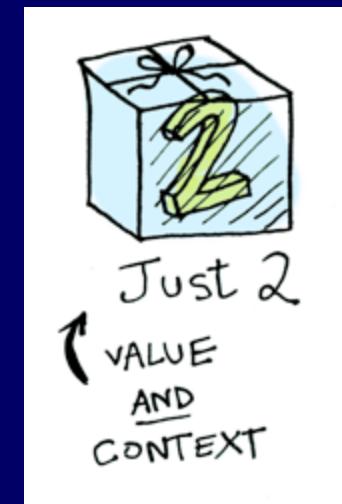
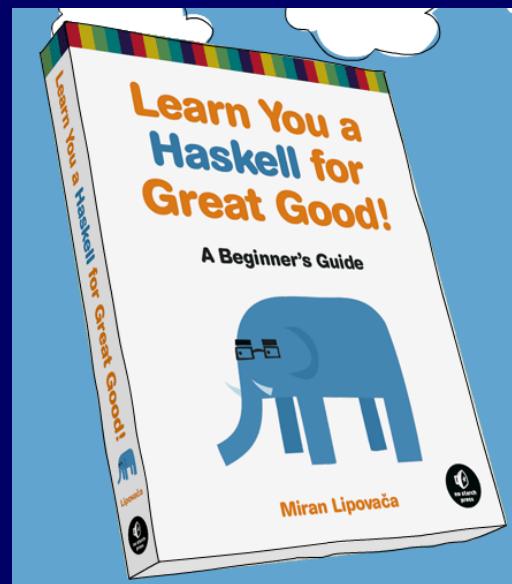
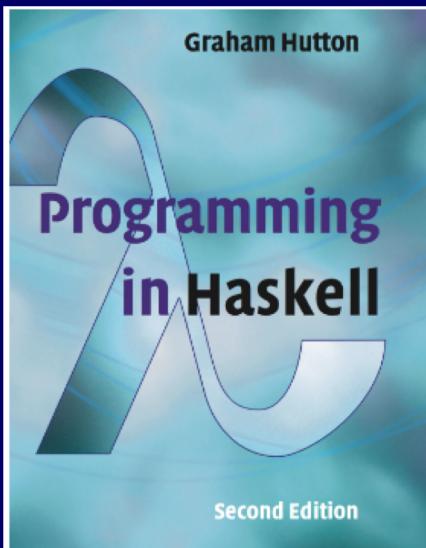


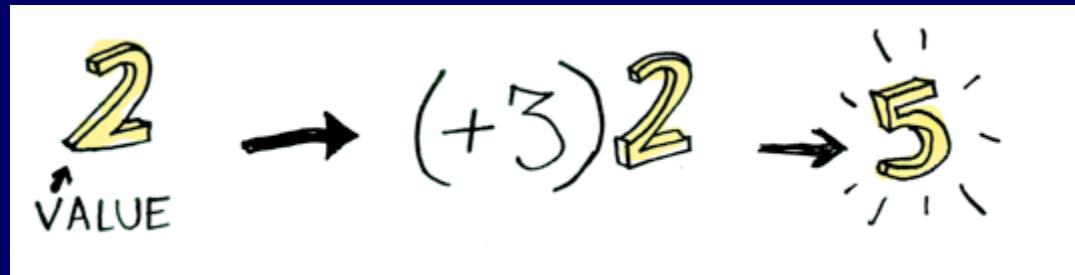
PROGRAMMING IN HASKELL

Functors, Applicatives and monads

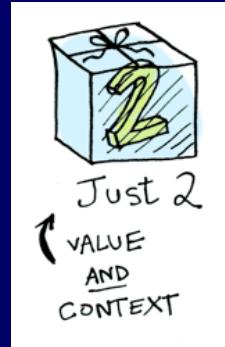


Based on lecture notes by Graham Hutton,
the book “Learn You a Haskell for Great Good”,
pictures from Aditya Bhargava

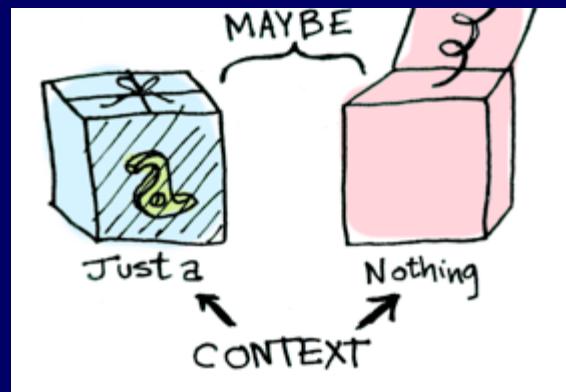
Applying a function to a simple value



Any value can be in a context

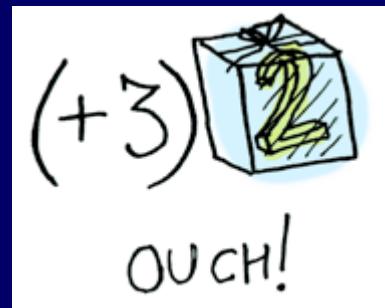


when you apply a function to this value,
you'll get different results **depending on
the context**



Functors

When a value is wrapped in a context, you can't apply a normal function to it



Enter fmap. Fmap knows how to apply functions to values that are wrapped in contexts.

Functors

To apply (+3) to fmap



Functors

1. TO MAKE A DATA TYPE f A FUNCTOR,

class Functor f where

$\rightsquigarrow \text{fmap} :: (a \rightarrow b) \rightarrow fa \rightarrow fb$

