Object Oriented Languages

There is a wide variety of things that like to be called OO

Object Oriented Languages

There is a wide variety of things that like to be called OO

The basic idea is the use of encapsulation of state within an object

Object Oriented Languages

There is a wide variety of things that like to be called OO

The basic idea is the use of encapsulation of state within an object

And things like classes and inheritance are not a fundamental part of being object oriented

Object Oriented Languages

There is a wide variety of things that like to be called OO

The basic idea is the use of encapsulation of state within an object

And things like classes and inheritance are not a fundamental part of being object oriented

Of course, these variants came about through lots of research and experimentation and have varying levels of success

Object Oriented Languages

There is a wide variety of things that like to be called OO

The basic idea is the use of encapsulation of state within an object

And things like classes and inheritance are not a fundamental part of being object oriented

Of course, these variants came about through lots of research and experimentation and have varying levels of success

As always, it's not a case of what is *better*, more what is *better for the application in hand*

Object Oriented Languages

Class Centred

Class Centred is by far the most well-known form of OO, and what many people think is *all* of OO

Object Oriented Languages

Class Centred

Class Centred is by far the most well-known form of OO, and what many people think is *all* of OO

Examples include C++, Java, Lisp, Smalltalk . . .

Object Oriented Languages

Class Centred

Class Centred is by far the most well-known form of OO, and what many people think is *all* of OO

Examples include C++, Java, Lisp, Smalltalk . . .

Typified by

Object Oriented Languages

Class Centred

Class Centred is by far the most well-known form of OO, and what many people think is *all* of OO

Examples include C++, Java, Lisp, Smalltalk . . .

Typified by

- classes (first-class or not first-class)

Object Oriented Languages

Class Centred

Class Centred is by far the most well-known form of OO, and what many people think is *all* of OO

Examples include C++, Java, Lisp, Smalltalk . . .

Typified by

- classes (first-class or not first-class)
- instances of those classes

Object Oriented Languages

Class Centred

Class Centred is by far the most well-known form of OO, and what many people think is *all* of OO

Examples include C++, Java, Lisp, Smalltalk . . .

Typified by

- classes (first-class or not first-class)
- instances of those classes
- methods attached to classes or generic function objects, shared by instances

Object Oriented Languages

Class Centred

Class Centred is by far the most well-known form of OO, and what many people think is *all* of OO

Examples include C++, Java, Lisp, Smalltalk . . .

Typified by

- classes (first-class or not first-class)
- instances of those classes
- methods attached to classes or generic function objects, shared by instances
- attributes/slots defined in classes, attached to instances (or classes)

Object Oriented Languages

Class Centred

Class Centred is by far the most well-known form of OO, and what many people think is *all* of OO

Examples include C++, Java, Lisp, Smalltalk ...

Typified by

- classes (first-class or not first-class)
- instances of those classes
- methods attached to classes or generic function objects, shared by instances
- attributes/slots defined in classes, attached to instances (or classes)
- single or multiple inheritance defined through the relationships between the classes

Object Oriented Languages

Brief aside

There is a lot of variation on terminology that reflects the many ways people think about OO

Object Oriented Languages

Brief aside

There is a lot of variation on terminology that reflects the many ways people think about OO

- For data: attribute, state, slot, member, value, element, variant, structure

Object Oriented Languages

Brief aside

There is a lot of variation on terminology that reflects the many ways people think about OO

- For data: attribute, state, slot, member, value, element, variant, structure
- For code: method, behaviour, action, message

Object Oriented Languages

Brief aside

There is a lot of variation on terminology that reflects the many ways people think about OO

- For data: attribute, state, slot, member, value, element, variant, structure
- For code: method, behaviour, action, message

Be aware of these variations!

Object Oriented Languages

Class Centred

Class centred languages are occasionally further divided by how they treat methods

- object receiver: Java, C++, . . .
- generic functions: Lisp, . . .

Object Oriented Languages

Class Centred

Class centred languages are occasionally further divided by how they treat methods

- object receiver: Java, C++, . . .
- generic functions: Lisp, . . .

The object receiver view of the world has a single object receiving a message, such as `a.plus(b)`, and chooses a method depending on the type of a single object (`a` in this case)

Object Oriented Languages

Class Centred

Class centred languages are occasionally further divided by how they treat methods

- object receiver: Java, C++, . . .
- generic functions: Lisp, . . .

The object receiver view of the world has a single object receiving a message, such as `a.plus(b)`, and chooses a method depending on the type of a single object (`a` in this case)

Generic functions look more like normal functions: `plus(a,b)` or `(plus a b)`, and they choose a method depending on the types of `a` *and* `b`

Object Oriented Languages

Class Centred

Note this is syntactic convenience. We could invent a syntax, say

```
(a,b).plus()
```

to emphasise the messaging, but it's simpler to use the function notation for the multiple receiver case (as long as you remember it's a *method call*, not a *function call*)

Object Oriented Languages

Class Centred

In this case methods are now attached to attached to *generic functions* (e.g., `plus`), rather than individual classes

Object Oriented Languages

Class Centred

In this case methods are now attached to attached to *generic functions* (e.g., `plus`), rather than individual classes

Terminology: from Java you might be used to saying “a method defined in a class” or “on a class” — this is not appropriate for the generic function approach

Object Oriented Languages

Class Centred

In this case methods are now attached to attached to *generic functions* (e.g., `plus`), rather than individual classes

Terminology: from Java you might be used to saying “a method defined in a class” or “on a class” — this is not appropriate for the generic function approach

This is because now a method can depend on multiple classes

Object Oriented Languages

Class Centred

In this case methods are now attached to attached to *generic functions* (e.g., `plus`), rather than individual classes

Terminology: from Java you might be used to saying “a method defined in a class” or “on a class” — this is not appropriate for the generic function approach

This is because now a method can depend on multiple classes

Saying “method in a class” is OK for Java, not for Lisp

Object Oriented Languages

Class Centred

Generic functions *dispatch* (choose a method) on the type of one or more objects

Object Oriented Languages

Class Centred

Generic functions *dispatch* (choose a method) on the type of one or more objects

So they are called *multiple dispatch* in contrast with (say) Java that is *single dispatch*

Object Oriented Languages

Class Centred

Generic functions *dispatch* (choose a method) on the type of one or more objects

So they are called *multiple dispatch* in contrast with (say) Java that is *single dispatch*

Generic functions look a lot like normal functions, but are actually *collections* of methods

Object Oriented Languages

Class Centred

```
(defgeneric foo (a b))

(defmethod foo ((a <number>) (b <number>)) ...)

(defmethod foo ((a <integer>) (b <integer>)) ...)

(defmethod foo ((a <number>) (b <float>)) ...)

(defmethod foo ((a <float>) (b <integer>)) ...)

...
```

Object Oriented Languages

Class Centred

```
(defgeneric foo (a b))

(defmethod foo ((a <number>) (b <number>)) ...)

(defmethod foo ((a <integer>) (b <integer>)) ...)

(defmethod foo ((a <number>) (b <float>)) ...)

(defmethod foo ((a <float>) (b <integer>)) ...)

...
```

Choosing the applicable method is more involved, but typically is the closest match, taking arguments left-to-right to break ties (more on this later)

General Remark

Methods, functions and generic functions are different things

General Remark

Methods, functions and generic functions are different things

Functions and methods are different things

General Remark

Languages like C **do not have methods**, only functions

General Remark

Languages like C **do not have methods**, only functions

Make sure you understand the difference between methods and functions: calling a C function a “method” is a clear indication that you don’t understand what you are talking about

General Remark

Languages like C **do not have methods**, only functions

Make sure you understand the difference between methods and functions: calling a C function a “method” is a clear indication that you don’t understand what you are talking about

A function is just some code

General Remark

A method comprises a function **plus** other class-related things needed to make OO work, in particular a reference to the object in question; perhaps also its class; and more as we shall see later

General Remark

A method comprises a function **plus** other class-related things needed to make OO work, in particular a reference to the object in question; perhaps also its class; and more as we shall see later

A generic function comprises zero or more methods

General Remark

A method comprises a function **plus** other class-related things needed to make OO work, in particular a reference to the object in question; perhaps also its class; and more as we shall see later

A generic function comprises zero or more methods

We have also seen *closures*, which are different again

General Remark

- function: code
- method: function plus object reference
- generic function: collection of methods
- closure: function plus environment

General Remark

- function: code
- method: function plus object reference
- generic function: collection of methods
- closure: function plus environment

Confusing these concepts will ensure loss of marks!

General Remark

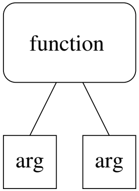
- function: code
- method: function plus object reference
- generic function: collection of methods
- closure: function plus environment

Confusing these concepts will ensure loss of marks!

Exercise. Think about methods that use closures

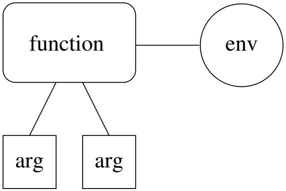
General Remark

Functions just have code and arguments



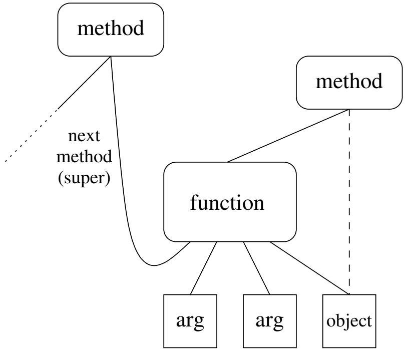
General Remark

Closures have code, arguments and environment



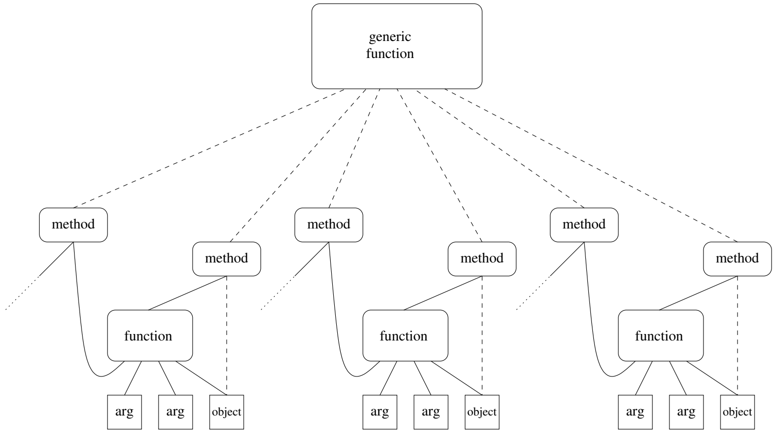
General Remark

Methods have code, arguments, the object and a *next method list*



General Remark

Generic functions are a collection of methods



Aside

For those interested in the mechanisms: a method call `obj.meth(x,y)` is often compiled into the equivalent of a normal function call with extra “hidden” arguments

```
meth_class_of_obj(obj, next_method_list, x, y)
```

and `obj` is accessible within the body of the function as the function argument `this` (or `self`, or just implicit)

Any `super` methods are contained in the `next_method_list`

Aside

While we are talking about these things, suppose we have

```
class Foo {
  int n;
  int inc(int m) { return n+m; }
}
```

Aside

While we are talking about these things, suppose we have

```
class Foo {
  int n;
  int inc(int m) { return n+m; }
}
```

The compiler will write a **function** something like

```
int Foo_int_inc_int(Foo self, int m) { return self.n + m; }
```

(ignoring questions of call by reference or value and super methods)

Aside

Then method calls such as

```
Foo f;
f.n = 23;
y = f.inc(42);
```

become ordinary function calls like

```
y = Foo_int_inc_int(f, 42);
```

Aside

Thus, in this example, there is *no runtime overhead in using method calls*

Aside

Thus, in this example, there is *no runtime overhead in using method calls*

The “method lookup” is done in the compiler and the resulting code is just as fast as calling a function

Aside

Thus, in this example, there is *no runtime overhead in using method calls*

The “method lookup” is done in the compiler and the resulting code is just as fast as calling a function

Other languages or systems might do the lookup at runtime, so for these kinds of system, a method is slower than a function

