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** ≈ 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 ≈ Java interface
The Eq Class

Class name

Type variable representing instance type

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

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

Member functions

Default definitions

\( \Rightarrow \) 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."
Exercises

1. Define a function \( \text{secs} :: \text{Time} \rightarrow \text{Int} \)

\[
\text{secs} (\text{Minutes } m) = 60 \times m \\
\text{secs} (\text{Seconds } s) = s
\]

2. Define an \( \text{Eq} \) instance for \( \text{Time} \) using \( \text{secs} \)

\[
\begin{align*}
\text{instance Eq Time where} \\
\text{Seconds } x \equiv \text{Seconds } y & \equiv x \equiv y \\
\text{Minutes } x \equiv \text{Minutes } y & \equiv x \equiv y \\
\text{Seconds } x \equiv \text{Minutes } y & \equiv x \equiv y \times 60 \\
\text{Minutes } x \equiv \text{Seconds } y & \equiv x \times 60 \equiv y
\end{align*}
\]
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 `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

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.

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

define either (<=) or compare
7. Define an `Ord` instance for `Time`.
8. Define an `Ord` instance for `IntSet`.
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]
qsort [] = []
qsort (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"
```
Exercises

9. Define a `Show` instance for `Time` that produces output like `0:13` or `10:06`. (Just minutes and seconds)
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,?]
```
Exercises

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)
```
More Polymorphic Instances


```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.

\[
\begin{align*}
\text{data } & \text{List } a = \text{Nil | Cons } a \ (\text{List } a) \\
& \text{deriving (Eq,Ord,Show)} \\
\text{Nil} & < \text{Cons 1 Nil} \quad \text{True} \\
\text{Cons 2 Nil} & < \text{Cons 2 (Cons 1 Nil)} \quad \text{True}
\end{align*}
\]
Multiple Class Constraints

12. What is the type of `isqrt`?

\[
isqrt x = \text{head} \left[ y \mid y \leftarrow [x, x-1 \ldots 1], y \cdot y \leq x \right]
\]
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>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 [] [3], Nothing</td>
<td>Int, Bool, Int -&gt; Int, [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 div, (:)</td>
<td>[], Maybe (-&gt;), Assoc</td>
<td>* -&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.

A functor is of kind $* \rightarrow *$

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 Maybe where
  fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)
  fmap f Leaf          = Leaf