2. THAT DATA TYPE
NEEDS TO DEFINE
HOW fmap WILL
WORK WITH IT.

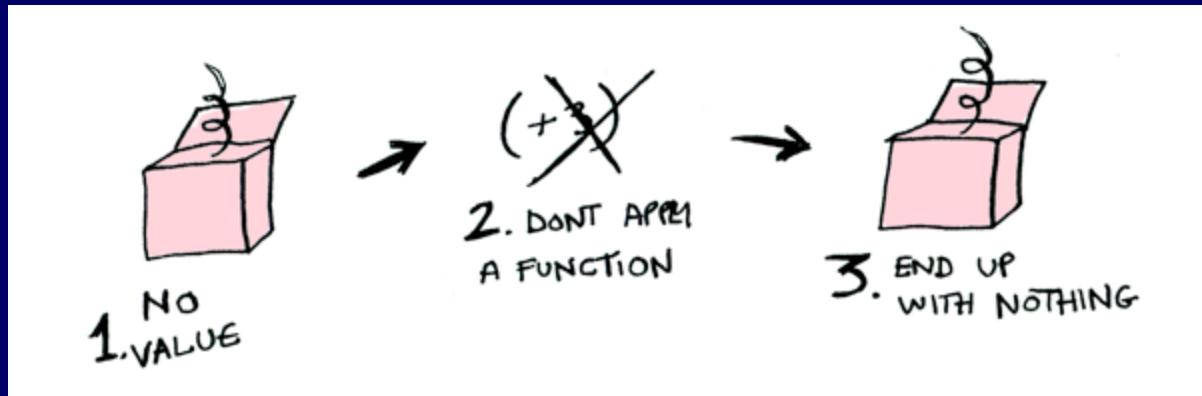
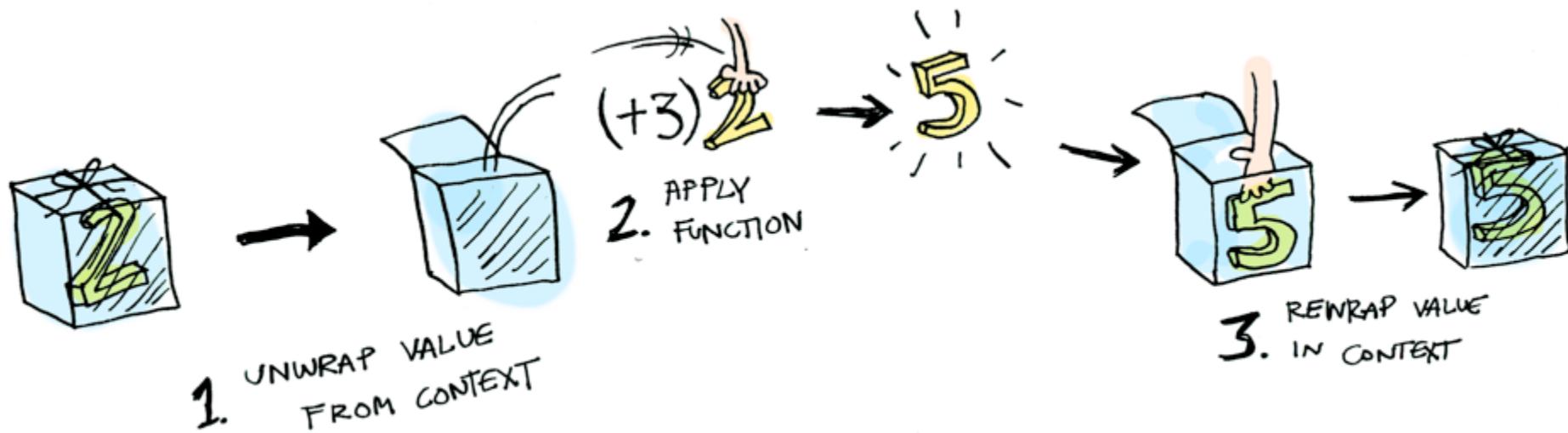
$\text{fmap} :: (a \rightarrow b) \rightarrow fa \rightarrow fb$

1. fmap TAKES A
FUNCTION
(LIKE $(+3)$)

2. AND A
FUNCTOR
(LIKE $\text{Just } 2$)

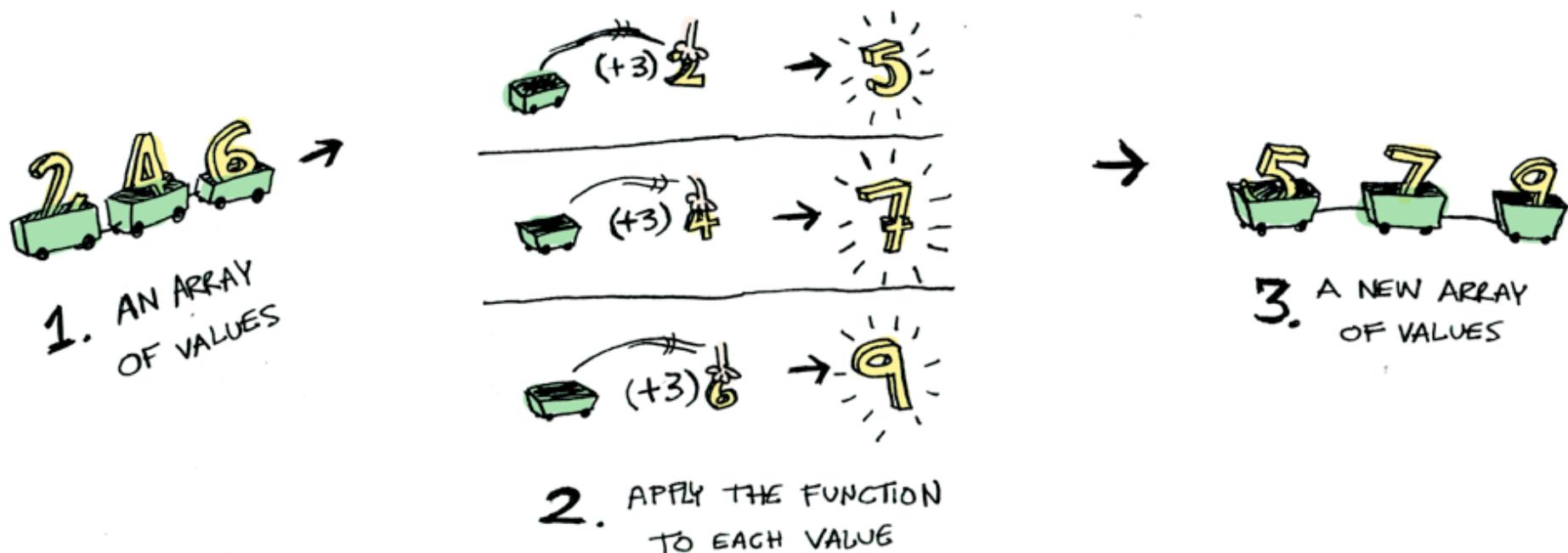
3. AND RETURNS
A NEW FUNCTOR
(LIKE $\text{Just } 5$)

Functors



Functors

main > fmap (+3) [2,4,6]



[5,7,9]

Functors

Functors are a typeclass, just like Ord, Eq, Show, and all the others. This one is designed to hold things that can be mapped over; for example, lists are part of this typeclass.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Essentially, fmap promotes an “ordinary” function, that takes $a \rightarrow b$, to a function that works over values in a context.

Map over Lists

map f xs applies f over all the elements of the list xs

map :: (a -> b) -> [a] -> [b]

map [] = []

map f (x:xs) = f x : map f xs

```
>map (+1) [1,2,3]  
[2,3,4]
```

```
>map even [1,2,3]  
[False,True,False]
```

Map over Binary Trees

Remember binary trees with data in the inner nodes:

```
data Tree a = Leaf
```

```
    | Node (Tree a) a (Tree a)  
deriving Show
```

They admit a similar map operation:

```
mapTree :: (a -> b) -> Tree a -> Tree b
```

```
mapTree _ Leaf = Leaf
```

```
mapTree f (Node l x r) =
```

```
    Node (mapTree f l) (f x) (mapTree f r)
```

Map over optional values

Optional values are represented with Maybe data

Maybe a = Nothing | Just a

How does a map operation over optional values look like?

