251 map check

- Limits of computability
- Function-oriented building blocks (in Racket)
  - Consider: implementation (GC), immutability, metaprogramming
- Types, patterns, and abstraction (in Standard ML)
  - Consider: software engineering implications
- Evaluation choices and orders
  - Consider: implementing environments, interpreters, delay
- Object-oriented building blocks (in Smalltalk, Java, Scala)
  - Consider: implementing objects
  - Design trade-offs: composition/classification, abstraction/extensibility
- Parallelism and concurrency
- Synthesis: unified models, problem-solving as language design

Language design principles

- Common threads in Lisp, ML, Smalltalk, Java?
- Smalltalk:
  - Surprising features of language definition?
  - Principles we have [not] seen before?
- Steele's plan for Java:
  - Issues with small/large languages?

Object-oriented programming

and key language semantics to support it

OO essence:

- Program design principles:
  - Objects model state/behavior of real-world entities/concepts
  - Organization by classification and encapsulation
  - Reuse via extensibility
- Key semantics:
  - Late binding / dynamic dispatch
  - Substitutivity and subtyping
  - Inheritance or delegation

Will contrast function-oriented principles/semantics later.

Roots: modeling the world

Simula 67
- Ole-Johan Dahl, Kristen Nygaard, Norwegian Computing Center
- Simulating social and industrial systems
- Objects, classes, inheritance, and subtyping

Smalltalk
- 1970s, Alan Kay, Adele Goldberg, Dan Ingalls, Xerox PARC
- Integrated programming system / interface, personal computers
- Everything is an object, communicating by messages (methods)

CLU:
- 1970s, Barbara Liskov, MIT
- Encapsulation and ADTs, ML-like object type system, …

…

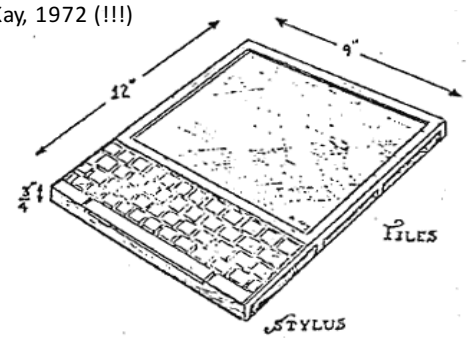
Smalltalk

- Goals:
  - Historical context
  - Core OO ideas from a different perspective
  - Pure OO – influenced many languages
- Non-goal: remember the syntax, program in it

Dynabook

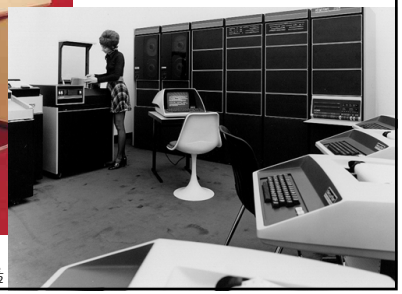
"A Personal Computer for Children of All Ages"

Alan Kay, 1972 (!!!)

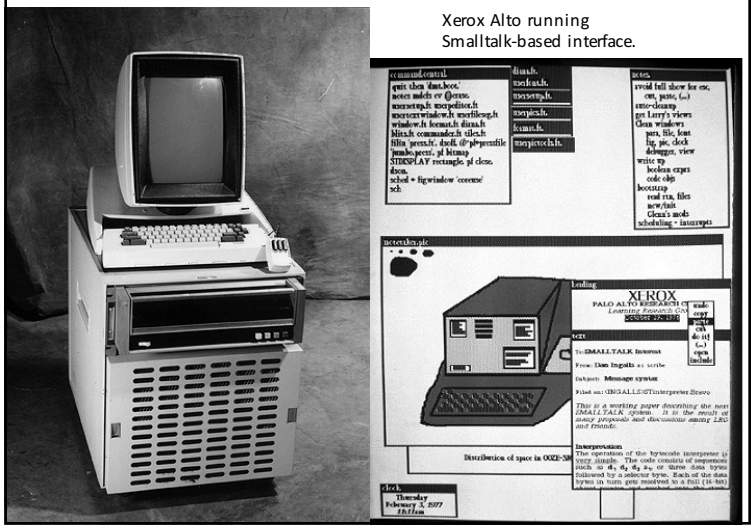


http://simh.trailing-edge.com/

By contrast:
PDP-11 "**minicomputers**"
c.a. early 1970s



http://www.poworldcom/article/249951/if_it_aint_broke_dont_fix_it_ancient_computers_in_use_today.html?page2



Xerox Alto running Smalltalk-based interface.

