Problem 1: Student Grade Evaluator

Create a Student struct that holds a student's name, subject-wise marks (dictionary), and total marks as a computed property. Use string methods to validate the student's name (must be non-empty, capitalized). Based on the percentage, use an enum Grade (A, B, C, Fail) to assign a grade inside a method.

Create a function that accepts a Student?, uses guard let to unwrap it, and prints a summary using a switch on the grade.

Problem 2: Banking System with Inheritance and Optionals

Create a base class BankAccount with subclasses SavingsAccount and CurrentAccount. Each class should have methods like deposit(amount:) and withdraw(amount:). The withdrawal should fail if balance goes below a limit (using optional return and control flow). Use optional chaining to safely call methods on potentially nil accounts.

Use a function to get an account by ID (returns optional), and gracefully handle all the edge cases using if let and guard let.

Problem 3: Library Book Tracker

Create a Book struct with properties like title, author, status (enum: .available, .issued, .reserved) with a property observer that logs when status changes. Create a Library class that has an array of books and a dictionary mapping usernames to borrowed books. Use loops and control flow to check availability, issue a book, and print current borrowers.

Problem 4: Custom Pizza Order System

Define a Pizza struct with toppings ([String]), size, and a computed property price. Add a failable initializer that fails if no toppings or size is invalid. Create an enum Size: String with values .small, .medium, .large. Create a function placeOrder(for:) that takes a Pizza? and prints the final price. Demonstrate string filtering, capitalization of toppings, and optional chaining with .joined().

Problem 5: Ride Booking App

Design a base class Ride and subclasses like BikeRide, CarRide, AutoRide. Each ride has a cost-per-km and a function calculateFare(distance:). Create an enum RideStatus { case requested, confirmed, cancelled }. Use a dictionary to track rides by user. Implement a function that confirms a ride based on certain conditions (e.g., minimum distance), using guards to validate input and optional chaining to safely access ride status or properties.

Problem 6: Payment Processing with Type Casting

Design a Swift-based payment system using class inheritance. Create a base class for all payment methods and subclasses for CreditCard, UPI, and Cash, each with their own specific data and functionality. For CreditCard, ensure that only valid card numbers can be used to create an instance. Store different users and their payment methods in a dictionary of type [String: Any], including some entries that don't represent valid payment methods.

Write a function that takes a user's name, inspects the runtime type of their payment method, and performs the appropriate action. Use safe type casting and optional handling to ensure your function doesn't crash on invalid or missing data.

Solutions

✅ Problem 1: Student Grade Evaluator

C/C++

```
import Foundation  // Importing Foundation framework (useful if you plan formatting or advanced features)
```

C/C++

```
// Step 1: Create an enum to represent student grades.
// Using enums ensures that only valid grade values are used in the program.
enum Grade {
    case A    // For 90–100%
    case B    // For 75–89.99%
    case C    // For 60–74.99%
    case Fail // For below 60%
}
```

C/C++

```
// Step 2: Create a structure to represent a Student.
// Structs in Swift are lightweight and well-suited for holding related data.
struct Student {

    // Property: name of the student
    var name: String {
        didSet {
            // Property observer to automatically capitalize the name when set
            name = name.capitalized
        }
    }
    // Dictionary to hold marks for each subject
    var marks: [String: Int]

    // Computed property: calculates the total marks from the
dictionary using a loop
    var total: Int {
        var sum = 0  // Initialize a sum variable
        for (_, value) in marks {
            // For each key-value pair in the dictionary, take
the value (marks) and add to sum
            sum += value
        }
        return sum  // Return the final sum of all marks
    }

    // Computed property: calculates percentage based on total
and number of subjects
    var percentage: Double {
        // Convert total and subject count to Double for decimal
precision
        return Double(total) / Double(marks.count)
    }

    // Method: Determines the grade based on percentage using a
switch statement
    func grade() -> Grade {
        switch percentage {
        case 90...100:
            return .A  // Excellent
        case 75..<90:
            return .B  // Good
        case 60..<75:
            return .C  // Average
        default:
            return .Fail  // Below 60% is considered a fail
        }
    }
}
```

