



# THE UNIVERSITY OF QUEENSLAND AUSTRALIA

School of ITEE  
CSSE2002/7023 — Semester 2, 2022  
Assignment 2  
**Due: 7<sup>th</sup> November 2022 16:00AEST**  
Revision: 2.1.0

## Abstract

The goal of this assignment is to implement and test a set of classes and interfaces<sup>1</sup>. These classes will be distinct from the classes implemented in the first assignment but will cover similar topics.

**Language requirements:** Java version 17, JUnit 4.

## Preamble

All work on this assignment is to be your own individual work. As detailed in Lecture 1, code supplied by course staff (from this semester) is acceptable, but there are no other exceptions. You are expected to be familiar with “What not to do” from Lecture 1 and <https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism>. If you have questions about what is acceptable, please ask course staff.

Programming for Engineers

Please carefully read the Appendix A document. It outlines critical mistakes which you should circumvent in order to avoid losing marks. This is being heavily emphasised here because these are critical mistakes which must be avoided. If at any point you are even slightly unsure, please check as soon as possible with course staff!

All times are given in *Australian Eastern Standard Time*. It is your responsibility to ensure that you adhere to this time-zone for all assignment related matters. Please bear this in mind, especially if you are enrolled in the *External* offering and may be located in a different time-zone.

## Introduction

In this assignment you will be developing a simple simulation of a busy fast food pizza business, building on a code model system centred around Inheritance and Interfaces.

You are working in collaboration with a fictional second team; who are developing the GUI.

For assignment 2, you will develop the code based on the supplied JavaDoc and will focus on splitting your development strategy, so that you are working on one part of a bigger project. Your task will rest on ensuring that the code you write will function correctly when it is connected to the already in development GUI system. You will fixate on developing the hierarchy of pizza creation and the infrastructure surrounding the creation of various types of pizzas. The pizzas will be ordered in three ways:

1. **Default Pizza** - A plain pizza with a selection of sauces, crusts and cheeses.
2. **Custom Pizza** - A pizza designed by using available topping ingredients; includes all *plain pizza* attributes

---

<sup>1</sup>From now on, classes and interfaces will be shortened to simply “classes”

3. **Menu Pizza** - A pizza ordered from off an external loaded menu; includes all *plain pizza* attributes

Your code will be designed to connect with an external codebase which will remain unknown to you. Your completed code will accept an order, process the order and return a completed order to a specified input and output specification, detailed in the Javadoc.

## Specification update

You are required to start working on the project right away. Nevertheless, in one weeks time, an update to your project specification will be uploaded. This will be provided on **Friday the 14<sup>th</sup> October, 2022**. In this addendum to the specification, additional works will be added to the current specification tasks. This will reflect the real pressures and target outcomes experienced in real-life projects. A Blackboard notification will inform you of this important addition/change to the specification when it is released.

## Story

Based on your first assignment, your reputation as a software engineering force is growing exponentially. With this, your successes have been noted by many a noble and notable business establishment and none more famous than the world renowned *Dr. Java's Pizza Parlour*.

Dr. Java Jewowski (*known colloquially and internationally as DJ*), the founding father of Dr. Java's Pizza Parlours has noted your talents and admires your software engineering prowess. DJ is keen to commission your famous skill-base and has requested your help. While it is recognised that you are newly developing company, DJ's wants to try you out with an initial starting project.

The pizza ordering system is needed to assist DJ's pizza company, with their migration to an all digital system. You will be working towards providing the input mechanism for the future GUI, that is also in parallel development.

Your task is to utilise the JavaDoc specification provided to produce the application required by Dr Java's Pizza Parlour. You are also required to produce adequate and appropriate JUnit testing, to enable the project to be successfully tested by the receiving GUI development team.

## Supplied Material

- This task sheet.
- Code specification document (Javadoc).<sup>2</sup>
- Gradescope, a website where you will submit your assignment.<sup>3</sup>
- In this assignment you are expected to create the entire project, without a template. No framework is supplied and you are expected to develop the packages and classes from the specification provided. You must be meticulous with your naming, to ensure that you spell and character case check every class package, name and method. Any deviation could cost you marks, as the automatic testing will fail. This represents the real life expectation when interfacing with ready made code-bases. Take extra care to avoid the critical mistakes described in Appendix A. Ensure that each of your files are:
  - is in the correct directory (do not change this!)
  - has the correct package declaration at the top of the file (do not change this!)

