Chapter 6 : Typeclasses.

*) a declaration of a type defines how the type in particular is created, a declaration of a typeclass defines how a set of types are consumed or used in computations. Typeclasses are a means of ad hoc polymorphism., because typeclass code is dispatched by type.

*) Typeclasses allow us to generalize over a set of types in order to define and execute a standard set of features for these types. for example, the ability to test values for equality is useful, and we'd want to be able to use that fn. for data of various types.

*) This goes on to say that we don't need separate equality functions for each different type of data; as long as our datatype implements, or instantiates, the Eq typeclass., we can use the std. functions.

*) similarly, all the numeric literals and their various types implement 'Num' typeclass., which happens to define a standard set of operations that can be used with any type of numbers.

*) Having an instance of a typeclass can be understood, as there is code that defines how the values & functions from that typeclass work for that type. When we use a typeclass method with one of the types that has such an instance,

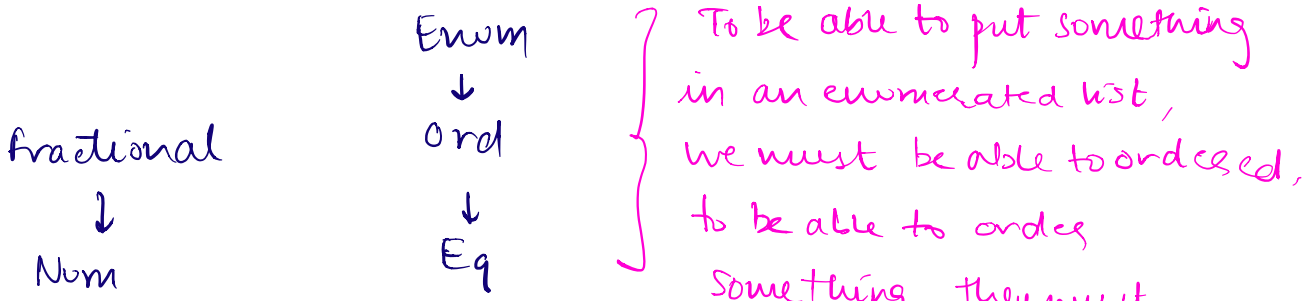
the compiler looks up the code that dictates how the function works for that type.

→ Bool

*) The information for any type includes the data declaration and which typeclasses it already has instances of.

*) Read typeclass needs to be avoided seriously. =

*) Typeclasses have hierarchy of sorts.



→ Eq typeclass : in Haskell, equality is implemented with Eq typeclass.

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

→ This tells us that we have an Eq typeclass that has 2 functions i.e. equality & inequality.

Applying == to Integer will bind 'a' to Integer

↓ can also be read as

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

→ These functions can be used for any type that implements Eq.

(*) The type of a is usually set by the leftmost occurrence and can't change in the signature Eq a ⇒ a → a → Bool

↓ Looking at 2 tuple

2 tuple has 2 elements, not necessarily of the same type. The equality of this type is defined over the equality of its elements separately i.e. the elements should individually implement the Eq typeclass.

(*) Typeclass deriving : means that we don't have to manually write instances of these typeclasses for each new datatype, we create.

→ Writing typeclass instances : done through Eq typeclass for practice.

*) A typeclass can be investigated by looking at the Hackage documentation for that typeclass. These documentations will have information about the methods that need to be defined to have a valid Eq. instance.

*) for Eq, either of (==) or (/=) will suffice, since one is the complement of the other. The option still exists to provide the programmer an option to do anything clever if needed., that makes equality checking faster for the particular datatype.

*) Unlike other languages, Haskell does not provide universal stringification (show/print) or equality (value or pointer equality) as this is not always sound or safe, regardless of what programming language we're using.

*) Keep your type class instances for a type in the same file as the type.

```
: data Trivial = Trivial

instance Eq Trivial where
  Trivial == Trivial = True
```

instance — The keyword to begin the declaration of the type class instance.

