Languages and Compilers
Dynamic Object-Oriented Languages

Adam Sampson
School of Design and Informatics
Abertay University

Today's plan

- This is a **languages** lecture
- “By the end of this module the student should be able to: [...] explain type systems and runtime support for modern programming languages”
- There are several ways to implement OO – let's start with the “traditional” approach

What do we mean by OO?

- Formally: you construct your program from **objects**, which have hidden **properties**
- You can send an object a **message**, which it responds to by running a **method**
  - … which can read/write properties, and send messages to other objects
- Introduced in Simula 67: a language for simulating real-world systems
  - Also had ideas of concurrency (see earlier lecture)
- Popularised and named by **Smalltalk**

Smalltalk

- Developed throughout 1970s at Xerox PARC, initially as a teaching language
- Language closely coupled to GUI – basically invented the modern IDE and OO programming
- Dynamically-typed
- “Pure” OO – everything is an object
  - To add 3 and 4, you send an “add 4” message to 3, and it replies with a 7 object
- An amazingly simple, elegant language – you can describe the syntax in one slide...

Smalltalk in one slide

- **Literals:** $c 'string' 42 1.234 #(1 2 3 4) #sym
- **Comments:** “This is a comment”
- **Declare local variables:** | var1 var2 |
- **Assignment:** var := expr. var ← expr.
- **Send message:** obj msg. obj withArg: arg.
  obj withTwo: arg args: arg. (remember this from Algol 60?)
  obj firstMsg; secondMsg.
  (obj1 + obj2) / obj3. (nearly any symbol as operator)
- **Answer message (return):** ^ expr. ↑ expr.
- **Blocks (anonymous functions):** [*some code*]
  [:arg1 :arg2 | *some code*]

Smalltalk in practice

- No syntax for defining classes/methods (in ANSI standard Smalltalk) – you use the GUI for this

```
exampleWithNumber: x
    "This is a small method that illustrates every part of Smalltalk
    method syntax [...]"
    |y|
    true & false not & (nil isNil) ifFalse: [self halt].
    y := self size + super size.
    #($a #a 'a' 1 1.0)
        do: [:each | Transcript
                        show: (each class name);
                        show: (each printString);
                        show: ' '].
    ^ x < y
```

Smalltalk in practice

- No syntax for defining classes/methods (in ANSI standard Smalltalk) – you use the GUI for this

```
exampleWithNumber: x
    "This is a small method that illustrates every part of Smalltalk
    method syntax [...]"
    |y|
    true & false not & (nil isNil) ifFalse: [self halt].
    y := self size + super size.
    #($a #a 'a' 1 1.0)
```

How you do an **if** (or **while** or ...):
send a message with a block to a boolean object!

true and *false* implement ifFalse: in different ways;
true does nothing, *false* executes the block.

Smalltalk in practice

- No syntax for defining classes/methods (in ANSI standard Smalltalk) – you use the GUI for this

This is a loop: you send a message with a block to an array object.

do: someBlock calls *someBlock* on each item in the array, with the item as an argument.

```
examp
    "Thi                                                    lk
    met
    |y|
    true
    y := self size + super size.
    #($a #a 'a' 1 1.0)
        do: [:each | Transcript
                        show: (each class name);
                        show: (each printString);
                        show: ' '].
    ^ x < y
```

Smalltalk in practice

- No syntax for defining
  standard Smalltalk) – y

e
d that lk

Dynamically-typed: our array can contain several objects of different types.

Standard library has a wide selection of objects and classes. Everything inherits from *Object* which gives standard methods like *class*.

Transcript is the output window. *Transcript show: x* prints *x* to the window.

```
(                sNil)
y := self size + super size.
#($a #a 'a' 1 1.0)
    do: [:each | Transcript
                        show: (each class name);
                        show: (each printString);
                        show: ' '].
^ x < y
```

Responding to messages

- Smalltalk is an example of an OO language that uses **dynamic dispatch**
  - Python, Ruby, Objective C are more recent examples, all strongly influenced by Smalltalk
- The language decides **at runtime** which method should be run when a message is sent to an object
- *3 printString. 'hello' printString.*
  Same message – different methods (int/string)
- *#(3 'hello') do: [:x | x printString].*
  The block can't know which method it's invoking until runtime...

