Type Classes

We already know that when declaring the type of a function you can either be super specific by giving the exact data type of a parameter (e.g. `Int` or `Char` etc.); or you can be super general using universal quantification, but what if you want a half-way house? Say you are making a function that two adds numbers. You don't care whether the numbers are stored as `Int`s or `Double`s you just want to add them. If you chose the specific option and define `add :: Int -> Int -> Int` it will only work on `Int`s; likewise you also hit trouble if you take the general route: you can't define `add :: a -> a -> a` what if someone gives you `Char`s? How does one even add characters?

Luckily the clever people at Haskell have gifted us type classes. You can think of type classes as families of types. Each family has its own rules and to be a member of a family a type must possess certain qualities. Once a member of a class the type must fulfil the promises of the type class. Each type class is defined with certain functions that any member must be able to perform. The `Num` class, for example, is the perfect class to solve our specificness problem. It is the class of types that basic arithmetic (add, subtract, multiply, negate) can be performed on:

```
class Num a where
      (+)  ::  a -> a -> a
      (-)  ::  a -> a -> a
      (*)  ::  a -> a -> a
   negate  ::  a -> a
```

Now we can define `add` so that it will work on any numeric `a`:

```
add :: Num a => a -> a -> a
add x y = x + y
```

Notice how the class only gives the type signatures of the functions it wants its members to be able to perform. That is because every type is different and unique. I don't write letters in the same way that you write letters, similarly `Int`s will add themselves differently from `Double`s. When a type joins a type class we say that it is an instance of that class. It is at this moment that the definitions for these functions are provided. You can think of this as a sort of initiation.

So let's make a new data type and take it through a fresher fair style tour of which type classes it could join.

```
data Fresher  =  CompSci
              |  Maths
```

Our bright eyed new type looks about at all the type classes it could join. It could join the `Eq` type class and learn how to compare different constructors of `Freshers`, learn that a `Maths` student is not the same as a `CompSci` student. It could join the `Show` society: a type class obsessed with showing off, requiring all its members to be confident enough to print out results onto the terminal. Then you've got the `Ord` type class where it could once and for all be decided which degree is the best. Don't forget the mythical `Monad` class, a class from legend that will only reveal itself to the truly worthy.

Now like the dutiful caring parents we are: let's help our type class initiate itself into the `Show` type class so that it can introduce itself.

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

instance Show Fresher where
                  show ComSci  =  "Hello I am a computer scientist!"
                  show Maths   =  "Hi I study maths!"
```

Now GHCi knows that if it wants to print out a `Maths` Fresher it needs to print "Hi I study maths!".