Further Aside

A clever compiler might even *inline* the function call

```
y = f.n + 42;
```

to avoid the cost of the function call

Even Further Aside

An even better compiler might even replace this by

```
y = 64;
```

as it “knows” what the current value of `f.n` is

Even Further Aside

An even better compiler might even replace this by

```
y = 64;
```

as it “knows” what the current value of `f.n` is

Exercise. Go to a compiler course

Object Oriented Languages

Object Centred

Less well recognised than the class centred languages are the *object centred* languages, but they are widely used since JavaScript is a major example

Object Oriented Languages

Object Centred

Less well recognised than the class centred languages are the *object centred* languages, but they are widely used since JavaScript is a major example

Examples include JavaScript, Self, . . .

Object Oriented Languages

Object Centred

Less well recognised than the class centred languages are the *object centred* languages, but they are widely used since JavaScript is a major example

Examples include JavaScript, Self, …

Typified by

Object Oriented Languages
Object Centred

Less well recognised than the class centred languages are the *object centred* languages, but they are widely used since JavaScript is a major example

Examples include JavaScript, Self, . . .

Typified by

- objects only, no classes

Object Oriented Languages

Object Centred

Less well recognised than the class centred languages are the *object centred* languages, but they are widely used since JavaScript is a major example

Examples include JavaScript, Self, . . .

Typified by

- objects only, no classes
- methods attached to objects

Object Oriented Languages

Object Centred

Less well recognised than the class centred languages are the *object centred* languages, but they are widely used since JavaScript is a major example

Examples include JavaScript, Self, . . .

Typified by

- objects only, no classes
- methods attached to objects
- slots attached to objects

Object Oriented Languages

Object Centred

Less well recognised than the class centred languages are the *object centred* languages, but they are widely used since JavaScript is a major example

Examples include JavaScript, Self, . . .

Typified by

- objects only, no classes
- methods attached to objects
- slots attached to objects
- direct construction and *cloning* to make instances

Object Oriented Languages

Object Centred

Less well recognised than the class centred languages are the *object centred* languages, but they are widely used since JavaScript is a major example

Examples include JavaScript, Self, ...

Typified by

- objects only, no classes
- methods attached to objects
- slots attached to objects
- direct construction and *cloning* to make instances
- no default inheritance, programmer defined inheritance, if required

Object Oriented Languages

List Constructor in JavaScript

```
function list() {
  this.size = 0
  this.node = {next: 0, prev: 0, data: 0}
  this.node.next = this.node
  this.node.prev = this.node
  this.push_back = function (x) {
                     var tmp = {next: this.node,
                                prev: this.node.prev,
                                data: x}
                     this.node.prev.next = tmp
                     this.node.prev = tmp
                     this.size += 1
                     return x
                   }
  this.toString = list_toString
  for (var i = 0; i < arguments.length; i++) {
    this.push_back(arguments[i])
  }
}
```

Object Oriented Languages

List Constructor in JavaScript

- `list`: the current object is referred to as `this`; other languages use `self`

Object Oriented Languages

List Constructor in JavaScript

- `list`: the current object is referred to as `this`; other languages use `self`
- `this.node = {next: 0, prev: 0, data: 0}`: sets the `node` slot to a structure value

Object Oriented Languages

List Constructor in JavaScript

- `list`: the current object is referred to as `this`; other languages use `self`
- `this.node = {next: 0, prev: 0, data: 0}`: sets the `node` slot to a structure value
- `this.push_back`: defines a method to add an item

Object Oriented Languages

List Constructor in JavaScript

- `list`: the current object is referred to as `this`; other languages use `self`
- `this.node = {next: 0, prev: 0, data: 0}`: sets the `node` slot to a structure value
- `this.push_back`: defines a method to add an item
- `this.toString = list_toString`: another method defined elsewhere

Object Oriented Languages

List Constructor in JavaScript

- `list`: the current object is referred to as `this`; other languages use `self`
- `this.node = {next: 0, prev: 0, data: 0}`: sets the `node` slot to a structure value
- `this.push_back`: defines a method to add an item
- `this.toString = list_toString`: another method defined elsewhere
- `for ...`: code to execute when making an object