Eq — The typeclass the instance is providing

Trivial — the typeclass the instance is being provided for

where — terminates the initial declaration and beginning of the instance, this is followed by the methods being implemented.

```
(==) Trivial Trivial
  = True.
```

Infix implementation of the operator, with the arguments and then followed by the result separated by '='.

↓

Prefix based implementation could also have been done.

```
: data DayOfWeek =
    Mon | Tues | Weds | Thurs | Fri | Sat | Sun
```

we can't print these, since we didn't provide a

Show instance

```
: data Date =
    Date DayOfWeek Int
```

```
instance Eq DayOfWeek where
  (==) Mon Mon = True
  (==) Tues Tues = True
  (==) Wed Wed = True
  (==) Thur Thur = True
  (==) Fri Fri = True
  (==) Sat Sat = True
  (==) Sun Sun = True
  (==) _ _ = False
```

```
instance Eq Date where
  (==) (Date weekDay
        dayOfMonth)
       (Date weekDay'
        dayOfMonth')
    = weekDay == weekDay'
      && dayOfMonth ==
         dayOfMonth'
```

*) Take special care to avoid writing partial functions

↓

functions that don't handle all the possible cases, so there are possible scenarios in which we haven't defined any way for the code to evaluate.

↓

we can switch on the warning flags on the compiler to get more information about such cases. for example -Wall

↓

This is done in REPL using (:set -Wall)

↓

Better to have an unconditional case to match to

everything as the fall to case.

*) providing Eq constraints might be needed for types that accept arguments, and then the functions defined on the polymorphic arguments would require for the arguments on the basis of which the current type is being built for can be constrained. for example

constraint for type variable being employed in the instance derivation.

```
data Identity a = Identity a

instance Eq a => Eq (Identity a) where
  (==) (Identity v) (Identity v') = v == v'
```

Doing this Haskell ensures we don't attempt to check equality with values that don't have an Eq instance at compile time.

→ exercises for writing instance for Eq typeclass for datatypes.

a)
```
data TwoIntegers = Two Integer Integer

instance Eq TwoIntegers where
  (==) (Two i1 i2) (Two i1' i2') = ((i1 == i1') && (i2 == i2'))
```

b)
```
data StringOrInt =
    TisAnInt Int
  | TisAString String
```

```
instance Eq StringOrInt where
  (==) (TIsAnInt a) (TIsAnInt a') = (a == a')
  (==) (TIsAString s) (TIsAString s') = (s == s')
  (==) _ _ = False
```

more on the lines of function ↑

c) `data Pair a = Pair a a`

only present once as type variable and then used as many places as required

```
instance Eq a => Eq (Pair a) where
  (==) (Pair a1 a2) (Pair a1' a2') = (a1 == a1') && (a2 == a2')
```

d)
```
data EitherOr a b =
    Hello a
  | Goodbye b

instance (Eq a, Eq b) => Eq (EitherOr a b) where
  (==) (Hello c) (Hello c') = (c == c')
  (==) (Goodbye d) (Goodbye d') = (d == d')
  (==) _ _ = False
```

→ Integral typeclass

```
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
```

typeclass constraint, meaning that any type that implements this typeclass must already have instances for Real & Enum typeclasses.

↓

and hence employ methods of both these typeclasses.

↓

Real typeclass itself requires an instance of Num.

*) Since Real cannot override the methods of Num, this typeclass inheritance is only additive and ambiguity problems are conveniently avoided.

*) The typeclass that primarily provides the operation, invariantly is the implicit constrained applied on that operation. The generalisation lazy specification happens if the definition of the function is supportive.

→ Type defaulting typeclasses: When we have a typeclass constrained (ad hoc) polymorphism, and need to evaluate it, the polymorphism must be resolved to a specific concrete type. This concrete type should have instances to all required typeclass.

*) The use of polymorphic values without the ability to infer a specific type and no default rule will cause GHC to complain about an ambiguous type.

