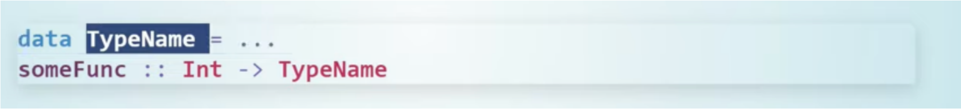
Week05 Presentation

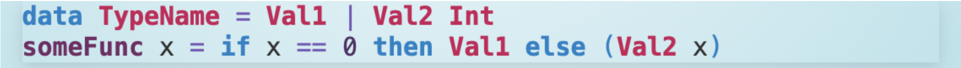
--- to help with Exercise 04

Type Constructors -> Defines the name of your new type



```
data TypeName = ...
someFunc :: Int -> TypeName
```

Value Constructors -> Allow you to define new values and wrap other values.



```
data TypeName = Val1 | Val2 Int
someFunc x = if x == 0 then Val1 else (Val2 x)
```

-I.E. val1 is already a value of data type 'TypeName' val2 requires an int , and then creates a value. On it's own is a value constructor (something like a function)

Wrapping up

A data constructor is a "function" that takes 0 or more values and gives you back a new value.
A type constructor is a "function" that takes 0 or more types and gives you back a new type.

Can write shortcuts in Haskell:

```
data UniversityID = StudentID3 {macID3:: String,
studentNum3 :: String}
| FacultyID {macID3 :: String, facultyNum:: String, salary:: Float }
```

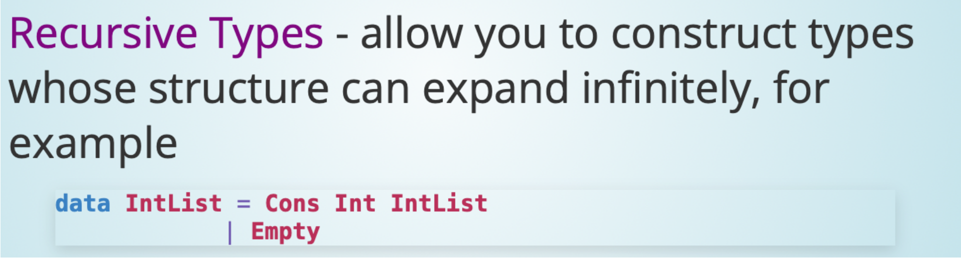
Vs:

```
data UniversityID: StudentID3 String String | FacultyID String String String
```

What is the value Constructor and what is the Type constructor?

Reminder: Recursive Types:

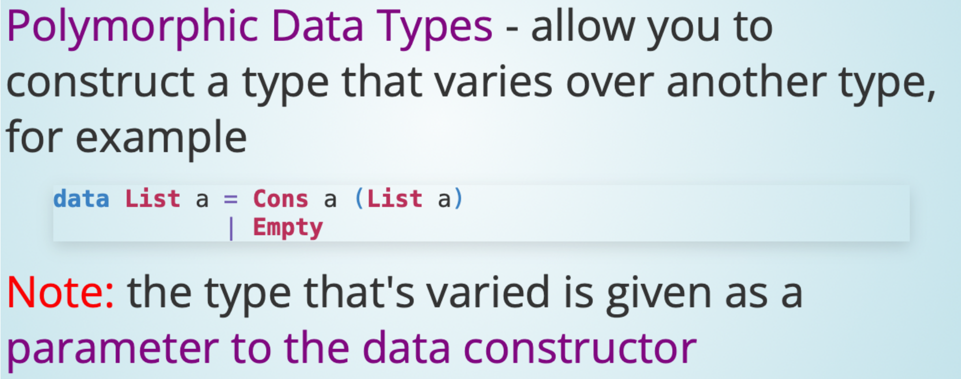
- **Can construct types - which could essentially expand itself infinitely. i.e. Natural Numbers**
- **Define itself within the type.**



Recursive Types - allow you to construct types whose structure can expand infinitely, for example

```
data IntList = Cons Int IntList
             | Empty
```

Polymorphic Data Types:



Polymorphic Data Types - allow you to construct a type that varies over another type, for example

