



Object-Oriented Design Lecture

About Double Dispatch

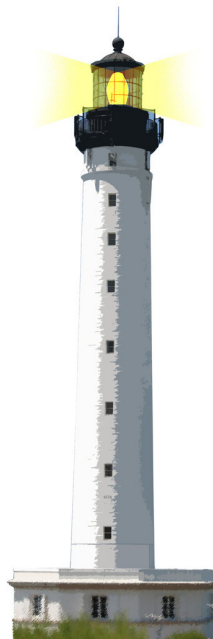
Stéphane Ducasse

<http://stephane.ducasse.free.fr>

<http://car.mines-douai.fr/luc>



<http://www.pharo.org>



Outline

- Some fun exercises
- Thinking about them
- Discovering double dispatch
- Stepping back



Exercise 1

Given

primitive `addi(i,j)` returns $i + j$
primitive `addf(f1,f2)` returns $f1 + f2$
`i.asFloat()` returns a float



Adding Integer and Float

$1 + 2$

> 3

$1.1 + 2$

> 3.1

$2 + 1.3$

$>$

3.3

$1.1 + 2.2$

> 3.3

Implement +

But with

- Not a single explicit conditional
- No static type support



A first hint

- Two classes Integer and Float



Let us see

```
Integer >> + aNumber  
"fill me up :)"
```

```
Float >> + aNumber
```

```
"fill me up :)"
```



Another key hint

When you execute a method you know that the receiver is from the class of the method!



Even more hints

- Remember the Boolean implementation
- Sending a message to an object is a choice operator



Let us get started

Imagine that we add one method `sumWithInteger: anInteger`

```
Integer >> + aNumber
```

```
Integer >> sumWithInteger: anInteger
```

```
Float >> + aNumber
```

```
"fill me up :)"
```



Look like an easy definition

```
Integer >> + aNumber
```

```
Integer >> sumWithInteger: anInteger  
  ^ addi(self, anInteger)
```

```
Float >> + aNumber
```

```
"fill me up :)"
```

How do we connect them?

```
Integer >> + aNumber
```

```
  ^ aNumber sumWithInteger: self
```

```
Integer >> sumWithInteger: anInteger
```

```
  ^ addi(self, anInteger)
```

```
Float >> + aNumber
```

```
"fill me up :)"
```



On Float too

```
Integer >> + aNumber  
^ aNumber sumWithInteger: self
```

```
Integer >> sumWithInteger: anInteger  
^ addi(self, anInteger)
```

```
Float >> + aNumber
```

```
Float >> sumWithInteger: anInteger  
"fill me up :)"
```



On Float too

```
Integer >> + aNumber  
^ aNumber sumWithInteger: self
```

```
Integer >> sumWithInteger: anInteger  
^ addi(self, anInteger)
```

```
Float >> + aNumber
```

```
Float >> sumWithInteger: anInteger  
^ addf(self, asFloat(anInteger))
```



Supporting 1.2 + 2

```
Integer >> + aNumber  
  ^ aNumber sumWithInteger: self  
Integer >> sumWithInteger: anInteger  
  ^ addi(self, anInteger)
```

```
Float >> + aNumber  
  ^ aNumber sumWithFloat: self
```

```
Float >> sumWithInteger: anInteger  
  ^ addf(self, asFloat(anInteger))
```



Supporting 1.2 + 2

```
Integer >> + aNumber
```

```
  ^ aNumber sumWithInteger: self
```

```
Integer >> sumWithInteger: anInteger
```

```
  ^ addi(self, anInteger)
```

```
> Integer >> sumWithFloat: aFloat
```

```
>  ^ addf(aFloat, asFloat(self))
```

```
Float >> + aNumber
```

```
  ^ aNumber sumWithFloat: self
```

```
Float >> sumWithInteger: anInteger
```

```
  ^ addf(self, asFloat(anInteger))
```

```
> Float >> sumWithFloat: aFloat
```

```
>  ^ addf(self, aFloat)
```



Ok now relax

- Take a pen and follow the calls to the following expressions
- Follow with your fingers if necessary :)

$1 + 2$

$1.1 + 2$

$2 + 1.3$

$1.1 + 2.2$

Key point

`Integer` >> + aNumber
^ aNumber sumWithInteger: `self`

Two choices/messages:

- one for +
- one for sumWithInteger:



Exercise2: How to add Fraction?

```
f:= Fraction num: 1 denum: 2.
```

```
f num
```

```
> 1
```

```
f denum
```

```
> 2
```

```
f asFloat
```

```
> 0.5
```

```
1/2 + 3
```

```
3 + 3.3
```

```
1.3 + 2/5
```

```
1/3 + 4/3
```

Introducing Fraction

```
Fraction >> + aNumber  
  ^ aNumber sumWithFraction: self  
...
```



Introducing Fraction

```
Fraction >> + aNumber  
  ^ aNumber sumWithFraction: self  
Fraction >> sumWithFraction: aFrac  
...
```

Introducing Fraction

```
Fraction >> + aNumber
```

```
^ aNumber sumWithFraction: self
```

```
Fraction >> sumWithFraction: aFrac
```

```
^ Fraction num: (self num * aFrac denum) + (aFrac num * self denum)
```

```
denum: aFrac denum * self denum
```

```
...
```



Taking care of Integer and Float

```
Fraction >> + aNumber
```

```
  ^ aNumber sumWithFraction: self
```

```
Fraction >> sumWithFraction: aFrac
```

```
  ^ Fraction num: (self num * aFrac denum) + (aFrac num * self denum)  
    denum: aFrac denum * self denum
```

```
Integer >> sumWithFraction: aFrac
```

```
...
```

```
Float >> sumWithFraction: aFrac
```

```
...
```



Introducing Fraction

```
Fraction >> + aNumber
```

```
^ aNumber sumWithFraction: self
```

```
Fraction >> sumWithFraction: aFrac
```

```
^ Fraction num: (self num * aFrac denum) + (aFrac num * self denum)  
  denum: aFrac denum * self denum
```

```
...
```

```
Integer >> sumWithFraction: aFrac
```

```
^ Fraction num: (self * aFrac denum) + aFrac num denum: aFrac denum
```

```
Float >> sumWithFraction: aFrac
```

```
^ addf(self, aFrac asFloat)
```



Full code for Fraction

```
Fraction >> + aNumber
  ^ aNumber sumWithFraction: self
Fraction >> sumWithFraction: aFrac
  ^ Fraction num: (self num * aFrac denum) + (aFrac num * self denum)
    denum: aFrac denum * self denum
Fraction >> sumWithInteger: anInteger
  ^ Fraction num: (self num + anInteger * aFrac denum) denum: aFrac denum
Fraction >> sumWithFloat: aFloat
  ^ addf(self aFloat, aFloat)
Integer >> sumWithFraction: aFrac
  ^ Fraction num: (self * aFrac denum) + aFrac num denum: aFrac denum
Float >> sumWithFraction: aFrac
  ^ addf(self, aFrac asFloat)
```



Ok now relax

- Take a pen and follow the calls to the following expressions
- Follow with your fingers if necessary :)

$$1/2 + 3$$

$$3 + 3.3$$

$$1.3 + 2/5$$

$$1/3 + 4/3$$

Stepping back

- We can add Fraction without changing any previous method
- Another example of "Sending a message is making a choice"
- We send two messages
 - + to select Integer, Float, Fraction
 - then the message sumWith... to reselect the correct definition in Integer, Float, Fraction

Different kinds of messages

- Primary operations
- Double dispatching methods



Double Dispatch

- Essence of Visitor Design Pattern (see Lecture)
- Double dispatch is a clear illustration of **Do not ask, Tell** OOP tenet
- Used really frequently for event, drawing, ...



When not using Double Dispatch

- No class to dispatch on
- We need an different instance of dispatch to



What about overloading

- Double dispatch is also useful in statically typed languages
- Avoid overloading for double dispatch - some type systems do not work well (see other lectures on Overloading problem)

Conclusion

- Powerful
- Modular
- Just sending an extra message to an argument and using late binding

A course by

Stéphane Ducasse

<http://stephane.ducasse.free.fr>

and

Luc Fabresse

<http://car.mines-douai.fr/luc>



Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France

<https://creativecommons.org/licenses/by-nc-nd/3.0/fr/>