```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Binary Methods with OOP: Double Dispatch

Example

To show the issue:

- Include variants **`String`** and **`Rational`**
- (Re)define **`Add`** to work on any pair of **`Int`**, **`String`**, **`Rational`**
  - Concatenation if either argument a **`String`**, else math

Now just defining the addition operation is a *different* 2D grid:

	`Int`	**`String`**	**`Rational`**
`Int`			
`String`			
`Rational`			

Worked just fine with functional decomposition -- what about OOP…

What about OOP?

Starts promising:

- Use OOP to call method **`add_values`** to one value with other value as result

```
class Add
  …
  def eval
    e1.eval.add_values e2.eval
  end
end
```

Classes **`Int`**, **`MyString`**, **`MyRational`** then all implement

- Each handling 3 of the 9 cases: "add **`self`** to argument"

```
class Int
  …
  def add_values v
    … # what goes here?
  end
end
```

First try

- This approach is common, but is "not as OOP"
  - *So do not do it on your homework*

```
class Int
  def add_values v
    if v.is_a? Int
       Int.new(v.i + i)
    elsif v.is_a? MyRational
       MyRational.new(v.i+v.j*i,v.j)
    else
       MyString.new(v.s + i.to_s)
  end
end
```

- A "hybrid" style where we used dynamic dispatch on 1 argument and then switched to Racket-style type tests for other argument
  - Definitely not "full OOP"

Another way…

- **`add_values`** method in **`Int`** needs "what kind of thing" **`v`** has
  - Same problem in **`MyRational`** and **`MyString`**

- In OOP, "always" solve this by calling a method on **`v`** instead!

- But now we need to "tell" **`v`** "what kind of thing" **`self`** is
  - We know that!
  - "Tell" **`v`** by calling different methods on **`v`**, passing **`self`**

- Use a "programming trick" (?) called *double-dispatch*…

Double-dispatch "trick"

- **`Int`**, **`MyString`**, and **`MyRational`** each define all of **`addInt`**, **`addString`**, and **`addRational`**
  - For example, **`String`**'s **`addInt`** is for adding concatenating an integer argument to the string in **`self`**
  - 9 total methods, one for each case of addition

- **`Add`**'s **`eval`** method calls **`e1.eval.add_values e2.eval`**, which dispatches to **`add_values`** in **`Int`**, **`String`**, or **`Rational`**
  - **`Int`**'s **`add_values`**: **`v.addInt self`**
  - **`MyString`**'s **`add_values`**: **`v.addString self`**
  - **`MyRational`**'s **`add_values`**: **`v.addRational self`**

  So **`add_values`** performs "2nd dispatch" to the correct case of 9!

[Definitely see the code]

Why showing you this

- Honestly, partly to belittle full commitment to OOP
- To understand dynamic dispatch via a sophisticated idiom
- Because required for the homework
- To contrast with *multimethods* (optional)

Optional note: Double-dispatch also works fine with static typing

  - See Java code
  - Method declarations with types may help clarify