```haskell
-- MAKING OUR OWN TYPES AND TYPECLASSES --
module Shapes (
        Point(..),
        Shape(..),
        surface,
        nudge,
        baseCircle,
        baseRectangle
) where -- Export types like [TYPE]([Constructors]|..)
import qualified Data.Map as Map

-- ALGEBRAIC DATA TYPES INTRO --

-- data Bool = False | True
-- We use 'data' to say that we're defining a new data type. The part before the = is the name of the type, and the parts after are the value
-- constructors. They're the different values the type can have.
-- We define data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1 | 2 | ... | 2147483647 (it doesn't actually have the ... ).

-- How can we represent a shape in Haskell? We can use a tuple (x-cord, y-cord, radius), but that only works for circles, and it's not exactly
-- very specific about what value is what. This could also represent a 3-D vector or a bunch of other things. It's better to use a bunch of
-- different things to represent a shape.

-- data Shape = Rectangle Float Float Float Float | Circle Float Float Float deriving (Show) -- OLD --
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)

-- The Shape type is made up of two different constructors (Rectangle and Circle). Rectangle takes four Floats, the first two are coordinates
-- of the lower left corner and the last two are coordinates of the upper right corner. With the Cirle, the first two are the coordinates of
-- the center, and the last is the radius.

-- Constructors are functions like everything else.
-- An important thing to see is the type declaration here. We take a SHAPE. Not a Circle or a Rectangle. The only thing that's a type is Shape.
-- Just like we can't write a function with True -> Int
surface :: Shape -> Float
-- surface (Circle _ _ r) = pi * r ^ 2 -- OLD --
-- surface (Rectangle x1 y1 x2 y2) = (abs $ x1 - x2) * (abs $ y1 - y2)
surface (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x1 - x2) * (abs $ y1 - y2)
surface (Circle _ r) = pi * r ^ 2


-- We can also pattern match against constructors. Because we aren't worried about the positioning of the Circle, we only need the radius.
-- If we just try to give Haskell "Circle 5 5 10", we get an error because it doesn't know how to display that on the screen.
-- We can add 'deriving ([TYPE])' to the data declaration in order to tell Haskell that it can be shown.

-- Since value constructors are functions, we can map them and partially apply them.
mapFunction :: Float -> Float -> [Shape]
mapFunction a b = map (Circle (Point 5.0 4.0)) [a,b,5,6,7]

-- We have a good Shape data type, but we can make it better by implimenting a Point data type (line 14)
-- How about a function that nudgs a shape?
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b = Rectangle (Point (x1+a) (y1+b)) (Point (x2+a) (y2+b))

-- If we don't want to have to deal directly with points, we can define a function that wil create a base circle or a base rectangle and then
-- just nudge that.
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r
baseRectangle :: Float -> Float -> Shape
baseRectangle width height = Rectangle (Point 0 0) (Point width height)
-- Now we can just nudge in order to declare the new position without having to type out Point all the time.


-- RECORD SYNTAX --

-- If we want to make a person, we could define it as
-- data Person = Person String String Int Float String String deriving (Show)
-- But then we would have to define a lot of different functions to get a specific value, and the placement of the differnet types doesn't make
-- it apparent what value goes where.
-- Instead, we can define it like this
--data Person = Person {
--      firstName :: String,
--      lastName :: String,
--      age :: Int,
--      height :: Float,
--      phoneNumber :: String,
--      flavor :: String
--} deriving (Show)
-- Now, Haskell already made functions for firstName, lastName, etc. to return the values.

-- There's also another benefit to using record syntax. If we just define a Car as
-- data Car = Car String String Int deriving (Show)
-- Then when we try to show a car, it only prints out "Car "Ford" "Mustang" 1967"
-- If we define a car as
data Car = Car {
        company :: String,
        model :: String,
        year :: Int
} deriving (Show)
-- Then we show a car
doCar :: Car
doCar = Car {company = "Ford", model = "Mustang", year = 1967}

-- It will display like this:
-- Car {company = "Ford", model = "Mustang", year = 1967}
-- We also don't have to put the specific names in a certain order because we are assigning using a key-value-ish system.


