OOP terminology

- **Object**:
  - Dynamic entity that receives **messages**
  - Encapsulates **instance variables** (generally imperative) and **methods** (aka **instance methods**)
  - Special variable `self` refers to the host object
- **Class**: Construct for defining objects
  - Specifies methods for objects
  - Defines object initialization in **class methods**
  - New objects created by sending message to class
  - May **inherit** behavior from **parent** or **super** class
  - An object **instantiated** from a class is called an **instance** of that class.
- **Dynamic dispatch**: Another term for message sending

Example: List select (aka filter)

```
-> (val ns (List withAll: '(1 2 3 4 5)))
List( 1 2 3 4 5 )
-> (ns select: [block (n) ((n mod: 2) = 0)])
List( 2 4 )
```

Vocabulary notes:

- `List` is a class
- `withAll` is a class method
  - It encapsulates a use of `new`
- `ns` is an object and an instance of class `List`
- `ns` is sent the message `select`
- `select` is an instance method of class `List`
- Bonus: `block` construct analogous to `lambda`

Design process in an OO setting

No interrogation about form!

Design process still works

1. **Each method defined on a class**
   - **The class knows the form!**
2. **Class determines**
   - **How object is formed (*class method*)**
   - **From what parts (*instance variables*)**
   - **How object responds to messages (*instance method*)**

Each form of data gets its own methods in its own class!

Using classes to implement select

Class determines how object responds.

Key classes in lists:

- **class `ListSentinel`: an instance denotes end of list**
- **class `Cons`: instance is cons cell with `car` and `cdr` instance variables**

```
(method select: (_) self) ;; on ListSentinel

(method select: (aBlock)  ;; on Cons
   ([aBlock value: car] ifFalse:ifTrue:
       {(cdr select: aBlock)}
       {(((Cons new) car: car)
                     cdr: (cdr select: aBlock))}))
```

Dynamic dispatch replaces if

In OOP, conditionals are deprecated

Instead, idiomatic code uses dynamic dispatch.

We'll expore via example: List filtering via iteration

- Uses imperative idioms (common in OOP)

Iteration in Scheme

Ask value about form:

```
(define app (f xs)
  (if (null? xs)
      'do-nothing
      (begin
         (f (car xs))
         (app f (cdr xs)))))
```

Iteration in uSmalltalk

Use dynamic dispatch:

Instead of (app f xs), we have

```
(xs do: f-block)
```

For each element `x` in `xs`,

send `(f-block value: x)`

Example: Iteration

```
-> (val ms (ns select: [block (n) ((n mod: 2) = 0)]))
List( 2 4 )

-> (ms do: [block (m) ('element print) (space print)
                      ('is print) (space print)
                      (m println)])
element is 2
element is 4
nil
```

Implementing iteration

What happens if we send “do f” to an empty list?

What happens if we send “do f” to a cons cell?

Iteration by dynamic dispatch

Sending do: to the empty list:

```
(method do: (aBlock) nil)
    ; nil is a global object
```

Sending do: to a cons cell:

```
(method do: (aBlock)
    (aBlock value: car)
    (cdr do: aBlock))
```

Look! No if! Decisions made by dynamic dispatch

The six questions

1. Values are objects (even `true`, 3, `"hello"`)
   Even *classes* are objects!
   There are no functions—only methods on objects

The six questions

2. **Syntax:**
   - **Mutable variables**
   - **Message send**
   - **Sequential composition of mutations and message sends (side effects used commonly)**
   - **"Blocks" (objects and closures in one, used as continuations)**
   - **No `if` or `while`. Functionality implemented by passing continuations to Boolean objects.**

Syntax comparison: Impcore

```
Exp = LITERAL of value
    | VAR     of name
    | SET     of name * exp
    | IF      of exp * exp * exp
    | WHILE   of exp * exp
    | BEGIN   of exp list
    | APPLY   of name * exp list
```

Syntax comparison: Smalltalk

```
Exp = LITERAL of rep
    | VAR     of name
    | SET     of name * exp
    | IF      of exp * exp * exp
    | WHILE   of exp * exp
    | BEGIN   of exp list
    | APPLY   of name * exp list
    | SEND    of exp * name * exp list
    | BLOCK   of name list * exp list
```

Syntax comparison: Smalltalk

```
Exp = LITERAL of rep
    | VAR     of name
    | SET     of name * exp
    | IF      of exp * exp * exp
    | WHILE   of exp * exp
    | BEGIN   of exp list
    | APPLY   of name * exp list
    | SEND    of exp * name * exp list
    | BLOCK   of name list * exp list
```

Syntax comparison: Smalltalk

```
Exp = LITERAL of rep
    | VAR     of name
    | SET     of name * exp
    | IF      of exp * exp * exp
    | WHILE   of exp * exp
    | BEGIN   of exp list
    | APPLY   of name * exp list
    | SEND    of exp * name * exp list
    | BLOCK   of name list * exp list
```

The six questions

3. Environments

- Name stands for a mutable cell containing an object:
  - Global variables
  - Instance variables

The six questions

4. Types

   There is no compile-time type system.

   At run time, Smalltalk uses behavioral subtyping, known to Rubyists as “duck typing”
5. Dynamic semantics
   - Main rule is method dispatch (complicated)
   - The rest is familiar
6. The initial basis is *enormous*
   - Why? To demonstrate the benefits of reuse, you need something big enough to reuse.