Haskell
Making Our Own Types and Typeclasses

http://igm.univ-mlv.fr/~vialette/?section=teaching

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

November 18, 2018

Making Our Own Types and Typeclasses

Algebraic data types intro

So far, we've run into a lot of data types: `Bool`, `Int`, `Char`, `Maybe`, etc. But how do we make our own?

One way is to use the `data` keyword to define a type.

Let's see how the `Bool` type is defined in the standard library.

```
data Bool = False | True
```

`data` means that we're defining a new data type.

Making Our Own Types and Typeclasses

Algebraic data types intro

```
data Bool = False | True
```

The part before the `=` denotes the type, which is `Bool`.

The parts after the `=` are **value constructors**. They specify the different values that this type can have.

The `|` is read as or. So we can read this as: *the `Bool` type can have a value of `True` or `False`*.

Both the type name and the value constructors have to be capital cased.

Making Our Own Types and Typeclasses

Algebraic data types intro

We can think of the `Int` type as being defined like this:

```
data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1 | 2 | ..
```

The first and last value constructors are the minimum and maximum possible values of `Int`.

It's not actually defined like this, the ellipses are here because we omitted a heapload of numbers, so this is just for illustrative purposes.

Shape

let's think about how we would represent a shape in Haskell.

Shape

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
```

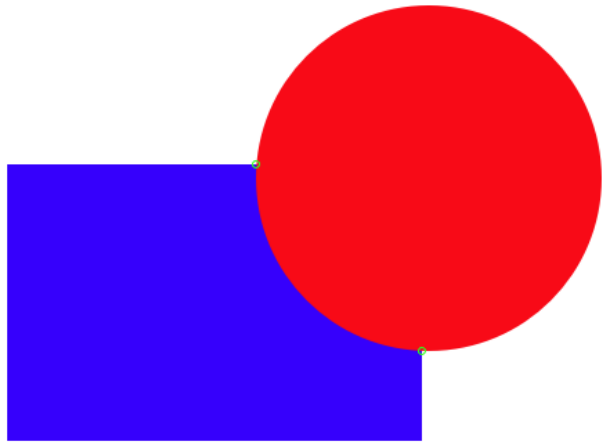
The `Circle` value constructor has three fields, which take floats: the first two fields are the coordinates of its center, the third one its radius.

The `Rectangle` value constructor has four fields which accept floats: the first two are the coordinates to its upper left corner and the second two are coordinates to its lower right one.

Shape

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
ghci>
```

Value constructors are functions like everything else.



Shape

Let's make a function that takes a shape and returns its surface:

```
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs (x2 - x1)) * (abs (y2 - y
```

```
ghci> surface $ Circle 10 20 10
314.15927
ghci> surface $ Rectangle 0 0 100 100
10000.0
ghci> let c = Circle 10 20 10
ghci> surface c
314.15927
ghci> let r = Rectangle 0 0 100 100
ghci> surface r
10000.0
ghci>
```

Shape

Haskell doesn't know how to display our data type as a string (yet):

```
ghci> Circle 10 20 30

<interactive>:27:1:
  No instance for (Show Shape) arising from a use of 'print'
  In a stmt of an interactive GHCi command: print it
ghci> Rectangle 0 0 100 100

<interactive>:28:1:
  No instance for (Show Shape) arising from a use of 'print'
  In a stmt of an interactive GHCi command: print it
ghci>
```

Shape

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
           deriving (Show)
```

```
ghci> Circle 10 20 30
Circle 10.0 20.0 30.0
ghci> Rectangle 0 0 100 100
Rectangle 0.0 0.0 100.0 100.0
ghci>
```

Shape

Value constructors are functions, so we can map them and partially apply them and everything. If we want a list of concentric circles with different radii, we can do this.

```
ghci> map (Circle 10 20) [4,5]
[Circle 10.0 20.0 4.0,Circle 10.0 20.0 5.0]
ghci>
```

If we want a list of rectangles with different lower right corners, we can do this.

```
ghci> map (\(x,y) -> Rectangle 0 0 x y) [(10,10),(11,11)]
[Rectangle 0.0 0.0 10.0 10.0,Rectangle 0.0 0.0 11.0 11.0]
ghci>
```

Shape

Our data type is good, although it could be better. Let's make an intermediate data type that defines a point in two-dimensional space.

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float
           | Rectangle Point Point deriving (Show)
```

Notice that when defining a point, we used the same name for the data type and the value constructor. This has no special meaning, although it's common to use the same name as the type if there's only one value constructor.

So now the `Circle` has two fields, one is of type `Point` and the other of type `Float`. Same goes for the rectangle.

