CSE 341
Section 9

Autumn 2019

Today’s Agenda

- Double Dispatch
- The Visitor Pattern
- Mixins

Double Dispatch

Dispatch Overview

Dispatch is the *runtime* procedure for looking up which function to call based on the parameters given:

- Ruby (and Java) use *Single Dispatch* on the implicit **self** (or "this") parameter
  - Uses runtime class of **self** to lookup the method when a call is made
  - This is what you learned in CSE 143
- *Double Dispatch* uses the runtime classes of both **self** and a single method parameter
  - Ruby/Java do not have this, but we can emulate it
  - This is what you will do in HW7
- You can dispatch on any number of the parameters and the general term for this is *Multiple Dispatch* or *Multimethods*

Emulating Double Dispatch

- To emulate double dispatch in Ruby (on HW7) just use the built-in single dispatch procedure ***twice!***
  - Have the principal method immediately call another method on its *first parameter*, passing **self** as an argument
  - The second call will implicitly know the class of the **self** parameter
  - It will also know the class of the *first parameter* of the principal method, because of *Single Dispatch*
- There are other ways to emulate double dispatch
  - Found as an idiom in SML by using case expressions

Double Dispatch Example: RPS

- Suppose we wanted to code up a game of “Rock-Paper-Scissors”:
  - A game that is played in rounds with 2 players.
  - Each player gets to pick a weapon: one of “Rock”, “Paper”, or “Scissors”.
- Each combination results in a winner/loser (except when both are the same):
  - Rock beats Scissors
  - Paper beats Rock
  - Scissors beats Paper

Double Dispatch Example: RPS

- **What are the different combinations of games?**
  - Player 1 fights Player 2 with a tool, and Player 2 responds, which determines the outcome.

Player 1

Player 2

	Rock	**Paper**	**Scissors**
Rock	Tie	Paper wins	Rock wins
Paper	Paper wins	Tie	Scissor wins
Scissors	Rock wins	Scissor wins	Tie

Double Dispatch Example: RPS

- **How could we represent this in an OOP way?**
  - How does “Class 1” fight “Class 2”? How do we encode the “tool”? How do we encode the “outcome”?

Class 1

Class 2

	Rock	**Paper**	**Scissors**
Rock	Tie	Paper wins	Rock wins
Paper	Paper wins	Tie	Scissor wins
Scissors	Rock wins	Scissor wins	Tie

Double Dispatch Example: RPS

Code!

Double Dispatch Exercise: What's the table? (hint, it's 2x2)

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

    def fWithA a
        "(a, a) case"
    end

    def fWithB b
        "(b, a) case"
    end
end
```

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

    def fWithA a
        "(a, b) case"
    end

    def fWithB b
        "(b, b) case"
    end
end
```

Double Dispatch Exercise: What's the table?

Class 1

Class 2

	A	**B**
A	(a,a) case	(b,a) case
B	(a,b) case	(b,b) case

Extending RPS I

- **What if we wanted to extend our game to add Laser into the game**
  - What would we have to change so that we could still play this game, but with another action?

	Rock	**Paper**	**Scissors**
Rock	Tie	Paper wins	Rock wins
Paper	Paper wins	Tie	Scissor wins
Scissors	Rock wins	Scissor wins	Tie

Extending RPS I

- **What if we wanted to extend our game to add Laser into the game**
  - What would we have to change so that we could still play this game, but with another action?

	Rock	**Paper**	**Scissors**
Rock	Tie	Paper wins	Rock wins
Paper	Paper wins	Tie	Scissor wins
Scissors	Rock wins	Scissor wins	Tie
Laser			

Extending RPS I

- **What if we wanted to extend our game to add Laser into the game**
  - What would we have to change so that we could still play this game, but with another action?

	Rock	**Paper**	**Scissors**	**Laser**
Rock	Tie	Paper wins	Rock wins	Laser Wins
Paper	Paper wins	Tie	Scissor wins	Laser Wins
Scissors	Rock wins	Scissor wins	Tie	Laser Wins
Laser	Laser Wins	Laser Wins	Laser Wins	Laser Wins

Mixins

Mixins

- Collection of methods
  - Unlike class, you cannot instantiate it
- Can include any number of mixins
- Provides powerful extensions to the class with little cost

Mixins

- It’s just “Copy and paste the code into the class”
  - Have access to instance functions
  - Have access to instance variables

Mixin Example

```
module Doubler
  def double
    self + self # assume included in classes w/ +
  end
end
class String
  include Doubler
end
class AnotherPt
  attr_accessor :x, :y
  include Doubler
  def + other
    ans = AnotherPt.new
    ans.x = self.x + other.x
    ans.y = self.y + other.y
    ans
  end
end
```

Method Lookup Rules

1. Current class
2. Current class's mixins
   a. Latest included mixin
   b. …..
   c. Earliest included mixin
3. Current class's super class
4. Current class's super class's mixins
5. Current class's super class's super class
6. Current class's super class's super class's mixins
7. ……

Comparable

It provides you methods to compute <, >, ==, !=, >=, <=

What's needed?

- Define function **<=>** (spaceship operator)
  - Return negative, 0 or positive number

Very similar to Java Comparable interface which requires CompareTo

Enumerable

It provides you methods to iterator over the object -> supports map, find!

What's needed?

- Define function **each**
  - Each will either call each of other object or will yield result

Very similar to Java Iterable interface

The Visitor Pattern

The Visitor Pattern

- A template for handling a functional composition in OOP
  - OOP wants to group code by classes
  - We want code grouped by functions
    - This makes it easier to add operations at a later time.
- Relies on Double Dispatch!!!
  - Dispatch based on (VisitorType, ValueType) pairs.
- Often used to compute over AST's (abstract syntax trees)
  - Heavily used in compilers

accept(visitor, arg)

```
class Int
  attr_reader :i
  def initialize i
    @i = i
  end
  def accept(visitor, arg=nil)
    visitor.visitInt(self, arg)
  end
end
```

A Sample Visitor

```
class Stringer # ⇐ operation we want to add
  def visitInt(int, arg)
    int.i.to_s
  end
  def visitFraction(frac, arg) # ... end
  def visitRational(rational, arg) # ... end
end

Int.new(5).accept(Stringer.new())

class Checker #... end
class Summoner # ... end
```