Background – Constructors (1A)

Young Won Lim
7/7/18

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

This document was produced by using OpenOffice.

Based on

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

Data Constructor

$$\text{data } \underbrace{\text{Color}}_{\text{Type Constructor}} = \underbrace{\text{Red | Green | Blue}}_{\text{Data Constructors values}}$$

Red is a *constructor* that contains a *value* of the type **Color**.

Green is a *constructor* that contains a *value* of the type **Color**.

Blue is a *constructor* that contains a *value* of the type **Color**.

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Variable binding examples

```
data Color = Red | Green | Blue
    deriving (Eq, Ord, Show)


pr :: Color -> String
pr x
  | x == Red   = "Red"
  | x == Green = "Green"
  | x == Blue  = "Blue"
  | otherwise  = "Not a Color"

*Main> pr Red             x ← Red
"Red"
*Main> pr Green           x ← Green
"Green"
*Main> pr Blue            x ← Blue
"Blue"
```

```
Prelude> data Color = Red | Green | Blue
deriving(Eq, Ord, Show)

Prelude> let x = Red              x ← Red
Prelude> let y = Green            x ← Green
Prelude> let z = Blue             x ← Blue

Prelude> show(x)
"Red"
Prelude> show (y)
"Green"
Prelude> show(z)
"Blue"
```

Data Constructor with Parameters

data Color = **RGB** Int Int Int

Type **Constructor** type	**Data** **Constructors** (a function returning a value)

RGB is not a value but a *function* taking three Int’s and *returning a value*

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Data Constructor with Parameters – type declaration

```
data Color = RGB Int Int Int
```

```
RGB :: Int -> Int -> Int -> Color
```

a function type declaration

RGB is a **data constructor** that is a *function*
taking three Int values as its arguments,
and then uses them to construct a new value.

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Type Constructors and Data Constructors

A **type constructor**

- a "function" that takes 0 or more types
- returns a new **type**.

Type constructors with parameters

allows slight variations in types

type **SBTree** = **BTree** String

type **BBTree** = **BTree** Bool

BTree String returns a new type

BTree Bool returns a new type

A **data constructor**

- a "function" that takes 0 or more values
- returns a new **value**.

Data constructors with parameters

allows slight variations in values

RGB 12 92 27 → #0c5c1b

RGB 255 0 0

RGB 0 255 0

RGB 0 0 255

returns a value of Color type

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Type Constructor

Consider a binary tree to store Strings

```
data SBTree = Leaf String  |  Branch String SBTree SBTree
```

Type
Constructor
type

Data
Constructors
(functions returning a value)

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Data Constructors – type declarations

Consider a binary tree to store Strings

```
data SBTree = Leaf String  |   Branch String SBTree SBTree
```

SBTree
Type
Constructor

Leaf
Data
Constructor

Branch
Data
Constructor

```
Leaf        :: String -> SBTree
Branch      :: String -> SBTree -> SBTree -> SBTree
```

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Similar Type Constructors

Consider a binary tree to store Strings

```
data SBTree = Leaf String  |   Branch String SBTree SBTree
```

Consider a binary tree to store Bool

```
data BBTree = Leaf Bool  |  Branch Bool BBTree BBTree
```

Consider a binary tree to store a parameter type a

```
data BTree a = Leaf a  |   Branch a (BTree a) (BTree a)
```

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Type Constructor with a Parameter

```
data SBTree = Leaf String  |  Branch String SBTree SBTree
data BBTree = Leaf Bool    |  Branch Bool BBTree BBTree
```

```
data BTree a = Leaf a      |  Branch a (BTree a) (BTree a)
```

a **type** **variable** **a**
as a parameter to the type constructor.

BTree has become a function.
It takes a **type** as its argument
and it returns a **new** **type**.

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

( ) : the unit type

( ) is both a **type** and a **value**.

( ) is a special **type**, pronounced “**unit**”,
has one **value** **( )**, sometimes pronounced “**void**”

the **unit type** has only one **value** which is called **unit**.

data **( )** = **( )**
( ) :: **( )** **Value** :: **Type**

the **unit type ( )**
the **void value ( )**

It is the same as the void type **void** in Java or C/C++.

https://stackoverflow.com/questions/20380465/what-do-parentheses-used-on-their-own-mean

Unit Type

a **unit type** is a type that allows *only one value* (and thus can hold *no information*).

It is the same as the void type **void** in Java or C/C++.

```
:t
Expression :: Type
```

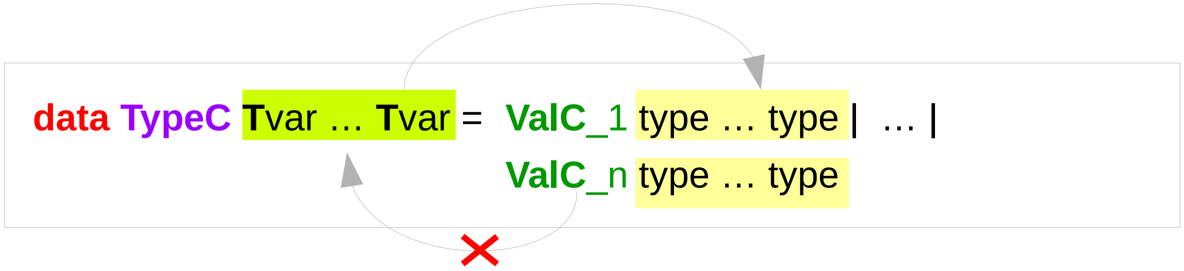
```
data Unit = Unit

Prelude> :t Unit
Unit :: Unit
```

```
Prelude> :t ()
() :: ()
```

https://stackoverflow.com/questions/20380465/what-do-parentheses-used-on-their-own-mean

Type Language and Expression Language



data TypeC Tvar … Tvar = ValC_1 type … type | … |
ValC_n type … type

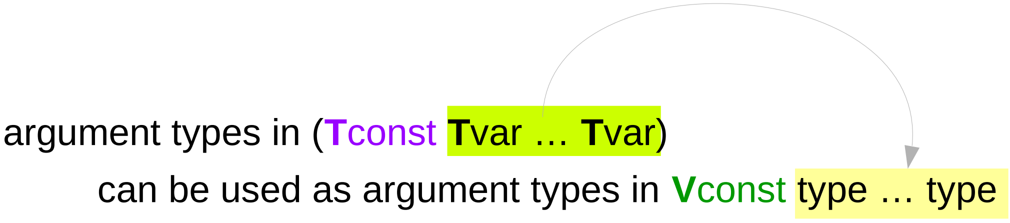
A new **datatype** declaration

TypeC (**T**ype **Cons**tructor) is added to *the type language*

ValC (**Val**ue **Cons**tructor) is added to *the expression language*

and *its pattern sub-language*

must not appear in types

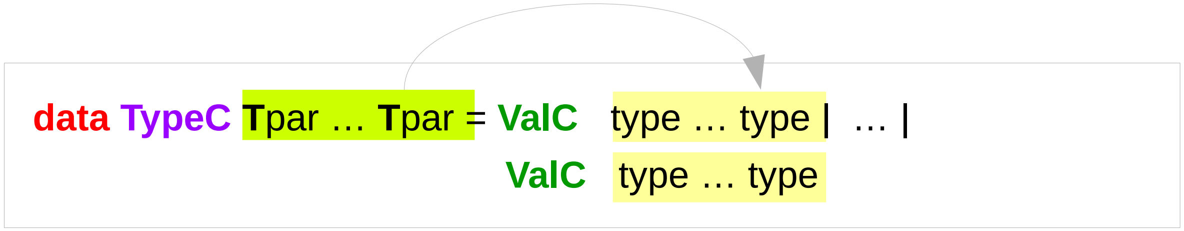


argument types in (Tconst Tvar … Tvar)

can be used as argument types in Vconst type … type

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Datatype Declaration



```
data TypeC Tpar … Tpar = ValC   type … type |  … |
                          ValC   type … type
```

A new **datatype** declaration

The keyword **data** introduces a new **datatype** declaration,

- the **new type** TypeC Tpar … Tpar
- its **values** ValC type … type | … | ValC type … type

datatype

data type

data type = data

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Datatype Declaration Examples

```
data Tree a =     Leaf |  Node (Tree a) (Tree a)

Tree              (Type Constructor)
Leaf or Node      (Value Constructor)
```

```
data  Type = Value
```

```
data ( ) =   ( )

( )      (Type Constructor)
( )      (Value Constructor)
```

the type (), often pronounced "Unit"
the value (), sometimes pronounced "void"

the type () containing only one value ()

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Type Synonyms

```
type String = [Char]

phoneBook :: [(String,String)]
```

no data constructor

```
type PhoneBook = [(String,String)]

phoneBook :: PhoneBook
```

no data constructor

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]

phoneBook :: PhoneBook
```

no data constructor

```
phoneBook =
    [("betty","555-2938")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ]
```

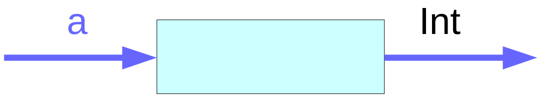
http://learnyouahaskell.com/making-our-own-types-and-typeclasses

Type Synonyms for Functions

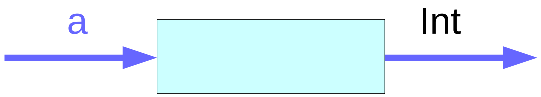
```
type Bag a = a -> Int                          no data constructor

data Gems = Sapphire | Emerald | Diamond deriving (Show)
```

a -> Int



Bag a



```
type Bag a = a -> Int
type Bag Int = Int -> Int
type Bag Char = Char -> Int
```

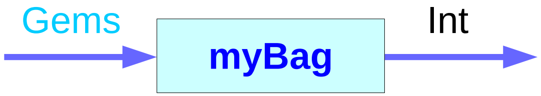
https://stackoverflow.com/questions/14166641/haskell-type-synonyms-for-functions

Type Synonyms for Functions

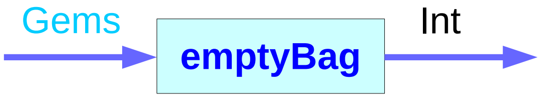
```
type Bag a = a -> Int                          no data constructor

data Gems = Sapphire | Emerald | Diamond deriving (Show)
```

myBag :: Bag Gems



emptyBag :: Bag Gems



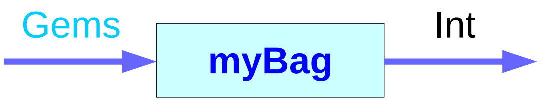
https://stackoverflow.com/questions/14166641/haskell-type-synonyms-for-functions

Type Synonyms for Functions

```
type Bag a = a -> Int                    no data constructor

data Gems = Sapphire | Emerald | Diamond deriving (Show)
```

```
myBag :: Bag Gems
myBag Sapphire = 3
myBag Diamond = 2
myBag Emerald = 0
```



Sapphire	3
Diamond	2
Emerald	0

```
emptyBag :: Bag Gems
emptyBag Sapphire = 0
emptyBag Diamond = 0
emptyBag Emerald = 0
```

Gems
emptyBag
Int

Sapphire	0
Diamond	0
Emerald	0

https://stackoverflow.com/questions/14166641/haskell-type-synonyms-for-functions

Pattern matching function

```
data Person = Person String String Int Float String String deriving (Show)

      Type        Data
      Const       Const
```

```
let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

```
firstName :: Person -> String
firstName  (Person firstname _ _ _ _ _) = firstname
```

```
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

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

Toward the Record Syntax

```
data Person = Person String String Int Float String String deriving (Show)

let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

pattern matching functions	
firstName :: Person -> String firstName (Person firstname _ _ _ _ _) = firstname	firstName guy ► “Buddy”
lastName :: Person -> String lastName (Person _ lastname _ _ _ _) = lastname	lastName guy ► “John”
age :: Person -> Int age (Person _ _ age _ _ _) = age	age guy ► 43
height :: Person -> Float height (Person _ _ _ height _ _) = height	height guy ► 184.2
phoneNumber :: Person -> String phoneNumber (Person _ _ _ _ number _) = number	phoneNumber guy ► “526-2928”
flavor :: Person -> String flavor (Person _ _ _ _ _ flavor) = flavor	flavor guy ► “Chocolate”

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

The Record Syntax

```
data Person = Person {  fName           :: String
                     ,  lName           :: String
                     ,  age             :: Int
                     ,  ht              :: Float
                     ,  ph              :: String
                     ,  flavor          :: String
                     } deriving (Show)
```

```
let guy = Person{  fName="Buddy",
                   lName="John",
                   age=43,
                   ht=184.2,
                   ph=”526-2928”,
                   flavor=“Orange” }
```

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

The Record Syntax Example

```
data Car = Car String String Int deriving (Show)
```

non-record

```
Car "Ford" "Mustang" 1967
```

```
data Car = Car {company :: String, model :: String, year :: Int} deriving (Show)
```

record

```
Car {company = "Ford", model = "Mustang", year = 1967}
```

```
Car "Ford" "Mustang" 1967  ★
```

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

Accessor Functions

```
data Person = Person {  fName            :: String
                     ,  lName            :: String
                     ,  age              :: Int
                     ,  ht               :: Float
                     ,  ph               :: String
                     ,  flavor           :: String
                     } deriving (Show)

let guy = Person { fName="Buddy", lName="John", age=43, ht=184.2, ph=”526-2928”, flavor=“Orange” }
```

accessor functions

```
fName          :: Person -> String
lName          :: Person -> String
age            :: Person -> Int
ht             :: Person -> Float
ph             :: Person -> String
flavor         :: Person -> String
```

```
fName      guy   ►  “Buddy”
lName      guy   ►  “John”
age        guy   ►  43
ht         guy   ►  184.2
ph         guy   ►  “526-2928”
flavor     guy   ►  “Orange”
```

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

Update Functions

```
data Configuration = Configuration
                                        { username          :: String
                                         , localHost         :: String
                                         , currentDir        :: String
                                         , homeDir           :: String
                                         , timeConnected   :: Integer
                                        }
```

```
username :: Configuration -> String                              -- accessor function  (automatic)
localHost :: Configuration -> String
-- etc.

changeDir :: Configuration -> String -> Configuration          -- update function
changeDir cfg newDir =
    if directoryExists newDir                    -- make sure the directory exists
        then cfg { currentDir = newDir }
        else error "Directory does not exist"
```

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

Typeclass and Instance Example

```
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

Instance of a typeclass (1)

```
data State a = State { runState :: Int -> (a, Int) }


instance Show (State a) where                    not working!



instance (Show a) => Show (State a) where
    show (State f) = show [show i ++ " => " ++ show (f i) | i <- [0..3]]


getState = State (\c -> (c, c))


putState count = State (\_ -> ((), count))
```

(State a) is an instance of Show

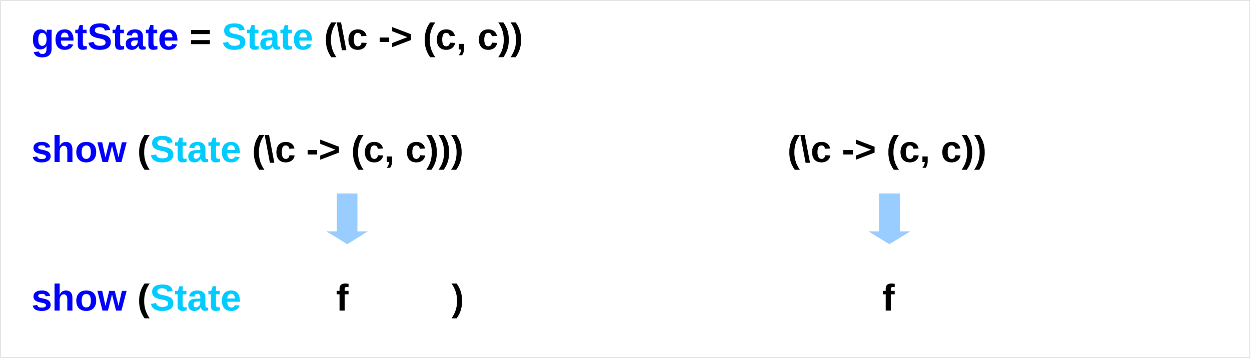
a should be an instance of Show

