

# PROGRAMMING IN HASKELL



## Chapter 11.2 Functors, Applicatives



# Before we start on Functors, remember our Maybe

1

```
data Maybe a = Nothing | Just a
```

we can define:

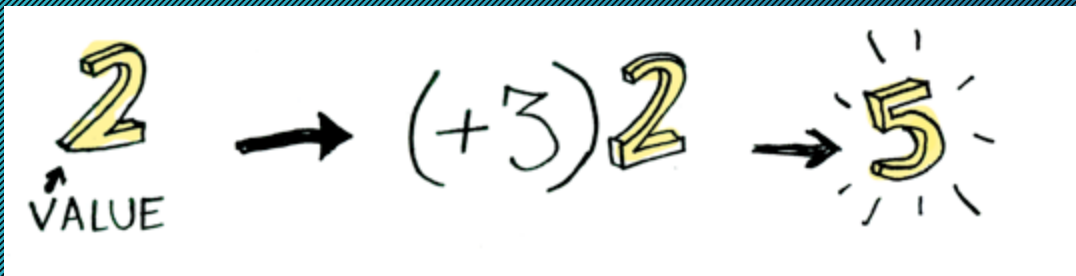
```
safediv :: Int → Int → Maybe Int  
safediv _ 0 = Nothing  
safediv m n = Just (m `div` n)
```

```
safehead :: [a] → Maybe a  
safehead [] = Nothing  
safehead xs = Just (head xs)
```



# Applying a function to a simple value

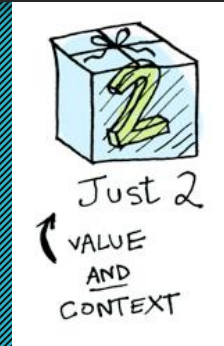
2



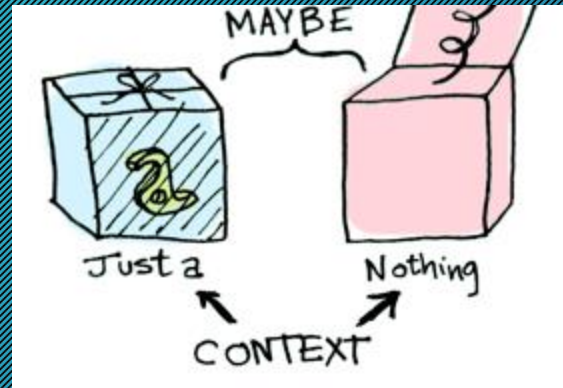


# Any value can be in a context

3



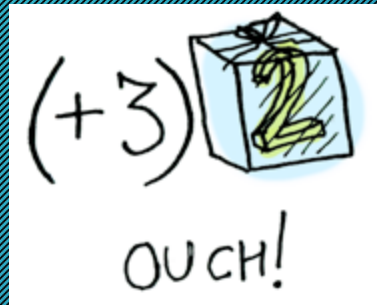
when you apply a function to this value, you'll get different results **depending on the context**





# 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

6

1. TO MAKE A DATA TYPE  $f$   
A FUNCTOR,

class Functor  $f$  where  
 $\rightarrow \text{fmap} :: (a \rightarrow b) \rightarrow f a \rightarrow f b$

2. THAT DATA TYPE  
NEEDS TO DEFINE  
HOW  $\text{fmap}$  WILL  
WORK WITH IT.

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

1.  $\text{fmap}$  TAKES A  
FUNCTION  
(LIKE  $(+3)$ )

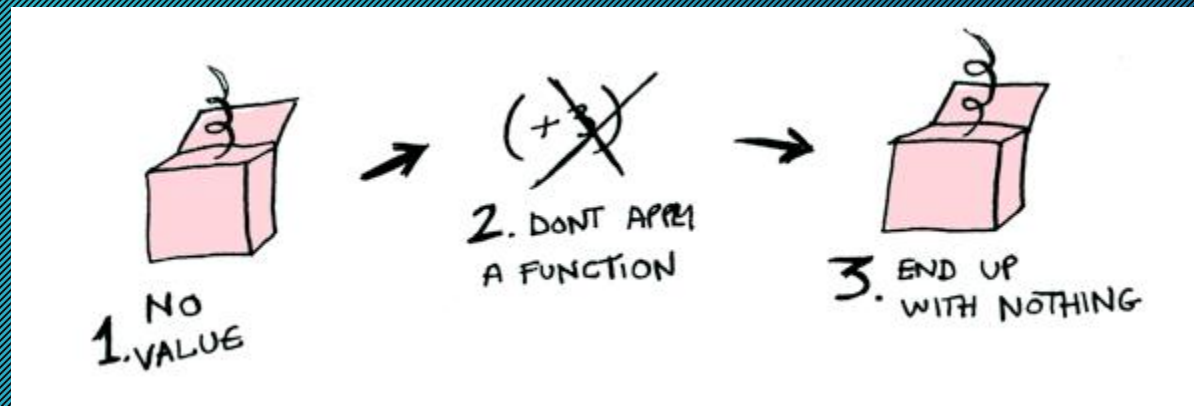
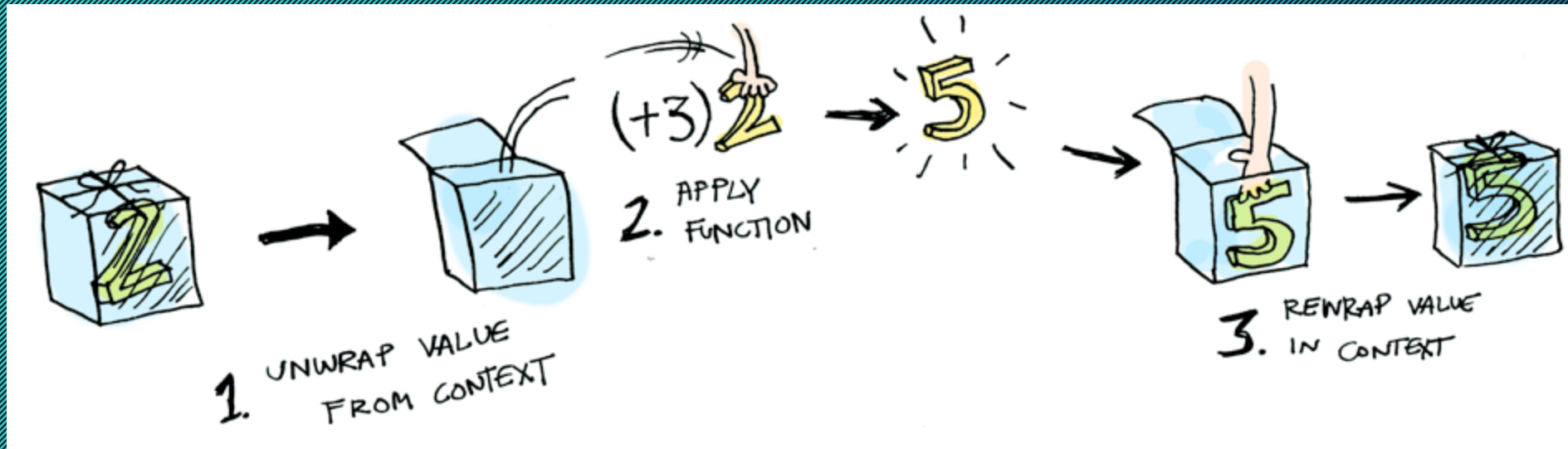
2. AND A  
FUNCTOR  
(LIKE  $\text{Just } 2$ )

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



# Functors

7

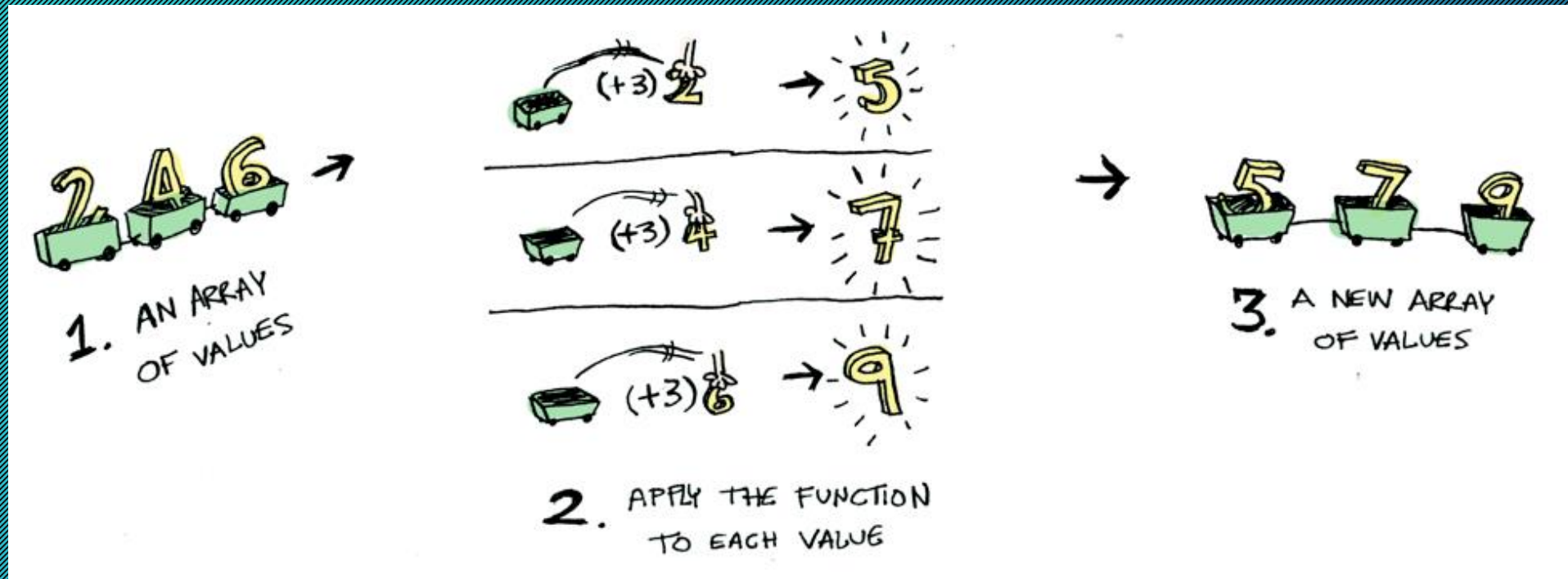




# Functors

8

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



[5,7,9]



# Functors

9

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

10

`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

11

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

12

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

13

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

14

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

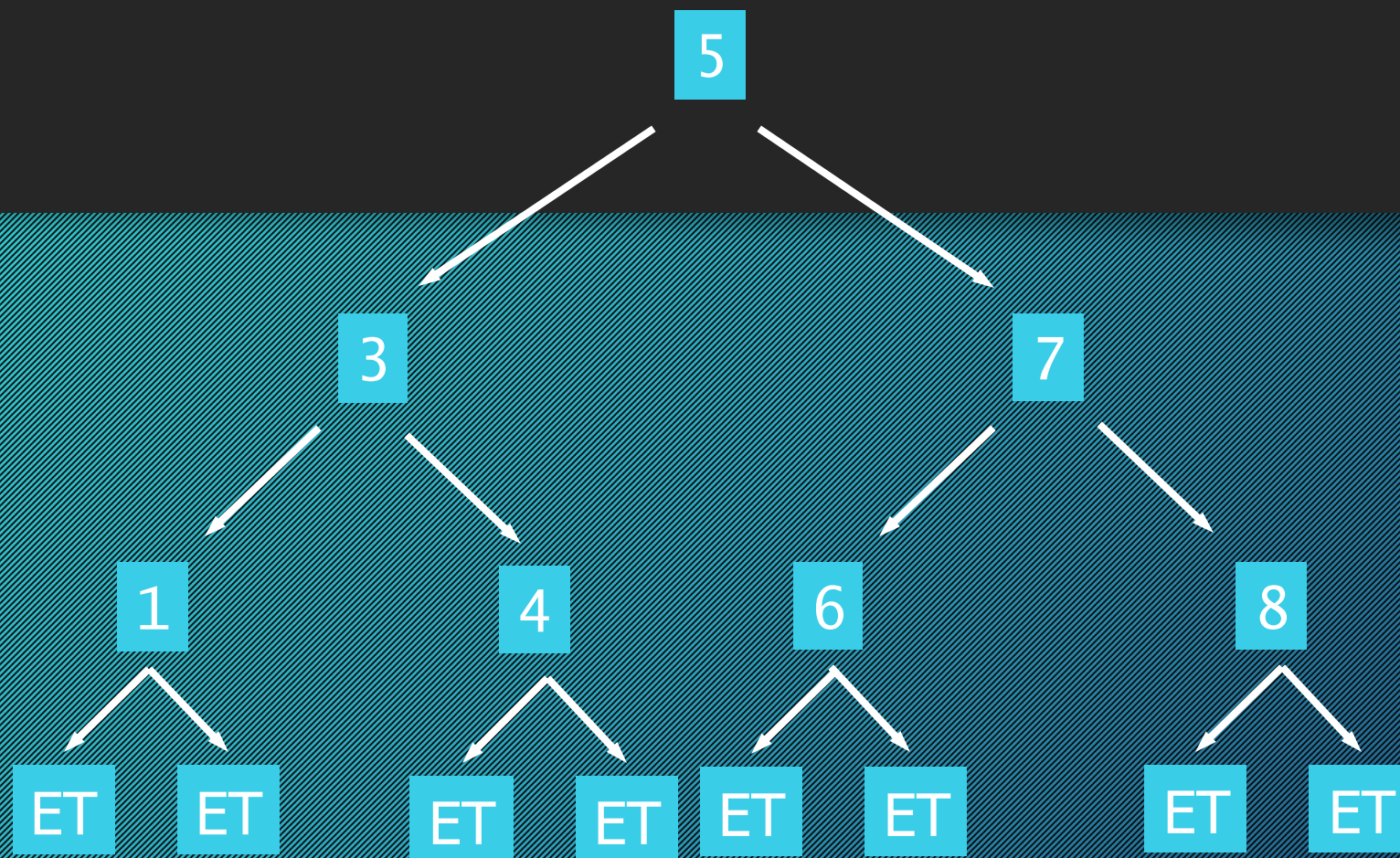
```
data Tree a = EmptyTree
  | Node a (Tree a) (Tree a)
  deriving (Show,Read,Eq)
```