---

<sup>2</sup>Detailed in the **Javadoc** section

<sup>3</sup>Detailed in the **Submission** section

- has the correct public class or public interface declaration. Note that you may still need to make classes abstract, extend classes, implement interfaces etc., as detailed in the Javadoc specification.

As the first step in the assignment (after reading through the specifications) you should create the packages described and populate them with empty classes, which are named precisely as they are described.

## Javadoc

Code specifications are an important tool for developing code in collaboration with other people. Although assignments in this course are individual, they still aim to prepare you for writing code to a strict specification by providing a specification document (in Java, this is called Javadoc).

You will need to implement the specification precisely as it is described in the specification document.

The Javadoc can be viewed in either of the two following ways:

1. Open <https://csse2002.uqcloud.net/assignment/2/> in your web browser. Note that this will only be the most recent version of the Javadoc.
2. Navigate to the relevant assignment folder under **Assessment** on Blackboard and you will be able to download the Javadoc .zip file containing HTML documentation. Unzip the bundle somewhere, and open `doc/index.html` with your web browser.

## Tasks

1. Fully implement each of the classes and interfaces described in the Javadoc.
2. Write JUnit 4 tests for all the methods in the following classes:
  - **CustomPizza**  
(in a class called `CustomPizzaTest`)
  - **Topping**  
(in a class called `ToppingTest`)

## Marking

The 100 marks available for the assignment will be divided as follows:

Symbol	Marks	Marked	Description
<i>FT</i>	50	Electronically	Functionality according to the specification
<i>CF</i>	5	Electronically	Conformance to the specification
<i>SL</i>	10	Electronically	Code style: Structure and layout
<i>CR</i>	15	By course staff	Code style: Design review
<i>JU</i>	20	Electronically	Whether JUnit tests identify and distinguish between correct and incorrect implementations

The overall assignment mark will be  $A_2 = FT + CF + SL + CR + JU$  with the following adjustments:

1. If *FT* is 0, then the manual code style review will not be marked. *CR* will be automatically 0.
2. If *SL* is 0, then the manual code style review will not be marked. *CR* will be automatically 0.

3. If  $SL + CR > FT$ , then  $SL + CR$  will be capped at a maximum of  $FT$ .

- For example:  $FT = 17, CF = 5, SL = 7, CR = 12, J = 13$   
 $\Rightarrow A_2 = 17 + 5 + (7 + 12) + 13.$   
 $\Rightarrow A_2 = 17 + 5 + (19) + 13.$  Limitation will now be applied.  
 $\Rightarrow A_2 = 17 + 5 + (17) + 13.$

The reasoning here is to place emphasis on good quality functional code.

Well styled code that does not implement the required functionality is of no value in a project, consequently marks will not be given to well styled code that is not functional.

## Functionality Marking

The number of functionality marks given will be

$$FT = \frac{\text{Unit Tests passed}}{\text{Total number of Unit Tests}} \cdot 50$$

## Conformance

Conformance is marked starting with a mark of 5.

Every single occurrence of a conformance violation in your solution then results in a 1 mark deduction, down to a minimum of 0. Note that multiple conformance violations of the same type will each result in a 1 mark deduction.

Conformance violations include (but are not limited to):

- Placing files in incorrect directories.
- Incorrect package declarations at the top of files.
- Using modifiers on classes, methods and member variables that are different to those specified in the Javadoc. Modifiers include **private**, **protected**, **public**, **abstract**, **final**, and **static**. For example, declaring a method as public when it should be private.
- Adding extra public methods, constructors, member variables or classes that are not described in the Javadoc.
- Incorrect parameters and exceptions declared as thrown for constructors.
- Incorrect parameters, return type and exceptions declared as thrown for methods.
- Incorrect types of public fields.

## Code Style

### Code Structure and Layout

The Code Structure and Layout category is marked starting with a mark of 10.

Every single occurrence of a style violation in your solution, as detected by *Checkstyle* using the course-provided configuration<sup>4</sup>, results in a 0.5 mark deduction, down to a minimum of 0. Note that multiple style violations of the same type will each result in a 0.5 mark deduction.

Note: There is a plug-in available for *IntelliJ* which will highlight style violations in your code. Instructions for installing this plug-in are available in the Java Programming Style Guide on Blackboard (Learning Resources → Guides). If you correctly use the plug-in and follow the style requirements, it should be relatively straightforward to get high marks for this section.

---

<sup>4</sup>The latest version of the course *Checkstyle* configuration can be found at <http://csse2002.uqcloud.net/checkstyle.xml>. See the Style Guide for instructions.

## Code Review

Your assignment will be style marked with respect to the course style guide, located under Learning Resources → Guides. The marks are broadly divided as follows:

Metric	Marks Allocated
Commenting	5
Naming	4
Code Design	3
Programming Practices	3

*Note that style marking does involve some aesthetic judgement (and the marker's aesthetic judgement is final).*

Note that the plugin available for IntelliJ mentioned in the Code Structure and Layout section *cannot* tell you whether your code violates style guidelines for this section. You will need to manually check your code against the style guide.

The Code Review is marked starting with a mark of 0. Each section will make a judgment with penalties are then applied where applicable, to a maximum of 15.

Metric	How it is marked
Naming	<p>Misnamed variables</p> <p>e.g.</p> <ul style="list-style-type: none"> <li>→ Non-meaningful or one-letter names <ul style="list-style-type: none"> <li>• <code>String temp; // bad naming</code></li> <li>• <code>char a; // bad naming</code></li> <li>• <code>int myVar, var, myVariable; // all bad naming</code></li> </ul> </li> <li>→ Variable names using Hungarian notation <ul style="list-style-type: none"> <li>• <code>int roomInteger; // bad naming</code></li> <li>• <code>List&lt;Gate&gt; gateList; // bad naming ('gates' is better)</code></li> </ul> </li> </ul>
Commenting	<p>Javadoc comments lacking sufficient detail</p> <p>e.g.</p> <ul style="list-style-type: none"> <li>→ Insufficient detail or non-meaningful Javadoc comments on (any) classes</li> <li>→ Insufficient detail or non-meaningful Javadoc comments on (any) methods</li> <li>→ Insufficient detail or non-meaningful Javadoc comments on (any) constructors</li> <li>→ Insufficient detail or non-meaningful Javadoc comments on (any) class variables</li> <li>→ etc.</li> </ul> <p>Lack of inline comments, or comments not meaningful</p> <p>e.g.</p> <ul style="list-style-type: none"> <li>→ There needs to be sufficient comments which explain your code so that someone else can readily understand what is going. Someone should not need to guess or make assumptions.</li> <li>→ Lack of inline comments, or comments not meaningful in methods</li> <li>→ Lack of inline comments, or comments not meaningful in constructors</li> <li>→ Lack of inline comments, or comments not meaningful for variables</li> <li>→ etc.</li> </ul>
Code Design	<p>Code design issues</p> <p>e.g.</p> <ul style="list-style-type: none"> <li>→ Using class member variables where local variables could be used</li> <li>→ Duplicating sections of code instead of extracting into a private helper method</li> <li>→ Having functions that exceed 100 lines in length</li> <li>→ Class content is laid out in a way which is not straightforward to follow</li> <li>→ Methods are laid out in Classes or Interfaces in a way which is not straightforward to follow</li> <li>→ Method content is laid out in a way which is not straightforward to follow</li> <li>→ Variables are not placed in logical locations</li> </ul>
Programming Practices	<p>Programming Best Practices issues</p> <p>e.g.</p> <ul style="list-style-type: none"> <li>→ Using class member variables where local variables could be used</li> <li>→ Incorrect use of inheritance by duplicating code</li> <li>→ Incorrect exception handling</li> <li>→ Using magic numbers without explanatory comments <ul style="list-style-type: none"> <li>• <code>object.someMethod(50); // what does 50 mean? What is the unit/metric?</code></li> </ul> </li> </ul>

## JUnit Test Marking

See Appendix B for more details.

The JUnit tests that you provide in `CustomPizzaTest` and `ToppingTest` will be used to test both correct *and* incorrect implementations of the `CustomPizza` and `Topping` classes. Marks will be awarded for test sets which distinguish between correct and incorrect implementations<sup>5</sup>. A test

<sup>5</sup>And get them the right way around

class which passes every implementation (or fails every implementation) will likely get a low mark. Marks will be rewarded for tests which pass or fail correctly.

There will be some limitations on your tests:

1. If your tests take more than 20 seconds to run, or
2. If your tests consume more memory than is reasonable or are otherwise malicious,

then your tests will be stopped and a mark of zero given. These limits are very generous (e.g. your tests should not take anywhere near 20 seconds to run).

## Electronic Marking

The electronic aspects of the marking will be carried out in a Linux environment. The environment will not be running Windows, and neither IntelliJ nor Eclipse (or any other IDE) will be involved. OpenJDK 17 will be used to compile and execute your code and tests.

It is critical that your code compiles.

If your submission does not compile, **you will receive zero** for Functionality (FT).

## Submission

### How/Where to Submit

Submission is via Gradescope.

Instructions for submitting to Gradescope will be made available on Blackboard (under Assessment → Assignment 2) within one week after the assignment specification has been released. You will not be able to submit your assignment before then.

You must ensure that you have submitted your code to Gradescope *before* the submission deadline. Code that is submitted after the deadline will **not** be marked (1 nanosecond late is still late).

You may submit your assignment to Gradescope as many times as you wish before the due date, however only your **last submission made before the due date will be marked**.

### What to Submit

Your submission should have the following internal structure:

`src/`    folders (packages) and `.java` files for classes described in the Javadoc  
`test/`   folders (packages) and `.java` files for the JUnit test classes

A complete submission would look like:

```
src/io/MenuLoader.java

src/exceptions/TooManyToppingsException.java
src/exceptions/PizzaFormatException.java

src/menu/Order.java
src/menu/Menu.java
src/menu/CustomerOrder.java
src/menu/MenuItem.java

src/gui/Display.java

src/main/Launcher.java

src/pizza/MenuPizza.java
src/pizza/Pizza.java
src/pizza/CustomPizza.java

src/pizza/ingredients/Sauces.java
src/pizza/ingredients/Bases.java
src/pizza/ingredients/Cheeses.java
src/pizza/ingredients/Topping.java

test/pizza/ingredients/ToppingTest.java (must be test directory, not src!)
test/pizza/CustomPizzaTest.java (must be test directory, not src!)
```

Ensure that your classes and interfaces correctly declare the package they are within. For example, `MenuPizza.java` should declare `package pizza;`.

**Do not** submit any other files (e.g. no `.class` files).

Note that `CustomPizzaTest` and `ToppingTest` will be compiled individually against a sample solution without the rest of your test files.

### Provided set of unit tests

A small number of the unit tests (about 10-20%) used for assessing Functionality (FT) (not conformance, style, or JUnit tests) will be provided in Gradescope prior to the submission deadline, which you will be able to test your submission against. In addition, a small number of the JUnit faulty solutions used for assessing JUnit will be provided in Gradescope prior to the submission deadline.

The purpose of this is to provide you with an opportunity to receive feedback on whether the basic functionality of your classes and tests is correct or not. Passing all the provided unit tests does **not** guarantee that you will pass all of the full set of unit tests used for functionality marking.

Instructions about the provided set of unit tests will be made available on Blackboard (under Assessment → Assignment 2) one week after the assignment specification has been released. Instructions will not be provided before then.

## Late Submission

Assignments submitted after the submission deadline of 16:00 on 7<sup>th</sup> November 2022 (by any amount of time), will receive a mark of zero unless an extension is granted as outlined in the



Electronic Course Profile — see the Electronic Course Profile for details.

Do not wait until the last minute to submit the final version of your assignment. A submission that starts before 16:00 but finishes after 16:00 will not be marked. Exceptions cannot be made for individual students, as this would not be fair to all other students.

## Assignment Extensions

All requests for extensions must be made via my.UQ as outlined in section 5.3 of the respective Electronic Course Profile. Please not directly email the course coordinator seeking an extension (you will be redirected to my.UQ).

## Remark Requests

To submit a remark of this assignment please follow the information presented here:

<https://my.uq.edu.au/information-and-services/manage-my-program/exams-and-assessment/querying-result>.

## Revisions

If it becomes necessary to correct or clarify the task sheet or Javadoc, a new version will be issued and an announcement will be made on the Blackboard course site.

## Appendix A:

### Critical Mistakes which can cause loss in marks.

#### THINGS YOU NEED TO AVOID

This is being heavily emphasised here because these are critical mistakes which must be avoided.

The way assignments are marked has been heavily revised this semester to address many of these issues where possible, but there are still issues which can only be avoided by making sure the specification is followed correctly.

Code may run fine locally on your own computer in IntelliJ, but it is required that it also builds and runs correctly when it is marked with the electronic marking tool in Gradescope. Your solution needs to conform to the specification for this to occur.

Correctly reading specification requirements is a key objective for the course. Ensure that you read all components including headings and footers.

- Files must be in the correct directories (*exactly*) as specified by the Javadoc. If files are in incorrect directories (*even slightly wrong*), you may lose marks for functionality in these files because the implementation does not conform to the specification.
- Files must have the exact correct package declaration at the top of the file. If files have incorrect package declarations (even slightly wrong), you may lose marks for functionality in these files because the implementation does not conform to the specification.
- You must implement the public and protected members exactly as described in the supplied documentation (*no extra public/protected members or classes*). Creating public or protected data members in a class when it is not required will result in loss of marks, because the implementation does not conform to the specification.

- Private members may be added at your own discretion.
- Never import the `org.junit.jupiter.api` package. This is from JUnit 5. This will automatically cause the marks for the JUnit section to be 0 because JUnit 5 functionality is not supported.
- Do NOT use any version of Java newer than 17 when writing your solution! If you accidentally use Java features which are only present in a version newer than 17, then your submission may fail to compile when marked. This will automatically cause the marks for associated files with this functionality to be 0.

## Appendix B:

### How your JUnit unit tests are marked.

The JUnit tests you write for a class (e.g. `CustomPizzaTest.java`) are evaluated by checking whether they can distinguish between a:

**correct** implementation of the respective class  
e.g. `CustomPizza.java`, made by the teaching staff, and

**incorrect** implementations of the respective class  
*“deliberately made (sabotaged) by the teaching staff”*.

First, we run your unit tests (e.g. `CustomPizzaTest.java`) against the correct implementation of the respective classes (e.g. `CustomPizza.java`).

We look at how many unit tests you have, and how many have passed. Let us imagine that you have 7 unit tests in `CustomPizzaTest.java` and 4 unit tests in `ToppingTest.java`, and they all pass (i.e. none result in `Assert.fail()` in JUnit4).

We will then run your unit tests in both classes (`CustomPizzaTest.java`, `ToppingTest.java`) against an incorrect implementation of the respective class (e.g. `CustomPizza.java`). For example, the `foo()` method in the `CustomPizza.java` file is incorrect.

We then look at how many of your unit tests pass.

`ToppingTest.java` should still pass 4 unit tests.  
However, we would expect that `CustomPizzaTest.java` would pass **fewer than** 7 unit tests.

If this is the case, we know that your unit tests can identify that there is a problem with this specific implementation of `CustomPizza.java`.

This would get you one identified faulty implementation towards your JUnit mark.

The total marks you receive for JUnit are the correct number of identified faulty implementations, out of the total number of faulty implementations which the teaching staff create.

For example, if your unit tests identified 60% of the faulty implementations, you would receive a mark of 60% of 20 → 12.00012/20.

## Appendix C:

### Revision History: Changes in v2.1.0

## **Menu.java**

- Added clear method to Menu

## **MenuLoader.java**

- MenuLoader::load() will not be tested.

## **Order.java**

- Added **Total printout** to the toString method when a discount is not applied.
- Added JavaDoc for default values which were missing.

## **CustomPizza.java**

- Added IllegalArgumentException for **null checks**.

## **MenuPizza.java**

- Added IllegalArgumentException for **null checks**.

## **Pizza.java**

- Added IllegalArgumentException for **null checks**.
- setName throws IllegalArgumentException if name.isBlank() is true.
- Order of toppings does not impact equality.

## **Topping.java**

- Added IllegalArgumentException for name.isBlank() check.
- **Added resetToppings()**.

Work hard and good luck.