Unit 0. Overview

Learning Outcomes

This unit provides an overview of the aims of CS2030/S and how the key concepts covered in CS2030/S are related to each other.

What is This Module About?

CS2030/S is designed for students who have gone through a typical basic programming module and have learned about problem-solving with simple programming constructs such as loops, conditions, and functions. In a typical introductory programming module, such as CS1010 and its variants at NUS, students tend to write small programs (in the order of tens or hundreds of lines of code) to solve a programming homework problem, work alone on their code, and move on to solve the next problem once the homework is done.

The first aim of CS2030/S is to change the students' mindset and to make them learn to write software that will continue to evolve as software requirements change and to write software that will be read and modified by other programmers (including their future selves).

The second aim of CS2030/S is to level up the complexity of programs that the students write, from the order of hundreds of lines to thousands of lines. CS2030/S bridges the students between writing toy programs to solve specific problems in CS1010 and writing larger real-world software in their later modules, such as CS2103 Software Engineering.

A programming language is the medium in which programmers can express their intention and construct software, and thus is critical to supporting the aims above. With the appropriate features and tools, one can tame the complexity of software, make the code written friendlier to other programmers, and easier to evolve. The third aim of CS2030/S is thus to expand the students' minds on different ways one can construct software and the principles behind some of the programming language constructs. In particular, CS2030/S focuses on *objects*, *types*, and *functions*, as three key constructs for building programmer-friendly software. It covers both object-oriented and functional paradigms as two different approaches to constructing software, with a strong emphasis on type safety.

The final aim of CS2030/S is to introduce students to programming language concepts and to bridge them from introductory programming to advanced modules such as programming language design and implementation. Part of CS2030/S introduces students to the design decisions behind some of the constraints and the workings behind the programming language compilation and execution, giving them a glimpse inside the programming system that so far has been mostly treated as a black box in introductory modules.

The Choice of Java

We decided to use one programming language throughout the module. This decision means that we need to pick a language that is strongly typed with static typing and supports both object-oriented and functional programming. Considering multiple factors, we decided to choose Java for CS2030/S, for its popularity, syntax familiarity, and smoother transitions to later modules in the NUS computing curriculum.

While Java is not the most elegant programming language when expressing programs in a functional style, we hope that students can still learn the principles of functional programming and apply them in other programming languages. This choice is a trade-off between having to switch to a different language in the middle of a module.

What This Module is not About

This is not a module on Java programming. We will not comprehensively cover Java syntax and features, except those relevant to the concepts we teach. In fact, we will avoid and even ban students from using certain Java features (such as `var`) for pedagogical purposes.

This is not a module on software engineering either. Software engineering is a broad discipline on its own and deserves another module. Rather, this module is about the programming principles and constructs on top of which programmers can design better software. To motivate the importance of these principles and constructs and see how they can be used, we will inevitably cover some of the software engineering design principles, such as Liskov Substitution Principle (the L in SOLID), Tell-Don't-Ask, Composition over Inheritance, etc. But we will not comprehensively cover object-oriented design or software design in general (e.g., we will not cover S, O, I, and D in SOLID).

Finally, CS2030/S is not a module that focuses on computational efficiency. We have CS2040/S for that. In CS2030/S, although reducing computational cost still plays a role, this is not the only cost that matters. CS2030/S is also concerned with the human cost of debugging or maintaining software. In striving for simpler software that is easier to maintain and extend, we may have to sacrifice computational efficiency.

Taming Complexity in Software Development

An underlying theme of CS2030/S is taming complexity in software development. There are objective metrics with which one can measure the complexity of software, but here, we will loosely define complexity as anything that increases the likelihood of bugs in a program.

Let's start by considering a simplified view of what a software program is. One can view a software program as a collection of data variables and instructions on how to modify these variables. A program is generally written to meet a given requirement: given one or more input variables, the program should perform the computation to produce the output variables, in a way that meets the requirement. Often, the program stores information in the intermediate variables while performing the computation.

As a student who has gone through an introductory programming module such as CS1010 and its variants, you should be familiar with the view above, and you should have some experience writing a program to solve a given computational problem. The program you have written for these introductory modules are small "toy" programs mostly -- they consist of only a few hundred lines and tens of variables, at most.

Software development in the real world, however, is far more complex than what you have experienced. A software program rarely solves a well-defined computational problem only. It often requires multiple components, such as user interface, data storage, and business rules, intricately interacting with each other to attain a set of functionalities.

As the requirement of the software becomes more complex, the number of variables that need to be kept track of increases; the logic of the computation the programmer needs to maintain the variables becomes more complicated. Further, it is often that the variables are interdependent. For instance, updating a variable might require updating another; how a variable should be updated might depend on another variable. As the number of variables increases, so is the number of relationships between the variables that the programmer has to keep track of. Failure to correctly maintain the variables and the relationship between them most likely will lead to bugs.

Further, real-world software rarely remains static. This property is again different from what you have experienced in your introductory programming module, where once the instructors release a programming assignment, they rarely go back and change the requirement. In the real world, software evolves -- new features are added, business rules change, and better algorithms are deployed. The code needs to be updated accordingly --

adding new variables and new computation; changing how variables are updated or are dependent on each other. Updating the code of an already-complex software program to keep up with the requirement, if not managed properly, can lead to bugs.

Real-world software is often the product of teamwork from multiple programmers, where the software development process is unlike what you have experienced in your introductory programming module, where you solve your homework individually. When multiple programmers work together, the interdependency between the states needs to be communicated and handled properly and consistently across the programmers. One programmer's modification to the code should not introduce bugs into another programmer's code.

Since software evolves, the notion of "multiple programmers" actually applies even to software developed by a single lone programmer across time. Changing one's code should not introduce new bugs to other parts of the code that is written some time ago.

Strategies to Tame Complexity

Good Software Development Practices

If you are taught properly in your introductory programming modules, you should already be familiar with good programming practices that help to tame the complexity and reduce the chances of bugs. These practices include

- Comment your code: Commenting your code provides *in situ* communication between you and other programmers on the team, as well as between you and your future self, on the non-obvious purpose of the states and the relationships between the states. Such comments help to enhance the understanding of what the code is doing and to remind whoever is updating to code to modify appropriately when the requirement changes.
- Use a coding convention: Adhering to a coding convention helps improve code readability, reducing the cognitive barrier when one programmer reads another programmer's code and allowing the reader to understand the code more easily and thoroughly.

CS2030/S will continue to enforce these good programming practices.

Functions

You should also be taught to always break your code down into functions, each one performing a simple, specific, task. The functions can then be composed to solve larger and more complex tasks. Functions are an important programming structure in taming code complexity, it allows programmers to (i) compartmentalize computation and its effects, reducing the number of interactions to a few well-defined ones (through arguments and return values); (ii) hide the implementation details so that they can be changed later without affecting other parts of the code; (iii) reuse computations and thus write code that is more succinct and easier to understand/change.

In CS2030/S, you will not only continue to break your computation into functions, but we will kick it up several notches. A major part of CS2030/S is to introduce you to more programming paradigms and language tools that allow you to compartmentalize computations, hide details, and reduce repetition.

The Abstraction Principle

The last point above about why it is important to code in small, reusable functions, follows what is called the *Abstraction Principle*[1]. The principle states that:

> "Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts."

This principle is something that we will visit over and over again in CS2030/S, applying it to different varying parts of a program. In the case of functions, the "varying parts" are the values on which we wish to perform the computation on. We will also apply this principle to (i) types, abstracting them out as parameterized types or subtypes, and (ii) to sub-computation, abstracting them out as first-class functions. These concepts: generics, subtypes, and first-class functions, underlies most of the content of CS2030/S.

Erecting an Abstraction Barrier

Another important strategy for taming complexity is the *abstraction barrier*. Let's separate the role of a programmer into two, in the context of writing functions: the *implementer*, who implements the function, and the *client*, who calls the function. The implementer should compartmentalize the internal variables and the implementation of the function, hiding them behind the abstraction barrier. The parameters and the return values are the only communication gateways across the barrier.

The abstraction barrier is something that we will refer to repeatedly in CS2030/S as well. We will see how we maintain this barrier not only in the context of functions, but also variables and computations on these variables together, by encapsulating them as *objects*, and hiding details from the client through *access modifiers*. These ideas form two of the core principles of *object-oriented programming*: *encapsulation* and *abstraction*.

Code for Change

The abstraction barrier, if erected and maintained properly, reduces code complexity. It, however, also reduces flexibility as the software evolves. If the client wishes to modify the computation protected by the abstraction barrier, it will need the help of the implementer. In CS2030/S, we will see two ways we can modify the computation behind the abstraction barrier, *without changing the code behind the barrier*.

First, we will introduce the concept of *inheritance* and *polymorphism*, the other two core principles of object-oriented programming. These object-oriented mechanisms allow programmers to easily extend or modify the behavior of existing code.

Second, we will introduce *closure*, an abstraction to computation and its environment, that we can pass into the functions behind the abstraction barrier to perform a computation. The second idea, if carried to the extreme in terms of flexibility, leads to the concept of *monad* in the functional programming paradigm. A monad is a computational structure that allows objects to be composed and manipulated in a succinct and powerful way.

Types

Allowing a programmer to change the behavior of the existing code without changing the code could lead to more bugs, if not managed properly. To prevent this, both the programming language system and the programmers, have to adhere to certain rules when extending or modifying the behavior of the existing code. Java and many other typed languages have *type systems* -- a set of rules that governs how variables, expressions, and functions interact with each other. You will learn about subtyping and the Liskov Substitution Principle, two notions that are important to constraining how inheritance and polymorphism should be used to avoid bugs.

A type system is also an important tool to reduce the complexity of software development. Constraining the interactions among the variables, expressions, and functions, it reduces the possible interdependence between these programming constructs. Furthermore, any attempt by programmers to break the constraint can be caught automatically by the compiler. By utilizing the type system properly, we can detect potential bugs before they manifest themselves.

A reason CS2030/S chooses to use Java is due to its type system. CS2030/S will introduce the concept of types, subtypes, compile-time vs. run-time types, variants of types, parameterized types, and type inferences, in the context of Java. We will see how we can define our own types (using *classes* and *interfaces*) and define relationships between them. We will see how we can define parameterized types and generic functions that take in types as parameters. These concepts apply to many other programming languages.

Eliminating Side Effects

We have discussed how functions can compartmentalize computations and limit their complexity within their body. For this approach to be effective, the function must not have any side effects -- such as updating a variable that is not within the function. Such functions, called *pure _functions*, are one of the key principles of the functional programming paradigm and is something that we will explore to kick off the section on functional paradigm in CS2030/S.

A related idea in object-oriented programming we will cover in CS2030/S is *immutability* -- once we create an object, the object cannot be changed. In order to update an object, we need to create a new one. With immutability and pure functions, we can guarantee that the same function invoked on the same objects will always return the same value. This certainty can help in understanding and reasoning about the code behavior.

---

1. This principle is formulated by Benjamin C. Pierce in his book "Types and Programming Languages." ↩