COMP2511

Tutorial 1

Introduction

- Hello! I’m James (he/him)
- 4th year Computer Science/Maths student, briefly did Psychology initially
- Second term tutoring this course!
- Spent the entire holidays playing video games (Marvel Rivals, Pokémon TCG Pocket, Ace Attorney)
- Loves pandas and similar animals like raccoons
- Very short (evidently)
- I’m very happy to be interrupted if you have any questions during the tutorial or if you’d like me to re-clarify anything!

Icebreakers

- Please introduce yourselves and your year/degree, and if you'd like, anything you would like to share!
  - Fun facts
  - Anything cool you did during the holidays (doing nothing counts!)
  - Your go-to food/drink spot on campus

Welcome to COMP2511!

- In previous courses, you became more proficient and confident in your ability as a **programmer**.
  - COMP1531: Working on large-scale projects as a team, web-based programming
  - COMP2521: Exploring various data structures and solving a range of algorithmic problems
- In this course, the focus is on developing your ability as **designers**, in the context of programming.

Assessments

- **Labs (15%)** - 7 labs, each *manually marked* out of 10. Your overall lab mark is out of 60, leaving a buffer for one lab.
- **Assignment 1 (15%)** - *individual* assignment where you will build a system from the ground up, assessing your understanding and application of the initial (yet extremely important) topics of the course.
- **Assignment 2 (20%)** - *pair* assignment assessing your ability to refactor an extensive codebase using techniques taught in the course.
- **Assignment 3 (optional, 8% bonus)** - to make up any marks lost in previous assignments, should be treated as extension.
- **Final Exam (50%)** - *40% hurdle*, approximately 50% of the exam's content will be very similar to labs and tutorial examples.

What is 'good' design?

- Design is inherently a subjective topic, so what do we necessarily mean by 'well-designed code' or 'good design'?
- What are some things that we can all agree are desirable in code?
  - At the text level, code that adheres to widely used **conventions** and stylistic **patterns** for readability.
  - Logic that is **correct**, yet **simple** enough to read and understand.
  - Being able to focus on how things operate at a **high level**, rather than having to worry about concrete implementation details (e.g. ADTS in 2521).
  - Having responsibilities and logic be **separated** into different parts that work together as a whole to form a **cohesive** program (e.g. your COMP1531 project).
  - Being able to easily **adapt** and account for changes in requirements.
- These are ideas that we will carry throughout the entire course!

What is Object-Oriented Programming?

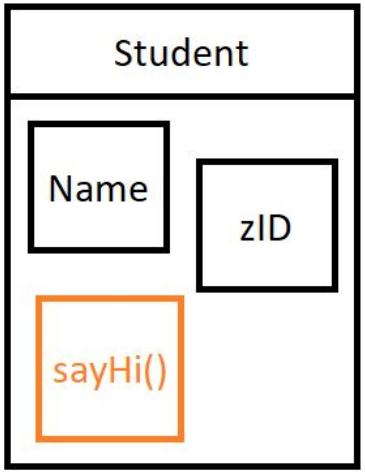
- **Object-oriented programming** is a specific style (i.e. a *paradigm*) of programming which dictates that logic should be organised around user-defined types called **classes** and the interactions between them.
- A class is a structure which holds its own data (like structs in C or interfaces in TypeScript) **and** methods (i.e. functions) that act on that data and potentially interact with other classes.
- A concrete *instance* of a class is referred to as an **object**.

Blueprint vs. Product

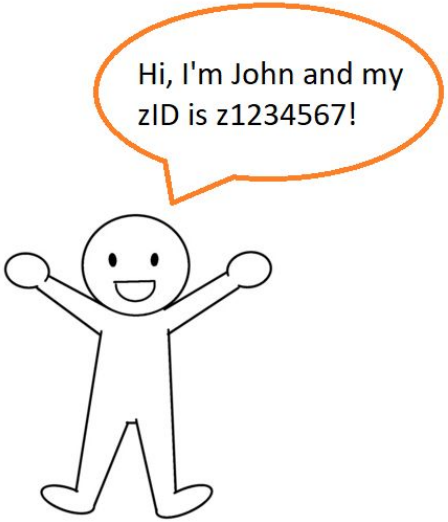
A blueprint of a house; the *idea* that informs what components the house will have and how it will be structured.

A house; the physical *realisation* of the blueprint (Sorry, I couldn't find the exact house shown in the blueprint!)

Class vs. Object



A class; the template/blueprint to create an object. Name/zID are **fields**, sayHi() is a **method**.



An object; an **instantiation** of the Student class. This object has its own name, zID and method to introduce itself.

(Some) Key Tenets of Object-Oriented Programming

- The bulk of this course will be around exploring the OOP paradigm, and how we can use it to design larger-scale systems **effectively**.
- Object-Oriented Programming is most commonly associated with the principles of **encapsulation**, **abstraction**, **inheritance** and **polymorphism**:
  - **Encapsulation** - grouping data and the mechanisms that act on that data together; this is facilitated through the use of classes. The fields and methods of a class should be very closely tied together.
  - **Abstraction** - hiding away unnecessary details of how things are implemented and only exposing the essential features or functionalities to users, i.e. you just need to know the *what*, and not the *why*.
  - We will look at **inheritance** and **polymorphism** next week, and more principles that guide 'good' applications of OOP in Week 4!

Cons of Object-Oriented Programming

- If a system has a lot of moving pieces that need to interact together, an object-oriented approach may be very suitable!
- However, that's not to say that OOP is the be-all end-all solution to *everything*. The aim of the course is to introduce you to this approach and get you to appreciate the ways in which it can be applied effectively, not 'the best thing invented since sliced bread'.
- Some potential drawbacks and pitfalls of object-oriented programming include:
  - Languages that support OOP are mostly high-level languages (such as Java and Python), where the overhead and memory cost of managing objects could get very large and simply not sufficient over a highly performant language like C.
  - Misuse of the paradigm could make programs even **more** complicated unnecessarily. A program that only has one main component certainly doesn't need to be implemented with an object-oriented approach!

Java and Gradle

- We will use **Java** for all code in this course.
- Java is a friendly entry-point for programmers getting started with OOP.
- Java shares similar syntax with C in its static typing and variable declarations.
- All code in Java has to exist within a class.
- Unlike C, Java has automatic **memory management**!
  - This means that you won't have to deal with things like memory leaks in 99% of cases.
- **Gradle** is a tool used for dependency and build management.
  - You shouldn't have to interact with Gradle outside of using some commands - it should be making your experience in this course easier, not more complicated. Treat it as a black-box to compile your projects!

Important Terminology

- Access modifiers
  - Keywords that dictate what can access and use particular class fields/methods.
  - **public** - *all* files and classes can access this field/method
  - **private** - *no* files and classes can access this field/method outside of the class in which it belongs
  - there are a couple more, but you will only need to use these in most cases!
- Constructors
  - You can think of these as functions/methods that return an instance of your class, given a list of parameters.
  - They are declared like typical methods, with the method being the name of the class itself.
  - e.g. a constructor for a class named Student with fields name and zID can be declared as public Student(String name, String zID) { ... }
- Instance fields/methods
  - In the Student example again, each concrete student should have their own name and zID, i.e. each Student instance has its own 'copy' of the field, making name and zID **instance fields**.
  - If s is an instance of Student (i.e. a variable of type Student), sayHi() can be invoked by doing s.sayHi()

Important Keywords

- **static**
  - In contrast to instance fields/methods, the **static** keyword declares that a field or method is shared across all instances of a class, i.e. the field/method belongs to the class itself, rather than to concrete instances.
  - If sayHi() was a static method, it could be invoked by doing Student.sayHi() [if sayHi() includes a student's name and zID, does it make sense for it to be a static method?].
  - Can you think of instances where a static field or method makes sense?
- **this**
  - When used in an instance method, refers to the actual instance in which the method was invoked from.
  - Useful for when the instance has a field that shares the same name as a local variable, as without this keyword the local variable is prioritised.
- **new**
  - You can think of this as the Java equivalent of malloc. This keyword is followed by the constructor for a certain object, and allocates memory to instantiate the class.
  - Student s = new Student("John", z1234567);

Live Coding

- HelloWorld.java
  - Write a program with a main method that prints out “Hello World!” to the terminal and run it, then push the code onto git.
  - [KEY TAKEAWAYS] Java output, writing and running the main method, git revision
- Sum.java
  - Write a program that uses the Scanner class which reads in a line of numbers separated by spaces, and sums them.
  - [KEY TAKEAWAYS] Java input, control flow, importing classes, using a static method
- Shouter.java
  - Inside a new file Shouter.java, write a program that stores a message and has methods for getting the message, updating the message and printing it out in all caps. Write a main() method for testing this class.
  - [KEY TAKEAWAYS] Declaring class fields and methods, constructors, encapsulation