Unit Testing

Unit testing is a form of white-box testing, in which test cases are based on an internal structure. **The tester chooses inputs to explore particular execution paths and also determines the appropriate output for a set of input.** The purpose of unit testing is to examine the individual components or pieces of methods/classes to verify functionality, ensuring the behavior is as expected.

The exact scope of a "unit" is often left to interpretation, but a nice rule of thumb is for a unit to contain the ***least amount of code that performs a standalone task*** (e.g. a single method or class). There is good reason that we limit scope when unit testing — if we construct a test that incorporates multiple aspects of a project, we have shifted focus from functionality of a single method to interaction between different portions of the code. If the test fails, and we don't know why it failed, we are left wondering whether the point of failure was within the method we were interested in or in the dependencies associated with that method.

Regression Testing

Complementing unit testing, regression testing makes certain that the latest fix, enhancement, or patch did not break existing functionality, by testing the changes you've made to your code. Changes to code are inevitable, whether they are modifications of existing code, or adding packages for new functionality; your code will certainly change. It is in this change that lies the most danger, so, with that in mind, regression testing is a must.

What Is JUnit?

JUnit is a unit testing framework for the Java programming language that plays a big role in regression testing. As an open-source framework, it is used to write and run repeatable automated tests.

As with anything else, the JUnit framework has evolved over time. The major change to make note of is the introduction of annotations that came along with the release of JUnit 4, which provided an increase in organization and readability of JUnits.

The JUnit library is not part of the Java standard and as such needs to be downloaded and configured separately. Unit and Regression testing is an incredibly complicated concept, and we will not be exploring all of the abilities of JUnit. The goal here is to introduce you the ***concept*** of unit testing and to get you started with writing and running some basic tests. This rabbit hole runs very deep and is more of a ***Software Engineering*** concern. But much like GIT, it will benefit your development as a programmer to begin as soon as possible, even if you aren't using these tools to their utmost capacity. Being familiar with the terminology and basic usage will give you a leg up and provide you with some direction to continue your study.

Here is the official sit for JUnit: https://junit.org/junit5/

The amount of information contained here will probably be overwhelming. I will take a simplistic approach.

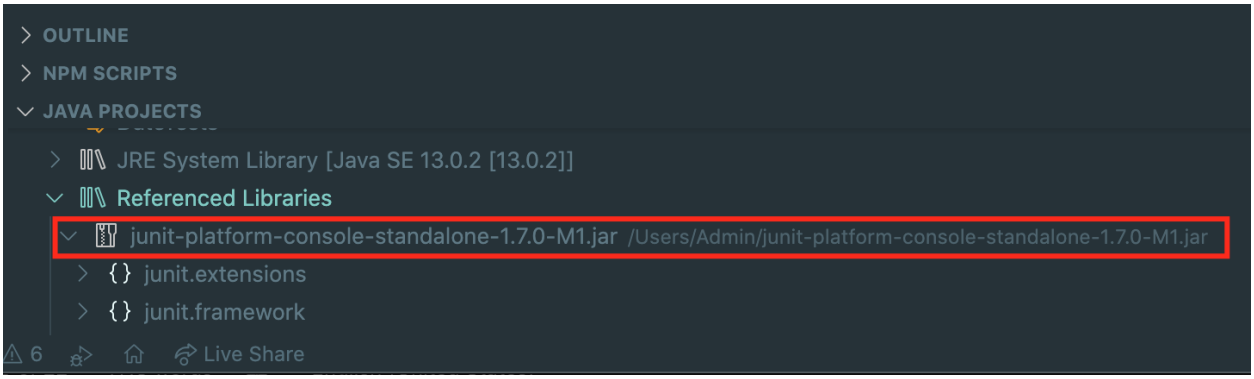
Configuration

The following steps are necessary to get VS Code set up for running JUnit tests.

1. **Download the standalone JAR file.** A "JAR" file is a ***Java Archive*** and is the standard way that Java libraries are compiled, configured and shared. You may have noticed JAR files in your exploration of your Java installation and in the configuration of JavaFX.
    a. **https://repo1.maven.org/maven2/org/junit/platform/junit-platform-console-standalone/1.7.0-M1/junit-platform-console-standalone-1.7.0-M1.jar**
    b. This JAR file needs to be stored somewhere easily accessible because we will need to tell VS Code where to find it so when we use the Java Test Runner extension, it will know where to find the classes. I placed this file in same location as the JavaFX JAR file from earlier in the semester. (My **HOME** directory)
2. Provide VS Code with the path to the JUnit JAR. This site describes this process well.
    a. https://supunkavinda.blog/vscode-editing-settings-json
    b. We want to configure VS Code's global settings so that you can import these JUnit classes from any project. VS Code's settings are stored in a JSON file called ***settings.json***. This file can be found in the following locations. You can change the ***settings.json*** from your user preferences. The changes are global. Therefore, it will affect all of your projects. Here are 2 ways to reach that global ***settings.json*** file.
        i. From within VS Code: *File -> Preferences -> Settings -> Extensions -> Scroll down and find "Edit in settings.json"*
        ii. Or in these paths in your OS
            1. **Windows** *%APPDATA%\Code\User\settings.json*
            2. **macOS** *$HOME/Library/Application Support/Code/User/settings.json*
            3. **Linux** *$HOME/.config/Code/User/settings.json*
    c. Add the following lines to your ***settings.json*** file. The second path is the most important. You will need to provide the ***full path for your JAR file location.*** Do not copy my path. This is for **MY** location. ***/Users/Admin*** is the path on my Mac.

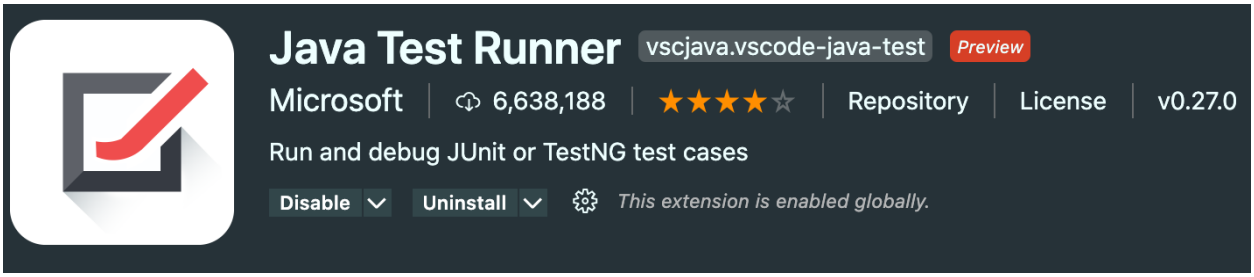
```
"java.project.referencedLibraries": [
    "lib/**/*.jar",
    "/Users/Admin/junit-platform-console-standalone-1.7.0-M1.jar"
],
```

You should now be able to see this library from within VS Code when working on a Java project. You may need to shut down and restart

