The next kind of OO is *delegation*

The next kind of OO is *delegation*

Delegation is

> *one kind of object, one kind of link*

The next kind of OO is *delegation*

Delegation is

one kind of object, one kind of link

In delegation, objects have a parent object

Object Oriented Languages

Delegation

The next kind of OO is *delegation*

Delegation is

> *one kind of object, one kind of link*

In delegation, objects have a parent object

Thus a form of inheritance, but to a parent *object*

Object Oriented Languages

Delegation

The next kind of OO is *delegation*

Delegation is

> *one kind of object, one kind of link*

In delegation, objects have a parent object

Thus a form of inheritance, but to a parent *object*

Also not a defining feature, but such languages often allow you to change your parent (and therefore your inherited behaviour) at runtime!

Object Oriented Languages

Delegation

- creating a new object is done by direct construction or cloning

Object Oriented Languages

Delegation

- creating a new object is done by direct construction or cloning
- an object contains its own attributes, behaviours and a link to a parent

Object Oriented Languages

Delegation

- creating a new object is done by direct construction or cloning
- an object contains its own attributes, behaviours and a link to a parent
- if there is an applicable method/attribute in the object, use it, otherwise pass to the parent (but the parent is an object, not a class)

Object Oriented Languages

Delegation

- creating a new object is done by direct construction or cloning
- an object contains its own attributes, behaviours and a link to a parent
- if there is an applicable method/attribute in the object, use it, otherwise pass to the parent (but the parent is an object, not a class)
- developed as this is a natural way of working and sharing code

Object Oriented Languages

Delegation

- creating a new object is done by direct construction or cloning
- an object contains its own attributes, behaviours and a link to a parent
- if there is an applicable method/attribute in the object, use it, otherwise pass to the parent (but the parent is an object, not a class)
- developed as this is a natural way of working and sharing code

Prototyping languages can mimic delegation by following an explicit reference to a contained parent object:
`obj.parent.method()`

Object Oriented Languages

Delegation

JavaScript supports delegation by means of a parent slot named `prototype` (later versions: `setPrototypeOf()`)

```
function base() { this.one = 1; }
function derived() { this.two = 2; }

var baseobj = new base();
derived.prototype = baseobj;  // set parent pointer
var a = new derived(), b = new derived();
// a.one -> 1
baseobj.one = 99;
// a.one -> 99
// b.one -> 99
```

All the instances in this example share the same parent object `baseobj`

Object Oriented Languages

Delegation

JavaScript is so dynamic as a language we can even

```
baseobj.three = 3;
// a.three -> 3
// b.three -> 3
```

So allowing global dynamic addition of behaviour: all this works with both slots and methods; overriding works as expected

JavaScript is so dynamic as a language we can even

```
baseobj.three = 3;
// a.three -> 3
// b.three -> 3
```

So allowing global dynamic addition of behaviour: all this works with both slots and methods; overriding works as expected

Exercise Compare with duck typing

Exercise Later versions of JavaScript (ECMAScript 6) have things called classes, but they are simply converted by the compiler into prototypes and closures. Read about this

Next are *traits*: with variants called *type classes* or *typeclasses*, *roles*, *interfaces*, *mixins*, or even *concepts*

Object Oriented Languages

Traits

Next are *traits*: with variants called *type classes* or *typeclasses*, *roles*, *interfaces*, *mixins*, or even *concepts*

The same basic idea has been reinvented several times, with some variations in detail

Object Oriented Languages

Traits

Next are *traits*: with variants called *type classes* or *typeclasses*, *roles*, *interfaces*, *mixins*, or even *concepts*

The same basic idea has been reinvented several times, with some variations in detail

Classically, *traits* have

> *two kinds of object, one kind of link*

Next are *traits*: with variants called *type classes* or *typeclasses*, *roles*, *interfaces*, *mixins*, or even *concepts*

The same basic idea has been reinvented several times, with some variations in detail

Classically, *traits* have

> *two kinds of object, one kind of link*

The link is to a parent

Object Oriented Languages

Traits

Next are *traits*: with variants called *type classes* or *typeclasses*, *roles*, *interfaces*, *mixins*, or even *concepts*

