CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 08: Type Classes

- Review: What is a type class?
- Basic Type Classes: Eq, Ord, Enum, Integral, Show, Read, Enum, Functor

Next time: an extended example of creating your own type classes.

Reading: Hutton Ch. 3 & 8.1-8.5; Learn you a Haskell… also has some nice material on type classes (link on class web site)!

NOTICE: We are merging discussions B2 and A4; if you are in B2, please go to KCB 107 to meet with A4 from now on!

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5
Hutton Appendix B

Recall:

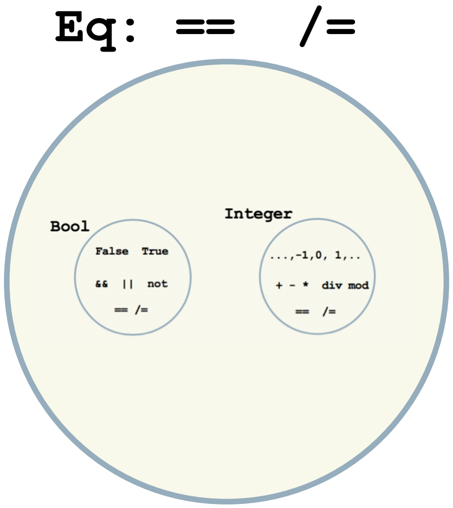
A type is a set of related values and a set of functions involving that type.

A type class is a set of types that share some overloaded functions.

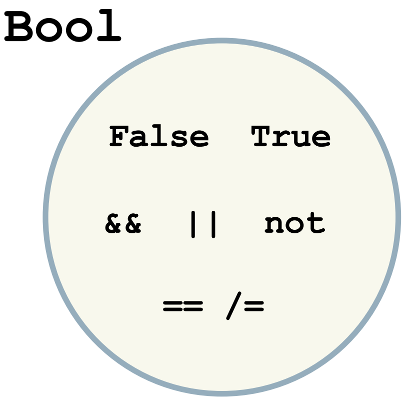
A type is an instance of a type class if

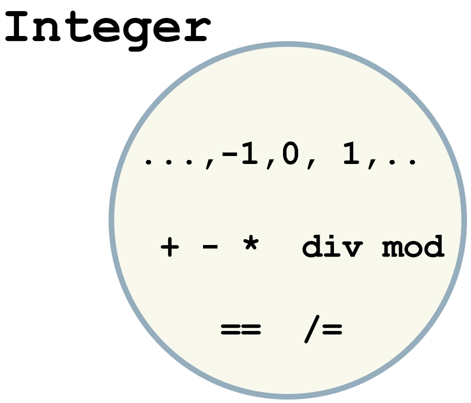
- ○ It **implements the functions** defining the class, and
- ○ It is defined as such by an **instance declaration** or

is derived by Haskell (more on this in a bit).



Example: both `Bool` and `Integer` are instances of `Eq`, defined by operators == and `/=`:





```
Main> False == True
False
Main> False /= False
False
Main> 4 == 8
False
Main> 2 /= 4
True
```

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

The type class **Ord** contains those types that can be totally ordered and compared using the standard relational operators:

```
(<)  :: Ord a => a -> a -> Bool

(>)  :: Ord a => a -> a -> Bool

(<=)  :: Ord a => a -> a -> Bool

(<=)  :: Ord a => a -> a -> Bool

min  :: Ord a => a -> a -> a

max  :: Ord a => a -> a -> a
```

A class constraint on a type variable restricts the types to those that are instances of the class.