Shape

We have to adjust our surface function to reflect these changes.

```
surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) = lx * ly
  where
    lx = abs $ x2 - x1
    ly = abs $ y2 - y1
```

```
ghci> surface (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> surface (Circle (Point 0 0) 24)
1809.5574
ghci>
```

Shape

How about a function that nudges a shape? It takes a shape, the amount to move it on the x axis and the amount to move it on the y axis and then returns a new shape that has the same dimensions, only it's located somewhere else.

```
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b =
  Rectangle p1 p2
    where
      p1 = Point (x1+a) (y1+b)
      p2 = Point (x2+a) (y2+b)
```

```
ghci> nudge (Circle (Point 34 34) 10) 5 10
Circle (Point 39.0 44.0) 10.0
ghci>
```

Shape

If we don't want to deal directly with points, we can make some auxilliary functions that create shapes of some size at the zero coordinates and then nudge those.

```
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r

baseRectangle :: Float -> Float -> Shape
baseRectangle w h = Rectangle (Point 0 0) (Point w h)
```

```
ghci> nudge (baseRectangle 40 100) 60 23
Rectangle (Point 60.0 23.0) (Point 100.0 123.0)
ghci>
```

Shape

You can, of course, export your data types in your modules.

To do that, just write your type along with the functions you are exporting and then add some parentheses and in them specify the value constructors that you want to export for it, separated by commas.

If you want to export all the value constructors for a given type, just write . . .

Shape

```
module Shapes
( Point(..)
, Shape(..)
, surface
, nudge
, baseCircle
, baseRectangle
) where
```

By doing `Shape(..)`, we exported all the value constructors for `Shape`, so that means that whoever imports our module can make shapes by using the `Rectangle` and `Circle` value constructors.

It's the same as writing `Shape (Rectangle, Circle)`.

Shape

We could also opt not to export any value constructors for `Shape` by just writing `Shape` in the export statement.

That way, someone importing our module could only make shapes by using the auxilliary functions `baseCircle` and `baseRectangle`.

Not exporting the value constructors of a data types makes them more abstract in such a way that we hide their implementation. Also, whoever uses our module can't pattern match against the value constructors.

Data.Map

`Data.Map` uses that approach. You can't create a map by doing `Map.Map [(1,2),(3,4)]` because it doesn't export that value constructor.

However, you can make a mapping by using one of the auxilliary functions like `Map.fromList`.

Remember, value constructors are just functions that take the fields as parameters and return a value of some type (like `Shape`) as a result. So when we choose not to export them, we just prevent the person importing our module from using those functions, but if some other functions that are exported return a type, we can use them to make values of our custom data types.

Record syntax

Record syntax

The info that we want to store about a person is: first name, last name, age, height and phone number.

```
data Person = Person String String Int Float String
              deriving (Show)
```

```
Prelude> let guy = Person "Jo" "Dalton" 43 184.0
Prelude> guy
Person "Jo" "Dalton" 43 184.0
Prelude>
```

Record syntax

```haskell
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height
```

Record syntax

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

Record syntax

The main benefit of this is that it creates functions that lookup fields in the data type.

By using record syntax to create this data type, Haskell automatically made these functions: `firstName`, `lastName`, `age` and `height`.

```
ghci: :t Person
Person :: String -> String -> Int -> Float -> Person
ghci: :t firstName
firstName :: Person -> String
ghci: :t lastName
lastName :: Person -> String
ghci: :t age
age :: Person -> Int
ghci: :t height
height :: Person -> Float
```

Record syntax

When we derive `Show` for the type, it displays it differently if we use record syntax to define and instantiate the type.

```
data Car = Car String String Int deriving (Show)
```

```
ghci: Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
ghci:
```

```
data Car = Car {company::String, model::String, year::Int}
  deriving (Show)
```

```
ghci: Car "Ford" "Mustang" 1967
Car {company = "Ford", model = "Mustang", year = 1967}
ghci:
```

Record syntax

```haskell
origin :: Car -> String
origin (Car {company="Ford",    model=_, year=_}) = "USA"
origin (Car {company="Renault", model=_, year=_}) = "France
origin (Car {company="BMW",     model=_, year=_}) = "Germar
origin _                                  = "???"
```

```
ghci: origin (Car "Ford" "Mustang" 1967)
"USA"
ghci: origin (Car "Renault" "Espace" 2016)
"France"
ghci: origin (Car "BMW" "Z3" 2000)
"Germany"
ghci: origin (Car "Opel" "Corsa" 1990)
"???"
ghci:
```

Type parameters

Type parameters

A value constructor can take some values parameters and then produce a new value.

For instance, the `Car` constructor takes three values and produces a car value.

In a similar manner, type constructors can take types as parameters to produce new types.

This might sound a bit too meta at first, but it's not that complicated. If you're familiar with templates in C++, you'll see some parallels.

Type parameters

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

The `a` here is the type parameter.

And because there's a type parameter involved, we call `Maybe` a **type constructor**.

Depending on what we want this data type to hold when it's not `Nothing`, this type constructor can end up producing a type of `Maybe Int`, `Maybe Car`, `Maybe String`, etc.