-- TYPE PARAMETERS --

-- A value constuctor can take a few parameters and return a result. As an example, Car takes three values and produces a car value.
```

```haskell
-- Type constuctors can also take values just like value constuctors to create new types.
-- data Maybe a = Nothing | Just a
-- Here, the a is the type parameter. Because there is a type parameter involved, we call this a type constuctor. Depending on what
-- we want this type constuctor to hold, it can produce Maybe Int, Maybe Car, etc. No value will have a type of just Maybe because it's not
-- exactly a type. It's a type constuctor. In order for it to be a type, it needs to have all of its parameters filled up.
-- So if we pass in a Char to Maybe, we get Maybe Char. The type of "Just 'a'" is Maybe Char.
-- We used a type that has a type parameter before Maybe. Lists have a type paramater. [Int], [Char], [[String]]. However you can't just have []

-- Type parameters are useful because ti allows us to make different types based on what you want contained. If the a in Just a is a String,
-- then the a in Maybe a is also a String.

-- The type of Nothing is Maybe a because its type is polymorphic. If some funciton requires Maybe Int, we can give it Nothing because Nothing
-- doesn't contain any values anyway, so it doesn't matter. The type Maybe a can act as Maybe Int or Maybe Double just like 5 can act as
-- an Int or Double. An empty list can act as a list of anything. That's why we can do [1,2,3] ++ [].

-- Using type parameters is very beneficial, but only when using them makes sense. For example, we wouldn't define Car as
-- {company :: a, model :: b, year :: c} because company and model will always be a String and the year will always be an Int. If our type
-- acts as some kind of a box, it's good to use them.
-- Here's a funciton to show the properties of a car.
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " " ++ m ++ " was made in " ++ show y

-- This function would have to be made if the car was declared with types a, b, and c.
-- tellCar' :: (Show a) => Car String String a -> String
-- tellCar' (Car {company = c, model = m, year = y}) = "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
-- As you can see the type declaration is very gross looking and only benefits us in that we can use any data type that is of the Show typeclass

-- So we would rather just use Car String String Int to make it simple. We usually use type parameters when the type inside of the value doesn't
-- really need to be a certian one for the type to work. A list of stuff is just a list of stuff. If we want it to be a list of integers,
-- we can specify that in the function's type declaration. The same goes for Maybe values. Maybe represents having either nothing or have one
-- of something. It doesn't matter what the type of that thing is.

-- Another example of a parameterized type that we've met is Map k v from Data.Map. The k is the key in the map, and the v is the value. This
-- is an example of where type parameters are useful. This allows us to have two types, but it doesn't matter what types they actually are.
-- So you can have a key that's an Int and a value that's a String. Or the other way around. Having maps parameterized allows us to have
-- mappings from any type to any other type as long as they key is within the Ord typeclass.

-- data (Ord k) => Map k v = ...
-- It's a very strong convention to never add typeclass constraints in data declarations because we don't benefit a lot. We just end up writing
-- more class constraints. If we do have the Ord k in the data delcaration, then we won't have to write Ord k when writing a function that uses
-- Map, but if we don't put it in, we don't need to have Ord k in our function type delcaration.
-- An example of this is toList. It's defined like toList :: Map k v -> [(k,v)]
-- If we had Ord k in our data declaration, we would need to write toList :: (Ord k) => Map k v -> [(k,v)] even though the function does nothing
-- with comparing by order.
-- So don't put typeclass constraints into data declarations even if it makes sense at the time because you'll have to put them into the
-- function declarations either way.

-- Here'd a 3D vector type
data Vector a = Vector a a a deriving (Show)
-- Function to add two vectors.
vplus :: (Num t) => Vector t -> Vector t -> Vector t
(Vector a b c) `vplus` (Vector d e f) = Vector (a+d) (b+e) (c+f)

-- Function to multiply two vectors together
vmult :: (Num a) => Vector a -> Vector a -> Vector a
(Vector a b c) `vmult` (Vector d e f) = Vector (a*d) (b*e) (c*f)

-- Function to add together the products of the scalar values of two vectors
scalarMult :: (Num a) => Vector a -> Vector a -> a
(Vector a b c) `scalarMult` (Vector d e f) = (a*d) + (b*e) + (c*f)
-- Notice how we didn't put the Num a constraint in the data declaration because we would need to do the same thing in the funciton declaration.


