Dynamic Dispatch

* Aka
  + Late binding
  + Virtual methods
* Call **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 *method lookup* as carefully as we defined variable lookup for our PLs

Review: variable lookup

Rules for “looking things up” is a key part of PL semantics

* ML: lookup variables in the appropriate environment
  + Lexical scope for closures
  + Field names (for records) are different: not variables
* Racket: Like ML plus let, letrec
* Ruby:
  + Local variables and blocks mostly like ML and Racket
  + But also have instance variables, class variables, methods (all more like record fields)
    - Look up in terms of **self**, which is special
      * Refers to the current object

Using **self**

* **self** maps to some “current” object
* Look up instance variable @x using object bound to **self**
* Look up class variables @@x using object bound to **self.class**
* Look up methods…
  + This is where dynamic dispatch takes place

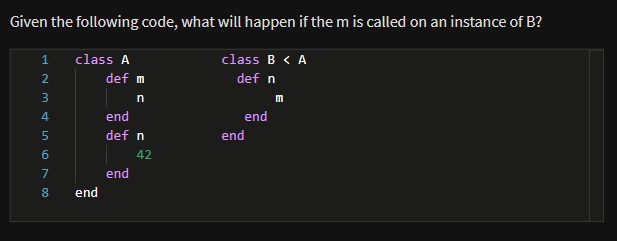
Ruby method lookup

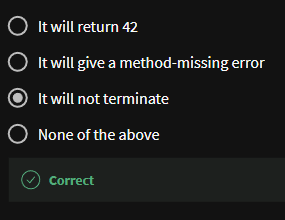
The semantics for method calls also known as message sends

*e0.m(e1, … , en)*

1. Evaluate e0, e1, …, en to objects obj0, obj1, …, objn
   * As usual, may involve looking up **self**, variables, fields, etc.
2. Let C be the 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
     + Definition of method\_missing in Object raises an error
   * Pataas ang evaluation (papuntang superclass)
4. Evaluate body of method picked:
   * With formal arguments bound to obj1, …, objn
   * *With self bound to obj0 -- this implements dynamic dispatch!*

Note: Step (3) complicated by mixins: will revise definition later





Punch-line again

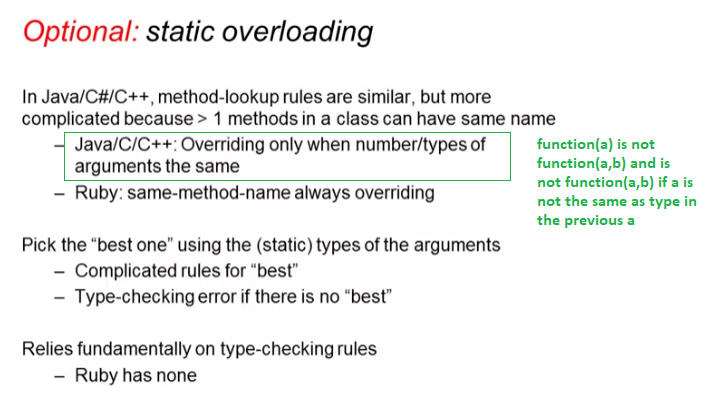
e0.m(e1, … , en)

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

* That way, any self calls in body of m use the receiver’s class,
  + Not necessarily the class that defined m
* This much is the same in Ruby, Java, C#, Smalltalk, etc.

Comments on dynamic dispatch

* This is why distFromOrigin2 worked in PolarPoint
* More complicated than the rules for closures
  + Have to treat self specially
  + May seem simpler only if you learned it first
  + Complicated does not necessarily mean inferior or superior



* We cannot use this in dynamic typing!! Ruby is dynamically typed