CSE 341: Section 9

Yuma & Taylor
University of Washington

Overview

- Homework(s) 5 & 6 check-in, reminder about homework 7
- Double dispatch
- Mixins
- Visitor Pattern

Double dispatch

Double dispatch: overview

What is dispatch? It's the *runtime procedure* used to determine which function to invoke based on given parameters.

- **Single Dispatch**: use `self` (c.f., Java's `this`) to determine which method to invoke.
- **Double Dispatch**: use the runtime class of both `self` *and* a single method parameter.

Double dispatch: emulating in Ruby

Ruby *does not* natively support double-dispatch, so we *emulate* it by doing single-dispatch twice.

1. Have the principal method immediately call another method on its argument, passing self as an argument to that method.
2. The second call now knows...
   a. (Implicitly) the class of `self`
   b. (Explicitly) the class of the argument, based on the method that was called

Double dispatch: example

	Even	Odd
Even	Even	Even
Odd	Even	Odd

Double dispatch: programming

Demo!

Double dispatch: Ruby example

```ruby
class Even
  def mult(m)
    m.mult_even self
  end
  def mult_even(m)
    Even.new(n * m.n)
  end
  def mult_odd(m)
    Even.new(n * m.n)
  end
end
```

```ruby
class Odd
  def mult(m)
    m.mult_odd self
  end
  def mult_even(m)
    Even.new(n * m.n)
  end
  def mult_odd(m)
    Odd.new(n * m.n)
  end
end
```

Double dispatch: SML example

```
datatype parity = Even of int | Odd of int

fun make_num n =
  case (n mod 2) of
       0 => Even n
     | _ => Odd n

fun mult m n =
  case (m, n) of
       (Even m, Even n) => Even (m * n)
     | (Even m, Odd n ) => Even (m * n)
     | (Odd m,  Even n) => Even (m * n)
     | (Odd m,  Odd n ) => Odd  (m * n)
```

Mixins

Mixins: overview

- A *mixin* is a collection of methods
  - Ruby modules and mixins are the same thing
- Different from a class because you cannot make an instance of a mixin
  - In Ruby (and many languages), usually a class can only have one superclass but can include any number of mixins
- *Including* a mixin in a class:
  - Makes the methods in the mixin part of the class
  - Methods in the mixin can reference methods and instance variables on self that are not defined in the mixin

Mixin Example

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

```
# Questionable style but
# still interesting...

class Fixnum
    include Doubler
end

class String
    include Doubler
end
```

Mixin Example

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

# simple 2D point class that
includes the Doubler Mixin --->
# Note: This class provides an
implementation of +
```

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

  def + other
    ans = Pt.new
    ans.x = self.x + other.x
    ans.y = self.y + other.y
    ans
  end
end
```

Method Lookup Rules with Mixins

Looking for a method **m** in receiver **obj**:

1. Check for m in *obj's class*
2. Check the *mixins that obj includes* (later mixins shadow earlier mixins)
3. Check for m in *obj's superclass*
4. Check the *mixins that obj's superclass includes*
5. etc...

Mixin methods are included in the same object, so it's usually bad style for mixin methods to use instance variables since names can clash.

Two Most Common Mixins in Ruby

`Comparable` (http://ruby-doc.org/core-2.2.3/Comparable.html)

- Defines <, >, ==, ! =, >=, <= in terms of <=>
  - In other words, all you have to do is define <=> and `include Comparable` to get <, >, ==, ! =, >=, <= for free
- The <=> operator is a comparison operator that returns -1, 0, or +1 depending on if the receiver is less than, equal to, or greater than the given other object
  - Similar to Java's `compareTo` method

Two Most Common Mixins in Ruby

`Enumerable` (http://ruby-doc.org/core-2.2.3/Enumerable.html)

- Defines many iterators (`map`, `inject`, `select`, `any?`, `all?`, etc.) in terms of `each`
  - In other words, all you have to do is define `each` and `include Enumerable` to get `map`, `inject`, `select`, `any?`, `all?`, etc. for free
- The `each` method must produce successive members of the collection
  - Conceptually similar to iterators in Java and other languages
- If you include both `Comparable` and `Enumerable`, you also get access to various sorting methods for free

Visitor pattern

Visitor pattern

Scenario: say you have some expression language and want to define a number of operations *over* that language.

E.g., convert arithmetic expression to a string, evaluate an arithmetic expression, add one to all constants, etc.

Visitor pattern: programming

Demo!