2 Type Classes

- **Show**: types whose elements can be printed
- **Eq**: types whose elements can be compared for equality
- **Num**: types supporting numeric operations
- **Functor**: type constructors that permit the definition of a map function
- **Applicative**: generalization of functors
- **Monad**: type constructors that encapsulate computations

(… we'll get to that later)
What Are Type Classes?

Type class $\approx$ set of types having a set of functions in common

Defining a type class: define names and types of required functions (member functions)

Make a type an instance of a class: give implementations for the member functions

For some classes, instances can be derived automatically

A type class can have (multiple) superclasses

$\text{Num} = \{\text{Int, Integer, Float, Double}\}$

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

instance Eq Grade where
  A == A = True
  ...

data Grade = ...
  deriving (Eq, Show)
```

Haskell type class $\approx$ Java interface
The Eq Class

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x/=y)
    x /= y = not (x==y)
```

Read as: “Type a is an instance of the Eq class if it has a function (==) or (/=) defined with the shown types.”

- **Class name:** Eq
- **Type variable representing instance type:** a
- **Default definitions:**
  - `x == y = not (x/=y)`
  - `x /= y = not (x==y)`
- **Member functions:**
  - `x == y` and `x /= y`

⇒ defining either (==) or (/=) suffices
An Eq Instance

**data** Time = Seconds Int  
| Minutes Int

**instance** Eq Time where

- Seconds x == Seconds y = x == y
- Minutes x == Minutes y = x == y
- Seconds x == Minutes y = x == y*60
- Minutes x == Seconds y = x*60 == y

Read as: "Time is an instance of the Eq class where the definition of (==) is as follows."
Data Types

```haskell
data Time = Seconds Int |
           Minutes Int
```

Definitions

1. Define a function `secs :: Time -> Int`.

   ```haskell
   secs (Minutes m) = 60 * m
   secs (Seconds s) = s
   ```

2. Define an `Eq` instance for `Time` using `secs`.

   ```haskell
   instance Eq Time where
   Seconds x == Seconds y = x == y
   Minutes x == Minutes y = x == y
   Seconds x == Minutes y = x == y * 60
   Minutes x == Seconds y = x * 60 == y
   ```
data IntSet = ESet
  | Ins Int IntSet

3. Discuss different approaches to defining an Eq instance for IntSet.

4. Define a function `member :: Int -> IntSet -> Bool`

5. Define a function `subset :: IntSet -> IntSet -> Bool`
Exercises

6. Define an `Eq` instance for `IntSet` using `subset`
Eq Class Constraints

What is the type of \texttt{elem}?

\begin{align*}
\text{elem} \ x \ (y:ys) &= x == y \ || \ \text{elem} \ x \ ys \\
\text{elem} \ _ \ [] &= \text{False}
\end{align*}

\begin{itemize}
\item \texttt{elem} :: \_ \rightarrow \_ \rightarrow \_ \quad \{ \text{has 2 parameters} \}
\item \texttt{elem} :: \_ \rightarrow [a] \rightarrow \text{Bool} \quad \{ 2nd \ par. \ is \ [ ] :: [a]; \ result \ is \ False \}
\item \texttt{elem} :: a \rightarrow [a] \rightarrow \text{Bool} \quad \{ \text{1st \ par. \ matches \ list \ args} \}
\item \texttt{elem} :: \text{Eq} \ a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool} \quad \{ (==) \ \text{function \ required \ for} \ a \}
\end{itemize}

Read as: "The type of \texttt{elem} is \( a \rightarrow [a] \rightarrow \text{Bool} \) for all types \( a \) that are instances of the \texttt{Eq} class."

Or: "If \( a \) is an instance of \texttt{Eq}, then the type of \texttt{elem} is \( a \rightarrow [a] \rightarrow \text{Bool} \)."
Use of Member Functions Leads To Class Constraints

elem :: Eq a => a -> [a] -> Bool
elem x (y:ys) = x == y || elem x ys
elem _ [] = False
The Ord Class

Ord is a subclass of `Eq`, since default definitions use `(==)`. To become an instance of `Ord`, a type must already be an instance of `Eq`.

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

\{ define either (<=) or compare \}

data Ordering = LT | EQ | GT
```

---

Type Classes
Exercises

7. Define an `Ord` instance for `Time`.
8. Define an `Ord` instance for `IntSet`.

```haskell
instance Ord IntSet where
    s < s' = subset s s' && not (s==s')
```

```haskell
data Time = Seconds Int |
            Minutes Int
```

```haskell
data IntSet = ESet |
              Ins Int IntSet
```

Bad idea!

Sets are only partially ordered

```haskell
instance Ord Time where
    t < t' = secs t < secs t'
```
**Ord Class Constraints**

---

**find** :: \(\text{Ord } a \Rightarrow a \rightarrow \text{Tree } a \rightarrow \text{Bool}\)

- \(\text{find } x \text{ Leaf } = \text{False}\)
- \(\text{find } x \text{ (Node } y \text{ l r) } | x = y = \text{True}\)
- \(\text{find } x \text{ (Node } y \text{ l r) } | x < y = \text{find } x \text{ l}\)
- \(\text{True } = \text{find } x \text{ r}\)

**qsort** :: \(\text{Ord } a \Rightarrow [a] \rightarrow [a]\)

- \(\text{qsort } [] = []\)
- \(\text{qsort } (x:xs) = \text{qsort } [y | y < xs, y \leq x] ++ [x] ++ \text{qsort } [y | y < xs, y > x]\)

---

Why isn’t also an \textbf{Eq} constraint required?
The Show Class

Types are made instances of Show to provide customized printable representations.