## An example run:

```
ghci> let nums = [8,6,4,1,7,3,5]
ghci> let numTree =
    foldr treeInsert EmptyTree nums
ghci> numTree
Node 5 (Node 3 (Node 1 EmptyTree EmptyTree) (Node 4 EmptyTree EmptyTree)) (Node 7 (Node 6 EmptyTree EmptyTree) (Node 8 EmptyTree EmptyTree))
```







# 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)
```

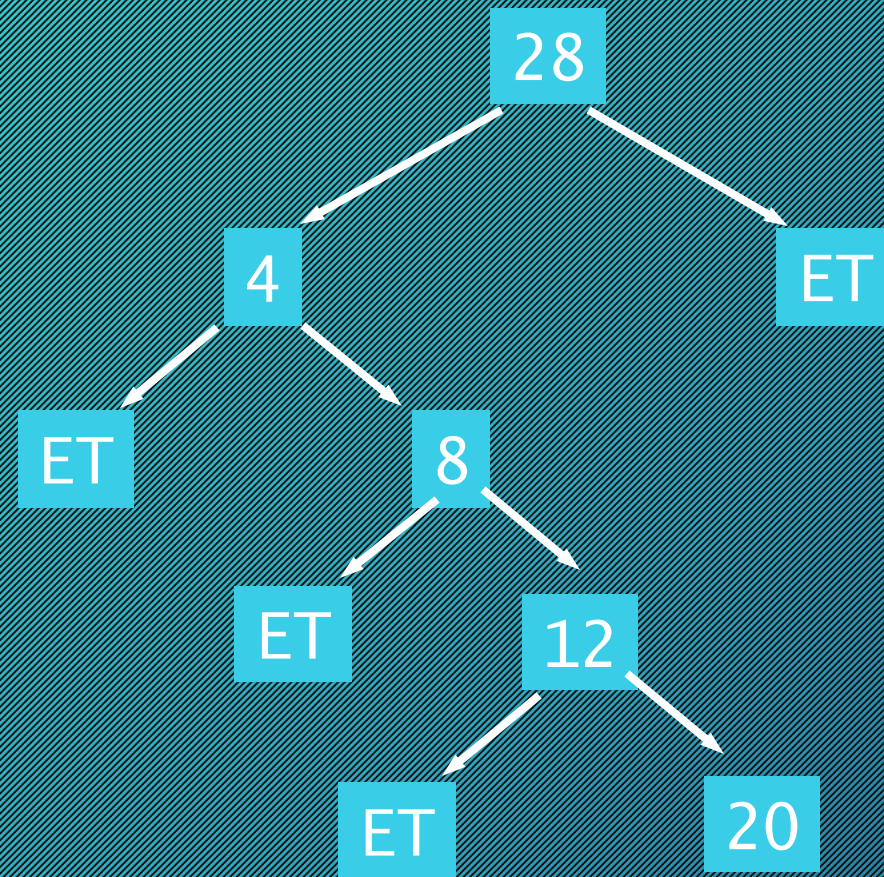


## Using the tree functor:

```
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 (Node 12 EmptyTree (Node 20 EmptyTree EmptyTree)))) EmptyTree
```







# Another functor: IO actions

26

```
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

27

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

28

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?

