**Day 4:** Null Safety and Exception Handling / Advanced Features (Extensions and Coroutines)

**Task 1:** Implement null safety features to handle the absence of transaction data.

package day4  
  
data class Transaction(val id: String, val amount: Double?, val description: String?)  
  
fun handleTransaction(transaction: Transaction?) {  
 val transactionId = transaction?.id ?: "No ID"  
 val transactionAmount = transaction?.amount ?: 0.0  
 val transactionDescription = transaction?.description ?: "No description"  
  
 *println*("Transaction Details:")  
 *println*("ID: $transactionId")  
 *println*("Amount: $transactionAmount")  
 *println*("Description: $transactionDescription")  
  
 transaction?.*let* **{** *println*("Processing transaction with ID: ${**it**.id}")  
 **}** ?: *run* **{** *println*("Transaction data is not available.")  
 **}**}  
  
fun main() {  
 val validTransaction = Transaction("12", 300.50, "Payment for services")  
 val nullTransaction: Transaction? = null  
  
 *println*("Valid Transaction:")  
 *handleTransaction*(validTransaction)  
  
 *println*("\nNull Transaction:")  
 *handleTransaction*(nullTransaction)  
}

//Output

Valid Transaction:

Transaction Details:

ID: 12

Amount: 300.5

Description: Payment for services

Processing transaction with ID: 12

Null Transaction:

Transaction Details:

ID: No ID

Amount: 0.0

Description: No description

Transaction data is not available.

**Task 2:** Write custom exception classes to handle errors related to transaction processing.

package day4  
// Base exception for transaction errors  
open class TransactionException(message: String) : Exception(message)  
  
// Custom exception for insufficient balance  
class InsufficientBalanceException(  
 message: String = "Insufficient balance for the transaction"  
) : TransactionException(message)  
  
// Custom exception for invalid transaction ID  
class InvalidTransactionIdException(  
 message: String = "Invalid transaction ID provided"  
) : TransactionException(message)  
  
// Custom exception for transaction amount errors  
class InvalidTransactionAmountException(  
 message: String = "Transaction amount must be greater than zero"  
) : TransactionException(message)

package day4  
  
data class Transaction2(  
 val id: String,  
 val amount: Double  
)  
  
fun processTransaction(transaction: Transaction2?, balance: Double) {  
 // Check for null transaction  
 if (transaction == null) {  
 throw InvalidTransactionIdException("Transaction object is null")  
 }  
  
 // Validate transaction ID  
 if (transaction.id.*isBlank*()) {  
 throw InvalidTransactionIdException("Transaction ID cannot be blank")  
 }  
  
 // Validate transaction amount  
 if (transaction.amount <= 0) {  
 throw InvalidTransactionAmountException("Amount must be greater than zero")  
 }  
  
 // Check for sufficient balance  
 if (balance < transaction.amount) {  
 throw InsufficientBalanceException("Available balance: $balance, required: ${transaction.amount}")  
 }  
  
 // Process the transaction  
 *println*("Transaction processed successfully: $transaction")  
}  
  
fun main() {  
 try {  
 val balance = 50.0  
 val transaction = Transaction2("12345", 100.0)  
  
 *processTransaction*(transaction, balance)  
 } catch (e: TransactionException) {  
 *println*("Error: ${e.message}")  
 }  
  
 try {  
 val invalidTransaction = Transaction2("", -10.0)  
 *processTransaction*(invalidTransaction, 200.0)  
 } catch (e: TransactionException) {  
 *println*("Error: ${e.message}")  
 }  
}

//Output

Error: Available balance: 50.0, required: 100.0

Error: Transaction ID cannot be blank

**Task 3:** Create extension functions for the List<Transaction> class to calculate total expenses and incomes.

package day4  
  
data class Transaction3(val id: String, val amount: Double)  
  
fun List<Transaction3>.totalIncomes(): Double {  
 return this.*filter* **{ it**.amount > 0 **}** .*sumOf* **{ it**.amount **}**}  
  
fun List<Transaction3>.totalExpenses(): Double {  
 return this.*filter* **{ it**.amount < 0 **}** .*sumOf* **{ it**.amount **}** .*let* **{ it** \* -1 **}** // Convert the total to a positive number for readability  
}  
  
fun main() {  
 val transactions = *listOf*(  
 Transaction3("1", 100.0), // Income  
 Transaction3("2", -50.0), // Expense  
 Transaction3("3", 200.0), // Income  
 Transaction3("4", -30.0), // Expense  
 Transaction3("5", -20.0) // Expense  
 )  
  
 val totalIncomes = transactions.*totalIncomes*()  
 val totalExpenses = transactions.*totalExpenses*()  
  
 *println*("Total Incomes: $totalIncomes")  
 *println*("Total Expenses: $totalExpenses")  
}

//Output

Total Incomes: 300.0

Total Expenses: 100.0

**Task 4:** Use coroutines to handle simultaneous processing of importing and exporting transaction data without blocking the main thread.  
  
package day4  
import kotlinx.coroutines.\*  
data class Transaction4(val id: String, val type: String, val amount: Double)  
  
suspend fun exportTransactions(processedTransactions: List<Transaction4>) {  
 delay(2000) // Simulate a delay for data export  
 *println*("Exported processed transactions: ${processedTransactions.*map* **{ it**.id **}**}")  
}  
  
// Function to process transactions  
suspend fun processTransactions(  
 transactions: List<Transaction4>,  
 filterType: String,  
 threshold: Double  
): List<Transaction4> {  
 delay(1000) // Simulate processing delay  
 return transactions.*filter* **{ it**.type != filterType && **it**.amount > threshold **}**}  
  
fun main() = *runBlocking* **{** *println*("Starting transaction processing...")  
  
 // Create the transaction list in the main function  
 val transactions = *listOf*(  
 Transaction4("1", "debit", 150.0),  
 Transaction4("2", "credit", 200.0),  
 Transaction4("3", "debit", 100.0),  
 Transaction4("4", "credit", 300.0),  
 Transaction4("5", "credit", 120.0)  
 )  
  
 // Launch coroutine to handle processing and exporting  
 val job = *launch* **{** val processedTransactions = *async* **{** processTransactions(transactions, filterType = "debit", threshold = 100.0) **}** val exportJob = *async* **{** exportTransactions(processedTransactions.await()) **}** exportJob.await()  
 **}** job.join() // Wait for all tasks to complete  
 *println*("Transaction processing completed.")  
**}**

**//Output**

Starting transaction processing...

Exported processed transactions: [2, 4, 5]

Transaction processing completed.