-- DERIVED INSTANCES --

-- Typeclassesa are somewhat like Java interfaces, but not exactly. A type can be made an instance of a typeclass if it supports that behavior.
-- Example: The Int type is an instance of the Eq typeclass because the Eq typeclass defines behavior for stuff that can be equated. And
-- because Integers can be equated, Int is a part of the Eq typeclass. The real usefulness comes with the functions that act as the interface
-- for Eq, namely == and /=. If a type is a part of the Eq typeclass, we can use the == function with values of that type. That's why
-- 4 == 4 or "foo" /= "bar" typecheck.

-- Let's look at how Haskell can automatically make our type an instance of Eq, Ord, Enum, Bounded, Show, and Read. Haskell can derive
-- the behavior of our types if we use teh deriving keyword in our data declaration.
-- Consider the Person data type
--data Person = Person {
--      firstName :: String,
--      lastName :: String,
--      age :: Int
--}
-- Say we have a list of persons, and none of them are the same. Would it make sense to have to compare two persons to see if two references
-- are the same person? Yes. So we would add on a "deriving (Eq)" to the data declaration so that we can compare two Persons with ==
-- When we try to an an ==, Haskell will look to see if the value constructors match.
-- ALL VALUES INSIDE OF THE CONSTRUCTOR MUST ALSO BE OF THE EQ TYPECLASS
-- Since String and Int are both of the Eq typeclass, we're good to go.
data Person = Person {
        firstName :: String,
        lastName :: String,
        age :: Int
} deriving (Eq, Show, Read)

-- Let's test out how People are compared.
comparePeople :: Bool
comparePeople = mike == joe
                        where
                                mike = Person {firstName = "Mike", lastName = "Pike", age = 24}
                                joe = Person {firstName = "Joe", lastName = "Bob", age = 36}

-- Of course, since Person is of type Eq, we can use it as the a in functions that have Eq a in their type signature. Such as elem.
-- If we had a list of people and wanted to see if a certain one was a part of that list, then we could simply do
-- [person] `elem` [listofpeople]

-- If we add a derivation of Show to the Person type, we can print it out to the terminal. If we tried to Show before we added it, then
```

```haskell
-- Haskell would have complained about us trying to show something that it claims it doesn't know how to show.

-- When we add Read to the Person type, we can read in a value like "let mike = read "Person {firstName=\"Mike\", lastName=\"Diamond\",
-- age = 43}" :: Person" and it would read in Mike as a Person.

-- We can also read in parameterized types like Maybe values.
-- read "Just 't'" :: Maybe Char

-- The way we can derive instances of the Ord typeclass are by order of definition. Say we define the data type of Bool.
-- data Bool = False | True deriving (Ord)
-- What's cool about this is that the value that's defined first is considered the least, and the one that's defined last is considered
-- to be the greatest. So True is considered to be greater than False.
-- In the Maybe a datatype, the Nothing value is defined before the Just a value, so the value of Nothing is less than Just a. However,
-- if we compare two Just a values, it just compares the 'a' value. We can't do something like Just (*3) > Just (*2) because those are
-- functions.

-- We can use algebraic data types to make enumerations and the Enum and Bounded typeclasses can help us with that.
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday deriving (Enum, Bounded, Eq, Ord, Show, Read)
-- Because all of the value constructors are nullary (take no parameters), we can make these a derivation of the Enum typeclass.
-- the Enum typeclass is for things that have predicessors and successors. We also can give it Bounded type because Bounded is for things that
-- have a definite upper and lower bound. While we're at it, let's make it an instance of a bunch of different typeclasses.
-- Because it's a part of the Show and Read typeclass, we can easily change things to and from Strings.
doWed :: Day
doWed = Wednesday
readDay :: String -> Day
readDay a = read a :: Day

-- Because Day is a part of the Eq and Ord typeclass, we can equate and compare two days.
eqDay :: Day -> Day -> Bool
eqDay a b = a == b
compareDay :: Day -> Day -> Ordering
compareDay a b = a `compare` b

-- We can also get maxBound :: Day and minBound :: Day. Also we can see what the successors and predicessors are.