30

```
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

31

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?

32

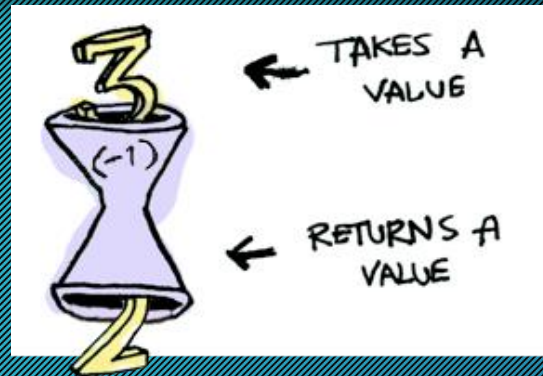
- The availability of the `fmap` method relieves us from having to recall, read, and write a plethora of differently named mapping methods (`maybeMap`, `treeMap`, `weirdMap`, `ad infinitum`). As a consequence, code becomes both cleaner and easier to understand. On spotting a use of `fmap`, we instantly 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

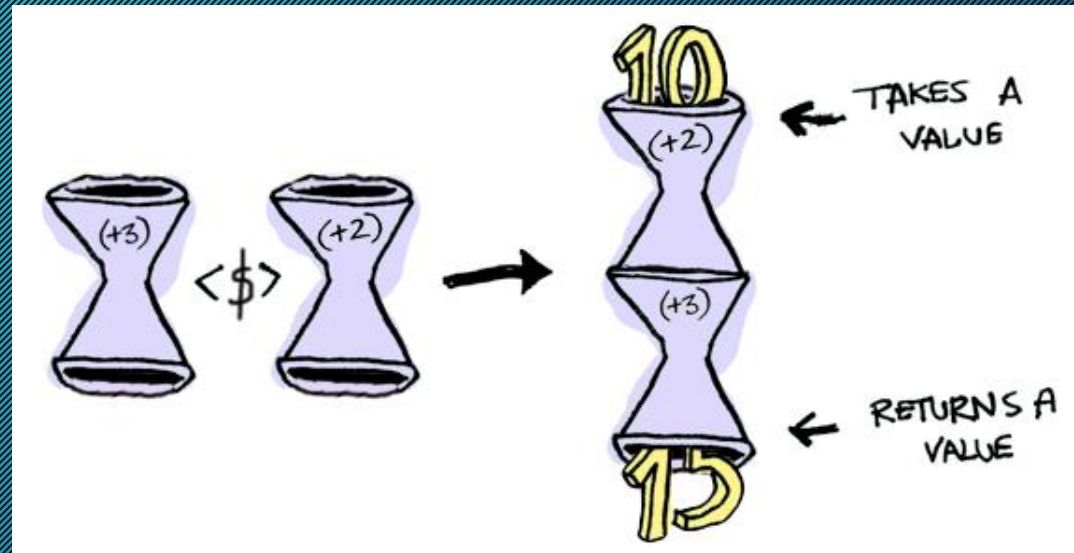
33

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



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

