COMP2511

Tutorial 3

Last Week's Tutorial

- Inheritance
- Method Overriding

This Week's Tutorial

- Testing, Debugging, Exceptions
- Domain Modelling, UML Diagrams
- Design by Contract

Notices

- Assignment-i has been released on GitLab, and is due Week 5 Friday.
  - Make sure that you give yourself ample time to plan out your the design for the assignment before jumping straight into coding!

Testing - JUnit

- JUnit is the main testing framework we use in this course, similar to what Jest is to COMP1531.
- Each testing suite is separated into its own class, and each of the methods defined in that class will be the individual unit tests.
  - For example, if you were making a test suite to test Task 1 of Assignment 1, this could be put in a file called `Task1Tests.java`, and a method in the class could be `testCreateTrain()`.
- Unit tests are identified using the `@Test` tag over method names, like the `@Override` tag for method overriding.
  - This tag is required to identify a block of code as a test.
- In each unit test, conditions are checked using **assertions**, such as `assertEquals()`, `assertTrue()` and `assertThrows()`. If any assertion in a unit test fails, the whole test fails.

Testing - JUnit Example (from Assignment 1)

```java
public class TaskAExampleTests {
    @Test
    public void testCreateStations() {
        TrainsController controller = new TrainsController();

        controller.createStation("station1", "DepotStation", 1.0, 1.0);
        assertEquals(new StationInfoResponse("station1", "DepotStation", new Position(1.0, 1.0)),
                controller.getStationInfo("station1"));

        controller.createStation("station2", "CargoStation", 10.0, 10.0);
        assertEquals(new StationInfoResponse("station2", "CargoStation", new Position(10.0, 10.0)),
                controller.getStationInfo("station2"));

        controller.createStation("station3", "PassengerStation", 19.6, 15);
        assertEquals(new StationInfoResponse("station3", "PassengerStation", new Position(19.6, 15)),
                controller.getStationInfo("station3"));

        assertListAreEqualIgnoringOrder(List.of("station1", "station2", "station3"), controller.listStationIds());
    }
```

Testing - IDE Debugging

- At this point in the course, your main method of debugging has most likely been through using print statements.
- We have access to debugging techniques that allow us to step through the execution of the Java program (similar to GDB for C).
- A **breakpoint** is a specific spot in your code where you would like to intentionally pause execution.
    - In VS Code, they can be set on the *glyph margin*, which is to the left of each of the line numbers.
    - By running your code in **debug mode**, you can stop execution at breakpoints, allowing you to inspect the state of variables and step through lines of code to analyse the control flow of the program (if statements, while loops...).

Exceptions

- An **exception** is an event that occurs when an error is encountered during the execution of a program.
  - When the error occurs, an *exception object* containing information about the error is created and **thrown** to the runtime system for it to handle.
  - In simpler terms, think of exceptions as flags that something has gone wrong, such as division by zero or trying to dereference a null object.
- In Java, exceptions are classified into two categories:
  - **Checked exceptions**, which need to be included in a method's signature if it can potentially be thrown and caught in a *try-catch* block (eg. `IOException`).
  - **Unchecked exceptions**, which do not need to be included in a method's signature or caught in a try-catch block (eg. `ArithmeticException`, `IllegalArgumentException`).

Live Example: Debugging (Pt. 1)

- The Wondrous Sequence is generated by the simple rule:
  - If the current term is even, the next term is half the current term.
  - If the current term is odd, the next term is three times the current term, plus 1.
  - For example, the sequence generated by starting with 3 is [3, 10, 5, 16, 8, 4, 2, 1].
  - If the starting term is 1, then an empty list is returned.
- Inside **src/wondrous** there is an implementation of this algorithm, and a single test for the function which currently fails.
- Use the debug tools and the given algorithm to determine why the test is failing, and fix the bug.

Live Example: Debugging (Pt. 2)

- Are we confident that the implementation is completely bug-free now? Create a new unit test for a potentially untested case.
- Alongside the above, modify the method such that if `start` is less than 1, an `IllegalArgumentException` is thrown. Write a corresponding test case for this.
    - In many cases when we throw an exception we need to update the method signature and existing tests but here we don't - why is this?

