Chapter 9 – A Full Console Application

Let's take a top-down approach on this one. We are going to create a **basic banking app** that allows us to deposit, withdraw, and earn interest on the balance. The app also stores and loads the data from a file. It also handles multiple accounts, so it needs to keep track of names and account numbers; note that account numbers are unique, but names are not.

> The ultimate goal of this chapter is for you to create your own application that has a multi-class design. This could be league stats for teams or players, order entry for a restaurant, inventory system for a business, mark system for a school, etc. Try to keep this in mind as you are going through this chapter.

To run the system, we use a simple state machine with just 2 states each with its own menu. Before continuing, you should try to figure out what should be included in these menus. It may vary from what I have come up with; this depends on how much or little you know about banks. Don't worry if you don't get it, this takes practice and you just want to improve over time. If this were a real application, you would easily spend more than a minute doing this, and you may also consult with a subject expert. The other consideration is how flexible your system is for additions/changes in the future.

Main Menu:

```
P - show customers
Z - show all customer transactions
A - add customer
N - select customer by name
# - select customer by number
I - pay interest on all active accounts
Q - quit
```

Customer Menu:

```
B - balance
D - deposit
W - withdrawal
T - show transactions
X - delete current customer
C - command menu
Q - quit
```

The next major design is deciding what account data is needed. If this were a full app, we would need lots of data like legal information to verify that the person is who he claims to be. However, we are making a basic app so all we need is a name, account number, and a list of transactions.

> **Exercise 9.1: Make a list of necessary customer attributes and submit this as a Word document. I will compare it to my list. Be as realistic as possible. Don't add frivolous data just to make your list longer. Half a mark for 6 items of my list, and a full mark for over 6 items.**

The final major piece is deciding what transaction data is needed. Again, we are only storing minimal data: the amount, transaction type, and transaction date. Real data would include a description, location, employee that was responsible, how the money is transferred in/out such as cash, cheque, electronic transfer, delayed transfer; each of which generates more data.

When creating a large app, you need to have a plan so that downtime (when the app is broken and can't run) is minimized. The plan should also be organized into independent sections so that programmers aren't waiting for others to finish their work before starting (called project dependencies). The sections for this app are: menus, the account class, the transaction class, file routines, and miscellaneous routines. If we were programming in a team, there would be a meeting to discuss the API's (Application Program Interface) and all the API's would be created with empty methods. This way allows everyone to code and when sections are ready, they can be dropped in and tested. The last part of a good plan is that it has platform independence, this way it can be easily ported to different platforms. For the banking app, I have ported it from C++ to Java and Objective-C. If you compare the code, they have a lot of similarities.

Transaction class API:

```
class Transaction {
        TRANSACTION_TYPES   t_type;
        double              amount;
        int                 date;

        Transaction();
        Transaction(TRANSACTION_TYPES t, double amount);
        string saveString();
        void readLine(string line);
};
```

This is a small class and therefore doesn't require a lot of methods. Basically this class just holds data. *saveString* and *readLine* are methods to serialize the data, that is, to prepare the data to save on the disk. For an actual app, we would send transactions to a database server.

Account class API:

```
class Account {
        private: array      transactions;
        string              name;
        int                 number;
        int                 deleted;

        Account();
        Account(string name, int acctno, double amount);
        string saveString();
        void readLine(string line);
        bool deposit(double amount);
        bool interest(double amount);
        bool withdraw(double amount);
        double balance();
        void showTransactions();
};
```

This class has a few more methods than Transactions. Notice that transactions are private; instead of letting other code access the transactions array, the Account class provides all the necessary methods to add transactions and display transactions. This allows us to implement the transactions in different ways if necessary without breaking the code. For example, if we changed the transactions from a linked list to an array, everyone accessing the transactions would have their code broken.

File routines:

```
bool docsSave(string filename, string str);
string docsLoad(string filename);
bool docsExist(string filename);
```

Loading and saving files is a bit tricky, so it's best to create methods to handle this. For the program, we simply want to be able to save and load a string from methods. We want these methods to handle errors, buffering, opening, and closing of files.

Miscellaneous routines:

```
int intDate();
static string getTrimmedInput();
```

On the platform independence theme, we want to get a consistent integer date. Since we frequently get input, we also want to trim the input to reduce errors. Trimming is removing leading and trailing blank and other white spaces. Trimming also ensures that the input command is at the first character.

Menu System:

```
USER_STATE state = COMMAND;
if (docsExist(fileName)) {
	// read all the accounts before starting
}

do {
	switch (state) {
	case COMMAND:
	default:
		print Command Menu
		break;

	case ACCOUNT:
		print Account Menu
		break;
	}

	print "Q - quit\n";
	print "Please enter command: ";
	string inputString = getTrimmedInput();

	switch (inputString[0]) {
	default:
		print "Unrecognized command\n";
		break;
	case 'Q':
		// save all the accounts before exiting
		return 0;
	}
} while (true);
```

This menu look should look familiar to you. It has the same input loop that we've always used. The difference is that it uses a *switch* statement instead of *if..else* statements. Although it only handles quitting, it is easy to add commands as they are completed.

It is important to be able to understand the design because there is very little detail concerning implementation. This is the same as using the standard libraries; we understand what we get from calling a library method even though we don't know how it's implemented. By calling several library methods, we can get a fully function program.

Java: Arrays – fixed size, sort; ArrayLists – adjustable size; String - https://docs.oracle.com/javase/7/docs/api/java/lang/String.html, split;

Obj-C: NSArray, sortArrayWith..; NSMutableArray - adjustable; NSString - https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSString_Class/, componentsSeparatedBy..;

When standard libraries are available, it's a good idea to use them because they are generally efficient and bug free. However, you still need to use them properly to avoid errors. Because the languages now diverge, just skip to the relevant sections. Note that I frequently say function (since I'm old school): functions return a value and procedures do not return a value; in Java and Objective-C they say methods which means function or procedure.

We have looked at character strings, but let's look at the library strings.

Strings - C++

C++ uses *string*.

Strings - Java

Java uses *String*.

Strings - Objective-C

Objective-C uses *NSString*. It is powerful and contains many methods including the ones that you have coded-up. Many methods in the documentation are deprecated which means they are no longer used in newer libraries, so use the newer methods to avoid errors. There are many ways to construct NSString and even more methods are available.

We have looked at the basic arrays, let's look at arrays with more support.

Arrays - C++

Arrays - Java

Arrays - Objective-C

We start with NSArray.

Enough with the background, let's look at some code before continuing. We will look at the syntax for creating a class and how to create and delete objects. Recursion can be an elegant solution, but in general you want to avoid it if possible because for large lists, it will be a big hit for the stack.