SUBCLASING

Example (to be continued)

```ruby
class Point
  attr_accessor :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
  def distFromOrigin
    Math.sqrt(@x*@x + @y*@y)
  end
  def distFromOrigin2
    Math.sqrt(x*x + y*y)
  end
end

class ColorPoint < Point
  attr_accessor :color
  def initialize(x, y, c)
    super(x, y)
    @color = c
  end
end
```

An object has a class

```ruby
p = Point.new(0,0)
cp = ColorPoint.new(0,0,"red")
p.class # Point
p.class.superclass # Object
cp.class # ColorPoint
cp.class.superclass # Object
cp.is_a? Point # true
cp.instance_of? Point # false
cp.is_a? ColorPoint # true
cp.instance_of? ColorPoint # true
```

- Java note: `instanceof` is like Ruby’s `is_a?`
- Semantics: an instance of `ColorPoint` is a `Point` but not an `instanceof` `Point`
- Using these methods is usually non-OOP style

Review

- Ruby: classes, methods, variables, arrays, hashes, blocks, procs
- Today:
  - Subclassing
  - Duck typing
  - Dynamic dispatch
Example continued

- Consider alternatives to subclassing:

```ruby
class ColorPoint < Point
  attr_accessor :color
  def initialize(x, y, c="black")
    super(x, y, c)
    @color = c
  end
end
```

- Could instead:
  - modify Point to include colors
  - create ColorPoint by copy-pasting code into it
  - make ColorPoint have a Point as an instance variable

Modify Point

- Instead of creating ColorPoint, could add methods to Point
  - That could mess up other users and subclassers of Point

```ruby
class Point
  attr_accessor :color
  def initialize(x, y, c="black")
    @x = x
    @y = y
    @color = c
  end
end
```

Copy-Paste

- Instead of subclassing Point, could copy/paste the methods
  - Means the same thing if you don't use methods like is_a? and superclass, but of course code reuse is nice

```ruby
class ColorPoint
  attr_accessor :x, :y, :color
  def initialize(x, y, c="black")
    @pt = Point.new(x, y)
    @color = c
  end
  def x
    @pt.x
  end
  # similar "forwarding" methods
  # for y, x=, y=
end
```

Instance variable

- Instead of subclassing Point, could use a Point instance variable
  - Define methods to send same message to the Point
  - But for ColorPoint, subclassing makes sense: less work and can use a ColorPoint wherever code expects a Point whenever code expects a Point

```ruby
class ColorPoint
  attr_accessor :color
  def initialize(x, y, c="black")
    @pt = Point.new(x, y)
    @color = c
  end
  def x
    @pt.x
  end
  .. # similar "forwarding" methods
  # for y, x=, y=
end
```

Overriding

- ThreeDPoint is more interesting than ColorPoint because it overrides distFromOrigin and distFromOrigin2
  - Gets code reuse, but highly disputable if it is appropriate to say a ThreeDPoint "is a" Point
  - Still just avoiding copy/paste

```ruby
class ThreeDPoint < Point
  def initialize(x, y, z)
    @x = x
  end
  def distFromOrigin # distFromOrigin2 similar
    d = super
    Math.sqrt(d*d + @z*@z)
  end
end
```

So far...

- With examples so far, objects are not so different from closures
  - Multiple methods rather than just 'call me'
  - Explicit instance variables rather than environment where function is defined
  - Inheritance avoids helper functions or code copying
  - 'Simple' overriding just replaces methods

- But there is one big difference:

  Overriding can make a method defined in the superclass call a method in the subclass
Example: Equivalent except constructor

```ruby
class PolarPoint < Point
  def initialize(r, theta)
    @r = r
    @theta = theta
  end
  def x
    @r * Math.cos(@theta)
  end
  def y
    @r * Math.sin(@theta)
  end
  def distFromOrigin
    @r
  end
end
```

- Also need to define `x=` and `y=` (see code file)
- Key punchline: `distFromOrigin2`, defined in `Point`, “already works”

```
def distFromOrigin2
  Math.sqrt(x**2+y**2)
end
```

- Why: calls to `self` are resolved in terms of the object’s class

DUCK TYPING

Subtyping

- Now that we have subclassing, you might wonder about subtyping
- In Java, if `ColorPoint` is subclass of `Point`, then anywhere a `Point` is expected, a `ColorPoint` could be used instead
- But Ruby doesn’t have (static) types, so what does it use instead?

Duck Typing

“If it walks like a duck and quacks like a duck, it’s a duck”

- Or don’t worry that it may not be a duck

When writing a method you might think, “I need a `Foo` argument” but really you need an object with enough methods similar to `Foo`’s methods that your method works

- Embracing duck typing is always making method calls rather than assuming/testing the class of arguments

Pro: More code reuse; very OO approach

- What messages an object receive is all that matters

Con: Almost nothing is equivalent

- `x+x` versus `x**2` versus `2*x`
- Callers might assume a lot about how callees are implemented

Duck Typing Example

```ruby
def mirror_update pt
  pt.x = pt.x * (-1)
end
```

- Natural thought: “Takes a `Point` object (definition not shown here), negates the `x` value”
  - Makes sense, though a `Point` instance method more OO
- Closer: “Takes anything with getter and setter methods for `@x` instance variable and multiplies the `x` field by `-1`”
- Closer: “Takes anything with methods `x=` and `x` and calls `x=` with the result of multiplying result of `x` and `-1`
- Duck typing: “Takes anything with method `x=` and `x` where result of `x` has a `*` method that can take `-1`. Sends result of calling `x` the `*` message with `-1` and sends that result to `x=`”

