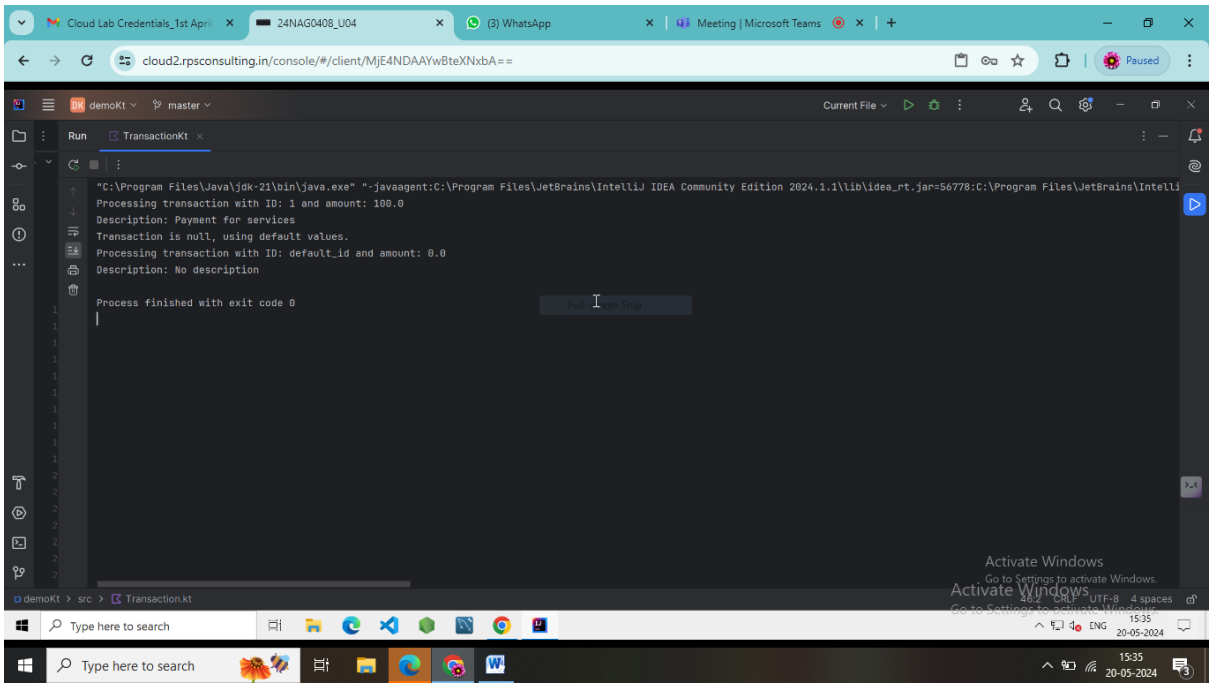
Day-4, Assignment-1

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

```
// Define a data class for Transaction
data class Transaction(val id: String, val amount: Double, val description: String?)
// Function to process a transaction which might be null
fun processTransaction(transaction: Transaction?) {
    // Safe call operator to check for null
    transaction?.let {
        println("Processing transaction with ID: ${it.id} and amount: ${it.amount}")
        // Safe call on a nullable property
        println("Description: ${it.description ?: "No description provided"}")
    } ?: run {
        // Handle the case when transaction is null
        println("No transaction data available.")
    }
}
fun getTransactionDescription(transaction: Transaction?): String {
    return transaction?.description ?: "Description not available"
}
fun printTransactionDetails(transaction: Transaction? = null) {
    val id = transaction?.id ?: "Unknown ID"
    val amount = transaction?.amount ?: 0.0
    val description = transaction?.description ?: "No description provided"
    println("Transaction Details: ID=$id, Amount=$amount, Description=$description")
}
fun printTransactionId(transaction: Transaction?) {
```

```
    // Will throw an exception if transaction is null
    println("Transaction ID: ${transaction!!.id}")
}
fun handleTransaction(transaction: Transaction?) {
    if (transaction == null) {
        println("Transaction is null, using default values.")
        val defaultTransaction = Transaction("default_id", 0.0, "No description")
        processTransaction(defaultTransaction)
    } else {
        processTransaction(transaction)
    }
}
fun main() {
    val transaction1: Transaction? = Transaction("1", 100.0, "Payment for services")
    val transaction2: Transaction? = null
    handleTransaction(transaction1)
    handleTransaction(transaction2)
}
```

Output:

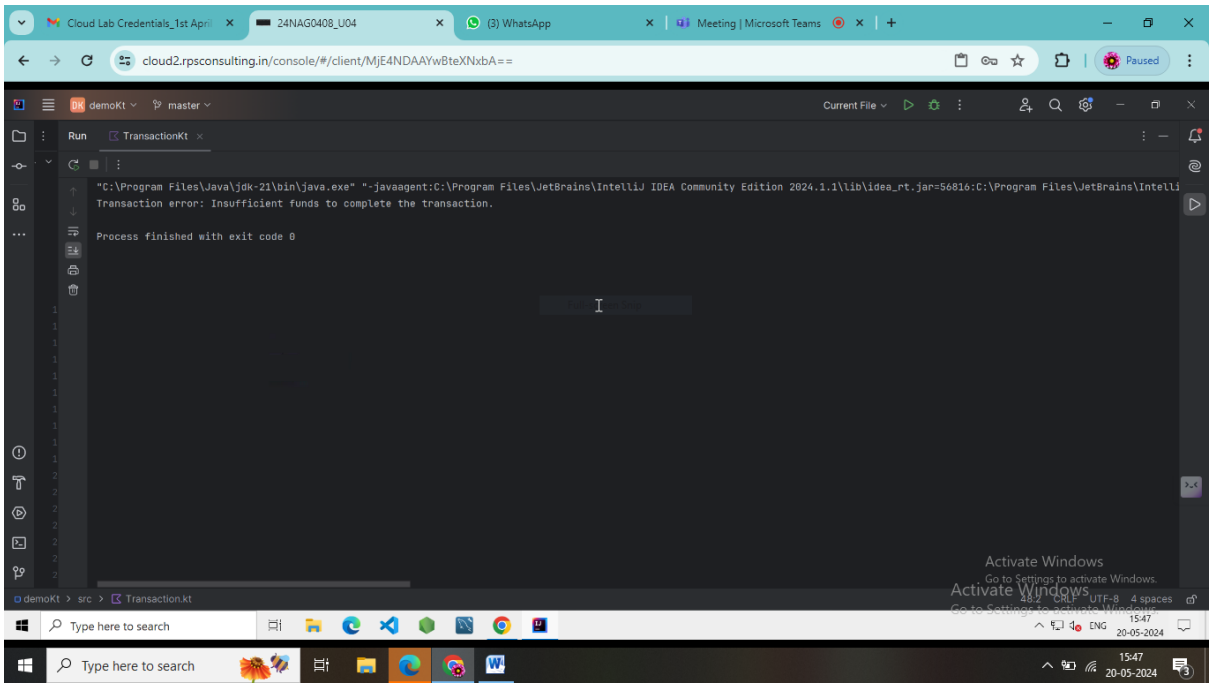


Task 2: Write custom exception classes to handle errors related to transaction processing.

```
// Base class for all transaction-related exceptions
open class TransactionException(message: String) : Exception(message)

// Exception class for invalid transaction amounts
class InvalidAmountException(message: String) : TransactionException(message)

// Exception class for insufficient funds
class InsufficientFundsException(message: String) : TransactionException(message)

// Exception class for unauthorized transactions
class UnauthorizedTransactionException(message: String) : TransactionException(message)

// Exception class for transaction timeout
class TransactionTimeoutException(message: String) : TransactionException(message)

// Exception class for general transaction failures
class TransactionFailureException(message: String) : TransactionException(message)

fun processTransaction(amount: Double, accountBalance: Double, authorized: Boolean) {
    if (amount <= 0) {
```

```
        throw InvalidAmountException("Transaction amount must be greater than zero.")
    }
    if (!authorized) {
        throw UnauthorizedTransactionException("User is not authorized to perform this transaction.")
    }
    if (accountBalance < amount) {
        throw InsufficientFundsException("Insufficient funds to complete the transaction.")
    }
    // Simulate transaction processing...
    val transactionSuccessful = simulateTransactionProcessing(amount)
    if (!transactionSuccessful) {
        throw TransactionFailureException("Transaction failed due to an unknown error.")
    }
}
fun simulateTransactionProcessing(amount: Double): Boolean {
    // Simulate some processing logic
    return true // Or false if the transaction fails
}
fun main() {
    try {
        processTransaction(100.0, 50.0, true)
    } catch (e: TransactionException) {
        println("Transaction error: ${e.message}")
    }
}
```

Output:



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

```
data class Transaction(
    val amount: Double,
    val type: TransactionType
)
enum class TransactionType {
    INCOME, EXPENSE
}
// Extension function to calculate total expenses
fun List<Transaction>.totalExpenses(): Double {
    return this.filter { it.type == TransactionType.EXPENSE }
            .sumOf { it.amount }
}
// Extension function to calculate total incomes
fun List<Transaction>.totalIncomes(): Double {
```

```
    return this.filter { it.type == TransactionType.INCOME }
            .sumOf { it.amount }
}

fun main() {
    val transactions = listOf(
        Transaction(100.0, TransactionType.INCOME),
        Transaction(50.0, TransactionType.EXPENSE),
        Transaction(200.0, TransactionType.INCOME),
        Transaction(30.0, TransactionType.EXPENSE)
    )
    val totalExpenses = transactions.totalExpenses()
    val totalIncomes = transactions.totalIncomes()
    println("Total Expenses: $$totalExpenses")
    println("Total Incomes: $$totalIncomes")
}
```

Output: