Dynamic Dispatch and Inheritance

Variable lookup
Key piece of semantics in any language.

- ML, Racket:
  - Just one kind of variables.
  - Lexical scope – unambiguous binding
  - Field names (in records) are not variables: no "lookup"

- Smalltalk, Java, Scala ...
  - Local variables same
  - More limited scope if no first-class/higher-order functions
  - Instance variables, methods
  - Look up in terms of special self / this "variable"

Method lookup: dynamic dispatch
Two key questions:

- General case:
  What m is run by ____ . m() ?

- Specific case:
  What m is run by this . m() ?

Quick look at classes in Scala
(take notes)

```scala
class Point(val x: Double, val y: Double) {
  def getX(): Double = x
  def getY(): Double = y
  def distFromOrigin: Double = {
    Math.sqrt(getX() * getX() + getY() * getY())
  }
}
```

```scala
class PolarPointA(val r: Double, val theta: Double) extends Point(0.0, 0.0) {
  override defgetX(): Double = r * Math.cos(theta)
  override defgetY(): Double = r * Math.sin(theta)
  override def distFromOrigin: Double = r
}
```
**Method lookup**

```scala
class Point(val x: Double, val y: Double) {
  def getX(): Double = x
  def getY(): Double = y
  def distFromOrigin(): Double = {
    Math.sqrt(this.getX()*this.getX() + this.getY()*this.getY())
  }
}
```

```scala
class PolarPointB(val r: Double, val theta: Double) extends Point(0.0, 0.0) {
  override def getX(): Double = {
    this.r * Math.cos(this.theta)
  }
  override def getY(): Double = {
    this.r * Math.sin(this.theta)
  }
}
```

**Dynamic dispatch** (a.k.a. late binding or virtual methods)

*The unique OO semantics feature.*

**Key questions:**
- Which distToOrigin is called?
- Which x and y getters are called by that distToOrigin?

*this* refers to the **current object**, not the containing class.

- *this.foo()* uses **late binding (dynamic dispatch)** to find foo
- **NOT** lexical scope

**Dynamic Dispatch is not just...**

```
obj0.m(obj1,...,objn)
```

```
m(obj0,obj1,...,objn)
```

Is *this* just an implicit parameter that captures a first argument written in a different spot?

**NO!** "What *m* means" is determined by class of *obj0!"

Must inspect *obj0* before starting to execute *m.*

*this* is different than any other parameters.

**Key artifacts of dynamic dispatch**

- Why overriding works...

```
distFromOrigin in PolarPointA
```

- Subclass's definition of *m* "shadows" superclass's definition of *m* when dispatching on object of subclass (or descendant) even if dispatching from method in superclass.

- More complicated than the rules for closures
  - Have to treat *this* specially
  - May seem simpler only if you learned it first
  - Complicated != inferior or superior
**Closed vs. open**

ML: closures are closed

fun even x = if x=0 then true else odd (x-1)
and odd x = if x=0 then false else even (x-1)

May shadow even, but calls to odd above still “do what we expect”

(* does not change odd: too bad, would help *)
fun even x = (x mod 2)=0

(* does not change odd: good, would break *)
fun even x = false

---

**OOP trade-off: implicit extensibility**

Any method that calls overridable methods (even on this) can have behavior changed by subclass even if it is not overridden.
- On purpose, by mistake?
- Behavior depends on calls to overridable methods

- **Harder** to reason about “the code you’re looking at”
  - Avoid by disallowing overridable: “private” or “final” methods

- **Easier** for subclasses to extend existing behavior without copying code
  - Assuming superclass method is not modified later

---

**Closed vs. open**

Most OOP languages: subclasses can change the behavior of superclass methods they do not override.

class A {
  def even(x: Int): Boolean = {
    if (x == 0) true else odd(x-1)
  }
  def odd(x: Int): Boolean = {
    if (x == 0) false else even(x-1)
  }
}
class B extends A {
  override def even(x: Int): Boolean = x % 2 == 0
}
class C extends A {
  override def even(x: Int): Boolean = false
}

---

**FP trade-off: explicit extensibility**

A function that calls other functions may have its behavior modified only if it calls functions passed as arguments.

- **Easier** to reason about “the code you’re looking at”
  - Calls to argument functions (i.e., sources of unknown behavior) are explicit.

- **Harder** for other code to extend existing behavior without copying code
  - Only by functions as arguments to higher-order functions.
Overloading is static.

More rules:
- **overloading**: > 1 methods in class can have same name
- **overriding**: if and only if same number/types of arguments

Pick the “best one” using the static types of the arguments
- Complicated rules for “best”
- Type-checking error if there is no “best”
- Some confusion when expecting wrong over-thing

---

**super:**

**Static dispatch** *(a.k.a early binding)*

- Calls to `e.m2()` where `e` has declared class `C`
  - *(the lexically enclosing class is this’s “declared class”)*
  - *always resolve* to “closest” method `m2` defined in `C` or `C`’s ancestor classes
  - completely ignores run-time class of object result of `e`
- ... similar to lexical scope for method lookup with inheritance.
- A given method call **always** resolves to same method definition.
  Determined before running program.
- **used for super**