Today

- Prototype-based languages
- Self
- JavaScript
Self

- David Ungar (Stanford) and Randall Smith (Xerox PARC)
- late 1980s

- Not a popular success
  - But, very influential
  - language design: JavaScript
  - implementation: Java, C#, JavaScript, most scripting languages

- Now at Version 4.4 (Linux and Mac)
- [http://selflanguage.org/](http://selflanguage.org/)
Smalltalk

- First “modern” OO language – Alan Kay, 1970s
  - Everything is an object (numbers, lists, procedures)
  - Garbage collection
  - Closures
- Exploratory programming
  - Dynamically typed
  - Add new code/evaluate expressions with no compilation required
- Try it: squeak.org
Smalltalk

• Class-based language
  • Every object has a class
    • defines structure of the object

• But, *everything* is an object
  • so, classes are also objects

• What is the class of a class?
Meta-regress

- **Object** is the base class of all objects
  - 3 is an instance of Object
- **Class** is the base metaclass of all classes
  - Integer is an instance of Class
- **Metaclass** is the class of all metaclasses
  - Class is an instance of Metaclass

- Metaclass is an instance of ... ?
Self: goal is simplicity

- Reduce number of concepts in the language
- no classes
- everything is an object
  - no builtin control structures (these are also objects)
  - procedures = prototypes of activation records
- no variables
  - no distinction between accessing a variable and sending a message (= calling a method)
Prototypes

• To create an object, **clone** an existing object (the **prototype**)

• Can **add** new methods
• Can **override** existing methods
• Can even **remove** methods
Inheritance

• Use **delegation** rather than inheritance

• Every object has one or more **parent** slots

• When sending a message (= calling a method), if object cannot handle the message, forward to the parent
Inheritance in Self

• Objects can have more than one parent
  • multiple inheritance!

• Parents can be changed at run time
  • dynamic inheritance!

• useful when developing/debugging code
• can refactor running code
Smalltalk vs. Self

**Smalltalk**

- (class)
- (superclass)
- (inst vars)
- (methods)

**Self**

- clone
- point
- how to clone objects

Figure 2. Object creation in Smalltalk and in Self.

To create a new point in Smalltalk, the new message is sent to the class `Point`. The new method—found in `Point`'s class's superclass—uses information in its receiver (`Point`) to define the size and format of the new object.

To create a new point in Self, the clone message is sent to the prototypical point. The clone method copies its receiver. Because the point slot resides in the root, any object can create a point.
Messages

- When message is sent, object is searched for **slot** with message name
- If none found, all parents are searched
  - Runtime error if more than one parent has a slot with the same name
- If slot is found, its contents evaluated and returned.
  - Runtime error if no slot found
Flexibility

Computing a value instead of storing it

<table>
<thead>
<tr>
<th>parent*</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>print</td>
<td>print objects</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parent*</th>
<th>+</th>
<th>add points</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>parent*</th>
<th>x</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>x:</td>
<td>←</td>
</tr>
<tr>
<td></td>
<td>y:</td>
<td>←</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parent*</th>
<th>x</th>
<th>random number generator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>y:</td>
<td>←</td>
</tr>
</tbody>
</table>

Shared state

<table>
<thead>
<tr>
<th>parent*</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>print</td>
<td>print objects</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parent*</th>
<th>+</th>
<th>add points</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>parent*</th>
<th>x</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x:</td>
<td>←</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parent*</th>
<th>y</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y:</td>
<td>←</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parent*</th>
<th>y</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y:</td>
<td>←</td>
</tr>
</tbody>
</table>
Activation records

Activation in SELF

shared behavior for all objects

print
clone

how to print objects
how to clone objects

shared behavior for points

parent*

(a point) + (another point)

(a point)

+ 

(arg)

<code>

parent* 

x: 3
y: 5
x: ←
y: ←

parent* 

x: 7
y: 9
x: ←
y: ←

self* 

arg
<code>

(clone x: x + arg x) y: y + arg y

Figure 5. This figure shows what happens when the point (3, 5) is sent the "+" message with argument (7, 9). Lookup for plus starts at (3, 5) and finds a matching slot in the object holding shared behavior for points. Since the contents of the slot is a method object, it is cloned, the clone's argument slot is set to the argument of the message, and its parent is set to the receiver. When the code for "+" executes, the lookup for x will find the receiver's x slot by following the inheritance chain from the current activation record. It will also find the contents of the "arg" slot in the same way. It is this technique of having the lookup for the implicit "self" receiver start at the current activation that allows local variables, instance variables, and method lookup to be unified in SELF.
Typing

• In a class-based language, every class defines a type

• Most prototype-based languages are dynamically typed

• Why?
Typing

• In a class-based language, every class defines a type

• Most prototype-based languages are dynamically typed
  • Why?
  • Difficult to describe the type succinctly (no class names)
Disadvantages of classes?

