COMP105 Lecture 19

More Complex Custom Types

More complex constructors

More complex constructors can contain **other types**

```
data Point = Point Int Int deriving (Show, Read, Eq)
```

```
ghci> Point 1 4
Point 1 4

ghci> read "Point 10 10" :: Point
Point 10 10

ghci> Point 2 2 /= Point 3 1
True
```

More complex constructors

It is common to use **pattern matching** to work with complex constructors

```
shift_up (Point x y) = Point x (y+1)
```

```
ghci> shift_up (Point 1 1)
Point 1 2
```

```
ghci> :t shift_up
shift_up :: Point -> Point
```

Example

```haskell
move :: Point -> Direction -> Point

move (Point x y) North = Point x (y+1)
move (Point x y) South = Point x (y-1)
move (Point x y) East  = Point (x+1) y
move (Point x y) West  = Point (x-1) y
```

```
ghci> move (Point 0 0) North
Point 0 1
```

Even more complex constructors

Types can have **multiple constructors** each of which can have their own types

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

```
ghci :t Circle 2.0
Circle 2.0 :: Shape
```

```
ghci> :t Rect 3.0 4.0
Rect 3.0 4.0 :: Shape
```

Example

```haskell
area :: Shape -> Float

area (Circle radius) = pi * radius**2
area (Rect x y) = x * y
```

```
ghci> area (Circle 2.0)
12.566371

ghci> area (Rect 3.0 4.0)
12.0
```

Records

You can use data types to build custom **records**...

```
data Person = Person String String Int String
```

```
get_first_name  (Person x _ _ _) = x
get_second_name (Person _ x _ _) = x
get_age         (Person _ _ x _) = x
get_nationality (Person _ _ _ x) = x
```

```
ghci> get_age (Person "joe" "bloggs" 25 "UK")
25
```

Record syntax

To make things easier, Haskell provides a **record syntax**

```
data Person = Person { firstName :: String,
                       secondName :: String,
                       age :: Int,
                       nationality :: String}
                                  deriving(Show)
```

```
ghci> Person "joe" "bloggs" 25 "UK"
Person {firstName = "joe", secondName = "bloggs",
          age = 25, nationality = "UK"}
```

Record syntax

When you use the record syntax, Haskell automatically creates **getter** functions for each parameter

```
gchi let joe = Person "joe" "bloggs" 25 "UK"
```

```
gchi> firstName joe
"joe"
```

```
ghci> secondName joe
"bloggs"
```

Record syntax

Records can be created **out of order** (normal data types cannot)

```
data Example = Example { a :: String, b :: Int}
                                       deriving (Show)
```

```
ghci> Example "one" 2
Example {a = "one", b = 2}
```

```
ghci> Example {b = 3, a = "zero"}
Example {a = "zero", b = 3}
```

Example

```haskell
data AdvShape =   AdvCircle Point Float
                | AdvRect   Point Point
                                 deriving (Show)


area' (AdvCircle _ radius) = pi * radius**2


area' (AdvRect (Point x1 y1) (Point x2 y2)) =
    let
        w = abs (x1 - x2)
        h = abs (y1 - y2)
    in
        fromIntegral (w * h)
```

Exercises

1. Copy the `Point` type into your file. Write a function `distance :: Point -> Int` that returns the sum of the x and y coordinates of the point
2. Create a custom type `HTTPResponse` that has two constructors: `Data` has an `Int` (response code) and a `String` (data), while `Error` has a `String` containing the error
3. Write a record type for a `Student` that has information about the students name as a string, their address as a string, and their marks as a list of integers