Lesson 14: Using type classes
The type *SixSidedDice* in need for a type class

We create the type *SixSidedDice* whose values are the six sides of the dice:

```
data SixSidedDice = S1 | S2 | S3 | S4 | S5 | S6
```

As in Lesson 13, we can derive *SixSidedDice* as an instance of the class *Show*:

```
data SixSidedDice = S1 | S2 | S3 | S4 | S5 | S6 deriving (Show)
```

In that way, we can print the values of the different values of the type *SixSidedDice*:

```
ghci> S1
S1
ghci> S2
S2
ghci> S6
S6
```

Now, let us improve that code and print instead the English word for each number!
The type **SixSidedDie** in need for a type class

To that purpose, we create an instance of `Show` for the type `SixSidedDice`:

```haskell
instance Show SixSidedDie where
  show S1 = "one"
  show S2 = "two"
  show S3 = "three"
  show S4 = "four"
  show S5 = "five"
  show S6 = "six"
```

One obtains in this way a more interesting output than previously with `deriving`:

```
ghci> S1
one
ghci> S2
two
ghci> S6
six
```
Type classes and polymorphism

Suppose that one starts by creating the **SixSidedDice** type:

```haskell
data SixSidedDice = S1 | S2 | S3 | S4 | S5 | S6
```

and then defines the function `show` without integrating it in the class `Show`:

```haskell
show :: SixSidedDice -> String
show S1 = "one"
show S2 = "two"
show S3 = "three"
show S4 = "four"
show S5 = "five"
show S6 = "six"
```

This compiles properly in **ghci** but one gets the error

```
Ambiguous occurrence 'show'
```

when one tries to use the `show` function. The reason is that the definition for `show` just given is conflicting with the definition provided by the type class. We will understand better the situation when we know more about the Haskell module system.
Ad’hoc polymorphism in Haskell

Haskell includes **ad’hoc polymorphism** where the behavior of the function depends on the expected type of the result. Consider for instance the `read` function:

You want a function which solves the problem of turning a String into whatever type you expect

```
read "10"
```

Ad’hoc polymorphism means that `read` behaves as you expect, given the type that you tell it you want back.

If you specify that you want an `Int`, then `read` returns an `Int`

```
::Int -> 10
```

If you specify that you want a `Double`, then `read` returns a `Double`

```
::Double -> 10.0
```

The word **ad’hoc** is used here to differentiate from **parametric polymorphism**.
Minimal complete definition

Remember that the type class \textbf{Eq} is defined as follows:

\begin{verbatim}
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
\end{verbatim}

However, it appears sufficient to define the equality predicate \((==)\) as follows:

\begin{verbatim}
instance Eq SixSidedDie where
  (==) S6 S6 = True
  (==) S5 S5 = True
  (==) S4 S4 = True
  (==) S3 S3 = True
  (==) S2 S2 = True
  (==) S1 S1 = True
  (==) _ _ = False
\end{verbatim}
Indeed, we can check in ghci that defining \( == \) is sufficient to define \( /= \):

```haskell
ghci> S6 == S6
True
ghci> S6 == S5
False
ghci> S5 /= S6
True
ghci> S6 /= S6
False
```

Note that, in this case, the predicate \( /= \) is simply defined by negating \( == \).

**Question:** how do we know what minimal information is necessary to turn a type of interest (such as `SixSidedDice`) into an instance of a given type class?
Hackage and Hoogle

You may use Hackage to find out the minimal complete definition of a type class.

This is what the Hackage repository tells you about the type class **Eq**:

**Description of the Eq type class on the Hackage repository**

It may be also a good idea to explore the Hoogle API search engine!

You may use Hoogle to find all the functions (like `show`) of type `a -> String`:

**List of functions of type `a -> String` suggested by Hoogle**

Here is also what Hoogle has to say about the keyword **Eq**:

**Searching for the keyword Eq in Hoogle**
Implementing *SixSidedDice* as an instance of *Ord*

A distinctive property of the sides of a dice is that they can be ordered!

We thus want to implement *SixSidedDice* as an instance of the type class *Ord*.

Let us recall the definition of the type class *Ord*:

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<=) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

How shall we proceed?
Partial definition of compare for *SixSidedDice*

Here follows a partial definition of the `compare` function:

```haskell
instance Ord SixSidedDie where
    compare S6 S6 = EQ
    compare S6 _  = GT
    compare _  S6 = LT
    compare S5 S5 = EQ
    compare S5 _  = GT
    compare _  S5 = LT
```

Even with a clever use of pattern matching, filling out the complete definition would require a lot of work. Imagine how large this definition would be for a 20-sided dice!
We find thus wiser to derive the fact that `SixSidedDice` is an instance of `Ord`, by proceeding in the following way:

```haskell
data Test1 = AA | ZZ deriving (Eq, Ord)
data Test2 = ZZZ | AAA deriving (Eq, Ord)
```

One can check in `ghci` that the types `Test1` and `Test2` are ordered:

```haskell
ghci> AA < ZZ
True
ghci> AA > ZZ
False
ghci> AAA > ZZZ
True
ghci> AAA < ZZZ
False
```

Note that we observed the same phenomenon at the end of Lesson 13.
Deriving **SixSidedDice** as an instance of **Ord**:

```haskell
data SixSidedDice = S1 | S2 | S3 | S4 | S5 | S6 deriving (Eq, Ord)
```

Note that the data constructors are increasingly ordered from left to right.
The type class **Enum** of sequentially ordered types

The definition of the type class **Enum** follows:

```haskell
class Enum a where
    succ :: a -> a
    pred :: a -> a
    toEnum :: Int -> a
    fromEnum :: a -> Int
    enumFrom :: a -> [a]
    enumFromThen :: a -> a -> [a]
    enumFromTo :: a -> a -> [a]
    enumFromThenTo :: a -> a -> a -> [a]
```

However, as in the case of **Eq**, in order to implement an instance of the type class, it is sufficient to implement only the two methods `toEnum` and `fromEnum`. 
Implementing **SixSidedDice** as an instance of **Enum**

This leads to the simple implementation below:

```haskell
instance Enum SixSidedDice where
  toEnum 0    = S1
  toEnum 1    = S2
  toEnum 2    = S3
  toEnum 3    = S4
  toEnum 4    = S5
  toEnum 5    = S6
  toEnum _    = error "No such value"

fromEnum S1  = 0
fromEnum S2  = 1
fromEnum S3  = 2
fromEnum S4  = 3
fromEnum S5  = 4
fromEnum S6  = 5
```
Implementing **SixSidedDice** as an instance of **Enum**

Once **SixSidedDice** recognized as an instance of the type class **Enum**, we can enjoy some of the benefits:

```
ghci> [S1 .. S6]
[one,two,three,four,five,six]
ghci> [S2,S4 .. S6]
[two,four,six]
```

But what happens with an infinite list?

```
ghci> [S1 .. ]
[one,two,three,four,five,six,** Exception: No such value**
```

Here, we get an error because we did not handle the case of having a missing value.
Deriving **SixSidedDice** as an instance of **Enum**

This leads us to derive **SixSidedDice** as an instance of **Enum**, in the usual way:

```hs
data SixSidedDice = S1 | S2 | S3 | S4 | S5 | S6 deriving (Enum)
```

The previous infinite list is treated in a much nicer way:

```
ghci> [S1 .. ]
[one,two,three,four,five,six]
```

As a matter of fact, Haskell is pretty magical when it comes to deriving type classes. Hence, if you do not have a good reason to implement your own, deriving is not only easier, but also often better!
Type classes for more complex types

Suppose that we define the type **Name** as synonym:

```
    type Name = (String,String)
```

and that you then define a list names of values of type **Name**:

```
    names :: [Name]
    names = ["Emil","Cioran"]
          , ("Eugene","Thacker")
          , ("Friedrich","Nietzsche")]
```

As you may remember from Lesson 4, there is a problem in sorting:

```
ghci> import Data.List
ghci> sort names
["Emil","Cioran"", ("Eugene","Thacker") , ("Friedrich","Nietzsche") ]
```
Type class for more complex types

In order to resolve that problem, it is tempting to write:

```haskell
instance Ord Name where
    compare (f1,l1) (f2,l2) = compare (l1,f1) (l2,f2)
```

Unfortunately, when one tries to load that code, one gets an error!

The reason is that `Name` is the type `identical` to the type `String` which is already an instance of the type class `Ord` of ordered types.

So, one needs to create a new data type `Name` instead of defining it as a synonym.
Type classes for more complex types

Creating the new data type \texttt{Name} is done as expected:

\begin{verbatim}
data Name = Name (String, String) deriving (Show, Eq)
\end{verbatim}

Now, it becomes possible to define \texttt{Name} as an instance of \texttt{Ord} in the desired way:

\begin{verbatim}
instance Ord Name where
  compare (Name (f1,l1)) (Name (f2,l2)) = compare (l1,f1) (l2,f2)
\end{verbatim}

Another solution (not developed here) would be to use \texttt{newtype}, as follows:

\begin{verbatim}
newtype Name = Name (String,String) deriving (Show,Eq)
\end{verbatim}

in order to define a synonym not necessarily instance of the original type classes.
Type classes for more complex types

We are now ready to sort our original list `names` of philosophers and poets:

```haskell
names :: [Name]
names = ["Emil","Cioran"]
      , ["Eugene","Thacker"]
      , ["Friedrich","Nietzsche"]
```

and check that the sorting works now exactly as it should:

```haskell
ghci> import Data.List
ghci> sort names
["Emil","Cioran"], ["Friedrich","Nietzsche"], ["Eugene","Thacker"]
```

Nicely done, do you agree?
Conclusion: the type class road map in Haskell

In this map, arrows from one class to another indicate a **superclass relationship**:

In Unit 3, we will explore the more abstract type classes **Semigroup** and **Monoid**.

In Unit 5, we will look at a family of type classes **Functor**, **Applicative** and **Monad** which will provide us with a way to model the **context of a computation**.
Thank you for your attention!