*) We can declare more specific (monomorphic) functions from more general (polymorphic) functions.

```
let add = (+) :: Integer → Integer → Integer
```

*) The actual type is the type we've provided and the expected type is the type that the compiler expects.

The expected type can be more concrete, but the actual type cannot be more concrete than expected.

→ Ord typeclass : Ord is constrained by Eq

Ordering value : LT, GT, EQ , compare function

↓

works for all the types that implement the ord typeclass., and returns an ordering value.

↓

The order in which the data value constructors are provided for a type, decides their weightage. for example True > False, This can be attributed to the way Bool is defined : data Bool = True | False.

*) If a partially applied fn is not stored or provided a reference or just left to the REPL to print, then it will throw an error, since `→` does not have an instance of show, and hence can't be printed on invoking of print function.

```
print :: Show a ⇒ a → IO ()
```

*) writing instances is one of the most necessary skills in Haskell.

→ Ord instances: Expressing that Friday is always the best day

```
instance Ord DayOfWeek where
    compare Fri Fri = EQ
    compare Fri _   = GT
    compare _   Fri = LT
    compare _   _   = EQ
```

*) Make sure that Ord instances agree with Eq instances i.e if x == y, then compare x y should return EQ.

*) Also, we want our Ord instances to define a sensible total order. This is partially done by covering all cases and not writing partial instances.

*) Ord implies Eq, hence putting Ord constraint on functions in place of Eq causes no issues, because if a is ordered it is Eq anyway, but it would limit the types that can use this fn

→ Enum : typeclass that covers types and enumerable, therefore has known predecessors and successors.

```
enumFrom :: a → [a]
```
: This fn. takes a starting value and build a list with the succeeding items of the same type.

```
enumFromThenTo 1 10 100
```
→ arithmetic progression based list where first, second and last element are returned as a list.

→ Show : provides for the creation of human-readable string representations of structured data.

*) Printing and side effects : The print function is not just applied to the arguments that are in scope but also acted to affect the world outside its scope in some way, namely by showing us the result on the screen. This is known as side effect, i.e. a potentially observable result apart from the value the expression evaluates to.

↓

Haskell manages effects by separating effectful computations from pure computations in ways that preserve the predictability and safety of function evaluation.

↓

effect-bearing computations themselves become more composable and easier to reason about, i.e. makes it relatively easy to reason and predict the result of our functions.

↓

print is defined in the Prelude standard as a function to output a value of any printable type to the std. o/p device.

↓

instances of the class Show

↓

The return type of print is IO (). This implies an IO action that returns a value of the type ().

↓

An IO action simply refers to an action, when performed has side effects, including reading from i/p and printing to the screen. and will contain a return value.

↓

Here, the () denotes an empty tuple, referred to as unit. Unit is a value, and also a type that has this one inhabitant, that essentially represent nothing.

↓

Printing a string to the terminal doesn't have a meaningful return value. But an IO action, like any expression in Haskell, can't return nothing; it must return something. So we simply use unit to represent the return value at the end of our IO action.

↓

That is, the print function will first do the IO action of printing the string to the terminal and then complete the action, marking an end to the execution of the function and delimination of the side effects, by returning (). Does not print the () to the screen, but it is implicitly present.

↓

IO String is more of a means of producing a string, which may require performing side effects along the way before we get any string value.

myval :: String ⇒ a String value

ioString :: IO String ⇒ method or means of obtaining a value of type String / by performing side effects

" Getting a string value, and on the way, performing this side effects based operations like IO "

*) Working with Show → Invoking the Show typeclass also invokes its methods, specifically a method of taking our values and turning them into values that can be printed to the screen.

↓

A minimal implementation of an instance of Show only requires that show or showsPrec be implemented. for example.

Typeclass is just a blueprint.

↑

Functionality should be called via some instance

```
data Mood = Blah.

instance Show Mood where
    show _ = "Blah"
```

↓

Mostly, we'll be deriving the Show typeclass, and very rarely writing our own.