```
*Main> fmap (+3) (+2) 10
```



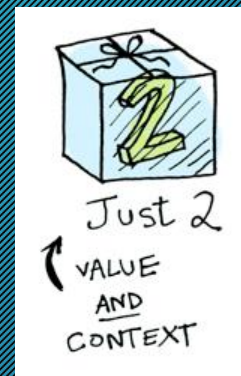


# Applicatives

34

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

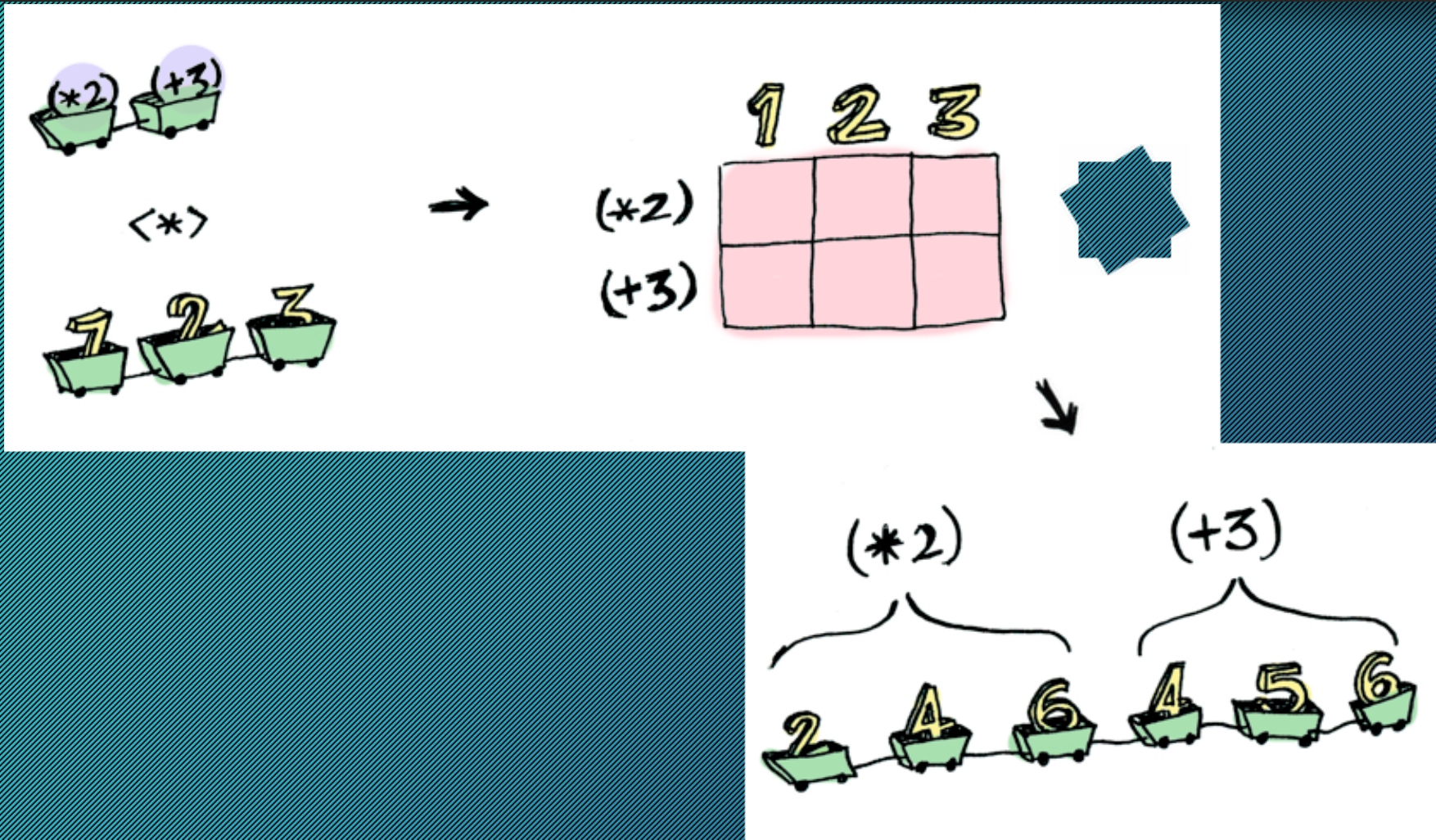
But our functions are wrapped in a context too!





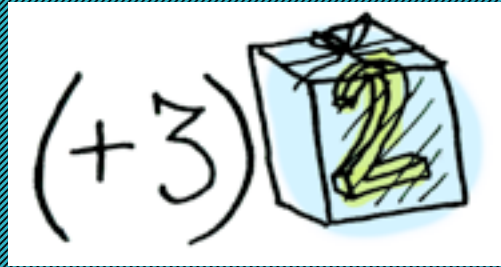
# Applicatives

35

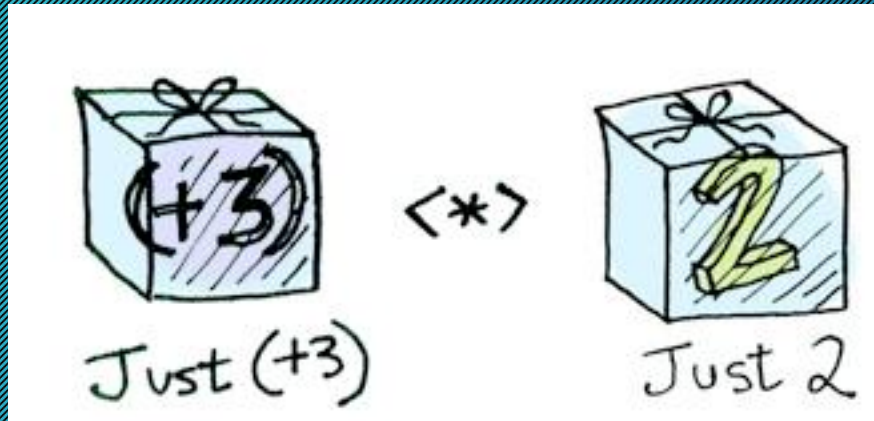




# Functors apply a function to a wrapped value:



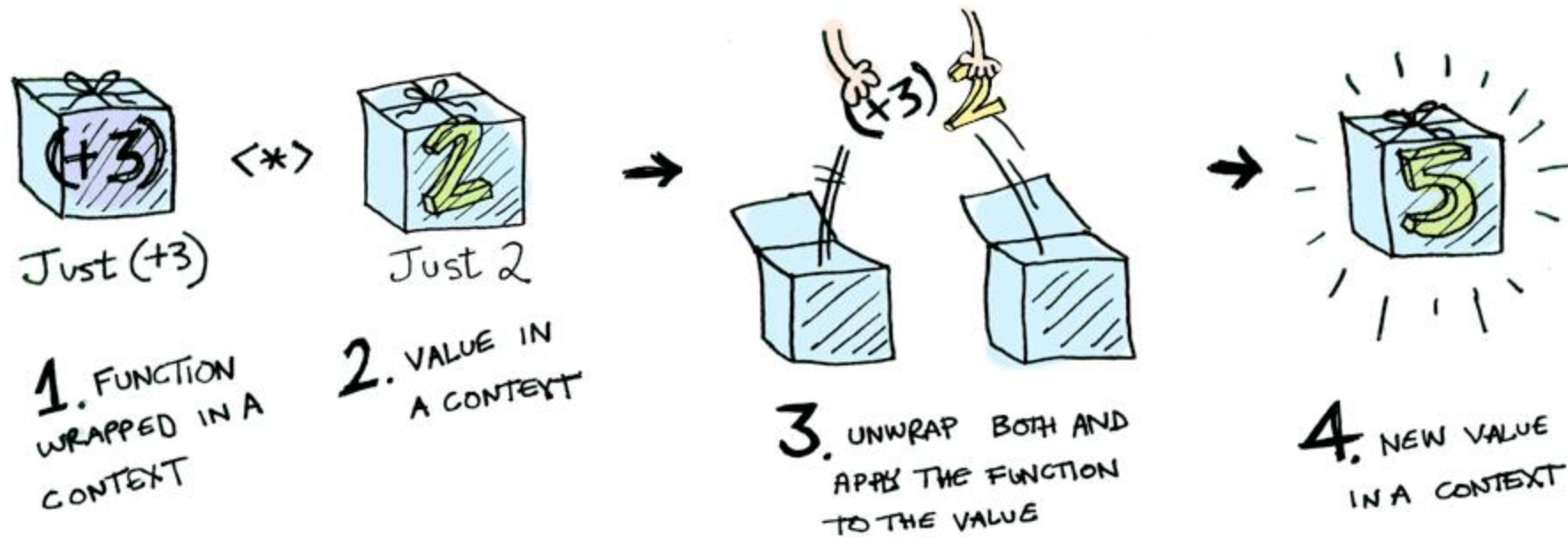
Applicatives apply a wrapped function to a wrapped value:





# Applicatives

37





# The Applicative TypeClass has two main functions

38

```
pure :: a -> f a  
(<*>) :: f (a -> b) -> f a -> f b
```

These need to be defined if we are writing an instance of Applicative, e.g.

```
instance Applicative Maybe where  
  pure x = Just x  
  Nothing <*> _ = Nothing  
  (Just f) <*> something = fmap f something
```



