**Day 3: Interfaces, Encapsulation, and Advanced Concepts / Collections and Generics**

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

//Interface class

package task9to12  
interface Exportable {  
 fun exportToCSV(): String  
}

//Main class inheriting exportable interface

package task9to12  
import java.time.LocalDate  
// Base Transaction class implementing Exportable  
open class Transaction( val id: Int, val description: String, val amount: Double, val date: LocalDate):Exportable {  
 override fun exportToCSV(): String {  
 return "$id,$description,$amount,$date"  
 }  
  
 open fun details(): String {  
 return "Transaction(id=$id, description='$description', amount=$amount, date=$date)"  
 }  
}  
  
// Income class inheriting from Transaction  
class Income(id: Int, description: String, amount: Double, date: LocalDate, val source: String): Transaction(id, description, amount, date){  
 override fun exportToCSV(): String {  
 return "${super.exportToCSV()},Income,$source"  
 }  
  
 override fun details(): String {  
 return "Income(id=$id, description='$description', amount=$amount, date=$date, source='$source')"  
 }  
}  
  
// Expense class inheriting from Transaction  
class Expense(id: Int, description: String, amount: Double, date: LocalDate, val category: String) : Transaction(id, description, amount, date) {  
 override fun exportToCSV(): String {  
 return "${super.exportToCSV()},Expense,$category"  
 }  
  
 override fun details(): String {  
 return "Expense(id=$id, description='$description', amount=$amount, date=$date, category='$category')"  
 }  
}  
  
fun main() {  
 val transactions = *listOf*(  
 Income(1, "Salary", 3000.0, LocalDate.of(2025, 1, 10), "Company A"),  
 Expense(2, "Groceries", 150.0, LocalDate.of(2025, 1, 12), "Food"),  
 Expense(3, "Electricity Bill", 75.5, LocalDate.of(2025, 1, 15), "Utilities"),  
 Income(4, "Freelancing", 500.0, LocalDate.of(2025, 1, 20), "Client B")  
 )  
  
 *println*("CSV Export:")  
 transactions.*forEach* **{** *println*(**it**.exportToCSV()) **}**}

//Output

CSV Export:

1,Salary,3000.0,2025-01-10,Income,Company A

2,Groceries,150.0,2025-01-12,Expense,Food

3,Electricity Bill,75.5,2025-01-15,Expense,Utilities

4,Freelancing,500.0,2025-01-20,Income,Client B

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

package task9to12  
import java.time.LocalDate  
open class Transaction10(id: Int, description: String, amount: Double, date: LocalDate) {  
 private var \_id: Int = id  
 private var \_description: String = description  
 private var \_amount: Double = amount  
 private var \_date: LocalDate = date  
  
 // Public getters and setters  
 var id: Int  
 get() = \_id  
 private set(value) {  
 if (value > 0) {  
 \_id = value  
 } else {  
 throw IllegalArgumentException("ID must be positive.")  
 }  
 }  
  
 var description: String  
 get() = \_description  
 set(value) {  
 if (value.*isNotBlank*()) {  
 \_description = value  
 } else {  
 throw IllegalArgumentException("Description cannot be empty.")  
 }  
 }  
  
 var amount: Double  
 get() = \_amount  
 set(value) {  
 if (value >= 0) {  
 \_amount = value  
 } else {  
 throw IllegalArgumentException("Amount cannot be negative.")  
 }  
 }  
  
 var date: LocalDate  
 get() = \_date  
 set(value) {  
 if (!value.isAfter(LocalDate.now())) {  
 \_date = value  
 } else {  
 throw IllegalArgumentException("Date cannot be in the future.")  
 }  
 }  
  
 open fun details(): String {  
 return "Transaction(id=$id, description='$description', amount=$amount, date=$date)"  
 }  
}  
  
// Income class inheriting from Transaction  
class Incomes(id: Int, description: String, amount: Double, date: LocalDate, source: String) : Transaction10(id, description, amount, date) {  
  
 private var \_source: String = source  
  
 var source: String  
 get() = \_source  
 set(value) {  
 if (value.*isNotBlank*()) {  
 \_source = value  
 } else {  
 throw IllegalArgumentException("Source cannot be empty.")  
 }  
 }  
  
 override fun details(): String {  
 return "Income(id=$id, description='$description', amount=$amount, date=$date, source='$source')"  
 }  
}  
  
// Expense class inheriting from Transaction  
class Expenses(id: Int, description: String, amount: Double, date: LocalDate, category: String) : Transaction10(id, description, amount, date) {  
  
 private var \_category: String = category  
  
 var category: String  
 get() = \_category  
 set(value) {  
 if (value.*isNotBlank*()) {  
 \_category = value  
 } else {  
 throw IllegalArgumentException("Category cannot be empty.")  
 }  
 }  
  
 override fun details(): String {  
 return "Expense(id=$id, description='$description', amount=$amount, date=$date, category='$category')"  
 }  
}  
  
fun main() {  
 val income = Incomes(1, "Salary", 3000.0, LocalDate.of(2025, 1, 10), "Company A")  
 *println*(income.details())  
  
 val expense = Expenses(2, "Groceries", 150.0, LocalDate.of(2025, 1, 12), "Food")  
 *println*(expense.details())  
  
 // Attempting to modify properties  
 income.amount = 3500.0 // Works fine  
 *println*("Updated Income: ${income.details()}")  
  
 try {  
 income.amount = -500.0 // Throws an exception  
 } catch (e: IllegalArgumentException) {  
 *println*("Error: ${e.message}")  
 }  
}

//Output

Income(id=1, description='Salary', amount=3000.0, date=2025-01-10, source='Company A')

Expense(id=2, description='Groceries', amount=150.0, date=2025-01-12, category='Food')

Updated Income: Income(id=1, description='Salary', amount=3500.0, date=2025-01-10, source='Company A')

Error: Amount cannot be negative.

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

package task9to12  
  
import java.time.LocalDate  
//Base class  
open class Transaction11(val id: Int, val description: String, val amount: Double, val date: LocalDate) {  
 override fun toString(): String {  
 return "Transaction11(id=$id, description='$description', amount=$amount, date=$date)"  
 }  
}  
  
// IncomeEx class inheriting from Transaction  
class IncomeEx(id: Int, description: String, amount: Double, date: LocalDate, val source: String):Transaction11(id, description, amount, date){  
 override fun toString(): String {  
 return "IncomeEx(id=$id, description='$description', amount=$amount, date=$date, source='$source')"  
 }  
}  
  
// Expense class inheriting from Transaction  
class ExpenseEx(id: Int, description: String, amount: Double, date: LocalDate, val category: String) : Transaction11(id, description, amount, date) {  
 override fun toString(): String {  
 return "ExpenseEx(id=$id, description='$description', amount=$amount, date=$date, category='$category')"  
 }  
}  
  
// Function to print any collection of transactions  
fun <T : Transaction11> printTransactions(transactions: Collection<T>) {  
 *println*("Transactions:")  
 transactions.*forEach* **{** *println*(**it**) **}**}  
  
// Function to calculate the total amount in any collection of transactions  
fun <T : Transaction11> calculateTotalAmount(transactions: Collection<T>): Double {  
 return transactions.*sumOf* **{ it**.amount **}**}  
  
// Function to filter transactions in a map by a predicate  
fun <K, V : Transaction11> filterTransactionsInMap(transactionMap: Map<K, V>, predicate: (V) -> Boolean): Map<K, V> {  
 return transactionMap.*filterValues*(predicate)  
}  
  
fun main() {  
 val transactionsList = *listOf*(  
 IncomeEx(1, "Salary", 3000.0, LocalDate.of(2025, 1, 10), "Company A"),  
 ExpenseEx(2, "Groceries", 150.0, LocalDate.of(2025, 1, 12), "Food"),  
 ExpenseEx(3, "Electricity Bill", 75.5, LocalDate.of(2025, 1, 15), "Utilities")  
 )  
  
 val transactionsSet = *setOf*(  
 IncomeEx(4, "Freelancing", 500.0, LocalDate.of(2025, 1, 20), "Client B"),  
 ExpenseEx(5, "Fuel", 60.0, LocalDate.of(2025, 1, 18), "Transport")  
 )  
  
 val transactionsMap = *mapOf*(  
 1 *to* IncomeEx(6, "Bonus", 1000.0, LocalDate.of(2025, 1, 25), "Company A"),  
 2 *to* ExpenseEx(7, "Rent", 1200.0, LocalDate.of(2025, 1, 1), "Housing")  
 )  
  
 // Using generic functions  
 *println*("Handling List:")  
 *printTransactions*(transactionsList)  
 *println*("Total Amount (List): ${*calculateTotalAmount*(transactionsList)}")  
  
 *println*("\nHandling Set:")  
 *printTransactions*(transactionsSet)  
 *println*("Total Amount (Set): ${*calculateTotalAmount*(transactionsSet)}")  
  
 *println*("\nHandling Map:")  
 val filteredMap = *filterTransactionsInMap*(transactionsMap) **{ it**.amount > 500 **}** filteredMap.*forEach* **{** (key, value) **->** *println*("Key: $key, Value: $value")  
 **}**}

//Output

Handling List:

Transactions:

IncomeEx(id=1, description='Salary', amount=3000.0, date=2025-01-10, source='Company A')

ExpenseEx(id=2, description='Groceries', amount=150.0, date=2025-01-12, category='Food')

ExpenseEx(id=3, description='Electricity Bill', amount=75.5, date=2025-01-15, category='Utilities')

Total Amount (List): 3225.5

Handling Set:

Transactions:

IncomeEx(id=4, description='Freelancing', amount=500.0, date=2025-01-20, source='Client B')

ExpenseEx(id=5, description='Fuel', amount=60.0, date=2025-01-18, category='Transport')

Total Amount (Set): 560.0

Handling Map:

Key: 1, Value: IncomeEx(id=6, description='Bonus', amount=1000.0, date=2025-01-25, source='Company A')

Key: 2, Value: ExpenseEx(id=7, description='Rent', amount=1200.0, date=2025-01-01, category='Housing')

**Task 12:** Utilize Kotlin's collection libraries to manage a collection of User objects, enabling the addition and removal of users.  
  
package task9to12  
  
data class User(val id: Int, val name: String, val email: String)  
  
class UserManager{  
 private val users: MutableList<User> = *mutableListOf*()  
  
 // Add a new user  
 fun addUser(user: User): Boolean {  
 return if (users.*any* **{ it**.id == user.id **}**) {  
 *println*("User with ID ${user.id} already exists.")  
 false  
 } else {  
 users.add(user)  
 *println*("User added: $user")  
 true  
 }  
 }  
  
 // Remove a user by ID  
 fun removeUserById(id: Int): Boolean {  
 val userToRemove = users.*find* **{ it**.id == id **}** return if (userToRemove != null) {  
 users.remove(userToRemove)  
 *println*("User removed: $userToRemove")  
 true  
 } else {  
 *println*("User with ID $id not found.")  
 false  
 }  
 }  
  
 // Get all users  
 fun getAllUsers(): List<User> {  
 return users.*toList*() // Returns an immutable copy of the list  
 }  
  
 // Find a user by email  
 fun findUserByEmail(email: String): User? {  
 return users.*find* **{ it**.email == email **}** }  
}  
  
fun main() {  
 val userManager = UserManager()  
  
 userManager.addUser(User(1, "Ajay", "ajay@gmail.com"))  
 userManager.addUser(User(2, "Balu", "balu@gmail.com"))  
 userManager.addUser(User(3, "Charan", "charan@gmail.com"))  
  
 // Attempting to add a duplicate user  
 userManager.addUser(User(2, "Balu", "balu@gmail.com"))  
  
 // Removing a user  
 userManager.removeUserById(3)  
  
 // Attempting to remove a non-existent user  
 userManager.removeUserById(5)  
  
 // Displaying all users  
 *println*("\nAll users:")  
 userManager.getAllUsers().*forEach* **{** *println*(**it**) **}** // Finding a user by email  
 val foundUser = userManager.findUserByEmail("ajay@gmail.com")  
 *println*("\nFound user: $foundUser")  
}

//Output

User added: User(id=1, name=Ajay, email=ajay@gmail.com)

User added: User(id=2, name=Balu, email=balu@gmail.com)

User added: User(id=3, name=Charan, email=charan@gmail.com)

User with ID 2 already exists.

User removed: User(id=3, name=Charan, email=charan@gmail.com)

User with ID 5 not found.

All users:

User(id=1, name=Ajay, email=ajay@gmail.com)

User(id=2, name=Balu, email=balu@gmail.com)

Found user: User(id=1, name=Ajay, email=ajay@gmail.com)