C/C++

```
// Step 3: Function to print the student report
// Accepts an optional Student (Student?) to demonstrate optional handling
func printStudentReport(for student: Student?) {

    // Use guard let to safely unwrap the optional student.
    // If student is nil, print error and exit early.
    guard let student = student else {
        print("Invalid student record")
        return
    }

    // Once unwrapped, we use the student object below:

    print("Name: \(student.name)")  // Print the capitalized name
    print("Total Marks: \(student.total)")  // Show manually calculated total marks
    print("Percentage: \(student.percentage)%")  // Show percentage

    // Use a switch statement to display the grade based on enum
    switch student.grade() {
    case .A:
        print("Grade: A (Excellent)")
    case .B:
        print("Grade: B (Good)")
    case .C:
        print("Grade: C (Average)")
    case .Fail:
        print("Grade: Fail (Needs Improvement)")
    }
}
```

C/C++

```
// Step 4: Create a sample student using the memberwise initializer
let s1 = Student(
    name: "aman",   // lowercase name to test capitalization
    marks: ["Math": 80, "Science": 90, "English": 85]  // Dictionary of subjects and marks
)
```

C/C++

```
// Step 5: Call the reporting function with the student
printStudentReport(for: s1)
```

📘 Step-by-Step Explanation (Concepts in Focus)

🔹 `enum`

We used an enum `Grade` to avoid string-based grade checks. This improves safety and readability.

🔹 `struct` and Initializers

`Student` is a struct (value type). Swift provides a default initializer based on its properties. We added:

- A **property observer** to auto-capitalize `name`
- **Computed properties** to calculate `total` and `percentage`
- A **method** `grade()` to return a `Grade` enum based on logic

🔹 Dictionary

We used a dictionary to store subject-wise marks, making it flexible for any number of subjects.

🔹 Optional Handling

In `printStudentReport(for:)`, we used `guard let` to unwrap the optional `Student?`. This ensures safety when working with optional values.

🔹 `switch` Control Flow

Switching on `Grade` enum improves clarity and future-proofing (e.g., adding `.Distinction` would be easy).

Problem 2: Banking System with Inheritance and Optionals

C/C++

```
// Step 1: Define the base class `BankAccount`

class BankAccount {



    // Property to store account balance

    var balance: Double



    // Failable initializer to reject invalid account creation

    // This will fail (return nil) if initial balance is below ₹1000

    init?(initialBalance: Double) {

        guard initialBalance >= 1000 else {

            // Exit the initializer early if balance is too low
            return nil
        }
        self.balance = initialBalance  // Set the balance if validation passes
    }

    // Method to deposit amount into account
    func deposit(amount: Double) {
        balance += amount  // Add amount to existing balance
    }

    // Method to withdraw an amount from the account
    // Returns an optional Double (nil if insufficient funds)
    func withdraw(amount: Double) -> Double? {
        if amount <= balance {
            balance -= amount  // Deduct amount from balance
            return amount      // Return withdrawn amount
        } else {
            return nil         // Return nil if withdrawal not possible
        }
    }
```

```
}
```

C/C++

```
// Step 2: Define a subclass for a savings account
class SavingsAccount: BankAccount {

    // Property specific to SavingsAccount
    var interestRate: Double = 0.05  // 5% interest rate by default
}
```

C/C++

```
// Step 3: Define a subclass for a current account
class CurrentAccount: BankAccount {

    // Property specific to CurrentAccount
    var overdraftLimit: Double = 2000  // ₹2000 overdraft limit
}
```

C/C++

```
// Step 4: Simulate fetching an account by ID (returns an optional)

// The function can return either a SavingsAccount or CurrentAccount, or nil

func getAccountById(_ id: Int) -> BankAccount? {

    if id == 1 {

        // Return a SavingsAccount with valid initial balance

        return SavingsAccount(initialBalance: 5000)

    } else if id == 2 {

        // Return a CurrentAccount with valid initial balance

        return CurrentAccount(initialBalance: 10000)

    } else {

        // Return nil for unknown IDs

        return nil

    }

}
```

C/C++

```
// Step 5: Use the account with optional chaining and control flow



// Try to get an account by ID
```

```
let account = getAccountById(1)  // This might return nil if ID is invalid

// Try to withdraw ₹2000 from the account using optional chaining
let withdrawn = account?.withdraw(amount: 2000)

// Use optional binding to check if withdrawal was successful
if let amount = withdrawn {
    print("✅ Withdrawal successful: ₹\(amount)")  // Withdrawal succeeded
} else {
    print("❌ Withdrawal failed or account not found")  // Either nil or failure
}
```

🧠 Step-by-Step Concept Explanation

🔹 Classes and Inheritance

We created a base class `BankAccount` and two subclasses:

- `SavingsAccount` adds an `interestRate`
- `CurrentAccount` adds an `overdraftLimit`

Each subclass **inherits** balance and methods (`deposit`, `withdraw`) from the base class.

🔹 Failable Initializer

We used `init?()` in `BankAccount` to **reject accounts with low initial balance**. If the balance is less than ₹1000, the object won't be created (returns `nil`).

🔹 Optional Return from Methods

The `withdraw(amount:)` method returns `Double?`. If the withdrawal can't happen (due to low funds), it returns `nil`. This allows the caller to handle failure safely.

🔹 Optional Chaining

We safely tried to withdraw money using:

None

```
account?.withdraw(amount: 2000)
```

This ensures that the method is only called **if the account is not nil**.

🔹 Optional Binding

Using:

None

```
if let amount = withdrawn {

    // success

}
```

We safely unwrap the optional result of the withdrawal and confirm if it succeeded.

✅ This program simulates a **real-world banking system** and demonstrates safe, object-oriented programming in Swift using **classes, inheritance, optionals, and control flow**.

✅ Problem 3: Library Book Tracker

📘 Full Swift Code with Detailed Line-by-Line Explanation

C/C++

```
// Step 1: Define an enum to represent the status of a book
enum BookStatus {
    case available     // Book is ready to be issued
    case issued        // Book is currently issued to a user
    case reserved      // Book is reserved (but not yet issued)
}
```

C/C++

```
// Step 2: Define a `Book` struct that represents a single book
struct Book {
    var title: String      // Title of the book
    var author: String     // Author name

    // Enum property to track the status of the book
    var status: BookStatus {
        didSet {
            // Property observer: gets triggered whenever `status` changes
            print("📚 Status of '\(title)' changed to: \(status)")
        }
    }
}
```

C/C++

```
// Step 3: Define a `Library` class to manage the book collection and operations
class Library {

    // Array to store all books in the library
    var books: [Book] = []

    // Dictionary to keep track of issued books
    // Key: username, Value: book title
    var borrowedBooks: [String: String] = [:]

    // Method to add a new book to the library
    func addBook(_ book: Book) {
        books.append(book)  // Append book to books array
    }
```

```swift
    // Method to issue a book to a user
    func issueBook(title: String, to user: String) {
        // Loop over books using indices, because `status` is mutable
        for i in 0..<books.count {
            // Match book title and check if it is available
            if books[i].title == title && books[i].status == .available {
                books[i].status = .issued              // Change book status
                borrowedBooks[user] = books[i].title   // Record user → book mapping
                print("✅ Book '\(title)' issued to \(user)")
                return
            }
        }
        // If no match found or book not available
        print("❌ Book '\(title)' is not available for issue")
    }

    // Method to return a book from a user
    func returnBook(from user: String) {
        // Use optional binding to get the borrowed title (if exists)
        if let title = borrowedBooks[user] {

            // Loop through books to find the one being returned
            for i in 0..<books.count {
                if books[i].title == title {
                    books[i].status = .available     // Mark book as available again
                    borrowedBooks[user] = nil        // Remove mapping from dictionary
                    print("✅ Book '\(title)' returned by \(user)")
                    return
                }
            }
        } else {
            // No book was found for this user
            print("⚠️ No book found to return for user: \(user)")
        }
    }
}
```

C/C++

```
// Step 4: Create some book objects using memberwise initializer

let b1 = Book(title: "Swift Basics", author: "Apple", status:
.available)

let b2 = Book(title: "iOS Guide", author: "Apple", status:
.available)
```

C/C++

```
// Step 5: Create an instance of Library and add books to it

let lib = Library()     // Create a new Library object

lib.addBook(b1)         // Add first book

lib.addBook(b2)         // Add second book
```

C/C++

```
// Step 6: Simulate issuing and returning books


// Try to issue a book to a user

lib.issueBook(title: "Swift Basics", to: "Aman")


// Now return the book

lib.returnBook(from: "Aman")
```

🧠 Step-by-Step Concept Explanation

🔹 `enum` — BookStatus

We use `enum BookStatus` to model the possible states of a book:
`available`, `issued`, and `reserved`.
Enums make our code more **readable and type-safe**.

🔹 `struct` — Book

A `Book` is modeled as a struct with a title, author, and status.
We use a **property observer (`didSet`)** to print a message whenever the book's status changes — this simulates a "log system."

🔹 `class` — Library

The `Library` is a reference type (class) because its state changes as users borrow/return books. It stores:

- An array of books (`[Book]`)
- A dictionary of borrowed books (`[String: String]`)

🔹 Loops with `0..<books.count`

We loop using indices so we can **mutate** book objects (e.g., change status). You can't modify values directly when using `for book in books`.

🔹 Optional Handling

We safely check whether a user has a book issued using:

None

```
if let title = borrowedBooks[user] {

    // ...

}
```

If there is no entry, we notify the user.

✅ This mini-library system demonstrates how to manage state with structs, enums, classes, dictionaries, arrays, and control flow. It’s very close to real-world app modeling.

✅ Problem 4: Custom Pizza Order System

C/C++

```
// Step 1: Define an enum `Size` to represent pizza sizes

// Enums with raw values let us convert from Strings to enum safely

enum Size: String {

    case small

    case medium

    case large

}
```

C/C++

```
// Step 2: Define a struct `Pizza` to model a pizza order

struct Pizza {
    var toppings: [String]     // Array of topping names (e.g., ["cheese", "olives"])
    var size: Size             // Size of the pizza, using our enum

    // Computed property to calculate price based on size and number of toppings
    var price: Double {
        let base = 100.0  // Base price for all pizzas
        let toppingCost = Double(toppings.count) * 20.0  // ₹20 per topping

        // Add extra charge depending on pizza size
        switch size {
        case .small:
            return base + toppingCost
        case .medium:
            return base + toppingCost + 50
        case .large:
            return base + toppingCost + 100
        }
    }
    // Failable initializer to ensure valid size and at least one topping
    init?(toppings: [String], size: String) {
        // Guard to ensure there is at least one topping
        guard !toppings.isEmpty,
              // Use enum's rawValue initializer to convert from String to Size enum
              let parsedSize = Size(rawValue: size.lowercased()) else {
            return nil  // Fail initialization if invalid input
        }

        // Capitalize the toppings (e.g., "cheese" → "Cheese")
        self.toppings = toppings.map { $0.capitalized }
        self.size = parsedSize
    }
}
```

C/C++

```
// Step 3: Define a function to place an order
// The function accepts an optional Pizza and uses optional binding to validate it
```

```
func placeOrder(for pizza: Pizza?) {

    // Use `guard let` to unwrap the optional pizza safely
    guard let pizza = pizza else {
        print("❌ Invalid pizza order. Please check size or toppings.")
        return
    }

    // Join all toppings into a single string, separated by commas
    let toppingList = pizza.toppings.joined(separator: ", ")

    // Print the complete order summary
    print("🍕 Pizza Size: \(pizza.size.rawValue.capitalized)") // Capitalize for display
    print("🧀 Toppings: \(toppingList)")
    print("💵 Total Price: ₹\(pizza.price)")
}
```

C/C++

```
// Step 4: Try creating a pizza with toppings and size
// This will use our custom (failable) initializer

let myPizza = Pizza(toppings: ["cheese", "olives"], size: "Medium")
```

C/C++

```
// Step 5: Pass the optional pizza into the function

// Even if myPizza was nil, the function would handle it gracefully

placeOrder(for: myPizza)
```

🧠 Step-by-Step Concept Explanation

🔹 `enum Size: String`

We use a **String-backed enum** to represent pizza sizes: `.small`, `.medium`, `.large`.
This allows us to convert from raw strings like `"Medium"` to enum using:

None

```
Size(rawValue: size.lowercased())
```

🔹 `struct Pizza`

The Pizza model includes:

- An array of toppings
- A computed `price` property
- A **failable initializer (`init?`)** that validates input

The initializer uses `guard` to ensure:

1. Toppings are not empty
2. Size string maps to a valid `Size` enum

We also use:

C/C++

```
toppings.map { $0.capitalized }
```

To make toppings display neatly.

🔹 `guard let` and `optional chaining`

The `placeOrder(for:)` function is designed to work even if the pizza is `nil`.
Using `guard let` lets us unwrap and safely use the pizza, or exit early with an error message.

🔹 String joining and display

To make toppings display nicely, we used:

C/C++

```
.joined(separator: ", ")
```

And capitalized enum values and toppings for a professional-looking output.

✅ This mini-project teaches how to model **real-world customizable items** (like pizza), perform input validation using optionals, and elegantly display data using computed properties and string formatting.

✅ Problem 5: Ride Booking App Simulator

C/C++

```
// Step 1: Define an enum to represent different ride statuses

enum RideStatus {
    case requested     // Ride is just requested, not yet
confirmed

    case confirmed     // Ride is accepted/confirmed

    case cancelled     // Ride is cancelled

}
```

C/C++

```
// Step 2: Define a base class `Ride` to represent a generic ride

class Ride {



    var user: String           // Name or ID of the user booking
the ride

    var distance: Double       // Distance of the ride in
kilometers

    var status: RideStatus     // Current status of the ride



    // Designated initializer

    init(user: String, distance: Double) {

        self.user = user

        self.distance = distance

        self.status = .requested  // Default status when ride is
created
```

```
    }

    // Method to calculate fare (can be overridden in subclasses)
    func calculateFare() -> Double {
        return distance * 10.0    // Generic fare rate: ₹10/km
    }
}
```

C/C++

```
// Step 3: Subclass `Ride` into `BikeRide` and override the fare method
class BikeRide: Ride {
    // Bikes are cheaper, so override fare calculation
    override func calculateFare() -> Double {
        return distance * 5.0     // ₹5/km for bike
    }
}
```

C/C++

```
// Step 4: Another subclass: CarRide
class CarRide: Ride {
    // Cars are more expensive
    override func calculateFare() -> Double {
        return distance * 15.0    // ₹15/km for car
    }
}
```

C/C++

```
// Step 5: Dictionary to track each user's ride
// Key: Username, Value: Ride instance
var userRides: [String: Ride] = [:]
```

C/C++

```
// Step 6: Function to confirm a ride
// Takes a username, uses guard to unwrap and validate ride
func confirmRide(for user: String) {

    // Use guard to safely unwrap the ride for given user
    guard let ride = userRides[user] else {
        print("❌ No ride found for user: \(user)")
        return
    }

    // Check if distance is acceptable
    if ride.distance < 1 {
        print("⚠️ Ride distance too short. Must be at least 1 km.")
        return
    }

    // If everything is valid, update status and show fare
    ride.status = .confirmed

    // Use overridden calculateFare method depending on ride type
    let fare = ride.calculateFare()

    print("✅ Ride confirmed for \(user)")
    print("📍 Distance: \(ride.distance) km")
    print("💰 Fare: ₹\(fare)")
}
```

```
C/C++
// Step 7: Example usage — create a CarRide for a user and
confirm it


// Create and store the ride object into the dictionary

userRides["Aman"] = CarRide(user: "Aman", distance: 6.0)


// Now confirm the ride for Aman

confirmRide(for: "Aman")
```

🧠 Step-by-Step Conceptual Explanation

🔹 `enum RideStatus`

We define the possible ride statuses using an enum:

```
C/C++
enum RideStatus { case requested, confirmed, cancelled }
```

This makes the status more readable and less error-prone than using strings like `"requested"`.

🔹 `class Ride` & Subclasses

`Ride` is a base class with shared properties:

- `user`: who booked it
- `distance`: how long the ride is
- `status`: default `.requested`

It has a method:

None

```
func calculateFare() -> Double
```

which gets **overridden** in subclasses like `BikeRide` and `CarRide`.

Each subclass provides its own rate logic:

- Bike → ₹5/km
- Car → ₹15/km
- Ride → ₹10/km (default)

🔹 Dictionary to Track Rides

We use a dictionary:

C/C++

```
var userRides: [String: Ride] = [:]
```

To associate each user with a specific ride instance.

🔹 Optional Chaining & `guard`

In the function `confirmRide(for:)`, we use:

C/C++

```
guard let ride = userRides[user] else {

    print("No ride found")

    return

}
```

This **safely unwraps** the ride object, preventing runtime crashes.

🔹 Polymorphism

Even though the dictionary stores values of type `Ride`, calling:

C/C++

```
ride.calculateFare()
```

will execute the **correct overridden method** (e.g., from `CarRide`) because of **polymorphism**.

✅ This problem simulates a real-world app like Uber or Ola. It helps you understand **object-oriented programming**, **control flow**, and **optional safety** — essential skills for iOS development.

