Objects
Edward Z. Yang
What are the central ideas of OO?

- dynamic dispatch
- encapsulation
- subtyping
- inheritance

How can we understand OO with the tools of this class?

- by reducing objects to known concepts (Simula)
- by simplifying objects to a core idea (Smalltalk, Self)
What are the central ideas of OO?

dynamic dispatch • encapsulation • subtyping • inheritance
Anatomy of an object

send a message (method invocation)

hidden data

<table>
<thead>
<tr>
<th>msg1</th>
<th>method1</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>msgn</td>
<td>methodn</td>
</tr>
</tbody>
</table>

compare with AOTs: behavioral rather than structural
Dynamic dispatch

object → message (arguments)

operation (arguments) → operation

static

dynamic

object & message

compare: \( n_1 \rightarrow \text{add}(n_2) \) versus \( \text{add}(n_1,n_2) \)

(first-class functions = dynamic dispatch)
Encapsulation

public abstract view

private detailed view

message
Encapsulation

Interface

Implementation

what messages are understood
Subtyping

relation between interfaces

ColoredPoint
  x-coord
  y-coord
  color
  move
  change-color

Point
  x-coord
  y-coord
  move

if interface A contains all of interface B, then A objects can be used as B objects
Inheritance relation between implementation

class ColoredPoint extends Point {
    // reuse implementation of move
    void changeColor(...) {
        ...
    }
    ...
    }

Subtyping $\neq$ Inheritance

interfécis  implementation
OO principle:

Group data and code together
Comparative example: Shapes

Haskell

data Shape = Square Point Point 
| Circle Point Length

center :: Shape \rightarrow Point
move :: Shape \rightarrow Point \rightarrow Shape
render :: Shape \rightarrow IO ()
Comparative example: Shapes

Objects

Shape
- Rectangle
- Circle

Shape center move render

differing private implementations
How can we understand OO with the tools of this class?

- by reducing objects to known concepts (Simula)
- by simplifying objects to a core idea (Smalltalk, Self)
Simula

objects as activation records
Algol 60

Simula 67
- classes & subtyping
- a language for simulation
- Nygaard & Dahl

Smalltalk

C++

...
Recall: Activation Records

<table>
<thead>
<tr>
<th>access</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>2</td>
</tr>
</tbody>
</table>

(Nota: access link points to the "textually enclosing block instance")
Recall: Activation Records

Similar situation in manually memory managed languages (C, C++)

- return 2 -
Recall: Activation Records


“Objects already existed: they just needed to be freed from the stack discipline.”

Simula called these “blocks”

Need a GC!

<table>
<thead>
<tr>
<th>access</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>2</td>
</tr>
</tbody>
</table>
Objects in Simula

Class
Function that returns pointer to its activation record

Object
Activation record produced by call to class

Object access
Dot operator to access variables in record

statically typed!
Example: Find radius & center of circle passing through three distinct points.
Example: Find radius & center of circle passing through three distinct points
Example: Find radius & center of circle passing through three distinct points
Example: Find radius & center of circle passing through three distinct points.
Plan: Points, lines, circles \rightarrow objects

Point
  equals(aPoint) : boolean
  distance(aPoint) : real

Line
  parallelTo(aLine) : boolean
  meets(aLine) : REF(Point)

Circle
  intersects(aCircle) : REF(Line)
Plan: Points, lines, circles → objects

Point
  equals(aPoint): boolean
  distance(aPoint): real

Line
  parallelTo(aLine): boolean
  meets(aLine): REF(Point)

Circle
  intersects(aCircle): REF(Line)
class Point(x,y); real x,y;
begin
  boolean procedure equals(p); ref(Point) p;
    if p =/= none then
      equals := abs(x - p.x) + abs(y - p.y) < 0.00001
    real procedure distance(p); ref(Point) p;
      if p == none then error else
        distance := sqrt(( x - p.x )**2 + (y - p.y) ** 2);
end ***Point***