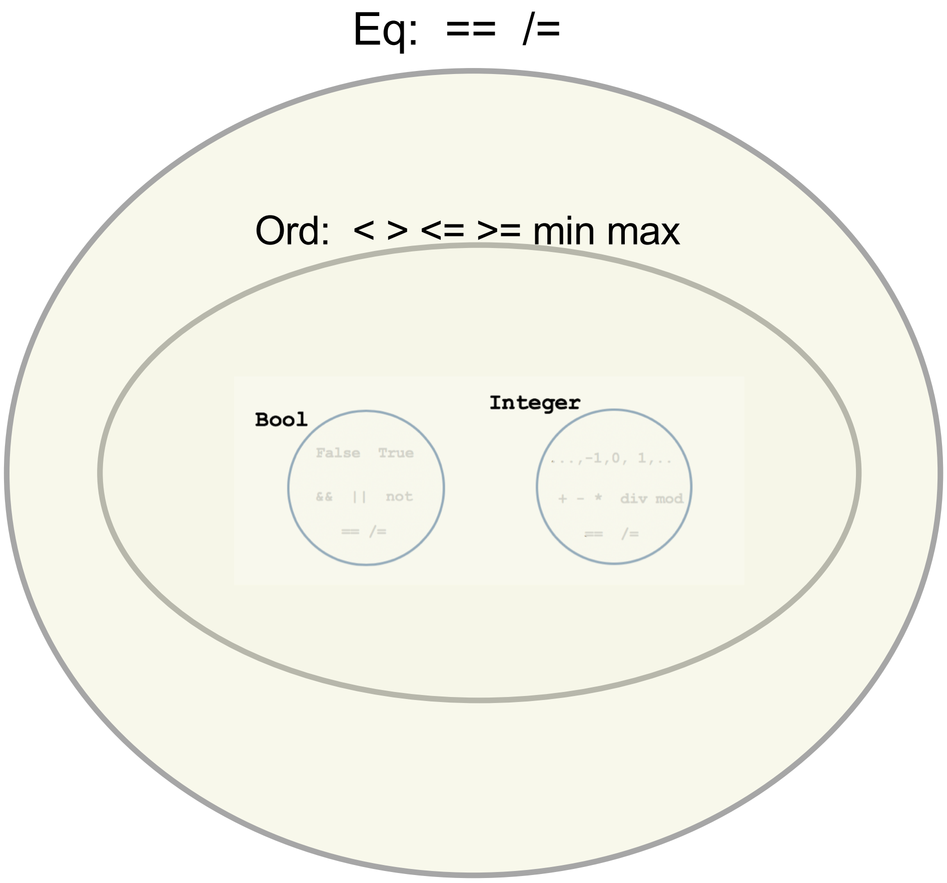
It is a kind of restricted polymorphism, similar to generic types in Java that implement some interface:

```
public static <T extends Comparable<T>> int compare(T t1, T t2){
    return t1.compareTo(t2);
}
```

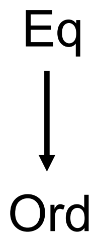
Type Classes and Overloading

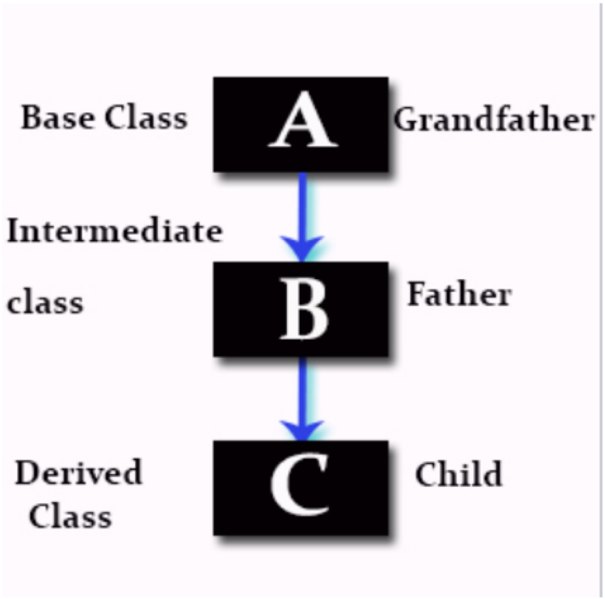
Reading: Hutton Ch. 3.8, 3.9, 8.5

Every instance of **`Ord`** is an instance of **`Eq`**, i.e., **`Ord`** ⊆ **`Eq`**, which is similar to inheritance in Java and object-oriented languages:



```
class Eq a => Ord a where
   .....
```





Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

`Bool`, `Char`, `Strings`, lists and tuples, and all the numeric types are instances of `Ord`:

```
Main> False < True
True
Main> 3 < 6
True
Main> 4.5 == 4.5
True
Main> [2,3] == [2,3]
True
Main> [1,2,3] < [1,3]
True
Main> [1,2,3] < [1,2,3,4]
True
Main> (2,3) >= (2,4)
False
Main> "Hi" < "Hi Folks!"
True
Main> max "hi" "there"
"there"
```

Relational tests on tuples and lists is lexicographic and recursive:

```
Main> [(2,"hi"),(5,"there")] <
      [(2,"hi"),(5,"folks")]
False
```

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

Enum – enumerable types

The **Enum** class contains types which can be put into 1-to-1 correspondence with the integers:

```
class  Enum a  where
    succ, pred      :: a -> a
    toEnum          :: Int -> a
    fromEnum        :: a -> Int
    enumFrom        :: a -> [a]                -- [n..]
    enumFromThen    :: a -> a -> [a]           -- [n,n'..]
    enumFromTo      :: a -> a -> [a]           -- [n..m]
    enumFromThenTo  :: a -> a -> a -> [a]      -- [n,n'..m]
```

To make your own data type an instance of `Enum`, you just have to define `toEnum` and `fromEnum`.

The important thing about the `Enum` class is the convenient syntax shown in the comments, which provides functionality similar to Python's `range(..)` function:

```
Main> [3..7]
[3,4,5,6,7]
Main> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

```
Main> [1,3..20]
[1,3,5,7,9,11,13,15,17,19]
Main> [1..]    -- infinite!
[1,2,3,4,5,6,7,8,9,10,11,12,1
14,15,16,17,18,19,20,21,22,23
```

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

`Num` – numeric types

The **`Num`** class contains numeric values, and consists of the following overloaded operators:

```
(+)  :: Num a => a -> a -> a

(*)  :: Num a => a -> a -> a

(-)  :: Num a => a -> a -> a

negate :: Num a => a -> a

abs  :: Num a => a -> a

signum :: Num a => a -> a
```

Hm...   where is division?

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

`Integral` – integer types

These are the instances of `Num` whose values are integers, and support integer division and modulus:

```
div :: Integral a => a -> a -> a

mod :: Integral a => a -> a -> a

Main> div 5 3
1
Main> 5 `div` 3
1
Main> mod 10 4
2
Main> 10 `mod` 4
2
```

Note that mod and div are prefix functions, to turn any function into infix, use back-quotes.

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

`Fractional` – floating-point types

These are the instances of Num whose values are floating point, and support floating-point division and reciprocation:

```
(/) :: Fractional a => a -> a -> a

recip :: Fractional a => a -> a
```

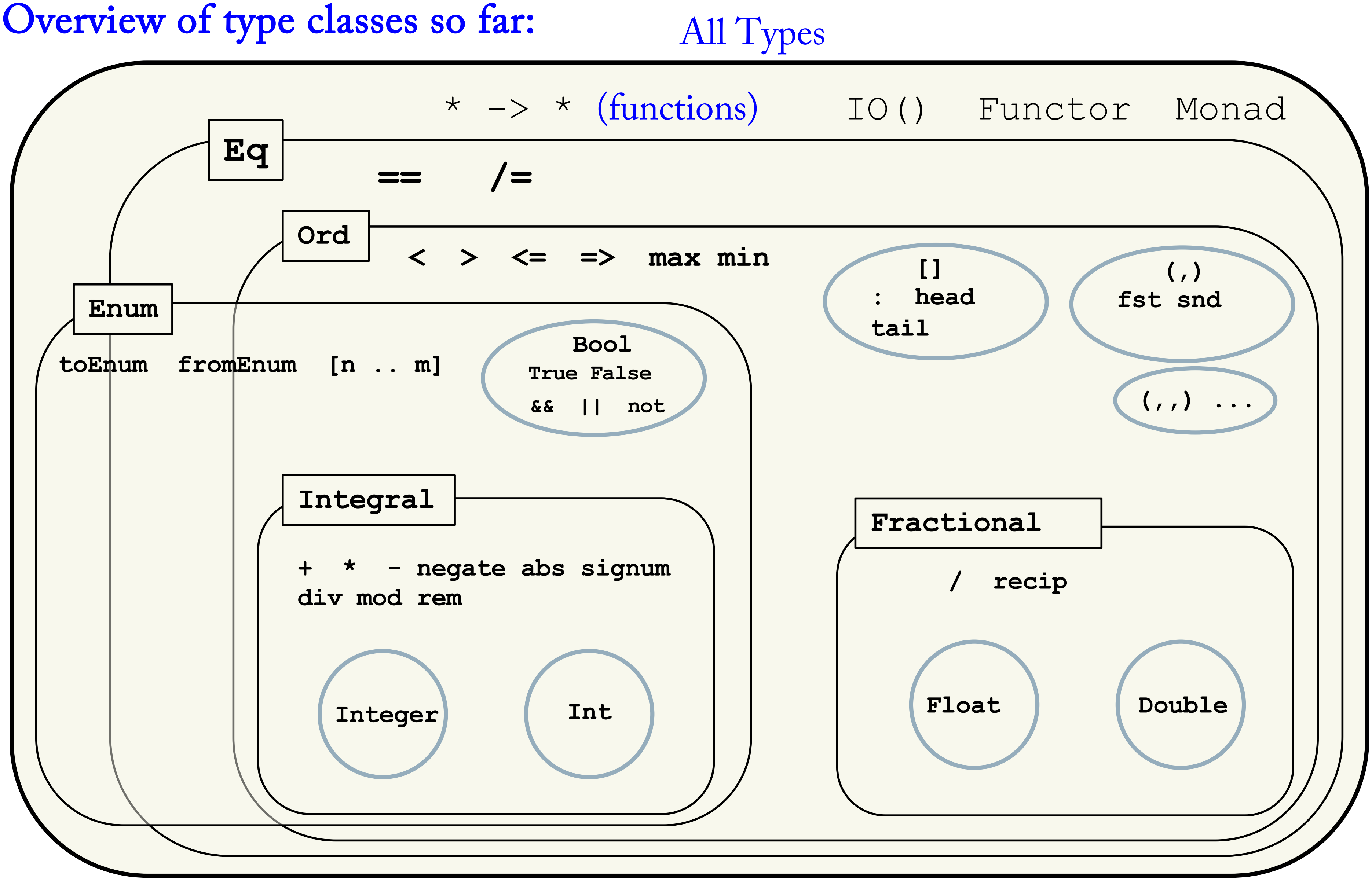
```
Main> 4.0 / 2.2
1.8181818181818181
Main> recip 5
0.2
Main>  4 / 2
2.0
Main> 5 / 2
2.5
Main> 5 / 2.2
2.2727272727272725
```

The symbols for integers are overloaded, so there is no ”type-coercion” from integer to float here. The values are already fractional!

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

Overview of type classes so far:



All Types

* -> * (functions)   IO()   Functor   Monad

Eq

== /=

Ord

< > <= => max min

Enum

toEnum fromEnum [n .. m]

Bool
True False
&& || not

[]
: head
tail

(,)
fst snd

(,,) ...

Integral

+ * - negate abs signum
div mod rem

Integer

Int

Fractional

/ recip

Float

Double

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

Practical Advice on using Numeric types in Haskell (for this course)

- Use only `Integer` and `Double` (or `Rational`) unless there is a good reason.

- Remember that ordinary integer constants (`3, 4, (-9)`) are overloaded and can be used in floating-point contexts:

```
Main> :t (/)
(/) :: Fractional a => a -> a -> a
Main> 3 / 4
0.75
Main> incr :: Integer -> Integer ; incr x = x + 1
Main> :t incr
incr :: Integer -> Integer
Main> (incr 3) / 4

<interactive>:21:1: error:
    • No instance for (Fractional Integer) arising from a use of '/'
    • In the expression: (incr 3) / 4
      In an equation for 'it': it = (incr 3) / 4
```

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

Practical Advice on using Numeric types in Haskell (for this course)

- Use `fromIntegral` to convert an `Integer` (or Int) expression into a `Fractional` typ to use in floating-point operations:

```
Main> :t incr
incr :: Integer -> Integer
Main> (incr 3) / 4
     <interactive>:21:1: error: etc.

Main> (fromIntegral (incr 3)) / 6
0.6666666666666666
```

- Use `truncate`, `ceiling`, and `round` to convert float-point into `Integral` types:

```
Main> truncate 3.4
3
Main> ceiling 3.4
4
Main> round 3.4
3
```

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

Show – types that have a String representation for printing

Show has a single method which converts its input to a Strint:

```
show :: Show a => a -> String
```
-- String == [Char]

```
Main> show 6
"6"
Main> show 5.6
"5.6"
Main> show True
"True"
Main> show [2,3,4]
"[2,3,4]"
Main> show (3,'a',True)
"(3,'a',True)”
Main> show 'a'
"'a'"
Main> show "hi there"
"\"hi there\""
```

All the basic Haskell types are instances of Show, but remember that function types are never in SHOW:

```
*Main> incr x = x+1
*Main> incr

<interactive>:67:1: error:
    • No instance for (Show (Integer -> Integer))
        arising from a use of ‘print’
        (maybe you haven't applied a function to
enough arguments?)
    • In a stmt of an interactive GHCi command:
print it
```

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

Read – types that have a String representation which can be converted into the actual type.

Read has a single method which converts a String into a type:

```
show :: Read a => String -> a
```

-- String == [Char]

However, because of **overloaded** symbols, you will need to specify what type to read into:

```
Main> read "5"
*** Exception: Prelude.read: no parse

Main> read "5" :: Integer
5

Main> read "5" :: Double
5.0
```

Type annotations can be added to any expression if needed to help Haskell figure out the type:

```
Main> x = (4::Float)/4.45
Main> x
0.8988764
Main> :t x
x :: Float
```

Type Classes: Functors

So far all our type classes have been with basic (non-function) data.

How do we make all this higher-order?

Let’s examine the `Functor` type class, which provides for map-like functions. Recall that `map` has the type

```
map :: (a -> b) -> [a] -> [b]
```

We would like to provide this kind of functionality for arbitrary data types, not just lists. For example, we’d like to map over `Maybe` or trees or ….

But what is the type of a map over an arbitrary data type? For example, over a `Maybe` it would have to be

```
map :: (a -> b) -> Maybe a -> Maybe b
```

This would allow us to apply a function inside a `Maybe`.

Type Classes: Functors

This is the purpose of the Functor type class, which is defined as follows:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

This is an example of a type class which doesn't provide any implementation, just requires that any instance must provide an implementation of `fmap`.

What is **`f`** in this declaration? It seems to be a type constructor, since it takes an argument: **`f a`**

In the type classes defined so far, the type variable stood for concrete data types such as **`Int`** or **`Bool`**. Now **`f`** is a type constructor which itself takes a single type parameter **`a`**.

Type Classes: Functors

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

To make a type an instance of the Functor data type, we need to declare it as an instance:

```
instance Functor [] where
    fmap = map
```

Notice the `[]` ; you might think we would write `[a]`, but that is a concrete type, and `[]` is provided as a type constructor.

Now `fmap` works the same as map:

```
Main> fmap (*2) [1..3]
[2,4,6]
Main> map (*2) [1..3]
[2,4,6]
```

Type Classes: Functors

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

To create a map on Maybe types, we can do this:

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

Notice carefully that we did not say

```
instance Functor (Maybe a) where
```

Functor wants a type constructor, not a type!

Type Classes: Functors

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b



instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing

Main> fmap (++ " Folks!") (Just ”Hi there “)
Just ”Hi there Folks!"
Main> fmap length (Just ”Hi there!“)
Just 9
Main> fmap (++  " Folks!")  Nothing
Nothing
Main> fmap (*2) (Just 200)
Just 400
Main> fmap (*2) Nothing
Nothing
```

Type Classes: Functors

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b



instance Functor Tree where
    fmap f Null = Null
    fmap f (Node left x right)
       = Node (fmap f left) (f x) (fmap f right)

Main>  fmap (*2) Null
Null
Main> (foldr treeInsert Null [5,7,3,12])
Node (Node Null 3 Null) 5 (Node Null 7 (Node Null 12 Null))

Main> fmap (*2) (foldr treeInsert Null [5,7,3,2,1,7])
Node (Node Null 6 Null) 10 (Node Null 14 (Node Null 24 Null))
```

Class and Instance Declarations

This material is taken directly from Hutton Ch. 8.5

A new type class can be declared using Haskell's **`class`** declaration; in fact, if you check out the Prelude (Hutton, Appendix B), you will see declarations of the standard classes discussed last time, starting with:

```
class Eq a where
      (==), (/=) :: a -> a -> Bool

      x /= y = not (x == y)
```

This means that for a type a to be an instance of the class Eq, it must have equality and inequality operators with the appropriate types.

Note that this assumes you will define `==`, and then /= is defined from `==`.

Class and Instance Declarations

Reading: Hutton Ch. 8.5

If you want to make a type an instance of Eq, you use an instance declaration, and provide implementations of the == operator (since /= is defined by default for the class Eq):

```
instance Eq Bool where
     False == False = True
     True  == True  = True
     _     == _     = False
```

But you can also override (substitute for) the default operators/functions.

```
instance Eq Bool where
     False == False = True
     True  == True  = True
     _     == _     = False

     False /= False = False
     True  /= True  = False
     _     /= _     = True
```

Class and Instance Declarations

Reading: Hutton Ch. 8.5

Classes can also be extended. For example, Ord is declared in the Prelude to extend Eq:

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

   min x y | x <= y    = x
           | otherwise = y            otherwise evals to False

   max x y | x <= y     = y
           | otherwise  = x
```

For a type to be an instance of Ord it must be an instance of Eq and also give implementations of the 6 operators shown above; but since default definitions for 2 of them are already given , you only need to give the missing 4:

```
instance Ord Bool where
   False < True = True
   _     < _    = False

   b <= c = (b < c) || (b == c)
   b > c  = c < b
   b >= c = c <= b
```

Class and Instance Declarations

Reading: Hutton Ch. 8.5

Derived Instances

When you define a new class, you want to avoid having to define operators/functions already defined somewhere else, so you make it an instance of built-in or already-defined classes, and thereby inherit the operators/functions already defined elsewhere.

The **`deriving`** mechanism allows you to do this in a simple way. For example, in the Prelude, the type Bool is actually defined by:

```
data Bool = False | True  deriving (Eq, Ord, Show, Read)
```

Note: When you do this, any component types used in your data declaration must already have these types:

```
data Shape = Circle Float | Rect Float Float   deriving (Eq, Show)
```

Float must already be an instance of Eq and Show

```
data Maybe a  = Nothing |  Just a  deriving   (Eq, Show)
```

Whatever type you instantiate for a must be an instance of the classes Eq and Show.