Recall Bird/Owl

```
public class Bird
{
   public String tweet() { return "tweet"; }
   public String peck(Bird b)
      { return "A bird pecks a bird"; }
   public String peck(Owl o)
      { return "A bird pecks an owl"; }
}

public class Owl extends Bird
{
   public String tweet() { return "hoot"; }
   public String bob() { return "bob bob"; }
   public String peck(Bird b)
      { return "An owl pecks a bird"; }
   public String peck(Owl o)
      { return "An owl pecks an owl"; }
}
```


Recall Bird/Owl (cont)

Recall dynamic dispatch:

```
Bird owl1 = new Owl();
Bird owl2 = new Owl();

owl1.tweet();

owl1.peck(owl2);
```

This produces non-symmetric results.

`Object.equals(Object)`

```
public class Point
{
   int x; int y;
   public boolean equals(Object o)
   {
      if (o instanceof Point)
         return false;
      Point p = (Point)o;
      return (p.x == x && p.y == y);
   }
}

public class ColorPoint extends Point
{
   Color c;
   public boolean equals(Object o)
   {
      if (o instanceof ColorPoint)
         return false;
      ColorPoint p = (ColorPoint)o;
      return (super.equals(o) && c.equals(p.c));
   }
}
```


`Object.equals(Object)` (cont)

What are the results of:

```
Point      p1 = new Point(5,10);
Point      p2 = new Point(5,10);
Point     cp1 = new ColorPoint(5,10,Color.red);
Point     cp2 = new ColorPoint(5,10,Color.red);
ColorPoint c1 = new ColorPoint(5,10,Color.red);
ColorPoint c2 = new ColorPoint(5,10,Color.red);

p1.equals(p2);
p1.equals(cp2);
p1.equals(c2);

cp1.equals(p2);
cp1.equals(cp2);
cp1.equals(c2);
```

But `equals()` should be symmetric!!

Multimethods

What we need is *multimethods*!

Change notation of `Object.equals(Object)` to `equals(Object, Object)`.

`Dynamically dispatch` on the runtime types of both `Object`s.

(*non-java code*)

```
boolean equals(Point p, Point q)
{
   return (p.x == q.x && p.y == q.y);
}

boolean equals(ColorPoint p, ColorPoint q)
{
   return (resend && equals(p.color,q.color));
}
```


Multimethods (cont)

Method is chosen at runtime that best matches runtime type of arguments:

```
Point p1 = new Point(5,10);
Point p2 = new Point(5,10);
Point c1 = new ColorPoint(5,10,Color.black);
Point c2 = new ColorPoint(5,10,Color.black);

equals(p1,p2);
equals(p1,c2);
equals(c1,p2);
equals(c1,c2);
```


Multimethod Summary

Good for: arithmetic operations, comparisons, etc.

Examples: `display(Shape, OutputDevice)`

Multimethods dispatch on the runtime types of all arguments.

Methods called is most specific one that matches arguments.

Cecil supports multimethods.

Simulating Double-Dispatch

(*this is Java code*)

```
public class Point
{
   public boolean equals(Point p)
      { return p.equalsPoint(this); }
   public boolean equalsPoint(Point p)
      { return(p.x == x && p.y == y); }
   public boolean equalsColorPoint(ColorPoint p)
      { return equalsPoint(p); }
}

public class ColorPoint
{
   public boolean equals(Point p)
      { return p.equalsColorPoint(this); }
   public boolean equalsColorPoint(ColorPoint p)
      { return (equalPoint(p) && p.c.equals(c)); }
}
```

Note: typically we would define `boolean equals(Object)`.

Double-Dispatch (cont)

What are the results of?

```
Point      p1 = new Point(5,10);
Point      p2 = new Point(5,10);
Point     cp1 = new ColorPoint(5,10,Color.red);
Point     cp2 = new ColorPoint(5,10,Color.red);
ColorPoint c1 = new ColorPoint(5,10,Color.red);
ColorPoint c2 = new ColorPoint(5,10,Color.red);

p1.equals(p2);
p1.equals(cp2);
p1.equals(c2);

cp1.equals(p2);
cp1.equals(cp2);
cp1.equals(c2);
```


Smalltalk

Object-oriented.

