Outline

Dispatch Overview

Mixins

The Visitor Pattern
Dispatch Overview

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

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

**Noun** Single Dispatch (plural single dispatches)

1. (computing) A dispatch method where the implementation of a function or method is chosen solely on the type of the instance calling the method.
Dispatch Overview

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

**Noun** Single Dispatch (plural single dispatches)

1. (computing) A dispatch method where the implementation of a function or method is chosen solely on the type of the instance calling the method.

- Ruby (and Java) use **Single Dispatch** on the implicit `self` parameter
  - Uses runtime class of `self` to lookup the method when a call is made
**Dispatch Overview**

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

**Noun** Single Dispatch (plural single dispatches)

1. (computing) A dispatch method where the implementation of a function or method is chosen solely on the type of the instance calling the method.

- Ruby (and Java) use **Single Dispatch** on the implicit `self` parameter
  - Uses runtime class of `self` to lookup the method when a call is made
- **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)
  - You will do this in **HW7** (my favorite homework tied with HW5)
Dispatch Overview

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

**Noun** Single Dispatch (plural single dispatches)

1. (computing) A dispatch method where the implementation of a function or method is chosen solely on the type of the instance calling the method.

- Ruby (and Java) use **Single Dispatch** on the implicit `self` parameter
  - Uses runtime class of `self` to lookup the method when a call is made
- **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)
  - You will do this in **HW7** (my favorite homework tied with HW5)

**Multiple Dispatch** (or **Multimethods**) is the generalization of Double Dispatch
Dispatch Overview
Emulating Double Dispatch

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
Emulating Double Dispatch in Ruby is as simple as using the built-in Single Dispatch twice.

\footnote{The method being called}
Emulating Double Dispatch in Ruby is as simple as using the built-in Single Dispatch twice

- Have the principal method\(^1\) call another method on its first parameter and pass yourself (i.e. literally `self`) as an argument

---

\(^1\)The method being called
Emulating Double Dispatch in Ruby is as simple as using the built-in Single Dispatch twice

- Have the principal method\(^1\) call another method on its first parameter and pass yourself (i.e. literally `self`) as an argument
- The method the principal method is calling will implicitly know the class of the `self` parameter passed to it (it was defined to deal with this class)

\(^1\)The method being called
Emulating Double Dispatch in Ruby is as simple as using the built-in Single Dispatch twice

- Have the principal method\(^1\) call another method on its first parameter and pass yourself (i.e. literally `self`) as an argument
- The method the principal method is calling will implicitly know the class of the `self` parameter passed to it (it was defined to deal with this class)
- By Single Dispatch, the method the principal method is calling will also know the class of the principal method’s first parameter

\(^1\)The method being called
Mixins

A **mixin** is just a collection of methods

- Less than a class (there are no instances of mixins)

Languages with **mixins** will typically let a class have one superclass, but *any* number of mixins it wants to include
Mixins

A **mixin** is just a collection of methods

- Less than a class (there are no instances of mixins)

Languages with **mixins** will typically let a class have one superclass, but *any* number of mixins it wants to include

When a class includes a mixin, the methods from the mixin are now part of the class
Mixins

A **mixin** is just a collection of methods
- Less than a class (there are no instances of mixins)

Languages with **mixins** will typically let a class have one superclass, but *any* number of mixins it wants to include

**When a class includes a mixin, the methods from the mixin are now part of the class**
- Extending or overriding depends on the order in which mixins are included in the class definition
- Often more powerful than helper methods because **mixin** methods have access to `self` (and instance variables) not defined in the **mixin**
Mixins

module Doubler
  def double
    # Assumes this is included in classes with ‘+’
    self + self
  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
**Mixins** change our lookup rules slightly

Given an object $O$ that is receiving a message $m$: 
Mixins change our lookup rules slightly

Given an object $O$ that is receiving a message $m$:

- Look for $m$ in $O$’s class. If it wasn’t there,
Mixins change our lookup rules slightly

Given an object \( O \) that is receiving a message \( m \):

- Look for \( m \) in \( O \)'s class. If it wasn’t there,
- Look for \( m \) in \( O \)'s mixins. If it wasn’t there,
Mixins change our lookup rules slightly

Given an object $O$ that is receiving a message $m$:

- Look for $m$ in $O$’s class. If it wasn’t there,
- Look for $m$ in $O$’s mixins. If it wasn’t there,
- Look for $m$ in $O$’s superclass. If it wasn’t there,
Mixins change our lookup rules slightly

Given an object \( O \) that is receiving a message \( m \):

- Look for \( m \) in \( O \)'s class. If it wasn’t there,
- Look for \( m \) in \( O \)'s mixins. If it wasn’t there,
- Look for \( m \) in \( O \)'s superclass. If it wasn’t there,
- Look for \( m \) in \( O \)'s superclass’ mixins. If it wasn’t there,
- ...
Mixins change our lookup rules slightly.

Given an object $O$ that is receiving a message $m$:

- Look for $m$ in $O$’s class. If it wasn’t there,
- Look for $m$ in $O$’s mixins. If it wasn’t there,
- Look for $m$ in $O$’s superclass. If it wasn’t there,
- Look for $m$ in $O$’s superclass’ mixins. If it wasn’t there,
- ...

Regarding *instance variables*, the *mixin* methods are included in the same object

- It is bad style for mixin methods to use instance variables since names can clash.
Mixins
The Two Big Ones

Here are two powerful mixins in Ruby

• Comparable — Defines $<$, $>$, $\leq$, $\geq$, $\neq$ in terms of $\leq \Rightarrow$
  • http://ruby-doc.org/core-2.2.3/Comparable.html

• Enumerable — Defines many iterators (e.g. map, find) in terms of each
  • http://ruby-doc.org/core-2.2.3/Enumerable.html
Mixins
The Two Big Ones

Here are two powerful mixins in Ruby

- **Comparable** — Defines <, >, >=, <=, != in terms of <=>
  - [http://ruby-doc.org/core-2.2.3/Comparable.html](http://ruby-doc.org/core-2.2.3/Comparable.html)

- **Enumerable** — Defines many iterators (e.g. `map`, `find`) in terms of `each`
  - [http://ruby-doc.org/core-2.2.3/Enumerable.html](http://ruby-doc.org/core-2.2.3/Enumerable.html)
The Visitor Pattern

• OOP wants code grouped by classes

• We want code grouped by functions

• Grouping by function makes it easier to add functionality later

This pattern relies on Double Dispatch

• Dispatch is based on (Visitor Type, Value Type) pairs

• Heavily used in compilers

• Often used to compute over ASTs (abstract syntax trees)
The Visitor Pattern

A template for handling a functional composition in OOP
The Visitor Pattern

A template for handling a functional composition in OOP

- OOP wants code grouped by classes
- We want code grouped by functions
  - Grouping by function makes it easier to add functionality later
The Visitor Pattern

A template for handling a functional composition in OOP

- OOP wants code grouped by classes
- We want code grouped by functions
  - Grouping by function makes it easier to add functionality later

This pattern relies on **Double Dispatch**

- Dispatch is based on \((\text{Visitor Type}, \text{Value Type})\) pairs
The Visitor Pattern

A template for handling a functional composition in OOP

- OOP wants code grouped by classes
- We want code grouped by functions
  - Grouping by function makes it easier to add functionality later

This pattern relies on **Double Dispatch**

- Dispatch is based on \(<\text{Visitor Type}, \text{Value Type}>\) pairs

Heavily used in compilers

- Often used to compute over ASTs (abstract syntax trees)