Smalltalk example: Point class

- Class definition written in tabular form

class name	Point
super class	Object
class vars	pi
instance vars	x y
class messages and methods	
⟨...names and code for methods...⟩	
instance messages and methods	
⟨...names and code for methods...⟩	

Encapsulation: Accessible only from Point methods

constructors

Smalltalk figures / code from Steve Freund

Instance Messages and Methods

Instance methods

```
moveDx: dx Dy: dy | |
    x <- dx + x
    y <- dy + y
```

Usage

```
pt moveDx: 1 Dy: 1
```

In Java:

```
void moveDxDy(int dx,
              int dy) {
     x = x + dx;
     y = y + dy;
}
```

```
pt.moveDxDy(1,1);
```

Instance Messages and Methods

Instance methods

```
moveDx: dx Dy: dy | |
    x <- dx + x
    y <- dy + y

x: xcoord y: ycoord | |
    x <- xcoord
    y <- ycoord
```

Usage

```
pt moveDx: 1 Dy: 1
```

```
pt x:3 y:2
```

```
void xy(int xcoord,
        int ycoord) {
  x = xcoord;
  y = ycoord;
}
```

```
pt.xy(3,2);
```

Instance Messages and Methods

Instance methods

```
moveDx: dx Dy: dy | |
    x <- dx + x
    y <- dy + y

x: xcoord y: ycoord | |
    x <- xcoord
    y <- ycoord

x | | ^x
y | | ^y
draw | |
    ⟨...draw point...⟩
```

Read instance variable

Examples

```
pt moveDx: 1 Dy: 1

pt x:3 y:2

z <- pt x + pt y
```

Class Messages and Methods

Class methods

```
newX: xval Y: yval  | |
  ^ self new x: xval y: yval
```

Examples

```
p <- Point newX:3 Y:2
```

(`new` is method inherited from `Object`)

```
class Point {
   static Point newXY(int xval,
                      int yval) {
     Point temp = new Point();
     temp.xy(xval, yval);
     return temp;
   }
}
```

```
p = Point.newXY(3,2);
```

Class Messages and Methods

Class methods

```
newX: xval Y: yval  | |
  ^ self new x: xval y: yval

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

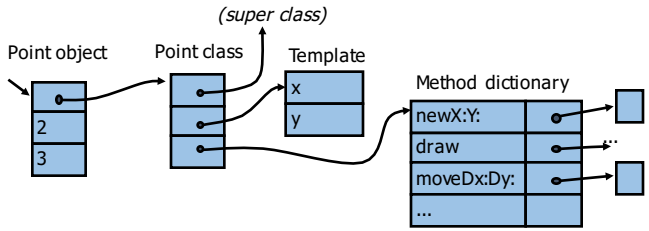
Examples

```
p <- Point newX:3 Y:2

p <- Point newOrigin
```

```
class Point {
   static Point newOrigin() {
     Point temp = new Point();
     temp.xy(0, 0);
     return temp;
   }
}
```

