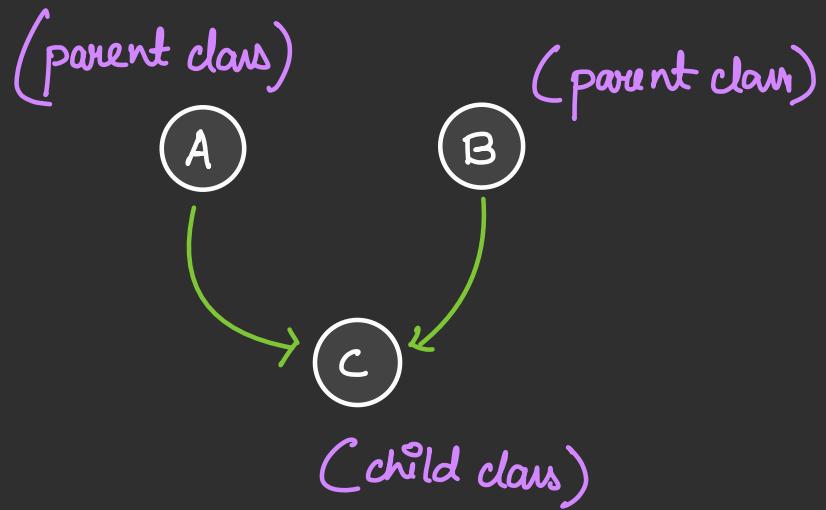


Lecture 3: Functional Programming

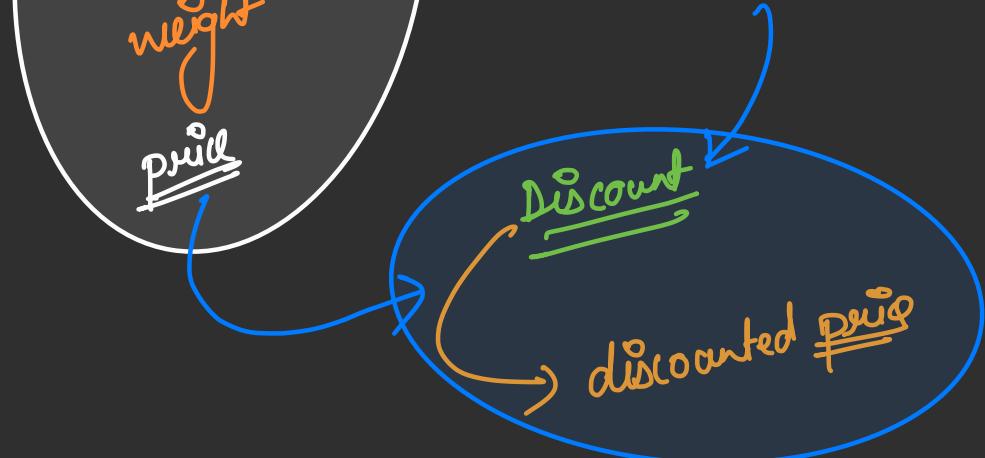
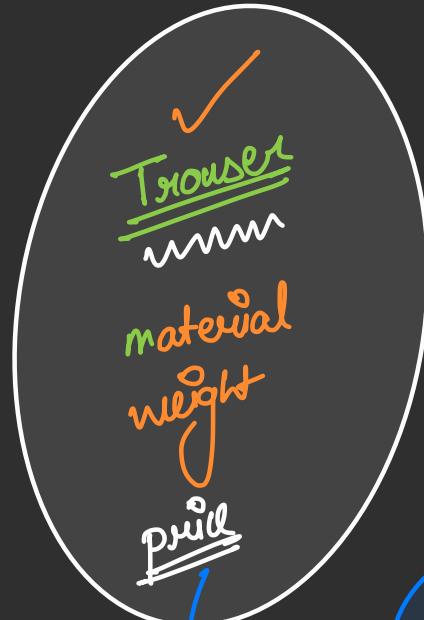
Agenda

- ① Multiple Inheritance & MRO ✓
- ② Functional programming ✓
- ③ Lambda Functions ✓
- ④ Higher Order Functions ✓
- ⑤ Decorators [Next Lecture]