Object Oriented Languages

List Constructor in JavaScript

This would be used like

```
var l = new list("hello", 1, "world");
l.push_back(2);
var len = l.size;
```

Object Oriented Languages

List Constructor in JavaScript

This would be used like

```
var l = new list("hello", 1, "world");
l.push_back(2);
var len = l.size;
```

Note: no class definition, only how to make an object

Object Oriented Languages

Note that object centred languages are often dynamically typed, while class centred languages are often statically typed

Object Oriented Languages

Note that object centred languages are often dynamically typed, while class centred languages are often statically typed

But these are separate concepts that should not be confused

Object Oriented Languages

Note that object centred languages are often dynamically typed, while class centred languages are often statically typed

But these are separate concepts that should not be confused

Some class centred languages are dynamic, e.g., Common Lisp can redefine its classes as it is running

Object Oriented Languages

Class centred OO could be thought of as

two kinds of object, two kinds of link

Object Oriented Languages

Class centred OO could be thought of as

two kinds of object, two kinds of link

Namely classes and non-classes, inheritance and instance

Object Oriented Languages

Prototyping

Prototyping is then

> *one kind of object, no links*

Object Oriented Languages

Prototyping

Prototyping is then

> *one kind of object, no links*

JavaScript is a *prototyping* language

Object Oriented Languages

Prototyping

Prototyping is then

> *one kind of object, no links*

JavaScript is a *prototyping* language

NB: don't confuse this usage with languages that are used for prototyping!

Object Oriented Languages

Prototyping

- an object contains its own attributes (slots) and behaviours (methods), not a class

Object Oriented Languages

Prototyping

- an object contains its own attributes (slots) and behaviours (methods), not a class
- attribute and behaviour lookup are both by interrogating the object

Object Oriented Languages

Prototyping

- an object contains its own attributes (slots) and behaviours (methods), not a class
- attribute and behaviour lookup are both by interrogating the object
- creating a new object is done by direct construction or by *cloning*, i.e., copying an existing object: the *prototype*

Object Oriented Languages

Prototyping

- an object contains its own attributes (slots) and behaviours (methods), not a class
- attribute and behaviour lookup are both by interrogating the object
- creating a new object is done by direct construction or by *cloning*, i.e., copying an existing object: the *prototype*
- no inheritance in the class-centred sense, but an object can itself call other methods as it sees fit: an object could contain an object of another type and treat that as its parent, calling its methods explicitly

Object Oriented Languages

Prototyping

Though not a defining feature of prototyping, these languages often allow dynamic addition of attributes and behaviours to objects:

```
function obj() { this.one = 1; this.two = 2; }
var a = new obj(), b = new obj();
a.three = 3;
// b.three is undefined
```

Object Oriented Languages
Prototyping

Though not a defining feature of prototyping, these languages often allow dynamic addition of attributes and behaviours to objects:

```
function obj() { this.one = 1; this.two = 2; }
var a = new obj(), b = new obj();
a.three = 3;
// b.three is undefined
```

- used in *differential inheritance*: clone an object then add a new behaviour

Object Oriented Languages

Prototyping

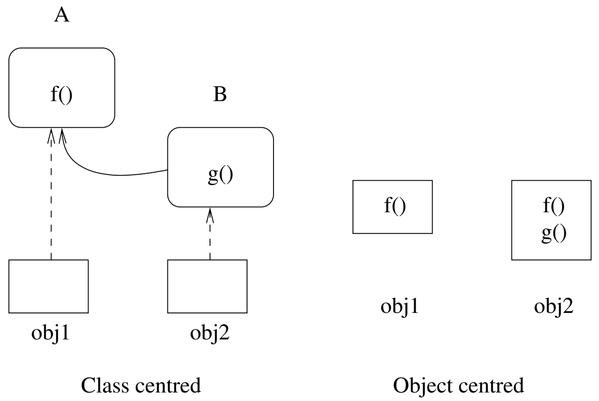
Though not a defining feature of prototyping, these languages often allow dynamic addition of attributes and behaviours to objects:

```
function obj() { this.one = 1; this.two = 2; }
var a = new obj(), b = new obj();
a.three = 3;
// b.three is undefined
```

- used in *differential inheritance*: clone an object then add a new behaviour
- again, different from class-centred inheritance as the cloned object contains all its own methods and attributes

Object Oriented Languages

Prototyping



In class-centred, `obj2` gets `f` and `g` from its classes

In object centred, they are self-contained

Object Oriented Languages

Prototyping

- less efficient (requires runtime lookups) but more flexible

Object Oriented Languages

Prototyping

- less efficient (requires runtime lookups) but more flexible
- it was developed as real code is never as simple as a tidy class hierarchy might provide: we might want some behaviour of a parent but not all its behaviour. Prototyping allows us to gather together whatever we need from wherever we want without constraint

Object Oriented Languages

Delegation

The next kind of OO is *delegation*

Object Oriented Languages

Delegation

The next kind of OO is *delegation*

Delegation is

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

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

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 behaviour) at runtime!

Object Oriented Languages

Delegation

- an object contains its own attributes, behaviours and link to a parent

Object Oriented Languages

Delegation

- an object contains its own attributes, behaviours and link to a parent
- attribute lookup is via the object

Object Oriented Languages

Delegation

- an object contains its own attributes, behaviours and link to a parent
- attribute lookup is via the object
- behaviour lookup is a little like class centred: if there is an applicable method in the object, use it, otherwise pass to the parent (but the parent is an object, not a class)

Object Oriented Languages

Delegation

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

Object Oriented Languages

Delegation

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

Object Oriented Languages

Delegation

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

Prototyping languages can mimic delegation by following an explicit reference to a parent object

Object Oriented Languages

Delegation

Later versions of JavaScript support delegation by means of a parent slot named `prototype`

```
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 Oriented Languages

Delegation

JavaScript automatically follows the parent chain; other prototyping languages might not

Object Oriented Languages

Delegation

JavaScript automatically follows the parent chain; other prototyping languages might not

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

Object Oriented Languages

Delegation

JavaScript automatically follows the parent chain; other prototyping languages might not

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

Object Oriented Languages

Traits

Classically, *traits* is

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

Object Oriented Languages

Traits

Classically, *traits* is

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

The link is to a parent

Object Oriented Languages

Traits

Classically, *traits* is

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

The link is to a parent

Objects, as usual, plus a special kind of object called a *trait*

Object Oriented Languages

Traits

Classically, *traits* is

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

The link is to a parent

Objects, as usual, plus a special kind of object called a *trait*

Traits encapsulate *behaviours* of objects: the methods can be pulled out of the object and have a separate existence in a trait

Object Oriented Languages

Traits

Classically, *traits* is

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

The link is to a parent

Objects, as usual, plus a special kind of object called a *trait*

Traits encapsulate *behaviours* of objects: the methods can be pulled out of the object and have a separate existence in a trait

Thus we can reuse behaviour independently of the parent hierarchy

Object Oriented Languages

Traits

Classically, *traits* is

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

The link is to a parent

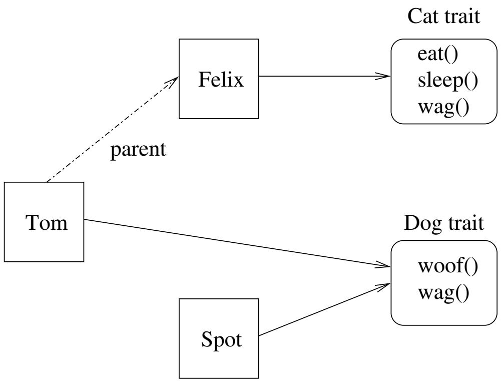
Objects, as usual, plus a special kind of object called a *trait*

Traits encapsulate *behaviours* of objects: the methods can be pulled out of the object and have a separate existence in a trait

Thus we can reuse behaviour independently of the parent hierarchy

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

Object Oriented Languages
Traits



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

Object Oriented Languages

Traits

- an object contains its own attributes and links to a trait and (optionally) a parent

Object Oriented Languages

Traits

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

Object Oriented Languages

Traits

- an object contains its own attributes and links to a trait and (optionally) a parent
- attribute lookup is via the object
- if there is an applicable method in the trait, use it, otherwise pass to the parent

Object Oriented Languages

Traits

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

Object Oriented Languages

Traits

- an object contains its own attributes and links to a trait and (optionally) a parent
- attribute lookup is via the object
- if there is an applicable method in the trait, use it, otherwise pass to the parent
- creating a new object is done by direct construction or cloning
- developed as this allows independent sharing of behaviour

Object Oriented Languages

Traits

Traits have recently had a resurgence in popularity

Object Oriented Languages

Traits

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, Perl (roles), Ruby, Rust, Java, Go, Common Lisp

Object Oriented Languages

Traits

A trait was originally a collection of methods, but now means a variety of things, sometimes under different names

Object Oriented Languages

Traits

A trait was originally a collection of methods, but now means a variety of things, sometimes under different names

Often it now means a collection of method *signatures*, i.e., just the method names with the types of their arguments and result, no actual code

Object Oriented Languages

Traits

A trait was originally a collection of methods, but now means a variety of things, sometimes under different names

Often it now means a collection of method *signatures*, i.e., just the method names with the types of their arguments and result, no actual code

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

Object Oriented Languages

Traits

A trait was originally a collection of methods, but now means a variety of things, sometimes under different names

Often it now means a collection of method *signatures*, i.e., just the method names with the types of their arguments and result, no actual code

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

Traits are not exclusively in object centred languages; the parent link also optional; an object (or class) can attach to more than one trait

Object Oriented Languages

Traits

Java interfaces are list of signatures; Go also has interfaces

Object Oriented Languages

Traits

Java interfaces are list of signatures; Go also has interfaces

Java 8 introduced something like full traits with its *default interface methods*, i.e., some code

Object Oriented Languages

Traits

Java interfaces are list of signatures; Go also has interfaces

Java 8 introduced something like full traits with its *default interface methods*, i.e., some code

This is like traits in Rust: generally signatures, but allows some code to use as a default

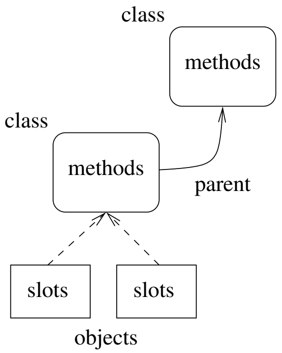
Object Oriented Languages

Traits

Exercise. Also read about Common Lisp *mixins*

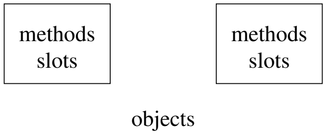
Exercise. Rust uses traits extensively: currently without inheritance through parent links, but with inheritance in the traits. Read about this

Object Oriented Languages



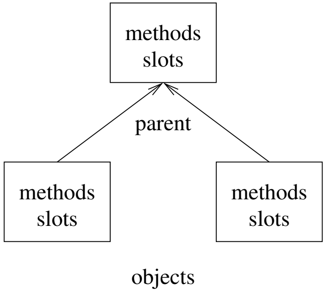
Class Centred

Object Oriented Languages



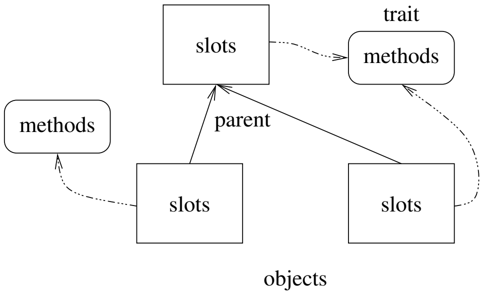
Prototype

Object Oriented Languages



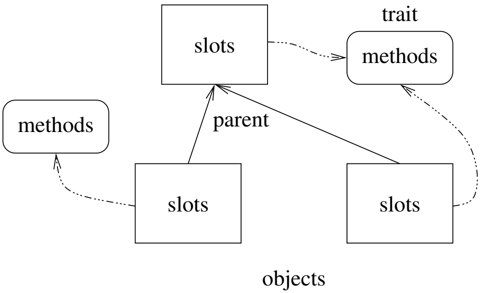
Delegation

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