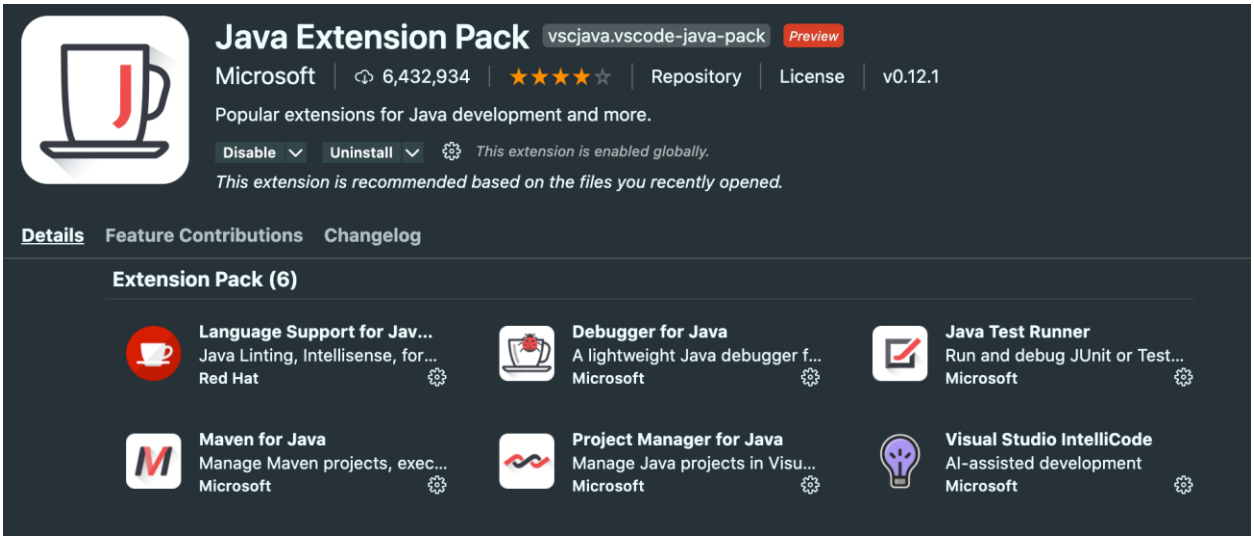


Installing JUnit is the first part of this task. The second part is ensuring that we have a framework that can actually run the tests and provide meaningful feedback on the test results. Most modern IDEs come with the ability to generate and run JUnit tests. VS Code is no different.

There is a VS Code extension called ***Java Test Runner*** that we will be using



Java Test Runner comes bundled with the ***Java Extension Pack*** that you should have installed at the beginning of the semester. Check your VS Code extensions to ensure you have this extension installed.



NOTE: The configurations we just made only allow VS Code to interact with the JUnit framework. Normal compilation and execution of JUnit tests using ***javac*** and ***java*** would require further configuration of PATH variables. We are not going to worry about that. We will only be using the ***Java Test Runner*** framework.

JUnit and Unit Testing Concepts

Assertions: An *assertion* is a statement in a programming language that enables you to test your assumptions about your program. For example, if you write a method that calculates the speed of a particle, you might assert that the calculated speed is less than the speed of light. If the result of the calculation violates this assertion the program should throw an error.

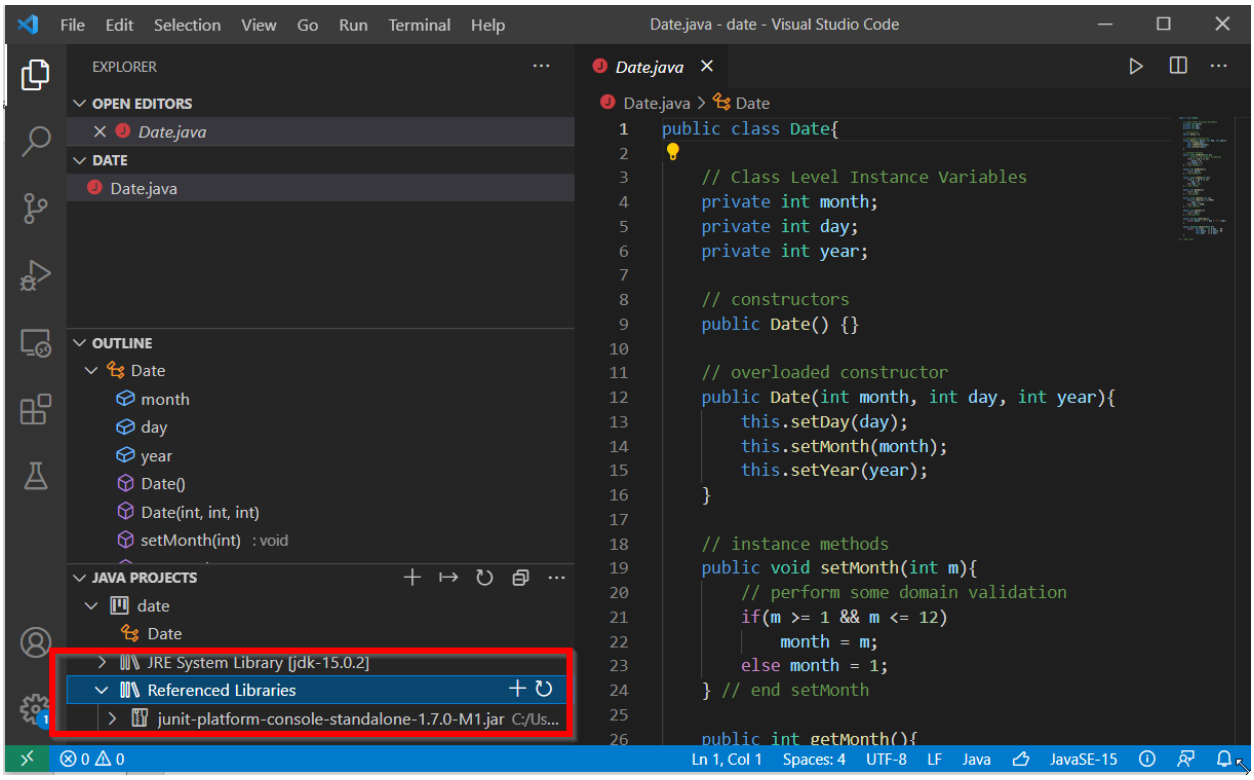
Each assertion contains a ***boolean expression*** that you believe will be true when the assertion executes. If it is not true, the system will throw an error. By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the unit in question is free of errors.