```haskell
class Show a where
    show :: a -> String
...

instance Show Bool where
    show True  = "T"
    show False = "F"
```
9. Define a Show instance for Time that produces output like 0:13 or 10:06.

(Just minutes and seconds)
Class Instances for Polymorphic Types

A `Show` instance for type `a` is required to define the `Show` instance for the type `Maybe a`.

```haskell
instance Show a => Show (Maybe a) where
  show (Just a) = show a
  show Nothing = "?"
```

```haskell
safeHd :: [a] -> Maybe a
safeHd [] = Nothing
safeHd (x:_ ) = Just x
```

```haskell
> map safeHd [[2,3],[],[5],[]]
[2,?,5,?]
```
10. Define an `Eq` instance for `List a`.

```haskell
instance Eq a => Eq (List a) where
  Nil       == Nil       = True
  Cons x xs == Cons y ys = x==y && xs==ys
  _         == _         = False
```

```haskell
instance Show a => Show (Maybe a) where
  show (Just a)  = show a
  show Nothing  = "?"
```

```haskell
data List a = Nil | Cons a (List a)
```

```haskell
data List a = Nil | Cons a (List a)

instance Show a => Show (List a) where
  show xs = "[" ++ showElems xs ++ "]

showElems :: Show a => List a -> String
showElems Nil          = ""
showElems (Cons x Nil) = show x
showElems (Cons x xs)  = show x ++ "," ++ showElems xs
```
Automatic Instances

Instance definitions for classes will be derived automatically through a top-down, left-to-right traversal of terms.

- `Nil < Cons 1 Nil` is `True`
- `Cons 2 Nil < Cons 2 (Cons 1 Nil)` is `True`
Multiple Class Constraints

12. What is the type of `isqrt`?

```
isqrt x = head [y | y <- [x,x-1 .. 1], y*y<=x]
```

```haskell
isqrt :: (Num a, Enum a, Ord a) => a -> a
```
Defining Type Classes

Why define your own type classes?

Generalize code /
Delay design decisions
Abstract from concrete representation

class Position p where
  distance :: p -> p -> Float
  neighbors :: p -> [p]
  ...

Preserve polymorphism
Instead of giving up polymorphism, limit it

class Pixel p where
  off :: p
  ...
  emptyRow :: Pixel p => Row p
  emptyRow = off:emptyRow

Generalize computational patterns

class Functor f where ...
class Monad m where ...
Type Classes vs. HOFs

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [y | y<-xs, y<=x] ++ [x] ++ qsort [y | y<-xs, y>x]
```

```
qsort :: (a -> a -> Bool) -> [a] -> [a]
qsort _ [ ] = [ ]
qsort le (x:xs) = qsort [y | y<-xs, y `le` x] ++ [x] ++
    qsort [y | y<-xs, not (y `le` x)]
```
# Abstractions and Levels

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Level</th>
<th>value</th>
<th>type</th>
<th>kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>value</td>
<td>3, True, succ</td>
<td>Int, Bool, Int -&gt; Int</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[], [3], Nothing</td>
<td>[a], [Int], Maybe a</td>
<td></td>
</tr>
<tr>
<td>variable</td>
<td></td>
<td>x, xs, y</td>
<td>a, b, c</td>
<td></td>
</tr>
<tr>
<td>function</td>
<td></td>
<td>succ, Just</td>
<td>[], Maybe</td>
<td>* -&gt; *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>div, (:)</td>
<td>(-&gt;), Assoc</td>
<td>* -&gt; *</td>
</tr>
</tbody>
</table>

*Type constructors* (overloading: `constructor` class)

*Types* (overloading: `type` class)
The Functor Class

Functor is a constructor class; the elements are not types but type constructors.

A functor is of kind * -> *

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

instance Functor Tree where
  fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)
  fmap f Leaf = Leaf

instance Functor [] where
  fmap = map
13. Define a type class `BiFunctor` for binary type constructors that has a member function `bmap`.