Map over optional values

Optional values are represented with Maybe data

Maybe a = Nothing | Just a

How does a map operation over optional values look like?

```
mapMay :: (a -> b) -> Maybe a -> Maybe b  
mapMay _ Nothing = Nothing  
mapMay f (Just x) = Just (f x)
```

Map over optional values

mapMay applies a function over a value, only if it is present

```
>mapMay (+1) (Just 1)  
Just 2
```

```
>mapMay (+1) Nothing  
Nothing
```

Maps have similar types

`map :: (a -> b) -> [a] -> [b]`

`mapTree :: (a -> b) -> Tree a -> Tree b`

`mapMay :: (a -> b) -> Maybe a -> Maybe b`

`mapT :: (a -> b) -> T a -> T b`

The difference lies in the type constructor

- ▶ `[]` (list), `Tree`, or `Maybe`
- ▶ Those parts need to be applied to other types

Functors

A type constructor which has a “map” is called a functor

```
class Functor f where  
fmap :: (a -> b) -> f a -> f b
```

```
instance Functor [] where  
-- fmap :: (a -> b) -> [a] -> [b]
```

```
fmap = map
```

```
instance Functor Maybe where  
-- fmap :: (a -> b) -> Maybe a -> Maybe b  
fmap = mapMay
```

Higher Kinded Abstraction

```
class Functor f where  
  fmap :: (a -> b) -> f a -> f b
```

In Functor the variable f stands for a type constructor

- ▶ A “type” which needs to be applied
- ▶ This is called higher-kinded abstraction
 - ▶ Not generally available in a programming language
 - ▶ Haskell, Scala and Rust have it
 - ▶ Java, C# and Swift don’t

Functors generalize maps

Suppose you have a function operating over lists

```
inc :: [Int] -> [Int]
inc xs = map (+1) xs
```

You can easily generalize it by using fmap

```
inc :: Functor f => f Int -> f Int
inc xs = fmap (+1) xs
```

Note that in this case the type of elements is fixed to Int, but the type of the structure may vary

(<\$>) instead of fmap

Many Haskellers use an alias for fmap

(<\$>) = fmap

This allows writing maps in a more natural style, in which the function to apply appears before the arguments

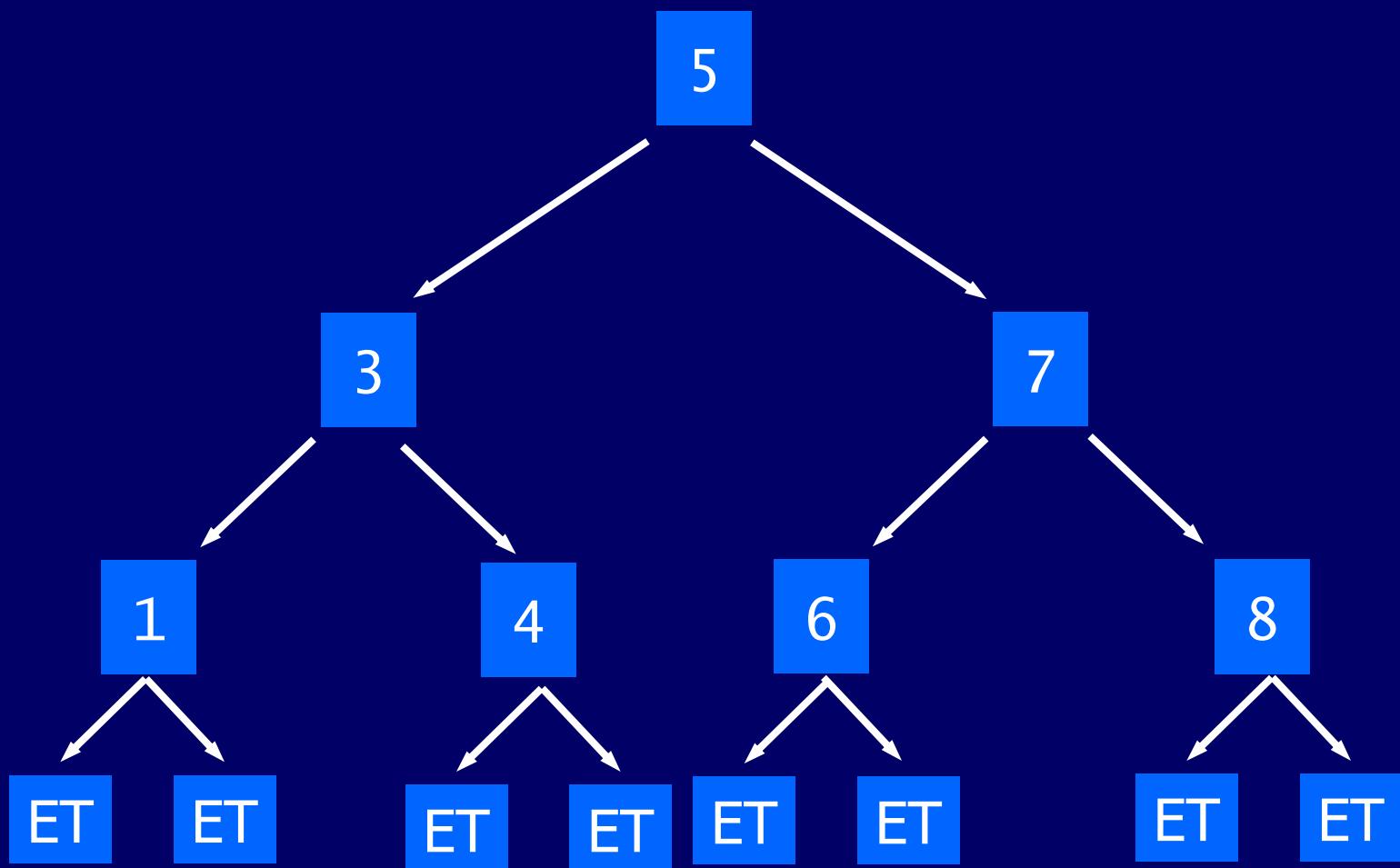
inc xs = (+1) <\$> xs

Back to trees

```
data Tree a = EmptyTree  
| Node a (Tree a) (Tree a)  
    deriving (Show,Read,Eq)
```

An example run:

```
ghci> let nums = [8,6,4,1,7,3,5]
ghci> let numsTree =
      foldr treeInsert EmptyTree nums
ghci> numsTree
Node 5 (Node 3 (Node 1 EmptyTree EmptyTree) (N
ode 4 EmptyTree EmptyTree)) (Node 7 (Node 6 Em
ptyTree EmptyTree) (Node 8 EmptyTree EmptyTre
e))
```



Back to functors:

If we looked at fmap as though it were only for trees, it would look something like:

$(a \rightarrow b) \rightarrow \text{Tree } a \rightarrow \text{Tree } b$

We can certainly phrase this as a functor, also:

```
instance Functor Tree where
    fmap f EmptyTree = EmptyTree
    fmap f (Node x leftsub rightsub) =
        Node (f x) (fmap f leftsub)
                                         (fmap f rightsub)
```

Using the tree functor:

