COMP2511

Tutorial 3

Last Week's Tutorial

- Inheritance
  - Using existing classes to create new classes which (typically) have more functionality
- Method overriding
  - Specialising the behaviour of inherited methods

This Week’s Tutorial

- Testing
  - Writing JUnit tests
  - Using VSCode debugging tools
  - Exceptions
- Domain Modelling
  - Designing systems before writing any concrete code
  - Creating UML diagrams
- Extra stuff (if time permits)
  - Method overriding and static fields/methods (code review)
  - Method hiding (extension material)

Notices

- Assignment 1 has been released on GitLab.
    - You can use the `2511 dryrun ass1` command on CSE servers to check that your work compiles before submitting it.
- Please fill in the Assignment 2/3 Partner Preference form by **Week 4 Friday**, regardless of whether you have a specific partner in mind.

Assignment 1 Tips

- This is a new assignment, so it's possible that there are small issues that haven't been ironed out. If you are unsure of anything on the spec, make sure to ask on the forum or use the sample implementation!
- If you have not started yet, taking the time to read the spec is already a huge step forward.
    - 45% of your mark will be based on the quality of your design, which you can only start thinking about once you know what you have to do!
    - Thinking about your design is something you can do while doing basically anything else; you do not have to be in front of a computer coding to do this!
    - I would heavily recommend having a read of the whole spec before creating sketches/drafts of your design, so that you can easily adapt your code as you move through the tasks (i.e. think of the longer term consequences of your design choices).