p :- new Point(1.0, 2.5);
q :- new Point(2.0, 3.5);
if p.distance(q) > 2 then ...
class Point(x,y); 
begin 
  real x,y;
  boolean procedure equals(p); ref(Point) p; 
  if p =/= none then 
    equals := abs(x - p.x) + abs(y - p.y) < 0.00001 
  real procedure distance(p); ref(Point) p; 
  if p == none then error else 
    distance := sqrt(( x - p.x )**2 + (y - p.y) ** 2); 
end ***Point***

p :- new Point(1.0, 2.5);
q :- new Point(2.0, 3.5);
if p.distance(q) > 2 then ...
class Point(x,y); real x,y;
begin
  boolean procedure equals(p); ref(Point) p;
  if p =/= none then 
    equals := abs(x - p.x) + abs(y - p.y) < 0.00001
  real procedure distance(p); ref(Point) p;
  if p == none then error else 
    distance := sqrt(( x - p.x )**2 + (y - p.y) ** 2);
end ***Point***

p :- new Point(1.0, 2.5);
qu :- new Point(2.0, 3.5);
if p.distance(q) > 2 then ...

return the value by assigning it
(some still-alive langs still have this syntax; e.g. Matlab, Fortran)

Special operator for reference assignment
class Point(x,y); real x,y;

begin
  boolean procedure equals(p); ref(Point) p;
    if p =/= none then
      equals := abs(x - p.x) + abs(y - p.y) < 0.00001
    real procedure distance(p); ref(Point) p;
      if p == none then error else
        distance := sqrt((x - p.x)**2 + (y - p.y)**2);
end ***Point***

p :- new Point(1.0, 2.5);
q :- new Point(2.0, 3.5);
if p.distance(q) > 2 then ...
Object representation

| access link | 1.0 |
| real x      | 2.5 |
| proc equals |     |
| proc dist   |     |

NB: Simula did not support closures; environment pointer for procedures defined in class “hard-coded” in spec. Like Algol 68, functions could be passed but not returned.
class Line(a, b, c); real a, b, c;
begin
  boolean procedure parallelto(l); ref(Line) l;
    if l =/= none then parallelto := ...
  end;
ref(Point) procedure meets(l); ref(Line) l;
begin real t;
  if l =/= none and ~parallelto(l) then ...
end;
real d;
d := sqrt(a**2 + b**2);
if d = 0.0 then error else
begin
  d := 1/d;
  a := a*d; b := b*d; c := c*d;
end;
end *** Line***
class Line(a, b, c); real a, b, c;

begin
    boolean procedure parallelto(l); ref(Line) l;
    if l /= none then  parallelto := ...

    ref(Point) procedure meets(l); ref(Line) l;
    begin real t;
        if l /= none and ~parallelto(l) then ...
    end;

    real d;
    d := sqrt(a**2 + b**2);
    if d = 0.0 then error else
    begin
        d := 1/d;
        a := a*d;  b := b*d;  c := c*d;
    end;
end *** Line***
class Line(a,b,c); real a,b,c;
begin
  boolean procedure parallelto(l); ref(Line) l;
  if l =/= none then parallelto := ...

  ref(Point) procedure meets(l); ref(Line) l;
  begin real t;
  if l =/= none and ~parallelto(l) then ...
  end;

  real d;
  d := sqrt(a**2 + b**2);
  if d = 0.0 then error else
  begin
    d := 1/d;
    a := a*d;  b := b*d;  c := c*d;
  end;
end *** Line***
Derived classes (inheritance)

class A
A class B
A class C
B class D

d := new D(...)
Virtual versus activation records

class A;
begin
  integer procedure f();
  begin f := 0 end;
  integer procedure g();
  begin g := f end;
end;

A class B;
begin
  integer procedure f();
  begin f := 1 end;
end;

b :- new B();
b.g % What is this value?
Virtual versus activation records

class A;
begin
  integer procedure f();
  begin f := 0 end;
  integer procedure g();
  begin g := f end;
end;

A class B;
begin
  integer procedure f();
  begin f := 1 end;
end;

b :- new B();
b.g % What is this value?
Virtual versus activation records

class A;
begin
  integer procedure f();
  begin f := 0 end;
  integer procedure g();
  begin g := f end;
end;

A class B;
begin
  integer procedure f();
  begin f := 1 end;
end;

b :- new B();

b.g % What is this value?  

It's Φ!
Simula 67: Summary

Hugely influential language, with

- Classes
- Objects
- Inheritance
- Subtyping
- Virtual methods
- Inner (combine parent code)
- Inspect/Qua (instanceof/cast)

Features for simulation

Event Loop
Simula 67: Summary

Some missing things...

Encapsulation (added later)
Self/Super (Smalltalk)
  (did have this(class))
Class variables (use globals instead)
Exceptions (whatever)
Simula 67: Summary

Class: Function that returns pointer to its activation record

Object: Activation record produced by call to class

Subtyping: By class hierarchy

Inheritance: By prefixing
a shift... to dynamically typed OO languages
Smalltalk

Everything is an object... even classes
Smalltalk

- Popularized objects
- Developed at Xerox PARC

"Take the hardest and most profound thing you need to do, make it great, and then build every easier thing out of it". That was the promise of LISP and the lure of lambda—[what it] needed was a better "hardest and most profound" thing. Objects should be it.

- Influenced by Simula, but very different
- All operations messages to objects

"The most powerful language in the world...in a page of code."

More history: http://worrydream.com/EarlyHistoryOfSmalltalk/
Revolutionary idea in the 1970s
Smalltalk: Both a PL & an operating system for non-programmers

Revolutionary idea in the 1970s
Smalltalk terminology

- **Object**: Instance of a class
- **Class**: Definition of the behavior of objects
- **Selector**: Name of message (method name)
  - (could be forwarded)
- **Message**: Selector + parameter values
- **Method**: Code to respond to message
- **Instance variable**: Data stored in object
- **Subclass**: Incrementally modified parent class
Smalltalk semantics

- Everything is an object
- Objects communicate by sending/receiving messages.
- Objects have their own state
- Every object is an instance of a class
- A class provides behavior for its instances
Instance messages and methods

\[ x: \text{xcoord} \quad y: \text{ycoord} \]
\[ x \leftarrow \text{xcoord} \]
\[ y \leftarrow \text{ycoord} \]
\[ \text{moveDx: } dx \quad \text{Dy: } dy \]
\[ x \leftarrow dx + x \]
\[ y \leftarrow dy + y \]
\[ x \quad ^\wedge x \]
\[ y \quad ^\wedge y \]
\[ \text{draw} \]

...code to draw point...

Selectors

\[ x:y: \text{is a mixfix operator} \]

\[ \text{pt x:2 y:3} \]

message
Instance messages and methods

\[
x: \text{xcoord} \quad y: \text{ycoord} \quad | \quad |
\]
\[
x \leftarrow \text{xcoord}
\]
\[
y \leftarrow \text{ycoord}
\]

moveDx: \ dx \quad Dy: \ dy \quad | \quad |
\[
x \leftarrow dx + x
\]
\[
y \leftarrow dy + y
\]
\[
x \quad \quad \uparrow x
\]
\[
y \quad \quad \uparrow y
\]

draw \quad | \quad |
\[
...\text{code to draw point}...
\]

Return

**Mutable assignment**

**Instance variables (private)**
instance variables are private always
## Point

<table>
<thead>
<tr>
<th>class name</th>
<th>Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super class</td>
<td>Object</td>
</tr>
<tr>
<td>class var</td>
<td>pi</td>
</tr>
<tr>
<td>instance var</td>
<td>x y</td>
</tr>
</tbody>
</table>

**class messages and methods**

```ruby
...names and code for methods...
```

**instance messages and methods**

```ruby
...names and code for methods...
```
Class messages and methods

newX:xvalue Y:yvalue   |   |
^ self new x: xvalue
     y: yvalue

newOrigin   |   |
^ self new x: 0
     y: 0

initialize   |   |
    pi <- 3.14159

Classes are objects too!
Send the "new" message to self object (Point class is object)

```smalltalk
newX:xvalue Y:yvalue  ||
^ self new x: xvalue
    y: yvalue
```