14. Define the pair type constructor `(,)` as an instance of `BiFunctor`. 
More on BiFunctors

define

class BiFunctor f where
  bmap :: (a -> c) -> (b -> d) -> f a b -> f c d

15. Define three functions mapL, mapR, and mapLR to map a function along the left, right, and both arguments of a type given by a binary type constructor.

\[
\text{mapL} :: \text{BiFunctor } f \Rightarrow (a \rightarrow c) \rightarrow f a b \rightarrow f c b \\
\text{mapR} :: \text{BiFunctor } f \Rightarrow (b \rightarrow c) \rightarrow f a b \rightarrow f a c \\
\text{mapLR} :: \text{BiFunctor } f \Rightarrow (a \rightarrow b) \rightarrow f a a \rightarrow f b b \\
\]
Using BiFunctors

```haskell
data Graph a b = G [a] [(Int,Int,b)]
```

16. Define `Graph` as an instance of `BiFunctor`.

17. What meaning do the functions `mapL` and `mapR` have for elements of type `Graph a b`?
Applying Wrapped Functions

How do you apply a function to a `Maybe` or `[ ]` value?

- `fmap succ (Just 3) ↞ 4`
- `fmap succ [2,3,4] ↞ [3,4,5]`

How do you apply a binary function to a `Maybe` or `[ ]` value?

- `??? (+) (Just 3) (Just 4) ↞ Just 7`
- `??? (+) [2,3,4] [5,9] ↞ [7,11,8,12,9,13]`

With `fmap` we can handle only one argument:

- `justPlus :: Maybe (Int -> Int)
  justPlus = fmap (+) (Just 3) ↞ Just (3+)`
- `listPlus :: [Int -> Int]
  listPlus = fmap (+) [2,3,4] ↞ [(2+),(3+),(4+)]`
Extending `fmap`

```
justPlus :: Maybe (Int -> Int)
jjustPlus = fmap (+) (Just 3) = Just (3+)

(Just 4) → Just 7
```

```
listPlus :: [Int -> Int]
listPlus = fmap (+) [2,3,4] = [(2+),(3+),(4+)]

[5,9] → [7,11,8,12,9,13]
```

We need a function:
```
<*> :: Maybe (Int -> Int) -> Maybe Int → Maybe Int
```

Compare with:
```
fmap :: (Int -> Int) -> Maybe Int → Maybe Int
```

We need a function:
```
<*> :: [Int -> Int] -> [Int] → [Int]
```

Compare with:
```
fmap :: (Int -> Int) -> [Int] → [Int]
```
The Applicative Class

**class** Functor f => Applicative f where
pure :: a -> f a
(<<*>>) :: f (a -> b) -> f a -> f b

**instance** Applicative Maybe where
pure = Just
Nothing <*> _ = Nothing
(Just f) <*> m = fmap f m

Applicative is a **subclass** of Functor and a **superclass** of Monad.

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

**instance** Applicative [] where
pure x = [x]
fs <*> xs = [f x | f <- fs, x <- xs]
Using Applicative

```haskell
class Functor f => Applicative f where
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

```
Just (+) <*> Just 3 <*> Just 4 ↠ Just 7
```

- `Maybe (Int -> (Int -> Int))`
- `Maybe Int`
- `Just (+) <*> Just 3 <*> Just 4 ↠ Just 7`
Using Applicative

\[(+) \ 3 \ 4 \rightarrow 7\]

\[((+) \ $ \ 3) \ $ \ 4 \rightarrow 7\]

\[\text{infixr} \ 0 \ $\]

\[\text{Just} \ (+) \ <*> \ \text{Just} \ 3 \ <*> \ \text{Just} \ 4 \rightarrow \ \text{Just} \ 7\]

\[\text{infixl} \ 4 \ <*>\]

\[\[(+)\,(*)\] \ <*> \ [2,3] \ <*> \ [5,9] \rightarrow [7,11,8,12,10,18,15,27]\]

= 

\[\{f \ x \ y \mid f \leftarrow [(+)\,(*)], \ x \leftarrow [2,3], \ y \leftarrow [5,9]\}]\]
Multi-Parameter Classes

… useful, but handle with care!

… best dealt with by reading Mark Jones’s paper.

**Haskell vs. Java Classes**

<table>
<thead>
<tr>
<th>Haskell</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>value (no state)</td>
<td>≈ object</td>
</tr>
<tr>
<td>function</td>
<td>method</td>
</tr>
<tr>
<td>type</td>
<td>class</td>
</tr>
<tr>
<td>type class</td>
<td>interface</td>
</tr>
<tr>
<td>defining class instance</td>
<td>implementing interface</td>
</tr>
<tr>
<td>defining subclass</td>
<td>extending interface</td>
</tr>
<tr>
<td>derived instances</td>
<td></td>
</tr>
<tr>
<td>default methods</td>
<td></td>
</tr>
<tr>
<td>multi-par. type classes</td>
<td></td>
</tr>
<tr>
<td>constructor classes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nested classes/ interfaces</td>
</tr>
</tbody>
</table>
More Practice

Define a `Show` instance for `IntSet` that produces output like `{2,3}`.  
(Don’t forget to remove duplicates.)

Generalize the definition of `IntSet` into a polymorphic definition `Set a`, 
where `a` represents the type of set elements.

Extend all instance definitions to `Set a`.

Define a binary tree type `Tree a b` that stores `a` values in nodes and `b` values in leaves.

Define `Eq` and `Show` instances for `Tree a b`.

Define a `BiFunctor` instance for `Tree`.

Define `Functor` instance for `Tree a`. 