JUnit testing operates on this concept of ***assertions.*** You will see this concept in the names of the ***test methods*** we use.

The following screen shots will walk you through the setup of a JUnit test for our setter methods in the Date class in VS Code. In particular we are concerned with the overloaded **setMonth** methods and we want to ensure that they correctly allow valid months to be entered and deny invalid months to be entered. One of the methods accepts an integer (month number) and the overloaded method accepts a String (month name)

```java
// instance methods
public void setMonth(int m){
    // perform some domain validation
    if(m >= 1 && m <= 12)
        month = m;
    else month = 1;
} // end setMonth

//overloaded setMonth
public void setMonth(String month){
    int monthNumber = this.getMonthNumber(month);
    if(monthNumber > 0)
        this.month = monthNumber;
} // end overloaded setMonth
```

1. Create a new workspace with the Date.java class from the course materials
2. Make sure you can see the JUnit jar file under ***referenced libraries***



You can see the amazing views that VS Code provide us. We now have an ***outline*** for our classes, a reference to the local Java System Library and a reference to the JUnit framework. We will be unlocking incredible VS Code behaviors as we move forward.

1. Add a new java source code file to your workspace. Name the file ***DateTests.java***

Add the following import statements

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
import static org.junit.jupiter.api.Assertions.assertEquals;
```

1. ***Test*** and ***BeforeEach*** are annotations that we will use to specify the types of methods they are. Any method annotated with **@Test** is considered a test method and will be run as such by the ***test runner.*** Any method annotated with **@BeforeEach** will be executed ***before each*** test method. This is a good way to run any code that needs to fire before a test method.
2. ***assertEquals*** is a JUnit method that we will use to formulate our testing approach. Since we are tsting our compareTo method we will be looking for the values **{-1, 0, 1}** in our tests. Let's write a simple unit test to verify the correctness of our compareTo. For instance, we would be interested in determining if our compareTo method can correctly determine if two dates are equal. The compareTo method ***should*** return ***0*** if two dates are equal, therefore we would want to ***assert*** that "1/2/2021" compared to "1/2/2021" is ***equal*** to ***0.*** We would also want to define ample tests to cover all cases of date equality.

JUnit Assertion Documentation:
https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html

Take a look at all the static methods here. It is a master class in method overloading.

I have added an import statement for the ***assertTrue*** JUnit method

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;
```

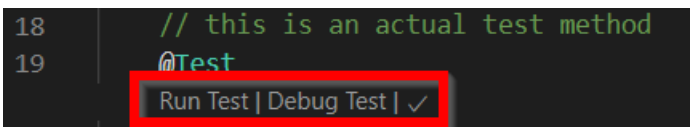
Demonstration of some basic Unit Tests

In the following scenario an ***assertEquals*** JUnit test is being set up. Also shown is a @BeforeEach method

```java
public class DateTests {

    // Date instance for testing
    Date testDate;

    // run this method before each @Test method
    // gives us a starting point for each test
    @BeforeEach
    public void setup(){
        testDate = new Date(3, 21, 2021);
    }

    // this is an actual test method
    @Test
    Run Test | Debug Test | ✓
    public void testSetMonth(){
        // get orginal month
        int originalMonth = testDate.getMonth();

        // call the method we are testing: setMonth(String)
        testDate.setMonth("Jimuary");

        // get the month after setter call.
        // should remain unchanged from initialization
        // if testMonth == originalMont our validation worked
        int testMonth = testDate.getMonth();

        // use the assertEquals method. The "expectation" is the first arg
        // the "actual" is the second arg
        // we are "asserting" that the original month and the testMonth are equal
        assertEquals(originalMonth, testMonth);
    }
}
```

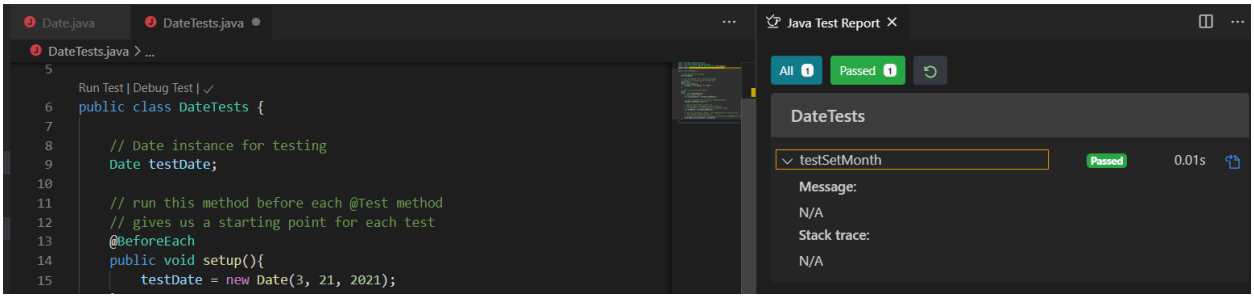
Ensure that you understand each idea shown in this file. Consult the JUnit documentation if you need to.

Right above the signature for **testSetMonth** you will see a small menu for executing the ***Java Test Runner.*** Notice that there is also a way to debug the tests. Like anything else, your tests can issues. You need to pay close attention.



Java Test Runner Quirks: When executing a test, the default behavior of the test runner is to only alert to failing tests, so you won't see any bells or whistles if your test passes. Clicking on the little arrow will show the feedback pane for any test.

Try running the test now. This test should pass, so click the arrow to show the feedback. You can see that 1 test was executed and 1 test passed. This is what we want.



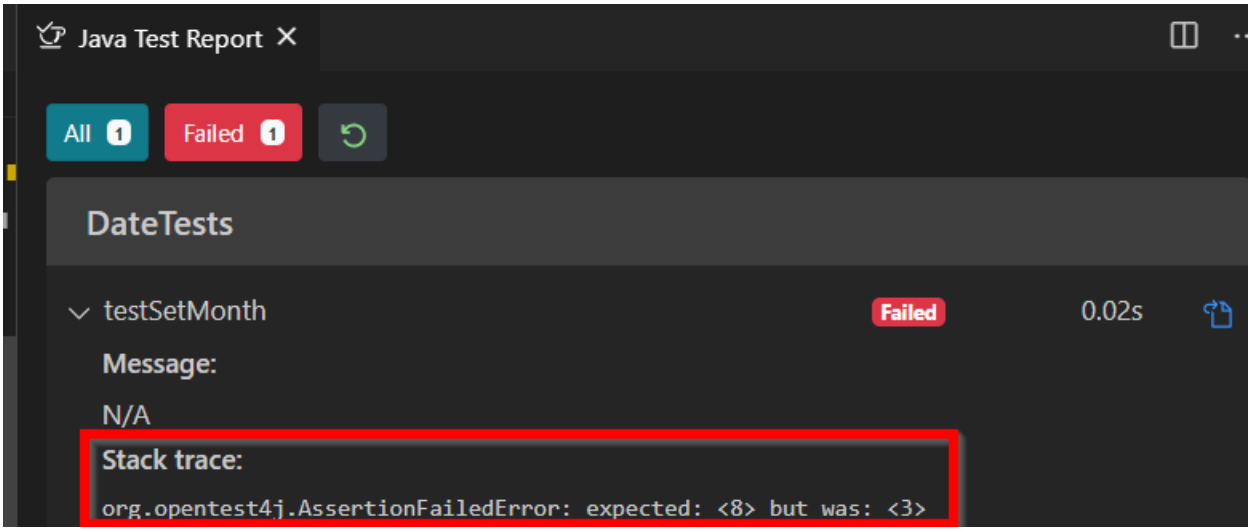
I'll now add a bug to illustrate what a failing test looks like. I changed the domain validation code in the **setMonth(String)** method to this. This code will not allow any changes to the instance variable "month", even if they are valid.

```
//overloaded setMonth
public void setMonth(String month){
    int monthNumber = 0; //this.getMonthNumber(month);
    if(monthNumber > 0)
        this.month = monthNumber;
} // end overloaded setMonth
```

The following assertEquals was added to the test method. In order to get a "pass" on a single method ALL assertions must be true.

```
int expectedMonth = 8;
testDate.setMonth("August");
testMonth = testDate.getMonth();
assertEquals(expectedMonth, testMonth);
```

Study the output of the failed test. Notice that we are told the expectation was **8** but the actual was **3.** This means that the method did not perform up to our expectations. Understand that your expectations have to first be correct before you can start testing.



Testing the "equals" method. This method is a great candidate for unit testing due its complexity. Remember, the equals method is designed to perform a **deep comparison** between two instances of the same class. This requires cycling through each of the instance variables, or a subset of instance variables (whatever meets your team's goals) and checking to see if they are **all equal.**

Here is the Date class' equals method. Remember that **"this"** refers to the ***calling instance***, the Date object that was used to call the method. So if the invocation is

dateOne.equals(dateTwo);

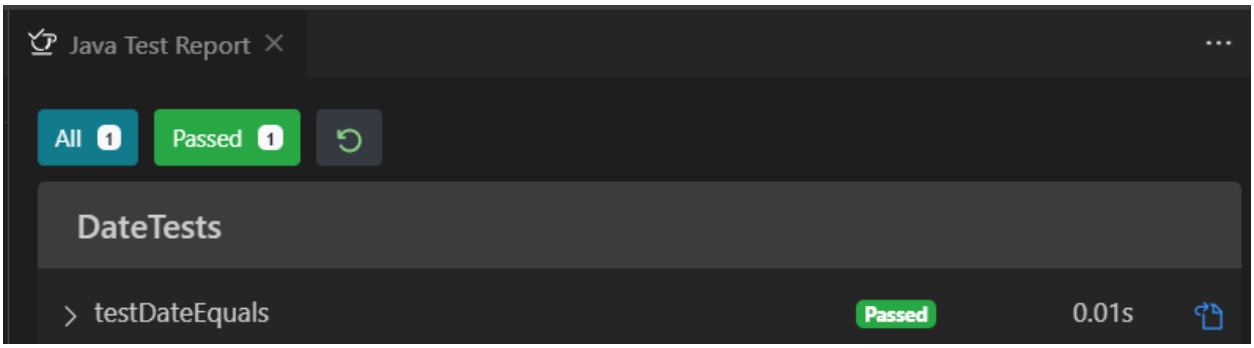
Then "this" refers to **dateOne** and **d** will refer to **dateTwo**

```
// perform a deep comparison between
// "this" date and the parameter Date
public boolean equals(Date d){
    return  this.day    == d.day    &&
            this.month  == d.month  &&
            this.year   == d.year;
}
```

Here is the test method. Notice that this method utilizes the JUnit ***assertTrue*** method. This method operates on a boolean argument which makes it perfect for testing our equals method because ***equals*** returns a boolean. If the argument condition is **true**, the method passes and vice versa.

My testing approach was to clone the existing test instance into a new Date by copying its instance variables. This ***should*** return true.

```
@Test
Run Test | Debug Test |
public void testDateEquals(){
    // use the Date initiliazed in setup()
    int month   = testDate.getMonth();
    int day     = testDate.getDay();
    int year    = testDate.getYear();

    // build identical instance
    Date toCompare = new Date(month, day, year);

    // use assertTrue
    assertTrue( testDate.equals( toCompare ));
}
```

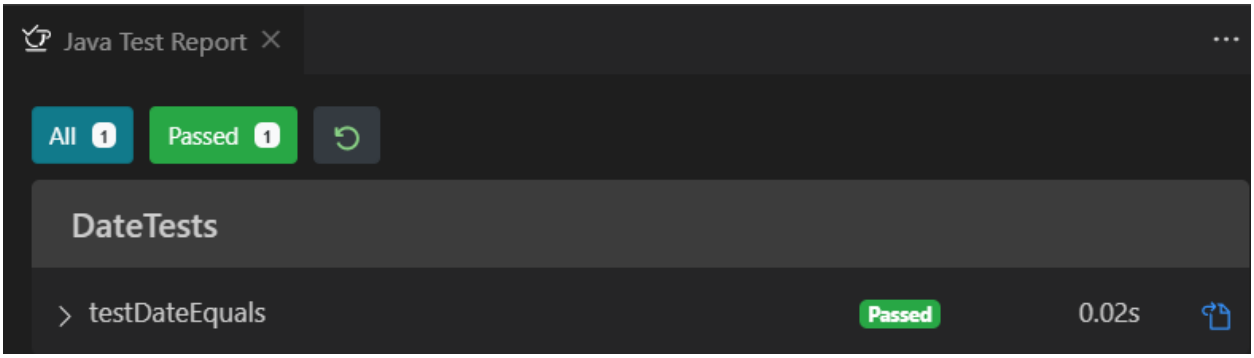


Hallelujah!

It's also a good idea to test the ability of our equals method to return **false.** JUnit has an ***assertFalse*** method just for this. Be sure to import it!

This scenario is a little more complicated because we need to verify that our equals method can catch different ***days, months and years.*** Any of these can cause two dates to be not equal.

```
@Test
Run Test | Debug Test | ✓
public void testDateEquals(){
    // use the Date initiliazed in setup()
    int month   = testDate.getMonth();
    int day     = testDate.getDay();
    int year    = testDate.getYear();

    // build identical instance
    Date toCompare = new Date(month, day, year);
    // use assertTrue
    assertTrue( testDate.equals( toCompare ));

    // continue using the Date from setup()
    // test the month. day and year are the same
    Date falseDate = new Date(2, 21, 2021);
    // use assertFalse
    assertFalse( falseDate.equals( testDate ));
    // test the day
    falseDate.setMonth(3);  // equal to testDate
    falseDate.setDay(2);    // different from testDate
    assertFalse( falseDate.equals( testDate ));
    // test the year
    falseDate.setDay(21);   // equal to testDate
    falseDate.setYear(1900);// different from testDate
    assertFalse( falseDate.equals( testDate ));
}
```



Got all passes on this test. The equals method testing is a good example of the challenging nature of defining appropriate tests. You need to **ensure coverage**. All cases need to be tested and caught and our testing methods can become as complex, or more so, than our code in question. This may seem a little arduous but you'll have these tests saved and ready to use again any time changes are made to the method being tested. Any patch, or update, or logic refactor **should not break these tests.** This is the basic concept of regression testing . . . any updates should not break existing functionality. You can reuse these unit tests anytime you are about to push an update to an existing code base.