-- TYPE SYNONYMS --

-- Because String and [Char] are the same thing, that's done with type synonyms.
-- type String = [Char]
-- We aren't defining a new type here. We're simply making what is essentialy an alaias.
-- In Data.Map, we first represented a phonebook with an association list before converting it to a map. Let's look at the phonebook we made.
-- phoneBook :: [(String, String)]
-- We see that the type is [(String, String)], but tells us that it maps from Strings to Strings, but not much else.
-- type PhoneBook = [(String, String)]
-- Now the type of our phoneBook can be
-- phoneBook :: PhoneBook
-- Let's make a type synonym for String as well.
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name, PhoneNumber)]
-- Using type synonyms in code is good because it lets people know more about what the code should do and how certain things should be made.
-- Here's a function to check if a phone number and name is in a phone book.
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pn pb = (name,pn) `elem` pb
-- If we decided not to use type synonyms, our type would be String -> String -> [(String, String)] -> Bool, which is more confusing.
-- It's important to remember not ot go overboard with type synonyms though. They're good when you're working with stuff that is long and
-- can be confusing, but they shouldn't be used to just replace a simple Int or Char every here and there.

-- Type synonyms can also be parameterized!
type AssocList k v = [(k,v)]
-- Now a function that has the type signature of (Eq k) => k -> AssocList k v -> Maybe v because AssocList is a type constructor that
-- can take stuff like Int String.

-- Just like we can partially apply functions, we can also partially apply type parameters to get what we want. Like this:
type IntMap v = Map.Map Int v
-- or we could do it like this
-- data IntMap = Map Int
-- Either way, the IntMap constructor takes one parameter and that is the type that the Int will point to in the map.

-- The difference between value constructors and type constructors is that we can use a value constructor to make a value, but we can't use
-- a type construstor to make a new value. We just are changing the way we can interpret a type. We can't do AssocList [(1,2), (3,4)] but
-- we can do [(1,2), (3,4)] :: AssocList Int Int, which will make the numbers inside assume the type of Int.

-- Another cool data type is the Either data type.
-- data Either a b = Left a | Right b deriving (Eq, Ord, Show, Read)
-- It has two value constructors. If Left it used, then the contents are of type a, and if Right is used, its contents are of type b. So we
-- can use this type to encapsulate a value of one type or another and then we get a value that's Either a or b. We usually will pattern match
-- on both Left and Right and we get different stuff based on which one of them it was.

-- We've used Maybe a to represent the results of computations that could have either failed or not. Sometimes Maybe a isn't good enough because
-- Nothing doesn't really tell us much other than something has failed. It's good for functions that can only fail one way or if we don't care
-- if they fail or not. We can use Either a b to tell us about errors. Left values are used for errors, while Right values are used for results.

-- Here's a little example with lockers.
data LockerState = Taken | Free deriving (Eq, Show)
type Code = String
type LockerMap = Map.Map Int (LockerState, Code)

lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNum mapp =
        case Map.lookup lockerNum mapp of
                Nothing -> Left $ "Locker " ++ show lockerNum ++ " doesn't exist."
                Just (state, code) -> if state == Free
                                                        then Right code
                                                        else Left $ "Locker " ++ show lockerNum ++ " is already taken."
-- So calling lockerLookup 5 $ Map.fromList [(5,(Free,"382938"))] will return Right "382938".
-- We could have used Maybe a to represent out results, but then we wouldn't know why they failed.


-- RECURSIVE DATA STRUCTURES --
```

```haskell
-- Because we create data types with fields that are of some concrete type, we can also define types that take the same type as a parameter.
-- Let's use algebraic data types to impliment our own list then
-- data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
-- The Cons is basically using :, so we're adding to the list. We take a head value and the rest of the list, or we take an empty list.
-- Typing "Cons 5 (Cons 4 (Cons 3 Empty))" will give us something like a list where Cons is :. Just like 5:4:3:[]

-- We can define functions to be automatically infix by making them only comprised of special characters, so we can also do the same to
-- constructors.
infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)
-- The infix_ makes something right associative or left associative. infixr and infixl. The number defines the fixity. The higher the number,
-- the more presidence is gets over the others. So * has 7, and + has 6. If you write 4 * 5 + 7, it's the same as (4 * 5) + 7.
