CSE 341 AB: Section 9

Josh Pollock

Office Hours: Tuesdays 3:00pm - 4:00pm

Questions?

HW 5, HW 6, early HW 7
Lecture Material

Agenda

- Modules
  - Namespaces
  - Mixins
- Double Dispatch
- Visitor Pattern (if time)

Modules

Ruby Modules

A module is like a class, **except:**

- You define it with the **`module`** keyword.
- You can't create instances of it. (No **`new`**.)
- You can't specify a superclass for it. (No **`< superclass`**.)
- You can include it in a class using the **`include`** keyword.

Why These Design Choices?

Modules serve two purposes in Ruby:

1. Namespaces
    a. Define constants and class methods.
    b. E.g. **`Math::PI`** and **`Math.cos(5)`**

2. Mixins
    a. Assume a class defines a certain instance method(s) (e.g. **`<=>`** or **`each`**).
    b. Define default impls of other instance methods that use that method(s). (e.g. **`<=`**, **`map`**)
    c. Classes can **`include`** the module, define the instance method, and get the others for free.

`Math` Namespace

Mixins

Simple Examples

`Comparable` Mixin

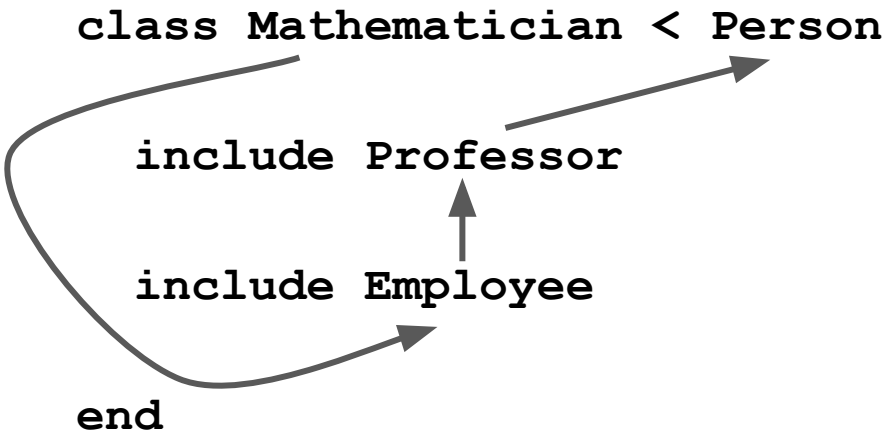
Assumes class defines <=>.

`Enumerable` Mixin

Assumes class defines **`each`**.

Method Lookup Order with Mixins

1. Class
2. Mixins, bottom up
3. Superclass



Double Dispatch

Multiple Dispatch

What arguments determine the **`method`** definition we use?

```
obj.method(a, b, c)
```

Single dispatch

```
obj.method(a, b, c)
```

Double dispatch

```
obj.method(a, b, c)
```

Triple dispatch

```
obj.method(a, b, c)
```

...

Problem: Ruby only has single dispatch!
How can we emulate double dispatch?

Aside: A Different Way to Write Method Calls

obj.method_name(arg1, arg2, …, argn)

obj.send(:method_name, arg1, arg2, …, argn)

(Almost) the same! The only difference is that send allows you to call private methods, too.

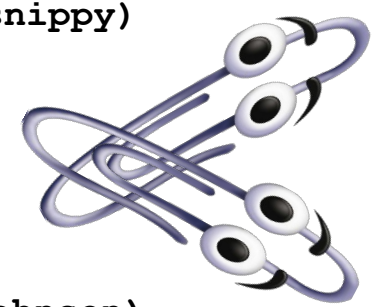
The pedagogical advantage of send is that it makes the OOP thought process clearer.

We are sending (dynamically *dispatching*) a method and its args to an object.

Double Dispatch Example

```
dwayne_johnson = Rock.new
snippy = Scissors.new

dwayne_johnson.fight(snippy)
```



```
send(:fight, snippy)
```

```
dwayne_johnson
```

```
send(:fightWithRock, dwayne_johnson)
```

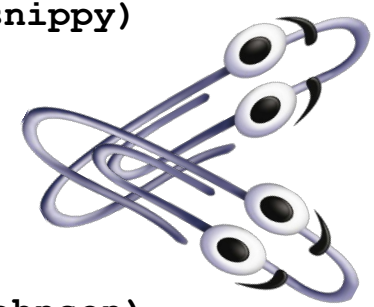
```
snippy
```

Double Dispatch Example

```
dwayne_johnson = Rock.new
snippy = Scissors.new

dwayne_johnson.fight(snippy)
```

The sender's type is encoded in this method.



```
send(:fight, snippy)
```

→ `dwayne_johnson`

```
send(:fightWithRock, dwayne_johnson)
```

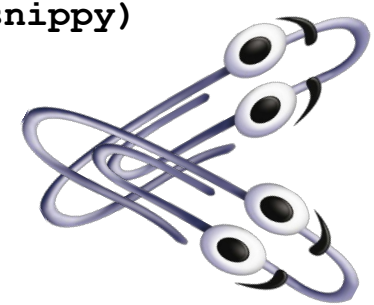
→ `snippy`

Double Dispatch Example

```
dwayne_johnson = Rock.new
snippy = Scissors.new

dwayne_johnson.fight(snippy)
```

dwayne_johnson is actually **self**.



`send(:fight, snippy)` → `dwayne_johnson` → `send(:fightWithRock, self)` → `snippy`

Demo!

The Kicker

Multiple dispatch is branching on the types of the arguments to a method.

This is much easier to do in a language with pattern matching constructs!

Visitor Pattern

A Common Pattern in Compilers

I have an AST and I want to…

- interpret it.
- print it as a string.
- serialize it to some bytes.
- compile it to an abstract machine.
- partially evaluate it.

A Common Pattern in Compilers

I have an AST and I want to…

- interpret it.
- print it as a string.
- serialize it to some bytes.
- compile it to an abstract machine.
- partially evaluate it.

These are all recursive traversals (visitors) on each variant of the AST.
Remember how HW5 went! Evaluate the subexpressions, then combine them.

These rely on pattern matching, but how can we write these in Ruby?

Use Double Dispatch!

Dispatch lets us do pattern matching.

Make every variant a class. This allows us to define some default behaviors.

Each traversal is a **visitor** and each AST variant is a **node**.

We will use **double dispatch** to match on nodes and visitors.
But crucially, *the first dispatch only needs to be written once!*