method to apply?

We need to find the class precedence list for an argument

So Telos provides a generic function
`compute-class-precedence-list`

There is a method on this for `<simple-class>` that does the standard thing with CPLs, as described previously

Methods: how do we find the right method to apply?

We need to find the class precedence list for an argument

So Telos provides a generic function
`compute-class-precedence-list`

There is a method on this for `<simple-class>` that does the standard thing with CPLs, as described previously

You can add a method yourself if you want to something different, e.g., reverse the order, or omit some classes, or add some strange kind of multiple inheritance, etc.

We need to take the CPL and choose a method (or methods) from it

We need to take the CPL and choose a method (or methods) from it

The generic function `compute-method-lookup-function` is used for this

We need to take the CPL and choose a method (or methods) from it

The generic function `compute-method-lookup-function` is used for this

The standard method returns a function that simply picks the first on the list

We need to take the CPL and choose a method (or methods) from it

The generic function `compute-method-lookup-function` is used for this

The standard method returns a function that simply picks the first on the list

Method combination can be implemented by specialising `compute-method-lookup-function`

We need to take the CPL and choose a method (or methods) from it

The generic function `compute-method-lookup-function` is used for this

The standard method returns a function that simply picks the first on the list

Method combination can be implemented by specialising `compute-method-lookup-function`

And so on