# Polymorphism

## Polymorphic Functions

```
id :: a -> a
id x = x
```

- Takes in one input of any data type, and outputs the input

## Adhoc Polymorphism

Type of (==) = instance of a type class = `(==) :: Eq a => a -> a -> Bool`

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

Example:

```
data Fruit = Apple | Orange
```

Manual Way:

```
instance Eq Fruit where
  Apple == Apple = True
  Orange == Orange = True
  _ == _ = False
```

Better Way:

```
data Fruit = Apple | Orange
  deriving Eq
```

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

A list of type `[a]` can be compared if and only if we have a way of comparing the elements themselves (of type `a`).

```
instance Eq a => Eq [a] where
  [] == [] = True
  (x : xs) == (y : ys) = (x == y) && (xs == ys)
  _ == _ = False
```

Num class provides numerical operations:

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a -> a
  ...
  fromInteger :: Integer -> a -- Only example of type casting in
Haskell
```

# Show class

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

Can use `deriving Show`.

# Instances of a class

There can only be one instance of a class for each data type.