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 functions (see later), 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 functions (see later), shared by instances
- attributes/slots/values defined in classes, attached to instances (or shared within 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 functions (see later), shared by instances
- attributes/slots/values defined in classes, attached to instances (or shared within classes)
- single or multiple inheritance defined through the relationships between the classes

Object Oriented Languages

Class Centred

And, of course, there are languages that don’t fit this simple view

Object Oriented Languages

Class Centred

And, of course, there are languages that don't fit this simple view

Exercise Ruby first looks for methods in the object; then the class; then parent classes. Read about this

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

OO languages are occasionally further divided by how they do methods:

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

Object Oriented Languages

Class Centred

OO languages are occasionally further divided by how they do methods:

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

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

Object Oriented Languages

Class Centred

OO languages are occasionally further divided by how they do methods:

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

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

(We ignore the extra complications of overloading here)

Object Oriented Languages

Class Centred

OO languages are occasionally further divided by how they do methods:

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

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

(We ignore the extra complications of overloading here)

This is the familiar “object dot method name” syntax

On the other hand, *generic functions* look more like normal functions: `plus(x,y)` or `(plus x y)`, and they choose a method depending on the types of both `x` *and* `y`

Object Oriented Languages

Class Centred

On the other hand, *generic functions* look more like normal functions: `plus(x,y)` or `(plus x y)`, and they choose a method depending on the types of both `x` *and* `y`

They are sometimes called *multimethods*

Note this is syntactic convenience. We might write

```
(x,y).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*

Note this is syntactic convenience. We might write

```
(x,y).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*

Exercise Compare with *pair types* and, more generally, *product types*

Object Oriented Languages

Class Centred

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

Object Oriented Languages

Class Centred

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

The methods are to be found in the generic function “object”, not a class

Object Oriented Languages

Class Centred

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

The methods are to be found in the generic function "object", not a class

As there may be more than one class involved

Object Oriented Languages

Class Centred

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

The methods are to be found in the generic function “object”, not a class

As there may be more than one class involved

For example, a multimethod `wombat` with methods defined on (`int`, `String`) and (`double`, `int`) — which class would it be defined on?

Object Oriented Languages

Class Centred

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

The methods are to be found in the generic function “object”, not a class

As there may be more than one class involved

For example, a multimethod `wombat` with methods defined on (`int`, `String`) and (`double`, `int`) — which class would it be defined on?

That doesn’t make sense, so we put them elsewhere, in the generic function named `wombat`, and not worry about attaching them to classes

Object Oriented Languages

Class Centred

And when defining new methods we will refer to the generic function, not to a class or classes

Object Oriented Languages

Class Centred

And when defining new methods we will refer to the generic function, not to a class or classes

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

Object Oriented Languages

Class Centred

And when defining new methods we will refer to the generic function, not to a class or classes

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

A (multi)method can depend on multiple classes

Object Oriented Languages

Class Centred

And when defining new methods we will refer to the generic function, not to a class or classes

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

A (multi)method can depend on multiple classes

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

Object Oriented Languages

Class Centred

Methods have two identifying things: their names and the class(es) of the object(s) they are called on

Object Oriented Languages

Class Centred

Methods have two identifying things: their names and the class(es) of the object(s) they are called on

Object receiver collects them by the (single) class

Object Oriented Languages

Class Centred

Methods have two identifying things: their names and the class(es) of the object(s) they are called on

Object receiver collects them by the (single) class

Generic functions collect them by the name

Object Oriented Languages

Class Centred

Methods have two identifying things: their names and the class(es) of the object(s) they are called on

Object receiver collects them by the (single) class

Generic functions collect them by the name

A choice of approach, but not symmetric, as a method only has one name, but can depend on more than one class

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*

In use, generic functions look a lot like normal functions, but are actually *collections* of methods

Object Oriented Languages

Class Centred

```
(defgeneric foo (x y))

(defmethod foo ((x <number>) (y <number>)) ...)

(defmethod foo ((x <integer>) (y <integer>)) ...)

(defmethod foo ((x <number>) (y <float>)) ...)

(defmethod foo ((x <float>) (y <integer>)) ...)

...
```

Object Oriented Languages

Class Centred

```
(defgeneric foo (x y))

(defmethod foo ((x <number>) (y <number>)) ...)

(defmethod foo ((x <integer>) (y <integer>)) ...)

(defmethod foo ((x <number>) (y <float>)) ...)

(defmethod foo ((x <float>) (y <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

Methods, functions and generic functions are different things

Functions and methods are different things

They both execute code, but they do it in very different ways

General Remark

Non-OO languages like C **do not have methods**, only functions

General Remark

Non-OO 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

Non-OO 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

They are not interchangeable words

General Remark

Non-OO 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

They are not interchangeable words

You can’t use a function a like a method

General Remark

A function is just some code

General Remark

A function is just some code

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

General Remark

A function is just some code

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

A function describes behaviour; a method is behaviour *plus* data

General Remark

A function is just some code

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

A function describes behaviour; a method is behaviour *plus* data

A generic function comprises zero or more methods

General Remark

A function is just some code

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

A function describes behaviour; a method is behaviour *plus* data

A generic function comprises zero or more methods

You may have also seen *closures*, which are different again

General Remark

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

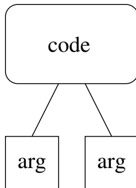
General Remark

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

Confusing these concepts will ensure loss of marks!

General Remark

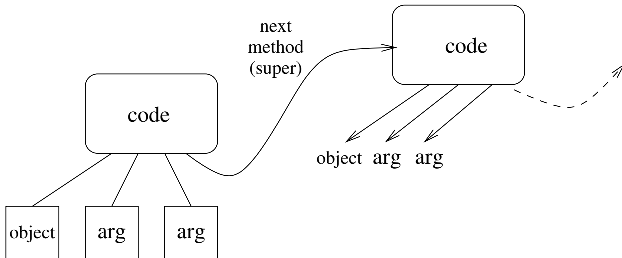
Functions just have code and data (arguments)



Function are code

General Remark

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



Methods have code, the object and next methods

General Remark

Within the body of the code the object is often referred to as a special argument named `self` or `this`; or it can be implicit, such as referring to a slot as simply “`x`” rather than “`this.x`” or “`self.x`”

General Remark

Within the body of the code the object is often referred to as a special argument named `self` or `this`; or it can be implicit, such as referring to a slot as simply "`x`" rather than "`this.x`" or "`self.x`"

In the case of an overridden method, the next most specific applicable method is sometimes available to call by `super` or `call-next-method` or similar

General Remark

Within the body of the code the object is often referred to as a special argument named `self` or `this`; or it can be implicit, such as referring to a slot as simply “`x`” rather than “`this.x`” or “`self.x`”

In the case of an overridden method, the next most specific applicable method is sometimes available to call by `super` or `call-next-method` or similar

In an object receiver language, the next method would be a method in a superclass of the class of the object

General Remark

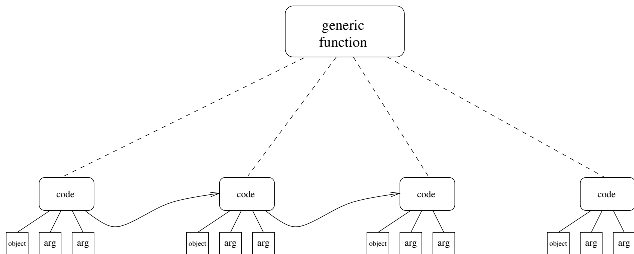
Within the body of the code the object is often referred to as a special argument named `self` or `this`; or it can be implicit, such as referring to a slot as simply “`x`” rather than “`this.x`” or “`self.x`”

In the case of an overridden method, the next most specific applicable method is sometimes available to call by `super` or `call-next-method` or similar

In an object receiver language, the next method would be a method in a superclass of the class of the object

In a generic function it is more complicated

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 parameter `this` or `self` or similar

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

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 parameter `this` or `self` or similar

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

So this is more name mangling

Object Oriented Languages

Hint: if you write
`x.foo()`
it's probably a method call

If you write
`foo(x)`
it's probably a function or generic function call

Object Oriented Languages

Hint: if you write
`x.foo()`
it's probably a method call

If you write
`foo(x)`
it's probably a function or generic function call

But not always: some object-receiver languages allow you to call a method with function syntax

Object Oriented Languages

Hint: if you write
`x.foo()`
it's probably a method call

If you write
`foo(x)`
it's probably a function or generic function call

But not always: some object-receiver languages allow you to call a method with function syntax

More on this later, but we need to introduce the other kinds of OO

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, Lua, 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, Lua, 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, Lua, 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, Lua, Self, ...

Typified by

- objects only, **no classes**
- methods attached to objects, i.e., stored in the object

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, Lua, Self, ...

Typified by

- objects only, **no classes**
- methods attached to objects, i.e., stored in the object
- 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, Lua, Self, . . .

Typified by

- objects only, **no classes**
- methods attached to objects, i.e., stored in the object
- 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, Lua, Self, ...

Typified by

- objects only, **no classes**
- methods attached to objects, i.e., stored in the object
- slots attached to objects
- direct construction and *cloning* to make instances
- **no inheritance**

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`

Object Oriented Languages

List Constructor in JavaScript

- `list`: the current object is referred to as `this`
- `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`
- `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`
- `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 with code defined elsewhere

Object Oriented Languages

List Constructor in JavaScript

- `list`: the current object is referred to as `this`
- `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 with code defined elsewhere
- `for ...`: more 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

Object centred languages are often dynamically typed, while class centred languages are often statically typed

Object Oriented Languages

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

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 while 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

Prototyping is then

one kind of object, no links

Prototyping is then

one kind of object, no links

JavaScript is an example of a *prototyping* language

Prototyping is then

one kind of object, no links

JavaScript is an example of a *prototyping* language

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

Object Oriented Languages

Prototyping

- creating a new object is done by direct construction or by *cloning*, i.e., copying an existing object: the *prototype*

Object Oriented Languages

Prototyping

- creating a new object is done by direct construction or by *cloning*, i.e., copying an existing object: the *prototype*
- an object contains its own attributes (slots) and behaviours (methods)

Object Oriented Languages

Prototyping

- creating a new object is done by direct construction or by *cloning*, i.e., copying an existing object: the *prototype*
- an object contains its own attributes (slots) and behaviours (methods)
- attribute and behaviour lookup are both by interrogating the object

Object Oriented Languages

Prototyping

- creating a new object is done by direct construction or by *cloning*, i.e., copying an existing object: the *prototype*
- an object contains its own attributes (slots) and behaviours (methods)
- attribute and behaviour lookup are both by interrogating the object
- 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
```

- used in *differential inheritance*: clone an object then add a new behaviour or modify an existing behaviour (or attribute)

Object Oriented Languages

Prototyping

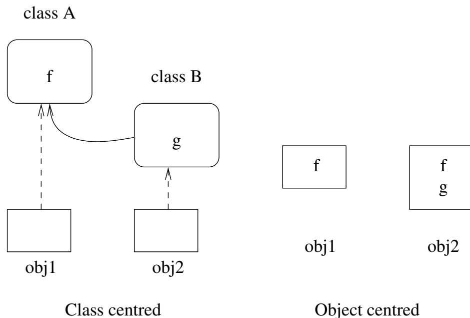
Though not a defining feature of prototyping, these languages often allow dynamic addition of attributes and behaviours to objects:

```
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 or modify an existing behaviour (or attribute)
- again, different from class-centred inheritance as the cloned object contains all its own methods and attributes

Object Oriented Languages

Prototyping



Class vs. object centred methods

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

In object centred, they are self-contained

Object Oriented Languages

Prototyping

- Prototyping gives less efficient code (requires runtime lookups) but more flexible programming

Object Oriented Languages

Prototyping

- Prototyping gives less efficient code (requires runtime lookups) but more flexible programming
- 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 (or parents) but not all their behaviours. Prototyping allows us to gather together whatever we need from wherever we want without constraint