Parameterized "container" types as effect types:

- Think of [], Maybe, Either e as types of effects or effectful programs, not always as types of data structures.
- "Effect" is very broad and usually not given a precise definition, but if you know the phrase "function with a side-effect", that's the idea. In the Haskell culture, it refers to things a mathematical function cannot do, e.g. no answer, multiple answers, accessing state variables, performing I/O—the latter two lead to getting different answers at different times.
- **f :: Int -> String** in Haskell means that **f 4 :: String** is the same string every time, without effect. If some specific kind of effect is desired, we make a parameterized type E to Maybe or [] to represent it and change types to **f :: Int -> E String**, so **f 4 :: E String** does not have to give the same string every time, instead just the same "effectful action" every time.
- It suffices to focus on "E String" (so without "Int ->") and think of it as the type of effectful programs that give string answers. E.g., if m1, m2 :: Maybe String, think of them as two programs that give string answers (the effect is that it could fail).
- In this frame of mind, the methods of Functor, Applicative, and later Monad are connectives—combining basic effectful programs into complex ones. E.g. **liftA2 (++) m1 m2** combines m1 and m2.
- Consider "Maybe":
    - **foo :: Maybe Int** now means a program that may succeed and return an Int (conveyed by Just), or may fail (conveyed by Nothing).
    - Suppose **f :: Int -> String**.
    - **fmap f foo :: Maybe String** now means a program that runs foo, but converts the answer, if any, using f.
    - **pure 4 :: Maybe Int** now means a program that succeeds and returns 4. Note that it avoids using Maybe's effect of failure.
    - Suppose **bar :: Maybe Int**.
    - **liftA2 (+) foo bar :: Maybe Int** now means a composite program that runs foo to try to obtain a number. If running foo is successful, it runs bar. If that is successful too, the overall answer is the sum.
    - E.g. I have a recipMay function for reciprocals, but to better handle division-by-zero, I use Maybe to convey successes and failures. I then use it to write an addRecip function to add two reciprocals, again involving Maybe in anticipation of failure. This is my first version:

```
recipMay :: Double -> Maybe Double
recipMay a | a == 0 = Nothing
           | otherwise = Just (1 / a)    -- or: pure (1 / a)
```

```
addRecipV1 x y =
  case recipMay x of
    Nothing -> Nothing
    Just x_recip -> case recipMay y of
      Nothing -> Nothing
      Just y_recip -> Just (x_recip + y_recip)
```

This is my second version. It uses the new way of thinking:

```
recipMay :: Double -> Maybe Double
recipMay a | a == 0 = Nothing
           | otherwise = Just (1 / a)    -- or: pure (1 / a)

addRecipV2 :: Double -> Double -> Maybe Double
addRecipV2 x y = liftA2 (+) (recipMay x) (recipMay y)
```

Monads:

- Monads are just beefed up applicative functors.
- A monad is a way to structure computations in terms of values and sequences of computations using those values. Monads allow the programmer to build up computations using sequential building blocks, which can themselves be sequences of computations. The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required.
- It is useful to think of a monad as a strategy for combining computations into more complex computations.
- Monads chain operations in some specific, useful way.
- Monads apply a regular function to a wrapped value and return a wrapped value. This is similar to what a functor does. The difference between a monad and a functor is that with monads, the functions aren't expecting a wrapped value.
- E.g. Consider the code and output below:

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

```
*Main> half 4
Just 2
*Main> half 5
Nothing
*Main> half 10
Just 5
```

What if we wanted to pass in (Just 10) to half?

```
*Main> half (Just 10)

<interactive>:91:1: error:
  • Non type-variable argument in the constraint: Integral (Maybe a)
    (Use FlexibleContexts to permit this)
  • When checking the inferred type
      it :: forall a. (Integral (Maybe a), Num a) => Maybe (Maybe a)
```

It gives an error.

However, if we do (Just 10) >>= half, we get back Just 5, as shown below.

```
*Main> (Just 10) >>= half
Just 5
```

In this example, the function, half, isn't expecting a wrapped value of (Just 10). We did half x = …, not half (Just x) = …. However, by using >>=, we were able to apply (Just 10) to the half function. Here's what >>= is doing behind the scenes:

- When we do "Just 10 >>= …", the >>= takes the 10, only.
- Hence, when we did (Just 10) >>= half, it was like doing half 10.

- We can also chain monads.
  E.g.

```
*Main> (Just 8) >>= half >>= half
Just 2
*Main> (Just 4) >>= half >>= half
Just 1
*Main> (Just 4) >>= half >>= half >>= half
Nothing
```

- The monad class is defined like this:

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

  (>>=) :: m a -> (a -> m b) -> m b
  x >> y = x >>= \_ -> y

  fail :: String -> m a
  fail msg = error msg
```

(>>=) :: m a -> (a -> m b) -> m b → >>= takes in a monadic value and a function that takes in a regular value but outputs a monadic value and applies the function on the monadic value.
E.g. Suppose you have something like **Just x >>= \b -> …**, here "b" is not wrapped in "Just". This is because the lambda function takes a regular value.

The first function that the Monad type class defines is **return**. It's the same as pure, only with a different name. Its type is **(Monad m) => a → m a**. It takes a value and puts it in a minimal default context that still holds that value. In other words, it takes something and wraps it in a monad. It always does the same thing as the pure function from the Applicative type class.
Note: return is nothing like the return that's in most other languages. It doesn't end function execution or anything, it just takes a normal value and puts it in a context.
It is here for historical reasons (because Applicative is more recent). Do not think of return in terms of control flow. It does not exit anything.

The next function is **>>=**, or **bind**. It's like function application, only instead of taking a normal value and feeding it to a normal function, it takes a monadic value (a value with a context) and feeds it to a function that takes a normal value but returns a monadic value.

Next up, we have **>>**. It's default implementation is **foo >> bar = foo >>= _ -> bar**. It comes in handy when you don't need foo's answer, only its effect. We call this operator "then". We use this function when we want to perform actions in a certain order, but don't care what the result of one is. Consider the example below:

```
Prelude> print "x" >>= \_ -> print "y"
"x"
"y"
Prelude> print "x" >> print "y"
"x"
"y"
```

The final function of the Monad type class is **fail**. We never use it explicitly in our code. Instead, it's used by Haskell to enable failure in a special syntactic construct for monads.

- The Monad methods satisfies these axioms:
    1. **Left Identity Law:** This law states that if we take a value, put it in a default context with return and then feed it to a function by using >>=, it's the same as just taking the value and applying the function to it. To put it formally:
       return x >>= k = k x
       E.g.

       ```
       Prelude> (\x -> Just(x+1)) 3
       Just 4
       Prelude> return 3 >>= (\x -> Just(x+1))
       Just 4
       ```

       Notice that both statements get the same result.
    2. **Right Identity Law:** This law states that if we have a monadic value and we use >>= to feed it to return, the result is our original monadic value. Formally:
       m >>= return = m
       E.g.

       ```
       Prelude> [1,2,3,4] >>= (\x -> return x)
       [1,2,3,4]
       Prelude> Just 3 >>= (\x -> return x)
       Just 3
       ```

    3. **Associativity:** This law says that when we have a chain of monadic function applications with >>=, it shouldn't matter how they're nested. Formally written: Doing **(m >>= f) >>= g** is just like doing **m >>= (\x -> f x >>= g)**

       Associative laws justify re-grouping. This is important for many refactoring and implementing long chains by recursion.
- Monad subsumes Applicative and Functor. From return and >>= we can get the methods of Applicative and Functor:
  fmap f xs = xs >>= (\x -> return (f x))
  liftA2 op xs ys = xs >>= (\x -> ys >>= (\y -> return (op x y)))

- Monads need to use both applicative and functor.
  E.g. Consider the code and output below.

```
data Maybe2 a = Nothing2 | Just2 a deriving Show

instance Monad Maybe2 where
    -- >>= :: m a -> (a -> m b) -> m b
    Nothing2 >>= f = Nothing2
    Just2 a >>= f = f a
```

```
*Main> :reload

monad_maybe.hs:3:10: error:
    • No instance for (Applicative Maybe2)
        arising from the superclasses of an instance declaration
    • In the instance declaration for 'Monad Maybe2'
  |
3 | instance Monad Maybe2 where   |          ^^^^^^^^^^
[1 of 1] Compiling Main             ( monad_maybe.hs, interpreted )
Failed, no modules loaded.
```

Now, if I add the applicative and functor instances, then I don't get that compilation error.

```
data Maybe2 a = Nothing2 | Just2 a deriving Show

instance Functor Maybe2 where
    -- fmap :: (a -> b) -> fa -> fb
    fmap f Nothing2 = Nothing2
    fmap f (Just2 a) = Just2 (f a)

instance Applicative Maybe2 where
    -- pure :: a -> f a
    -- <*> :: f(a -> b) -> f a -> f b
    pure = Just2
    Nothing2 <*> _ = Nothing2
    _ <*> Nothing2 = Nothing2
    Just2 f <*> Just2 a = Just2 (f a)

instance Monad Maybe2 where
    -- >>= :: m a -> (a -> m b) -> m b
    Nothing2 >>= f = Nothing2
    Just2 a >>= f = f a
```

```
Prelude> :reload
[1 of 1] Compiling Main             ( monad_maybe.hs, interpreted )
Ok, one module loaded.
```

- Examples of Maybe2:

```haskell
data Maybe2 a = Nothing2 | Just2 a deriving Show

instance Functor Maybe2 where
    -- fmap :: (a -> b) -> fa -> fb
    fmap f Nothing2 = Nothing2
    fmap f (Just2 a) = Just2 (f a)

instance Applicative Maybe2 where
    -- pure :: a -> f a
    -- <*> :: f(a -> b) -> f a -> f b
    pure = Just2
    Nothing2 <*> _ = Nothing2
    _ <*> Nothing2 = Nothing2
    Just2 f <*> Just2 a = Just2 (f a)

instance Monad Maybe2 where
    -- >>= :: m a -> (a -> m b) -> m b
    Nothing2 >>= f = Nothing2
    Just2 a >>= f = f a


half x = if even x
    then Just2 (x `div` 2)
    else Nothing2
```

```
*Main> (Just2 10) >>= half
Just2 5
*Main> Nothing2 >>= half
Nothing2
*Main> (Just2 100) >>= half >>= half >>= half
Nothing2
*Main> (Just2 100) >>= half >>= half
Just2 25
```