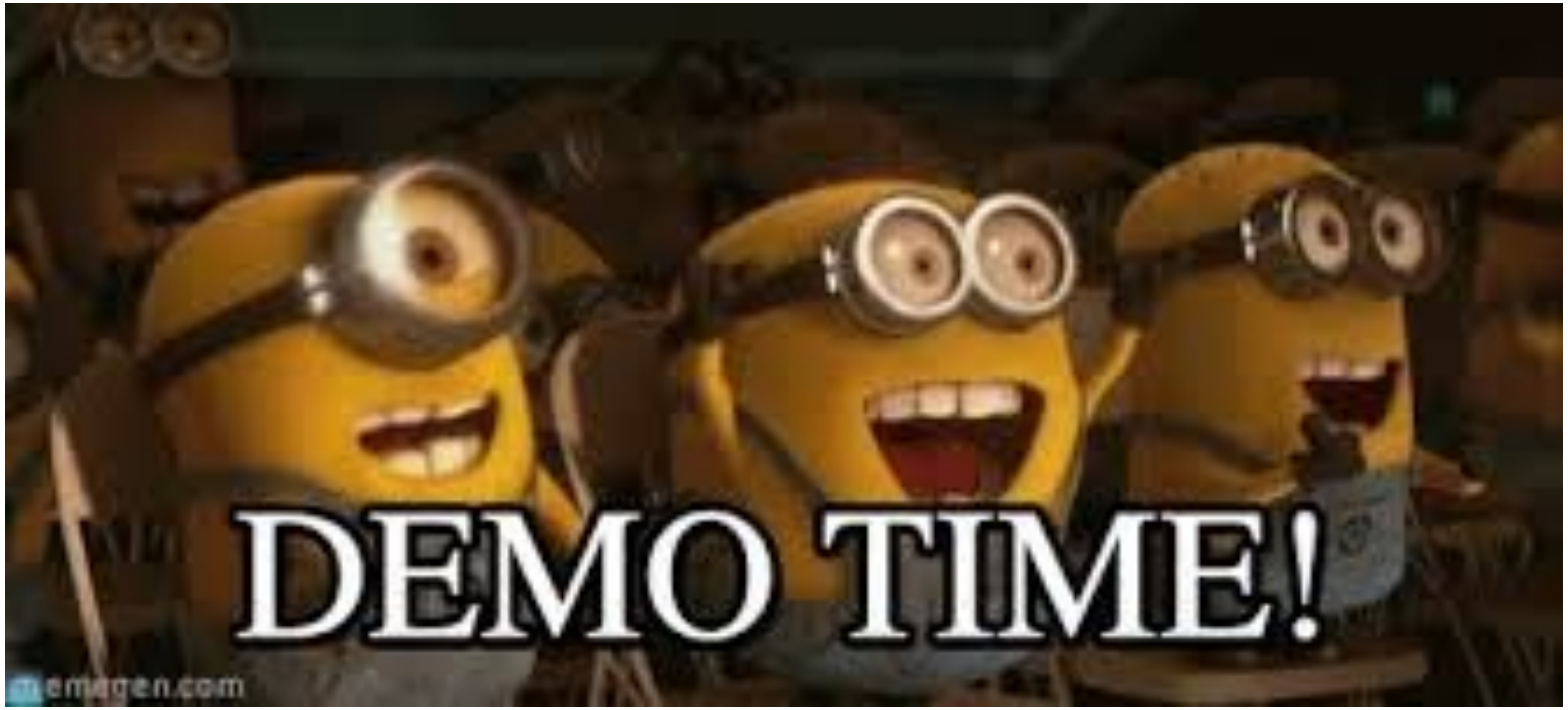
CS339: Abstractions and Paradigms for Programming

ADTs and Type Classes

Manas Thakur
CSE, IIT Bombay

Autumn 2024

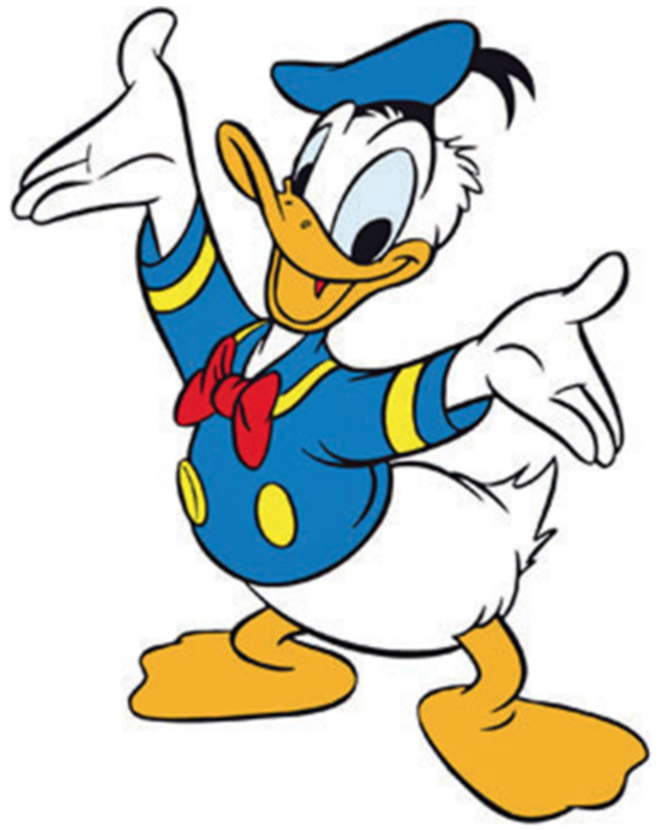
The best way to begin (and even finish!)



Types Types Types

- What is a type?
- What is it useful for?
- What is type checking?
- Difference between a strongly and a weakly typed language?
- Why shouldn't we always use the most general type?
- Static vs dynamic typing?
- Untyped languages?
- Haskell tries to offer the best of all worlds: Type inference!

Duck typing!
If it walks like a duck
and it quacks like a duck,
then it must be a duck.



Abstract Data Types

- Type with operations
- What's *abstract* about it?
  - Hides the representation details.
  - Recall our `Complex` example.
- Definition of `Bool` in Haskell:

```
data Bool = False | True
```

Data declaration

Data constructors

Examples of ADTs

```
data Move = North | South | East | West
```

Named constructors

- What are constructors in OO languages?
- Check the types of `North`, `South`, `East` and `West`.
- Now we can use these types in functions:

```
type Pos = (Int,Int)
move :: Move -> Pos -> Pos
move North (x,y) = (x,y+1)
move South (x,y) = (x,y-1)
move East  (x,y) = (x+1,y)
move West  (x,y) = (x-1,y)
```

```
moves :: [Move] -> Pos -> Pos
moves [] p = p
moves (m:ms) p = moves ms (move m p)
```

Data Constructors with Arguments

```
data Shape = Circle Float
           | Rectangle Float Float
```

- These constructors are essentially functions, which return a value of type Shape. Check their types again!
- We can again define functions using them:

Notice
pattern matching!

```
area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rectangle ...
```

- Why won't `area :: Circle -> Float` work?
  - Same reason why `foo :: True -> Int` won't work!

Type Parameters

```
data Maybe a = Nothing | Just a
```

Type parameter

- `Maybe` is a very useful type for handling failures.

```
head :: [a] -> a
head [] = error “Empty list”
head (x:_) = x
```

Can’t return this!

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x
```

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)
```

Type Classes

- What's the problem with the following types for +:

Too specific

Too generic

```
(+) :: Int -> Int -> Int
(+) :: Float -> Float -> Float
(+) :: a -> a -> a
```

- We would like to define (+) for only *number-like* types.
- Check the type of Haskell's built-in sort function:

```
> import Data.List
> :t sort
> sort :: Ord a => [a] -> [a]
```

- It sorts lists containing elements of any type `a` that belongs to the **type class** `Ord` ("`Ord a`" being a **type constraint**).

Type Classes (Cont.)

- Check the type of + now:

```
> :t (+)
> (+) :: Num a => a -> a -> a
```

`a` is a type variable

- Here is the declaration of the type class `Num` from Haskell:

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

- Class `Num` is a subclass of type classes `Eq` and `Show`.
- Any subclass of `Num` would need to define `(+)`, `(-)`, `negate`, etc.

Instantiating Type Classes

- A type can be made an instance of a type class by *instantiating* that class:

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

- Now any type can instantiate `Eq` by defining `==` (`/=` is already defined):

```
instance Eq Bool where
  False == False = True
  True  == True  = True
  _     == _     = False
```

- Sometimes we just want to use the pre-defined behavior.

Deriving from Type Classes

```
data Student = Student { name :: String,
                         rollNum :: Int }
```

- We can create *named* values as follows:

```
harry = Student { name = “harry”, rollNum = 123 }
```

- But we can’t “show” `harry` yet! (Notice the error.)
- Derive from something that knows how to *show*:

```
data Student = Student { name :: String,
                         rollNum :: Int
                       } deriving Show
```

- Now typing `harry` on the prompt works!

Recursive Types

- What does this type describe?

```
data Nat = Zero | Succ Nat
```

- Does this remind of something from **Lab2**?
  - Church numerals!

```
nat2int :: Nat -> Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n
```

```
add :: Nat -> Nat -> Nat
add m n = int2nat (nat2int m + nat2int n)
```

Classwork

The best way to finish (and even begin!)