



Practice exercises on Building Modular Projects

Project 1: Common Shared Types Library

1. Using the stack build tool, create a project called **lost-and-found** in your projects folder.
 - a. In one to three sentences, describe the role of each file / folder present in the root folder of your project.
 - b. Provide a definition to both the Stackage and Hackage platform and elaborate on the differences between the two and the roles they play
 - c. Which file and keywords are modified when there is a need to select a specific snapshot or a specific version of the ghci compiler
2. Under the src/Common folder within the root of your project, create the following folders and files with the illustrated structure:

```
src
  Common
    Compound
      CompoundTypes.hs
    Constructors
      Constructors.hs
      Constructors
        Email
          Uutils
            Emails.hs
        Constructors
          Numeric
            Uutils
              Numerics.hs
        Constructors
          String
            Uutils
              Strings.hs
    Simple
      SimpleTypes.hs
```



3. What are the right names for the modules unfolding from the following file? Hint: The folder structure is key here

```
CompoudTypes.hs  
Emails.hs  
Numerics.hs  
Strings.hs  
SimpleTypes.hs
```

4. In the file **SimpleTypes.hs** create the following newtypes, and type synonyms

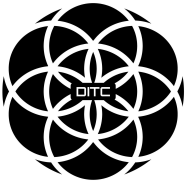
a. Newtypes:

```
LostItemId  
UserId  
ShortDescription  
LongDescription  
EmailAddress  
Telephone  
CategoryId  
CategoryCode  
ParentCategoryId  
  
...  
  
ValidationError  
DomainError
```

For example:

```
newtype LostItemId = LostItemId String deriving (Eq, Ord, Show)
```

b. Type synonyms:



```
ErrorMessage  
Country  
Reason
```

For example:

```
type ErrorMessage = String
```

- c. Using the ***mapLeft :: (a -> c) -> Either a b -> Either c b*** function from the ***either package*** found on the ***hackage platform***, define the following ***mapValidationError*** and ***mapDomainError*** that transform simple error messages types (***Either ErrorMessage b***) respectively into ***Either ValidationError b*** and ***Either DomainError b*** types

```
mapValidationError :: Either ErrorMessage b -> Either ValidationError b  
mapValidationError = undefined  
  
mapDomainError :: Either ErrorMessage b -> Either DomainError b  
mapDomainError = undefined
```

5. Within the ***src/Common/Constructors/String/Utils/Strings.hs*** define the following helper functions to construct strings

```
createString :: String -> (String -> a) -> Int -> String -> Either  
ErrorMessage a  
createString fieldName ctor maxLen str = undefined  
  
createBoundedString :: String -> (String -> a) -> Int -> Int -> String  
-> Either ErrorMessage a  
createBoundedString fieldName ctor minLen maxLen str = undefined
```



```
createStringOpt :: String -> (String -> a) -> Int -> String -> Either  
ErrorMessage (Maybe a)  
createStringOpt fieldName ctor maxLen str = undefined  
  
createEmail :: String -> (String -> EmailAddress) -> String -> Either  
ErrorMessage EmailAddress  
createEmail fieldName ctor str = undefined
```

6. Within the ***src/Common/Constructors/Email/Utils/Emails.hs*** define the following helper functions to construct emails

```
createEmail :: String -> (String -> EmailAddress) -> String -> Either  
ErrorMessage EmailAddress  
createEmail fieldName ctor str = undefined
```

7. Within the ***src/Common/Constructors/Numeric/Utils/Numeric.hs*** define the following helper functions to construct numeric values

```
createNum :: (Num a, Eq a, Ord a, Show a) => String -> (a -> a) -> a ->  
a -> a -> Either ErrorMessage a  
createNum fieldName ctor minVal maxVal i = undefined
```

8. Within the ***src/Common/Constructors/Constructors.hs*** file, define the following ***private constructors*** for your wrapped types defined in ***SimpleTypes.hs***

```
createLostItemId :: String -> Either ErrorMessage LostItemId  
-- LostItemId is constrained to 9 characters exactly.
```



```
createUserId :: String -> Either ErrorMessage UserId
-- UserId is constrained to 12 characters exactly.

createShortDescpt :: String -> Either ErrorMessage ShortDescription
-- ShortDescription is constrained to 250 characters maximum

createLongDescpt :: String -> Either ErrorMessage LongDescription
-- LongDescription is constrained to 5000 characters maximum

crtEmailAddress :: String -> Either ErrorMessage EmailAddress
-- Email addresses have 2 not null parts separated by the @
```

9. Create a module `CompoundTypes` under the `Common` folder that uses types from `SimpleTypes.hs` to define its own types

- a. Create a sum data type ***EnablementStatus*** that specifies whether a `Category` is enabled or not. It uses the `Reason` type synonym to capture the motivation for changing the status.

```
...

data ... = Enabled ... | Disabled ...

...
```

- b. Create a data type named ***CategoryInfo*** with the following fields:

```
...

categoryId :: CategoryId
categoryCode :: CategoryCode
categoryEnablementStatus :: EnablementStatus
categoryDescription :: LongDescription
categoryRelatedSubCategories :: Set CategoryId

...
```



- c. Create a data type named **ParentInfo** with the following fields:

```
...  
  
parentInfoId :: ParentCategoryId  
parentInfoCode :: CategoryCode  
  
...
```

- d. Create a sum data type **Category** that has the following characteristics:
- A category can be Root or Sub with category information (CategoryInfo)
 - Root categories carry the information about the category (CategoryInfo)
 - Sub categories carry the info not only about the category info (CategoryInfo) but also optional information about their Parent information (ParentInfo).

```
...  
  
data ... = RootCategory ... | Sub ... ...  
  
...
```

10. Within the CompoundTypes.hs file under the Common folder create the following functions that uses types from SimpleTypes to define its own types

```
toEnablementStatus :: (String, String) -> Either ErrorMessage  
EnablementStatus  
toEnablementStatus (anEnablementType, anEnablementReason) = undefined  
  
fromEnablement :: EnablementStatus -> (String, String)  
fromEnablement = undefined
```

11. Within the CompoundTypes.hs file under the Common folder create the following functions that uses types and the mapValidator function from SimpleTypes.hs

```
toLostItemId :: String -> Either ValidationError  
toLostItemId = undefined
```



```
toUserId :: String -> Either ValiationError
toUserId = undefined

checkIsSubCategoryAndEnabled :: Category -> Either DomainError Bool
checkIsSubCategoryAndEnabled = undefined
```

12. What are cyclic dependencies? Have you run into some during this project? If yes, explain how to solve the issue.

Project 2: A Geometry Library

1. Create a new project “geometry” with the **stack command** and list all the default dependencies that are present in it.
2. Create a file called Geometry.hs in the src folder that defines the **Geometry** module including the following functions:

```
...

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c

cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectangleArea a b * 2 + rectangleArea a c * 2 +
    rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
```



```
rectangleArea a b = a * b
```

3. Create two folders: **Geometry** and **Algebra** under the src folder, then create a file **Monoid.hs** in the Algebra folder. Try to import the Geometry module within the Monoid.hs file and make sense of the error that arises.

4. Create **three sub-modules** to the Geometry module and determine where they should be placed. (It can be inferred from the hierarchical structures of the folders within the project)

Geometry.Sphere exporting the following functions

```
...  
  
volume :: Float -> Float  
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)  
  
area :: Float -> Float  
area radius = 4 * pi * (radius ^ 2)
```

Geometry.Cuboid exporting the following functions

```
...  
  
volume :: Float -> Float -> Float -> Float  
volume a b c = rectangleArea a b * c  
  
area :: Float -> Float -> Float -> Float  
area a b c = rectangleArea a b * 2 + rectangleArea a c * 2 +  
             rectangleArea c b * 2  
  
rectangleArea :: Float -> Float -> Float  
rectangleArea a b = a * b
```

Geometry.Cube exporting the following functions



```
...  
  
import qualified Geometry.Cuboid as Cuboid  
  
volume :: Float -> Float  
volume side = Cuboid.volume side side side  
  
area :: Float -> Float  
area side = Cuboid.area side side side
```

5. Explained in great detail what ***import qualified Geometry.Cuboid as Cuboid*** does and why it is needed in this specific context

6. Provide a definition for the following concepts

- a. Module
- b. Library
- c. package