Programming Languages

Object-oriented programming

Different ways of conceiving programming

The paradigms we have seen are founded on logic:

Functional programming	programs are proofs.
Logic programming	programs are formulas.

Object-oriented programming

It draws inspiration from various other disciplines. For example:

- Biology: adaptability and resilience of living organisms.
- Architecture: designing "cathedrals instead of pyramids".

Programs are made up of components, called **objects**, that interact by exchanging **messages**.

The physical or conceptual entities of the problem domain that one wants to model are represented as objects.

The behavior of objects should faithfully reflect those aspects of the "real-world" entities that interest us.

Brief historical perspective

OOP emerged around 1970 to abstract common techniques from procedural programming:

1. Passing records[1] to allow reentrant code.
   (A superior alternative to global variables).
2. Grouping functions into modules.
3. Polymorphism through indirection:
   a record contains pointers to functions that manipulate it.

Some highly influential object-oriented languages:

- Simula . . . . Dahl & Nygaard, ∼1967
- Smalltalk . . . . Kay, ∼1972
- Self . . . . Ungar & Smith, ∼1987

[1] I.e., *tuples* or `structs`.

Objects and messages

1. An OO environment is composed of **objects**.
2. An object can send a **message** to another. A message represents a request to the receiving object, to carry out one of its operations.
3. The **interface** of an object is the set of messages it is capable of responding to.
4. A **method** is the procedure an object uses to respond to a message. That is, it is the effective implementation of the operation requested by the message.
5. The way an object carries out an operation may depend on **external collaborators**[2] as well as its internal **state**, given by a set of **internal collaborators**[3].

[2] Also called *parameters* or *arguments* of the received message.

[3] Also called *attributes* or *instance variables* of the receiving object.

Objects and messages

Example

An object representing a 5×2 rectangle:

Interface:	**area**
Attributes:	`width` . . . . $\langle$object representing 5$\rangle$
	`height` . . . . $\langle$object representing 2$\rangle$
Methods:	**area**
	`^ width * height`

Encapsulation

Encapsulation principle

One can only interact with an object through its interface.
The internal state of an object is inaccessible from the outside.

Consequences of encapsulation

1. Switching between two representations of the same entity does not modify the observable behavior of the system.
   Example. A set of integers can be represented with a linked list or a balanced binary tree, without the user noticing a difference in behavior.
2. *“Duck typing”.* An object can be swapped for another that implements the same interface.
   Example. If I expect to interact with a searcher that responds to the message “`search: text`”, I could be provided with an object that responds to that message and also keeps usage statistics.

Variants of object orientation

OO environments can have different characteristics:

- Synchronous vs. asynchronous message sending.
- Message sending with return vs. without return.
- Mutable vs. immutable objects.
- Class-based vs. prototype-based.
- Single inheritance vs. multiple inheritance.
- ...

The environment we will use (Smalltalk) has typical characteristics.

Classes and instances

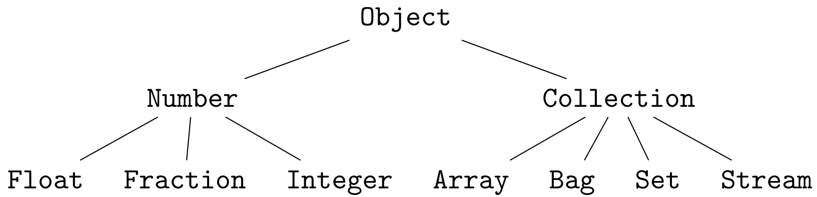
Every object is an **instance** of some **class**.

- A class is an object that abstracts the common behavior of all its instances.
  Example. `(1 @ 2)` is an instance of the class `Point`.
- All instances of a class have the same attributes.
  Example. All instances of `Point` have attributes `x` and `y`.
- All instances of a class use the same method to respond to the same message.
  Example. The messages `(1 @ 2) rho` and `(3 @ 4) rho` are resolved with a method implemented in the class `Point`.

