Dynamic Dispatch

semantic essence
of "object-oriented" programming languages
(OOP)
How are names resolved?

Key piece of semantics in any language.

• ML, Racket:
  – Just one kind of variables.
  – Lexical scope – unambiguous binding
  – Record field names are not variables: no "lookup"

• Java, ...:
  – Local variables: lexical scope (more limited)
  – Instance variables, methods
    • Look up in terms of special self / this "variable"
    • it's more complicated...
Method lookup in OO languages

Two key questions for Java:

– General case:
  What m is run by ___ . m() ?

– Specific case:
  What m is run by this . m() ?
class Point {
    double x, y;
    Point(double x, double y) {
        this.x = x; this.y = y;
    }
    double getX() { return this.x; }
    double getY() { return y; }
    double distFromOrigin() {
        return Math.sqrt(this.getX() * this.getX()
                        + getY() * getY());
    }
}

class PolarPoint extends Point { // poor design, useful example
    double r, theta;
    PolarPoint(double r, double theta) {
        super(0.0, 0.0); this.r = r; this.theta = theta;
    }
    double getX() { return this.r * Math.cos(this.theta); }
    double getY() { return r * Math.sin(theta); }
}

Point p = ...; // ???
p.getX(); // ???
p.distFromOrigin(); // ????
Method lookup: example

Key questions:
- Which `distToOrigin` is called?
- Which `getX`, `getY` methods does it call?

```java
Point p = ...; // ???
p.getX(); // ???
p.distFromOrigin(); // ???
```
Dynamic dispatch
(a.k.a. late binding, virtual methods)

The unique OO semantics feature.

Method call: \texttt{e.m()} \texttt{()}

Evaluation rule:
1. Under the current environment, evaluate \texttt{e} to value \texttt{v}.
2. Let \texttt{C} refer to the class of the receiver object \texttt{v}.
3. Until class \texttt{C} contains a method definition \texttt{m()} \{ \texttt{body} \} let \texttt{C} refer to the superclass of the current \texttt{C} and repeat step 3.
4. Under the environment of class \texttt{C}, extended with the binding \texttt{this} \rightarrow \texttt{v}, evaluate the \texttt{body} found in step 3.

Note: \texttt{this} refers to \texttt{current receiver object}, not containing class.
- \texttt{this.m()} uses \texttt{dynamic dispatch} just like other calls.
- \texttt{NOT} lexical scope, not dynamic scope
Dynamic Dispatch is not ...

\[ \text{obj0}\.m(\text{obj1}, \ldots, \text{objn}) \neq m(\text{obj0}, \text{obj1}, \ldots, \text{objn}) \]

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

NO!
"What m means" is determined by run-time class of \text{obj0}!

Must inspect \text{obj0} before starting to execute \text{m}.

\text{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) in all contexts, even if dispatching from method in superclass.

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

ML: closures are, well, closed.

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

May shadow even, but calls to odd are unaffected.

\[
\begin{align*}
\text{(* does not change odd: too bad, would help *)} \\
\text{fun even } x &= (x \text{ mod } 2) = 0
\end{align*}
\]

\[
\begin{align*}
\text{(* does not change odd: good, would break *)} \\
\text{fun even } x &= \text{false}
\end{align*}
\]
Closed vs. open

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

class A {
    boolean even(int x) {
        if (x == 0) return true;
        else return odd(x-1);
    }
    boolean odd(int x) {
        if (x == 0) return false;
        else return even(x-1);
    }
}
class B extends A {  // improves odd in B objects
    boolean even(int x) { return x % 2 == 0; }
}
class C extends A {  // breaks odd in C objects
    boolean even(int x) { return false; }
}
OOP trade-off: implicit extensibility

Any method that calls overridable methods may have its behavior changed by a 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.”
  – Sources of unknown behavior are pervasive:
    all overridable methods transitively called by this method.
  – Avoid by disallowing overriding: “private” or “final”

• **Easier** for subclasses to extend existing behavior without copying code.
  – Assuming superclass method is not modified later
FP trade-off: explicit extensibility

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

• Easier to reason about “the code you're looking at.”
  – Sources of unknown behavior are explicit: calls to argument functions.

• Harder for other code to extend existing behavior without copying code.
  – Only by functions as arguments.
Aside: *overloading* is static.

**overloading:**
> 1 methods in class have same name

**overriding:**
if and only if same number/types of arguments

Pick the "best" overloaded method using the *static* types of the arguments

– Complicated rules for “best”
– Some confusion when expecting wrong over-thing
**static dispatch**  
(a.k.a early binding, non-virtual methods)

- Lookup method based on static type of receiver.
- 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.

- Same method call *always* resolves to same method definition.  
- Determined statically by type system *before* running program.

- *used for super* in Java, non-virtual methods in C++
class Point {
    double x, y;
    Point(double x, double y) {
        this.x = x; this.y = y;
    }
    double getX() { return this.x; } 
    double getY() { return y; }
    double distFromOrigin() {
        return Math.sqrt(this.getX() * this.getX() + this.getY() * this.getY());
    }
}

class PolarPoint extends Point { // poor design, useful example
    double r, theta;
    PolarPoint(double r, double theta) {
        super(0.0, 0.0); this.r = r; this.theta = theta;
    }
    double getX() { return this.r * Math.cos(this.theta); } 
    double getY() { return r * Math.sin(theta); }
}

Point p = ...; // ???
p.getX(); // ???
p.distFromOrigin(); // ???