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 (!!!)
Smalltalk example: Point class

- Class definition written in tabular form

<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 vars</td>
<td>pi</td>
</tr>
<tr>
<td>instance vars</td>
<td>x y</td>
</tr>
<tr>
<td>class messages and methods</td>
<td>(...names and code for methods...)</td>
</tr>
<tr>
<td>instance messages and methods</td>
<td>(...names and code for methods...)</td>
</tr>
</tbody>
</table>

Encapsulation:
Accessible only from Point methods

Constructors

Instance Messages and Methods

Instance methods

| moveDx: dx Dy: dy | pt moveDx: 1 Dy: 1 |
| moveDx: dx Dy: dy | pt moveDxDy(1,1); |
| x <- dx + x | x = x + dx; |
| y <- dy + y | y = y + dy; |

Usage

pt moveDxDy(1,1);

In Java:

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

Usage

pt moveDxDy(1,1);

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

Usage

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
  
  z <- pt x + pt y

**Examples**

- pt moveDx: 1 Dy: 1

**Class Messages and Methods**

**Class methods**

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

- newOrigin |

  ^ self new x: 0 y: 0

**Examples**

- pt: 3 y: 2

**Class Metadata and Object Representation**

**Two 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`.

What function is called? Does not depend on pt, 1, or 1.

1. Smalltalk: "send message" to object
   - `pt moveDx: 1 Dy: 1`
     Method called depends on `pt`.

2. ColorPoint Methods
   **Instance Methods**
   - `x: xcoord y: ycoord c: col |
     x <- xcoord
     y <- ycoord
     color <- c`
   - `color | | ^color`
   - `draw | | ...`

   **Class Methods**
   - `newX: x Y: y C: cv | |`
     `^self new x:xv y:yv C:cv`
   - `newOrigin | |`
     `^self newX:0 Y:0 C:red`

Inheritance

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

<table>
<thead>
<tr>
<th>class name</th>
<th>ColorPoint</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:xv Y:yv C:cv</td>
<td>( ... code ... )</td>
</tr>
<tr>
<td>instance messages and methods</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td></td>
</tr>
<tr>
<td>draw</td>
<td></td>
</tr>
</tbody>
</table>

new instance variable
new method
override Point method

Point / ColorPoint representation
Substitutivity

\[
\begin{align*}
\text{pt} & \leftarrow ??? \\
\text{pt moveDx: 1 Dy: 1} & \quad \text{Smalltalk} \\
\text{pt} & \ 	ext{draw} \\
\text{pt = ???; } & \\
\text{pt.moveDxDy(1,1); } & \quad \text{Java} \\
\text{pt.draw();}
\end{align*}
\]

Subtyping and Substitutivity

\[
\begin{align*}
\text{void f() { } } & \\
\text{Rectangle r = } & \quad \text{New Rectangle();} \\
\text{r.moveTo(100,100); } & \\
\text{r.hide(); } & \\
\text{}} & \\
\text{void g() { } } & \\
\text{FilledRectangle r = } & \quad \text{New FilledRectangle();} \\
\text{r.moveTo(100,100); } & \\
\text{r.setFillColor(Color.red); } & \\
\text{r.hide(); } & \\
\text{}}
\end{align*}
\]

Subtyping and Substitutivity

\[
\begin{align*}
\text{class Rectangle { } } & \\
\text{private int x,y,w,h; } & \\
\text{void moveTo(int x, int y); } & \\
\text{void setSize(int width, int height); } & \\
\text{void show(); } & \\
\text{void hide(); } & \\
\text{}} & \\
\text{class FilledRectangle { } } & \\
\text{private int x,y,w,h; } & \\
\text{private Color c; } & \\
\text{void moveTo(int x, int y); } & \\
\text{void setSize(int width, int height); } & \\
\text{void show(); } & \\
\text{void hide(); } & \\
\text{void setFillColor(Color color);} & \\
\text{Color getFillColor(); } & \\
\text{}}
\end{align*}
\]

Collection Hierarchy

\[
\begin{align*}
\text{isEmpty, size, } & \\
\text{includes: , ...} & \\
\text{add: remove: } & \\
\text{at: \quad at:Put: } & \\
\text{Indexable, includes: } & \\
\text{contains: } & \\
\text{replaceFrom:to:With: } & \\
\text{Inheritance } & \\
\text{Subtyping}
\end{align*}
\]

11/10/15
Key things to revisit more precisely

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