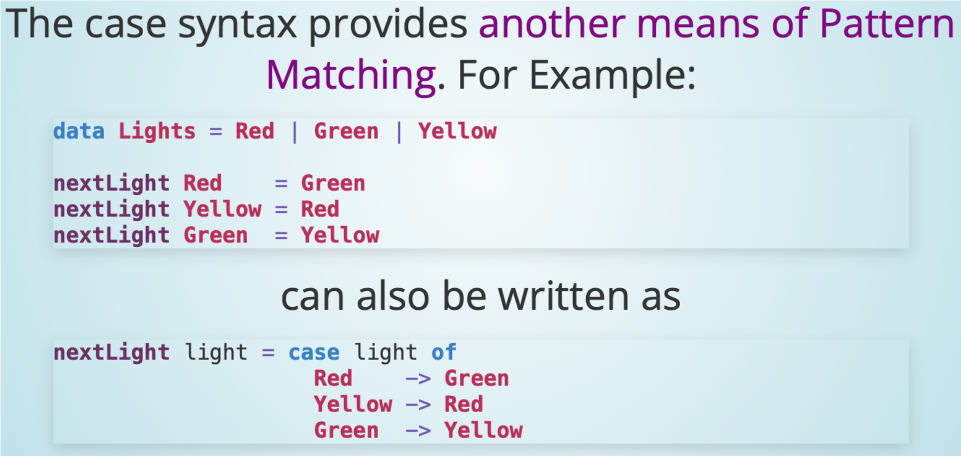
```
data List a = Cons a (List a)
            | Empty
```

Note: the type that's varied is given as a parameter to the data constructor

i.e. 'a' is a placeholder for any type. Now our data type List can contain a list of any type.

-- What are the base cases in these recursive types?

Case Syntax:



The case syntax provides another means of Pattern Matching. For Example:

```
data Lights = Red | Green | Yellow

nextLight Red    = Green
nextLight Yellow = Red
nextLight Green  = Yellow
```

can also be written as

```
nextLight light = case light of
                    Red    -> Green
                    Yellow -> Red
                    Green  -> Yellow
```

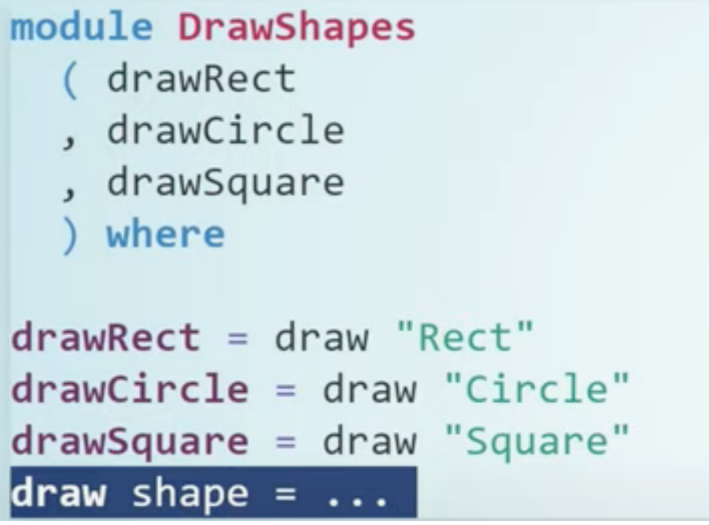
In the first definition, we write multiple definitions of the functions with specific values given to it. The second has one definition.

Prelude

Standard Library!!

- Collection of useful code already written for you

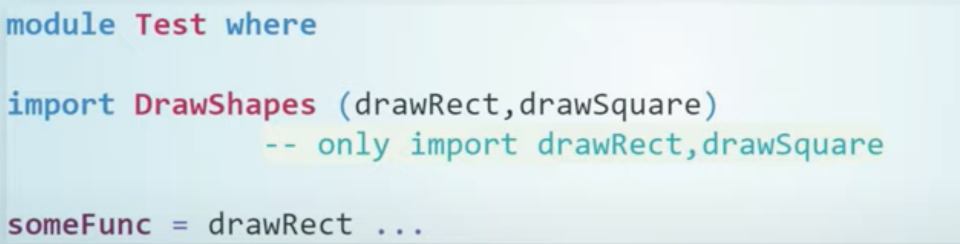
Module



```
module DrawShapes
   ( drawRect
   , drawCircle
   , drawSquare
   ) where

drawRect = draw "Rect"
drawCircle = draw "Circle"
drawSquare = draw "Square"
draw shape = ...
```

```
module Test where
Import DrawShapes (DrawRect, drawSquare) -- will only import the two functions specified , i.e. no access to drawCircle.
```



```
module Test where

import DrawShapes (drawRect,drawSquare)
                   -- only import drawRect,drawSquare

someFunc = drawRect ...
```

- DrawShapes module would be in one file, and the Test.hs would be in another file.