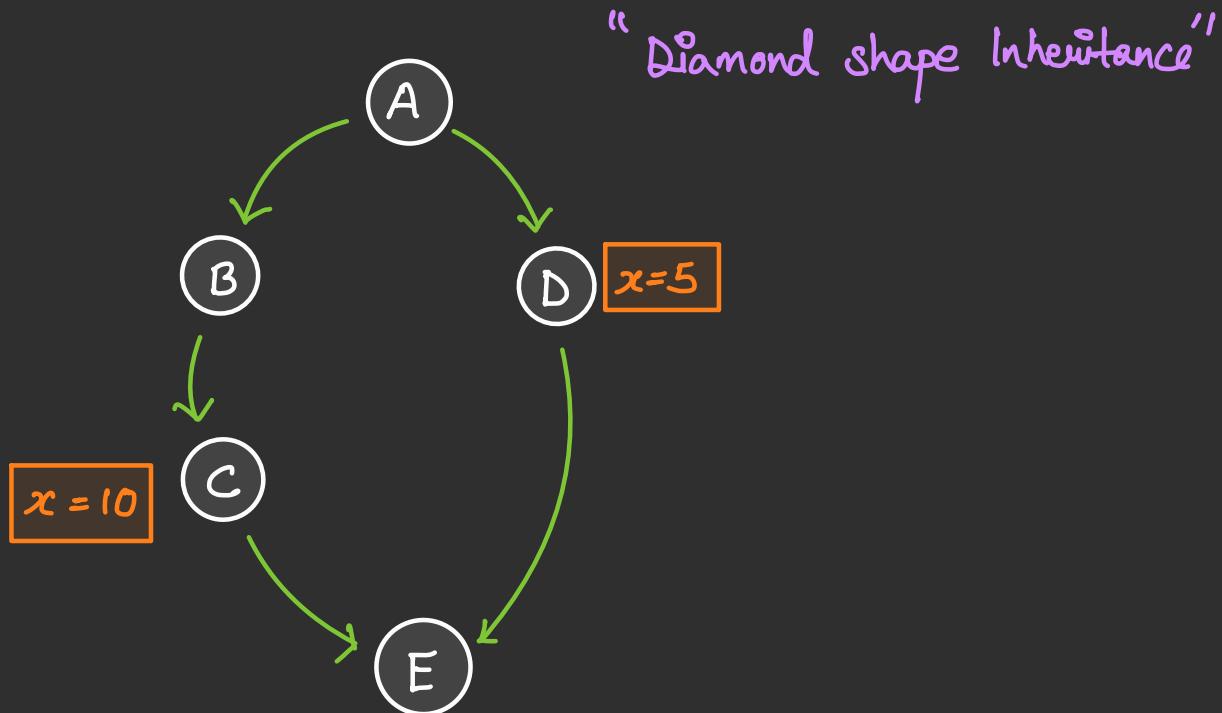
Multiple Inheritance



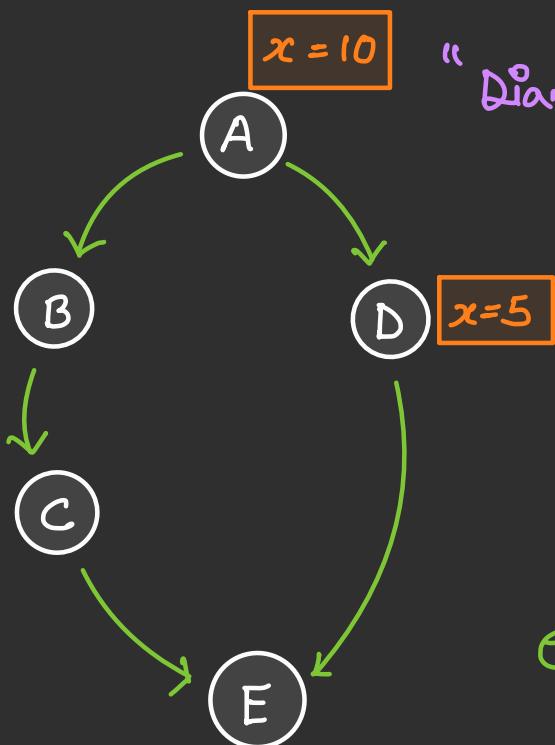
Retail Industry



Method Resolution Order (MRO)



Method Resolution Order (MRO)



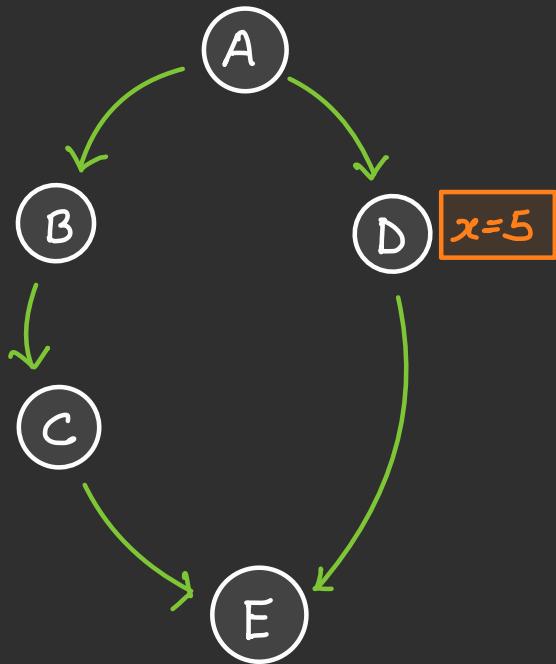
“Diamond shape Inheritance”

