ITCS473 Software Quality Assurance and Testing
JUnit Exercise

This exercise is to be done **individually**.

Please note that you should do the lab work sheet completely by continuing at home after the lab session is over but you must submit before 23.55 of the class day. Once it is finished, please submit your solution as a .zip file on MyCourses. Name your file as ITCS473_[YOUR ID]_JUnit.zip.

Exercise 1

1. Fork the Rational repository from `https://github.com/MUICT-SERU/Rational` to your GitHub account. This is a simple Java program to compute mathematical operations on rational numbers (i.e., fractions).
2. Clone the forked repository to your local machine.
3. Open the project in IntelliJ. When you're asked which type of project you will use to open, select "Maven". The project is tested with Java 8. So, make sure you set the Project SDK and Project language level to 1.8 (`File > Project Structure > Project` menu).

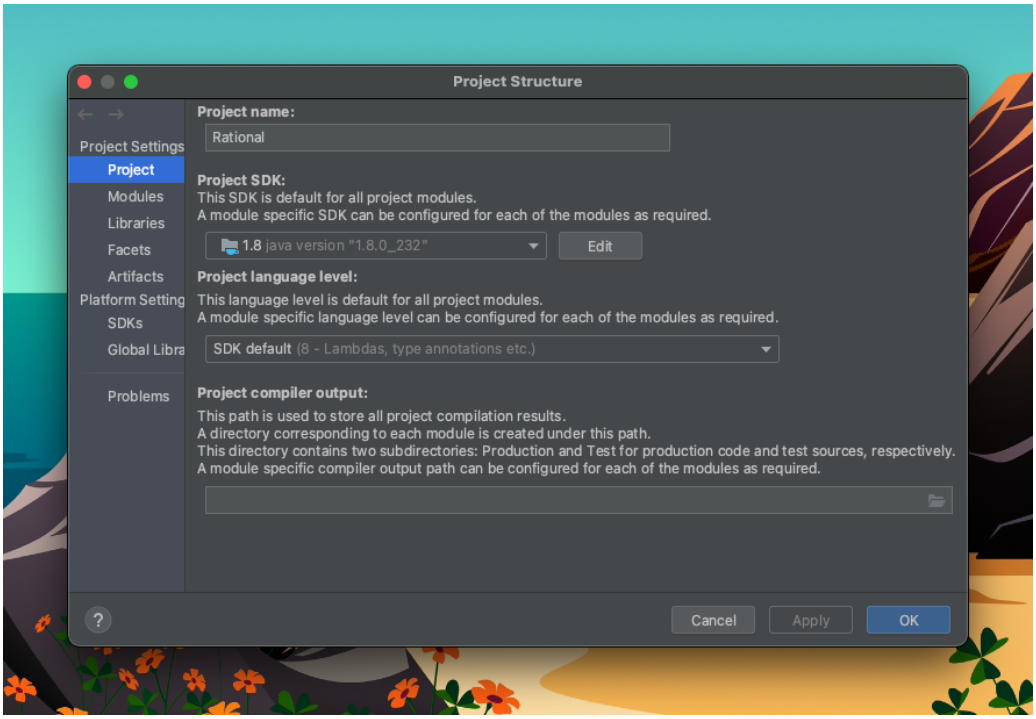

Figure 1: Java version configuration

4. Consider the class `Rational` in `src.main.java` with its partial implementation below.

```
class Rational {
        long numerator,denominator;

        class Illegal extends Exception {
                String reason;
                Illegal (String reason) {
                        this.reason = reason;
                }
        }

        Rational() {
                // to be completed
        }

        Rational(long numerator, long denominator) throws Illegal {
                // to be completed
        }

        // find the reduce form
        private void simplestForm() {
                long computeGCD;
                computeGCD = GCD(Math.abs(numerator), denominator);
                numerator /= computeGCD;
                denominator /= computeGCD;
        }

        // find the greatest common denominator
        private long GCD(long a, long b) {
                if (a%b ==0) return b;
                else return GCD(b,a%b);
        }

        /***
        * Compute an addition of the current rational number
        * to another given rational number
        * @param x the rational number to be added to the
        * current rational number
        */
        public void add(Rational x) {
                numerator = (numerator * x.denominator) + (x.numerator * denominator);
                denominator = (denominator * x.denominator);
                simplestForm();
        }

        /***
        * Compute a subtraction of the current rational number
        * to another given rational number
        * @param x the rational number to be subtracted from
        * the current rational number
        */
        public void subtract(Rational x) {
                // to be completed
        }

        /***
        * Compute a multiplication of the current rational number
        * to another given rational number
        * @param x the rational number to be multiplied to the
        * current rational number
        */
        public void multiply(Rational x) {
                // to be completed
        }
```

```
        /***
        * Compute a division of the current rational number
        * to another given rational number
        * @param x the rational number to be divided by the
        * current rational number
        */
        public void divide(Rational x) {
                // to be completed
        }

        /***
        * Check if the given rational number equals to
        * the current rational number
        * @param x the rational number to be compared to
        * the current rational number
        * @return true if the given rational number equals
        * to the current, false otherwise
        */
        public boolean equals(Object x) {
                // to be completed
                return true; // TODO: This needs to be modified.
        }

        /***
        * Compare the current rational number to the
        * current rational number
        * @param x the rational number to be compared to
        *  the current rational number
        * @return -1 if the current rational number is less than
        * the given number, 0 if they're equal, 1 if the current
        * rational number is larger than the given number
        */
        public long compareTo(Object x) {
                // to be completed
                return -1; // TODO: this needs to be modified.
        }

        /***
        * Give the formatted string of the rational number
        * @return the string representation of the rational
        * number. For example, "1/2", "3/4".
        */
        public String toString() {
                // to be completed
                return ""; // TODO: This needs to be modified.
        }

        public static void main(String[] args) {
                System.out.println("This␣is␣Rational␣class.");
        }
}
```

5. Implement a `RationalTest` JUnit test cases in `src.test.java` package that tests the above class **before** you create a full implementation for `Rational`. Apply the knowledge of input space partitioning that you learned in class to the test case design.

6. You will see that the `RationTest` class already has one test case.

```
import org.junit.Assert;
import org.junit.Test;

public class RationalTest {
        @Test
```

```
        public void testAdd() {
                Rational x = new Rational();
                x.numerator = 1;
                x.denominator = 2;
                Rational y = new Rational();
                y.numerator = 1;
                y.denominator = 4;
                x.add(y);
                Assert.assertEquals(3, x.numerator);
        }
}
```

7. Try to execute this test case by clicking the green arrow near the method name. You'll see the test result.

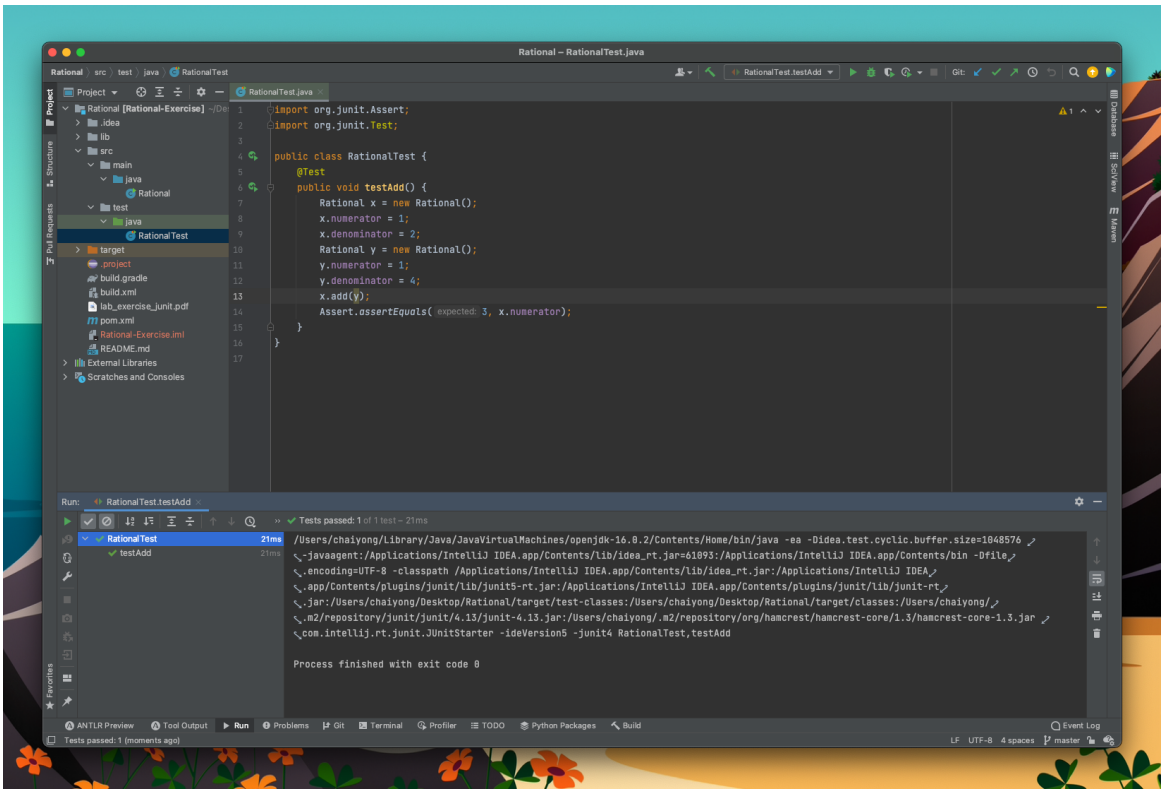

Figure 2: JUnit execution result in IntelliJ

8. After you have defined all the test cases for all the public methods in `Rational`, create a a full implementation for `Rational`. This is called Test-Driven Development (TDD).
9. Ensure that all tests pass and that your implementation of `Rational` is complete.

Exercise 2

Now use ant to test your implementation. Ant is a widely used building tool in the past. Now it is not that popular anymore although some projects still use it. Nonetheless, it is useful for you to start learning build tools from ant because of its simplicity. Several of the modern building tools' concepts also come from ant.

"Apache Ant is a software tool for automating software build processes which originated from the Apache Tomcat project in early 2000 as a replacement for the Make build tool of Unix. It is similar to Make, but is

implemented using the Java language and requires the Java platform. Unlike Make, which uses the Makefile format, Ant uses XML to describe the code build process and its dependencies." – Wikipedia

Step 1

- Enable the ant window by going to `View > Tool Windows > Ant`.
- In your project directory, create an ant build file (`build.xml` as shown below and available in the repository) with a `compile` target that compiles all Java source files using `javac` ant task.

```
<project>
  <target name="clean">
    <delete dir="build"/>
  </target>
  <target name="compile" depends="clean">
    <mkdir dir="build/classes"/>
    <javac srcdir="src" destdir="build/classes">
    <classpath location="lib/junit-4.13.jar" />
  </javac>
  </target>
  <target name="jar" depends="compile">
    <mkdir dir="build/jar"/>
    <jar destfile="build/jar/Rational.jar" basedir="build/classes">
    <manifest>
      <attribute name="Main-Class" value="Rational"/>
    </manifest>
    </jar>
  </target>
  <target name="run" depends="jar">
    <java jar="build/jar/Rational.jar" fork="true"/>
  </target>
</project>
```

- Ensure that your target can be executed successfully by going to the ant window in IntelliJ and `run ant > compile` and see the results of your build in the Messages window.

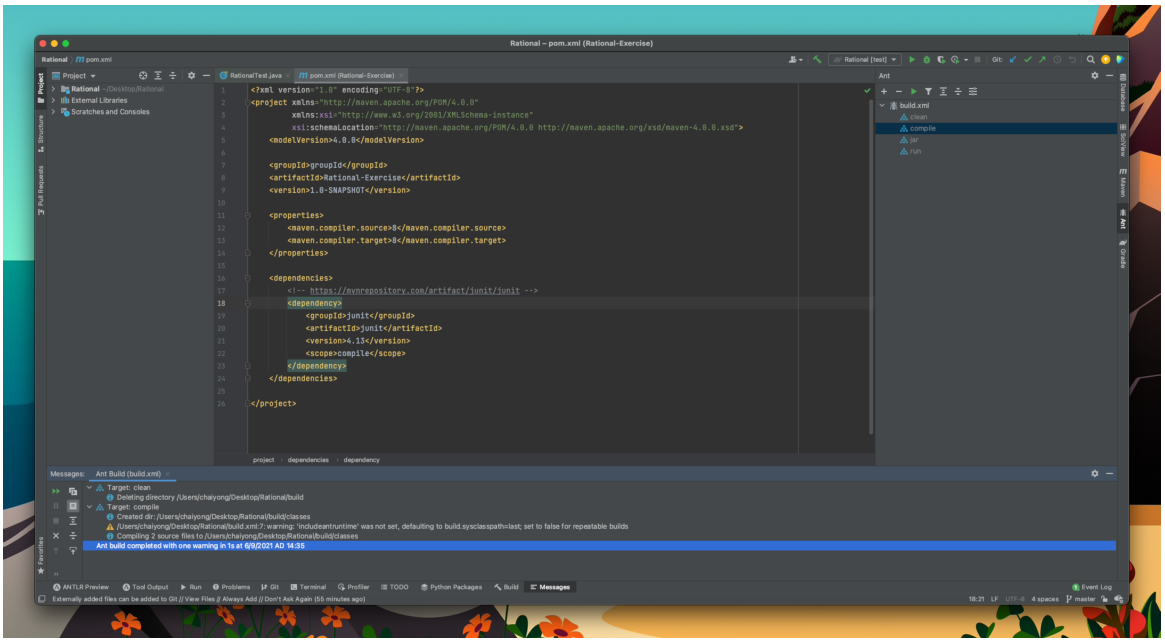

Figure 3: Ant tool window and its build result (Messages menu)

- Try to understand how ant tasks work. Do you understand each of the ant task and their dependencies?

Step 2

- Create a `test` target which depends on the `compile` target.
- Add the following `junit` task inside the `test` target.

```
<target name="test" depends="compile">
  <junit showoutput="yes" printsummary="yes" haltonfailure="no">
    <classpath location="build/classes" />
    <classpath location="lib/junit-4.13.jar" />
    <classpath location="lib/hamcrest-core-1.3.jar" />
    <test name="RationalTest" />
  </junit>
</target>
```

- Go to the ant window in IntelliJ and run `ant > test` and see the results of your test cases.

Step 3

You can tell the `junit` task to create a report of running the test cases by adding `todir attribute` with a directory name and specify the report format.

- Modify the `junit` task as shown below.

```
<target name="test" depends="compile">
  <mkdir dir="report"/>
    <junit showoutput="yes" printsummary="yes" haltonfailure="no">
      <classpath location="build/classes" />
      <classpath location="lib/junit-4.13.jar" />
      <classpath location="lib/hamcrest-core-1.3.jar" />

      <test name="RationalTest" todir="report">
        <formatter type="plain" />
        <formatter type="xml" />
    </test>
  </junit>
</target>
```

- After running `ant > test`, check the `report` folder. What do you find in there?
- If you have the `ant` tool installed on your local machine, you can also execute it in the terminal by using the command `ant compile` or `ant test`.

Exercise 3

Now use Maven to test your implementation. Maven is a more modern building tool than ant and it is still widely used nowadays.

"Maven addresses two aspects of building software: how software is built, and its dependencies. Unlike earlier tools like Apache Ant, it uses conventions for the build procedure. Only exceptions need to be specified. An XML file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins. It comes with pre-defined targets for performing certain well-defined tasks such as compilation of code and its packaging. Maven dynamically downloads Java libraries

and Maven plug-ins from one or more repositories such as the Maven 2 Central Repository, and stores them in a local cache." – Wikipedia

Follow the instruction below.

1. Your project already contains Maven configuration file called `pom.xml`.
2. Switch your IntelliJ to Maven mode by right clicking on the `pom.xml` file and select "Add as Maven project". You'll see the Maven window appears on the right.
3. Open **pom.xml** file and add the JUnit dependency below within the `<project></project>` tags.

```
<dependencies>
  <!-- https://mvnrepository.com/artifact/junit/junit -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

4. Try to build the project using Maven by going to the Maven window and select `Lifecycle > compile`. See the result.

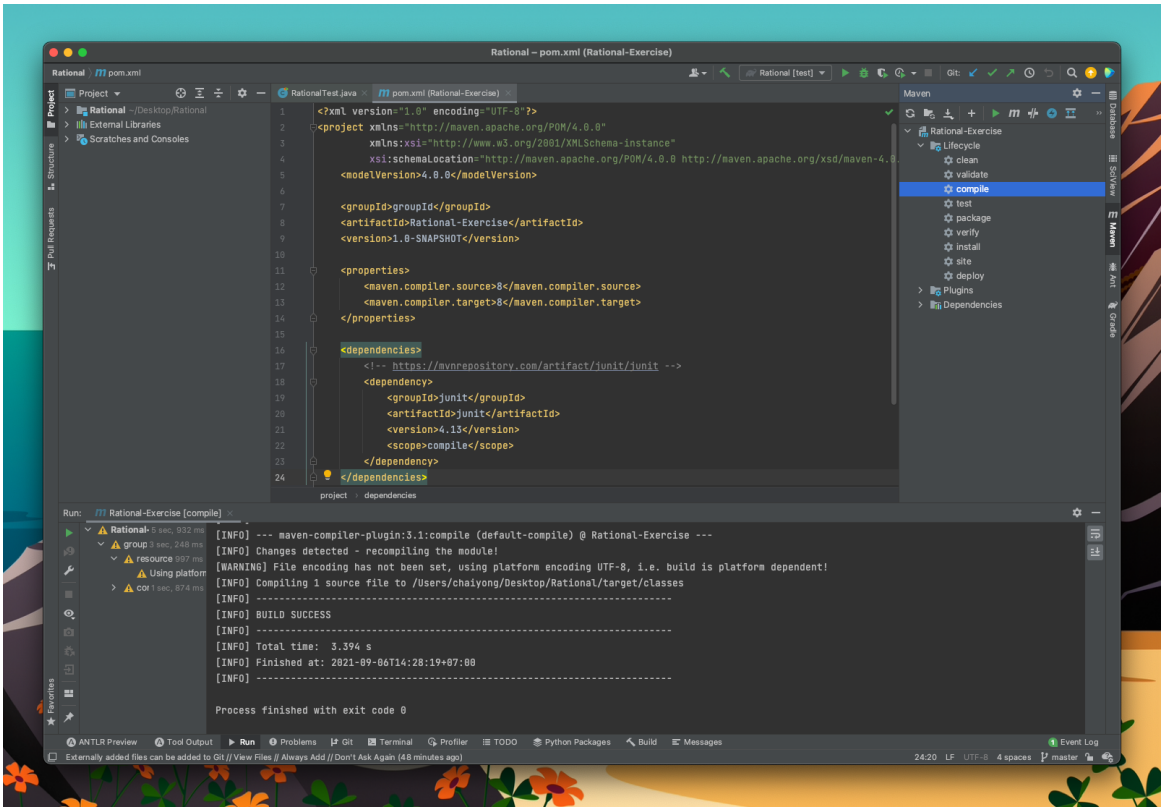

Figure 4: Maven build resul

5. Build the project with the command `Lifecycle > install`. Check the test result and make sure that all the tests pass.
6. If you have the `ant` tool installed on your local machine, you can also execute it in the terminal by using the command `mvn install`.

Exercise 4

Lastly, you will practice using Gradle for building and testing your program. Gradle is one of the latest building tools.

"Gradle is a build automation tool for multi-language software development. It controls the development process in the tasks of compilation and packaging to testing, deployment, and publishing. Supported languages include Java (Kotlin, Groovy, Scala), C/C++, and JavaScript. Gradle builds on the concepts of Apache Ant and Apache Maven, and introduces a Groovy- & Kotlin-based domain-specific language contrasted with the XML-based project configuration used by Maven. Gradle uses a directed acyclic graph to determine the order in which tasks can be run, through providing dependency management." – Wikipedia

Step 1

Enable the Gradle windown in IntelliJ by going to `View > Tools Window > Gradle`. You'll see the Gradle window appears on the right.

Step 2

1. Open the `build.gradle` file. You'll see the following configuration.

```
plugins {
     id 'java'
}

group 'org.example'
version '1.0-SNAPSHOT'

repositories {
  mavenCentral()
}

dependencies {
  testImplementation group: 'junit', name: 'junit', version: '4.13'
}
```

2. Notice that Gradle also relies on the Maven Central repository to download the project's dependencies. Also, Gradle in this project is configured to use JUnit 4.13 for testing.
3. Execute the tests using Gradle by selecting `Tasks > Verification > test`. See the result.

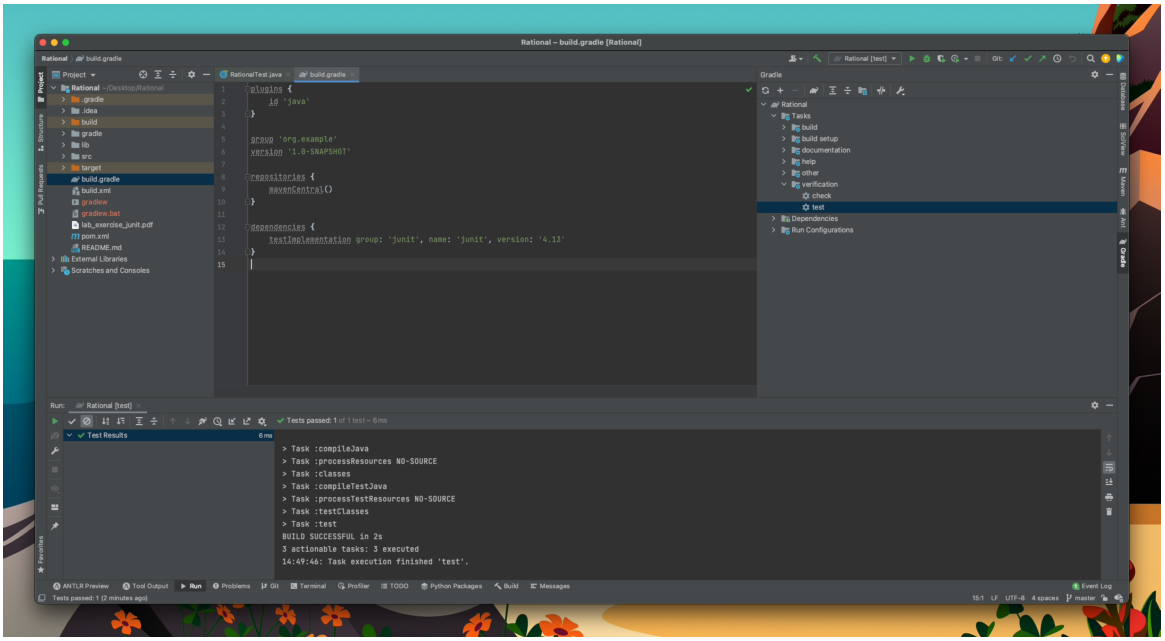

Figure 5: Gradle build and test result

4. Gradle also generates test reports in nicely-looking HTML format. To see the test report, go to the `build` folder and go into `reports`. Then, open the `index.html` file. You will see the test report as shown below.

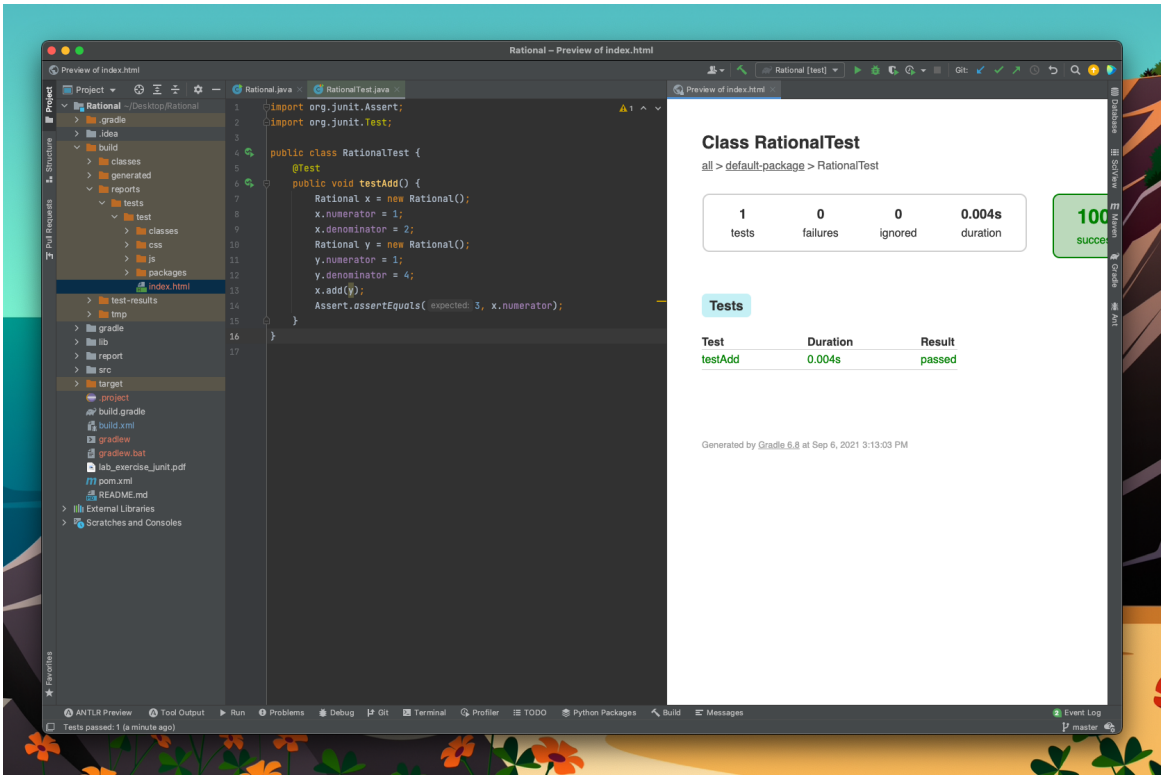

Figure 6: Gradle test report

Submission

After you have finished all the exercise steps, add the updated files and commit. Then push the changes to your GitHub repository. Submit the GitHub repository link to your `Rational` repository on MyCourses.