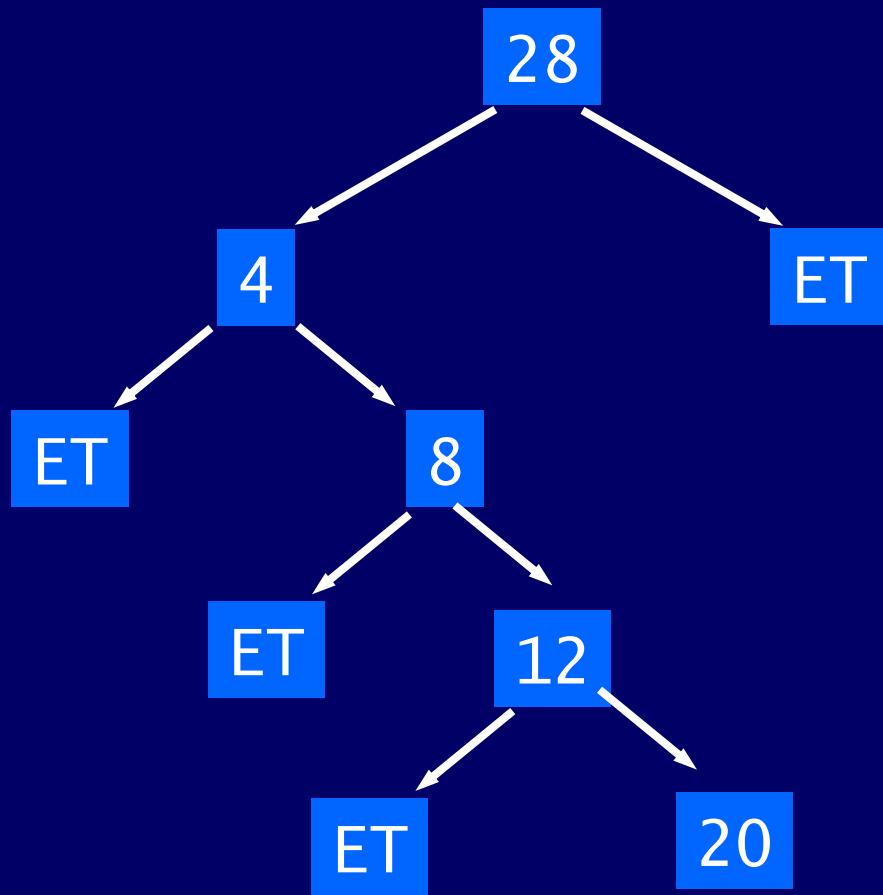
```
ghci> fmap (*2) EmptyTree
```

```
EmptyTree
```

```
ghci> fmap (*4) (foldr treeInsert
```

```
EmptyTree [5,7,3,2,1,7])
```

```
Node 28 (Node 4 EmptyTree (Node 8 EmptyTree (N  
ode 12 EmptyTree (Node 20 EmptyTree EmptyTree  
)))) EmptyTree
```



Another functor: IO actions

```
main = do line <- getLine
          let line' = reverse line
          putStrLn $ "You said " ++ line' ++ " backwards!"
          putStrLn $ "Yes, you really said" ++ line' ++ "
backwards!"
```

In the above code, we are getting a line as an IO action, then reversing it and printing it back out.

But – an IO is a functor, which means it is designed to be mapped over using fmap!

Another functor: IO actions

Old way:

```
main = do line <- getLine
          let line' = reverse line
              putStrLn $ "You said " ++ line' ++ " backwards!"
              putStrLn $ "Yes, you really said " ++ line' ++ "
backwards!"
```

Better way: use `fmap`! This `getLine` has type `IO String`, so `fmap` with a string function will map the function over the result of `getLine`:

```
main = do line <- fmap reverse getLine
          putStrLn $ "You said " ++ line ++ " backwards!"
          putStrLn $ "Yes, you really said " ++ line ++ "
backwards!"
```

Functors and IOs

In general, any time you are binding an IO action to a name, only to call functions on that name, use `fmap` instead!

```
import Data.Char
import Data.List

main = do line <- fmap (intersperse '-' . reverse . map
                      toUpper) getLine
          putStrLn line
```

```
> main
hello there
E-R-E-H-T- -O-L-L-E-H
```

How do these type?

```
ghci> :t fmap (*2)
fmap (*2) :: (Num a, Functor f) => f a -> f a
ghci> :t fmap (replicate 3)
fmap (replicate 3) :: (Functor f) => f a -> f [a]
```

- The expression `fmap (*2)` is a function that takes a functor `f` over numbers and returns a functor over numbers.
 - That functor can be a list, a `Maybe`, an `Either` String, whatever.
- The expression `fmap (replicate 3)` will take a functor over any type and return a functor over a list of elements of that type.

What will one of these do?

```
ghci> fmap (replicate 3) [1,2,3,4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]
ghci> fmap (replicate 3) (Just 4)
Just [4,4,4]
ghci> fmap (replicate 3) Nothing
Nothing
```

- The type `fmap (replicate 3) :: (Functor f) => f a -> f [a]` means that the function will work on any functor. What exactly it will do depends on which functor we use it on.
 - If we use `fmap (replicate 3)` on a list, the list's implementation for `fmap` will be chosen, which is just `map`.
 - If we use it on a `Maybe a`, it'll apply `replicate 3` to the value inside the `Just`, or if it's `Nothing`, then it stays `Nothing`.

Functor laws

There are two laws any functor MUST follow if you define them:

```
fmap id = id  
fmap (g . f) = fmap g . fmap f
```

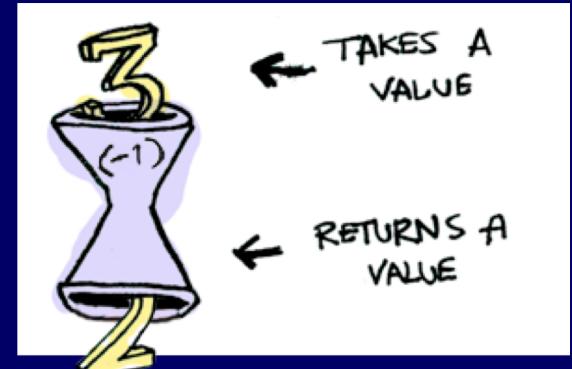
- If we can show that some type obeys both functor laws, we can rely on it having the same fundamental behaviors as other functors when it comes to mapping.
- We can know that when we use `fmap` on it, there won't be anything other than mapping going on behind the scenes and that it will act like a thing that can be mapped over, i.e. a functor.
- This leads to code that is more abstract and extensible, because we can use laws to reason about behaviors that any functor should have and make functions that operate reliably on any functor.

Takeaway: WHY?

- The availability of the `fmap` method relieves us from having to recall, read, and write a plethora of differently named mapping methods (`maybeMap`, `treeMap`, `weirdMap`, ad infinitum). As a consequence, code becomes both cleaner and easier to understand. On spotting a use of `fmap`, we instantly have a general idea of what is going on. Thanks to the guarantees given by the functor laws, this general idea is surprisingly precise.
- Using the type class system, we can write `fmap`-based algorithms which work out of the box with any functor - be it `[]`, `Maybe`, `Tree` or whichever you need. Indeed, a number of useful classes in the core libraries inherit from `Functor`.