```
p = Point.newOrigin();
```

Class Metadata and Object Representation

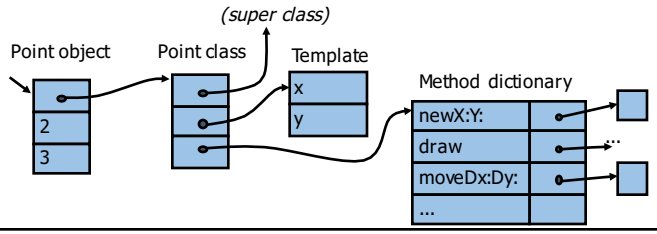


Three primary operations

- object creation
- method lookup (dispatch)
- field lookup

Dynamic dispatch (preview)

- ML, Racket: lexical scope!
  - `(moveDxDy pt 1 1)`
    What function is called? Does ***not*** depend on pt, 1, or 1.
- Smalltalk: "send message" to object
  - `pt moveDx: 1 Dy: 1`
    Method called depends on `pt`.



Inheritance

Implementation technique:
Reuse representation and behavior of Point
to build a related (and more specific) ColorPoint.

class name	ColorPoint	
super class	Point	
class var		
instance var	color	new instance variable
class messages and methods		
newX:xv Y:yv C:cv	⟨ ... code ... ⟩	new method
instance messages and methods		
color	| | ^color	override Point method
draw	⟨ ... code ... ⟩	

ColorPoint Methods

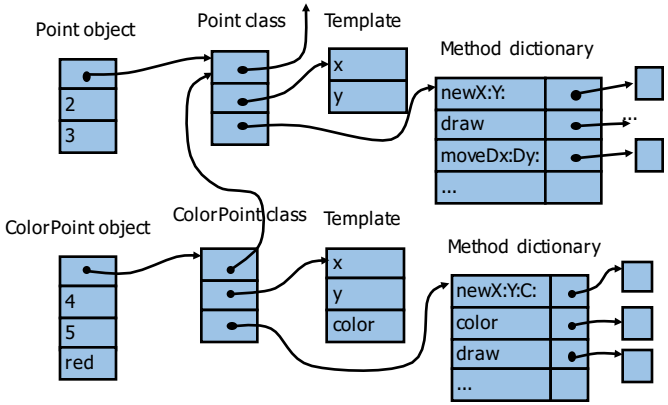
Instance Methods

```
x: xcoord y: ycoord c: col | |
    x <- xcoord
    y <- ycoord
    color <- c

color | | ^color
draw | | ...
```

Class Methods

```
newX: xv Y: yv C: cv ||
   ^self new x:xv y:yv c:cv

newOrigin ||
   ^self newX:0 Y:0 C:red
```

Point / ColorPoint representation



Substitutivity

```
pt <- ???
pt moveDx: 1 Dy: 1
pt draw
```

Smalltalk

Point?
ColorPoint?
Something else?
Does it matter?

```
pt = ???;
pt.moveDxDy(1,1);
pt.draw();
```

Java

Subtyping and Substitutivity

```
class Rectangle {
  private int x,y,w,h;
  void moveTo(int x, int y);
  void setSize(int width, int height);
  void show();
  void hide();
}

class FilledRectangle {
  private int x,y,w,h;
  private Color c;
  void moveTo(int x, int y);
  void setSize(int width, int height);
  void show();
  void hide();
  void setFillColor(Color color);
  Color getFillColor();
}
```

Subtyping and Substitutivity

```
void f() {
  Rectangle r =
    new Rectangle();
  r.moveTo(100,100);
  r.hide();
}
```

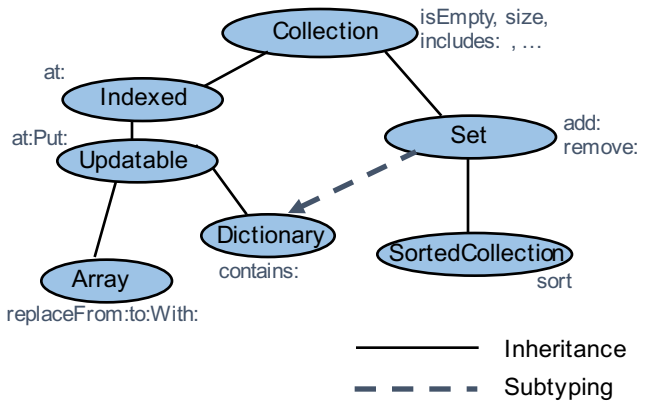
```
void f() {
  Rectangle r =
    new FilledRectangle();
  r.moveTo(100,100);
  r.hide();
}
```

?

```
void g() {
  FilledRectangle r =
    new FilledRectangle();
  r.moveTo(100,100);
  r.setFillColor(Color.red);
  r.hide();
}
```

```
void g() {
  FilledRectangle r =
    new Rectangle();
  r.moveTo(100,100);
  r.setFillColor(Color.red);
  r.hide();
}
```

Collection Hierarchy



Key things to revisit more precisely

- Encapsulation
- Dynamic dispatch (method lookup)
- Inheritance
- Subtyping and substitutivity
- How all of these interact