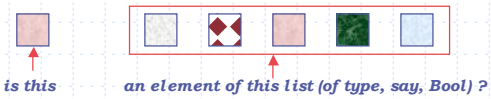
type classes



is this *an element of this list (of type, say, Bool) ?*

```
isinBList  :: Bool -> [Bool] -> Bool
isinBList x [ ]          = False
isinBList  x ( y : ys ] =  (x ==Bool y) || isinBList  x ys
```


if the list was of type [Int]

```
isinIList  :: Int -> [Int] -> Bool
isinIList x [ ]          = False
isinIList  x ( y : ys ] =  (x ==Int y) || isinIList  x ys
```

generically

```
isinList  :: a -> [a] -> Bool
```

and restrict a to only those types that have equality defined over them

overloading

there are two kinds of function which work over more than one class

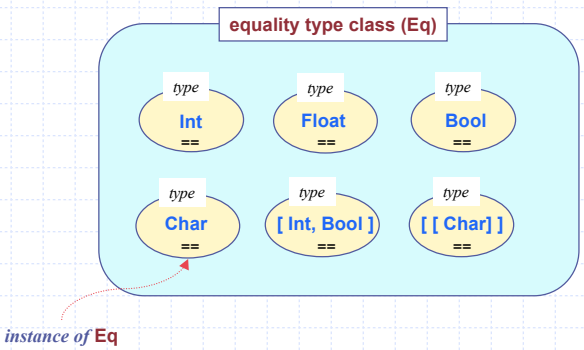
- **polymorphic** - single definition which works over all its types

```
length :: [a] -> Int

length [ ]       = 0
length (x : xs)  = 1 + length xs
```

- **overloaded** - (e.g. equality, +, show) that can be used for many types but have different definitions for different types

type classes - collection of types




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

```
same3 :: Int -> Int -> Int -> Bool
same3  m n p    = (m == n) && (n == p)
```

in the context of

```
same3  :: Eq a  => a -> a -> a -> Bool
same3  m n p    = (m == n) && (n == p)
```

thus restricting a to types such as:

- Char,
- Int,
- (Int, Bool),
- Float,

etc.

```
isinList  ::  Eq a   => a -> [ a ] -> Bool
isinList x [ ]           = False
isinList  x ( y : ys ] =  (x ==y) || isinList  x ys
```

a -
- Bool
- Char
- Int
- (Int, Int)

definition of Eq

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

signature

derived class Ord

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

```
compare x  y
  | x == y       = EQ
  | x <= y       = LT
  | otherwise  = GT
```

class Ord inherits the operations of Eq

class **Enum**

```
class Ord a => Enum a where
    toEnum          ::  Int  -> a
    fromEnum        ::  a  -> Int
    enumFrom        ::  a -> [a]
    enumFromThen    ::  a -> a -> [a]
    enumFromTo      ::  a -> a -> [a]
    enumFromThenTo  ::  a -> a -> a -> [a]
```

[n ..]
[n, m ..]
[n .. m]
[n, n' .. m]

fromEnum and **toEnum** convert between **a** and **Int**; in case of Char

```
ord :: Char -> Int
ord = fromEnum

ord :: Char -> Int
ord = fromEnum
```



```
class Bounded a where
    minBound, maxBound  ::  a
```

types
Int, Char, Bool, Ordering

```
type ShowS = String -> String

class Show a where
    showPrec ::  Int -> a -> ShowS
    show         :: a -> String
    showList     :: [a] -> ShowS
```

most types belong to **Show**

numeric types in Haskel

Int	fixed precision integers
Integer	all integers represented accurately
Float	floating point numbers
Double	Float in double precision
Rational	

the basic class to which all numeric types belong is Num


```
class (Eq a Show a) a => Num a where
    (+), (-), (*)          ::  a -> a -> a
    negate                 ::  a  -> a
    abs, signum            ::  a -> a
    fromInteger            ::  Integer -> a
    fromInt                ::  Int -> a

    x - y                  =  x + negate y
    fromInt                =  fromIntegral
```

integer types belong to the class Integral whose signature include:

```
quot, rem :: a -> a -> a
div, mod  :: a -> a -> a
```