The same basic idea has been reinvented several times, with some variations in detail

Classically, *traits* have

> *two kinds of object, one kind of link*

The link is to a parent

Objects, as usual, plus a special kind of thing called a *trait* (often not an object in the OO sense)

Traits are not classes, but they do gather together and encapsulate *behaviours* of objects: the methods are now not in the object but have a separate existence in a trait

Traits are not classes, but they do gather together and encapsulate *behaviours* of objects: the methods are now not in the object but have a separate existence in a trait

Thus we can reuse behaviour independently of the parent hierarchy, which purely about object structure

Object Oriented Languages

Traits

Traits are not classes, but they do gather together and encapsulate *behaviours* of objects: the methods are now not in the object but have a separate existence in a trait

Thus we can reuse behaviour independently of the parent hierarchy, which purely about object structure

An object could have the behaviour (trait) of a dog while its parent could have the behaviour of a cat

Object Oriented Languages

Traits

Traits are not classes, but they do gather together and encapsulate *behaviours* of objects: the methods are now not in the object but have a separate existence in a trait

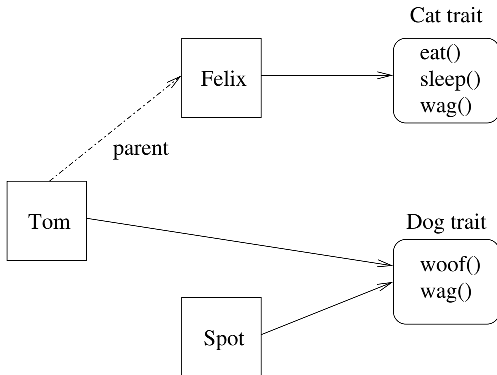
Thus we can reuse behaviour independently of the parent hierarchy, which purely about object structure

An object could have the behaviour (trait) of a dog while its parent could have the behaviour of a cat

Traits are normally associated with methods, though some languages allow them to contain functions, too

Object Oriented Languages

Traits



Traits keep functions/methods separate

Tom `wags` like a dog, but `sleeps` like a cat

Object Oriented Languages

Traits

- an object contains its own attributes and references to a trait (or traits) and (optionally) a link to a parent

Object Oriented Languages

Traits

- an object contains its own attributes and references to a trait (or traits) and (optionally) a link to a parent
- attribute lookup is via the object

Object Oriented Languages

Traits

- an object contains its own attributes and references to a trait (or traits) and (optionally) a link to a parent
- attribute lookup is via the object
- if there is an applicable behaviour in the trait, use it, otherwise pass to the object's parent and look in its trait

Object Oriented Languages

Traits

- an object contains its own attributes and references to a trait (or traits) and (optionally) a link to a parent
- attribute lookup is via the object
- if there is an applicable behaviour in the trait, use it, otherwise pass to the object's parent and look in its trait
- creating a new object is done by direct construction or cloning

Object Oriented Languages

Traits

- an object contains its own attributes and references to a trait (or traits) and (optionally) a link to a parent
- attribute lookup is via the object
- if there is an applicable behaviour in the trait, use it, otherwise pass to the object's parent and look in its trait
- creating a new object is done by direct construction or cloning
- developed as this allows sharing of behaviour independent of sharing of structure (a `Dog` and a `Car` could share a `move` method)

Comparing classes and traits:

Comparing classes and traits:

Classes: structure and behaviour tied together in things called "classes"

Object Oriented Languages

Traits

Comparing classes and traits:

Classes: structure and behaviour tied together in things called "classes"

Code reuse is by inheritance of classes: if you inherit a class you get both structure and behaviour

Comparing classes and traits:

Classes: structure and behaviour tied together in things called “classes”

Code reuse is by inheritance of classes: if you inherit a class you get both structure and behaviour

Traits: structure and behaviour in separate places and can be used independently

Comparing classes and traits:

Classes: structure and behaviour tied together in things called "classes"

Code reuse is by inheritance of classes: if you inherit a class you get both structure and behaviour