No value can have a type of just `Maybe`, because that's not a type per se, it's a type constructor. In order for this to be a real type that a value can be part of, it has to have all its type parameters filled up.

Type parameters

```
ghci: :t Just
Just :: a -> Maybe a
ghci: :t Nothing
Nothing :: Maybe a
ghci: :t Just 1
Just 1 :: Num a => Maybe a
ghci: :t Just 'a'
Just 'a' :: Maybe Char
ghci: :t Just "ab"
Just "ab" :: Maybe [Char]
ghci: :t Just Nothing
Just Nothing :: Maybe (Maybe a)
ghci: :t Just (Just ["ab", "cd"])
Just (Just ["ab", "cd"]) :: Maybe (Maybe [[Char]])
ghci:
```

Type parameters

Notice that the type of `Nothing` is `Maybe a`.

Its type is **polymorphic**. If some function requires a `Maybe Int` as a parameter, we can give it a `Nothing`, because a `Nothing` doesn't contain a value anyway and so it doesn't matter.

The `Maybe a` type can act like a `Maybe Int` if it has to, just like `5` can act like an `Int` or a `Double`.

Similarly, the type of the empty list is `[a]`. An empty list can act like a list of anything. That's why we can do `[1,2,3] ++ []` and `["ha","ha","ha"] ++ []`.

Parameterized type

Another example of a parameterized type that we've already met is `Map k v` from `Data.Map`.

The `k` is the type of the keys in a map and the `v` is the type of the values.

This is a good example of where type parameters are very useful. Having maps parameterized enables us to have mappings from any type to any other type, as long as the type of the key is part of the `Ord` typeclass.

If we were defining a mapping type, we could add a typeclass constraint in the data declaration:

```
data (Ord k) => Map k v = ...
```

Parameterized type

However, it's a very strong convention in Haskell to never add typeclass constraints in data declarations.

Why? Well, because we don't benefit a lot, but we end up writing more class constraints, even when we don't need them.

If we put or don't put the `Ord k` constraint in the data declaration for `Map k v`, we're going to have to put the constraint into functions that assume the keys in a map can be ordered.

But if we don't put the constraint in the data declaration, we don't have to put `(Ord k) =>` in the type declarations of functions that don't care whether the keys can be ordered or not.

Parameterized type

Let's implement a 3D vector type and add some operations for it.

```
data Vector a = Vector a a a deriving (Show)

vplus :: (Num t) => Vector t -> Vector t -> Vector t
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)

vectMult :: (Num t) => Vector t -> t -> Vector t
(Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)

scalarMult :: (Num t) => Vector t -> Vector t -> t
(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n
```

Parameterized type

```
ghci: :t Vector
Vector :: a -> a -> a -> Vector a
ghci: let v = Vector 1 2 3
ghci: v `vplus` v
Vector 2 4 6
ghci: v `vectMult` 10
Vector 10 20 30
ghci: v `scalarMult` v
14
ghci:
```

Parameterized type

```
ghci: let v = Vector 1 2 3.0
ghci: v
Vector 1.0 2.0 3.0
ghci: :t v
v :: Fractional a => Vector a
ghci: let v' = Vector 1 2 3 `vplus` Vector 1.0 2.0 3.0
ghci: v'
Vector 2.0 4.0 6.0
ghci: :t v'
v' :: Fractional t => Vector t
ghci: Vector 1 2 '3'
<interactive>:113:8:
    No instance for (Num Char) arising from the literal '1'
    In the first argument of 'Vector', namely '1'
    In the expression: Vector 1 2 '3'
    In an equation for 'it': it = Vector 1 2 '3'
ghci:
```

Parameterized type

```
ghci: Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
ghci: Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
Vector 12 9 19
ghci: Vector (Vector 1 2 3) (Vector 4 5 6) (Vector 7 8 9)
Vector (Vector 1 2 3) (Vector 4 5 6) (Vector 7 8 9)
ghci:
```

Derived instances

Derived instances

In the Typeclasses 101 section, we explained the basics of typeclasses.

We explained that a typeclass is a sort of an interface that defines some behavior.

A type can be made an instance of a typeclass if it supports that behavior.

Derived instances

the `Int` type is an instance of the `Eq` typeclass because the `Eq` typeclass defines behavior for stuff that can be equated.

And because integers can be equated, `Int` is a part of the `Eq` typeclass.

The real usefulness comes with the functions that act as the interface for `Eq`, namely `==` and `/=`.

If a type is a part of the `Eq` typeclass, we can use the `==` functions with values of that type.

That's why expressions like `4 == 4` and `"foo" /= "bar"` typecheck.

Derived instances

Consider this data type:

```
data Person = Person { name :: String
                     , age :: Int
                     } deriving (Eq)
```

When we derive the `Eq` instance for a type and then try to compare two values of that type with `==` or `/=`, Haskell will see if the value constructors match (there's only one value constructor here though) and then it will check if all the data contained inside matches by testing each pair of fields with `==`.

There's only one catch though, the types of all the fields also have to be part of the `Eq` typeclass. But since both `String` and `Int` are, we're OK.

Derived instances

```
ghci: let mike40 = Person {name = "Michael", age = 40}
ghci: let mike43 = Person {name = "Michael", age = 43}
ghci: let adam40 = Person {name = "Adam", age = 40}
ghci: let adam43 = Person {name = "Adam", age = 43}
ghci: mike40 == mike43
False
ghci: adam40 == adam43
False
ghci: mike40 == adam40
False
ghci: mike43 == mike43
True
ghci: mike40 == Person {name = "Michael", age = 40}
True
ghci:
```

Derived instances

Of course, since `Person` is now in `Eq`, we can use it as the a for all functions that have a class constraint of `Eq a` in their type signature, such as elem.

```
ghci: let ps = [mike40, mike43, adam40, adam43]
ghci: mike40 `elem` ps
True
ghci: Person {name = "Michael", age = 43} `elem` ps
True
ghci: import Data.List
ghci Data.List: let f p = p == adam40
ghci Data.List: let ps' = filter f ps
ghci Data.List: length ps'
1
ghci Data.List:
```

`Read` and `Show` typeclasses

The `Show` and `Read` typeclasses are for things that can be converted to or from strings, respectively.

Like with `Eq`, if a type's constructors have fields, their type has to be a part of `Show` or `Read` if we want to make our type an instance of them.

Let's make our `Person` data type a part of `Show` and `Read` as well.

```
data Person = Person { name :: String
                     , age :: Int
                     } deriving (Eq, Show, Read)
```

Read and Show typeclasses

Now we can print a person out to the terminal.

```
ghci: adam40
Person {name = "Adam", age = 40}
ghci: "adam40 is " ++ show adam40
"adam40 is Person {name = \"Adam\", age = 40}"
ghci: let f p = p == adam40
ghci: let ps = [mike40, mike43, adam40, adam43]
ghci: filter f ps
[Person {name = "Adam", age = 40}]
ghci: filter (\p -> p == adam40)  ps
[Person {name = "Adam", age = 40}]
ghci: filter (== adam40)  ps
[Person {name = "Adam", age = 40}]
ghci:
```

Read and Show typeclasses

Read is pretty much the inverse typeclass of Show. Show is for converting values of our a type to a string, Read is for converting strings to values of our type.

```
ghci: read "Person {name = \"Bob\", age = 20}" :: Person
Person {name = "Bob", age = 20}
ghci: let bob20 = read "Person {name = \"Bob\", age = 20}" :: Person
ghci: bob20
Person {name = "Bob", age = 20}
ghci: bob20 == read "Person {name = \"Bob\", age = 20}"
True
ghci:
```

`Ord` typeclass

We can derive instances for the `Ord` type class, which is for types that have values that can be ordered.

```
data Bool = False | True deriving (Eq, Ord)
```

Because the `False` value constructor is specified first and the `True` value constructor is specified after it, we can consider `True` as greater than `False`.

```
Prelude: False' `compare` True'
LT
Prelude: False' < True'
True
Prelude: False' > True'
False
Prelude: True' < False'
False
Prelude:
```

`Ord` typeclass

In the `Maybe` a data type, the `Nothing` value constructor is specified before the `Just` value constructor, so a value of `Nothing` is always smaller than a value of `Just` something

```
Prelude: Nothing < Just 100
True
Prelude: Nothing > Just (-49999)
False
Prelude: Just 3 `compare` Just 2
GT
Prelude: Just 100 > Just 50
True
Prelude: Nothing `compare` Nothing
EQ
Prelude: Just 5 `compare` Just 5
EQ
Prelude:
```

`Ord` typeclass

But

```
Prelude: Just (== 1) `compare` Just (== 1)

<interactive>:9:7:
    No instance for (Eq a0) arising from a use of '=='
    The type variable 'a0' is ambiguous
    Note: there are several potential instances:
      instance Eq a => Eq (GHC.Real.Ratio a) -- Defined in 'GHC.Real'
      instance Eq Integer -- Defined in 'integer-gmp:GHC.Integer.Type'
      instance Eq a => Eq (Maybe a) -- Defined in 'Data.Maybe'
      ...
```

`Bounded` and `Enum` typeclasses

We can easily use algebraic data types to make enumerations and the `Bounded` and `Enum` typeclasses help us with that.

```
data Day = Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
         | Sunday
```

`Bounded` and `Enum` typeclasses

Because all the value constructors are nullary (take no parameters, i.e. fields), we can make it part of the `Enum` typeclass.

The `Enum` typeclass is for things that have predecessors and successors.

We can also make it part of the `Bounded` typeclass, which is for things that have a lowest possible value and highest possible value.

And while we're at it, let's also make it an instance of all the other derivable typeclasses and see what we can do with it.

Bounded and Enum typeclasses

```
data Day = Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
         | Sunday
  deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

Bounded and Enum typeclasses

```
ghci: Tuesday
Tuesday
ghci: show Tuesday
"Tuesday"
ghci: succ Tuesday
Wednesday
ghci: pred Tuesday
Monday
ghci: pred $ pred Tuesday
*** Exception: pred{Day}: tried to take 'pred'
    of first tag in enumeration
ghci: succ $ succ Saturday
*** Exception: succ{Day}: tried to take 'succ'
    of last tag in enumeration
ghci: read "Saturday" :: Day
Saturday
ghci:
```

Bounded and Enum typeclasses

```
Main: Monday == Sunday
False
ghci: Monday < Sunday
True
ghci: Sunday `compare` Monday
GT
ghci: minBound :: Day
Monday
ghci: maxBound :: Day
Sunday
ghci: [minBound .. maxBound] :: [Day]
[Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday]
ghci: [Monday .. Friday]
[Monday,Tuesday,Wednesday,Thursday,Friday]
ghci:
```

Type synonyms

Type synonyms

We mentioned that when writing types, the `[Char]` and `String` types are equivalent and interchangeable.

That's implemented with **type synonyms**.

Type synonyms don't really do anything per se, they're just about giving some types different names so that they make more sense to someone reading our code and documentation.

Here's how the standard library defines String as a synonym for [Char].

```
type String = [Char]
```

Type synonyms

The `type` keyword might be misleading to some, because we're not actually making anything new (we did that with the `data` keyword), but we're just making a synonym for an already existing type.

If we make a function that converts a string to uppercase and call it `toUpperString` or something, we can give it a type declaration of

```
toUpperString :: [Char] -> [Char]
```

or

```
toUpperString :: String -> String.
```

Both of these are essentially the same, only the latter is nicer to read.

Type synonyms

Type synonyms can also be parameterized. If we want a type that represents an association list type but still want it to be general so it can use any type as the keys and values, we can do this:

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

Now, a function that gets the value by a key in an association list can have a type of
`(Eq k) => k -> AssocList k v -> Maybe v`.

`AssocList` is a type constructor that takes two types and produces a concrete type, like `AssocList Int String`, for instance.

Type synonyms

Just like we can partially apply functions to get new functions, we can partially apply type parameters and get new type constructors from them.

Just like we call a function with too few parameters to get back a new function, we can specify a type constructor with too few type parameters and get b ack a partially applied type constructor.

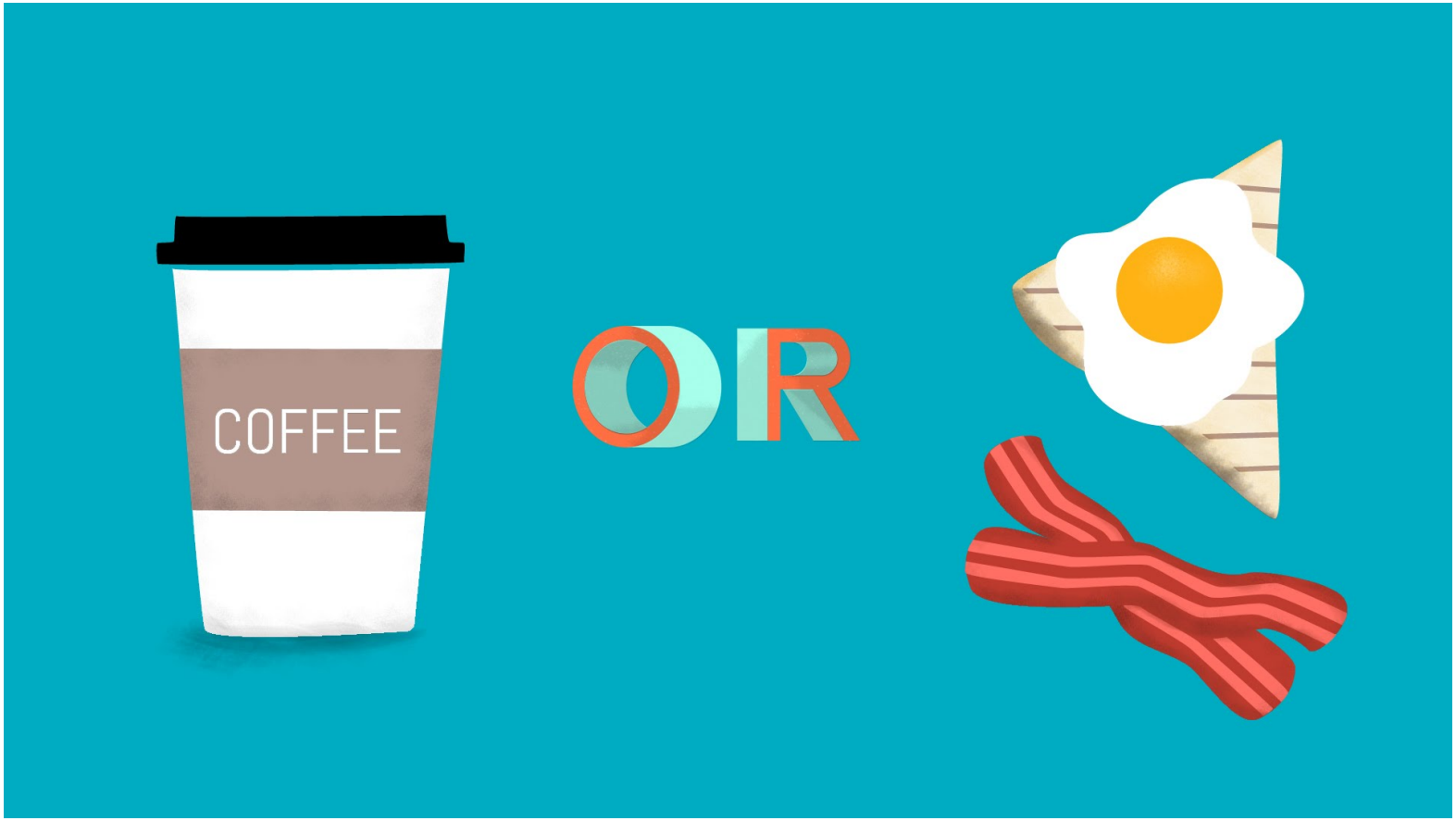
If we wanted a type that represents a map (from `Data.Map`) from integers to something, we could either do this:

```
type IntMap v = Map Int v
```

Or we could do it like this:

```
type IntMap = Map Int
```

Type synonyms



Either

Another cool data type that takes two types as its parameters is the `Either a b` type.

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

It has two value constructors.

If the `Left` is used, then its contents are of type `a` and if `Right` is used, then its contents are of type `b`.

So we can use this type to encapsulate a value of one type or another and then when we get a value of type `Either a b`, we usually pattern match on both `Left` and `Right` and we different stuff based on which one of them it was.

Either

```
ghci: Left 10
Left 10
ghci: :t Left 10
Left 10 :: Num a => Either a b
ghci: Right "abc"
Right "abc"
ghci: :t Right "abc"
Right "abc" :: Either a [Char]
ghci:
```

Recursive data structures

Recursive data structures

Let's use algebraic data types to implement our own list

```
data List a = Empty | Cons a (List a)
  deriving (Show, Read, Eq, Ord)
```

It's either an empty list or a combination of a head with some value and a list.

```
ghci: Empty
Empty
ghci: 3 `Cons` Empty
Cons 3 Empty
ghci: 2 `Cons` (3 `Cons` Empty)
Cons 2 (Cons 3 Empty)
ghci: 1 `Cons` (3 `Cons` (3 `Cons` Empty))
Cons 1 (Cons 3 (Cons 3 Empty))
ghci:
```

Recursive data structures

We called our `Cons` constructor in an infix manner so you can see how it's just like `:`.

We can define functions to be automatically infix by making them comprised of only special characters. We can also do the same with constructors, since they're just functions that return a data type.

```
infixr 5 :-:
data List a = Empty | a :-: (List a)
  deriving (Show, Read, Eq, Ord)
```

First off, we notice a new syntactic construct, the fixity declarations. When we define functions as operators, we can use that to give them a fixity (but we don't have to).

Recursive data structures

```
ghci: 1 :-: 2 :-: 3 :-: Empty
1 :-: (2 :-: (3 :-: Empty))
ghci: let l = 1 :-: 2 :-: 3 :-: Empty
ghci: 0 :-: l
0 :-: (1 :-: (2 :-: (3 :-: Empty)))
ghci:
```

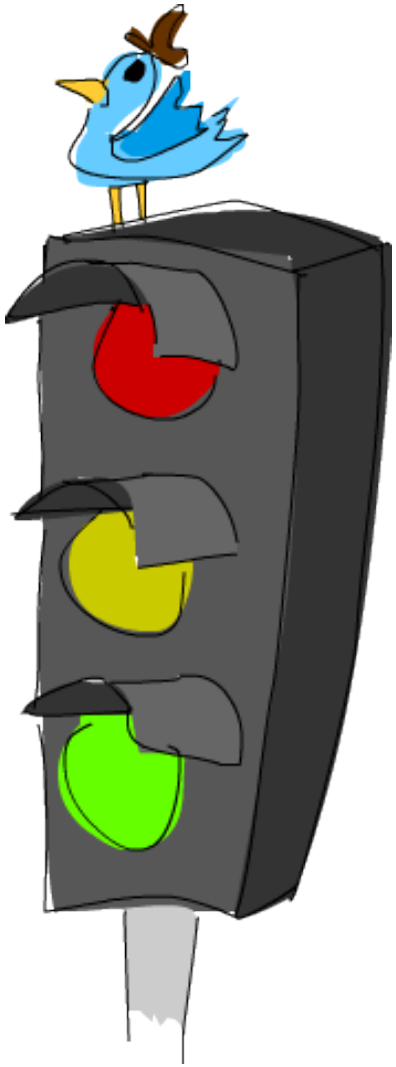
Recursive data structures

Let's make a function that adds two of our lists together.

```
infixr 5  .++
(.++) :: List a -> List a -> List a
Empty .++ ys = ys
(x :-: xs) .++ ys = x :-: (xs .++ ys)
```

```
ghci: let l = 1 :-: 2 :-: 3 :-: Empty
ghci: let l' = 4 :-: 5 :-: Empty
ghci: l .++ l'
1 :-: (2 :-: (3 :-: (4 :-: (5 :-: Empty))))
ghci: l' .++ l
4 :-: (5 :-: (1 :-: (2 :-: (3 :-: Empty))))
ghci:
```

Typeclasses 102



Typeclasses 102

A typeclass defines some behavior (like comparing for equality, comparing for ordering, enumeration) and then types that can behave in that way are made instances of that typeclass.

The behavior of typeclasses is achieved by defining functions or just type declarations that we then implement.

So when we say that a type is an instance of a typeclass, we mean that we can use the functions that the typeclass defines with that type.

Typeclasses have pretty much nothing to do with classes in languages like Java or Python. This confuses many people, so I want you to forget everything you know about classes in imperative languages right now.

Typeclasses 102

For example, the `Eq` typeclass is for stuff that can be equated.

It defines the functions `==` and `/=`.

If we have a type (say, `Car`) and comparing two cars with the equality function `==` makes sense, then it makes sense for `Car` to be an instance of `Eq`.

Typeclasses 102

This is how the `Eq` class is defined in the standard prelude:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Typeclasses 102

First off, when we write `class Eq a where`, this means that we're defining a new typeclass and that's called `Eq`.

The `a` is the type variable and it means that a will play the role of the type that we will soon be making an instance of `Eq`.

It doesn't have to be called `a`, it doesn't even have to be one letter, it just has to be a lowercase word.

Then, we define several functions. It's not mandatory to implement the function bodies themselves, we just have to specify the type declarations for the functions.

Typeclasses 102

Anyway, we did implement the function bodies for the functions that `Eq` defines, only we defined them in terms of mutual recursion.

We said that two instances of `Eq` are equal if they are not different and they are different if they are not equal.

We didn't have to do this, really, but we did and we'll see how this helps us soon.

If we have say `class Eq a where` and then define a type declaration within that class like `(==) :: a -> -a -> Bool`, then when we examine the type of that function later on, it will have the type of `(Eq a) => a -> a -> Bool`.

Typeclasses 102

```
data TrafficLight = Red | Yellow | Green
```

Notice how we didn't derive any class instances for it. That's because we're going to write up some instances by hand, even though we could derive them for types like `Eq` and `Show`.

Here's how we make it an instance of Eq.

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

Typeclasses 102

We did it by using the `instance` keyword.

So `class` is for defining new typeclasses and `instance` is for making our types instances of typeclasses.

When we were defining `Eq`, we wrote `class Eq a where` and we said that `a` plays the role of whichever type will be made an instance later on.

We can see that clearly here, because when we're making an instance, we write `instance Eq TrafficLight where`. We replace the a with the actual type.

Because `==` was defined in terms of `/=` and vice versa in the class declaration, we only had to overwrite one of them in the instance declaration.

Typeclasses 102

We did it by using the `instance` keyword.
If `Eq` was defined simply like this:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

we'd have to implement both of these functions when making a type an instance of it, because Haskell wouldn't know how these two functions are related.

The minimal complete definition would then be: both `==` and `/=`.

Typeclasses 102

Let's make this an instance of `Show` by hand, too.

To satisfy the minimal complete definition for `Show`, we just have to implement its `show` function, which takes a value and turns it into a string.

```
instance Show TrafficLight where
    show Red    = "Red light"
    show Yellow = "Yellow light"
    show Green  = "Green light"
```

Typeclasses 102

```
ghci: Red
Red light
ghci: Red == Red
True
ghci: Red == Green
False
ghci: Red /= Yellow
True
ghci: Red `elem` [Red, Yellow, Green]
True
ghci: [Red, Yellow, Green]
[Red light,Yellow light,Green light]
ghci:
```

Typeclasses 102

`Maybe` in itself isn't a concrete type, it's a type constructor that takes one type parameter (like `Char` or something) to produce a concrete type (like `Maybe Char`).

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

From the type declarations, we see that the `a` is used as a concrete type because all the types in functions have to be concrete (remember, you can't have a function of the type `a -> Maybe` but you can have a function of `a -> Maybe a` or `Maybe Int -> Maybe String`).

Typeclasses 102

```
instance Eq (Maybe m) where
    Just x == Just y   = x == y
    Nothing == Nothing = True
    _ == _             = False
```

This is like saying that we want to make all types of the form `Maybe` something an instance of `Eq`. We actually could have written `(Maybe something)`, but we usually opt for single letters to be true to the Haskell style.

The `(Maybe m)` here plays the role of the `a` from `class Eq a where`. While `Maybe` isn't a concrete type, `Maybe m` is.

By specifying a type parameter (`m`, which is in lowercase), we said that we want all types that are in the form of `Maybe m`, where `m` is any type, to be an instance of `Eq`.

Typeclasses 102

There's one problem with this though.

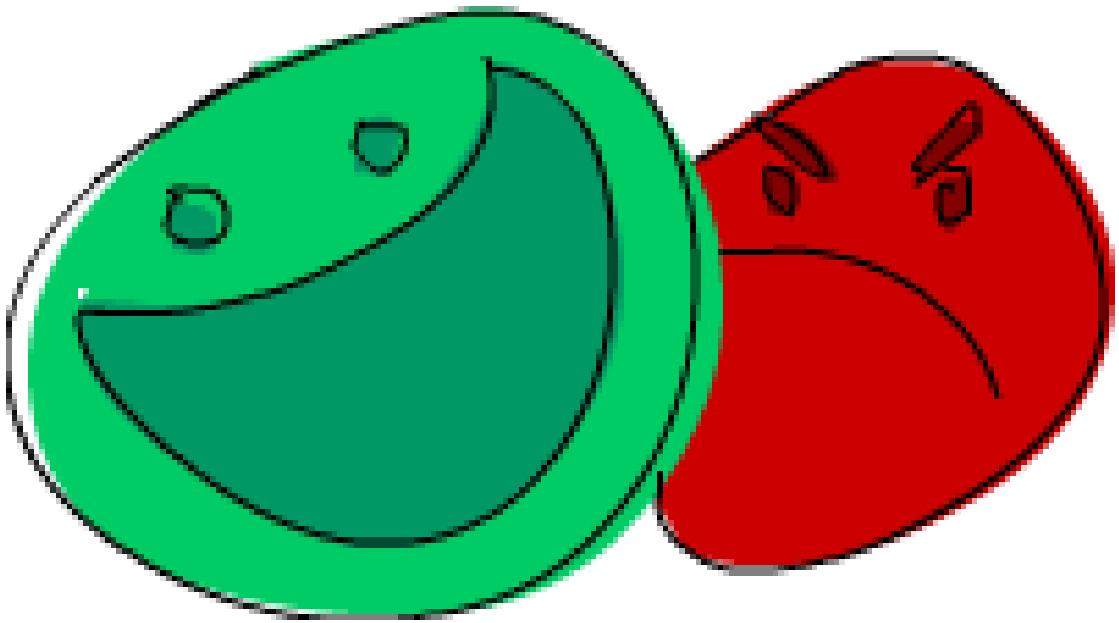
We use `==` on the contents of the `Maybe` but we have no assurance that what the `Maybe` contains can be used with `Eq`!

That's why we have to modify our instance declaration like this:

```
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y   = x == y
    Nothing == Nothing = True
    _ == _             = False
```

We want all types of the form `Maybe m` to be part of the `Eq` typeclass, but only those types where the `m` (so what's contained inside the `Maybe`) is also a part of `Eq`.

A yes-no typeclass



A yes-no typeclass

In JavaScript and some other weakly typed languages, you can put almost anything inside an if expression.

For example, you can do all of the following:

```
if (0) alert("YEAH!") else alert("NO!")

if ("") alert ("YEAH!") else alert("NO!")

if (false) alert("YEAH") else alert("NO!)

etc
```

A yes-no typeclass

```
class YesNo a where
    yesno :: a -> Bool

instance YesNo Int where
    yesno 0 = False
    yesno _ = True

instance YesNo [a] where
    yesno [] = False
    yesno _  = True

instance YesNo Bool where
    yesno = id

instance YesNo (Maybe a) where
    yesno (Just _) = True
    yesno Nothing  = False
```

A yes-no typeclass

```
ghci: yesno $ length []
False
ghci: yesno "haha"
True
ghci: yesno ""
False
ghci: yesno $ Just 0
True
ghci: yesno True
True
ghci: yesno []
False
ghci: yesno [0,0,0]
True
ghci: :t yesno
yesno :: YesNo a => a -> Bool
ghci:
```

A yes-no typeclass

```
yesnoIf :: (YesNo y) => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult =
  if yesno yesnoVal then yesResult else noResult
```

```
ghci: yesnoIf [] "YEAH!" "NO!"
"NO!"
ghci: yesnoIf [2,3,4] "YEAH!" "NO!"
"YEAH!"
ghci: yesnoIf True "YEAH!" "NO!"
"YEAH!"
ghci: yesnoIf (Just 500) "YEAH!" "NO!"
"YEAH!"
ghci: yesnoIf Nothing "YEAH!" "NO!"
"NO!"
ghci:
```