Background – Type Classes (1B)

Young Won Lim
6/23/18

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Based on

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

Typeclasses and Instances

typeclasses are like interfaces

defines some behavior
- comparing for *equality*
- comparing for *ordering*
- *enumeration*

instances of that **typeclass**

types possessing such behavior

such *behavior* is defined by

- **function definition**
- function **type declaration** only

a function definition

```
(==) :: a -> a -> Bool
x == y = not (x /= y)
```

- **a type declaration**

a function type declaration

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

- **a type declaration**

A function definition can be **overloaded**

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Typeclasses and Type

typeclasses are like interfaces

defines some behavior
- comparing for *equality*
- comparing for *ordering*
- *enumeration*

instances of that **typeclass**
types possessing such behavior

```
class AAA bbb where
    func1 :: a -> b -> c
    func2 :: b -> c -> a
```

```
instance AAA BBB where
    func1 definition
    func2 definition
```

a **type** is an **instance** of a **typeclass** implies

the function types declared by the **typeclass**
are defined (implemented) in the **instance**
so that the functions can be used,
which the **typeclass** defines with that **type**

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Instance Example

the Eq typeclass

defines the functions **==** and ***/=***

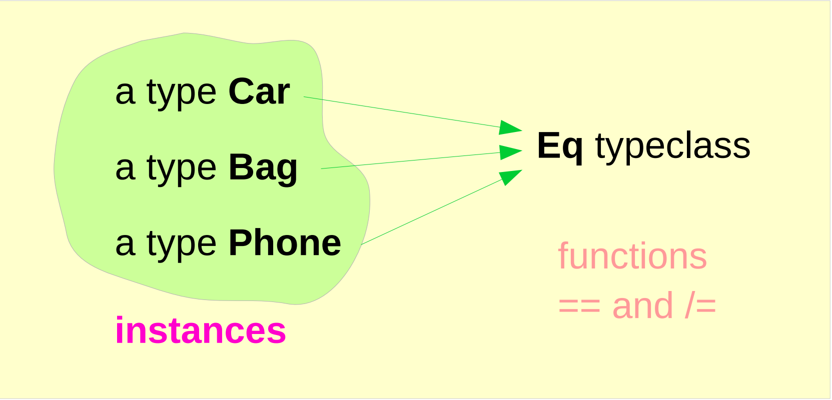
a type Car

comparing two cars c1 and c2 with the equality function ==

The Car type is an **instance** of Eq **typeclass**

Instances : various **types**

Typeclass : a group or a class of these similar types



http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

TrafficLight Type Example (1)

```
class Eq a where
    (==) :: a -> a -> Bool          - a type declaration
    (/=) :: a -> a -> Bool          - a type declaration
    x == y = not (x /= y)           - a function definition
    x /= y = not (x == y)           - a function definition
```

```
data TrafficLight = Red | Yellow | Green
```

```
instance Eq TrafficLight where
    Red    == Red     = True
    Green  == Green   = True
    Yellow == Yellow  = True
    _ == _            = False
```

```
ghci> Red == Red
True
ghci> Red == Yellow
False
ghci> Red `elem` [Red, Yellow, Green]
True
```

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

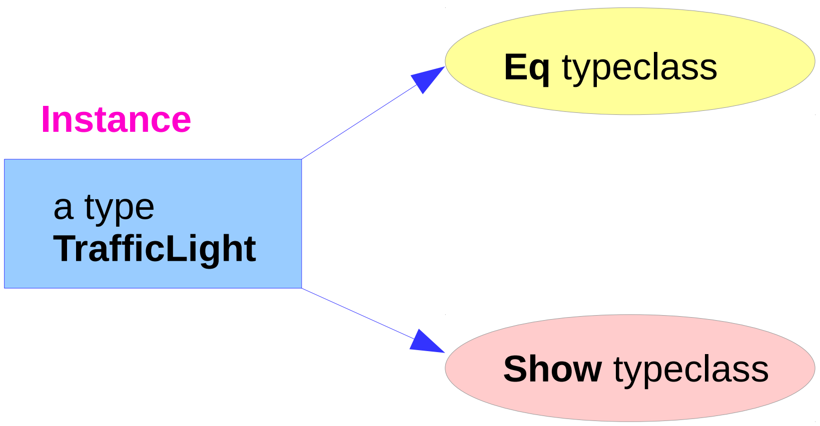
TrafficLight Type Example (2)

```
class Show  a where
    show :: a  -> String
    * * *
```

