Presentation Week03: Conditionals & Functions

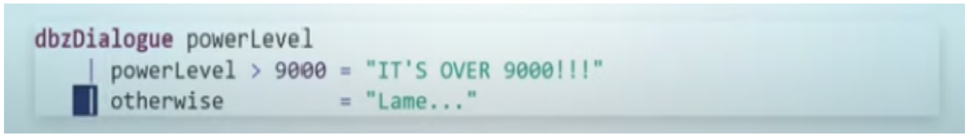
If Expressions:

- If **(conditional expression that will give a Boolean)** then **(This happens)** else **(This happens)**
- **Must evaluate to ONE value**
-
    - **Function :: Integer -> String**

Function n = If (cond) else (A STRING) THEN (A STRING) <-- both return values must be a string

Guarded Equations

- Alternative but equivalent to If expressions



```
dbzDialogue powerLevel
  | powerLevel > 9000 = "IT'S OVER 9000!!!"
  | otherwise         = "Lame..."
```

- | are the guards

Undefined/ Errors - Partial Function :

- Used to cover cases that should never be reached
    - If True then 0 else undefined <-- always true

- If program reaches undefined, it will crash
- ERROR function is the same as undefined, except it outputs an error message
- Probably better to use 'error' function, since it outputs a string explaining the problem

Pattern Matching:

- Pattern matches allows you to write pre-defined values for function definitions
- i.e. not :: Bool -> Bool -- instead of passing in values:

```
Not False = True
Not True = False
```

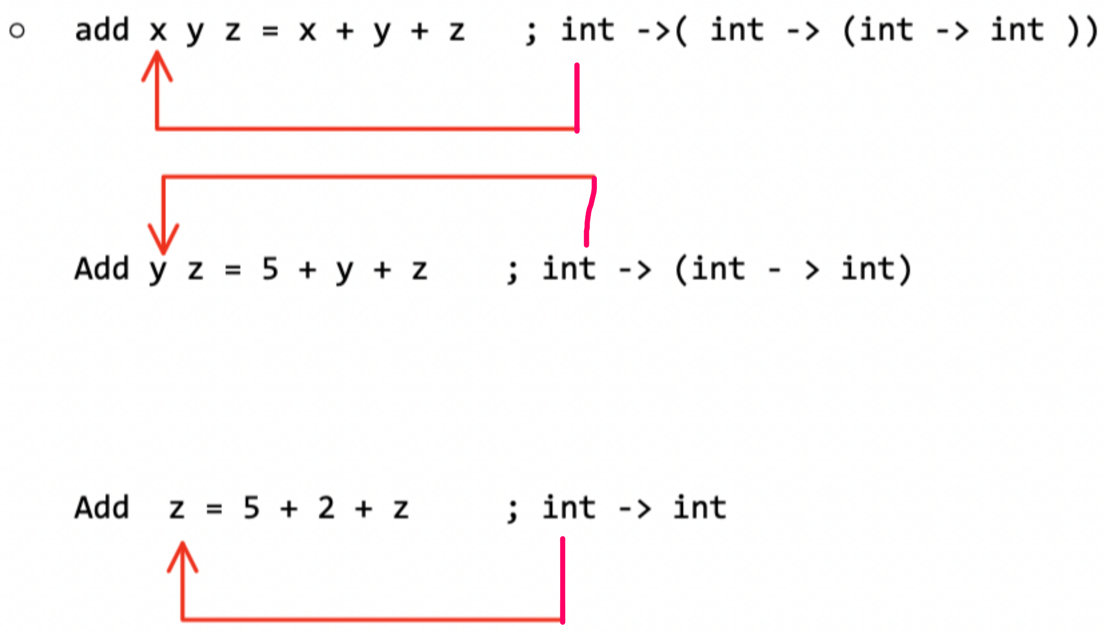
So now when Haskell sees you using the 'Not' Function defined, whenever you pass it False, the result will True, and vice versa.

Currying:

- Process of transforming a function that takes multiple arguments into a function that takes just a single argument, and returns another function, which accepts further arguments
- Haskell functions are curried by default.

addn x y = x + y
- -- int -> (int -> int)
- -- function takes an int, and returns another function.

