COMP2511

Tutorial 1

Introduction

- Hello! I’m James (he/him)
- 4th year Computer Science/Maths student, briefly did Psychology initially
- Fourth term tutoring this course!
- Loves playing video games, 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!
    - I do have a tendency of losing track of time and occasionally going on tangents or trying to over-explain things. I’ll try to be much more wary of this, but don’t be afraid to call me out on this if it happens!

Icebreakers

- Please introduce yourselves and your year/degree, and if you'd like, anything you would like to share!
    - fun facts about yourself
    - anything cool you did during the holidays (doing nothing counts!)
    - any recent obsessions (music, games, hobbies)?!
    - your go-to food/drink spot on campus
    - how many hours of sleep you got last night

Welcome to COMP2511!

- In previous courses, you became more proficient and confident in your abilities as **programmers**.
    - 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 (take sum of all marks), leaving a buffer for 10 marks.
- **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%)** - *individual* (!) assignment, where you will be using some more advanced applications of OOP, and be presented with a scenario to assess your understanding of software architecture.
- **Final Exam (50%)** - *40% hurdle*, approximately 50% of the exam will be very similar in style to lab exercises 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 (**abstraction**).
  - Having responsibilities and logic be **separated** into different parts that work together as a whole to form a **cohesive** unit (e.g. hopefully, your COMP1531 project!).
  - Having the ability to easily **adapt** to account for changes in requirements.
- These are key ideas that we will carry throughout the entire course!

Object-Oriented Programming

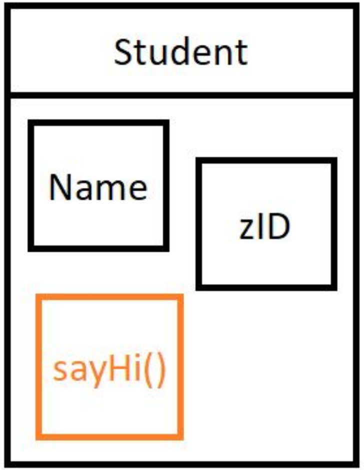
- During the first half of the term, we will be focusing on **object-oriented programming** (formerly, the entire focus of the course!).
- Object-oriented programming (OOP) is a programming paradigm (i.e. style/model) which dictates that logic should be organised around user-defined types called **classes** 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**.

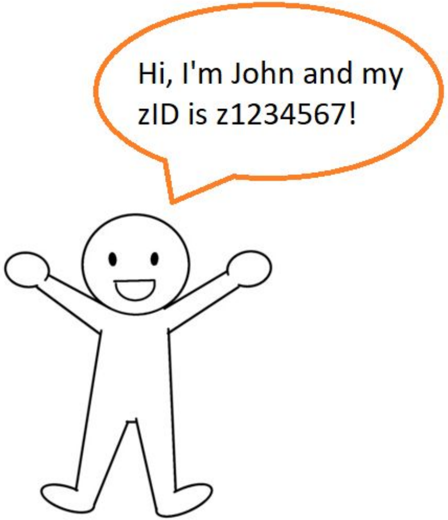
Blueprint vs. Product

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

The actual house; the physical *realisation* of 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

- We will be exploring how we can apply OOP to design larger-scale **extensible**, **flexible**, **maintainable** and **reusable** systems.
- 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 *how*.
    - We will look at **inheritance** and **polymorphism** next week, and more principles that guide 'better' 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, OOP **isn't** something that you can just apply in every single scenario without consideration.
- Some potential drawbacks and pitfalls of OOP:
  - Object-oriented languages (e.g. Java, Python) are mostly high-level languages, where the overhead and memory cost of managing objects could get very large. An object-oriented approach may not be suitable if you need a very performant/speedy program.
  - It's fairly easy to misuse the paradigm, and actually make programs **more** complicated unnecessarily. If you are considering using OOP, you should always consider whether it's actually applicable to the scenario.

Java and Gradle

- We will use **Java** for all code in this course (specifically, Java 17).
- 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 *almost* all 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 they're in
  - there are a couple more, but you will only need to use these in most cases!
- **Constructors**
  - You can think of these as methods (functions) that create an instance of your class, given a list of parameters.
  - They are declared like typical methods, with the method name 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**.
  - **Instance methods** are methods are invoked (called) from concrete objects. For example, if `st` is of type `Student` and `sayHi()` is a method for Students, you can invoke it by doing `st.sayHi()`.

Important Keywords

- **static**
  - In contrast to instance fields/methods, the **`static`** keyword declares that a field or method *belongs to a class itself*, rather than being tied to concrete instances of the class.
  - You can think of these as being "global" across all instances of a specific class.
  - Can you think of an instance where a static field or method would make sense?
- **this**
  - When used in an instance method, **`this`** refers to the actual instance in which the method was invoked from.
  - This is useful for when you need to make reference to a class field that shares the same name as a local variable (e.g. `name` stored in a class, and `name` passed into a method). Without `this`, Java will prioritise the variable that is **local** in scope.
- **new**
  - You can think of this as the Java equivalent of `malloc` (from C). 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 to read 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
  - Write a class 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, defining constructors