All objects in heap.

Rooted class hierarchy.

Only dynamically typed (unlike Java).

Everything is an object: `5`, `nil`, `True`.

First-class closures – like `lambda` in Scheme.

Everything is a message send (method invocation) even control blocks.

Programmer can change any class in hierarchy!

Looks a bit like English.

Message Sends

Unary Messages:

```
Date today
Time now hours
Array new
```

Keyword Messages:

```
array new: 10
someArray at: 5 put: 10
```


Understanding Messages

Consider: `Time now hours + 1`

- Send `now` message to `Time` to get current time.
- Send *current time object* message `hours` to get the current hour (an integer).
- Send the message `+` with argument `1` to this integer.

Blocks

Blocks of code in Smalltalk are *first class*!!
(like `lambda` in Scheme)

```
| c1 c2  x |              "declare variables"
x  := 0.
c1 := [ x := x+1 ].       "c1 is a block"
c2 := [ :i | x := x+i ].  "c2 is a block w/ one param"

c1 value.                 "evaluate c1"
c2 value: 20.             "evaluate c2, with the argument 20"

x.                        "now x equals 21... "
```


Compare to Scheme

```
(let ((c1 0) (c2 0) (x 0))
     (set! x 0)
     (set! c1 (lambda ()  (set! x (+ x 1))))
     (set! c2 (lambda (i) (set! x (+ x i))))
     (c1)
     (c2 10)
     x
)
```


Booleans

Class `Boolean` has two subclasses `True` and `False`

```
True

   printOn: stream
      stream nextPutAll: 'true'

   & b   "evaluating and"
      ^ b

   | b   "evaluating or"
      ^ true

   not
      ^ false

   and: block   "short-circuit and"
      ^ block value

   or: block   "short-circuit or"
      ^ true
```


False

```
False

   printOn: stream
      stream nextPutAll: false'

   & b   "evaluating and"
      ^ false

   | b   "evaluating or"
      ^ b

   not
      ^ true

   and: block   "short-circuit and"
      ^ false

   or: block   "short-circuit or"
      ^ block value
```

Using Booleans

Consider the following:

```
(3=4) & (2>1)

(3=4) | (2>1)

(3=2) not

true & false not

(3=4) and: [(1/0) = 8]
```


Conditionals

```
True

   ifTrue: block
      ^ block value

   ifFalse: block
      ^ nil

   ifTrue: tBlock ifFalse: fBlock
      ^ tBlock value

False

   ifTrue: block
      ^ nil

   ifFalse: block
      ^ block value

   ifTrue: tBlock ifFalse: fBlock
      ^ fBlock value
```


Using Conditionals

Consider the following:

```
3=4 ifTrue: [x := 10].

x=y ifTrue: [x := 8] ifFalse: [x := 9].

x := x=y ifTrue: [8] ifFalse: [9].
```


Iteration

Blocks and boolean classes used to implement iteration also:

```
a := 1.
[a < 10] whileTrue: [Transcript show: a printString.  a := a+1].

a := 1.
[a > 10] whileFalse: [Transcript show: a printString.  a := a+1].

1 to: 10 do: [:x | Transcript show: x printString].
```

Defining classes

Defining a class is just sending a message to its parent class object:

```
Object subclass: #Stack
   instanceVariableNames: 'anArray top'
   classVariableNames: ''
   poolDictionaries: ''
```


Stack Methods

```
Stack

   push: item
      top := top+1.
      anArray at: top put: item


   pop | item |
      item := anArray at: top.
      top := top-1.
      ^ item

   setsize: n
      anArray := Array new: n.
      top := 0.

   isEmpty
      ^ top=0
```


Using the Stack

Consider:

```
S := Stack new.
S setsize: 10.
S inspect.
S push: 'hi there'.
S push: 3.14159.
S pop.
```


Summary of Smalltalk

All objects in heap.

Dynamically typed.

Everything is an object: `5`, `nil`, `True`.

Blocks are first-class closures – like `lambda` in Scheme.

Everything is a message send:
- Control blocks: conditionals, loops.
- Class instantiation.
- Subclassing.

Programmer can change any class in hierarchy!