To newly created Point object, send (mixfix) message x:y:

```smalltalk
newOrigin  ||
^ self new x: 0
    y: 0
```

initialize  ||
pi <- 3.14159

self could be overloaded: always points to actual object
## Inheritance

<table>
<thead>
<tr>
<th>class name</th>
<th>Color Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super class</td>
<td>Point</td>
</tr>
<tr>
<td>class var</td>
<td></td>
</tr>
<tr>
<td>instance var</td>
<td>color</td>
</tr>
<tr>
<td>class messages and methods</td>
<td></td>
</tr>
<tr>
<td>newX:Y:Cl</td>
<td>... code...</td>
</tr>
<tr>
<td>instance messages and methods</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>11^color</td>
</tr>
<tr>
<td>draw</td>
<td>... code..</td>
</tr>
</tbody>
</table>
If a class is an object...

What's the class of a class?
What's the class of a class?

```
Point object
class    class
x       3
y       2
```

Smalltalk-76: "Class"

Trouble: all class methods (e.g., method to handle message to Point class must be put in Class)
What's the class of a class?

Smalltalk-80: Metaclasses
What's the class of a class?

Object → Object class

↑ subclass of

Point → Point → Point class

instance of

Smalltalk-80: Metaclasses
What's the class of a class?

Object \rightarrow \text{Object class} \quad \text{subclass of} \quad \text{Metaclass}

Point \rightarrow \text{Point} \rightarrow \text{Point class} \quad \text{instance of}

Smalltalk-80: Metaclasses
What's the class of a class?

Object → Object class

Point → Point

Point → Point class

subclass of

instance of

Metaclass

Metaclass class

Smalltalk-80: Metaclasses
What’s the class of a class?

Object → Object class

Point → Point

Instance of

subclass of

Metaobject

Metaobject class

Smalltalk-80: Metaclasses
What’s the class of a class?

Object \rightarrow \text{Object class}

\text{subclass of}

Point \rightarrow \text{Point}

\text{instance of}

\text{Metaclass class}
Smalltalk summary

Class
- create objects that share methods
- internally records dictionary, parent, ...

Objects
- created by class, has private state

Encapsulation
- public methods, private state

Subtyping: implicit

Inheritance: subclasses, self, super
Self

Everything is an object; NO classes
Self

- Prototype-based OO language

- Randal Smith (Xerox PARC) and David Ungar (Stanford)
  - Successor to Smalltalk'80
  - "Self: The power of simplicity" OOPSLA '87

- Influence
  - JavaScript
  - Advances in compilation (esp. Java)
Self

- Everything is an object
- Everything done by messages
- No classes
- No variables

“A language for Smalltalk runtime structures”
Self semantics

- Clone
- Send message
- Add new slot
- Replace old slots
- Remove slots
Self semantics

Objects consist of named slots

Slots contain code to do various things...

Data Return contents upon eval

Assignment Set value of slot

Method Code to run

Parent Inherit slots from other object
Messages

- When message is sent, search for slot with name
- If not found, recursively search via *parent pointer
- When found, evaluate code in slot and return result
- Self points to message receiver
"Variables" are not dumb!

```
parent*   ...
+          \langle add points \rangle

<table>
<thead>
<tr>
<th>parent*</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>xi</td>
<td></td>
<td>←</td>
</tr>
<tr>
<td>yi</td>
<td></td>
<td>←</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parent*</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>\langle RNG \rangle</td>
</tr>
<tr>
<td>y</td>
<td>∅</td>
</tr>
<tr>
<td>yi</td>
<td>←</td>
</tr>
</tbody>
</table>
```
Object creation

- To create, copy old one (prototype)
- Add/modify/remove methods
- You can even change parent pointer
JavaScript prototypes

To create:
- `new F();`
  `__proto__ = F.prototype`
- `Object.create(p)`
  `__proto__ = p`

Not everything is a message
this [keyword](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) only defined when "received message"
Self versus classes

✓ Simpler! Avoids meta-classes

✗ Less structure; programmer discipline

(ESG adding classes, which desugar to prototypical inheritance)
What are the central ideas of OO?

dynamic dispatch • encapsulation • subtyping • inheritance

How can we understand OO with the tools of this class?

by reducing objects to known concepts (Simula)
b by simplifying objects to a core idea (Smalltalk, Self)