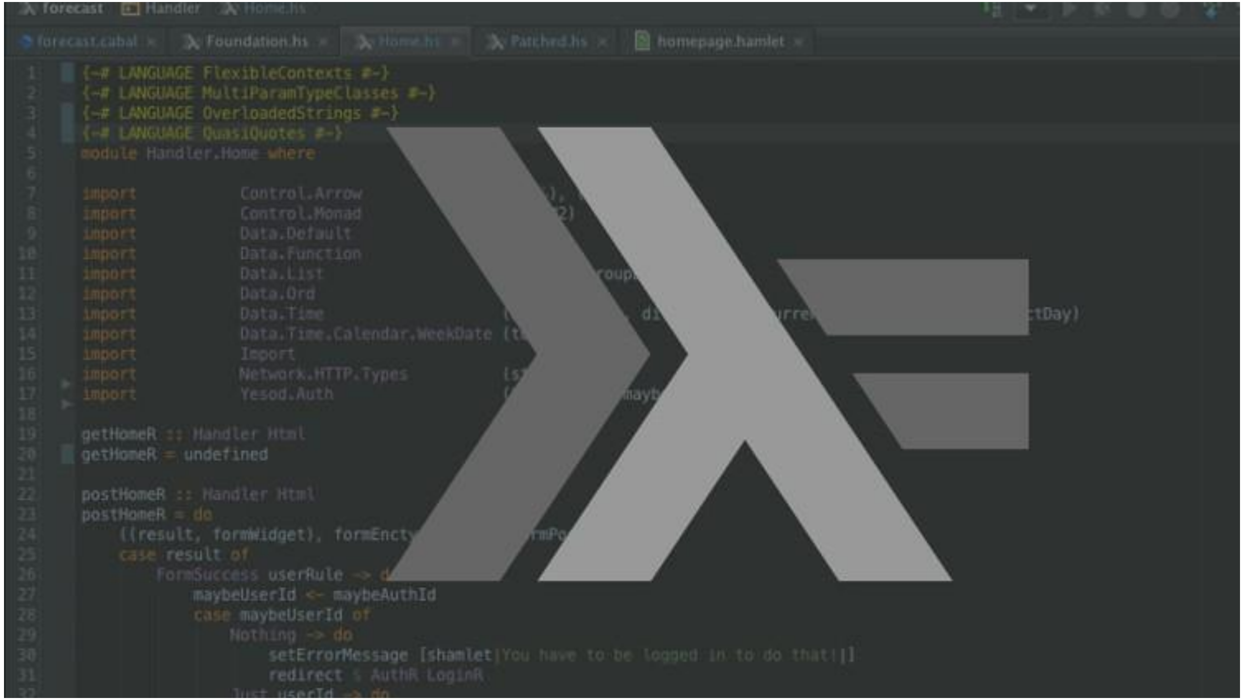
Custom Data Types



```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes #-}
module Handler.Home where

import            Control.Arrow                          ), 
import            Control.Monad                          2)
import            Data.Default
import            Data.Function
import            Data.List                              roup
import            Data.Time                   (        , di        urre                  ctDay)
import            Data.Time.Calendar.WeekDate (t
import            Import
import            Network.HTTP.Types          (s
import            Yesod.Auth                  (          mayb

getHomeR :: Handler Html
getHomeR = undefined

postHomeR :: Handler Html
postHomeR = do
    ((result, formWidget), formEncty          rmPo
    case result of
        FormSuccess userRule -> d
            maybeUserId <- maybeAuthId
            case maybeUserId of
                Nothing -> do
                    setErrorMessage [shamlet|You have to be logged in to do that!|]
                    redirect $ AuthR LoginR
                Just userId -> do
```

Custom Data Types

- In order to create a custom data type, we use the `data` keyword.

Custom Data Types

- In order to create a custom data type, we use the `data` keyword.

- For instance, the Bool type is defined in the standard library in this way:

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

`data` means we are defining a new data type.

Custom Data Types

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

- The parts after the = are *value constructors*. They specify the different values that this type can have.

Custom Data Types

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

- The parts after the = are *value constructors*. They specify the different values that this type can have.

- The | symbol is interpreted as "or". Therefore, we can say that the Bool type can have a value of either True or False. Both the type name and its value constructors must be capitalized.

Custom Data Types

- Now, let us consider how we could represent a shape in Haskell.
- One approach is to use tuples. For example, a circle could be represented as (53.1, 30.0, 12.8), where the first two values are the coordinates of the circle's center, and the third is the radius.
- While this works, those values could just as easily represent a 3D vector or something else entirely.

Custom Data Types

- A more effective solution would be to define **our own** type to represent a shape. Let us say a shape can be either a circle or a square. Here's how:

```
data Shape = Circle Float Float Float | Square Float
```

Custom Data Types

```
data Shape = Circle Float Float Float | Square Float
```

- The Circle value constructor has three fields, all of which are floats. Here, the first two fields represent the coordinates its center, the third one its radius.
- The Square value constructor has only one field (which accepts a float) that represents the side of the square.

Custom Data Types

- Internally, value constructors are actually functions that ultimately return a value of a data type.

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Square
Square :: Float -> Shape
```

Custom Data Types

- Let us now make a function that, given a shape, returns its area.

```
area :: Shape -> Float
area (Circle _ _ r) = pi * r ^ 2
area (Square s) = s * s
```

Custom Data Types

- Now, if we try to just print out `Circle 15 10 5` in the prompt, we'll get an error.

- This is because Haskell does not know yet how to display our data type as a string.

Custom Data Types

- When we try to print a value out in the prompt, Haskell first runs the `show` function to get the string representation of our value and then it prints that out to the terminal.

- Thus, we make our data type an instance of the `Show` class using `deriving (Show)`.