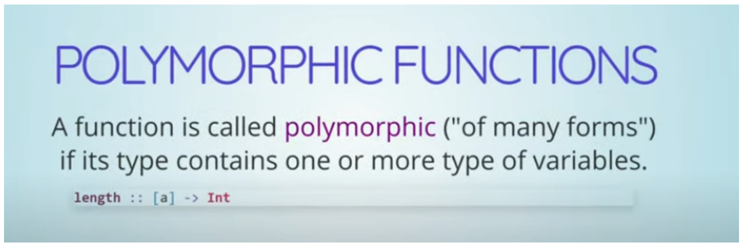
- add5 = addn 5
- returns a function with only ONE argument
- dont need to add a variable parameter - implicitly there



```
○   add x y z = x + y + z    ; int ->( int -> (int -> int ))

    Add y z = 5 + y + z     ; int -> (int - > int)

    Add  z = 5 + 2 + z      ; int -> int
```

To specify an uncurried function:
Use tuples: i.e. add: (int->int) -> int ; you cant specify one argument at a time

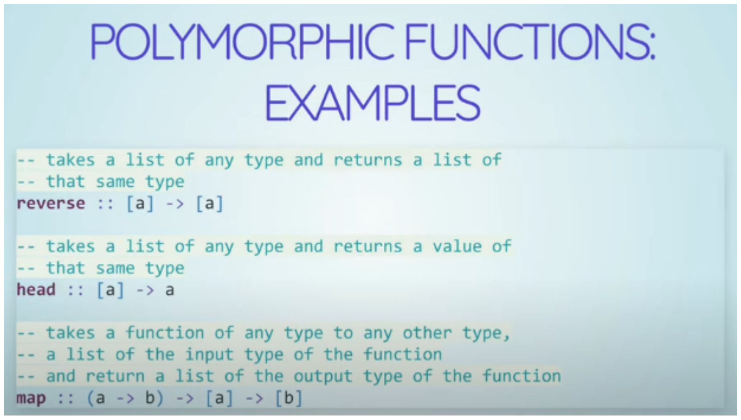
Polymorphic Functions



POLYMORPHIC FUNCTIONS

A function is called polymorphic ("of many forms") if its type contains one or more type of variables.

```
length :: [a] -> Int
```



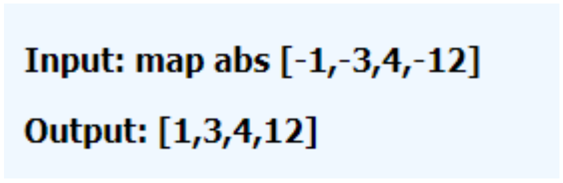
POLYMORPHIC FUNCTIONS: EXAMPLES

```
-- takes a list of any type and returns a list of
-- that same type
reverse :: [a] -> [a]

-- takes a list of any type and returns a value of
-- that same type
head :: [a] -> a

-- takes a function of any type to any other type,
-- a list of the input type of the function
-- and return a list of the output type of the function
map :: (a -> b) -> [a] -> [b]
```

** You have to make sure that if this function returns a value within a list, that it's the same type. i.e. [a] -> a

Map

Takes a list with type a, and a function, and returns the output type of that function - applies function to each element of the list .

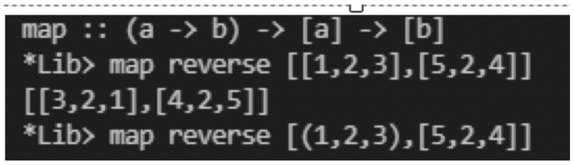


Input: map abs [-1,-3,4,-12]

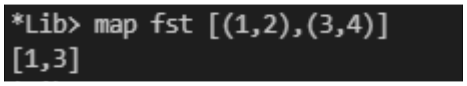
Output: [1,3,4,12]

In this case abs is the function argument of map, and the list [-1-3,-4,-12] is the list parameter of map.

(a->b) : means a function that gives a type to a type. Could be lists of lists:



```
map :: (a -> b) -> [a] -> [b]
*Lib> map reverse [[1,2,3],[5,2,4]]
[[3,2,1],[4,2,5]]
*Lib> map reverse [(1,2,3),[5,2,4]]
```



```
*Lib> map fst [(1,2),(3,4)]
[1,3]
```

Function Composition

```
-- When input parameter of a function f, is another function (i.e. g). The output type of
                         Z = f .g    ==. Z = f (g(x) )
-- when using this operator, argument is implicitly there.
```

Parenthesis Buildup

```
$  -> replaces parenthesis , therefore the parenthesis must run until the end.

            fun (x:xs) = foldr (+) 0 (map (+2) (reverse (init(x:xs))))
            fun1 (x:xs) = foldr (+) 0 $ map (+2) $ reverse $ init $ x:xs

Can't get the parenthesis in the middle, since it does not run until the end!
```

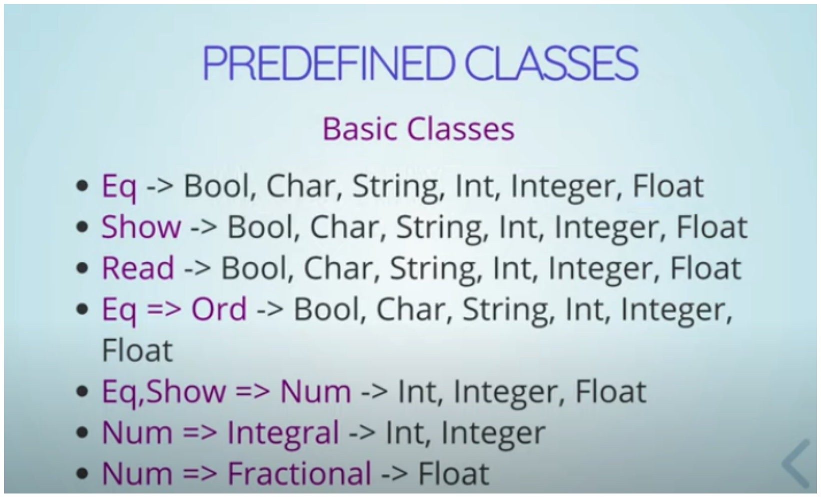
TYPE CLASSES

- Type classes is a sort of interface that defines behavior. The type classes have specified methods
    - Type class definition, only has type signature. In order to actually implement the methods described, you must create "instances" of the type class.

