



PAUL G. ALLEN SCHOOL  
OF COMPUTER SCIENCE & ENGINEERING

# CSE 341

# Section 9

Fall 2018

Adapted from slides by Nick Mooney, Nicholas Shahan, Cody Schroeder, and Dan Grossman

# Today's Agenda

- Double Dispatch Again
- 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.

The diagram shows a 3x3 payoff matrix for Rock-Paper-Scissors (RPS). The columns represent Player 1's choices (Rock, Paper, Scissors) and the rows represent Player 2's choices (Rock, Paper, Scissors). A curved line above the matrix is labeled "Player 1", and a curved line to the left is labeled "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

	<b>Rock</b>	<b>Paper</b>	<b>Scissors</b>
<b>Rock</b>	Tie	Paper wins	Rock wins
<b>Paper</b>	Paper wins	Tie	Scissor wins
<b>Scissors</b>	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

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

Class 2

# Extending RPS I

- **What if we wanted to extend our game to add an action to convert each of the tools to strings?**
  - What would we have to change so that we could still play this game, but with another action?

	<b>Rock</b>	<b>Paper</b>	<b>Scissors</b>
<b>Rock</b>	Tie	Paper wins	Rock wins
<b>Paper</b>	Paper wins	Tie	Scissor wins
<b>Scissors</b>	Rock wins	Scissor wins	Tie
<b>toString*</b>	Rock	Paper	Scissors

\* note: not a Class, but a method, because it only operates on 1 class, not 2.

# 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

# Visitor Example: RPS

Code!

# Extending RPS II

- **What if we wanted to extend our game to add a new tool: Laser?**
  - What would we have to change so that we could still play this game, but with 4 tools instead of 3?

	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	Tie



Have to do it the hard  
way... :(

Is functional better?

# 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"
  - Will override existing code
- 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. ....

# Comparable

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

What's needed?

- Define function  $\Leftrightarrow$  (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