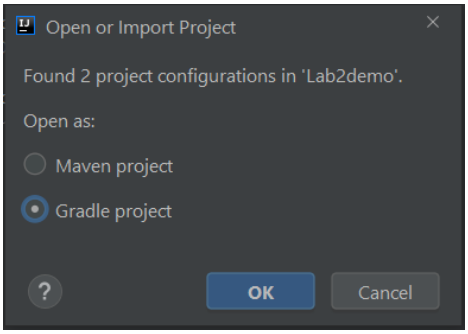
SENG275 – Lab 2
Due Wednesday Feb 3, 5:00 pm

Technical note: You now have a third repository in your gitlab.csc.uvic.ca account, named Gilded_Rose. When you first open the project in IntelliJ, you'll be asked if you want to open it as a Maven project or a Gradle project; choose 'Gradle':



Welcome to Lab 2. Throughout this document, some sections will be in Red, and others will be in **bold**. The sections in Red will be marked. The questions in **bold** will not be graded. However, you should give these questions some thought, as these are the sorts of questions that could appear on a quiz or midterm. This lab is worth 4% of your final grade.

Quick summary of what you need to do:

1. Complete the tests specified on Page 7 of this lab document, in GuildedRoseTest.java. (worth 3%)
2. Refactor GildedRose.java – improve its code quality, while continuing to test to make sure your changes haven't broken the program. (worth 1%)
3. Push your changes to GildedRose.java and GildedRoseTest.java back to your repository.

You're done!

Refactoring

It's common to encounter code that is working properly, but is poorly written. That is, the code is *functionally correct* (it provides the correct outputs for given inputs), but is of low quality – perhaps it is written using vague variable or method names, or maybe it's composed of one giant function with hundreds of lines, or perhaps it has dozens of nested conditional blocks, and as a result it's impossible to reason clearly about what the program is doing.

Refactoring is the process of taking this working (but poor-quality) code and modifying it so that it is much clearer and readable from the programmer's perspective, *without* accidently breaking it in the process. Test suites can give us confidence that the changes we've made to increase the quality of the code have not accidently introduced a *regression* – a bug where no bug existed before.

Suppose a change to working code caused a regression? Would this be an example of a failure, a fault or an error? Could it be all three?

Gilded Rose

We'll take a break from JPacman this week to try writing some tests and doing some refactoring on a different code base – the "Gilded Rose Kata". A 'programming kata' is an exercise intended to confront the programmer with a series of difficult decisions, and to build experience, good habits and confidence through repetition (the name comes from physical exercises which serve the same role in martial arts). There is no one correct way to complete a programming kata. Rather the hope is that the programmer will consider different possible approaches, and be able to justify their decisions. As a result, katas such as these are not only used by programmers to strengthen their skills, but also by hiring committees when evaluating prospective software engineering hires (usually as a take-home project).

The goal for the lab is to write tests in the GildedRoseTest.java class to confirm that the program functions correctly according to the rules included in the GildedRoseRequirements.txt document included in the repository, which is reproduced for your reference below. Once these tests are written, our further goal is to refactor the extremely-poorly written (but functionally correct) GildedRose.java class, to substantially improve its quality, and to ensure that the tests we have written continue to pass.

As you work your way through the exercises, keep in mind that just as exhaustive testing is impossible (and therefore there's no way to really tell that you've written 'enough' tests), refactoring is a never-ending process as well. There's always something that could be

improved, tweaked, abstracted and polished; identifying the point of diminishing returns is just as important when refactoring as when testing.

GildedRoseRequirements.txt:

Hi and welcome to team Gilded Rose. As you know, we are a small inn with a prime location in a prominent city ran by a friendly innkeeper named Allison. We also buy and sell only the finest goods. Unfortunately, our goods are constantly degrading in quality as they approach their sell by date. We have a system in place that updates our inventory for us. It was developed by a no-nonsense type named Leeroy, who has moved on to new adventures. Your task is to add the new feature to our system so that we can begin selling a new category of items. First an introduction to our system:

- All items have a SellIn value which denotes the number of days we have to sell the item
- All items have a Quality value which denotes how valuable the item is
- At the end of each day our system lowers both values for every item

Pretty simple, right? Well this is where it gets interesting:

- Once the sell by date has passed, Quality degrades twice as fast
- The Quality of an item is never negative
- "Aged Brie" actually increases in Quality the older it gets
- The Quality of an item is never more than 50
- "Sulfuras", being a legendary item, never has to be sold or decreases in Quality
- "Backstage passes", like aged brie, increases in Quality as its SellIn value approaches; Quality increases by 2 when there are 10 days or less. It increases by 3 when there are 5 days or less but Quality drops to 0 after the concert

We have recently signed a supplier of conjured items. This requires an update to our system:

- "Conjured" items degrade in Quality twice as fast as normal items

Feel free to make any changes to the UpdateQuality method and add any new code as long as everything still works correctly. However, do not alter the Item class or Items property as those belong to the goblin in the corner who will insta-rage and one-shot you as he doesn't believe in shared code ownership (you can make the UpdateQuality method and Items property static if you like, we'll cover for you).

Just for clarification, an item can never have its Quality increase above 50, however "Sulfuras" is a legendary item and as such its Quality is 80 and it never alters.

We're not going to worry about "Conjoured" items for this lab – ignore that part.

What?

Ok, the specification isn't written in a very rigorous or helpful style. Let's try to make things clearer:

- There's a fictional store, represented by the GildedRose class.
- The store can have an unlimited number of Items.

- An Item can be given a name, and each one is given a quality value and a SellIn value.
- Every call to updateQuality() means one day passes.
- If an item's SellIn number goes negative, that item is 'expired'.

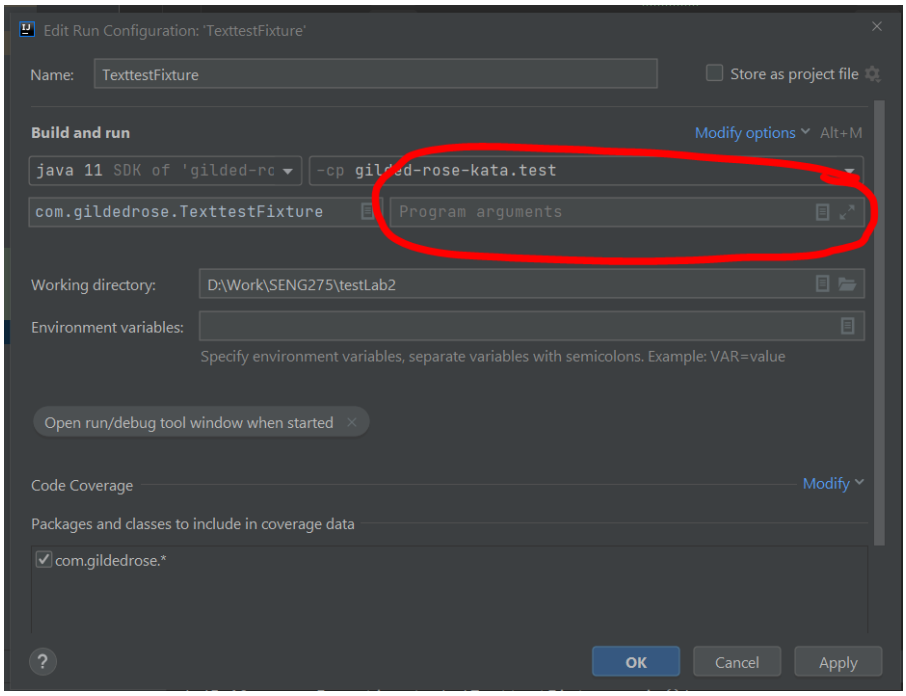
- There are four different types of items depending on their names – 'Aged Brie', 'Sulfruas, Hand of Ragnaros', 'Backstage passes to a TAFKAL80ETC concert', and ordinary items whose name is not one of the other three types.
- Ordinary items lose one from their SellIn value each day. Their quantity can never be greater than 50 or less than 0, and they lose 1 quality a day until they're expired, and then lose two quality a day.
- The other types of items have one or more exceptions to the rules for ordinary items.

- We are not allowed to alter the Item class in any way, or the items variable or the updateQuality() method in the GildedRose class. We are not allowed to add classes. We can only modify GildedRose.java or GildedRoseTest.

TesttextFixture.java

This is still probably kind of confusing, so take a look at TesttextFixture. This is a program that creates a store, adds nine items to it, and then runs for a certain number of 'days', calling updateQuality() each 'day', and printing out the name, SellIn and Quality values for each item each day. Run the program and see if you can see how the above rules are followed by the GildedRose implementation.

By default the program will run for 2 'days', but you can easily change that value – it takes the number of days as a command-line argument. You can provide a command line argument by right-clicking on TesttextFixture, and choosing 'More Run/Debug' -> 'Modify Run Configuration'. In the box that pops up, put the number of 'days' you want to run the program with in the 'Program Arguments' box.



If the TexttestFixture program prints out all the values for all these items, why shouldn't we just rely on it for our testing? Can't we just go to the console and read all the values each time to make sure they're correct?

Writing our first test

The GildedRoseTest.java file has a single test implemented already – a very simple test that checks if the name that an Item was created with is the name it reports having. It is currently failing. Fix this test.

Your TA will demonstrate two ways of writing a test that confirms one of the rules specified in the kata text document – that "Sulfuras, Hand of Ragnaros" never loses Quality.

The first approach is to create the item, cause a certain number of days to pass, and check that the item still has its original value. We would like to write it like this, for example:

```
@Test
void legendaryQualityCheck() {
    Item[] items = new Item[] {new Item("Sulfuras, Hand of Ragnaros", 5, 80)};
    GildedRose app = new GildedRose(items);
    nDaysPassed(app, 8);
    assertEquals(80, items[0].quality);
}
```

Of course, for that to work we'd need an 'nDaysPassed()' method:

```
private void nDaysPassed(GildedRose app, int n) {
    for (int i = 0; i < n; i++) {
        app.updateQuality();
    }
}
```

This presents a problem. Ideally, we'd like to call nDaysPassed with *every* possible number of days that might pass. This is impossible. The next best approach would be to create many individual tests, each of which would test a given number of days passing. This would be tedious and error prone to type, and would create very large test classes. Luckily, junit gives us the ParameterizedTest, which allows us to pass a parameter to our test method.

Parameterized Tests

First, we need to add a couple of imports:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
```

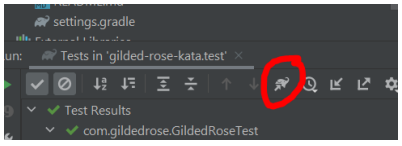
Then, we can write our test method with a parameter, and provide the arguments in a @ValueSource annotation.

```
@ParameterizedTest
@ValueSource(ints = {1,8,15,23,89,104,26,17,191,288})
void legendaryQualityCheckBetter(int n) {
    Item[] items = new Item[] {new Item("Sulfuras, Hand of Ragnaros", 5, 80)};
    GildedRose app = new GildedRose(items);
    nDaysPassed(app, n);
    assertEquals(80, items[0].quality);
}
```

Try running this, and note the output. Junit actually goes through the work of expanding our one test to (in this case) ten separate tests, and running each as if we'd typed it.

HTML test output with Gradle

Gilded Rose was bundled with a newer version of Gradle, so we have pretty HTML output available to us with our tests. Run the test suite, then click on the Gradle logo at the top of the test output box.



Test Exercises

Complete the following exercises in GildedRoseTest.java and push it to your repository by the due date. (worth 3% of your final grade)

1. Test that an ordinary item loses one quality before it 'expires' (while its sellIn value is 0 or greater), and that it loses two quality after it 'expires'.
2. Test that 'Aged Brie' *gains* quality instead of losing it (again, 1 point a day before it expires, and 2 points a day after).
3. Test that an ordinary item's quality cannot go below zero. Use a parameterized test, and test ten different possible durations.
4. Test that the quality of 'Aged Brie' cannot exceed 50. Use a parameterized test, and test ten different possible durations.
5. The quality of the the "Backstage passes to a TAFKAL80ETC concert" is governed by complicated rules. If one is created with a quality of 29 and a sellIn of 13, test that it has a quality of 50 after 12 days.
6. Given a backstage pass as initially created in #5 above, test that the quality of the item has dropped to 0 after 14 days.

Refactor Gilded Rose

By running your written tests and TesttextFixture, you've probably found that the GildedRose class is running properly – it is producing the correct outputs. If you've looked at the class itself, however, you've hopefully noticed that it is *very* poorly written. The remainder of the lab gives you an opportunity to refactor this program. A reminder of the rules:

- You are only allowed to alter GildedRose.java in your refactoring – you must leave Item.java alone.
- You cannot rename or delete updateQuality(), or replace the items field.
- Your refactored version of the program must still pass each of the tests you have written.

Your TA will demonstrate two or three improvements that can be made to the program, but we are deliberately not providing you with a lot of additional guidance for this task, as we want your own judgements about code quality to be your guide.

The refactoring part of this lab is worth 1% of your final grade. You must make 'substantial improvements' to the structure and clarity of the class, beyond what your TA demonstrates, in

order to receive these marks. Do not overdo it, though – there is no way to get the code 'perfect', and programmers with decades of experience discover new ways to improve Gilded Rose each time they sit down to try it.

Addendum: What about assertJ?

The Gilded Rose Kata repository ships with Junit only. If you'd prefer to use assertJ, of course you're welcome to do so. There are only a couple of steps necessary to bring the assertJ library into a project that doesn't already have it, and this gives us an opportunity to see how we can import an external library in IntelliJ.

Take a look at build.gradle in the root directory of your project. This file determines how dependencies like external libraries will be downloaded and added to your project. Modify the dependencies section so that it looks like this:

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.2'
    testImplementation 'org.junit.jupiter:junit-jupiter-params:5.6.2'
    testImplementation 'org.junit.jupiter:junit-jupiter-engine:5.6.2'
    testImplementation 'com.approvaltests:approvaltests:5.0.0'
    testImplementation ("org.assertj:assertj-core:3.18.1")
}
```

(The final line is the only one you need to add). This specifies the assertj-core library, and the version number you wish to use.

Save this file, and close IntelliJ. When you reload the project, you may be asked whether you wish to use a 'Gradle' or 'Maven' project. If so, choose 'Gradle'.

As always, you will need to add the following line to your imports at the top of a file to use assertJ assertions in that class:

```
import static org.assertj.core.api.Assertions.*;
```