:) Haskell → static type system i.e. the type of every expression is known at compile time which leads to safer code. ∴ everything in Haskell has a type.

↓

Haskell also has type inference.

↓

Type is a kind of a label that every expression has and it tells us the category of things that expression fits in.

↓

:t cmd. can be used to identify the type of an expression in ghci.

↓

:: is read out as 'has type of'

↓

Each type has its own type.

:) Functions also have types. We can choose to give explicit type declaration to functions when writing them. This is considered good practice except when writing very short functions.

for example : list comprehension that filters a string so that only caps remain.

maps string to string

```
removeNonUpperCase :: [Char] -> [Char]
removeNonUpperCase st = [c | c <- st, c `elem` ['A'..'Z']]
```

↓

can also use String instead of [Char]

for multiple parameters : function-name :: param1 → param2 → ... → return_type

Note: If we want to give our function a type declaration but are unsure as to what it should be, we can just always write the function w/o first and then check it with :t, since functions are expressions too.

→ some common data types in Haskell: i) Int : bounded,

(ii) Integer : unbounded Int → meaning it can be used to represent very big no.s

↓

like really big

(iii) Float : real floating point with single precision

(iv) Double : real floating point with double precision.

(v) Bool

(vi) Char

(vii) Tuple : are types but are dependent on length as well as the types of their components, so there is theoretically an ∞ no. of tuple types.

↓

even empty tuple is a type

→ functions that employ type variables are called polymorphic functions.

→ fst : fn. that returns the first component of a pair where pair is represented by a tuple.

•) Typeclass : sort of interface that defines some behaviour. If a type is a part of a typeclass, that means that it supports and implements the behaviour the typeclass describes.

↓ to define constraints over functions.

↓

Typeclasses are not like classes from OOP, rather can be understood better as JAVA interfaces.

↓

type signature of == function

Note: In Haskell, if a fn. comprises only of special characters, then it is considered infix by default.

class constraint ↖

$$(==) :: (Eq\ a) \Rightarrow a \rightarrow a \rightarrow Bool$$

↓

can be read as : the equality function takes any two values that are of the same type and returns a bool, given the type of those two values must be a member of the Eq. class. (the class constraint)

Eq. class → provides an interface for testing for equality. Any type where it makes sense to test for equality between two values of that type should be a member of the Eq. class. All standard Haskell types except IO and functions are part of the Eq. typeclass.

↓

$$elem :: (Eq\ a) \Rightarrow a \rightarrow [a] \rightarrow [Bool]$$

↓

↓

Uses '==' over a list to check whether some value we're looking for is in it.

→ some basic typeclasses : (i) Eq : types that support equality testing. The functions that its members support are (==) and (/=)

(==) → equal to

(/=) → not equal to

(ii) Ord : types that have an order → all types covered till now except fns. are part of Ord.

↓

supports <, >, <=, >=

↓

Compare function takes two Ord members of same type and returns an ordering

↓

To be a member of Ord, the type should be a member of Eq.

(iii) Show : members can be represented as Strings. All types discussed till now except functions are part of Show.

↓

most used function : show

(iv) * Read : opposite typeclass of Show. The read function takes a string representation of a type and returns the data corresponding to the type

↑

↓

this is only possible if there is a way for the program to infer the type based on some operation or context.

↓

for ex. read "4" - 2 >> 2

read "4" >> error → This can be countered by using explicit type annotations. Type annotations are a way of explicitly saying what the type of an expression should be.

↓

This can be done by adding '::' at the end of the expression and specifying a type.

```
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

(v) Enum: sequentially ordered members i.e. they can be enumerated.

↓

Advantage: its types can be used in list ranges.

↓

because of the defined successors and predecessors, succ and pred functions also work.

(), Bool, Char, Ordering, Int, Integer, Float and Double.

(vi) Bound : member types have a lower and upper bound.

↓

tuples are also part of Bound typeclass if its components are in it.

↓

maxBound, minBound

↓

type : (Bounded a) ⇒ a

↓

polymorphic constants i.e.

```
maxBound :: char
maxBound :: Int
maxBound :: Bool
```

(vii) Num typeclass contains all types that have the ability to act as numbers.

↓

whole no.s are polymorphic constants. They can act like any type that's a member of the Num typeclass.

```
20 :: Int
20 :: Integer
```

constraint

↓

examining the : t (*) : (*) :: (Num a) ⇒

a → a → a → output

two parameters

(viii) Integral : Int, Integer

(ix) Floating : Float, Double

fromIntegral :: (Numb, Integral a) ⇒ a → b

↓

takes an integer and returns a more general no.

↓

useful when we need integral and floating point numbers to work together.

has several constraints

↓

separated by commas, completely valid.

type definition

function name :: constraint ⇒ parameters → return type.

↓

good practice to define as much as possible