→ Instances are dispatched by type

(Typeclasses are dispatched by types)

*) Typeclasses are defined by the set of operations and values, all instances will provide. Typeclass instances are unique pairings of the typeclass & the type.

They define the ways to implement the typeclass methods for that type.

```
class Numberish a where
    fromNumber :: Integer → a
    toNumber   :: a → Integer
```

newtype : special case of data declarations, wherein it permits only one constructor and only one field.

```
newtype Age =
    Age Integer
    deriving (Eq, Show)

instance Numberish Age where
    fromNumber n = Age n
    toNumber (Age n) = n
```

The class definition of Numberish doesn't define any terms or code, we can compile & execute, only types are doing that.

↓

The code lives in instances for Age & Year.

```
newtype Year =
    Year Integer
    deriving (Eq, Show)

instance Numberish Year where
    fromNumber n = Year n
    toNumber (Year n) = n
```

```
sumNumberish :: Numberish a ⇒ a → a → a
sumNumberish a a' = fromNumber summed
    where integerOfA      = toNumber a
          integerOfAPrime = toNumber a'
          summed =
            integerOfA + integerOfAPrime
```

when sumNumberish (Age 10) (Age 10) is called, it used the definition in Age instance since it knew the type of our arguments.

↓

After the first parameter is applied to a value of type Age, it knows that all other occurences of type Numberish a ⇒ a must be Age.

↓

In some cases it is absolutely possible that we're not providing enough information to Haskell for it to identify a concrete.

```
class Numberish a where
    fromNumber :: Integer → a
    toNumber :: a → Integer
    defaultNumber :: a

newType Age =
    Age Integer
    deriving (Eq, Show)

instance Numberish Age where
    fromNumber n = Age n
    toNumber (Age n) = n
    defaultNumber = Age 65
```

```
newType Year =
    Year Integer
    deriving (Eq, Show)

instance Numberish Year where
    fromNumber n = Year n
    toNumber (Year n) = n
    defaultNumber = Year 1988
```

```
fromNumber 10 :: Age.
```

↑ to use fromNumber, we need to specify the type

↑ same goes with the fromNumber line

↑

(*) Now, if we were to refer to defaultNumber, there is no way for Haskell to know what we need. This can be rectified by using type assertion. Type assertion to dispatch, or specify, what typeclass instance we want our defaultNumber from.

→ Concrete types imply all the typeclasses they provide

```
add :: Int → Int → Int
add x y = x + y

addweird :: Int → Int → Int
addweird x y =
    if x > 1
    then x + y
    else x
```

No need to mention constraints here. They will typecheck. This is because the Int type has the typeclasses Num, Eq and Ord implemented, and hence works for their function definitions.

There are some caveats to keep in mind here when it comes to using concrete types. One of the nice things about parametricity and typeclasses is that you are being explicit about what you mean to do *with* your data which means you are less likely to make a mistake. `Int` is a big datatype with many inhabitants and many typeclasses and operations defined for it — it could be easy to make a function that does something unintended. Whereas if we were to write a function, even if we had `Int` values in mind for it, which used a polymorphic type constrained by the typeclass instances we wanted, we could ensure we only used the operations we intended. This isn't a panacea, but it can be worth avoiding concrete types for these (and other) reasons sometimes.

→ div mod → (quotient, rem)

print → show

→ cht f x y = (f x) == y

→ arith :: Numb ⇒ (a → b) → Integer → a → b

arith f x y = f y

---

*) Typeclass inheritance is when a typeclass has a superclass. It is a way of expressing that a typeclass requires another typeclass to be available for a given type before we can write an instance. example:

```
class Num a ⇒ Fractional a where
  (/) :: a → a → a
  recip :: a → a
  fromRational :: Rational → a
```

Here Fractional inherits from Num. This means that if any type wants to inherit Fractional, then that type

must already have an instance of NVM.