-- Otherwise we just wrote a :-: instead of Cons a.
-- Writing 5 :-: 5 :-: 4 :-: Empty give us ("5 :-: (5 :-: (4 :-: Empty))"
-- When we derive the Show typeclass, Haskell is treating this as a prefix function, so it puts parenthesis around the operator.
-- Remember 4 + 4 and (+) 4 4 are the same.

-- Let's make a function to add two of our new lists together.
-- This is how ++ is defined for normal lists.
-- infixr 5 ++
-- (++) :: [a] -> [a] -> [a]
-- [] ++ ys = ys
-- (x:xs) ++ ys = x : (xs ++ ys)
infixr 5 .++
(.++) :: List a -> List a -> List a
Empty    .++ ys = ys
(x:-:xs) .++ ys = x :-: (xs .++ ys)
-- And it works. If we wanted, we could write all of the functions that operated on list types.

-- Now it's time to make a binary search tree.
-- Here is what we can say about trees: It's either empty, or it's an element that contains a value and two trees.
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show, Eq, Read)
-- Our furst function is a utility function for just making a new tree with just one node.
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

-- Our second function is one that will insert an element into a tree.
treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
        | x == a = Node x left right
        | x < a  = Node a (treeInsert x left) right
        | x > a  = Node a left (treeInsert x right)

-- Function for determining if an element is in a tree
treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem _ EmptyTree = False
treeElem x (Node a left right)
        | x == a = True
        | x < a = treeElem x left
        | x > a = treeElem x right

-- Build a tree with "foldr treeInsert EmptyTree nums" where EmptyTree is the starting value, nums is the list, and treeInsert is our function.


-- TYPECLASSES 102 --

-- So far, we've learned how to use typeclasses, but we haven't learned how to make our own typeclasses.
-- Typeclasses are like interfaces in Java. It defines some behavior, and then, some types that can behave in that way are made instances of
-- that typeclass. So when we say a type is an instance of a typeclass, that means that we can use functions that typeclass defines with that
-- type (Show, Eq, Read, Ord, etc.).

-- Typeclasses have nothing to do with classes in things like Java or PHP or Python.

-- This is how the Eq typeclass is written.
--class Eq a where
--      (==) :: a -> a -> Bool
--      (/=) :: a -> a -> Bool
--      x == y = not (x /= y)
--      x /= y = not (x == y)
-- We define classes like "class [CLASSNAME] [TYPEVARIABLES] where" and then we state the functions of the typeclass. It's not manditory to
-- impliment the function bodies themselves but we need the type declarations.
-- We can't do much with a class once we have it, but once we start making some instances of that class, there is some nice funcitonality.
data TrafficLight = Red | Yellow | Green
-- This defines the different states of a traffic light. We didn't derive any classes for it because we are going to write up an instance by
-- hand
instance Eq TrafficLight where
        Red == Red = True
        Yellow == Yellow = True
        Green == Green = True
        _ == _ = False

-- So 'class' is for making a new Typeclass, but 'instance' is for making a data type an instance of a typeclasss.
-- In our example, we just replaced 'a' with 'TrafficLight' since it was the type.

-- Because == was defined along with /= in the Eq typeclass, we only needed to overwrite one of them in our instance declaratio because it
-- knows that in order to be true, it need to not be not true.
-- If Eq a where defined like
{-
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
-}
-- Then we would have to implement both == and /= in our instance.

-- Now let's make an instance of Show my hand.
instance Show TrafficLight where
        show Red = "Red light!"
        show Yellow = "Yellow light!"
        show Green = "Green light!"

-- We can also make a typeclass that is an extention of another typeclass. Like Num.
```

```haskell
-- class (Eq a) => Num a where
--    ...
-- So this is like writing "class Num a where" only our 'a' is a part of the Eq typeclass.

-- The difference between Maybe and TrafficLight is that Maybe isn't a concrete type, so it can be Maybe Int, Maybe String, or Maybe etc.
-- So how it is Eq'd since all functions need to use concrete types?
-- We can define Maybe as a non-concrete type, but the 'a' part of it is concrete.