- a type declaration

```
data TrafficLight = Red | Yellow | Green
```

```
instance Show TrafficLight where
     show Red = "Red light"
     show Yellow = "Yellow light"
     show Green = "Green light"
```




```
ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

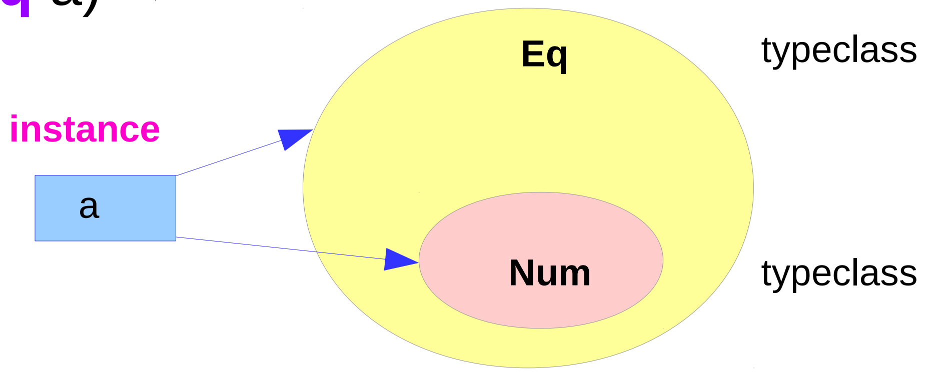
http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Class Constraints

```
class (Eq a) => Num a where
  ...
```

```
class Num a where
  ...
```

(Eq a) =>



Num : a subclass of **Eq**

class constraint on a class declaration

an instance of **Eq**
before being an instance of **Num**

the required function bodies can be defined in
- the class declaration
- an instance declarations,

we can safely use == because a is a part of **Eq**

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Class Constraints : class & instance declarations

class constraints in **class declarations**

to make a typeclass a **subclass** of another typeclass

subclass

```
class (Eq a) => Num a where
    …
```

class constraints in **instance declarations**

to express **requirements** about the contents of some type.

requirements

```
instance (Eq x, Eq y) => Eq (Pair x y) where
   Pair x0 y0 == Pair x1 y1 = x0 == x1 && y0 == y1
```

http://cmsc-16100.cs.uchicago.edu/2016/Lectures/07-type-classes.php

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Class constraints in instance declaration examples

```
instance (Eq m) => Eq (Maybe m) where
    Just x     ==  Just y          =     x == y   ⇐ Eq m
    Nothing  ==  Nothing           =     True
             _ == _                =     False
```

```
instance (Eq x, Eq y) => Eq (Pair x y) where
    Pair x0 y0 == Pair x1 y1 = x0 == x1 && y0 == y1
             ↑                    ↑            ↑
        Eq (Pair x y)            Eq x         Eq y
```

Derived instance

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

A Concrete Type and a Type Constructor

a : a concrete type

Maybe : not a concrete type
: a type constructor that takes one parameter
produces a concrete type.

Maybe a : a concrete type

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Instance of **Eq**

```
data TrafficLight = Red | Yellow | Green
```

to define our own type (defining a new data type)
allowed values are Red, Yellow, and Green
no class (type) instances

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)

instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

class :
defining new **typeclasses**

instance :
making types instances of a **typeclasses**

http://learnyouahaskell.com/making-our-own-types-and-typeclasses

Instance of **Show**

