Functor:

- Functor in Haskell is a kind of functional representation of different types which can be mapped over. It is a high level concept of implementing polymorphism. Types such as List, Map, Tree, etc. are instances of the Haskell Functor.
- Functor is a function which takes a function and returns another function.
- The Functor typeclass is basically for things that can be mapped over.
- A Functor is an inbuilt class with a function definition like:
  class Functor f where
  fmap :: (a -> b) -> f a -> f b → fmap takes a function and a functor and applies the function on the functor.
- Recall the map function. **map f [x, y, z] = [f x, f y, f z]**
  The map function can be applied to nothing more than a list of values (where values are of any type) whereas the fmap function can be applied to many more data types, all of which belong to the functor class (e.g. maybe, tuples, lists, etc.). Since the "list of values" data type is also a functor, because it provides an implementation for it, then fmap can be applied to is as well producing the very same result as map. In fact, map is just a fmap that works only on lists. The difference between map and fmap lies in their usage. Functor enables us to implement some more functionalists in different data types, like "Just" and "Nothing".

```
Prelude> :type map
map :: (a -> b) -> [a] -> [b]
Prelude> :type fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

- E.g. Consider the below code.

```
Prelude> map (subtract 1) [1,2,3]
[0,1,2]
Prelude> fmap (subtract 1) [1,2,3]
[0,1,2]
Prelude> map  (+7)(Just 10)

<interactive>:15:11: error:
   • Couldn't match expected type ‘[b]’
                  with actual type ‘Maybe Integer’
   • In the second argument of ‘map’, namely ‘(Just 10)’
     In the expression: map (+ 7) (Just 10)
     In an equation for ‘it’: it = map (+ 7) (Just 10)
   • Relevant bindings include it :: [b] (bound at <interactive>:15:1)
Prelude> fmap  (+7)(Just 10)
Just 17
Prelude> map  (+7) Nothing

<interactive>:17:11: error:
   • Couldn't match expected type ‘[b]’ with actual type ‘Maybe a0’
   • In the second argument of ‘map’, namely ‘Nothing’
     In the expression: map (+ 7) Nothing
     In an equation for ‘it’: it = map (+ 7) Nothing
   • Relevant bindings include it :: [b] (bound at <interactive>:17:1)
Prelude> fmap  (+7) Nothing
Nothing
```

Notice that map and fmap produce the same results on a list, but map doesn't work for types such as “Just” or “Nothing”, while fmap does.

- This is the standard library map function (**map f [x, y, z] = [f x, f y, f z]**). Here's an fmap implementation:
  fmap_List :: (a -> b) -> [] a -> [] b
  -- "[] a" means "[a]" in types.
  fmap_List f [] = []
  fmap_List f (x:xs) = f x : fmap_List f xs
- This is the definition of the Maybe type from the standard library:
  data Maybe a = Nothing | Just a
  Note: There are two perspectives for the Maybe type:
  - It's like a list of length 0 or 1.
  - It models having two possibilities: "no answer" and "here's the answer".

  Here's an fmap implementation:
  fmap_Maybe :: (a -> b) -> Maybe a -> Maybe b
  fmap_Maybe f Nothing = Nothing
  fmap_Maybe f (Just a) = Just (f a)
- This is the definition of the Either type from the standard library:
  data Either e a = Left e | Right a
  It's like Maybe, but the "no answer" case carries extra data, perhaps some kind of reason for why "no answer".
  Here's an fmap implementation:
  fmap_Either :: (a -> b) -> (Either e) a -> (Either e) b
  fmap_Either f (Left e) = Left e
  fmap_Either f (Right a) = Right (f a)
- **Note:** fmap must satisfy some axioms/laws:
  1. **Identity axiom/Functor Identity:**
     - The first functor law states that if we map the id function over a functor, the functor that we get back should be the same as the original functor. If we write that a bit more formally, it means that **fmap id = id**. So essentially, this says that if we do fmap id over a functor, it should be the same as just calling id on the functor. Id is the identity function, which just returns its parameter unmodified. It can also be written as **\x -> x**.
     - E.g.