Traits: structure and behaviour in separate places and can be used independently

Behaviour is not tied to structure

Traits have recently had a resurgence in popularity

Traits have recently had a resurgence in popularity

Though somewhat changed in their modern form

Object Oriented Languages

Traits

Traits have recently had a resurgence in popularity

Though somewhat changed in their modern form

Thing like traits appear in Python (roles), Perl (roles), Ruby, Rust, Java (interfaces), Swift (protocols), Go (interfaces), Common Lisp (mixins), Ruby (mixins), Haskell (typeclasses)

Object Oriented Languages

Traits

Traits have recently had a resurgence in popularity

Though somewhat changed in their modern form

Thing like traits appear in Python (roles), Perl (roles), Ruby, Rust, Java (interfaces), Swift (protocols), Go (interfaces), Common Lisp (mixins), Ruby (mixins), Haskell (typeclasses)

Exercise A lot of these have traditional classes with inheritance as well as trait-like things. Why have both?

Object Oriented Languages

Traits

A trait was originally just a collection of method declarations, but the word came to mean a variety of things, sometimes under different names

Object Oriented Languages

Traits

A trait was originally just a collection of method declarations, but the word came to mean a variety of things, sometimes under different names

These days it often means just a collection of method *signatures*, i.e., just the method names with the types of their parameters and result, no actual code

Object Oriented Languages

Traits

A trait was originally just a collection of method declarations, but the word came to mean a variety of things, sometimes under different names

These days it often means just a collection of method *signatures*, i.e., just the method names with the types of their parameters and result, no actual code

Acting as a requirement that a type must implement for itself the methods as described

A trait was originally just a collection of method declarations, but the word came to mean a variety of things, sometimes under different names

These days it often means just a collection of method *signatures*, i.e., just the method names with the types of their parameters and result, no actual code

Acting as a requirement that a type must implement for itself the methods as described

But, regardless of approach, a type that implements a trait has all the behaviour specified by that trait

Although some people reserve the word *interface* for a list of signatures

Object Oriented Languages

Traits

Although some people reserve the word *interface* for a list of signatures

```
interface Canid {
   public void woof();
   public void run(double speed);
}

class Dog extends Animal implements Canid {
   ...
}
```

Object Oriented Languages

Traits

Some languages (e.g., Java, Rust) allow the trait to include code, too, to use as a default when a type does not want to implement something itself

Some languages (e.g., Java, Rust) allow the trait to include code, too, to use as a default when a type does not want to implement something itself

And an object can attach to more than one trait, e.g., having the behaviours of *both* cats and dogs

Some languages (e.g., Java, Rust) allow the trait to include code, too, to use as a default when a type does not want to implement something itself

And an object can attach to more than one trait, e.g., having the behaviours of *both* cats and dogs

Again, allowing use of behaviour from many places, not just a parent of some sort

What about parent links in trait-based languages?

What about parent links in trait-based languages?

Not a defining features of traits: inheritance of structure is a separate issue

What about parent links in trait-based languages?

Not a defining features of traits: inheritance of structure is a separate issue

Some languages have parents, some don’t, some have full class-based inheritance

What about parent links in trait-based languages?

Not a defining features of traits: inheritance of structure is a separate issue

Some languages have parents, some don’t, some have full class-based inheritance

Traits are primarily about behaviour, not structure

Exercise For C++ geeks. C++20 introduced *concepts* as a way to constrain its templates. Read about this

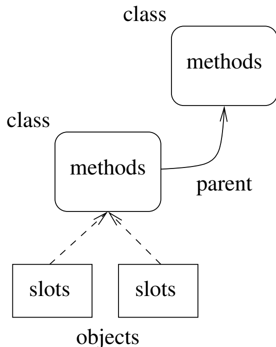
Exercise Also read about Common Lisp and Ruby *mixins*

Exercise Rust uses traits extensively, with “multiple inheritance” in the traits and no parent link in the instances. Read about this

Exercise Java also has “multiple inheritance” in its interfaces. Read about this

Let’s summarise these different kinds of OO

Object Oriented Languages

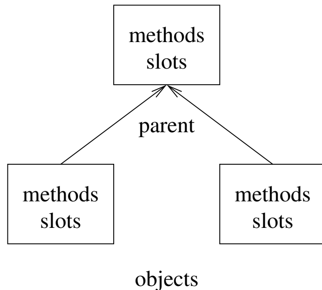


Class Centred (Object Receiver)