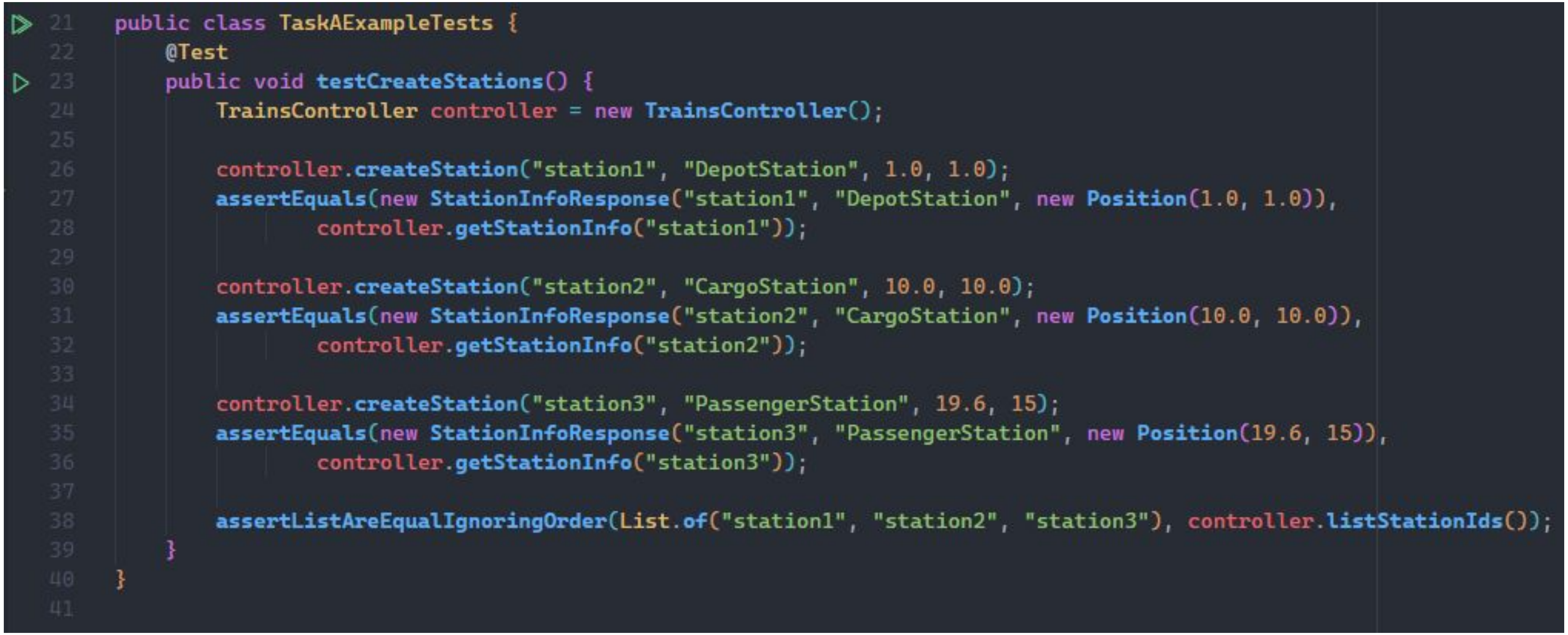
Assignment 1 Tips

- Do lots of testing (which we're covering today)!
  - It's likely that you'll have to scrap certain ideas and change lots of things around as you create your design and complete the assignment. This is a very natural part of the process!
  - While refactoring, you can only be confident that you haven't messed up your logic if you are consistently testing your code and have a good test suite.
  - While only a small portion (5%) of the marks are awarded for testing, it's very likely that the effort you put into your tests will have a correlation to the amount of autotests you pass (i.e. your automark).
- Remember that with design, there's not always just one correct answer.
  - I will be looking to see that you have outlined and justified your specific design choices in your blog, rather than marking based off of a checklist.

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 would be put in a file called `Task1Tests.java`, and a method could be `testCreateTrain()`.
- Unit tests are identified using the @Test tag over method names, like the @Override tag for method overriding.
  - This tag allows your extensions to recognise the method as a test and run it by clicking a green button (looks like a Play button, or a tick if the test has passed before).
- In each unit test, you can perform logic similar to how you would write a main function - conditions are checked using **assertions**, like `assertEquals()`, `assertTrue()` and `assertThrows()`. If any assertion in a unit test fails, the whole test fails.

Testing - JUnit Example (from Assignment 1)

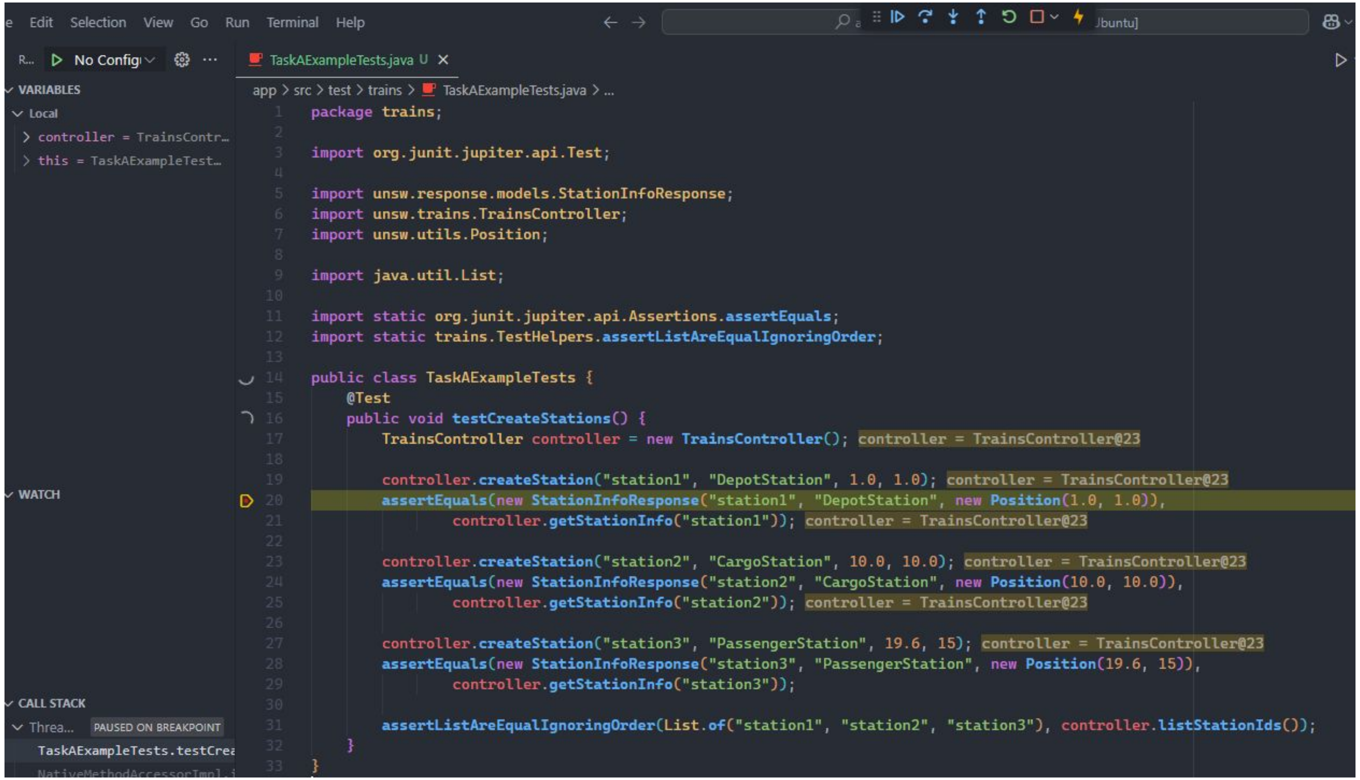


```java
public class TaskAExampleTests {
    @Test
    public void testCreateStations() {
        TrainsController controller = new TrainsController();

        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 of the course, your main method of debugging has most likely been through using print statements.
- We have access to some more advanced debugging techniques that allow us to step through the execution of the program, like 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 inspect the state of variables and the call stack at that point of time, and also step through the program to analyse the control flow of the program.

Testing - IDE Debugging Example



Edit
Selection
View
Go
Run
Terminal
Help
No Config
TaskAExampleTests.java U
app > src > test > trains > TaskAExampleTests.java > ...
VARIABLES
Local
controller = TrainsContr...
this = TaskAExampleTest...
WATCH
CALL STACK
Threa... PAUSED ON BREAKPOINT
TaskAExampleTests.testCrea
NativeMethodAccessorImpl.i

```java
package trains;

import org.junit.jupiter.api.Test;

import unsw.response.models.StationInfoResponse;
import unsw.trains.TrainsController;
import unsw.utils.Position;

import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static trains.TestHelpers.assertListAreEqualIgnoringOrder;

public class TaskAExampleTests {
    @Test
    public void testCreateStations() {
        TrainsController controller = new TrainsController(); controller = TrainsController@23

        controller.createStation("station1", "DepotStation", 1.0, 1.0); controller = TrainsController@23
        assertEquals(new StationInfoResponse("station1", "DepotStation", new Position(1.0, 1.0)),
                controller.getStationInfo("station1")); controller = TrainsController@23

        controller.createStation("station2", "CargoStation", 10.0, 10.0); controller = TrainsController@23
        assertEquals(new StationInfoResponse("station2", "CargoStation", new Position(10.0, 10.0)),
                controller.getStationInfo("station2")); controller = TrainsController@23

        controller.createStation("station3", "PassengerStation", 19.6, 15); controller = TrainsController@23
        assertEquals(new StationInfoResponse("station3", "PassengerStation", new Position(19.6, 15)),
                controller.getStationInfo("station3"));

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

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, you can think of exceptions as flags that something has gone wrong, such as division by zero or trying to dereference null.
- 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 from FileWriter).
  - **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.

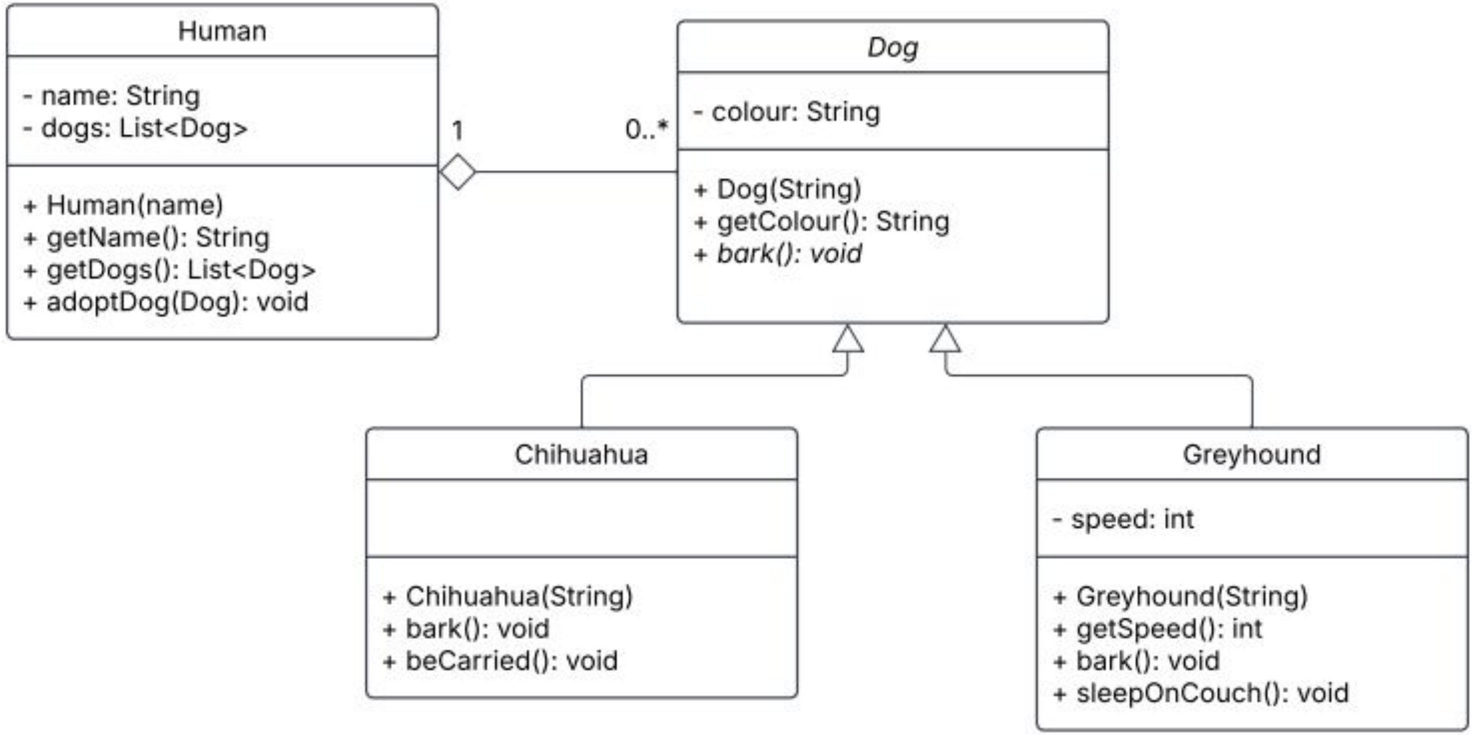
Live Example - Debugging (Pt. 2)

- Are we confident that the implementation is completely bug-free now? Create a new unit test for a case that could potentially fail.
- Alongside the above, modify the method such that if `start` is less than 1, an `IllegalArgumentException` is thrown. Write a corresponding test for this inside WondrousTest.
  - 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** (Unified Modeling Language) diagrams are commonly used to visually demonstrate the structure of a system in terms of its components (classes) and the relationship between each of its 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
  - which 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)

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 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.

Since UNSW is a world-leader in technology innovation, they want you to be able to model the behaviour of Time Travelling for any vehicle, and to model a time travelling car. A vehicle that travels in time stays in the same location but travels to a LocalDateTime.

Create a UML diagram which models the domain.

Extra Material

Code Example - Method Overriding and Static Fields

- Review the UniStudent and Point classes in **src/review**. What will each class’ main methods output?
    - [KEY TAKEAWAYS] Understanding when an implementation of a non-static method is determined (dynamic dispatch), how a static field works

[EXTENSION] Method Hiding

- Can static methods be overridden?
  - No, since static methods belong to specific classes.
  - Method overriding is associated with re-defining the functionality of a method defined in an earlier class, but this makes no sense if those methods belong to the earlier class itself, rather than concrete instances…
- Can static methods be inherited?
  - Perhaps surprisingly, yes!
  - This is because you can still call a static method from a concrete instance of the class. This is bad practice however, so you shouldn’t really run into this in the wild.
- With this in mind, is there something *similar* to method overriding for static methods?

[EXTENSION] Method Hiding

- If a subclass defines a static method with the **same method signature** as a static method in the superclass, then the method in the subclass *hides* the one in the superclass.
- If a concrete instance of a class calls a static method, the actual method called depends on its **static type**, or **type of reference**.
    - This is distinct to method *overriding*, where the actual method called is determined by the type of the actual object, and is determined dynamically (i.e. at run-time, rather than at compile-time through static checks). This mechanism is known as **dynamic dispatch**, and is used for all non-static methods in Java.

[EXTENSION] Code Example - Method Hiding

- Review the UniStudent class in **src/hiding**. What will the main method output? How does this differ to the functionality of UniStudent in **src/review**?
    - [KEY TAKEAWAYS] Understanding when an implementation of a static method is determined (static dispatch)