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**: type constructors (generalization of functors)
- **Monad**: 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

---

**Num** = \{Int, Integer, Float, Double\}

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

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

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

_Haskell type class ≈ Java interface_
The Eq Class

Class name

Type variable representing instance type

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

class Eq a where

\((==)\) :: \( a \rightarrow a \rightarrow \text{Bool} \)
\n\((/=)\) :: \( a \rightarrow a \rightarrow \text{Bool} \)

\( x == y = \text{not} (x/=y) \)
\( x /= y = \text{not} (x==y) \)

Member functions

Default definitions

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

```haskell
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."
Exercises

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

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

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

   `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`
Exercises

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

What is the type of `elem`?

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

- `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 Leads To Class Constraints

\[\text{elem} :: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}\]
\[
\text{elem } x \ (y:ys) = \ 
\text{x == y || elem } x \ yrhs
\]
\[
\text{elem } _\ [\] \ = \ 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.

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

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

data Ordering = LT | EQ | GT
Exercises

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

 dataType Time = Seconds Int |
| Minutes Int
data IntSet = ESet |
| Ins Int IntSet
Ord Class Constraints

```
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
```

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

Why isn't also an `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:_:xs) = Just x

> 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
  True
- Cons 2 Nil < Cons 2 (Cons 1 Nil)
  True
- Nil < Cons 1 Nil
  False
- Cons 2 Nil < Cons 2 (Cons 1 Nil)
  False
Multiple Class Constraints

12. What is the type of `isqrt`?

```haskell
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

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

- **Preserve polymorphism**
  Instead of giving up polymorphism, limit it

```haskell
class Pixel p where
  off :: p
  ...

emptyRow :: Pixel p => Row p
emptyRow = off:emptyRow
```

- **Generalize computational patterns**

```haskell
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>Value</th>
<th>Type</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>3, True, succ</td>
<td>Int, Bool, Int -&gt; Int, [a], [Int], Maybe a</td>
<td>*</td>
</tr>
<tr>
<td>variable</td>
<td>x, xs, y</td>
<td>a, b, c</td>
<td></td>
</tr>
<tr>
<td>function</td>
<td>succ, Just</td>
<td>[], Maybe</td>
<td>* → *</td>
</tr>
<tr>
<td></td>
<td>div, (:)</td>
<td>(-→), Assoc</td>
<td>* → *</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 $\star \to \star$

```haskell
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 [ ] where
  fmap = map

instance Functor Tree where
  fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)
  fmap f Leaf = Leaf
```
13. Define a type class `BiFunctor` for binary type constructors that has a member function `bmap`.

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

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

```
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
```

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.

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

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

mapLR :: BiFunctor f => (a -> b) -> f a a -> f b b
mapLR f = bmap f f
```
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) → Just 4`
- `fmap succ [2,3,4] → [3,4,5]`

How do you apply a binary function to 2 `Maybe` or `[ ]` values?

- `??? (+) (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

We need a function:

<*> :: Maybe (Int -> Int) -> Maybe Int -> Maybe Int

Compare with:

fmap :: (Int -> Int) -> Maybe Int -> Maybe Int

listPlus :: [Int -> Int]
listPlus = fmap (+) [2,3,4] = [(2+),(3+),(4+)] [5,9] ↠ [7,11,8,12,9,13]

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]`

Type Classes
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
```
Using Applicative

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

\[((+ \ $ \ 3) \ $ \ 4 \rightarrow 7)\] \hspace{1cm} \text{infixr 0 $}\]

Just \((+)<*>\) Just 3 \((<*>\) Just 4 \((<*>\) Just 7\]

\(\text{infixl 4 <*>)}\]

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

\[=\]

\[[f \times y \mid f \leftarrow [(+,\cdot)], \ x \leftarrow [2,3], \ y \leftarrow [5,9]]\]
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>nested classes/ interfaces</td>
<td></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 a `Functor` instance for `Tree a`.