Smalltalk

an object-oriented programming language

Object-Orientation

Basic ideas:

- **data abstraction (may or may not hide information)**
- **inheritance**
  reuse of implementations
- **subtyping**
  reuse of interfaces (“protocols”)
- **(disciplined) dynamic binding or late binding of functions**
  (aka open recursion)

Key goal is reuse

Object-oriented languages

- Simula (1967)
- Smalltalk (1980)
- LISP Flavors, Loops, CLOS,...
- C++, Objective C,...
- Eiffel, OWL,...
- PostScript in NeWS
- Modula-3, Oberon, Ada 95,...
- Python, Java, C#
- ⋮

The current language fad (since about 1990)

Object-oriented terminology

Term	Idea
class	type, encapsulation of data
object, instance	value
method	operation on an ADT
message	identification of an operation
message send	invocation
protocol, interface	valid messages
instance variables, fields	state
class variables	per-class state

Smalltalk

Pure object-oriented language:

- **Everything is an object**
- **Even values like `true`, `false`, `3`, `92` are objects**
- **Classes are objects!**
- **There are no functions: only methods on objects**

Limited control structures:

- **message passing**
- **sequential execution**

(Full Smalltalk also has “nonlocal return.”)

Smalltalk does not have `if` or `while` statements

- **they are simulated with objects (using CPS, continuation passing style)**

A bit like λ: magic is in the basis

Concrete syntax of μSmalltalk in EBNF

toplevel	$\longrightarrow$	***exp***
	$\mid$	**(use *file-name*)**
	$\mid$	**(val *variable-name exp*)**
	$\mid$	**(define *block-name* (*formals*) *exp*)**
	$\mid$	**(class *subclass-name superclass-name***
		($\{$*instance-variable-name*$\}$) $\{$*method-defn*$\}$)
method-defn	$\longrightarrow$	**(*method-defn-head method-name method-defn-tail*)**
method-defn-head	$\longrightarrow$	**method $\mid$ classMethod**
method-defn-tail	$\longrightarrow$	**(*formals*) $[$(locals *temporaries*)$]$ $\{$*exp*$\}$**
	$\mid$	**primitive *primitive-name***
formals	$\longrightarrow$	$\{$***formal-parameter-name***$\}$
temporaries	$\longrightarrow$	$\{$***temporary-variable-name***$\}$

Concrete syntax of μSmalltalk (continued)

exp	$\longrightarrow$	*variable-name*
	$\mid$	(set *variable-name exp*)
	$\mid$	(begin $\{$*exp*$\}$)
	$\mid$	(*message-name exp* $\{$*exp*$\}$)
	$\mid$	(block ($\{$*argument-name*$\}$) $\{$*exp*$\}$)
	$\mid$	[$\{$*exp*$\}$]
	$\mid$	*literal*
literal	$\longrightarrow$	*integer-literal*
	$\mid$	*#name*
	$\mid$	#($\{$*array-element*$\}$)
array-element	$\longrightarrow$	*integer-literal*
	$\mid$	*name*
	$\mid$	($\{$*array-element*$\}$)

Abstract syntax

μSmalltalk expressions (roughly speaking):

```
datatype exp = VAR     of name
             | SET     of name * exp
             | BEGIN   of exp list
             | SEND    of name * exp * exp list
             | BLOCK   of name list * exp list
             | LITERAL of rep
```

`VAR`, `SET`, and `BEGIN` as in Impcore.

`SEND` is message passing:

- Method is specified by name
- Always sent to a receiver
- Optional arguments
  (number determined by arity of name)

`BLOCK` and `LITERAL` are special objects.

Definition of `rep` (literal value representation) will be given later.

Protocol: the interface to an object

Object's protocol $\equiv$ set of messages it can respond to

Protocol for all objects (defined on class `Object`) is given on the next page

Note arities:

- **Symbolic messages: arity 1**
- **Keyword messages: arity is number of colons**

Protocol for all objects

`isKindOf: aClass`	Receiver inherits from arg?
`isMemberOf: aClass`	Receiver's class is arg?
`= anObject`	Equality
`!= anObject`	Inequality
`isNil`	Receiver is nil?
`notNil`	Receiver is not nil?
`print`	Print receiver.
`println`	Print receiver, then newline.
`error: aSymbol`	Error message
`subclassResponsibility`	Missing method

Simple examples

Every object inherits methods from `Object`

```
-> (+ 3 5)
8
-> (isKindOf: 3 Object)
<True>
-> (isMemberOf: 3 Object)
<False>
-> (isNil 3)
<False>
-> (isNil nil)
<True>
-> (println nil)
nil
nil
```

Simple examples (continued)

```
-> true
<True>
-> True
<class True>
-> false
<False>
-> False
<class False>
-> Object
<class Object>
```

Inheritance and Subtyping

“T inherits from Super”	Smalltalk, C++
“T is a subtype of Super”	Modula-3
“T extends Super”	Oberon, Java

T is the

- subclass
- derived class
- child class
- subtype

Super is the

- superclass
- base class
- parent class
- supertype

Inheritance

T inherit's Super's

- **state (instance variables)**
- **operations (methods)**

and therefore, also inherits

- **protocol**

Supertype's methods can be

- **"redefined" (Smalltalk)**
- **"overridden" (Modula-3)**

Subtyping or Polymorphism: may use a T wherever a Super is expected

Making subtyping precise

Subtyping is always *transitive*:
If τ_1 is subtype of τ_2 and τ_2 is subtype of τ_3, then τ_1 is subtype of τ_3.
Subtyping is frequently *reflexive* and *antisymmetric*.
(partial order)

Key is the subsumption rule or principle of substitution or is-a relationship:
If e is instance of τ_1 and τ_1 is subtype of τ_2, then e is instance of τ_2.
(This is implicit subsumption: no cast needed)

In Smalltalk, τ_1 is subtype of τ_2 iff protocol τ_1 contains every message in protocol τ_2 (and behaves suitably).
(not checked in dynamically typed system)

In C++, Modula-3, and Java, τ_1 is subtype of τ_2 iff τ_1 is subclass of (that is, inherits from) τ_2.

Decoupling subtyping from inheritance

Unusual in statically typed O-O languages

Smalltalk is dynamically typed, using width subtyping:

A subtype understands *more* messages:
If protocol τ_2 can respond to messages $\{m_1, \ldots, m_n\}$ and protocol τ_1 can respond to messages $\{m_1, \ldots, m_n, \ldots, m_{n+k}\}$, then τ_1 is subtype of τ_2.

If an object understands messages $m_1, \ldots, m_n$, and possibly more besides, you can use it wherever $m_1, \ldots, m_n$ are expected