```
data TypeName = ValueConstructorA FieldA1 ... FieldAN | ValueConstructorB ...
```

Functions that ultimately return a value of a data type. So the "fields" of a value constructor are really parameters.

They are **not** types.

Value constructors with no parameters.

Add `deriving (OtherClass)` to the end of the `data` declaration.

Use the same name for the data type and the value constructor.

```
module MyModule
( DataConstructor1(..)
, ...
, DataConstructorN
) where ...
```

The `..` indicates all value constructors of the type constructor should be exported.
Otherwise you can comma separate the ones you want to export, or leave off the parens altogether to not export any value constructors.

Instances of the type can be created only through auxiliary functions.

`Data.Map` can be constructed only through auxiliary functions like `fromList`.

A way of creating value constructors with automatic meaningfully named accessors.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving (Show)
```

Named parameters. Users can invoke the value constructor using curlies and comma-separated named values.

Like Scala's `Option`:

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

Use them when your type is a container of some kind. Otherwise don't, since your functions on the type will make strong assumptions about the types filled in.

Constraints should follow from functions that assume the constraint.
If you put the constraint on the data declaration you will have to put the constraint on **all** functions, even when the constraint is irrelevant to that particular function.

All the field types must also be of the `Eq` typeclass.

Ordering the value constructors least to greatest, in the order defined.

nullary

They can be used in list ranges.

Haskell's term for Scala's type aliases.
The syntax is the same as Scala's:

```
type Name = Type
```

Haskell's can be parameterized:

```
type AssocList k v = [(k,v)]
```

Haskell's can be "partially applied" using either of these notations:

```
type AssocListInt = AssocList Int
type AssocListInt v = AssocList Int v
```

A type that is fully applied, with no unbound type parameters.

Approximately:

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

Errors use `Left`, successful result uses `Right`.

Its used for functions that can fail in multiple ways. `Maybe`'s `Nothing` would not inform as to which way it failed.

It indicates which way an operator associates and how tightly.

`fixity num op`

Where `fixity` is one of `infix`, `infixr`, `infixl`, and `num` is the tightness of binding, and `op` is the operator in question. Larger numbers indicate tighter bindings.

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

```
class ClassName typevar1 ... typevarn where
    funcDefOrDeclaration1
    ...
    funcDefOrDeclarationN
```

Function declarations are simply type signatures like `ghci` would put out in response to `:t`.

```
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

The typeclass might implement functions in terms of one another, requiring a minimal complete definition to define one to bootstrap the others.

Adding a class constraint to a `class` declaration.

```
class (Eq a) => Num a where
    ...
```

`:info Name` in `ghci`.

It's for things that can be mapped over, like lists.

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

More or less, the type of a type.

```
:k Type
```