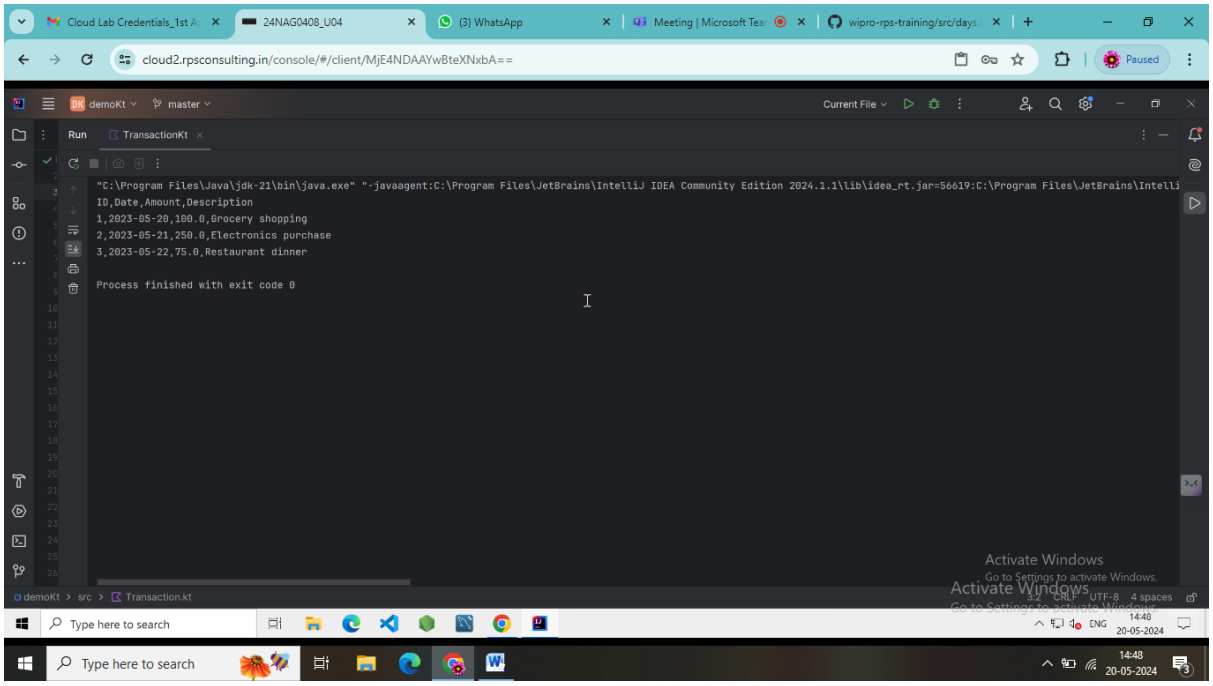
Day-3, Assignment-1

Task 9: Define an Exportable interface with a method to export transaction data to CSV.

```
interface Exportable {
    fun exportToCSV(transactions: List<Transaction>): String
}
data class Transaction(
    val id: Int,
    val date: String,
    val amount: Double,
    val description: String
)
class CSVExporter : Exportable {
    override fun exportToCSV(transactions: List<Transaction>): String {
        val header = "ID,Date,Amount,Description"
        val csvData = transactions.joinToString(separator = "\n") { transaction ->
            "${transaction.id},${transaction.date},${transaction.amount},${transaction.description}"
        }
        return "$header\n$csvData"
    }
}
fun main() {
    val transactions = listOf(
        Transaction(1, "2023-05-20", 100.0, "Grocery shopping"),
        Transaction(2, "2023-05-21", 250.0, "Electronics purchase"),
        Transaction(3, "2023-05-22", 75.0, "Restaurant dinner")
    )
```

```
    val exporter: Exportable = CSVExporter()
    val csv = exporter.exportToCSV(transactions)
    println(csv)
}
```

Output:



Task 10: Apply encapsulation to Transaction properties using getters and setters ensuring sensitive data is protected.

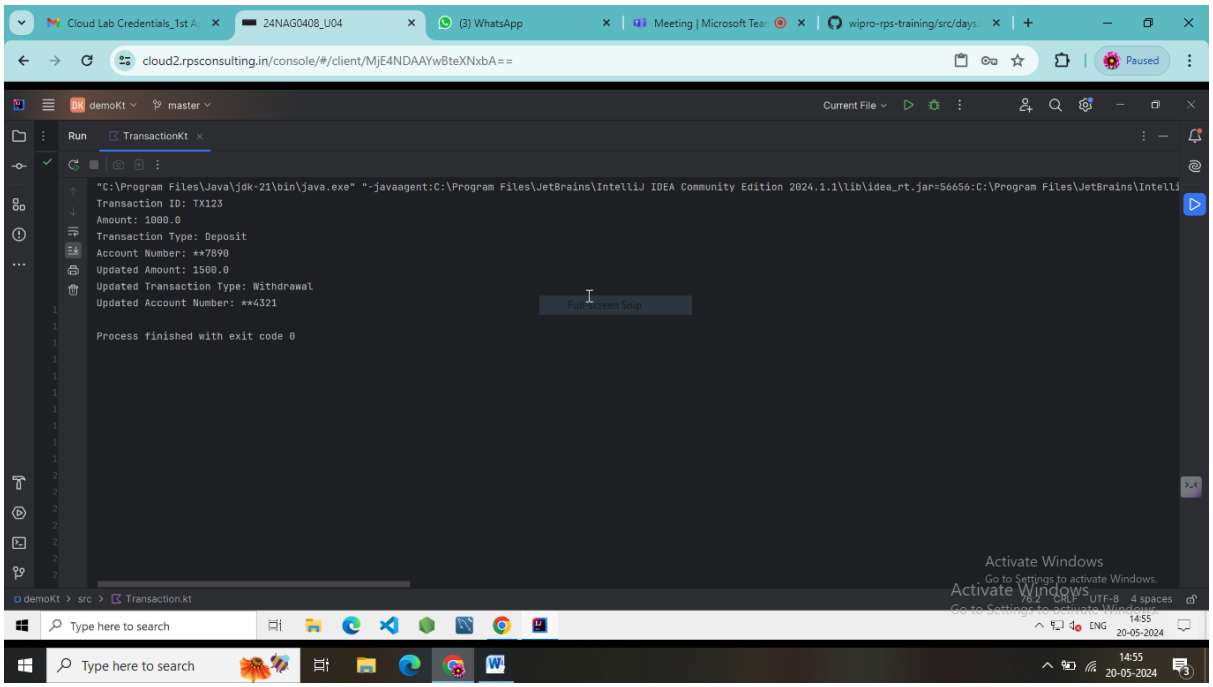
```
class Transaction {
    // Private backing fields
    private var _transactionId: String = ""
    private var _amount: Double = 0.0
    private var _transactionType: String = ""
    private var _accountNumber: String = ""
    // Public getter for transactionId
    val transactionId: String
        get() = _transactionId
```

```
// Public getter and setter for amount
var amount: Double
    get() = _amount
    set(value) {
        if (value >= 0) {
            _amount = value
        } else {
            throw IllegalArgumentException("Amount must be non-negative")
        }
    }
// Public getter and setter for transactionType
var transactionType: String
    get() = _transactionType
    set(value) {
        if (value in listOf("Deposit", "Withdrawal", "Transfer")) {
            _transactionType = value
        } else {
            throw IllegalArgumentException("Invalid transaction type")
        }
    }
// Public getter and setter for accountNumber with restricted access
var accountNumber: String
    get() = "**" + _accountNumber.takeLast(4) // Masked for privacy
    private set(value) { // Setter is private to protect data
        _accountNumber = value
    }
// Constructor
```

```
    constructor(transactionId: String, amount: Double, transactionType: String, accountNumber:
String) {
        this._transactionId = transactionId
        this.amount = amount
        this.transactionType = transactionType
        this._accountNumber = accountNumber // Direct access within the class
    }
    // Method to update account number, if needed
    fun updateAccountNumber(newAccountNumber: String) {
        if (newAccountNumber.length == 10) {
            _accountNumber = newAccountNumber
        } else {
            throw IllegalArgumentException("Account number must be 10 digits")
        }
    }
}
fun main() {
    val transaction = Transaction("TX123", 1000.0, "Deposit", "1234567890")
    println("Transaction ID: ${transaction.transactionId}")
    println("Amount: ${transaction.amount}")
    println("Transaction Type: ${transaction.transactionType}")
    println("Account Number: ${transaction.accountNumber}")
    // Update amount and transaction type
    transaction.amount = 1500.0
    transaction.transactionType = "Withdrawal"
    println("Updated Amount: ${transaction.amount}")
    println("Updated Transaction Type: ${transaction.transactionType}")
    // Attempt to update account number using method
```

```
    transaction.updateAccountNumber("0987654321")
    println("Updated Account Number: ${transaction.accountNumber}")
}
```

Output:



Task 11: Create generic functions to handle different types of collections (List, Set, Map) of transactions.

```
data class Transaction(val id: Int, val amount: Double, val description: String)

fun <C : Collection<Transaction>> filterTransactionsByAmount(collection: C, minAmount: Double): C
{
    return collection.filter { it.amount >= minAmount } as C
}

fun <C : Collection<Transaction>> sumTransactionAmounts(collection: C): Double {
    return collection.sumOf { it.amount }
}

fun <C : Collection<Transaction>> findTransactionById(collection: C, id: Int): Transaction? {
    return collection.find { it.id == id }
}
```

```
fun <K> filterTransactionsByAmount(map: Map<K, Transaction>, minAmount: Double): Map<K, Transaction> {
    return map.filterValues { it.amount >= minAmount }
}
fun <K> sumTransactionAmounts(map: Map<K, Transaction>): Double {
    return map.values.sumOf { it.amount }
}
fun <K> findTransactionById(map: Map<K, Transaction>, id: K): Transaction? {
    return map[id]
}
fun main() {
    val transactionsList = listOf(
        Transaction(1, 100.0, "Groceries"),
        Transaction(2, 250.0, "Electronics"),
        Transaction(3, 75.0, "Books")
    )
    val filteredList = filterTransactionsByAmount(transactionsList, 100.0)
    println("Filtered List: $filteredList")
    val totalAmountList = sumTransactionAmounts(transactionsList)
    println("Total Amount in List: $totalAmountList")
    val transactionListById = findTransactionById(transactionsList, 2)
    println("Transaction with ID 2 in List: $transactionListById")
    val transactionsMap = mapOf(
        1 to Transaction(1, 100.0, "Groceries"),
        2 to Transaction(2, 250.0, "Electronics"),
        3 to Transaction(3, 75.0, "Books")
    )
    val filteredMap = filterTransactionsByAmount(transactionsMap, 100.0)
```

```
    println("Filtered Map: $filteredMap")
    val totalAmountMap = sumTransactionAmounts(transactionsMap)
    println("Total Amount in Map: $totalAmountMap")
    val transactionMapById = findTransactionById(transactionsMap, 2)
    println("Transaction with ID 2 in Map: $transactionMapById")
}
```

Output: