Procedural vs Object-Oriented Programming

Introduction

In order to ascertain an appreciation of various programming paradigms, this project involved developing a primitive shop application in both a procedural language (C) and an Object-Oriented language (Java).

Language "Levels"

In its basest form, the lowest level of programming instruction is binary, or machine code, which is a sequence of 1's and 0's, that can be understood by the processor as instructions. It would be a near impossible venture to attempt code any kind of a complex system in this manner.

The next level up is assembly code, which is consists of blocks of primitive English language commands, instructing the processor to carry out simple low-level commands, such as ADD, SAVE etc. This is a big step up from machine code, however it is still very cumbersome and limited in its ability to allow for swift development of complex applications.

Next level is high-level languages, which includes C and Java, as well as many other examples. These languages are much closer to English language commands and make it far easier to develop complex structures, applications and systems.

Of the two, C would be the lower level language, in that it operates closer to the processor that Java.

Abstraction

Think of it as the amount of interpretation or unravelling that has to be done to the code, before the CPU can understand it. Or more correctly, the amount of low-level work that is hidden, or abstracted) from the programmer.

The higher-level the language, the more abstraction is involved. This simply means that the programmer can more easily get on with the business of coding for problem-solving, rather than worrying about the nitty gritty of assigning data to register addresses, or manually allocating memory space for variable of undefined size. The abstraction means that the language compiler, or interpreter, takes care of a certain amount of that for the programmer either at compile time (if using a compiler), prior to running the application, or dynamically at run-time, if using an interpreter.

C vs Java

While developing versions of the shop application for this assignment, there have been a number of obvious differences between the two languages.

1. Abstraction

As mentioned earlier in the document, abstraction is "code-speak" for how the language manages to hide away lower levels of complexity involved in the translation of the code into machine code, as well as other aspects such as memory management. Aspects of the back-end workings are hidden and taken care of in the background by the compiler, so that the developer does not need to worry about the nitty-gritty of how variables are stored and managed and how the code written gets from high level English, right down to machine code that can be understood and executed by the CPU. From the perspective of a desktop end-

user, All they need to do is to be able to carry out their work by interacting with a keyboard, a mouse and the VDU. They do not need to worry about what happens inside the tower case. They move the mouse, the pointer moves on the screen. The user doesn't care how that happens, just that it happens correctly and as expected.

2. Memory Management

In terms of memory management, the lower level languages require a lot more work in terms of the manual allocation and releasing of space in memory for storing variables and other run-time data. In the case of C, in some cases, where we are unsure about the amount of space that may be required for a particular variable value, we might declare that as a pointer in the code, or where we know what the maximum size might be, but the size of the variable will only be known at run-time, we may need to manually allocate memory of a certain size, using malloc().

Java, however, is a higher level language than C and it takes care of all of the manual interventions hat C requires "under the hood", so that the programmer simply needs to declare the type of variable and assign a variable name and an initial value, if appropriate. Java then worries about the size of the memory allocation required and takes care of the allocation and releasing of that space as it is required, or not. One exception to this might be the **float** vs **double datatypes**. If we know in advance that our floating point value will not fit within the range that is offered by the float type, then double can be used for larger values, as it allocations almost double the space of float. This is the extent to which a programmer needs to worry about memory management in Java.

3. Data Structure vs Object

C offers the use of **structs** in order to allow the application to work with more complex entities, that might have many characteristics that all relate to the same thing. For example, in this assignment the shop itself is not just a shop … it is an entity that has a number of variable characteristics, such as cash and stock, or the stock of a product, which has a product name, price and the amount of stock held. All of these need to be catered for in memory and they are all connected to the same "parent" thing, so the struct offers a way to link these characteristics together. The limitation here is that the struct is essentially an inanimate "object" (I use the term *object* here in "real" terms, as opposed to thinking of it as an object as part of an O-O paradigm), in that it has certain characteristics, or variables, associated with it, but it has no functionality in its own right. There is no action/functionality that can be done, that belongs to the struct. Any functions acting on the struct are external to it and are called externally, to manipulate the structs characteristics.

Java goes a step further and rather than using structs, it employs the concept of objects, hence the term *Object-Oriented* programming language. Objects differ from structs in that they have the same characteristics/variables associated with the parent object, but with Java, the object can also have functionality associated with a specific object, in the form of methods that are included within the object itself. There are a number of benefits to this, not least a much clearer coding structure. It also offers compartmentalization, where appropriate. An object method can be made public, meaning that it can be seen and used across the entire module/application that it is a part of, or private, meaning that it can only be seen and used within the scope of the object itself. This offers great security benefits for more sensitive applications.