```
State { runState = (\c -> (c, c)) }

State { runState = (\_ -> ((), c)) }
```

https://stackoverflow.com/questions/7966956/instance-show-state-where-doesnt-compile

Instance of a typeclass (2)



```
getState = State (\c -> (c, c))

show (State (\c -> (c, c)))              (\c -> (c, c))

show (State      f        )                    f
```

```
instance (Show a) => Show (State a) where
    show (State f) = show [show i ++ " => " ++ show (f i) | i <- [0..3]]
```

```
       i=0                        i=1                  i=2                  i=3
show [0 => show (f 0),           1 => show (f, 1),    2 => show (f, 2),    3 => show (f, 3)]

       (\c -> (c, c)) 0          (\c -> (c, c)) 1     (\c -> (c, c)) 2     (\c -> (c, c)) 3

             (0,0)               (1, 1)               (2, 2)               (3, 3)
```

https://stackoverflow.com/questions/7966956/instance-show-state-where-doesnt-compile

Instance of a typeclass (3)

```
data State a = State { runState :: Int -> (a, Int) }

instance (Show a) => Show (State a) where
    show (State f) = show [show i ++ " => " ++ show (f i) | i <- [0..3]]

getState = State (\c -> (c, c))
putState count = State (\_ -> ((), count))
```

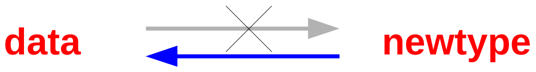
f ➡ (\c -> (c, c))

```
*Main> getState
["0 => (0,0)","1 => (1,1)","2 => (2,2)","3 => (3,3)"]

*Main> putState 1
["0 => ((),1)","1 => ((),1)","2 => ((),1)","3 => ((),1)"]
```

https://stackoverflow.com/questions/7966956/instance-show-state-where-doesnt-compile

newtype and **data**



data can <u>only</u> be replaced with **newtype** →
if the type has exactly *<u>one value constructor</u>*
which can have exactly only *<u>one field</u>*

It ensures that the trivial **wrapping** and **unwrapping**
of the single field is eliminated by the **compiler**.
(using newtype is faster)

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

data, type, and newtype

```
data      State s a = State { runState :: s -> (s, a) }
type      State s a = State { runState :: s -> (s, a) }
newtype   State s a = State { runState :: s -> (s, a) }
```

a new type, data constructor
an alias, no data constructor
a new type, data constructor

instance :	**data**(O),	**type**(X),	**newtype**(O)
overhead :	**data**(O),	**type**(X),	**newtype**(X)

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Single value constructor with a single field

simple wrapper types such as State **Monad**
are usually defined with **newtype**.

type : type synonyms

newtype State s a = **State {** runState :: s -> (s, a) **}**

A single value **constructor** : **State {** runState :: s -> (s, a) **}**
A single **field** : **{** runState :: s -> (s, a) **}**

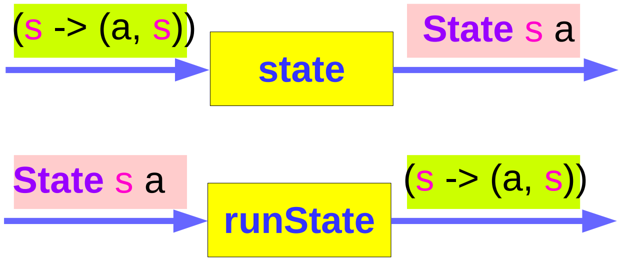
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Single value constructor with a single field

one constructor with one field means that
the new type and **the type of the field**
are in direct correspondence (**isomorphic**)

state :: (s -> (a, s)) -> **State** s a
runState :: **State** s a -> (s -> (a, s))

after the type is checked at compile time,
at run time the two types can be treated identically



(s -> (a, s))	**the type of the field**
State s a	**the new type**

https://stackoverflow.com/questions/2649305/why-is-there-data-and-newtype-in-haskell

Creating a new type class

to declare different new type class instances for a particular type,
or want to make a type abstract,

- wrap it in a **newtype**
- then the type checker treats it as a distinct new type
- but identical at runtime without incurring additional overheads.

Isomorphic relation means
that after the type is checked at compile time,
at run time the two types can be treated essentially the same,
without the overhead or indirection
normally associated with a data constructor.

https://stackoverflow.com/questions/2649305/why-is-there-data-and-newtype-in-haskell

data, newtype, type

	data	newtype	type
value constructors : number	many	only one	none
value constructors : evaluation	lazy	strict	N/A
value constructors : fields	many	only one	none
Compilation Time	affected	affected	affected
Run Time Overhead	runtime overhead	none	none
Created Type	a distinct new type	a distinct new type	a new name
Type Class Instances	type class instances	type class instances	no instance
Pattern Matching Evaluation	at least WHNF	no evaluation	same as the original
Usage	a new data type	higher level concept	higher level concept

https://stackoverflow.com/questions/2649305/why-is-there-data-and-newtype-in-haskell

data

data - creates new algebraic type with value constructors

- can have several value constructors
- value constructors are lazy
- values can have several fields
- affects both compilation and runtime, have runtime overhead
- created type is a distinct new type
- can have its own type class instances
- when pattern matching against value constructors,
  WILL be evaluated at least to weak head normal form (WHNF) *
- used to create new data type
  (example: Address { zip :: String, street :: String })

https://stackoverflow.com/questions/2649305/why-is-there-data-and-newtype-in-haskell

newtype

newtype - creates new “decorating” type with value constructor

- can have only one value constructor
- value constructor is strict
- value can have only one field
- affects only compilation, no runtime overhead
- created type is a distinct new type
- can have its own type class instances
- when pattern matching against value constructor,
  CAN not be evaluated at all *
- used to create higher level concept
  based on existing type with distinct set of
  supported operations or that is not
- interchangeable with original type
  (example: Meter, Cm, Feet is Double)

https://stackoverflow.com/questions/2649305/why-is-there-data-and-newtype-in-haskell

type

type - creates an alternative name (synonym)
for a type (like typedef in C)

- no value constructors
- no fields
- affects only compilation, no runtime overhead
- no new type is created (only a new name for existing type)
- can NOT have its own type class instances
- when pattern matching against data constructor,
  behaves the same as original type
- used to create higher level concept
  - based on existing type with the same set of supported operations (example: String is [Char])

https://stackoverflow.com/questions/2649305/why-is-there-data-and-newtype-in-haskell

newtype examples

```
newtype Fd = Fd CInt
-- data Fd = Fd CInt would also be valid

-- newtypes can have deriving clauses just like normal types
newtype Identity a = Identity a
  deriving (Eq, Ord, Read, Show)

-- record syntax is still allowed, but only for one field
newtype State s a = State { runState :: s -> (s, a) }

-- this is *not* allowed:
-- newtype Pair a b = Pair { pairFst :: a, pairSnd :: b }
-- but this is allowed (no restriction in data):
data Pair a b = Pair { pairFst :: a, pairSnd :: b }          -- two fields
-- and so is this:
newtype NPair a b = NPair (a, b)                 -- one value constructor
```

https://wiki.haskell.org/Newtype

References

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