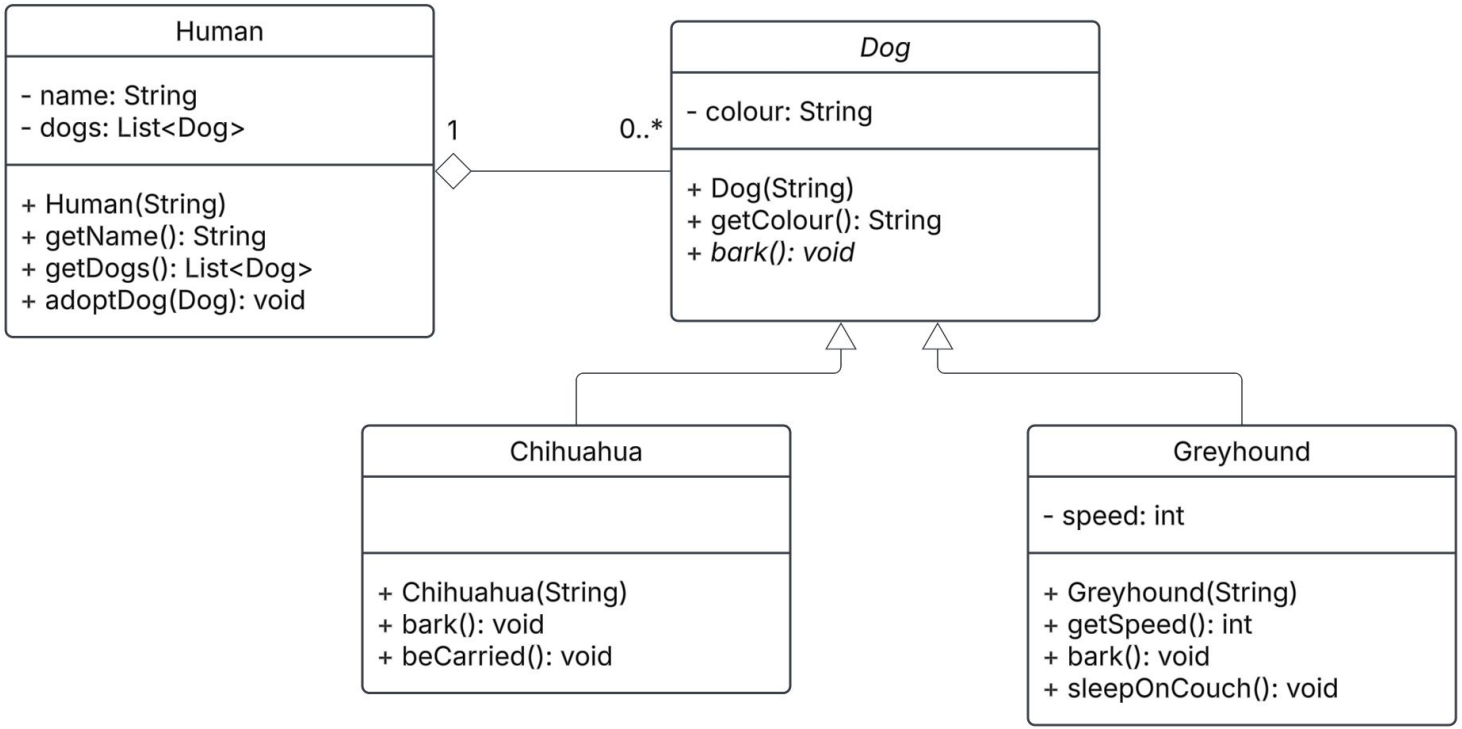
UML Diagrams

- **UML** (**U**nified **M**odeling **L**anguage) diagrams are commonly used to visually demonstrate the structure of a system in terms of its components (classes) and the relationships between each of the components.
- From a UML diagram, we should be able to tell a lot about the system, such as:
  - which classes are present in the system
  - what fields and methods each of the classes have
  - which classes are subclasses of other classes (**is-a** relationships)
  - which classes contain instances of other classes (**has-a** relationships) and the **cardinality** of the relationship (e.g. 1 car has 1 steering wheel, 1 class has many students, many students have 1 teacher)

Relationships in UML Diagrams

- For inheritance/implementation, the arrow points to the parent class being inherited/interface being implemented.
- For ‘has-a’ relationships, the diamond is on the side of the container.
- There are two types of ‘has-a’ relationships:
  - **Aggregation**: A contains B. B *can* exist independently of A (i.e. the existence of B is not directly tied to A). For example, a classroom has students.
  - **Composition**: A contains B. B *cannot* exist independently of A (i.e. the existence of B is directly tied to A). For example, a person has thoughts.

UML Diagram Example



Live Example: Domain Modelling

- A Car has one or more engines and a producer. The producer is a manufacturing company who has a brand name. Engines are produced by a manufacturer and have a max speed. There are only two types of engines within UNSW's cars:
  - Thermal Engines, which have a default max speed of 114, although they can be produced with a different max speed, and the max speed can change to any value between 100 and 250.
  - Electrical Engines, which have a default max speed of 180. This is the speed at which they are produced, and the max speed can change to any value that is divisible by 6.
- Cars are able to drive to a particular location x, y.

Additionally, we want to model the behaviour of a time travelling car, as well as time travelling for any vehicle. A ‘time travelling vehicle’ stays in the same location but travels to a LocalDateTime.

Create a UML diagram which models the domain.

Defensive Programming vs. Design by Contract

- **Defensive Programming** is a programming approach that accounts for unforeseen circumstances, and tries to ensure that programs will behave predictably despite unexpected inputs or user actions.
  - For example, checking for null inputs in every method before any logic is performed.
  - This approach makes programs more secure and robust, but could potentially obscure the main logic of the program and include lots of redundant code.
- **Design by Contract** is an approach where code is written around a set of clear and well-defined specifications, which *when adhered to* guarantees correct and expected behaviour.
  - For example, you would not need to perform any null checks if it is explicitly stated in the requirements (contract) that null will not be given as an input.
  - This can make a program's logic clearer, but makes it vulnerable when this contract is not abided to.

Design by Contract

- Terminology: In the context of a method,
  - a **precondition** is a condition on the **input** that the method will perform as expected.
  - a **postcondition** is a guarantee on what the method will output/perform, *given that the precondition holds*.
  - an **invariant** is a property of the system that is guaranteed to be maintained once the method has been performed.
    - this property can change *during* the method's execution, the importance is that it should hold once the method is completed.
- Essentially, if someone defines a precondition for a specific method and it isn't satisfied when the method is executed, then they can no longer make any assumptions about the correctness of the method.

The Liskov Substitution Principle

- The **Liskov Substitution Principle** (LSP) is an important concept in OOP, which states that everywhere an instance of a superclass is used, swapping it out for an instance of its subclass should not affect the correctness of the program.
- Contracts are inherited from superclasses to their subclasses.
- This means that:
  - **preconditions** in the subclass must either stay the same or be **loosened**,
  - **postconditions** in the superclass must either stay the same or be **tightened**.

Design by Contract and LSP

- Suppose B is a subclass of A. Is the inheritance in the following examples valid?
  - A has a method with a precondition that it accepts integers between 0-50. B overrides the method to accept any integer between 0-100.
    - This is valid. Any input that satisfies the preconditions of A's method also satisfies the preconditions of B's method.
    - The other way around would be invalid!
  - A has a method with a postcondition that it returns a sorted list of integers. B overrides the method to return an unsorted list.
    - This is invalid, as an unsorted list is not a sorted list so the postcondition is not satisfied by B's method.
    - The other way around would be valid!

Live Example: Design by Contract

- In **src/people**, there are a few classes which represent the people at a university.
  - Briefly discuss the preconditions and postconditions of the constructors, getters and setters in Person.java.
  - Fill in the preconditions and postconditions for setSalary in Person.java.
  - Discuss the validity of the subclasses of Person, and why they are/aren't valid subclasses.
  - Fix any issues identified.