Late Binding;
OOP as a Racket Pattern

Today

Dynamic dispatch aka late binding aka virtual method calls

- Call to `self.m2()` in method `m1` defined in class `C` can *resolve to* a method `m2` defined in a subclass of `C`
- Most unique characteristic of OOP

Need to define the semantics of objects and method lookup as carefully as we defined variable lookup for functional programming

Then consider advantages, disadvantages of dynamic dispatch

Then encoding OOP / dynamic dispatch with pairs and functions

- In Racket
- Complement to learning lambdas/closures in C++

Resolving identifiers

The rules for "looking up" various symbols in a PL is a key part of the language's definition

- So discuss in general before considering dynamic dispatch

- Racket: Look up variables in the appropriate environment
  - Key point of closures' lexical scope is defining "appropriate"

- Java:
  - Lexical scoping like Racket
  - But also have instance variables, class variables, and methods!
    - All of these combined with inheritance "break" lexical scope!

```
class A { int x;  int method() { … }  }
class B extends A { int method() { … }  }
```

in some main method somewhere:

```
B b = new B();
b.x = 17;
  // lookup x … where?
  //   - in current environment -> not found
  //   - in "environment for class B" -> not found
  //   - rule to look in "enclosing environment"
  //      doesn't help here either.
```

```
class A { int x;  int method() { … }  }
class B extends A { int method() { … }  }
```

in some main method somewhere:

```
A waitWhat;
if (Math.random() > .5)
  waitWhat = new A();
else
  waitWhat = new B();
```

`whatWhat.method()`

- Where do we look up `method`?
- Lexical scoping rules don't help us here, because lexical scoping resolves all variable/function references at compile-time. At compile-time, we don't know what's going to happen.

Takeaway:

Looking up the value for a field or the code for a method is different looking up a "regular" variable or "regular" function.

Java method lookup

The semantics for method calls

`e0.m(e1,…,en)`

1. Evaluate **`e0`**, **`e1`**, …, **`en`** to objects **`obj0`**, **`obj1`**, …, **`objn`**
   - As usual, may involve looking up **`this`**, variables, fields, etc.
2. Let **`C`** = the class of **`obj0`** (every object has a class)
3. [Complicated rules to pick "the best **`m`**" using the types of **`e0`**, **`e1`**, …, **`en`**]
   - Rules similar to Ruby except for this *static overloading*
4. Evaluate body of method picked:
   - With formal arguments bound to **`obj1`**, …, **`objn`**
   - With **`this`** bound to **`obj0`** -- this implements dynamic dispatch!

The punch-line again

`e0.m(e1,…,en)`

To implement dynamic dispatch, evaluate the method body with **`this`** mapping to the receiver

- That way, any **`this`** calls in the body use the receiver's class, everything works.

Comments on dynamic dispatch

- C++ only uses dynamic dispatch on virtual functions.
  - That's why in that weird example from a few weeks ago we had the "wrong" function being called.
- Java always does dynamic dispatch.

- More complicated than the rules for closures
  - Have to treat **`this`** specially
  - May seem simpler only because you learned it first
  - Complicated doesn't imply superior or inferior
    - Depends on how you use it…
    - Overriding does tend to be overused

The OOP trade-off

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 you're looking at"
  - Can avoid by disallowing overriding (Java `final`) of helper 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 with little more than pairs and functions that acts like objects with dynamic dispatch

Why do this?

- (Racket actually has classes and objects even though not everything is an object)

- Demonstrates how one language's *semantics* is an idiom in another language
- Understand dynamic dispatch better by coding it up
  - Roughly similar to how an interpreter/compiler would do it