

# CS339: Abstractions and Paradigms for Programming

*Imperative Programming*

**Manas Thakur**  
CSE, IIT Bombay



Autumn 2024

# An objective world...

---

- The world consists of independent **objects**, whose behaviour *changes* over time.
- The changing behaviour can be captured using a snapshot of the **state** of individual objects.
- The objects interact with each other and influence each other's states.
- In programming terms, the state of an object is modelled using **local state variables**.
- We next need to see how could we model such *time-varying states*.



# A bank account

---

## ➤ Classwork:

- Write a function to 'withdraw' a given amount if an account has enough balance.
- Another function should simulate the following over an account:
- Initial balance: 100
- Amounts to withdraw: 20, 90, 30



# Attempt 1

---

```
(define (withdraw balance amount)
  (if (>= balance amount)
      (- balance amount)
      (error "Insufficient balance"))))

(define (account-ops)
  (withdraw (withdraw (withdraw 100 20) 90) 30))

(account-ops)
```

➤ Output: “Insufficient balance”

# Attempt 2

---

```
(define (withdraw balance amount)
  (if (>= balance amount)
      (- balance amount)
      "Insufficient balance"))
```

```
(define (account-ops init-bal amounts)
  (if (null? amounts)
      init-bal
      (let ((bal (withdraw init-bal (car amounts))))
        (if (eq? bal "Insufficient balance")
            (account-ops init-bal (cdr amounts))
            (account-ops bal (cdr amounts)))))))
```

```
(account-ops 100 '(20 90 30))
```

**Problem:** We need to maintain the balance of each account explicitly while writing our operations.

- Output: 50
- What if we had two accounts? 1000 accounts?



# What we want

---

- An account that maintains its balance by itself.
- Users should not have to “remember” the balance of each account and supply it to the *withdraw* procedure.
- What does it need:
  - A way to *remember* the balance based on the history of transactions.
  - Which needs a way to *update* the balance after each transaction.
  - Which needs **mutation**!





# Mutations, assignments, states

---

- Scheme provides a special form `set!` to update the value of a variable:
  - `set! <name> <new-value>`
- Traditionally, this is called an **assignment**.
- **Mutations** enabled by assignments lead to the notion of a state after each assignment.

Pronounced as  
“set bang”

# Imperative Programming

---

- An assignment updates the state.
- A **sequence of assignments** leads to a sequence of states.
- Hence, assignments are *statements* (against expressions).
- Each assignment is like a *command* to change the state.
- (Apple Dictionary) **Imperative**. giving an authoritative command.
- The practice of giving a sequence of commands to update states, in the form of assignment statements, defines the paradigm of **imperative programming**.

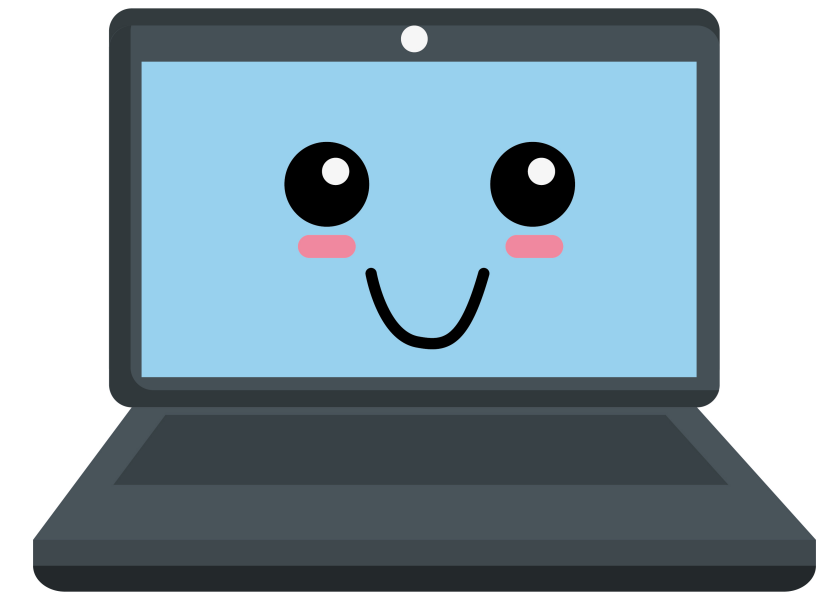




# Account withdrawals in an imperative world

---

```
(define balance 100)
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))
```



- Notice the **begin** keyword for *sequencing* multiple statements
- PC Question:
  - Can you recall a place where we have used ‘begin’ implicitly?
  - Answer: What about multiple defines in a procedure?

# Writing “withdrawal processors”

```
(define W1 (make-withdraw 100))  
(define W2 (make-withdraw 100))
```

```
> (W1 50)
```

```
50
```

```
> (W2 70)
```

```
30
```

```
> (W2 40)
```

```
“Insufficient funds”
```

```
> (W1 40)
```

```
10
```

```
(define (make-withdraw balance)  
  (lambda (amount)  
    (if (>= balance amount)  
        (begin (set! balance (- balance amount))  
                balance)  
        "Insufficient funds"))))
```

Multiple accounts without any hassle!



# But an account can also get money in!

---

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
                balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT"
                        m))))
  dispatch)
```

*A pakka object-oriented account :-)*



# Pros of Assignments

---

- Ability to **model the real world** in terms of objects with local state
- Proper **modularity**: independence of objects and users
- Simpler bank account code!



# Cons of Assignments

---

- Our nice, simple **substitution model** of procedure application **goes away** for a toss!
- Consider the following two procedures:

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
(define W (make-simplified-withdraw 25))
(W 20)
5
(W 10)
-5
```

**Imperative**

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))
(define D (make-decrementer 25))
(D 20)
5
(D 10)
15
```

**Functional**





# Con 1: Good-bye substitution model

```
(define (make-decrements balance)
  (lambda (amount)
    (- balance amount)))
```

```
((make-decrements 25) 20)
==> ((lambda (amount) (- 25 amount)) 20)
==> (- 25 20)
==> 5
```

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

```
((make-simplified-withdraw 25) 20)
==> ((lambda (amount) (set! balance (- 25 amount)) 25) 20)
==> (set! balance (- 25 20)) 25
==> Set balance to 5 and return 25
```

**Consequence: Simple substitution may lead to unexpected results.**





# Con 2a: Bye-bye referential transparency

Are D1 and D2 the same?

> Arguably yes, because each could be substituted for the other.

Are W1 and W2 the same?

> They look to be, but aren't:

(W1 20)

5

(W1 20)

-15

(W2 20)

5

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))
```

```
(define D1 (make-decrementer 25))
(define D2 (make-decrementer 25))
```

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

```
(define W1 (make-withdraw 25))
(define W2 (make-withdraw 25))
```

**Consequence: Can't reason about correctness by “looking” at the program.**



# Con 2b: Identity crisis

---

- Two bank accounts are different even if they have the same balance.
- A bank account remains to be the same even if its constituent data items or fields (i.e., the balance) changes.
- A rational number  $2/3$  is not the same if its constituent data items (either the numerator or the denominator) change.
- What's the **identity** of a bank account then?

**Consequence: Extra efforts required for implementing 'equals' methods!**



# Con 3: Inducing order into life

- Consider this ‘imperative’ factorial program:

```
(define (factorial n)
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! product (* counter product))
                  (set! counter (+ counter 1))
                  (iter))))
    (iter)))
```

Next class:

*A substitution for substitution!*

- What if we wrote or performed the assignments in opposite order:

```
(set! counter (+ counter 1))
(set! product (* counter product))
```

**Consequence: Difficult optimization/parallelization.**