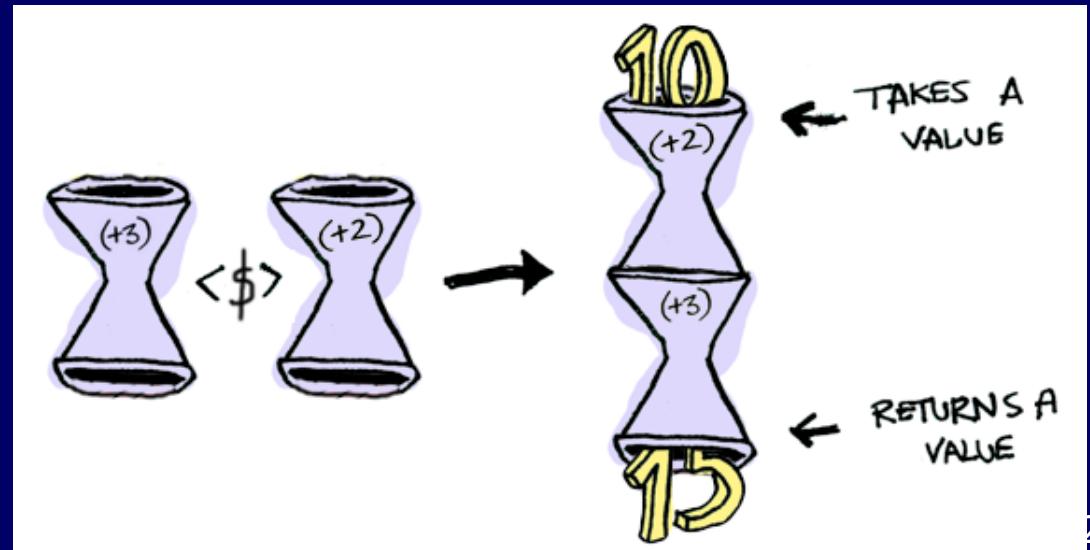
One last thing on functors

```
*Main> (-1) 3  
2
```



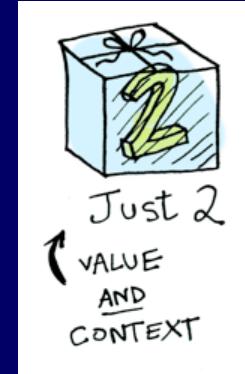
- What happens when you apply a function to another function?

```
*Main> fmap (+3) (+2) 2
```



Applicatives

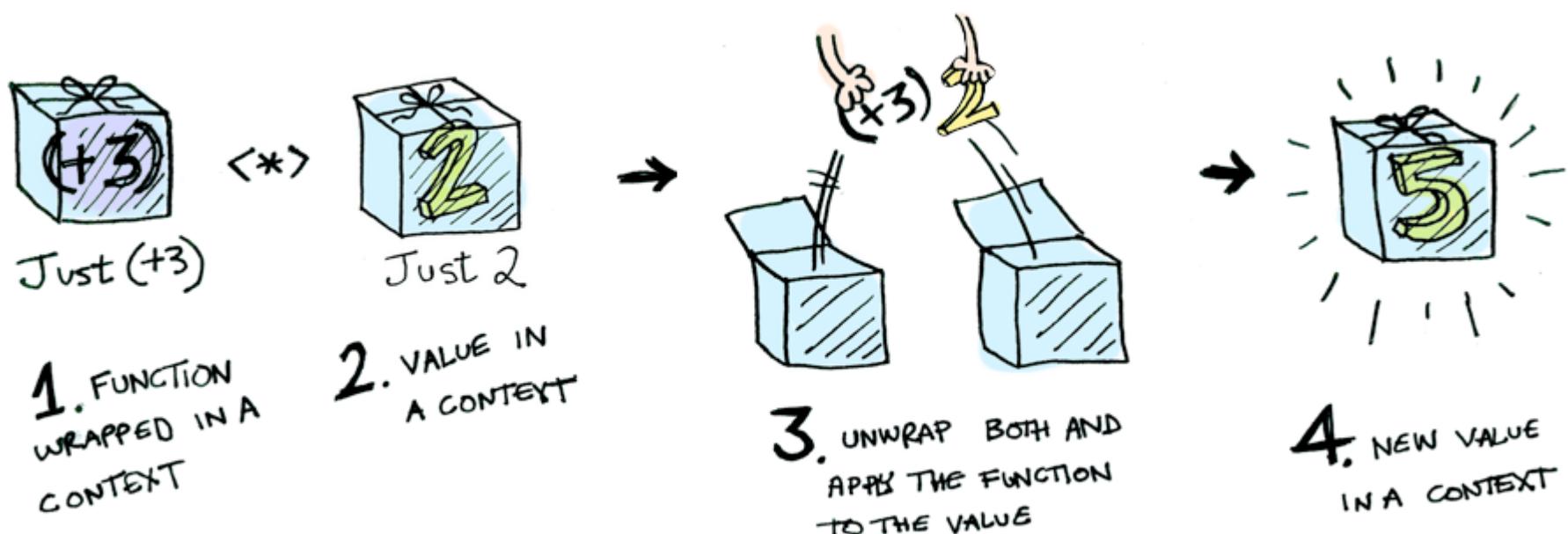
Applicatives take it to the next level. With an applicative, our values are wrapped in a context, just like Functors



