CSE 341:
Programming Languages

Section AC with Nate Yazdani

agenda

- method dispatch
- mixins
- visitor pattern

method dispatch

what is dispatch

- *method dispatch* — or just *dispatch* — is the protocol to look up the method body for a method call based on the classes of argument values
- the Ruby and Java languages incorporate *single dispatch* on the *receiver object*, the implicit **`self`** or **`this`** parameter
  - secretly, class constructors stash references to their method implementations inside all objects
  - single dispatch is what you learned in CSE 143

what is dispatch

resolution must (usually) happen at runtime

- *method dispatch* — or just *dispatch* — is the protocol to look up the method body for a method call based on the classes of argument values
- the Ruby and Java languages incorporate *single dispatch* on the *receiver object*, the implicit **`self`** or **`this`** parameter
  - secretly, class constructors stash references to their method implementations inside all objects
  - single dispatch is what you learned in CSE 143

multiple dispatch

- *double dispatch* is when the look-up protocol uses both the receiver object and some other parameter
  - neither Ruby nor Java have built-in support for this
  - we can emulate it, though, and in fact, we will on the next homework!

- double dispatch generalizes to using any number of parameters, and this general concept is *multiple dispatch* or *multimethods*

emulating multimethods

you can emulate multiple dispatch by using single dispatch once for each parameter for dispatch

strategy: a method *m* in each class *k* calls a specialized method m_k on the next parameter for dispatch, passing the remaining parameters and its own receiver object (*e.g.*, **`self`**)

each method in such a sequence (*e.g.*, m, m_k, $m_{k,k'}$, …) knows the class for one more argument, so the last method will know the class of every argument

other languages provide different mechanisms to accomplish the same outcome, like nested case expressions in SML

emulating multimethods

you can emulate multiple dispatch by using single dispatch once for each parameter for dispatch

strategy: a method *m* in each class *k* calls a specialized method m_k on the next parameter for dispatch, passing the remaining parameters and its own receiver object (*e.g.*, **`self`**)

each method in such a sequence (*e.g.*, m, m_k, $m_{k,k'}$, …) knows the class for one more argument, so the last method will know the class of every argument

induction?

other languages provide different mechanisms to accomplish the same outcome, like nested case expressions in SML

quick demo

example

```
class A
  def f x
    x.f_A self
  end

  def f_A a
    puts “(A, A)”
  end

  def f_B b
    puts “(B, A)”
  end
end
```

```
class B
  def f x
    x.f_B self
  end

  def f_A a
    puts “(A, B)”
  end

  def f_B b
    puts “(B, B)”
  end
end
```

triple dispatch!

exercise:

extend the previous example to do *triple dispatch* on arguments **x**, **y**, and **z**, printing **(**$C_{\mathbf{x}}$**,** $C_{\mathbf{y}}$**,** $C_{\mathbf{z}}$**)**, where $C_{\mathbf{x}}$ is the class of **x**, $C_{\mathbf{y}}$ is the class of **y**, and $C_{\mathbf{z}}$ is the class of **z** (either **A** or **B** by assumption)

bonus:

can you find a way to achieve the same result using reflection instead of multiple dispatch?

mixins

mixins

- a *mixin* is just a collection of methods
  - like class, but you can't instantiate it

- languages with mixins typically allow inheritance from one superclass and *inclusion* of any number of mixins
  - solves most use cases for multiple inheritance without all the pain

- including a mixin adds its methods to the class
  - prioritized by order of inclusion, so later mixins can override methods from earlier ones
  - mixin methods can access **`self`** like any other method

example

```
module Doubler
  def double
    self + self
  end
end

class String
  include Doubler
end

class Numeric
  include Doubler
end
```

```
class Point
  include Doubler
  attr_accessor :x, :y

  def initialize(x=0, y=0)
    @x = x; @y = y
  end

  def + other
    Point.new(self.x + other.x,
              self.y + other.y)
  end
end
```

quick demo

mixin method dispatch

- with mixins, need to revise the protocol for method look-up

- to find the right method body for a call *o*`.`*m*, look in the following places for a method with the name *m*:
  1. the class of *o*
  2. the mixins of the class of *o*, in order
  3. the superclass of *o*
  4. the mixins of the superclass of *o*, in order
  5. …

- instance variables work the same, because mixin and normal methods can interact with the same object
  - however, bad style for mixins to use “private” instance variables, because their names might conflict with a class’s!

common mixins

- two of the more popular and useful Ruby mixins:
  - **`Comparable`**, which implements `<`, `>`, `==`, `!=`, `>=`, and `<=` in terms of `<=>`
  - **`Enumerable`**, which implements iterators like **`map`** and **`find`** in terms of **`each`**

- these are emblematic of the spirit of mixins
  - classes implement simple core functionality (*e.g.*, `<=>` and **`each`**), and mixins provide convenient abstractions on top
  - classes don't have to waste their one superclass on some utility functionality!

visitor pattern

visitor pattern

- the “visitor pattern” is a “design pattern” in object-oriented programming for functional composition
  - in object-oriented programming, code is grouped into classes
  - we sometimes want to group code by their functionality, to make adding another operation easier in the future

- the visitor pattern is an application of double dispatch!

- commonly used with abstract syntax trees (a language implementation’s representation of program syntax)

visitor pattern

spread functionality across operand types

- the "visitor pattern" is a "design pattern" in object-oriented programming for functional composition
  - in object-oriented programming, code is grouped into classes
  - we sometimes want to group code by their functionality, to make adding another operation easier in the future

- the visitor pattern is an application of double dispatch!

- commonly used with abstract syntax trees (a language implementation's representation of program syntax)

quick demo