Type Classes + Quickcheck

CSE 130 11.19.21

```
add_int :: Int -> Int -> Int
add_double :: Double -> Double
```

```
add_int :: Int -> Int -> Int
add_double :: Double -> Double
```

Does polymorphism solve the problem?

```
add_int :: Int -> Int -> Int
add_double :: Double -> Double
```

Does polymorphism solve the problem?

add :: a -> a -> a

Too general!

```
add_int :: Int -> Int -> Int
add_double :: Double -> Double
```

Does polymorphism solve the problem?

add :: Num a => a -> a -> a

Examples

```
GHC.List
```

```
elem :: Eq a => a -> [a] -> Bool
maximum :: Ord a => [a] -> a
```

Data.Set

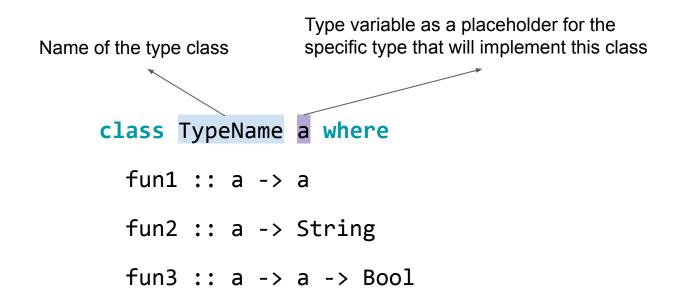
```
insert :: Ord a => a -> Set a -> Set a
```

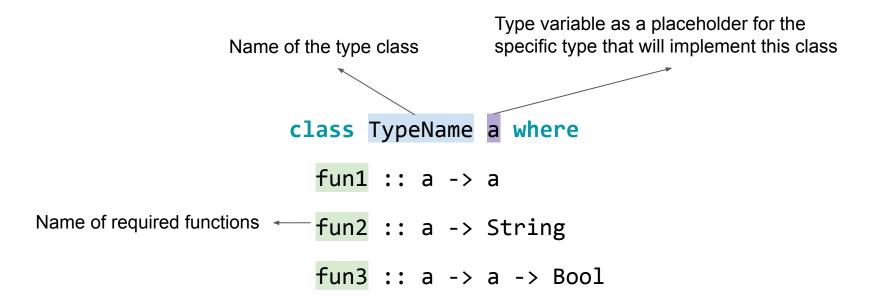
```
class TypeName a where
fun1 :: a -> a
fun2 :: a -> String
fun3 :: a -> a -> Bool
```

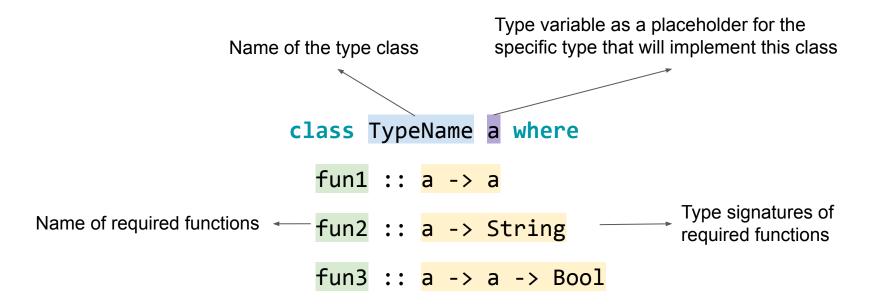
Name of the type class

```
class TypeName a where
fun1 :: a -> a
fun2 :: a -> String
```

fun3 :: a -> a -> Bool







class Num a where

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

class Eq a where

Context! Ord a implies Eq a class Eq a => Ord a where compare :: a -> a -> Ordering (<) :: a -> a -> Bool (<=) :: a -> a -> Bool (>) :: a -> a -> Bool (>=) :: a -> a -> Bool

How to use a type class

Color = Red | Blue | Green

Approach #1: deriving

Color = Red | Blue | Green deriving (Eq, Ord, Show)

Approach #2: defining instances

```
Color = Red | Blue | Green
```

instance Eq Color where

- (==) Red Red = True
- (==) Blue Blue = True
- (==) Green Green = True
- (==) _ _ = False

Practice

Property based testing with Quickcheck

Writing tests sucks!

A different approach

Property based testing:

Randomly generate inputs to your function, and check that certain *properties* hold

A different approach

Property based testing:

Randomly generate inputs to your function, check that certain *properties* hold **for all inputs**

Property = A boolean-valued function

For example

```
prop :: [a] -> Bool
prop xs = reverse (reverse xs) == xs
```

Practice