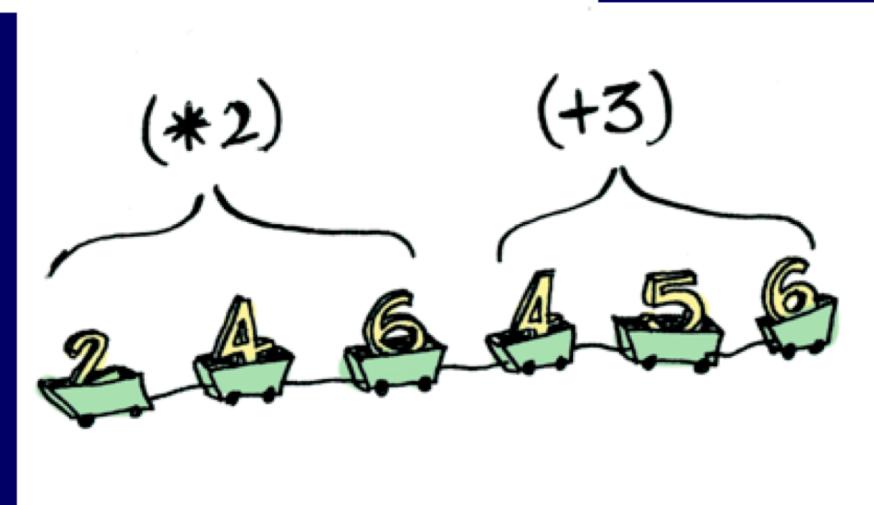
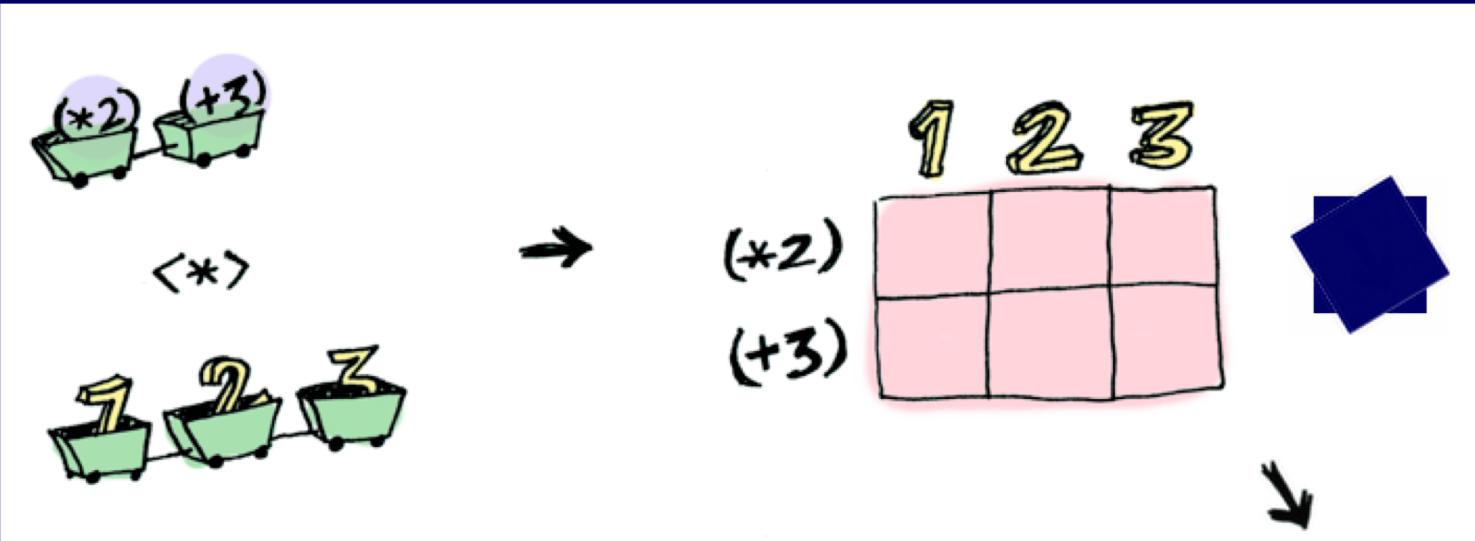
But our functions are wrapped in a context too!



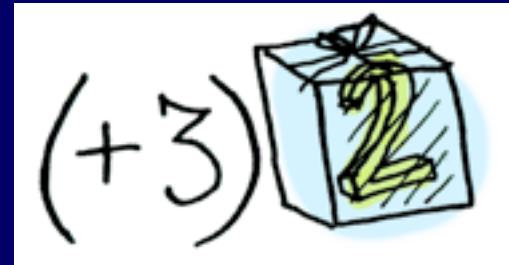
Applicatives



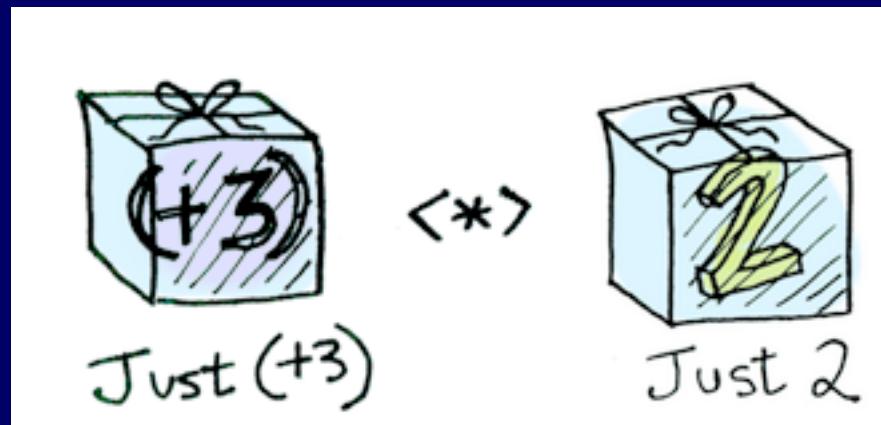
Applicatives



Functors apply a function to a wrapped value:

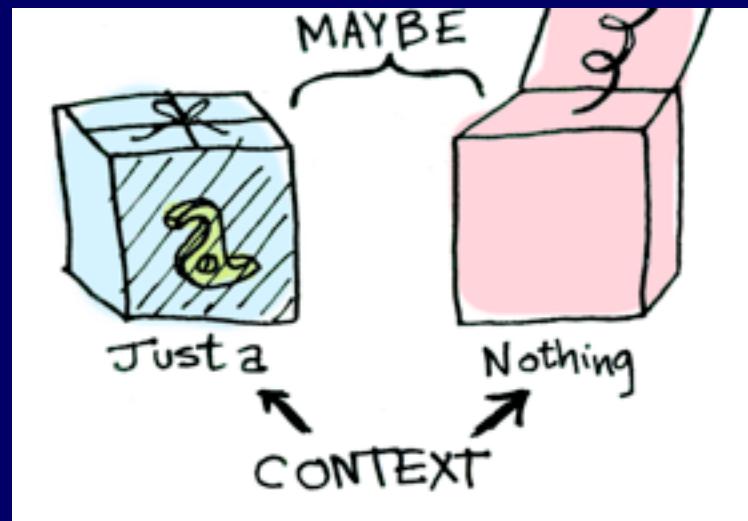


Applicatives apply a wrapped function to a wrapped value:



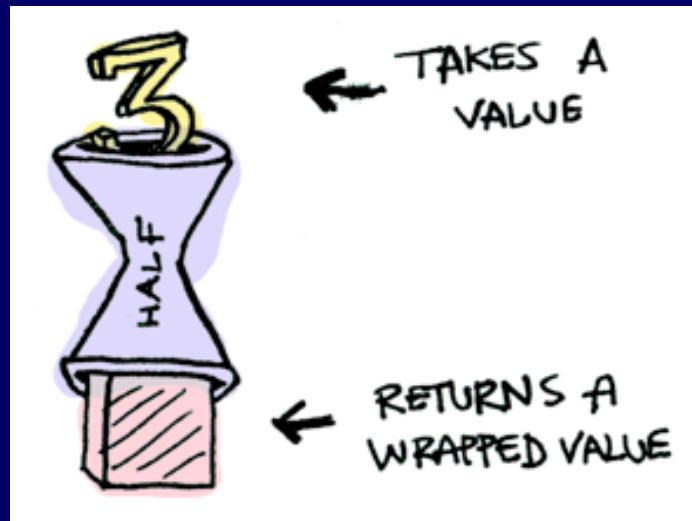
Monads

We look at Maybe to illustrate



Monads

Suppose half is a function that only works on even numbers:



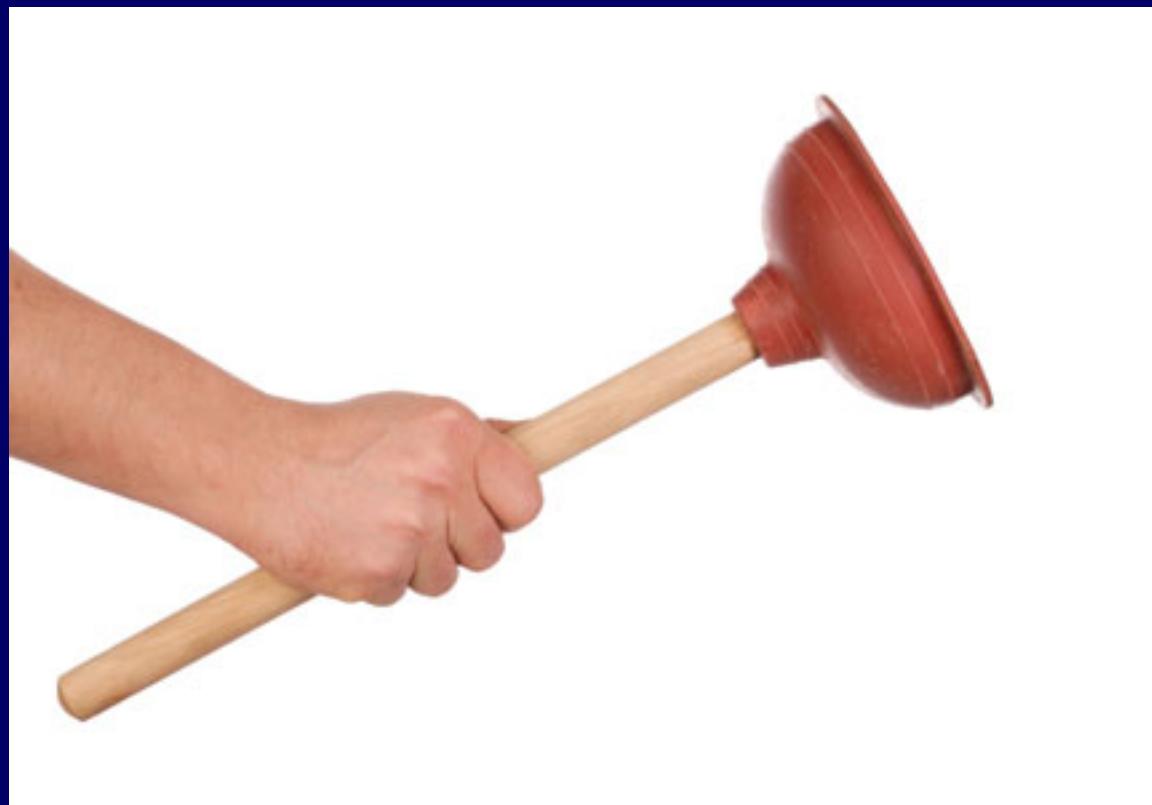
```
half x = if even x  
        then Just (x `div` 2)  
        else Nothing
```

Monads

What if we feed it a wrapped value?



We need to use `>>=` to shove our wrapped value into the function. Here's a photo of `>>=`:



Monads

How does it work?

```
> Just 3 >>= half Nothing  
> Just 4 >>= half Just 2  
> Nothing >>= half Nothing
```

Monads

What's happening inside? Monad is another typeclass. Here's a partial definition:

```
class Monad m where  
(>>=) :: m a -> (a -> m b) -> m b
```

Where $>>=$ is:

$(>>=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b$

The diagram illustrates the components of the (>>=) operator. It consists of three light purple circles with arrows pointing to specific parts of the operator's type signature:

- 1.** An arrow points to the first parameter $m a$, with the text "1. >>= TAKES A MONAD (LIKE Just 3)" below it.
- 2.** An arrow points to the second parameter $a \rightarrow m b$, with the text "2. AND A FUNCTION THAT RETURNS A MONAD (LIKE half)" below it.
- 3.** An arrow points to the result type $m b$, with the text "3. AND IT RETURNS A MONAD" below it.

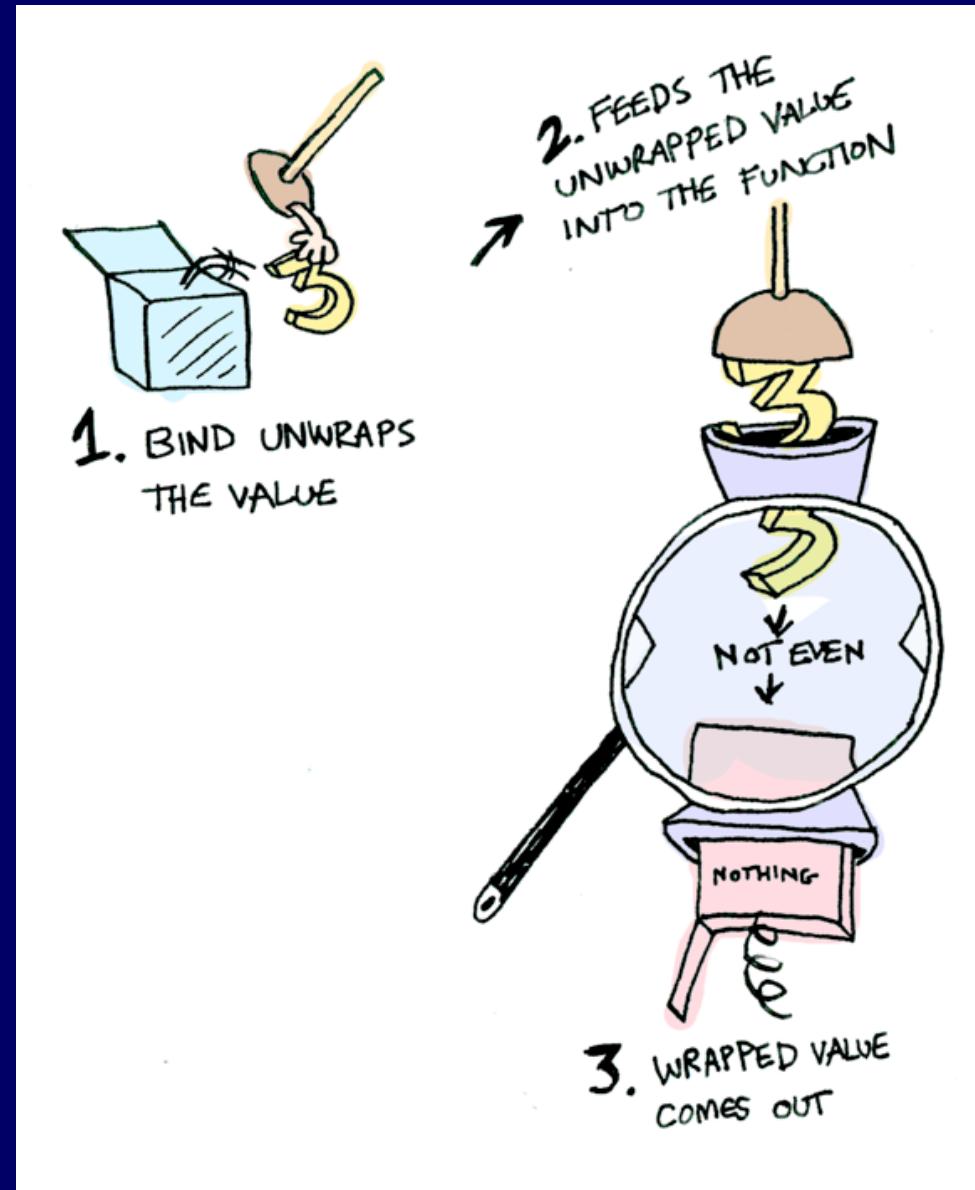
Monads

So Maybe is a Monad:

```
instance Monad Maybe where
    Nothing >>= func = Nothing
    Just val >>= func = func val
```

Monads

Here it is
in action
with
a Just 3!



Monads

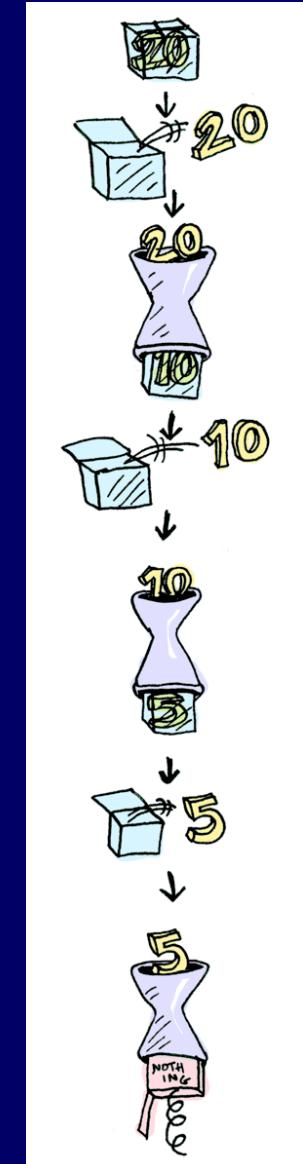
And if you pass
in a Nothing it's
even simpler:



Monads

You can also chain calls:

```
> Just 20 >>= half >>= half >>= half  
Nothing
```



What is a Monad? – Computerphile

Graham Hutton

Computer Science at the University of Nottingham



Monads

A monad is defined by 3 things:

- A type constructor m
- A function `return`
- An operator ($>>=$) – read as “bind”

Originally, these were introduced for IO, but they can be much more powerful (and difficult to understand).

The function and operator are methods of the Monad typeclass and have types themselves, which are required to obey 3 laws (more later):

```
return :: a -> m a  
(>>=)  :: m a -> (a -> m b) -> m b
```

Monads example: Maybe

```
return :: a -> Maybe a
       return x = Just x

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= g = case m of
             Nothing -> Nothing
             Just x -> g x
```

- Maybe is the monad, and return brings a value into it by wrapping it with Just.
- As for ($>>=$), it takes a $m :: \text{Maybe } a$ value and a $g :: a \rightarrow \text{Maybe } b$ function. If m is Nothing, there is nothing to do and the result is Nothing. Otherwise, in the Just x case, g is applied to x , the underlying value wrapped in Just, to give a Maybe b result.
- To sum it all up, if there is an underlying value of type a in m , we apply g to it, which brings the underlying value back into the Maybe monad.

Monads: motivation

Continuing this example:

Imagine a family database with two functions, which look up the parents of a person:

```
father :: Person -> Maybe Person  
mother :: Person -> Maybe Person
```

The `Maybe` is there in case our database is missing some information.

We then might want to build other functions, to look up grandparents, etc (next slide)

Monads: motivation

Now, let's find grandparents:

```
maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
  case mother p of
    Nothing -> Nothing
    Just mom -> father mom
```

But there's a better way! `Maybe` is a monad, so can do:

```
maternalGrandfather p = mother p >>= father
```

Monads:

Even more drastic:

```
bothGrandfathers :: Person -> Maybe (Person, Person)
bothGrandfathers p =
  case father p of
    Nothing -> Nothing
    Just dad ->
      case father dad of
        Nothing -> Nothing
        Just gf1 ->                                -- found first
grandfather
          case mother p of
            Nothing -> Nothing
            Just mom ->
              case father mom of
                Nothing -> Nothing
                Just gf2 ->                                -- found second
grandfather
                Just (gf1, gf2)
```

Becomes:

```
bothGrandfathers p =
  father p >>=
    (\dad -> father dad >>=
      (\gf1 -> mother p >>=    -- gf1 is only used in the
final return
        (\mom -> father mom >>=
          (\gf2 -> return (gf1,gf2 ))))
```

Monads: nuts and bolts

Monad the typeclass lives in `Control.Monad`, and has the following methods:

```
class Applicative m => Monad m where
    return :: a -> m a
    (=>)   :: m a -> (a -> m b) -> m b

    (.)     :: m a -> m b -> m b
    fail    :: String -> m a
```

- Aside from `return` and `bind`, there are two additional methods, `(.)` and `fail`.
 - Both of them have default implementations, and so you don't need to provide them when writing an instance.

Monads: nuts and bolts

Monad is a subclass of Applicative, which is a subclass of Functor. So these are also functors!

Note: (`>>`) and `fail` both have default implementations, and so you don't need to provide them when writing an instance.

`(>>)` sequences two monadic actions when the second action does not involve the result of the first, which is a common scenario for monads such as IO. Simple example:

```
printSomethingTwice :: String -> IO ()  
printSomethingTwice str = putStrLn str >> putStrLn str
```

Monads: back to that example

In fact, that grandfather example can be made even better if we use the do notation with braces and semi-colons.

This will look more like imperative code that we're used to! (Like IOs.)

Here, father and mother are functions that might fail to produce results, i.e. raise an exception, and when that happens, the whole do-block will fail, i.e. terminate with an exception.

```
bothGrandfathers p = do {  
    dad <- father p;  
    gf1 <- father dad;  
    mom <- mother p;  
    gf2 <- father mom;  
    return (gf1, gf2);  
}
```

Monad laws

Monads are required to obey 3 laws:

```
m >>= return      =  m          -- right unit  
return x >>= f    =  f x        -- left unit  
  
(m >>= f) >>= g  =  m >>= (\x -> f x >>= g)  -- associativity
```

Monads originally come from a branch of mathematics called Category Theory.

(Fortunately, it is entirely unnecessary to understand category theory in order to understand and use monads in Haskell.)