{-
instance Eq (Maybe a) where
        Just x == Just y = x == y
        Nothing == Nothing = True
        _ == _ = False
-}
-- This is basically saying that we want to make all tyes that are Maybe a a part of Eq. This is how Haskell would derive the instance too.

-- Most of the time, class constraints in class declarations are sued for making a class a subclass of another class, while type constraints
-- in instance declarations are used for making sure that a certian requirements are met.

-- When you see that there is a type parameter used in a class declaration, you need to surround it with parenthesis to keep it a part of where
-- you want it.

-- If you want to see what instances your typeclass has, type :info TYPECLASS for information.


-- A YES-NO TYPECLASS --

-- Let's make a class that will give us True or False based on if a value is considered to be "0" or not.
class YesNo a where
        yesno :: a -> Bool
-- Now let's define some instances.
instance YesNo [a] where
        yesno [] = False
        yesno _ = True
instance YesNo Int where
        yesno 0 = False
        yesno _ = True
instance YesNo Bool where
        yesno = id -- The 'id' returns the same parameter that it was given.
instance YesNo (Maybe a) where
        yesno (Just _) = True
        yesno Nothing = False
instance YesNo (Tree a) where
        yesno EmptyTree = False
        yesno _ = True
instance YesNo TrafficLight where
        yesno Red = False
        yesno _ = True

-- Let's make a function that returns something based on what you put in.
yesnoIf :: (YesNo y) => y -> a -> a -> a
yesnoIf yesNo yesV noV = if yesno yesNo then yesV else noV


-- THE FUNCTOR TYPECLASS --

-- The Functor typeclass is for things that can be mapped over, such as lists. Lists are a part of the Functor typeclass.
{-
class Functor f where
        fmap :: (a -> b) -> f a -> f b
-}
-- This is how it's implimented.
-- It's just like the map function, which takes a function (a -> b) and a list [a] and returns a list of the second type [b].
-- fmap is the same as map, but map is only for lists.

-- Here's an instance of Functor
{-
instance Functor [] where
        fmap = map
-}
-- We didn't write [a] or something because the class definition doesn't ask for a concrete type. It's asking for a type constructor.
-- The [] is a type constructor, so it can work. But we can't involve 'a', 'b', etc.
-- Since fmap is the same as map, we get the same results when operating on lists.
fmapTest :: (a -> b) -> [a] -> [b]
fmapTest = fmap

mapTest :: (a -> b) -> [a] -> [b]
mapTest = map

-- The Functor typeclass can work on anything that's like a box. So anything that can holds a value or any number of values.
-- Maybe can be put into it as well.
{-
instance Functor Maybe where
        fmap f (Just x) = Just (f x)
        fmap f (Nothing) = Nothing
-}
-- Once again, we wrote it as a type constructor instead of a type parameter with an 'a' or 'm' or whatever because it takes a type constructor.
functorTest :: Maybe String
functorTest = fmap (++ "Hello there") (Just "hi.")

instance Functor Tree where
        fmap f EmptyTree = EmptyTree
        fmap f (Node x lTree rTree) = Node (f x) (fmap f lTree) (fmap f rTree)
-- implementing an instance of the Functor typeclass over Trees.

{-
instance Functor (Either a) where
        fmap f (Right x) = Right (f x)
        fmap _ (Left x) = Left x
-}
-- Here's how the Either type is instanced. Bebause it takes two type parameters, but Functor only wants one, we can give it one so that the
-- other is left unknown. We only apply the function to the Right value because the Right and Left are usually of two different types. If we
-- wanted to apply a function over both L and R, then they would have to be of the same type.
```

```
{-
instance (Ord k) => Functor (Map.Map k) where
        fmap f x = Map.fromList $ map (\(k,v) -> (k, (f v))) $ Map.fromList x
-}


-- KINDS AND SOME TYPE-FU --

class Tofu t where
        tofu :: j a -> t j a
data Frank a b = Frank {frankField :: b a} deriving (Show)

data Berry a b c = Berry {yabba :: c, dabba :: b a} deriving (Show)

instance Functor (Berry a b) where
        fmap f (Berry {yabba = y, dabba = x}) = Berry {yabba = f y, dabba = x}

-- This will all be talked about a bit more at a later time.
```