Object Oriented Languages

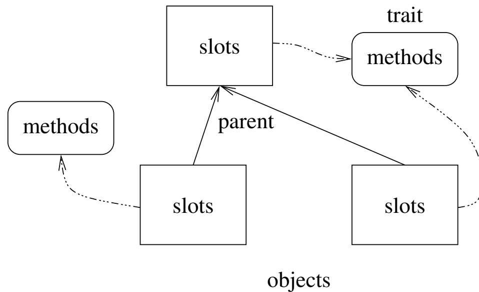
Prototyping

Object Oriented Languages



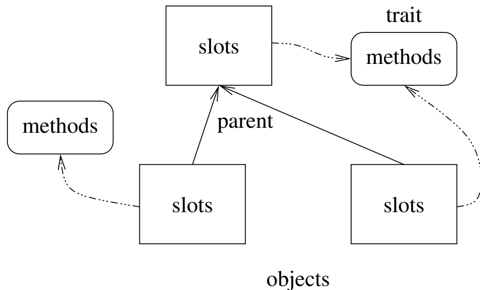
Delegation

Object Oriented Languages



Traits

Object Oriented Languages



Traits

One kind of link?

Object Oriented Languages

objects

links

	1	2
0	prototyping	
1	delegation	trait
2		class centred

Object Oriented Languages

Method Dispatch

Back to the methods: we now look at how to pick the right method to call

Object Oriented Languages

Method Dispatch

Back to the methods: we now look at how to pick the right method to call

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

Object Oriented Languages

Method Dispatch

Back to the methods: we now look at how to pick the right method to call

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

Back to the methods: we now look at how to pick the right method to call

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

Back to the methods: we now look at how to pick the right method to call

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 parameters

In this context "applicable" means "of the given name" and suitable parameters

So `obj.foo(42)` looks for methods with the name `foo` associated with the class of `obj` that take an integer argument

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

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

A familiar trade-off of speed against flexibility

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

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

We'll touch on this again later

But even statically typed OO languages can have an element of dynamic behaviour

But even statically typed OO languages can have an element of dynamic behaviour

This is reminiscent of the static vs. dynamic behaviour of languages, but applied just to how methods are chosen

Suppose we have a class `Animal` with subclass `Dog`, with a method `hastail` on `Animal` returning some default value, overridden (specialised) by `hastail` on `Dog`

Suppose we have a class `Animal` with subclass `Dog`, with a method `hastail` on `Animal` returning some default value, overridden (specialised) by `hastail` on `Dog`

If have code

```
Animal fido = new Dog(...);
...
...fido.hastail()...
```

what do we want to happen?

Object Oriented Languages

Method Dispatch

```
Animal fido = new Dog(...);
...
...fido.hastail()...
```

A `Dog` is an `Animal`, so it's OK for variable `fido` to have type `Animal`

Object Oriented Languages

Method Dispatch

```
Animal fido = new Dog(...);
...
...fido.hastail()...
```

A `Dog` is an `Animal`, so it's OK for variable `fido` to have type `Animal`

But which method do we want called?

Object Oriented Languages

Method Dispatch

```
Animal fido = new Dog(...);
...
...fido.hastail()...
```

A `Dog` is an `Animal`, so it's OK for variable `fido` to have type `Animal`

But which method do we want called?

The `Animal` method, as the variable `fido` has type `Animal`?

Object Oriented Languages

Method Dispatch

```
Animal fido = new Dog(...);
...
...fido.hastail()...
```

A `Dog` is an `Animal`, so it's OK for variable `fido` to have type `Animal`

But which method do we want called?

The `Animal` method, as the variable `fido` has type `Animal`?

The `Dog` method, as the variable `fido` contains a value of type `Dog`?