- Require programmers to understand a more complex model
  - To make a new kind of object, have to create a class first
  - To change an object, have to change the class

- But:
  - Does Self require programmer to reinvent structure?
  - Common to structure Self programs with traits: objects that simply collect behavior for sharing
Implementation

• Very high-level abstractions in the language
• Challenges:
  • Many function calls, small methods
  • Everything is a message, even variable access
  • No static type system
  • No class structure to enforce sharing
    • each object has its own copy of its methods

• Compiler must do a lot of work to optimize this code
  • adaptive run-time optimization
  • method inlining
• The students who worked on Self developed the technology in the HotSpot Java VM
  • Urs Hölzle: first VP of Engineering at Google, now VP of Operations
  • Lars Bak: developed the V8 JavaScript virtual machine in Chrome
Implementation

• Compiler must do a lot of work to optimize this code
  • adaptive run-time optimization
  • method inlining

• Side note:
  • The grad students working on Self formed a startup to build a VM for Smalltalk
  • Bought by Sun, evolved into HotSpot JVM
  • Urs Hölzle: VP of Operations at Google
  • Lars Bak: developed the V8 JavaScript VM in Chrome
JavaScript

- Brendan Eich
  - 1995 Netscape—remember them?
- OO language
- allows websites to run client-side code

- Standardized as ECMAScript
  - variants: JavaScript, JScript (MS), ActionScript (Adobe)
JavaScript

- Java-like syntax
- prototype-based object system

- object is a set of properties, a mutable map from strings to values
- a property that evaluates to a function is a method
Properties

- object is a mutable map from strings to values

  - obj.x = 1
  - equivalent to:
  - obj["x"] = 1

- assigning to a field that doesn’t exist creates the field
Functions

• Functions are just objects

• function (x) { return x + 1 }
  • returns an object whose call method increments its argument
Prototypes

• Each object has a single `__proto__` field that refers to another object

• property lookup:
  • search the object
  • if not found, search the prototype

• `__proto__` field initially refers to an empty object
Constructors

• Constructors are just functions
  • function C(x) { this.x = x; }

• Constructors have a **prototype** field
  • this is **not** the prototype of the function object
  • it’s the prototype of objects **created** by the constructor

• the constructor assigns its **prototype** to the **__proto__** field of the new object
• **new** creates an object based on the prototype field

• `new C(17)`
  • creates an empty object `z`
  • `set z.__proto__ = C.prototype`
  • calls `C` with `x=17`, `this=z`
  • returns `z`
List example

function List(v, n) { this.v = v; this.n = n; }

List.prototype.map = function(f) {
  return new List(f(this.v),
  this.n ? this.n.map(f) : null);
}

var l = new List(1, null);
delete(List.prototype.map);
List example

```javascript
function List(v, n) { this.v = v; this.n = n; }
List.prototype.map = function(f) {
    return new List(f(this.v),
        this.n ? this.n.map(f) : null);
}
```

[Diagram of JavaScript prototype chain]
function List(v, n) { this.v = v; this.n = n; }
List.prototype.map = function(f) {
    return new List(f(this.v),
        this.n ? this.n.map(f) : null);
}
var l = new List(1, null);
List example

function List(v, n) { this.v = v; this.n = n; }
List.prototype.map = function(f) {
    return new List(f(this.v),
    this.n ? this.n.map(f) : null);
}
var l = new List(1, null);
function List(v, n) { this.v = v; this.n = n; }
List.prototype.map = function(f) {
    return new List(f(this.v),
        this.n ? this.n.map(f) : null);
}
var l = new List(1, null);
delete(List.prototype.map);
Resources

• Download Self and play with the demo
  • selflanguage.org

• Flanagan, JavaScript: The Definitive Guide
  • most other JS books are not very good