  We will see more about *method dispatch* later.

Subclassing and inheritance

Each class is a **subclass** of some other class.
Classes are structured in a hierarchy. For example:



A class **inherits** all methods from its superclass.

A class may choose to **override** a method defined in the superclass with a more specific one.

Some classes are meant to abstract the behavior of their subclasses, but have no instances (e.g., the class `Number`). These are called **abstract** classes.

Basic examples

Let's evaluate the following commands:

1. `1 + 2`
2. `1 + 2 * 3` (Be careful with precedence)
3. `1 class`
4. `1 class superclass`
5. `3 squared / 2 squared`
6. `a := Array new: 10`
7. `a at: 1 put: 'hello'`
8. `a at: 1`

(Let's play a bit with the environment)

Let's define a class `Pair` and the following methods:

1. `Pair x: anObject y: anotherObject` — constructs a pair.
2. `pair x`, `pair y` — project the components.
3. `pair + anotherPair` — adds two pairs.
4. What happens if we nest pairs and use the addition?

Exercises

1. Add to the class `Collection` a method `map: aBlock`. Returns a collection, of the same “species”, that results from applying the block to each element of the receiving collection.
2. Add to the class `Collection` a method `minimize: aBlock`. Returns the minimum value of evaluating the block on the elements of the receiving collection.

Syntax of expressions and commands

Smalltalk is an **object-oriented environment**.
It is not quite appropriate to think of it as a *language*.
But we need a concrete syntax to describe methods.

"Minimal" syntax of expressions and commands

Expr	::=	`x`	*local name*
	|	`X`	*global name*
	|	*Expr* `msg`	*unary message*
	|	*Expr* ⟨`op`⟩ *Expr*	*binary message*
	|	*Expr* `msg1:`Expr_1 ...`msgN:`Expr_N	*keyword message*
	|	`x :=` *Expr*	*assignment*
Cmd	::=	*Expr*	*expression*
	|	`^`*Expr*	*return*
	|	*Expr*`.` *Cmd*	*sequence*

Local names refer to variables, attributes, and parameters.
Precedence: unary messages > binary messages > keyword messages.

Smalltalk syntax

Some other syntactic elements:

- Local variables are declared with `|x1 ... xn|`.
- Messages can be chained with “;”.
- There are six reserved words:

  `nil    true    false    self    super    thisContext`
- Notation for various literal types is included:
  - Numeric constants: `29`, `-1.5`, ...
  - Characters: `$a`, `$b`, `$c`, ...
  - Symbols: `#hello`, . . .
  - Strings: `'hello'`, . . .
  - . . .

Polymorphism

The same operation can work with objects that implement the same interface in different ways.

This feature of OOP can be used to build generic programs, that operate on objects regardless of their specific characteristics.

Example

```
z := OrderedCollection new.
z add: Cat new; add: Dog new; add: Duck new.
z do: [:animal | animal speak].   "meow woof quack"
```

- It is not necessary to do a `switch/case` to analyze what species of animal it is.
- Each object implements its own method to respond to the message “`speak`” in the appropriate way.

Polymorphism and control structures

Smalltalk has no control structures (`if`, `for`, `while`, etc.). These behaviors are implemented with message sends.

Let's see how to manually implement a conditional.

1. Let's define the following class hierarchy:

$$\begin{tikzcd}
 & \texttt{TruthValue} \arrow[dl, dash] \arrow[dr, dash] & \\
\texttt{True} & & \texttt{False}
\end{tikzcd}$$

2. Let's define methods to implement the message:

   ```
   aTruthValue ifTrue: x ifFalse: y
   ```

3. Let's define methods to implement the message:

   ```
   aTruthValue not
   ```

   leveraging polymorphism.
4. What happens if we evaluate the following command?

   ```
   True new ifTrue: 1
            ifFalse: Missile new launch
   ```

Blocks

A *block* or *closure* is an object that represents a command, i.e., a sequence of message sends.

The syntax is extended:

Expr	::=	...	
	|	`[`*Cmd*`]`	*block without parameters*
	|	`[:x1 ... :xN |`*Cmd*`]`	*block with parameters*

Blocks without parameters can be evaluated with the message:

```
block value
```

Blocks with N parameters can be evaluated with the message:

```
block value: arg1 value: arg2 ... value: argN
```

Example

1. `[1 + 1] value`
2. `[:x :y | x + y] value: 1 value: 2`
3. Let's modify the conditional to work with blocks.

Example — tree traversal

1. Let's define the following class hierarchy:

$$\begin{tikzcd}
 & \texttt{BinaryTree} \arrow[dl, dash] \arrow[dr, dash] & \\
\texttt{Nil} & & \texttt{Bin}
\end{tikzcd}$$

   and the following methods, with the expected semantics:

   ```
   Nil new
   Bin left: aTree root: data right: anotherTree
   ```

2. Let's define methods to implement the following message, which receives a single-parameter block and executes it on all elements of the tree following an inorder traversal:

   ```
   tree do: block
   ```

3. Let's define a method to implement the message:

   `tree size` (implement it using `do:`)

Two interesting points:

- Blocks can mutate the state of captured variables.
- The `size` method works for any collection that accepts the message `do:`.

Handling misunderstood messages

What happens if we evaluate the following command?

```
10 countTo: 20
```

- The object `10` receives a message `doesNotUnderstand: msg`.
- `msg` is an instance of the class `Message` that *reifies* the message "countTo: 20".

Inheritance and code reuse

Let's define a class `Robot` with the following interface:

1. `robot initialize` — initializes it at position `0@0`.
2. `robot position` — returns the current position.
3. `robot move: vector` — modifies the current position by adding a vector (which responds to messages `x` and `y`).

Now let's define a subclass `RobotWithUndo`:

1. `robot undo` — undoes the last move[4].

We are forced to override the methods `initialize` and `position`. But we don't want to copy and paste the code.

Super

The reserved word `super` refers to the same object as `self`. But `super m` indicates that the search for the method implementing the message `m` should start from the superclass of the current class.

[4]We can use `OrderedCollection`, `add:` and `removeLast`.

Method dispatch algorithm

Input

O: object to which a message is to be sent.

S: selector of the message to be sent. (E.g.: `at:put:`)

C: class in which to start searching for the method.

Output

M: method that should be executed to respond to the message,

or `NotUnderstood` if it does not exist.

Procedure

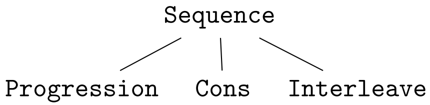
1. If C defines a method M for S, return M.
2. If not, let C' be the superclass of C.
   - 2.1 If C' is `nil`, return `NotUnderstood`.
   - 2.2 If not, set $C := C'$ and go back to step 1.

Method dispatch algorithm

- In general, when a message is sent, the method dispatch algorithm is used with C being the class of the object O.
- **Special case:**
  when a message is sent to `self`, O is the same receiving object.
- **Exception:**
  when a message is sent to `super`, O is the same receiving object while C is the superclass of the class of the method containing the message send to `super`.

Exercise — Streams

A *stream* is an object that represents an infinite sequence. It accepts a message `next`, which returns the current element and advances to the next element. We define a class hierarchy:



1. `Progression from: x applying: block` — constructs a *stream* that has `x` as its first element and computes the next element using the block.
2. `Cons head: anElement tail: aStream` — extends the given *stream* with an element at the head.
3. `Interleave between: s1 and: s2` — constructs a *stream* that alternates between the elements of `s1` and those of `s2`.
4. (More difficult). Define a method to implement the message `split`, which returns two *streams* that, when interleaved, would result in the original *stream*.

¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ? ? ? ? ? ? ? ? ?

Recommended reading

Chapters 1–4 of the book by Goldberg and Robson.
Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*.
Addison-Wesley, 1983.