# CSE 413
# Programming Languages & Implementation

Hal Perkins

Autumn 2012

Late binding and dynamic dispatch

(Based on CSE 341 slides by Dan Grossman)

# Today

- Dynamic dispatch, aka late binding, aka virtual method calls
  - Call to **`self.m2()`** in method **`m1`** defined in class **`C`** can *resolve* to method **`m2`** defined in a subclass of **`C`**
  - Most unique characteristic of OOP
- Define semantics of objects and method lookup carefully
- Look at advantages and disadvantages of dynamic dispatch
- What if you want dynamic dispatch in a language that doesn't have it built-in?

# Resolving identifiers

- The rules for "looking up" symbols in a programming language is a key part of the language's definition
  - Talk about this in general first, then dynamic dispatch
- Racket: Look up variables in the appropriate environment
  - Key point of closure's lexical scope is defining "appropriate"
  - Also includes let, let*, letrec
- Ruby: local variables and blocks mostly like Racket
  - But also have instance variables, class variables, and methods
  - Java is similar, but no explicit closures

# Ruby instance variables and methods

- **`self`** maps to some "current object"
- Look up variables in environment of method
- Look up instance variables using object bound to **`self`**
- Look up class variables using object bound to **`self.class`**

*Syntactic distinction* between local/instance/class names (**`x, @x, @@x`**) means no ambiguity or shadowing rules

- Contrast to Java where locals shadow fields with same name unless we use **`this.f`**

# Method names are different

- **`self`**, locals, instance variables, class variables all map to objects
- We said "everything is an object" in Ruby but that's not quite true
  - Method names
  - Blocks
  - Argument lists
- *First-class* values are things you can store, pass, return, etc.
  - In Ruby, only objects (almost everything) are first-class
  - Example: cannot do **`e.(if b then m1 else m2 end)`**
    - Have to do **`if b then e.m1 else e.m2 end`**
  - Example: can do **`(if b then x else y).m1`**

# Ruby message lookup

Semantics for method calls aka message sends

`e0.m(e1, …, en)`

1. Evaluate **`e0,e1,…,en`** to objects **`obj0,obj1,…,objn`**
   - Usual rules involving **`self`**, variable lookup, etc.
2. Let **`C`** = class of **`obj0`** (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 **`Object`**
   - If no **`m`** is found, call **`method_missing`** instead
     - Default definition raises an error
   - Mixins complicate this step – more in a moment
4. Evaluate body of method picked in step 3:
   - With parameters bound to arguments **`obj1,…,objn`**
   - With **`self`** bound to **`obj0`** – this implements dynamic dispatch!!

# Java message lookup (very similar)

Semantics for method calls aka message sends

```
e0.m(e1, …, en)
```

1. Evaluate **`e0,e1,…,en`** to objects **`obj0,obj1,…,objn`**
   - Usual rules involving **`this`**, variable lookup, etc.
2. Let **`C`** = class of **`obj0`** (every object has a class)
3. Complicated rules to pick "the best m" using static types of **`e0,e1,…,en`**
   - Static checking ensures suitable **`m`** (in fact the best **`m`**) will always be found
   - Rules similar to Ruby except for this static overloading
   - No mixins to worry about (& interfaces irrelevant here)
4. Evaluate body of method picked in step 3:
   - With parameters bound to arguments **`obj1,…,objn`**
   - With **`this`** bound to **`obj0`** – this implements dynamic dispatch!!

# Ruby mixins

Mixins change the lookup rules slightly

- When looking for receiver **`obj0`**'s method **`m`**, look in **`obj0`**'s class, then mixins that class includes (later includes shadow previous definitions), then **`obj0`**'s superclass, then the superclass's mixins, etc.

# The punch-line again

**`e0.m(e1, …, en)`**

To implement dynamic dispatch, evaluate the method body with **`self`** mapping to the receiver object (**`e0`**)

- That way, any **`self`** calls in the method body use the receiver's (**`e0`**'s) class
  - Not necessarily the class that defined the method being executed
- This is much the same in Ruby, Java, C++, C#, etc.

# Dynamic dispatch vs closures

- Dynamic dispatch is more complicated than the rules for closures
  - Have to treat **`self`** specially
  - May seem simpler only because you learned it first
  - Complicated doesn't imply better or worse
    - Depends on how you use it….
    - Overriding does tend to be overused

# Example (part 1)

In Racket, closures are closed.

```
(define (even x) (if (= 0 x) #t (odd  (- x 1))))
(define (odd x)  (if (= 0 x) #f (even (- x 1))))
```

If we shadow **odd** by redeclaring it in a nested scope, any call to **even** from the original closure will “do what we expect” – good thing too…

```
(letrec ((odd (lambda (x) 17))) (even 42))
```

# Example (part 2)

In Ruby (and other languages) subclasses can change behavior of methods they don't override

```
class A
  def even x
    if x==0 then true  else odd  (x-1) end
  end
  def odd x
    if x==0 then false else even (x-1) end
  end
end
class B < A  # improves odd in B objects
  def even x ; x % 2 == 0 end
end
class C < A  # breaks odd in C objects
  def even x ; false end
end
```

# Feature or bug? The OOP tradeoff

- 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
- Makes it harder to reason about "the code we're looking at"
  - Can avoid by disallowing overriding (Java **`final`**) of methods you call
- Makes it easier for subclasses to specialize behavior without copying code
  - Provided method in superclass isn't modified later

# Manual dynamic dispatch

Rest of lecture: write racket code using (mostly) pairs and functions to act like objects with dynamic dispatch(!)

Why????

- Demonstrates how one language’s semantics is an *idiom* in another language
- Maybe understand dynamic dispatch a bit better by coding it up
  - Much like a compiler/interpreter would do

# The plan

Many possibilities. Code in **`objects.rkt`** does this:

- An "object" has a list of field pairs and a list of method pairs

```
(struct obj (fields methods))
```

- Field-list element example:

```
(mcons 'x 17)
```

- Method-list element example:

```
(cons 'get-x (lambda (self args)… ))
```

Best to study the code, but a few highlights….

# Notes

- Association lists are sufficient for this example but not efficient for production dynamic dispatch.

- Not class-based. Each object has its own list of methods.

- The key "trick" is that every lambda (method) has an extra **`self`** argument
  - All regular "arguments" are in a list **`args`** for simplicity. Use **`car`**, **`cadr`**, … to extract individual arguments

# Key helper functions

Code to get/set fields and send messages (e.g., call functions with self bound properly) are plain old Racket functions:

**`(get obj field)`** – return field value

**`(set obj field val)`** – set field value

**`(send obj msg . args)`**

- send message **`msg`** to **`obj`** with parameters **`args`**
- Need to look up appropriate method in **`obj`** and call it with **`self`** bound to **`obj`**

Look for fields and messages by scan of assoc. list

# Constructing objects

- See function **`make-point`** for example
  - Plain old Racket function that creates an object **`(obj fieldlist methodlist)`**
  - Pair of association lists:

    **`fieldlist`** binds initial argument values

    **`methodlist`** is list of Racket functions
  - Use functions **`get`**, **`set`**, and **`send`** on result and inside "methods"
  - Call to self: **`(send self 'm …)`**

# “Subclassing”

- Can use **`make-point`** to write **`make-color-point`** or **`make-polar-point`** (see code)

- Build a new object using fields and methods from “super” “constructor”
  - Add new or overriding methods to the *beginning* of the list
  - **`send`** will find the first matching method
  - Since **`send`** passes the entire receiver for **`self`**, dynamic dispatch works as desired

# Is this “real”?

OK, Ruby, Java, C++, etc. are not normally implemented this way. Key differences:

- Objects have pointers to “class” objects with a single instance of the method table (vtable)
- Method lookup either uses a hash (Ruby, where methods can be added/deleted during execution) or a static vector (Java/C++/C# where possible methods are known at compile time)

But it does model the semantics correctly and is worth studying, if only for that.