DYNAMIC DISPATCH
Overriding: Example 1

- `ThreeDPoint` is more interesting than `ColorPoint` because it overrides `distFromOrigin` and `distFromOrigin2`
  - Gets code reuse, but highly disputable if it is appropriate to say a `ThreeDPoint` "is a" `Point`

```ruby
class ThreeDPoint < Point
  def initialize(x, y, z)
    @x = x
  end
  def distFromOrigin # distFromOrigin2 similar
    Math.sqrt(d*d + @z*@z)
  end
end
```

Overriding: Example 2

- `PolarPoint` also needs to define `x=` and `y=` (see code file)
  - Key punchline: `distFromOrigin2`, defined in `Point`, "already works"
  - Why: calls to `self` are resolved in terms of the object's class

```ruby
class PolarPoint < Point
  def initialize(r, theta)
    @r = r
    @theta = theta
  end
  def x
    @r * Math.cos(@theta)
  end
  def y
    @r * Math.sin(@theta)
  end
  def distFromOrigin
    @r
  end
end
```

Overriding

- What are the semantics of overriding?
- How does language decide which version of method to call?
- Can’t be based on static type of an object
  - Might think that you have a Point when really you have a ColorPoint
  - ... anyway, no such thing as static types in Ruby

Dynamic dispatch

- Also known as late binding or virtual methods
- Call `self.m2()` in method `m1` defined in class `C` can resolve to a method `m2` defined in a subclass of `C`
- So the code invoked by method call cannot be determined statically (at compile time); must be determined dynamically (at run time)
- ... a key idea of OO language design

Need to define the semantics of `method lookup` as carefully as we defined `variable lookup` for our PLs

Review: variable lookup

Rules for "looking things up" is a key part of PL semantics
- ML: Look up variables in the appropriate environment
  - Lexical scope for closures
  - Field names (for records) are different: not variables
  - Look up "inside" record value
- Ruby:
  - Local variables mostly like ML
  - Instance variables, class variables, methods: not like local variables
  - More like record fields: look up "inside" an object
  - Specifically: look up "inside" `self`

self

- `self` maps to the "current" object
- Look up instance variable `@x` using object bound to `self`
- Look up class variables `@@x` using object bound to `self.class`
- How to look up methods...?
Ruby method lookup

The semantics for method calls, aka message sends:
\[ e_0 \cdot m(e_1, \ldots, e_n) \]

1. Evaluate \( e_0, e_1, \ldots, e_n \) to objects \( \text{obj}_0, \text{obj}_1, \ldots, \text{obj}_n \)
   - As usual, may involve looking up \text{self}, variables, fields, etc.
2. Let \( C \) be the class of \( \text{obj}_0 \) (every object has a class)
3. If \( m \) is defined in \( C \), pick that method, else recur with the superclass of \( C \) unless \( C \) is already \text{Object}
   - If no \( m \) is found, call \text{method_missing} instead
     - Definition of \text{method_missing} in \text{Object} raises an error
4. Evaluate body of method picked:
   - With formal arguments bound to \( \text{obj}_1, \ldots, \text{obj}_n \)
   - With \text{self} bound to \( \text{obj}_0 \) -- this implements dynamic dispatch!

Note: Step (3) complicated by mixins: will revise definition in later lecture

Mini-review: dynamic dispatch

To implement dynamic dispatch, evaluate the method body with \text{self} mapping to the receiver (result of \( e_0 \))

- That way, any \text{self} calls in body of \( m \) use the receiver's class
  - Not necessarily the class that defined \( m \)
  - This is why \text{distFromOrigin2} worked in \text{PolarPoint}

- Evaluation rules in Ruby, Java, C#, Smalltalk, etc. much the same
  - with one complication...

Static overloading

In Java/C#/C++, several methods in a class can have same name
- Java/C/C++: Overriding only when number/types of arguments the same
- Ruby: same-method-name always overriding

Forces Java/C#/C++ to pick the “best method” using the (static) types of the arguments
- Complicated rules for “best”
- Type-checking error if there is no “best”

Ruby has no (static) type-checking rules, and no static overloading

Tradeoffs of dynamic dispatch

In ML (and other languages), closures “lock in” a value:

So we can shadow \text{odd}, but any call to the closure bound to \text{odd} above will “do what we expect”
- Does not matter if we shadow \text{even} or not

\[
\begin{align*}
\text{fun even } x &= \text{if } x=0 \text{ then true else odd (x-1)} \\
\text{and odd } x &= \text{if } x=0 \text{ then false else even (x-1)}
\end{align*}
\]

("does not change odd – too bad; this would improve it")

\[
\text{fun even } x = (x \mod 2)=0
\]

("does not change odd – good thing; this would break it")

\[
\text{fun even } x = \text{false}
\]

Tradeoffs of dynamic dispatch

Any method that makes calls to overridable methods can have its behavior changed in subclasses even if it is not overridden
- Maybe on purpose, maybe by mistake
- Observable behavior includes calls-to-overridable methods

- \text{Pro} easier for subclasses to affect behavior without copying code
  - Assuming method in superclass is not modified later

- \text{Con} harder to reason about “the code you’re looking at”
  - Can avoid by disallowing overriding
    - “private” or “final” methods