1 Type Classes

Consider the following operator: `+`

- For `Integer`s, we have

```
$ 2 + 3
5
```

- For `Double`s, we have

```
$ 2.9 + 3.5
6.4
```

But, we can also do things like

```
$ [2.9, 3.5] == [2.9, 3.5]
True

$ ("cat", 10) < ("cat", 20)
True
```

How does this work?

(Quiz.) Which of the following type annotations would work for `(+)`?

(a) `(+) :: Int -> Int -> Int`

(b) `(+) :: Double -> Double -> Double`

(c) `(+) :: a -> a -> a`

(d) Any of the above.

(e) None of the above.

The answer is **E**. If we picked A, then we can't add two `Double`s; the same idea applies for B. For C, we can't add, for example, two `Bool`s.

We need **ad-hoc polymorphism**. To do this, we make use of *type classes*.

1.1 Constrained Types

We note that

```
$ :type (+)
(+) :: (Num a) => a -> a -> a
```

Here, this is saying that `(+)` takes in two `a` values and returns an `a` value, such that `a` is an *instance of* the `Num` type class[1]. Then, `(Num a) =>` is the *constraint*.

Let's try to add two `Bool` values.

```
$ True + False
<interactive>
    No instance for (Num Bool) arising from a use of '+'
    In the expression: True + False
    In an equation for 'it': it = True + False
```

This means that `True` and `False` are of type `Bool`, but that `Bool` is not an instance of `Num`.

[1] In terms of Java, we can think of `Num` as an interface, so `a` would have to implement the `Num` interface.

(Quiz.) What would be a reasonable type for the equality operator?

(a) `(==) :: a -> a -> a`

(b) `(==) :: a -> a -> Bool`

(c) `(==) :: (Eq a) => a -> a -> a`

(d) `(==) :: (Eq a) => a -> a -> Bool`

(e) None of the above

The answer is **D**. Note that one example of something that can't really be compared are *functions*.

1.2 What is a Type Class?

A type class is a *collection of methods* (functions, operators) that must exist for every instance. Some useful type classes in the Haskell standard library are

- The `Eq` Type Class for **Equality**.

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

  Note that a type `T` is an instance of `Eq` if there are two functions

  - `(==) :: T -> T -> Bool` that determines if two `T` values are equal.
  - `(/=) :: T -> T -> Bool` that determines if two `T` values are not equal.

- The `Show` Type Class

```
class Show a where
    show :: a -> String
```

  This type class requires that instances are convertible to `String` so that it can be displayed. To see what we mean, note that

```
$ 2
2

$ show 2
"2"

$ show 3.14
"3.14"

$ show (1, "two", ([], [], []))
"(1,\"two\",([],[],[]))"
```

- The `Ord` Type Class for **Order**.

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

Note the `Eq a =>`. A type `T` is an instance of `Ord` if `T` is *also* instance of `Eq`, and it defines functions for comparing values for inequalities.

In other words, if `T` implements `Ord`, then it must also implement `Eq` (i.e., `Ord` depends on `Eq`).

1.3 Creating Type Classes

Consider the datatype

```
data Color = Red | Green
```

Let us now add a declaration for `Show` on `Color`:

```
instance Show Color where
    show Red        = "Red"
    show Green      = "Green"
```

Let's do the same thing for `Eq`:

```
instance Eq Color where
    (==) Red    Red     = True
    (==) Green  Green   = True
    (==) _      _       = False
    (/=) x      y       = not (x == y)
```

This is tedious, and this type isn't very complicated. Indeed, there is a way for us to *automatically* do this, using the `deriving` keyword.

```
data Color = Red | Green
    deriving (Eq, Show)
```