- 1 Left to Right
- 2 We go to parent when all child are covered

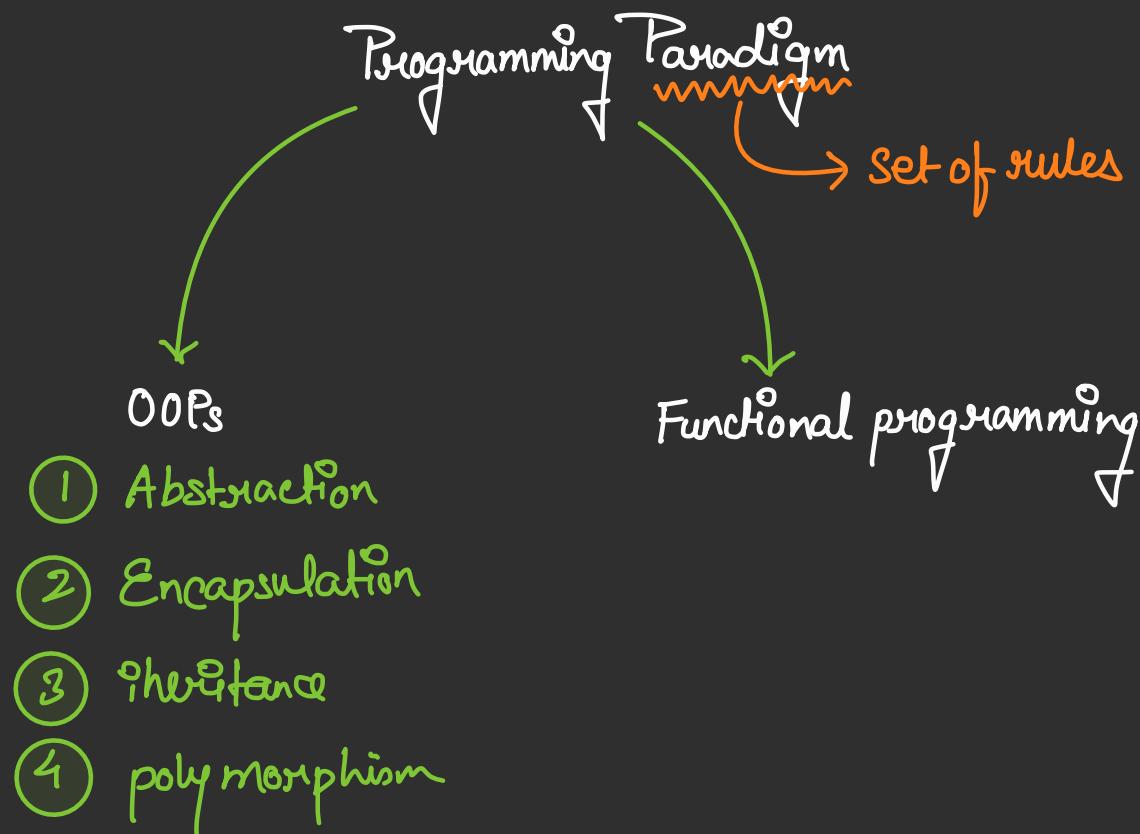
$e = EC \rightarrow \underline{\underline{x = 10}}$

Taking class E

$E \rightarrow C \rightarrow B \rightarrow D \rightarrow A$



Functional Programming



Function programming

It is a paradigm where computation is viewed as the evaluation of mathematical function

$a = [1, 2, 3, 4] \Rightarrow \text{square_num} = []$

loop & a^*
for num in a:
square_num.append(num * num)

Two major Reasons

- ① Immutable Data : It does not affect the original data .
- ② Declarative Style : In FP, you focus on "what you want to achieve" rather than "how" to achieve it

A hand-drawn diagram illustrating a variable pointing to a function. A green circle contains the letter 'a'. A green arrow points from this circle to the word 'def' in a green, handwritten-style font. Below 'def' is the word 'anything' and a green opening parenthesis '('. A green arrow points from the bottom of the 'def' text to a green 'return' keyword, which is followed by a green opening parenthesis '('. A green arrow points from the bottom of the 'return' keyword to a green closing parenthesis ')'. The entire diagram is drawn in green ink on a dark background.

```
graph TD; a((a)) --> def[def anything ()]; def --> return[return ()];
```

Imperative Approach

$a = [1, 2, 3, 4]$ \Rightarrow $\text{squared_num} = []$
for num in a:
 $\text{squared_num.append}(\text{num} * \text{num})$

loop & a^*a

Functional Approach

map, reduce, filter

$a = [1, 2, 3, 4]$

map(func, a) undefined func
list
 $(1, 4, 9, 16)$

def gen-exp (n):
 def exp (x):
 return x^n
 return exp

exp 5 = gen-exp (5)
exp 5 (2) = $2^5 = 32$

Diagram illustrating the execution flow:

- The outermost call is $\text{gen-exp}(5)$, which is highlighted with a green brace and a green return arrow.
- Inside gen-exp , the call $\text{exp}(x)$ is highlighted with a green brace and a green return arrow.
- The value x^n is highlighted with an orange circle and an orange arrow pointing to the result $2^5 = 32$.
- The final result 32 is highlighted with a green circle and a green arrow pointing to the value $\text{exp}^5(2)$.

~~exp 5~~ = $\text{gen-exp}(5)$
~~exp 5~~ (2) = 2^5