Type Classes

A type class captures a common aspect for a set of different types.

- **Show**: contains types whose elements can be printed
- **Eq**: contains types whose elements can be compared for equality
- **Num**: contains types supporting numeric operations
- **Functor**: contains type constructors that permit the definition of a map function
- **Monad**: contains type constructors that encapsulate computations (... *we'll get to that later*)
What Are Type Classes?

**Type class** ≈ 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

Haskell type class ≈ Java interface

---

Num = {Int, Integer, Float, Double}

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

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

data Grade = ...
  deriving (Eq, Show)
The Eq Class

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

---

**Exercise:**

1. Define a function `secs :: Time -> Int`
2. Define `Eq` instance for `Time` using `secs`
Another Eq Instance

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

**Exercise:** Think about different approaches to define an `Eq` instance for `IntSet`

**Exercise:** (1) Define a function `member :: Int -> IntSet -> Bool`
(2) Define a function `subset :: IntSet -> IntSet -> Bool`
(3) Define `Eq` instance for `IntSet` using `subset`
Eq Class Constraints

```
elem x (y:ys) = x==y || elem x ys
elem _ [] = False
```

What is the type of `elem`?

```
elem :: _ -> _ -> _ { has 2 parameters }
elem :: _ -> [a] -> Bool { 2nd par. is [] :: [a]; result is False }
elem :: a -> [a] -> Bool { 1st par. matches list args }
elem :: Eq a => a -> [a] -> Bool { (==) function required for a }
```

Read as: “The type of `elem` is `a -> [a] -> Bool` for all types `a` that are instances of the `Eq` class.”

Or: “If `a` is an instance of `Eq`, then the type of `elem` is `a -> [a] -> Bool`.”
**Use of Member Functions Causes Class Constraints**

```haskell
remove :: Eq a => a -> [a] -> [a]
remove _ [] = []
remove x (y:xs) | x==y = remove x xs
               | otherwise = y:remove x xs

lookup :: Eq a => a -> Assoc a b -> Maybe b
lookup x ((y,z):ys) | x/=y = lookup x ys
                    | otherwise = Just z
lookup _ [] = Nothing
```

**type** Assoc a b = [(a,b)]
The Ord Class

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

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

data Ordering = LT | EQ | GT
Ord Instances

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

**data** IntSet = ESet | Ins Int IntSet

**Exercise:** (1) Define an **Ord** instance for **Time**
(2) Define an **Ord** instance for **IntSet**

**instance** Ord Time where
  \( t < t' = \text{secs } t < \text{secs } t' \)

**instance** Ord IntSet where
  \( s < s' = \text{subset } s \text{ s'} \& \& \text{not } (s == s') \)

*Bad idea! Sets are only partially ordered*
Ord Class Constraints

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

find :: Ord a => a -> Tree a -> Bool
find x Leaf = False
find x (Node y l r) | x==y = True  
                     | x < y  = find x l 
                     | True   = find x r

**Exercise:** Why isn’t also an `Eq` constraint required?
The Show Class

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

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

instance Show Bool where
    show True  = "T"
    show False = "F"
Show Instances

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

**Exercise:** Define a `Show` instance for `Time` that produces output like 0:13 or 10:06. (Just minutes and seconds)

```haskell
instance Show Time where
    show t = show min ++ ":" ++ show0 sec
    where
        min = s `div` 60
        sec = s `mod` 60
        s = secs t

    show0 s = (if s <= 9 then "0" else ")" ++ show s
```
Class Instances for Polymorphic Types

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

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

Example:

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

> map safeHd [[2,3],[],[5],[]] [2,?,5,?]
```
More Polymorphic Instances

**data** List a = Nil | Cons a (List a)

**Exercise:** Define an `Eq` instance for `List a`

**instance** Eq a => Eq (List a) *where*

```
Nil == Nil        = True
Cons x xs == Cons y ys = x==y && xs==ys
_ == _            = False
```

(==) function defined on type `a`

(==) function defined on type `List a`
More Polymorphic Instances

**data** List a = Nil | Cons a (List a)

**Exercise:** Define a `Show` instance for `List a` that produces output like `[]`, `[3]`, `[3,4]`

**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

```haskell
data List a = Nil | Cons a (List a) 
  deriving (Eq, Ord, Show)
```

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

Nil < Cons 1 Nil
Cons 2 Nil < Cons 2 (Cons 1 Nil)
Multiple Class Constraints

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

**Exercise:** What is the type of `isqrt`?

```
isqrt :: (Num a, Enum a, Ord a) => a -> a
```

isqrt x = head [y | y <- [x,x-1 .. 1], y*y<=x]
Defining Type Classes

Why define your own type classes?

Generalize code / Delay design decisions
Abstract from concrete representation

Preserve polymorphism
Instead of giving up polymorphism, limit it

Generalize computational patterns

```
class Position p where
  distance :: p -> p -> Float
  neighbors :: p -> [p]
...
class Pixel p where
  off :: p
  ...
emptyRow :: Pixel p => Row p
emptyRow = off:emptyRow
```

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

\texttt{qsort :: Ord a => [a] \rightarrow [a]}

\texttt{qsort [] = []}

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

Replace class constraint with function parameter

\texttt{qsort :: (a -> a -> \text{Bool}) \rightarrow [a] \rightarrow [a]}

\texttt{qsort _ [] = []}

\texttt{qsort \texttt{lt} (x:xs) = qsort [y | y<-xs, y `lt` x] ++ [x] ++ qsort [y | y<-xs, not (y `lt` x)]}
Type Classes vs. HOFs

**Exercise**: Replace the function parameter in the function `zipWith` by a type class
## Abstractions and Levels

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

Types
(overloading: **type** class)

Type constructors
(overloading: **constructor** class)
The Functor Class

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

\[
\text{class}\ \text{Functor}\ f\ \text{where}\\
fmap\ ::\ (a\ \rightarrow\ b)\ \rightarrow\ f\ a\ \rightarrow\ f\ b
\]

A functor is of kind \( * \rightarrow * \)

\[
\text{instance}\ \text{Functor}\ \text{Maybe}\ \text{where}\\
fmap\ f\ (\text{Just}\ x)\ =\ \text{Just}\ (f\ x)\\nmap\ f\ \text{Nothing}\ =\ \text{Nothing}
\]

\[
\text{instance}\ \text{Functor}\ \text{Tree}\ \text{where}\\
fmap\ f\ (\text{Node}\ x\ l\ r)\ =\ \text{Node}\ (f\ x)\ (fmap\ f\ l)\ (fmap\ f\ r)\\nmap\ f\ \text{Leaf}\ =\ \text{Leaf}
\]

\[
\text{instance}\ \text{Functor}\ []\ \text{where}\\
fmap\ =\ \text{map}
\]
BiFunctors

**Exercise**: Define a class `BiFunctor` with a member `bmap` for binary type constructors.

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

**Exercise**: Define `(,)` (pair type constructor) as an instance of `BiFunctor`.

```haskell
instance BiFunctor (,) where
  bmap f g (x,y) = (f x,g y)
```
More on BiFunctors

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

mapLR :: BiFunctor f => (a -> b) -> f a a -> f b b
mapLR f = bmap f f
```

**Exercise**: 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.

```haskell
mapL :: BiFunctor f => (a -> c) -> f a b -> f c b
mapL f = bmap f id

mapR :: BiFunctor f => (b -> d) -> f a b -> f a d
mapR f = bmap id f

mapLR :: BiFunctor f => (a -> b) -> f a a -> f b b
mapLR f = bmap f f
```
Using BiFunctors

Exercise: (1) Define a binary type constructor `Graph` with type parameters for node and edge labels.
(2) Define a `BiFunctor` instance for `Graph`.

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

instance BiFunctor Graph where
    bmap f g (G ns es) = G (map f ns)
    (map ((\(v,w,l)\)->(v,w,g l)) es)

Exercise: What meaning do the functions `mapL` and `mapR` have for elements of type `Graph a b`?

mapNodes :: (a -> c) -> Graph a b -> Graph c b
mapNodes = mapL

mapEdges :: (b -> c) -> Graph a b -> Graph a c
mapEdges = mapR
```
Multi-Parameter Classes

Overloading with coupled parameters

class (Field k, Group a) => VectorSpace k a where
  scalarMult :: k -> a -> a

... plus examples in the context of monads
Multi-Parameter Classes

```haskell
class Collection c where
    empty :: c a
    insert :: a -> c a -> c a
...

instance Collection [] where
    empty = []
    insert = (:)

Overloading with constrained parameters

class Collection c a where
    empty :: c a
    insert :: a -> c a -> c a
...

data ListSet a = Set [a]

instance Eq a => Collection ListSet a where
    empty = Set []
    insert x (Set xs) = Set (if elem x xs then xs else x:xs)
```
## 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>nested classes/ interfaces</td>
</tr>
</tbody>
</table>
Exercises

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

**Exercise**: (1) Generalize the definition of `IntSet` into a polymorphic definition `Set a`, where `a` represents the type of set elements.
(2) Extend all instance definitions to `Set a`.

**Exercise**: (1) Define a binary tree type `Tree a b` that stores `a` values in nodes and `b` values in leaves.
(2) Define `Eq` and `Show` instances for `Tree a b`.
(3) Define a `BiFunctor` instance for `Tree`.
(4) Define a `Functor` instance for `Tree a`. 