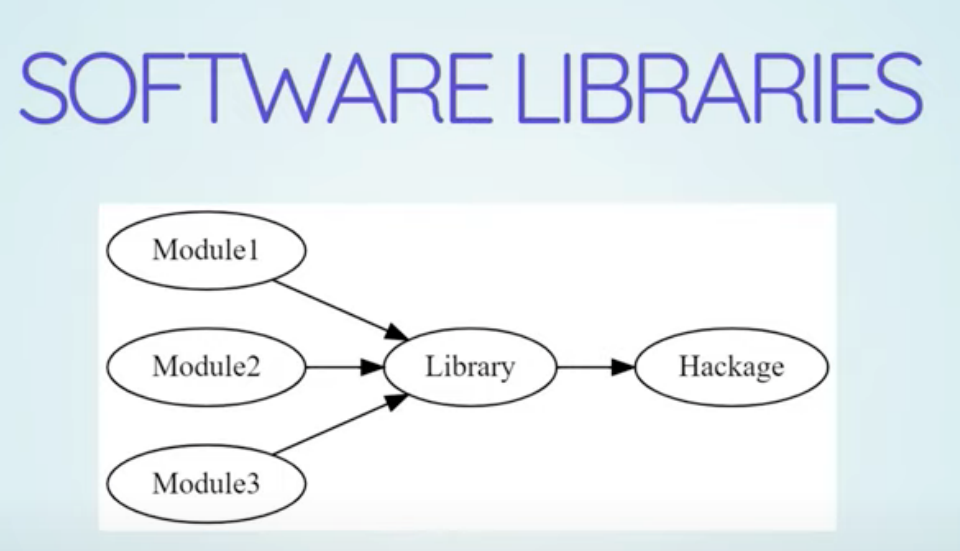
- When you compile the program (i.e. 'stack run' the main program gets run)

TIP:: Restart Haskell Language Server if giving you "Module not found" error.

putStrLn :: String -> IO ()

The keyword do introduces a sequence of statements which are executed in order

Libraries vs Modules


SOFTWARE LIBRARIES

Library -> Package all modules into a library
Hackage -> all haskell libraries are online

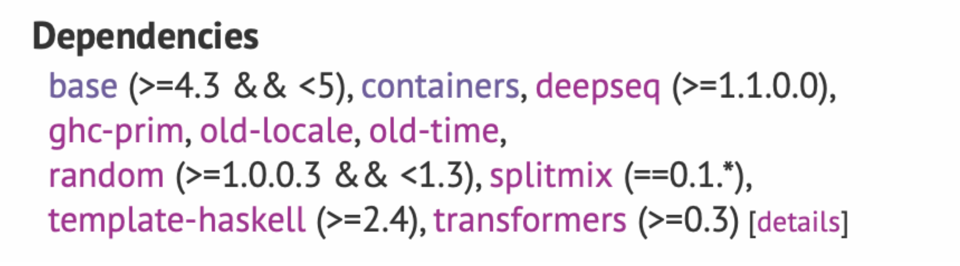
Hoogle.haskell.org : Gives you insightful list of all packages, libraries and modules in Haskell.

A **module** is a set of functions, types, classes, ... put together in a common namespace.
A **library** is a set of modules which makes sense to be together and that can be used in a program or another library.
A **package** is a unit of distribution that can contain a library or an executable or both. It's a way to share your code with the community.

'Base' -> Base library

- Basic Library
- Contains prelude, and other modules.

QuickCheck -> package
Dependencies -> other libraries that are needed for the current library


Dependencies

base (>=4.3 && <5), containers, deepseq (>=1.1.0.0),
ghc-prim, old-locale, old-time,
random (>=1.0.0.3 && <1.3), splitmix (==0.1.*),
template-haskell (>=2.4), transformers (>=0.3) [details]

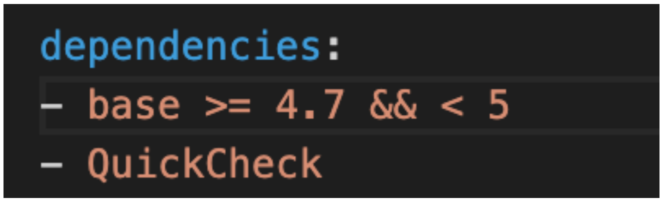
i.e. "uses this library past this version"

When you build a project with stack uses a resolver (a default) . If you install quickcheck, it knows which version to install and it works with the version of the compiler you are using.

'stack new --resolver=lts-14.27 ProjectName'

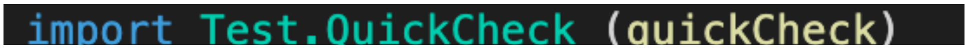
Package.yaml -> dependencies

Under dependencies:



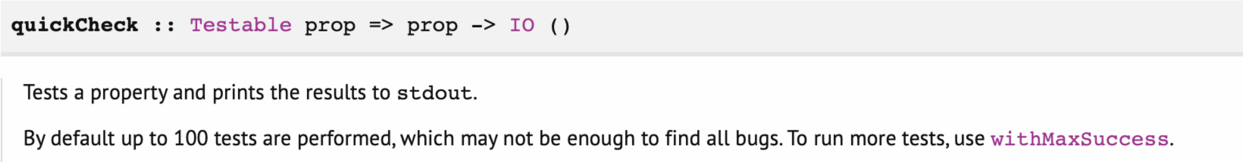
```
dependencies:
- base >= 4.7 && < 5
- QuickCheck
```