```
instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

ghci> Red **==** Red	◄	**instance Eq TrafficLight**
True		
ghci> Red **==** Yellow	◄	**instance Eq TrafficLight**
False		
ghci> Red `**elem**` [Red, Yellow, Green]	◄	**instance Eq TrafficLight**
True		
ghci> [Red, Yellow, Green]	◄	**instance Show TrafficLight**
[Red light,Yellow light,Green light]		

http://learnyouahaskell.com/making-our-own-types-and-typeclasses

Instance Maybe m

```
instance Eq Maybe where
   ...
```

```
instance Eq (Maybe m) where
   Just x == Just y = x == y
   Nothing == Nothing = True
   _ == _ = False
```

```
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

Maybe is not a concrete type
Maybe m is a concrete type

all types of the form **Maybe m**
to be part of the **Eq** typeclass,

but only those types where the **m**
(what's contained inside the **Maybe**)
is also a part of **Eq**.

http://learnyouahaskell.com/making-our-own-types-and-typeclasses

Polymorphism in Haskell

Haskell's combination of

- purity
- higher order functions
- parameterized algebraic data types
- typeclasses

allows us to implement **polymorphism** on a much higher level

Types in Haskell

- don't have to think about types belonging to a big hierarchy of types
- think about what the types can act like
- and then connect them with the appropriate typeclasses

Example:

An Int can act like a lot of things

- like an equatable thing,
- like an ordered thing,
- like an enumerable thing, etc.

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Open Typeclasses

Typeclasses are **open**:
- can define our own data type,
- can think about what it can act like
- can **connect** it with the **typeclasses** that define its behaviors.

the **type declaration** of a **function**
allows us to know a lot about a **function**

can define **typeclasses** that define behavior
that is very general and abstract.

Example:
typeclasses that define operations for seeing if two things are equal
or comparing two things by some ordering.
- those are very **abstract** and elegant behaviors,
- those are not anything very **special**
  because these operations are most common

Act
Behavior
Operation

Define
Connect

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Functors, Applicatives, Monads

functors: you apply a function to a **wrapped** value

applicatives: you apply a **wrapped** function to a **wrapped** value

monads: you apply a function that returns a **wrapped** value, to a **wrapped** value

functors: using **fmap** or **<$>**

applicatives: using **<*>** or **liftA**

monads: using **>>=** or **liftM**

https://softwareengineering.stackexchange.com/questions/303472/what-is-the-purpose-of-wrapped-values-in-haskell

Functors

Functors use the **fmap** or <$> functions

```
fmap :: Functor f => (a -> b) -> f a -> f b
<$>  :: Functor f => (a -> b) -> f a -> f b
```

This takes a function and applies to to the wrapped elements

```
fmap (\x -> x + 1) (Just 1)      -- Applies (+1) to the inner value, returning (Just 2)
fmap (\x -> x + 1) Nothing       -- Applies (+1) to an empty wrapper, returning Nothing

fmap (\x -> x + 1) [1, 2, 3]     -- Applies (+1) to all inner values, returning [2, 3, 4]
(\x -> x + 1) <$> [1, 2, 3]      -- Same as above [2, 3, 4]
```

https://softwareengineering.stackexchange.com/questions/303472/what-is-the-purpose-of-wrapped-values-in-haskell

Applicatives

Applicatives use the **<*>** function:

```
<*> :: Applicative f => f (a -> b) -> f a -> f b
```

This takes a wrapped function and applies it to the wrapped elements

```
(Just (\x -> x + 1)) <*> (Just 1)        -- Returns (Just 2)
(Just (\x -> x + 1)) <*> Nothing         -- Returns Nothing
Nothing <*> (Just 1)                     -- Returns Nothing
[(*2), (*4)] <*> [1, 2]                  -- Returns [2, 4, 4, 8]
```

https://softwareengineering.stackexchange.com/questions/303472/what-is-the-purpose-of-wrapped-values-in-haskell

Monads – return

There are two relevant functions in the **Monad typeclass**:

```
return :: Monad m => a -> m a
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

The return function takes a raw, unwrapped value,
and wraps it up in the desired monadic type.

```
makeJust :: a -> Maybe a
makeJust x = return x

let foo = makeJust 10                -- returns (Just 10)
```

https://softwareengineering.stackexchange.com/questions/303472/what-is-the-purpose-of-wrapped-values-in-haskell

Monads – bind

The bind function lets you
temporarily unwrap the inner elements of a **Monad**
and pass them to a function that performs some action
that wraps them back UP in the same monad.

