CS 360
Programming Languages
Late Binding

python php Scala Swift

Racket Ruby Java JavaScript Dart

Review

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.

Resolving identifiers

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

  - Let's review this in general before discussing late binding or dynamic dispatch.

- Racket: Look up variables in the appropriate environment.
  - Key point of closures' lexical scope is defining "appropriate."
  - We know lexical scope guarantees we always know what environment a variable lives in ***before*** the program starts running (at compile-time).
  - Aside: why should we care? Why is this important?

Resolving identifiers

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

- Let's review this in general before discussing late binding or dynamic dispatch.

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

- C++:
  - Virtual functions also seemingly 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()`? [where do we find its code?]
- 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.

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.
  - May seem simpler only because you learned it first.
  - Complicated doesn't imply superior or inferior.
    - Depends on how you use it…

The OOP trade-off

Any method that makes calls to overridable methods can have its behavior changed in subclasses.

- 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 lists and functions that acts like objects with dynamic dispatch.

Why do this?

  - (Racket actually has classes and objects even we haven't used them in this course.)

- 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.