SELF: The Power of Simplicity

David Ungar, Stanford
Randall B. Smith, Xerox PARC

OOPSLA 87
1967 - Simula67 by Dahl and Nygaard
1980 - Smalltalk-80 by Kay, Ingalls, and Goldberg
1985 - C++ by Stroustrup
1987 - Self by Ungar and Smith
1991 - Java by Gosling, Sheridan, and Naughton
1995 - Javascript by Eich
Is JavaScript popular? It’s hard to say. Some Ajax developers profess (and demonstrate) love for it. Yet many curse it, including me. I still think of it as a quickie love-child of C and Self.

Brendan Eich
https://brendaneich.com/2008/04/popularity/
principles of Self

- everything is an object
- prototypes, not classes
- all interactions are message passing
everything is an object

Smalltalk

primitive values

methods and closures

control structures

classes
((4 \text{ fac}) \text{ between: 10 And: 100}) \text{ ifTrue: “Hi!” False: ”Bye!”} 

\text{call “fac” method on 4, return 24} 

\text{call “between:And:” on 24 with args 10 and 100, return true} 

\text{call “ifTrue:False:” on true with args “Hi!” and “Bye!”, return “Hi!”}
objects are instances of classes

objects are clones of prototypes
objects are instances of classes

C++
Java
Smalltalk

objects are clones of prototypes

Javascript
Self
**classes**

- create objects by calling class constructor
- can modify methods only by subclassing
- classes need metaclasses, etc. (infinite regress!)

**prototypes**

- create objects by cloning prototype
- objects can have unique methods and fields
- no classes, no infinite regress
p := (Point new) x: 7 y: 9
p print

follow p’s class pointer, check if print is defined there

not defined there, so follow superclass pointer

found “print” in Object class! Invoke with receiver “p”
classes

```
p := (Point new) x: 1 y: 10
p print
```
follow p’s class pointer, check if print is defined there

not defined there, so follow superclass pointer

found “print” in Object class! Invoke with receiver “p”

to have different print method, need to create Point subclass
prototypes

p := (point clone) x: 7 y: 9
p print

does p have print method? no, so follow parent pointer to delegate

does Point delegate have "print"? no, so follow parent pointer to delegate

does Object delegate have print? yes, invoke with "p" as receiver
p := (point clone) x: 1 y: 10
p print

does p have print method? no, so follow parent pointer to delegate

does Point delegate have “print”? no, so follow parent pointer to delegate

does Object delegate have print? yes, invoke with “p” as receiver

to have special print method for p, define new slot in p -- no subclass needed!
activation as cloning

method invocation
clones prototype
activation record
field access and assignment are messages to current receiver (self)
state as behavior

\[ p . x \quad // \quad p.x \]

\[ p . x : 2 \quad // \quad p.x = 2 \]
example: points

root

traits

prototypes

_AddSlotsIfAbsent: (|
  traits = ()).
  prototypes = ().
|)
traits _AddSlotsIfAbsent:([cloneable=()])
traits cloneable _Define:()

    copy = (_Clone).

_example: points_
example: points

```
traits _AddSlotsIfAbsent: (|point=()|)
traits point _Define:(|
    parent* = traits cloneable.
    printString = …
    + aPoint = …
    - aPoint = …
|)
```
example: points

prototypes _AddSlotsIfAbsent (|point=()|)
prototypes point _Define:(|
  parent* = traits point.
  x <- 0.
  y <- 0.
|)

```
root

 traits

 prototypes

 point

 prototypes point _Define:(|
  parent* = traits point.
  x <- 0.
  y <- 0.
|)

 cloneable

 copy

 _Clone

 parent*
p

 x

 0

 x:

 0

 y

 y:

 0

 prototypes point _Define:(|
  x <- 0.
  y <- 0.
|)
```
example: points

```
((prototypes point) copy) x: 3 y: 4
```

```
parent* x: 0
y: 0

x: ⇐
y: ⇐

x 3
y 4

parent* x: ⇐
y: ⇐

X

Y

printString

[method]

_Clone

copy

 prototypes

 traits

 cloneable

 point

 root

```
is Self a good influence on modern languages?

what are the trade-offs of Self’s flexibility?

are there cases when simplicity should be abandoned?