It depends — in some applications we want the former, in other applications we want the latter

It depends — in some applications we want the former, in other applications we want the latter

In many applications the latter (use the type of the value), but this is not always the case

Object Oriented Languages

Method Dispatch

It depends — in some applications we want the former, in other applications we want the latter

In many applications the latter (use the type of the value), but this is not always the case

The former is *static dispatch*, using the type of the variable

Object Oriented Languages

Method Dispatch

It depends — in some applications we want the former, in other applications we want the latter

In many applications the latter (use the type of the value), but this is not always the case

The former is *static dispatch*, using the type of the variable

The latter is *dynamic dispatch*, using the type of the current object contained in the variable

Object Oriented Languages

Method Dispatch

It depends — in some applications we want the former, in other applications we want the latter

In many applications the latter (use the type of the value), but this is not always the case

The former is *static dispatch*, using the type of the variable

The latter is *dynamic dispatch*, using the type of the current object contained in the variable

Note we could later set `fido = new Cat(...)`, so the type of the contained object can change, while still being an `Animal`, and this might require a different `hastail`

Object Oriented Languages

Method Dispatch

Dynamic dispatch is sometimes also called:

- virtual method dispatch
- runtime dispatch
- late binding

Static dispatch is sometimes also called:

- early binding

Object Oriented Languages

Method Dispatch

A static method dispatch can be completely compiled away, as in the $a + b$ example when we were talking about dynamic and static languages

A static method dispatch can be completely compiled away, as in the $a + b$ example when we were talking about dynamic and static languages

A dynamic method dispatch will need the compiler to output some code to pick a method at runtime — at each dynamic method call in the code

A static method dispatch can be completely compiled away, as in the $a + b$ example when we were talking about dynamic and static languages

A dynamic method dispatch will need the compiler to output some code to pick a method at runtime — at each dynamic method call in the code

Exercise Though in a static single-inheritance language this dynamic lookup can be quite fast. Read about *dispatch tables*/*virtual method tables*/*vtables*

Object Oriented Languages

Method Dispatch

As both kinds of dispatch are useful, many OO languages support both and allow the programmer to specify which they want

As both kinds of dispatch are useful, many OO languages support both and allow the programmer to specify which they want

The programmer needs to be aware of the difference between the two, and the costs involved!

Object Oriented Languages

Method Dispatch

As both kinds of dispatch are useful, many OO languages support both and allow the programmer to specify which they want

The programmer needs to be aware of the difference between the two, and the costs involved!

Static fast, but less flexible; dynamic slower, but more flexible

Object Oriented Languages

Method Dispatch

As both kinds of dispatch are useful, many OO languages support both and allow the programmer to specify which they want

The programmer needs to be aware of the difference between the two, and the costs involved!

Static fast, but less flexible; dynamic slower, but more flexible

Exercise Find out if and how your favourite OO languages support this choice

Exercise Think about

```
Animal spot;
if (wombat() > 0) {
    spot = new Cat();
}
else {
    spot = new Dog();
}
... spot.hastail()...
```

where both `Cat` and `Dog` are subclasses of `Animal`

Next: we usually want more specific methods defined in a subclass to override (aka *specialise*) less specific methods in superclasses, but sometimes we want *method composition*

Object Oriented Languages

Method Composition

Next: we usually want more specific methods defined in a subclass to override (aka *specialise*) less specific methods in superclasses, 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

Next: we usually want more specific methods defined in a subclass to override (aka *specialise*) less specific methods in superclasses, 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

To make an instance of `B` it first runs the code for initialising an `A`, then topping up with the code for initialising a `B`

Object Oriented Languages

Method Composition

To make an instance of `B` it first runs the code for initialising an `A`, then topping up with the code for initialising a `B`

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

Object Oriented Languages

Method Composition

To make an instance of `B` it first runs the code for initialising an `A`, then topping up with the code for initialising a `B`

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

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

Object Oriented Languages

Method Composition

To make an instance of `B` it first runs the code for initialising an `A`, then topping up with the code for initialising a `B`

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

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

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

Exercise Java, Python and C# have destructors but call them *finalizers*. Read about the problems the GC languages have with destructors

Exercise Read about using the *Resource Acquisition Is Initialization* (RAII) programming idiom to prevent resource leaks