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

            instance Eq Bool where                                -- we are giving this instance type Bool, this is
            how double equals will be defined for bool.
            x and y therefore, must be type Bool
                  x == y = if x then (if y then True else False).      -- If 'true' then if 'true' then True, else False
                  (for when x is true)
                              else (if y then False else True)         -- If 'false' else if 'true' then False, else True
                              (for when x is false)
                  x /= y = not (x == y )
```

- By creating an instance of the Eq class with type bool, we can use the methods described in the class to compare where two Boolean values are equal.
- In other words:

Bool is the type, Eq is the TYPECLASS with methods defined, where we create an INSTANCE of the Eq class of type Bool, to enable us to compare two Bool values.



PREDEFINED CLASSES

Basic Classes

- Eq -> Bool, Char, String, Int, Integer, Float
- Show -> Bool, Char, String, Int, Integer, Float
- Read -> Bool, Char, String, Int, Integer, Float
- Eq => Ord -> Bool, Char, String, Int, Integer, Float
- Eq,Show => Num -> Int, Integer, Float
- Num => Integral -> Int, Integer
- Num => Fractional -> Float

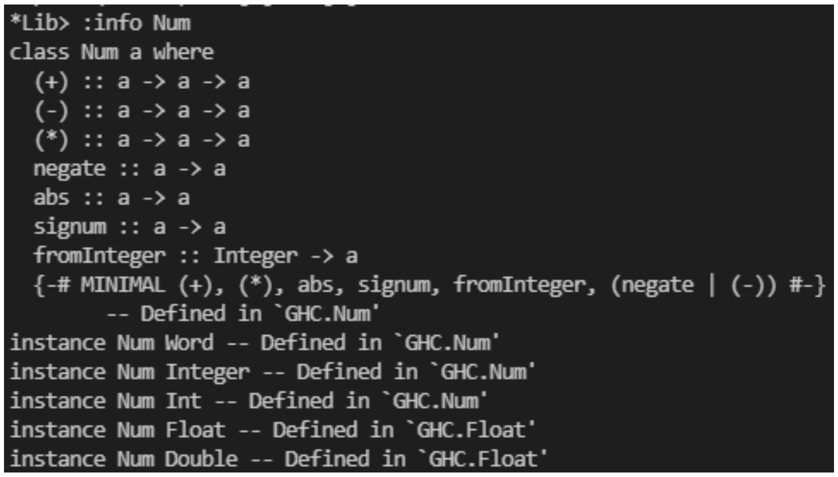
i.e. we can use any of the Eq operators with any of the types listed (Bool, Char, String, etc..)

Basic TYPECLASSES :

- Eq
- Ord
- Show
- Read
- Enum
- Bounded
- Num

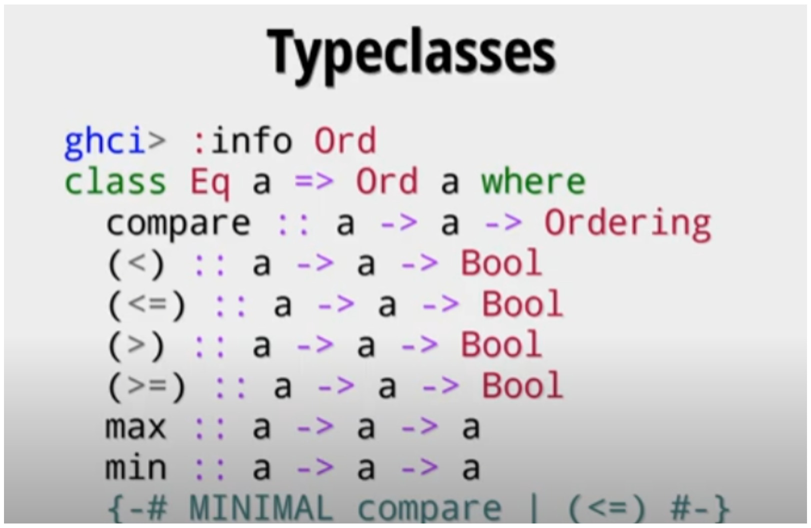
Eq => Ord ; (=>) bind operator. This means that in order to have an instance of the Ord class for any of the types, you already have to have an instance of Eq.

- Can be read as "assuming the constraint Eq holds, the methods in Ord have the specified types"
- Everything before the => is called a CLASS CONSTRAINT



```
*Lib> :info Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
        -- Defined in `GHC.Num'
instance Num Word -- Defined in `GHC.Num'
instance Num Integer -- Defined in `GHC.Num'
instance Num Int -- Defined in `GHC.Num'
instance Num Float -- Defined in `GHC.Float'
instance Num Double -- Defined in `GHC.Float'
```

^^ those are the functions defined within the class Num

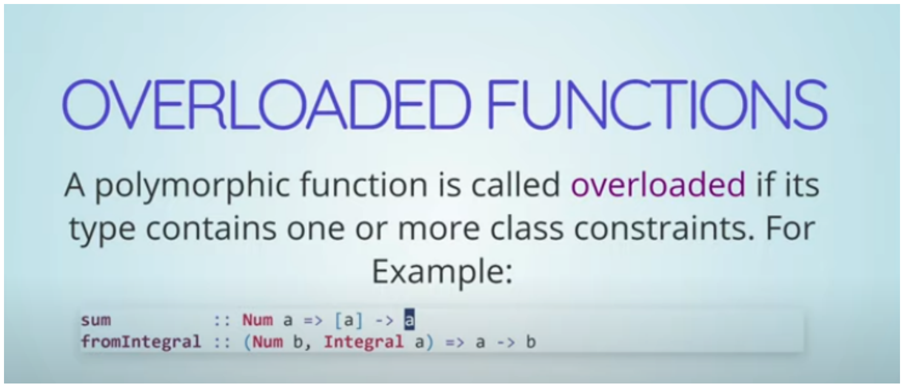


Typeclasses

```
ghci> :info Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  {-# MINIMAL compare | (<=) #-}
```

In order for a type to be in the Ord type class, it has to be in the Eq type class, so everything that can be ordered, can also be checked for equivalence

Overloaded functions



OVERLOADED FUNCTIONS

A polymorphic function is called overloaded if its type contains one or more class constraints. For Example:

```
sum          :: Num a => [a] -> a
fromIntegral :: (Num b, Integral a) => a -> b
```

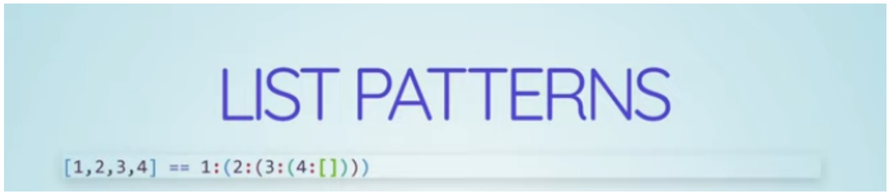
'sum' for example, can only be used with Num type class, but still polymorphic for instances defined with Num class.

Another example:

fromIntegral :: (Integral a, Num b) => a -> b

> for all types `a` and `b`,
>
> > if `a` is an instance of `Integral` and `b` is an instance of `Num`,
> >
> > then `fromIntegral` can take an `a` and produce a `b`.

This function converts a value of type `a` (which is an `Integral` type) to the `b` type (which is an instance of the more general `Num` class). E.g. you cannot add the integer `1` to the float `2` in Haskell without converting the former:



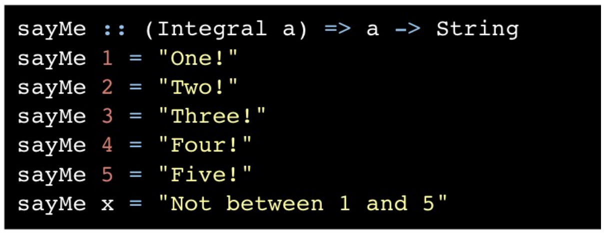
LIST PATTERNS

```
[1,2,3,4] == 1:(2:(3:(4:[])))
```

Lists actually look like the above

Pattern matching

Pattern matching means using definitive inputs that produce an outcome. I.e you can pattern match definitions of functions When this specific input gets passed into the Function, it pattern matches, to produce an output. Take the following as an example:



```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

http://learnyouahaskell.com/syntax-in-functions

Lambda expressions:

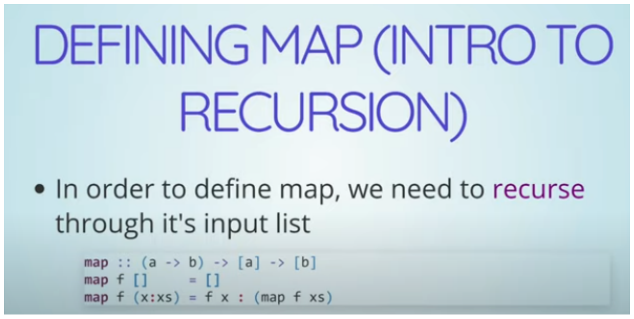
Functions without a name or "anonymous" functions
Expressed as shown below:

```
addOneList xs = map(\x -> x +1) xs
```

The function within 'map' is the anonymous func

RECURSION:

Using the function you are defining within itself.
For every recursive definition you NEED a base case. This means that at one point, the function must define some sort of behavior that does not use itself in the definition to end the recursive call. Without a base case, the function would call itself recursively indefinitely.



DEFINING MAP (INTRO TO RECURSION)

- In order to define map, we need to recurse through it's input list

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : (map f xs)
```

```
Map f [] = []              ---->      BASE CASE
Map f(x:xs)               -----> Recursive portion
```

Example :

```
F:    add x = x + 1

x:xs = [1,2,3]

map f (x:xs) = add [1] : map add [2,3]
            =add [1] : add[2] : map add [3]
            =add [1] : add[2] : add[3] : map add [].    ---This is why a base case is needed, if there was no definition
            'map f [] = [] '
                                                           the function would not stop calling itself.
            =add [1] : add[2] : add[3] : []
            =add [1] : add[2] : [4] : []
            =add [1] : [3] : [4] : []
            = [2] : [3] : [4] : []
            =[2,3,4]
```

** remember colon operator adds an element to the beginning of a list.
** Pattern matches to one of the definitions of map at every call of itself, until a base case is reached