## Not the Whole Story

There are some aspects of the typeclass system that haven't been discussed yet

- Some classes depend on other classes
- Some classes are themselves polymorphic
- Some classes are associated with type constructors

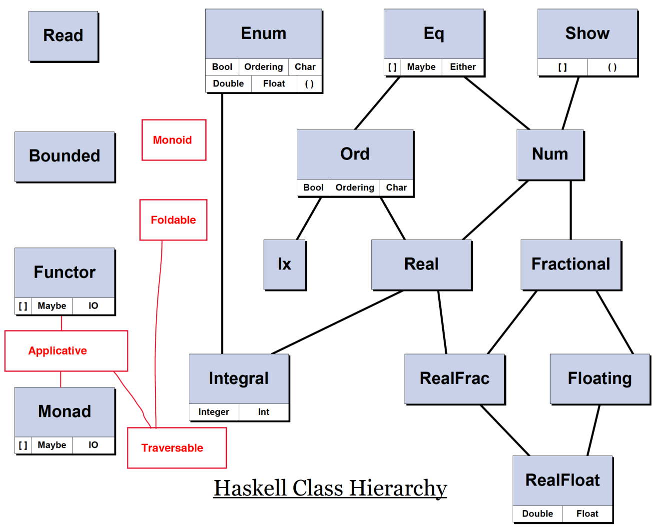
## Classes based on other Classes

- Here is part of the class declaration for `Ord`:

```
class  (Eq a) => Ord a  where
compare              :: a -> a -> Ordering
(<), (<=), (>=), (>) :: a -> a -> Bool
max, min             :: a -> a -> a
compare x y
     | x == y    =  EQ
     | x <= y    =  LT
     | otherwise =  GT
```

- The notation `(Eq a) =>` is a *context*, stating that the `Ord` class depends on the `Eq` class (why?)
- In order to define `compare`, we have to use `==`
  - So, for a type to belong to `Ord`, it must belong to `Eq`
  - Think of it as a form of inheritance

## Prelude Class Relationships




Haskell Class Hierarchy

Saeed Jahed, 2009, updated by A. Butterfield, 2016 to include class additions/modification introduced in GHC 7.10

## "Polymorphic" Type Classes (I)

How might we define an `Eq` instance for lists ?

- For `[Bool]`

```
instance Eq [Bool] where
  [] == []   =  True
  (b1:bs1) == (b2:bs2)   =  b1 == b2 && bs1 == bs2
  _ == _ = False
```

- For `[Int]`

```
instance Eq [Int] where
  [] == []   =  True
  (i1:is1) == (i2:is2)   =  i1 == i2 && is1 == is2
  _ == _ = False
```

- The red `==` above are where we use equality for `Bool` and `Int` respectively.
- Can't we do this polymorphically ?

## “Polymorphic” Type Classes (II)

- We can !

```
instance (Eq a) => Eq [a] where
  [] == []   =  True
  (x1:xs1) == (x2:xs2)   =  x1 == x2 && xs1 == xs2
  _ == _ = False
```

- We can define equality on `[a]` provided we have equality set up for `a`
- Here we are defining equality for a type constructor (`[]` for lists) applied to a type `a`:
  - so the class refers to a type built with a constructor

## Type-Constructor Classes

- Consider the class declaration for `Functor`

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

- Here we are associating a class with a *type-constructor* `f`
  - not with a type
  - See how in the type signature `f` is applied to type variables `a` and `b`.
  - So, `f` is something that takes a type as argument to produce a (different) type.

## Type Constructor Examples

- The `Maybe` type-constructor

```
data Maybe a = Nothing | Just a
```

- The `IO` type-constructor

```
data IO a = ...
```

- The `[]` type-constructor
  The type we usually write as `[a]` can be written as `[] a`
  i.e. the application of list constructor `[]` to a type `a`.

## Instances of `Functor`

- `Maybe` as a `Functor`

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

- `[]` as a `Functor`

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

- Both the above are straight from the Prelude.

