Lesson 16: Creating types with "and" and "or"
Sum types – combining types with "or"

The simplest sum type is probably the type `Bool` of booleans, defined as:

```haskell
data Bool = False | True
```

We have seen in Lesson 12 how to use sum types to define `Name` as the type of names, with two different formats for names:

```haskell

-- Types for names

type FirstName = String
type LastName = String
type MiddleName = String

data Name = Name FirstName LastName
          | NameWithMiddle FirstName MiddleName LastName

```

Here, a name consists of the data constructor `Name` followed by two strings or the data constructor `NameWithMiddle` followed by three strings.
The *Creator* type

Imagine that you start a bookshop, and want to build a database.

An interesting place to start is the difference between author and artist.

```haskell
data Creator = AuthorCreator Author | ArtistCreator Artist
```

where we may use the type *Name* just defined for authors:

```haskell
data Author = Author Name
```

and the sum type for artists, depending on whether they are a person or a band:

```haskell
data Artist = Person Name | Band String
```
**The Creator type – extended once**

Now, suppose that you want to add an entry for the famous American writer

H. P. Lovecraft

Of course, we could force ourselves to use his original name

Howard Phillips Lovecraft

However, this can be fixed by adding a new data constructor to the **Name** type:

```
data Name = Name FirstName LastName
            | NameWithMiddle FirstName MiddleName LastName
            | TwoInitialsWithLast Char Char Char LastName
```

Thanks to this new data construction, we can then define:

```
hpLovecraft :: Creator
hpLovecraft = AuthorCreator
              (Author
               (TwoInitialsWithLast 'H' 'P' "Lovecraft")
```
The **Creator** type – extended twice

Similarly, you may wish to add an entry for the American singer

Andrew W. K.

with two initials as first name. Instead of writing his real name

Andrew Fetterly Wilkes-Krier

you may prefer to add a new data constructor to the **Name** type:

```haskell
data Name = Name FirstName LastName
           | NameWithMiddle FirstName MiddleName LastName
           | TwoInitialsWithLast Char Char LastName
           | FirstNameWithTwoInits FirstName Char Char
```


Putting together your bookstore

Now that we have a powerful Creator type, we can define the type Book in the following way, using the record syntax:

```haskell
data Book = Book {
  author :: Creator,
  isbn :: String,
  bookTitle :: String,
  bookYear :: String,
  bookPrice :: String
}
```
Putting together your bookstore

We can also define the *VinylRecord* type in the following way:

```haskell
data VinylRecord = VinylRecord { artist :: Creator
 , recordTitle :: String
 , recordYear :: Int
 , recordPrice :: Double
}
```
Putting together your bookstore

Note that we need to use the different field names `bookPrice` and `recordPrice` instead of just using the same field `price` for both record types.

The reason is that each field creates a function

```
bookPrice :: Book -> Double
bookPrice (Book _ _ _ _ val) = val

recordPrice :: Record -> Double
recordPrice (Record _ _ _ _ val) = val
```

which would be conflicting if the two fields had the same name `price`. 
Putting together your bookstore

This leads one to define the `StoreItem` type for books and records:

```haskell
data StoreItem = BookItem Book | RecordItem VinylRecord
```
Refactoring the **StoreItem** type

Imagine that we forgot to include collectible toys!

We then fix the situation with the type:

```haskell
data CollectibleToy = CollectibleToy {
  name        :: String
, description :: String
, toyPrice    :: Double
}
```

The original **StoreItem** type is extended and refactored to include collectible toys:

```haskell
data StoreItem = BookItem Book
               | RecordItem VinylRecord
               | ToyItem CollectibleToy
```
Using the StoreItem type with a price function

We can then define the price function as follows:

```
price :: StoreItem -> Double
price (BookItem book) = bookPrice book
price (RecordItem record) = recordPrice record
price (ToyItem toy) = toyPrice toy
```

Exercise: turn Creator into an instance of the Show type class and then implement a madeBy function of type signature

```
madeBy :: StoreItem -> String
```

which does its best to determine who made a given item sold in the store.
Thank you for your attention!