QuickCheck (without the hyphen or version number)



```
import Test.QuickCheck (quickCheck)
```

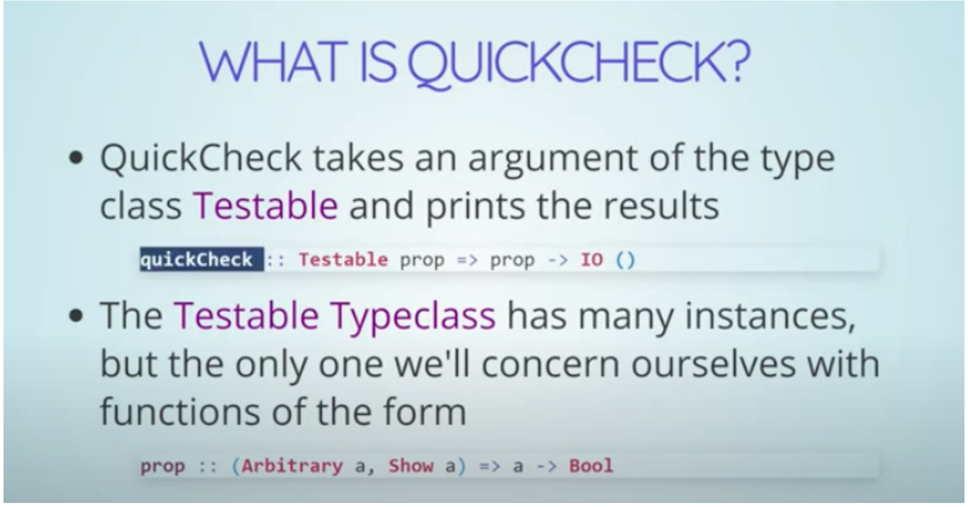
We are importing the Test module under the QuickCheck library, and are exclusively importing just the quickCheck function.
In Stackage.org < Test.QuickCheck, you can see more information on what this module does, and the functions it contains.



```
quickCheck :: Testable prop => prop -> IO ()
```

Tests a property and prints the results to `stdout`.

By default up to 100 tests are performed, which may not be enough to find all bugs. To run more tests, use `withMaxSuccess`.



WHAT IS QUICKCHECK?

- QuickCheck takes an argument of the type class Testable and prints the results

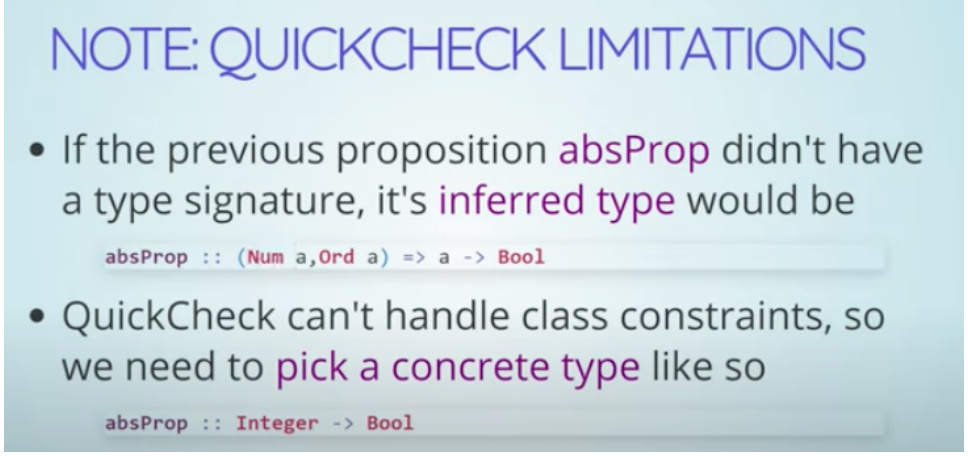
```
quickCheck :: Testable prop => prop -> IO ()
```

- The Testable Typeclass has many instances, but the only one we'll concern ourselves with functions of the form

```
prop :: (Arbitrary a, Show a) => a -> Bool
```

Prop is some "function" that returns a Bool

- Takes any input 'a' as long as it is a part of the 'Arbitrary' and 'Show' class.
- Therefore any default types in Prelude. If you make your own type -you'd have to worry about making it apart
- i.e. is a property that is a test passes it'll return true



NOTE: QUICKCHECK LIMITATIONS

- If the previous proposition absProp didn't have a type signature, it's inferred type would be

```
absProp :: (Num a,Ord a) => a -> Bool
```

- QuickCheck can't handle class constraints, so we need to pick a concrete type like so

```
absProp :: Integer -> Bool
```

In other words, we need to pick a SPECIFIC TYPE and not a general type (polymorphic)

Very useful for debugging and testing script!!