This can be used with the return function in trivial cases:

```
[1, 2, 3, 4] >>= (\x -> return (x + 1))     -- Returns [2, 3, 4, 5]
(Just 1) >>= (\x -> return (x + 1))         -- Returns (Just 2)
Nothing >>= (\x -> return (x + 1))          -- Returns Nothing
```

https://softwareengineering.stackexchange.com/questions/303472/what-is-the-purpose-of-wrapped-values-in-haskell

Monads – a binding operand

functions to chain together that don't require to use **return**.

```
getLine :: IO String                -- return String type value as a result
putStrLn :: String -> IO ()
```

function call examples

```
getLine >>= (\x -> putStrLn x)      -- gets a line from IO and prints it to the console
getLine >>= putStrLn                -- with currying, this is the same as above
```

https://softwareengineering.stackexchange.com/questions/303472/what-is-the-purpose-of-wrapped-values-in-haskell

Monads – a chain of functions

functions to chain together that don't require to use **return**.

```
getLine :: IO String                -- return String type value as a result
putStrLn :: String -> IO ()
read :: Read a => String -> a
show :: Show a => a -> String
```

-- composite function examples

-- reads a line from IO, converts to a number, adds 10 and prints it

```
getLine >>= (return . read) >>= (return . (+10)) >>= putStrLn . show
```

```
        String              a                 a              String -> ()
getLine  ➡  (return . read)  ➡  (return . (+10))  ➡  putStrLn . show
```

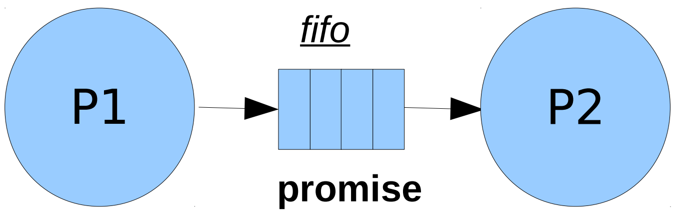
https://softwareengineering.stackexchange.com/questions/303472/what-is-the-purpose-of-wrapped-values-in-haskell

Promises and Mediators

the concept of **promises** (particularly in Javascript)

A **promise** is an **object** that acts as a placeholder
for the **result value** of an asynchronous, background **computation**,
like fetching some data from a remote service.

it serves as a **mediator**
between the asynchronous computation
and functions that need to **operate** on its anticipated **result**.



Act
Behavior
Operation

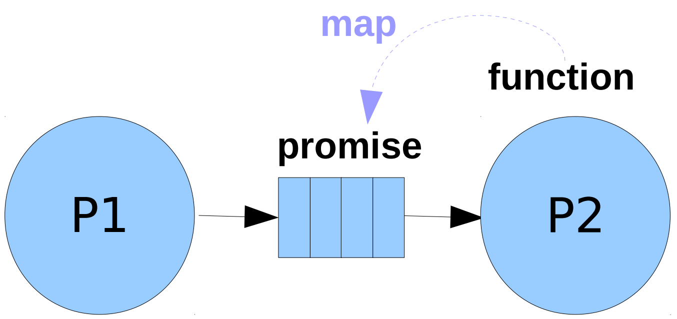
Define
Connect

https://softwareengineering.stackexchange.com/questions/303472/what-is-the-purpose-of-wrapped-values-in-haskell

Map a function over a promise

A mediator allows us to say
what **function** should apply
to the **result** of a background task,
before that task has completed.

When you **map** a **function** over a **promise**,
the value that your function should apply to
may not have been computed yet
and in fact, if there is an error somewhere
it may never be computed.