```
Prelude> fmap id (Just 3)
Just 3
Prelude> id (Just 3)
Just 3
Prelude> fmap id [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> id [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> fmap id []
[]
Prelude> id []
[]
Prelude> fmap id Nothing
Nothing
Prelude> id Nothing
Nothing
```

  2. **fmap fusion/fmap is a homomorphism:**
     - The second law says that composing two functions and then mapping the resulting function over a functor should be the same as first mapping one

function over the functor and then mapping the other one. Formally written, that means that **fmap (f . g)** = **fmap f . fmap g**. Or to write it in another way, for any functor F, the following should hold:
fmap (f . g) F = **fmap f (fmap g F)**.

- Doing **fmap g (fmap f xs)** should get the same result as doing
fmap (\x -> g (f x)) xs
I.e.
fmap g . fmap f = **fmap (g . f)**

- **Note:** We can do fmap Just [1]. This is because types are functions.
E.g.

```
*Main> fmap Just [1]
[Just 1]
```

```
*Main> fmap Just (Just Nothing)
Just (Just Nothing)
*Main> fmap Just (Just 2)
Just (Just 2)
```

- Functor on its own does not have much basic practical use, apart from providing a common name “fmap”, but it is much more useful when combined with the Applicative and Monad methods. It also has an advanced practical use. On the other hand, Functor is extremely important in category theory.

Applicative:

- An Applicative Functor is a normal Functor with some extra features provided by the Applicative Type Class. It is found in the Control.Applicative module and to use it, we need to do **import Control.Applicative**.
- The class is defined like such:

```
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  -- And default implementations because <*> and liftA2 are equivalent.
  liftA2 f as bs = fmap f as <*> bs
  fs <*> as = liftA2 (\f a -> f a) fs as
  -- And a couple of other methods with easy default implementations.
```

pure :: a -> f a → Pure takes a value and returns a functor of that value.
(<*>) :: f (a -> b) -> f a -> f b → <*> takes a functor with a function in it and another functor and applies the function to the second functor.
liftA2 :: (a -> b -> c) -> f a -> f b -> f c → liftA2 takes a function and 2 functors and applies the function on the 2 functors.

Looking at the first line, it states the definition of the Applicative class and it also introduces a class constraint. It says that if we want to make a type constructor part of the Applicative typeclass, it has to be in Functor first. That's why if we know that if a type constructor is part of the Applicative typeclass, it's also in Functor, so we can use fmap on it.

The first method it defines is called **pure**. Its type declaration is **pure :: a -> f a**. "f" plays the role of our applicative functor instance here. pure should take a value of any type and return an applicative functor with that value inside it. We take a value and we wrap it in an applicative functor that has that value as the result inside it.

pure for [], Maybe, and Either e work as follows:

```
-- [] version
pure a = [a]

-- Maybe version
pure a = Just a

-- Either e version
pure a = Right a
```

pure plays two roles:

1. The degenerate case when you have a 0-ary function and 0 lists, kind of.
   2-ary, **liftA2 :: (t1 -> t2 -> a) -> f t1 -> f t2 -> f a**
   1-ary, **fmap :: (t1 -> a) > f t1 -> f a**
   0-ary, **pure :: a -> f a**
2. fmap can be derived from pure and <*>.
   I.e. **fmap f xs = pure f <*> xs**

The second function it defines is **<*>**. It has a type declaration of **f (a -> b) -> f a -> f b**. <*> is sort of a beefed up fmap. Whereas fmap takes a function and a functor and applies the function inside the functor, <*> takes a functor that has a function in it and another functor and sort of extracts that function from the first functor and then maps it over the second one.
Note: We can use lambda functions with <*>.
E.g.

```
*Main Control.Applicative> Just (\x -> x**2) <*> Just (5)
Just 25.0
```

The third function it defines is **liftA2**. liftA2 just applies a function between two applicatives, hiding the applicative style that we've become familiar with. The reason we're looking at it is because it clearly showcases why applicative functors are more powerful than just ordinary functors. With ordinary functors, we can just map functions over one functor. But with applicative functors, we can apply a function between several functors. It's also interesting to look at this function's type as **(a -> b -> c) -> (f a -> f b -> f c)**. When we look at it like this, we can say that liftA2 takes a normal binary function and promotes it to a function that operates on two functors.

E.g.
liftA2 (+) [1,2,3] [4,5,6]
= [1+4, 1+5, 1+6, 2+4, 2+5, 2+6, 3+4, 3+5, 3+6]
= [5,6,7,6,7,8,7,8,9]

```
Prelude Control.Applicative> liftA2 (+) [1,2,3] [4,5,6]
[5,6,7,6,7,8,7,8,9]
```

E.g.
liftA2 (-) [10,20,30] [1,2,3]
= [10-1, 10-2, 10-3, 20-1, 20-2, 20-3, 30-1, 30-2, 30-3]
= [9,8,7,19,18,17,29,28,27]

```
Prelude Control.Applicative> liftA2 (-) [10,20,30] [1,2,3]
[9,8,7,19,18,17,29,28,27]
```

Note: We can use lambda functions with liftA2.
E.g.

```
*Main Control.Applicative> liftA2 (\x y -> x**y) (Just 2) (Just 3)
Just 8.0
```

- Here is the Applicative instance implementation for Maybe.

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

First off, pure. We said earlier that it's supposed to take something and wrap it in an applicative functor. We wrote **pure = Just**, because value constructors like Just are normal functions. We could have also written **pure x = Just x**.

Next up, we have the definition for <*>. We can't extract a function out of a Nothing, because it has no function inside it. So we say that if we try to extract a function from a Nothing, the result is a Nothing.

If the first parameter is not a Nothing, but a Just with some function inside it, we say that we then want to map that function over the second parameter. This also takes care of the case where the second parameter is Nothing, because doing fmap with any function over a Nothing will return a Nothing.

So for Maybe, <*> extracts the function from the left value if it's a Just and maps it over the right value. If any of the parameters is Nothing, Nothing is the result.

E.g.
Notice that if there's Nothing on either side of the <*>, the result is nothing.

```
Prelude> Just (+3) <*> Nothing
Nothing
Prelude> Nothing <*> Just (+3)
Nothing
```

Notice that the 2 statements below give the same result.

```
Prelude> Just (+3) <*> Just 9
Just 12
Prelude> pure (+3) <*> Just 9
Just 12
```

- The Applicative methods should satisfy the following axioms:
  1. **Applicative subsumes Functor**
     fmap f xs = pure f <*> xs
  2. **Applicative left-identity**
     pure id <*> xs = xs
     -- Compare with fmap identity! fmap id xs = xs
  3. **Applicative associativity, composition**
     gs <*> (fs <*> xs) = ((pure (.) <*> gs) <*> fs) <*> xs
     = (liftA2 (.) gs fs) <*> xs
     -- Analogy: g (f x) = (g . f) x
     -- It may help to elaborate the types. Assume:
     -- xs :: f a
     -- fs :: f (a -> b)
     -- gs :: f (b -> c)
     -- (.) :: (b -> c) -> (a -> b) -> (a -> c)
     -- Try to determine the types of the subexpressions
  4. **pure fusion, pure is a homomorphism**
     pure f <*> pure x = pure (f x)
     fmap f (pure x) = pure (f x)
  5. **pure interchange, almost right-identity**
     fs <*> pure x = pure (\f -> f x) <*> fs
     = fmap (\f -> f x) fs
- The first corollary is that the Applicative axioms imply the Functor axioms given fmap f xs = pure f <*> xs.
  Functor identity is immediate from Applicative left-identity. To deduce fmap fusion:

```
fmap g (fmap f xs)                    in Applicative terms
= pure g <*> (pure f <*> xs)          associativity
= ((pure (.) <*> pure g) <*> pure f) <*> xs   pure fusion
= (pure ((.) g) <*> pure f) <*> xs    pure fusion
= pure ((.) g f) <*> xs               infix notation
= pure (g . f) <*> xs                 in Functor terms
= fmap (g . f) xs
```

- The second corollary is that
  liftA3 (\x y z -> g x (f y z)) xs ys zs
  can be done by the following two equivalent ways:
  1. **(fmap (\x y z -> g x (f y z)) xs <*> ys) <*> zs**
  2. **fmap g xs <*> (fmap f ys <*> zs)**