## `Functor` instance for `Maybe`, with annotations

In more detail, first a reminder of the class definition:

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

Next, the instance, with type annotations:

```
instance  Functor Maybe  where
  fmap (f :: a -> b) (Nothing :: Maybe a)
                                                     =  Nothing :: Maybe b

  fmap f (Just (x :: a) :: Maybe a)
                                              = Just (f x :: b) :: Maybe b
```

## Instances and Type Declarations

- A type can only have one instance of any given class. Why? Because each instance is a specific implementation. Which one shoud the compiler pick?
- A type synonym therefore cannot have its own instance declaration.
  `type MyType a = ...`
  It simply is a shorthand for an existing type
- A user-defined algebraic datatype can have instance declarations
  `data MyData a = ...`
  In general we need to do this for `Eq`, `Show` in any case
- A user-cloned (new) type can also have instance declarations
  `newtype MyNew a = ...`
  A key use of `newtype` is to allow instance declarations for existing types (now "re-badged").

## Why not make `Expr` a member of the `Num` Class?

```
instance Num Expr where
  ...
```

This way we could then use standard arithmetic operators like `(+)` and `(*)` directly

```
Val 1.0 + Val 2.0 * Val 3.0
```

So, what does this involve? We need to look at the methods required for the `Num` class.

## Guided tour: the `Num a` Class

Class Members

```
(+), (-), (*)     :: a -> a -> a
negate            :: a -> a
abs, signum       :: a -> a
fromInteger       :: Integer -> a
```

Instances `Int`, `Integer`, `Float`, `Double`

Comments Required: `Eq`, `Show`
Most general notion of number available.
(Note lack of any form of division).

## Starting the Instance

We shall simply define each class function for `Expr` as a call to an external function, rather than defining them in place.

```
instance Num Expr where
  e1 + e2        =  addExpr e1 e2
  e1 - e2        =  subExpr e1 e2
  e1 * e2        =  mulExpr e1 e2
  negate e       =  negExpr e
  abs e          =  absExpr e
  signum e       =  signumExpr e
  fromInteger i  =  integerToExpr i
```

So far so good, but are storm-clouds looming?
What are the types of functions `addExpr` ... `integerToExpr` ?

## Typing the instance functions

We simply replace any occurrence of `a` in the class definition of `Num` by `Expr`.

```
addExpr       ::  Expr -> Expr -> Expr
subExpr       ::  Expr -> Expr -> Expr
mulExpr       ::  Expr -> Expr -> Expr
negExpr       ::  Expr -> Expr
absExpr       ::  Expr -> Expr
signumExpr    ::  Expr -> Expr
integerToExpr ::  Integer -> Expr
```

Ok, let's tackle `addExpr`

## Implementing (+) for Expr

Simplest approach, `(+)` for `Expr` is simply `Add`!

```
addExpr e1 e2 = Add e1 e2  -- or addExpr = Add !
```

```
> (Val 1.0) + (Val 1.0)
Add (Val 1.0) (Val 1.0)
```

Hmmm, maybe we'd prefer the following?

```
> (Val 1.0) + (Val 1.0)
(Val 2.0)
```

## Implementing (+) for Expr using simp

Lets use `simp` to see how far we can push things

```
addExpr e1 e2 = simp (Add e1 e2)  -- or addExpr = Add !
```

```
> (Val 1.0) + (Val 1.0)
(Val 2.0)
>(Var "x") + (Val 0.0)
(Var "x")
>(Var "x") + (Val 1.0)
Add (Var "x") (Val 1.0)
```

We can't use the Exercise One variant of `simp` that takes a dictionary. Why Not?

What dictionary would we use for `(Var "x") + (Val 1.0)` ?
There is no way to supply one, other than a built-in fixed dictionary behind the scenes.

## Moving on with `Expr` as `Num`

- Cases `Sub` and `Mul` are very similar to `Add`
- `negExpr` is trickier, but the following will do:

  ```
  negExpr e = simp (Sub (Val 0.0) e)
  ```

- `integerToExpr` is easy but seems strange:

  ```
  integerToExpr i = Val (fromInteger i)
  ```

  Here, `fromInteger` refers to the instance of `fromInteger` defined for the `Double` instance of `Num`.

## `Expr` is not adequate for `Num`

Now we run into trouble:

- `abs` cannot be implemented using any combination of add, subtract, multiply or divide.
  The only way forward would be to define a new `Expr` variant to represent the application of the absolute value operator, *e.g.*: `Abs Expr`
- `signum` cannot be implemented using any combination of add, subtract, multiply or divide.
  Again, the only way forward would be to define a new `Expr` variant to represent the application of the signume operator, *e.g.*: `SigNum Expr`

If this is worth it depends on our plans for `Expr` . . .