Implementing dynamic dispatch

- Let's look at **Lua** (1993—present) as an example
  - `https://www.lua.org/`
- Dynamically-typed, dynamic-dispatch language
- Designed for embedding into other applications
  - Used for scripting by many game engines, Lightroom, Reaper, VLC, Wireshark...
- Excellent manual: a really good overview of how the language works, for both programmers and language implementors

Lua's primitive types

- Lua has numbers, strings, booleans, etc... and:
- “The type **table** implements associative arrays, that is, arrays that can be indexed not only with numbers, but with any Lua value […] Tables are the sole data-structuring mechanism in Lua”
- “In particular, because functions are first-class values, table fields can contain functions. Thus **tables can also carry methods**”

Lua: sending a message

```
breakfast:add(“Weetabix”)
```

... which is syntactic sugar for …

```
breakfast.add(breakfast, “Weetabix”)
```

… which in turn is syntactic sugar for ...

```
breakfast[“add”](breakfast, “Weetabix”)
```

- An object is a table containing methods
- To send a message, look up the message in the table...
- … and then call the resulting function

Lua: defining a method

```
function Meal:add(food) …
```

... which is syntactic sugar for …

```
function Meal.add(self, food) …
```

... which is syntactic sugar for …

```
Meal[“add”] = function (self, food) ...
```

- A method is a function
- … with an extra argument to represent the object that the method is operating upon
  - You see this in other languages too – e.g. Python's `self` (explicit), C++'s `this` (implicit)

OO in Lua vs. other languages

- What we've just seen is actually pretty typical of how dynamic-dispatch OO works in most programming languages
- Each object has a **table of methods**, and sending a message means **looking up the method** and **calling the resulting function**
- … although Lua makes it a bit more visible than most languages!

Lua: classes vs. objects

- In practice, you don't normally need every object to have its own table of methods
  - Every Integer has exactly the same methods...
- Lua objects have a “metatable”
  - Methods not found in the object can be looked up via the metatable
  - Objects can share metatables
  - Inheritance is also handled this way

Classes vs. prototypes

- Lua and Javascript are **prototype-based** languages: each object has its own table
  - To define a class, you create a **template object** (*Meal*) which you then clone to create instances (so *breakfast* starts as a copy of *Meal*)
- Most dynamic languages have a table of methods for each **class**; each object refers to the class
  - e.g. in Python: `someobj.__class__`; in Smalltalk, `someobj class.`
  - Cleaner model for inheritance: each class has a reference to the superclass
  - “The Class object of the Object class...” (!)

OO terminology

- Smalltalk: “send `getHeight` message to object X, and it'll answer with its `height` property”
- C++: “call `getHeight` member function on object X, and it'll return its `height` member variable”
- Java (etc.): “call `getHeight` method on object X, and it'll return its `height` instance variable”

- The same idea expressed three ways – Smalltalk is describing the *concept*, C++ is describing the *implementation*
- Message ≠ method – a distinction being lost!

Pros and cons of dynamic dispatch

- **Pro:** it's pretty simple to implement!
- **Pro:** varying the types of objects becomes idiomatic – e.g. 2 ** 4 returns an Integer; 2 ** 99999 can return a BigInteger
- **Pro:** allows introspection and dynamic manipulation of methods, e.g. creating methods on the fly as they're required
- **Con:** it's usually **slow** – because every message requires a lookup to find the right method
  - A smart compiler can generate specialised code for particular types – e.g. Smalltalk compilers know about the true/false types and ifTrue/ifFalse...

Static dispatch

- A more efficient approach: decide at compile time which method will be executed for each message
- C++ uses this model by default: if you call a method on a base class, you get the base class's version – even if it's been overridden
- Fine if you're not using inheritance – and you can specify methods as `virtual` to get dynamic dispatch with a method table for each class

> “Actually, I made up the term 'object-oriented', and I can tell you I did not have C++ in mind.”
>
> – Alan Kay (designer of Smalltalk), 1997
>
> `https://www.youtube.com/watch?v=oKg1hTOQXoY`

Any questions?

- We've seen how dynamic dispatch is typically implemented, and some [dis]advantages
- I'd definitely recommend reading the **Lua manual**

  `https://www.lua.org/manual/`
- You may also like to have a play with **Squeak**, a FOSS modern implementation of Smalltalk

  `http://squeak.org/`

- **Next week:** runtime systems, and lazy evaluation