Defining Type Classes and Type Class Instances

In this note we will see how we can define our own instances of a type class. As a running example, consider the Card class and its friends, that we used in assignment 3:

```
data Suit = Clubs | Diamonds | Hearts | Spades
data Value = Ace | Num Int | Jack | Queen | King

data Card = Cd Suit Value
```

In the assignment, we automatically *derived* definitions for Eq and Show as well as Ord. We also used simply pairs for the cards. But in order to assign our own implementations of comparisions, we have to create a custom data type, so we use the prefix Cd for cards.

We will now implement "manual" definitions of Eq, Show and Ord, with slight variations.

```
instance Eq Suit where
    Clubs == Clubs         = True
    Diamonds == Diamonds = True
    Hearts == Hearts       = True
    Spades == Spades       = True
    _        == _           = False

instance Ord Suit where
    compare Clubs Clubs          = EQ
    compare Clubs _              = LT
    compare Diamonds Clubs       = GT
    compare Diamonds Diamonds = EQ
    compare Diamonds _           = LT
    compare Hearts Spades        = LT
    compare Hearts Hearts        = EQ
    compare Hearts _             = GT
    compare Spades Spades        = EQ
    compare Spades _             = GT
```

After these definitions, we can ask, for example whether Club < Diamonds and then get the answer True: The default implementation for < using our compare function kicks in.

For Show let's do something different! Each suit has a Unicode character corresponding to it. They come in "black" and "white" variants, depending on whether their interior is filled or not.

```
instance Show Suit where
    show Clubs      = "\x2663"
    show Diamonds = "\x2666"
    show Hearts     = "\x2665"
    show Spades     = "\x2660"
```

Now we can do something like this and see a beautiful symbol for a club: putStrLn $ show Clubs.

Let's further define Enum and Bounded instances. Bounded is easy:

```
instance Bounded Suit where
    minBound = Clubs
    maxBound = Spades
```

We can now ask for minBound :: Suit and we will see the Clubs symbol printed out.

```
instance Enum Suit where
    toEnum 0 = Clubs
    toEnum 1 = Diamonds
    toEnum 2 = Hearts
    toEnum 3 = Spades
    fromEnum Clubs    = 0
    fromEnum Diamonds = 1
    fromEnum Hearts   = 2
    fromEnum Spades   = 3
```

After that definition, we can do [minBound .. maxBound] :: [Suit] and see a list of the four suits.

Next, we will create instances of Eq, Ord, Bounded, Enum and Show for the Value type:

```
instance Eq Value where
    Ace   == Ace    = True
    Jack  == Jack   = True
    Queen == Queen  = True
    King  == King   = True
   Num x  == Num y  = x == y
    _     == _      = False

instance Ord Value where
    compare Ace Ace           = EQ
    compare Jack Jack         = EQ
    compare Queen Queen       = EQ
    compare King King         = EQ
    compare Ace _             = LT
    compare _ Ace             = LT
    compare _ King            = LT
    compare King _            = GT
    compare _ Queen           = LT
    compare Queen _           = GT
    compare _ Jack            = LT
    compare Jack _            = GT
    compare (Num n)  (Num m)  = compare n m

instance Bounded Value where
    minBound = Ace
    maxBound = King

instance Enum Value where
    toEnum 1     = Ace
    toEnum 11    = Jack
    toEnum 12    = Queen
    toEnum 13    = King
    toEnum n     = Num n
    fromEnum Ace      = 1
    fromEnum Jack     = 11
    fromEnum Queen    = 12
    fromEnum King     = 13
    fromEnum (Num n)  = n

instance Show Value where
    show Ace      = "A"
    show (Num n)  = show n
    show Jack     = "J"
    show Queen    = "Q"
    show King     = "K"
```

Now we should implement the same functionality for Card, which consists of a suit and a value. The convention we will follow is that “smaller values come first”. So we first compare the values and then compare the suits.

```
instance Eq Card where
    Cd s1 v1 == Cd s2 v2    =     s1 == s2 && v1 == v2

instance Ord Card where
    Cd s1 v1 `compare` Cd s2 v2    = compare v1 v2 `orElse` compare s1 s2
            where EQ `orElse` o    = o
                  o  `orElse` _    = o
```

We “show” a card by showing the value and the suit next to each other:

```
instance Show Card where
    show (Cd s v) = show v ++ show s
```

We can easily make Card and instance of Bounded too:

```
instance Bounded Card where
    minBound = Cd minBound minBound
    maxBound = Cd maxBound maxBound
```

Now minBound :: Card brings up the Ace of Clubs.

Lastly, Enum. We want to make sure we keep the ordering of the cards, starting with the 13 clubs cards at 1-13, then the diamonds cards 14-26, and so on. In order to do that, we can do some “modulo 13” math on the values of suits and cards. We need to do a bit of work for the toEnum function, because the numbers are “1-13” instead of “0-12”.

```
instance Enum Card where
    fromEnum (Cd s v) = fromEnum v + 13 * fromEnum s
    toEnum n = Cd s v where s = toEnum ((n-1) `div` 13)
                            v = toEnum ((n-1) `mod` 13 + 1)
```

Now we can put all the cards in one list easily:

```
[minBound .. maxBound] :: [Card]
```