Problem 6: Payment Processing with Type Casting

C/C++

```
// Step 1: Define a base class for all payment methods
class PaymentMethod {

    // A common property shared by all payment types to store the user name
    var user: String

    // Regular initializer - sets the user property
    init(user: String) {
        self.user = user
    }
}
```

C/C++

```
// Step 2: Subclass for Credit Card payments
class CreditCard: PaymentMethod {

    // New property specific to CreditCard
    var cardNumber: String

    // Failable initializer (can return nil)
    // This ensures the card number must be exactly 16 digits
    init?(user: String, cardNumber: String) {

        // Use guard to validate the card number length
        guard cardNumber.count == 16 else {
            return nil  // Initialization fails if length is not
16
        }

        self.cardNumber = cardNumber   // Set the property if
valid
        super.init(user: user)         // Call the superclass
initializer
    }

    // Method to simulate credit card processing
    func processPayment() {
        print("💳 Processing credit card for \(user)")
    }
}
```

C/C++

```
// Step 3: Subclass for UPI payments
class UPI: PaymentMethod {

    // UPI-specific property
    var upiID: String

    // Regular initializer with upiID
    init(user: String, upiID: String) {
        self.upiID = upiID
        super.init(user: user)
    }

    // Method to simulate UPI transaction
    func payViaUPI() {
        print("📱 Paying via UPI for \(user)")
    }
}
```

C/C++

```
// Step 4: Subclass for Cash payments
class Cash: PaymentMethod {

    // This subclass has no extra properties, just a behavior
    func collectCash() {
        print("💵 Collecting cash from \(user)")
    }
}
```

C/C++

```
// Step 5: Create a dictionary of users and their payment methods
// Since the types are different, we use `Any` to allow heterogenous values
let payments: [String: Any] = [
    // Valid CreditCard instance (force-cast to `Any` is optional here)
    "Aman": CreditCard(user: "Aman", cardNumber: "1234567890123456") as Any,


    // UPI payment method
    "John": UPI(user: "John", upiID: "john@upi"),


    // Cash payment
    "Alice": Cash(user: "Alice"),


    // Invalid object, just a string (not a PaymentMethod)
    "Unknown": "This is not a payment object"
]
```

C/C++

```
// Step 6: Define a function to process any payment based on user name
func processPayment(for name: String) {


    // Use `guard` to safely unwrap the optional value from the dictionary
    // If no entry is found, exit early
    guard let method = payments[name] else {
        print("❌ No payment method found for \(name)")
        return
    }

    // Use `type(of:)` to print the actual type of the value at runtime
    print("🔎 \(name)'s payment type: \(type(of: method))")

    // Step 6.1: Try to cast the method to CreditCard and call its method
    if let cc = method as? CreditCard {
        cc.processPayment()   // Calls processPayment() on CreditCard
    }

    // Step 6.2: Try to cast to UPI and call UPI-specific method
    else if let upi = method as? UPI {
        upi.payViaUPI()
    }
    // Step 6.3: Try to cast to Cash and call cash-specific method
    else if let cash = method as? Cash {
        cash.collectCash()
    }

    // Step 6.4: Catch-all for unknown types (e.g., String or invalid object)
    else {
        print("⚠️ Unknown or invalid payment method for \(name)")
    }
}
```

C/C++

```
// Step 7: Call the function with different user names

// Valid: should print credit card processing
processPayment(for: "Aman")

// Valid: UPI
processPayment(for: "John")
```

```
// Valid: Cash
processPayment(for: "Alice")

// Invalid: not a PaymentMethod
processPayment(for: "Unknown")
```

🧠 Concept-by-Concept Breakdown

🔹 1. `Any` Type

- `Any` can hold **any Swift type** — including classes, structs, or even primitive types like `String`.
- We used it in `[String: Any]` to allow storing various `PaymentMethod` subclasses in the same dictionary.

🔹 2. `type(of:)`

- `type(of:)` lets us inspect what the **actual runtime type** of an object is.
- Helps in debugging and conditionally executing logic based on type.

🔹 3. `as?` Optional Casting

- `as?` tries to **safely cast** an object to a specific type.
- Returns an optional: if it fails, you get `nil`, allowing safe branching using `if let`.

🔹 4. `guard let`

- Used to **safely unwrap optionals** early in the function.
- Great for input validation or early-exit patterns.

🔹 5. Class Inheritance

- `CreditCard`, `UPI`, and `Cash` all **inherit** from `PaymentMethod`.

- This allows polymorphism: even though the dictionary holds `Any`, we treat the values as `PaymentMethod` at runtime.

🔹 6. Failable Initializer

- `CreditCard.init?()` checks for a valid 16-digit number.
- If validation fails, it returns `nil` instead of a bad object — a safe initialization pattern.

✅ This problem simulates a real-world fintech app's backend, using safe Swift features like **type casting**, **optional safety**, and **OOP design** to manage dynamic user data.