Simplified Banking System - OOP Design Overview

Initial Prompt (High-Level Design & Core Objects):

"We're building a simplified banking system. How would you start modeling the core entities? What would be the main objects you'd identify?"
Expected Answer: "I'd start with an Account object. It would need properties like accountNumber, customerName, and balance."

Follow-up: "What operations would an Account typically support?"
Expected Answer: "Depositing money, withdrawing money, checking the balance, and probably displaying account information."

This leads to the Account class with its basic properties and deposit, withdraw, getBalance, and getAccountInfo methods.

Introducing Specialization (Inheritance):

"That's a good start. Now, in a real bank, we don't just have one generic account type. We have different kinds of accounts, like Savings Accounts and Checking Accounts. How would you model these, building upon your existing Account structure?"
Expected Answer: "I'd use inheritance. SavingsAccount and CheckingAccount could extend the base Account class."

Follow-up: "What makes a Savings Account different from a regular Account? What unique behavior or properties would it have?"
Expected Answer: "A Savings Account typically earns interest. So, it would need an interestRate

property and an applyInterest method."

Follow-up: "And a Checking Account? What's its unique feature?"
Expected Answer: "Checking Accounts often have an overdraft facility. So, it would need an overdraftLimit."

This prompts the creation of SavingsAccount and CheckingAccount extending Account, adding their specific properties and methods (applyInterest).

Probing Polymorphism (Method Overriding):

"Considering the withdraw operation, would withdraw work exactly the same way for a CheckingAccount as it does for a SavingsAccount or a basic Account?"
Expected Answer: "Not necessarily. A CheckingAccount might allow withdrawals even if the balance goes negative, as long as it's within the overdraft limit. So, I'd need to override the withdraw method in the CheckingAccount to include that specific logic."

Follow-up: "Excellent. How would you ensure that when you call withdraw on an account, the correct logic for that specific account type is executed?"
Expected Answer: "By overriding the method in the child class, JavaScript's prototypal inheritance (or class syntax's underlying mechanism) automatically handles this. The this context ensures the right withdraw is called based on the object's type."

This directly leads to the withdraw method override in CheckingAccount.

Managing Collections & Relationships (Aggregation/Composition):

"Now that we have different account types, how would you manage all the accounts within the entire bank? We need a way to store them and perform operations like finding an account or listing all accounts."
Expected Answer: "I'd create a Bank class. The Bank class would contain an array or a collection of Account objects."

Follow-up: "Is this a 'has-a' relationship? If the Bank ceases to exist, do the accounts necessarily cease to exist? What's that relationship called in OOP?"
Expected Answer: "Yes, it's a 'has-a' relationship. The accounts can exist independently of the bank instance (e.g., they could be transferred to another bank). This is typically referred to as aggregation."

Follow-up: "What methods would this Bank class need?"
Expected Answer: "Methods to addAccount, findAccount, and displayAllAccounts."

This explains the Bank class with its accounts array and methods for managing them.

Edge Cases, Validation, and Robustness:

"What about input validation? For instance, what if someone tries to deposit a negative amount?"
Expected Answer: "The deposit and withdraw methods should include checks to ensure the amount is positive. For withdrawal, it also needs to check for sufficient funds."

"How would you ensure that only valid Account objects (or its subclasses) can be added to the bank?"
Expected Answer: "I'd use instanceof inside the addAccount method to check if the passed object is an instance of Account or any of its derived classes."

These questions prompt the conditional logic within deposit, withdraw, and addAccount for robust error handling.