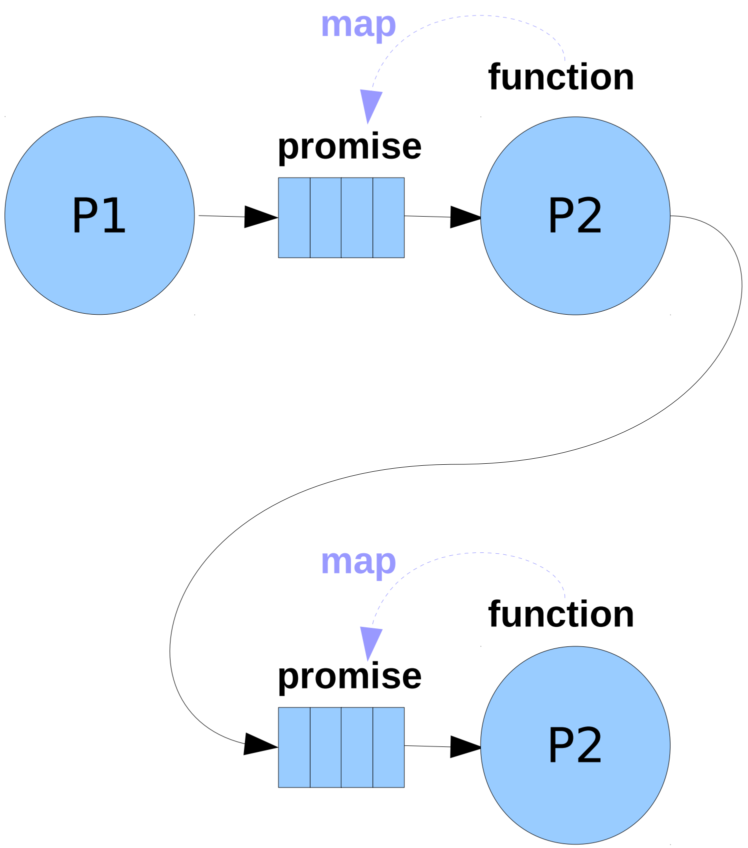
https://softwareengineering.stackexchange.com/questions/303472/what-is-the-purpose-of-wrapped-values-in-haskell

Chaining a function onto a promise

Promise libraries usually support
a **functorial/monadic** API
where you can chain a function onto a **promise**,
which produces another **promise**
that produces the **result** of applying that function
to the original **promise's** result.

the **value** of the **functor/monad** interface

Promises allow you to say
what **function** should apply
to the **result** of a background task,
before that task has completed.



https://softwareengineering.stackexchange.com/questions/303472/what-is-the-purpose-of-wrapped-values-in-haskell

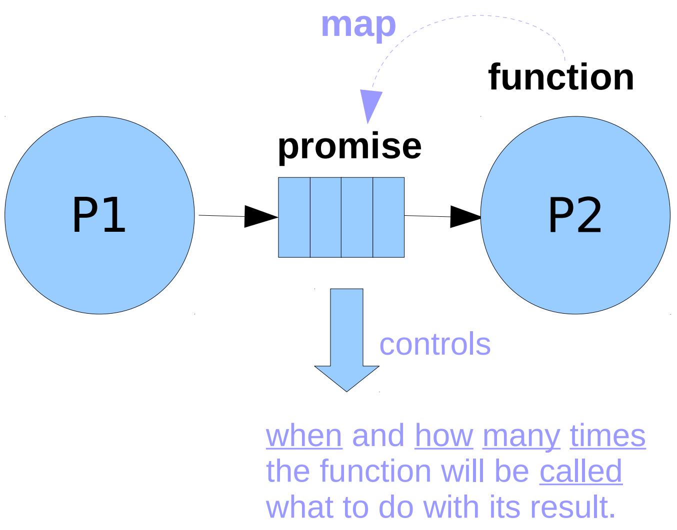
Interfaces

think **functor/applicative/monad**
as **interfaces** for **mediator objects**
that sit in between **functions** and **arguments**,
and connect them indirectly according to some policy.

The simplest way to use a function is
just to call it with some arguments;

but if you have first-class functions,
you have other, indirect options—

you can supply the function to a mediator object
that will control when and how many times the
function will be called, and what to do with its result.



https://softwareengineering.stackexchange.com/questions/303472/what-is-the-purpose-of-wrapped-values-in-haskell

Promises and Mediators

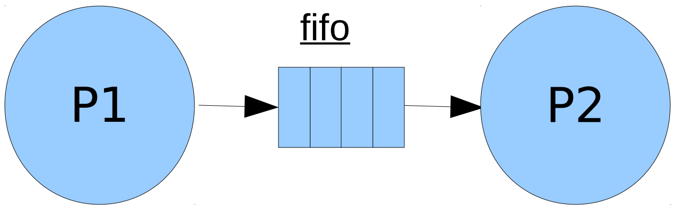
Promises are one such example:

They call the functions supplied to them

when the result of some background task is completed

The results of those functions are then handed over

to other promises that are waiting for them.



Act
Behavior
Operation

Define
Connect

https://softwareengineering.stackexchange.com/questions/303472/what-is-the-purpose-of-wrapped-values-in-haskell

General Monad - MonadPlus

Haskell's **Control.Monad** module defines a typeclass, **MonadPlus**, that enables abstract the common pattern eliminating **case** expressions.

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

```
class (Monad m) => MonadPlus m where
```

```
instance MonadPlus [] where
  mzero = []
  Mplus = (++)
```

```
instance MonadPlus Maybe where
  mzero = Nothing

  Nothing `mplus` ys  = ys
  xs      `mplus` _ = xs
```

http://book.realworldhaskell.org/read/programming-with-monads.html

General Monad - MonadPlus Laws

The class **MonadPlus** is used for monads that have a zero element and a plus operation:

```
class (Monad m) => MonadPlus m where
   mzero        :: m a
   mplus        :: m a -> m a -> m a
```

For lists, the zero value is [], the empty list. The I/O monad has no zero element and is not a member of this class.

```
m >>= \x -> mzero     =    mzero
mzero >>= m           =    mzero
```

The zero element laws:

```
m `mplus` mplus       =    m
mplus `mplus` m       =    m
```

The laws governing the mplus operator

The mplus operator is ordinary list concatenation in the list monad.

http://book.realworldhaskell.org/read/programming-with-monads.html

Functional Dependency (fundep)

```
class class Mult | a b -> c where
  (*) :: a -> b -> c
```

c is uniquely determined from **a** and **b**

.

Fundeps are not standard Haskell 98.
(Nor are multi-parameter type classes, for that matter.)
They are, however, supported at least in GHC and Hugs
and will almost certainly end up in Haskell'.

```
class class Mult where
  (*) :: a -> b -> c
```

https://wiki.haskell.org/Functional_dependencies

Eq, Ord, Show classes

Since equality tests between values are frequently used

most of your own data types should be members of **Eq**.

Prelude classes

- **Eq**
- **Ord**
- **Show**

for the convenience, Haskell has a way to declare

such "obvious" **instance definitions**

using the keyword **deriving**.

https://en.wikibooks.org/wiki/Haskell/Classes_and_types

Deriving instance example

```
data Foo = Foo {x :: Integer, str :: String}
    deriving (Eq, Ord, Show)
```

This makes **Foo** an **instance** of **Eq**

with an automatically generated definition of **==**

and also an **instance** of **Ord** and **Show**

```
data Foo = Foo {x :: Integer, str :: String}

instance Eq Foo where
   (Foo x1 str1) == (Foo x2 str2)
        = (x1 == x2) && (str1 == str2)
```

*Main> **Foo 3 "orange" == Foo 6 "apple"**

False

*Main> **Foo 3 "orange" /= Foo 6 "apple"**

True

https://en.wikibooks.org/wiki/Haskell/Classes_and_types

Deriving instance pros and cons

The **types** of **elements** inside the **data** type

must also be **instances of the class** you are deriving.

Deriving instances

- synthesis of functions for a limited set of predefined classes
- against the general Haskell philosophy :
  "built in things are not special",
- induces compact codes
- often reduces errors in coding
  (an example: an instance of Eq such that x == y
  would not be equal to y == x would be flat out wrong).

https://en.wikibooks.org/wiki/Haskell/Classes_and_types

Derivable Classes

Eq

Equality operators **==** and **/=**

Ord

Comparison operators **< <= > >=**; **min**, **max**, and **compare**.

Enum

For enumerations only. Allows the use of list syntax such as [Blue .. Green].

Bounded

Also for enumerations, but can also be used on types that have only one constructor.

Provides **minBound** and **maxBound** as the lowest and highest values that the type can take.

Show

Defines the function **show**, which converts a value into a string, and other related functions.

Read

Defines the function **read**, which parses a string into a value of the type,

and other related functions.

https://en.wikibooks.org/wiki/Haskell/Classes_and_types

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf