Maps from `Data.Map` can also be made a functor because they hold values (or not!). In the case of `Map k v`, `fmap` will map a function `v -> v'` over a map of type `Map k v` and return a map of type `Map k v'`.

Note, the `'` has no special meaning in types just like it doesn't have special meaning when naming values. It's used to denote things that are similar, only slightly changed.

Try figuring out how `Map k` is made an instance of `Functor` by yourself!

With the `Functor` typeclass, we've seen how typeclasses can represent pretty cool higher-order concepts. We've also had some more practice with partially applying types and making instances. In one of the next chapters, we'll also take a look at some laws that apply for functors.

Just one more thing! Functors should obey some laws so that they may have some properties that we can depend on and not think about too much. If we use `fmap (+1)` over the list `[1,2,3,4]`, we expect the result to be `[2,3,4,5]` and not its reverse, `[5,4,3,2]`. If we use `fmap (\a -> a)` (the identity function, which just returns its parameter) over some list, we expect to get back the same list as a result. For example, if we gave the wrong functor instance to our `Tree` type, using `fmap` over a tree where the left sub-tree of a node only has elements that are smaller than the node and the right sub-tree only has nodes that are larger than the node might produce a tree where that's not the case. We'll go over the functor laws in more detail in one of the next chapters.

Kinds and some type-foo

Type constructors take other types as parameters to eventually produce concrete types. That kind of reminds me of functions, which take values as parameters to produce values. We've seen that type constructors can be partially applied (`Either String` is a type that takes one type and produces a concrete type, like `Either String Int`), just like functions can. This is all very interesting indeed. In this section, we'll take a look at formally defining how types are applied to type constructors, just like we took a look at formally defining how values are applied to functions by using type declarations. **You don't really have to read this section to continue on your magical Haskell quest** and if you don't understand it, don't worry about it. However, getting this will give you a very thorough understanding of the type system.

So, values like `3`, `"YEAH"` or `takeWhile` (functions are also values, because we can pass them around and such) each have their own type. Types are little labels that values carry so that we can reason about the values. But types have their own little labels, called **kinds**. A kind is more or less the type of a type. This may sound a bit weird and confusing, but it's actually a really cool concept.

What are kinds and what are they good for? Well, let's examine the kind of a type by using the **`:k`** command in GHCI.

```
ghci> :k Int
Int :: *
```

A star? How quaint. What does that mean? A **`*`** means that the type is a concrete type. A concrete type is a type that doesn't take any type parameters and values can only have types that are concrete types. If I had to read **`*`** out loud (I haven't had to do that so far), I'd say *star* or just *type*.

Okay, now let's see what the kind of **`Maybe`** is.

```
ghci> :k Maybe
Maybe :: * -> *
```

The **`Maybe`** type constructor takes one concrete type (like **`Int`**) and then returns a concrete type like **`Maybe Int`**. And that's what this kind tells us. Just like **`Int -> Int`** means that a function takes an **`Int`** and returns an **`Int`**, **`* -> *`** means that the type constructor takes one concrete type and returns a concrete type. Let's apply the type parameter to **`Maybe`** and see what the kind of that type is.

```
ghci> :k Maybe Int
Maybe Int :: *
```

Just like I expected! We applied the type parameter to **`Maybe`** and got back a concrete type (that's what **`* -> *`** means. A parallel (although not equivalent, types and kinds are two different things) to this is if we do **`:t isUpper`** and **`:t isUpper 'A'`**. **`isUpper`** has a type of **`Char -> Bool`** and **`isUpper 'A'`** has a type of **`Bool`**, because its value is basically **`True`**. Both those types, however, have a kind of **`*`**.

We used **`:k`** on a type to get its kind, just like we can use **`:t`** on a value to get its type. Like we said, types are the labels of values and kinds are the labels of types and there are parallels between the two.

Let's look at another kind.

```
ghci> :k Either
Either :: * -> * -> *
```

Aha, this tells us that **`Either`** takes two concrete types as type parameters to produce a concrete type. It also looks kind of like a type declaration of a function that takes two values and returns something. Type constructors are curried (just like functions), so we can partially apply them.

```
ghci> :k Either String
```

```
Either String :: * -> *
ghci> :k Either String Int
Either String Int :: *
```

When we wanted to make **`Either`** a part of the **`Functor`** typeclass, we had to partially apply it because **`Functor`** wants types that take only one parameter while **`Either`** takes two. In other words, **`Functor`** wants types of kind `* -> *` and so we had to partially apply **`Either`** to get a type of kind `* -> *` instead of its original kind `* -> * -> *`. If we look at the definition of **`Functor`** again

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

we see that the `f` type variable is used as a type that takes one concrete type to produce a concrete type. We know it has to produce a concrete type because it's used as the type of a value in a function. And from that, we can deduce that types that want to be friends with **`Functor`** have to be of kind `* -> *`.

Now, let's do some type-foo. Take a look at this typeclass that I'm just going to make up right now:

```
class Tofu t where
    tofu :: j a -> t a j
```

Man, that looks weird. How would we make a type that could be an instance of that strange typeclass? Well, let's look at what its kind would have to be. Because `j a` is used as the type of a value that the `tofu` function takes as its parameter, `j a` has to have a kind of `*`. We assume `*` for `a` and so we can infer that `j` has to have a kind of `* -> *`. We see that `t` has to produce a concrete value too and that it takes two types. And knowing that `a` has a kind of `*` and `j` has a kind of `* -> *`, we infer that `t` has to have a kind of `* -> (* -> *) -> *`. So it takes a concrete type (`a`), a type constructor that takes one concrete type (`j`) and produces a concrete type. Wow.

OK, so let's make a type with a kind of `* -> (* -> *) -> *`. Here's one way of going about it.

```
data Frank a b  = Frank {frankField :: b a} deriving (Show)
```

How do we know this type has a kind of `* -> (* -> *) - > *`? Well, fields in ADTs are made to hold values, so they must be of kind `*`, obviously. We assume `*` for `a`, which means that `b` takes one type parameter and so its kind is `* -> *`. Now we know the kinds of both `a` and `b` and because they're parameters for **`Frank`**, we see that **`Frank`** has a kind of `* -> (* -> *) -> *` The first `*` represents `a` and the `(* -> *)` represents `b`. Let's make some **`Frank`** values and check out their types.

```
ghci> :t Frank {frankField = Just "HAHA"}
Frank {frankField = Just "HAHA"} :: Frank [Char] Maybe
ghci> :t Frank {frankField = Node 'a' EmptyTree EmptyTree}
```

```
Frank {frankField = Node 'a' EmptyTree EmptyTree} :: Frank Char Tree
ghci> :t Frank {frankField = "YES"}
Frank {frankField = "YES"} :: Frank Char []
```

Hmm. Because **`frankField`** has a type of form **`a b`**, its values must have types that are of a similar form as well. So they can be **`Just "HAHA"`**, which has a type of **`Maybe [Char]`** or it can have a value of **`['Y','E','S']`**, which has a type of **`[Char]`** (if we used our own list type for this, it would have a type of **`List Char`**). And we see that the types of the **`Frank`** values correspond with the kind for **`Frank`**. **`[Char]`** has a kind of **`*`** and **`Maybe`** has a kind of **`* -> *`**. Because in order to have a value, it has to be a concrete type and thus has to be fully applied, every value of **`Frank blah blaah`** has a kind of **`*`**.

Making **`Frank`** an instance of **`Tofu`** is pretty simple. We see that **`tofu`** takes a **`j a`** (so an example type of that form would be **`Maybe Int`**) and returns a **`t a j`**. So if we replace **`Frank`** with **`j`**, the result type would be **`Frank Int Maybe`**.

```
instance Tofu Frank where
    tofu x = Frank x
```

```
ghci> tofu (Just 'a') :: Frank Char Maybe
Frank {frankField = Just 'a'}
ghci> tofu ["HELLO"] :: Frank [Char] []
Frank {frankField = ["HELLO"]}
```

Not very useful, but we did flex our type muscles. Let's do some more type-foo. We have this data type:

```
data Barry t k p = Barry { yabba :: p, dabba :: t k }
```

And now we want to make it an instance of **`Functor`**. **`Functor`** wants types of kind **`* -> *`** but **`Barry`** doesn't look like it has that kind. What is the kind of **`Barry`**? Well, we see it takes three type parameters, so it's going to be **`something -> something -> something -> *`**. It's safe to say that **`p`** is a concrete type and thus has a kind of **`*`**. For **`k`**, we assume **`*`** and so by extension, **`t`** has a kind of **`* -> *`**. Now let's just replace those kinds with the *somethings* that we used as placeholders and we see it has a kind of **`(* -> *) -> * -> * -> *`**. Let's check that with GHCI.

```
ghci> :k Barry
Barry :: (* -> *) -> * -> * -> *
```

Ah, we were right. How satisfying. Now, to make this type a part of **`Functor`** we have to partially apply the first two type parameters so that we're left with **`* -> *`**. That means that the start of the instance declaration will be: **`instance Functor (Barry a b) where`**. If we look at **`fmap`** as if it was made specifically for **`Barry`**, it would have a type of **`fmap :: (a -> b) -> Barry c d a -> Barry c d b`**, because we just replace the **`Functor`**'s **`f`** with **`Barry c d`**. The third type parameter from **`Barry`** will have to change and we see that it's conviniently in its own field.

```
instance Functor (Barry a b) where
    fmap f (Barry {yabba = x, dabba = y}) = Barry {yabba = f x, dabba = y}
```

There we go! We just mapped the `f` over the first field.

In this section, we took a good look at how type parameters work and kind of formalized them with kinds, just like we formalized function parameters with type declarations. We saw that there are interesting parallels between functions and type constructors. They are, however, two completely different things. When working on real Haskell, you usually won't have to mess with kinds and do kind inference by hand like we did now. Usually, you just have to partially apply your own type to `* -> *` or `*` when making it an instance of one of the standard typeclasses, but it's good to know how and why that actually works. It's also interesting to see that types have little types of their own. Again, you don't really have to understand everything we did here to read on, but if you understand how kinds work, chances are that you have a very solid grasp of Haskell's type system.

Modules Table of contents Input and Output