Roses are Red,
Violets are Blue

Unexpected '{'
on line 32.

Meme Credit: Reddit user u/sachintripathi007

Lab 2: Javadoc, JUnit, & Debugging

CS 2112 Fall 2020

September 14 / 16, 2020

Javadoc Overview

- Javadoc is a tool for creating html documentation
- The documentation is generated from comments
- It produces actual html web pages
- Helps keep documentation consistent with the code

Doc Comments

- Doc comments start with /** and end with */.
- The /** and */ should be on their own lines
- Additional lines need to start with *
- Comments can have html tags

```
    /**
     * This is a javadoc comment
     */
```

Writing Good Docs

- First sentence: high level overview (no fluff)
  - Bad: This method is a method that computes the square root of a number
  - Good: Computes the square root
- Go into detail if necessary after first sentence
- Don't repeat things in the method declaration
  - Bad: The first argument is an integer
- Document preconditions and invariants
- Cover Edge Cases!

Javadoc Tags

- Use tags to help Javadoc parse your comments
- Tags start with a @ and are case sensitive
- It must be at the beginning of the line

```
/**
 * Prints the kth element of the list
 *
 * @author Alexander Lee
 * @param list The list whose element is to be printed
 * @param k    The index of the item to be printed
 */
 public void printK(List list, int k) {

 }
```

Important Tags

Some important tags

- @param: describes a specific parameter
- @return: describes the return value
- @throws: describes exceptions it throws

Javadoc and Eclipse

- To generate the doc, `Project > Generate Javadoc...`
- Eclipse automatically generates Javadoc comments if the method signature is already written
- Can configure Eclipse to complain about missing Javadoc comments: `Window (or Eclipse) > Preferences... > Java > Compiler > Javadoc`

What’s a ‘Unit’ Test?

When writing test cases, try to make them as small as possible. If you have e.g. one test that checks three things, consider breaking it into three tests that each check one thing.

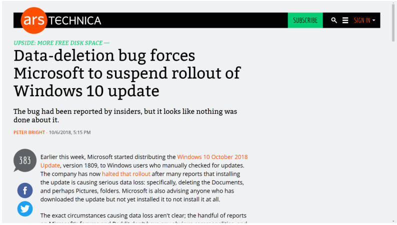
This way, if a test fails, you know exactly what is broken.

- Unit = Usually one method or a small group of methods
- Try to keep units as independent as possible
- Test and fix units as you go

Why Write Tests?

"Manual testing works just fine!"
- Famous Last Words

Why Write Tests?


ars TECHNICA

UPSIDE: MORE FREE DISK SPACE —

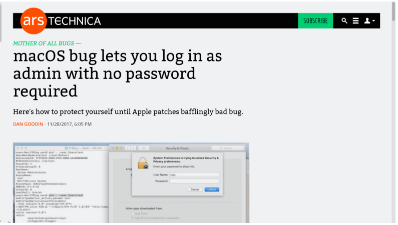
Data-deletion bug forces Microsoft to suspend rollout of Windows 10 update

The bug had been reported by insiders, but it looks like nothing was done about it.

PETER BRIGHT - 10/6/2018, 5:15 PM

Earlier this week, Microsoft started distributing the Windows 10 October 2018 Update, version 1809, to Windows users who manually checked for updates. The company has now halted that rollout after many reports that installing the update is causing serious data loss: specifically, deleting the Documents, and perhaps Pictures, folders. Microsoft is also advising anyone who has downloaded the update but not yet installed it to not install it at all.

The exact circumstances causing data loss aren't clear; the handful of reports


ars TECHNICA

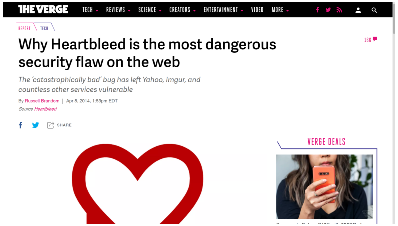
MOTHER OF ALL BUGS —

macOS bug lets you log in as admin with no password required

Here's how to protect yourself until Apple patches bafflingly bad bug.

DAN GOODIN - 11/28/2017, 6:05 PM

SUBSCRIBE
Security & Privacy
System Preferences is trying to unlock Security & Privacy preferences.
Enter your password to allow this.
User Name:
Password:
Cancel
Unlock
Allow apps downloaded from:
App Store
App Store and identified developers


THE VERGE

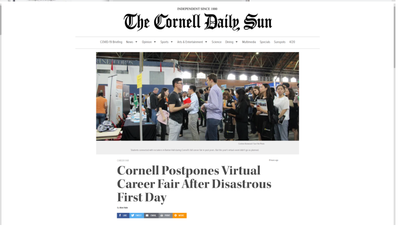
REPORT \ TECH

Why Heartbleed is the most dangerous security flaw on the web

The 'catastrophically bad' bug has left Yahoo, Imgur, and countless other services vulnerable

By Russell Brandom | Apr 8, 2014, 1:53pm EDT

Source Heartbleed


INDEPENDENT SINCE 1880

The Cornell Daily Sun

Cornell Postpones Virtual Career Fair After Disastrous First Day

Why Write Tests?

- There is too much code to manually test
- Manual testing is error-prone
- Make sure tests are actually run
  - Manual testing takes time, so is frequently skipped or put off
  - Automated test cases can be run automatically and as frequently as needed

Black vs. White Box Testing

Black box tests are written based on the specification alone. They can help ensure that the code works correctly from the client perspective.

White (or glass) box tests are written by looking at the implementation and writing tests to target the specific code. They can help find additional edge cases.

Writing Good Tests

- When a new bug is discovered, write a test case covering that bug. Not only does this check that the bug has been fixed, it also ensures that if the bug is ever reintroduced into the program, you'll know exactly where and when.
- Keep track of control flow to avoid writing superfluous test cases. If a method has a check that a variable `x` is not `null`, it's pointless to add in test cases for `x` being `null` between that check and the next time `x` gets assigned to.

  Keeping track of control flow can give you an idea of how many test cases a method should have. In general, one test case per possible control flow is sufficient, although it can be tricky to count all of the possible paths of execution.

Edge Cases

Make sure to cover all corner cases with your tests. When writing tests for data structures, always test with structures that have one or zero elements in them. When writing numerical tests, always test 0 and 1, negative numbers, min and max value, etc. With strings, test empty strings and strings with only one character.

In Java, it's also a good idea to add tests for `null` objects. Without proper testing, a `NullPointerException` might not be noticed until enough code has been written to make it difficult to track down where in the program the `null`s are coming from.

Eclipse: Setting Up JUnit

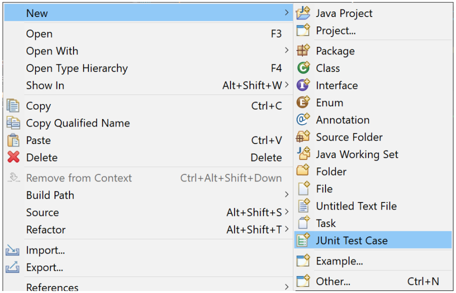
Setting up JUnit with Eclipse is fairly simple. Eclipse should have come with JUnit support, so all that you need to do is include the appropriate library.

To do this, right-click on your project and go to **Build Path** → **Add Libraries...**. Select JUnit 5. Once you have done this, Eclipse should be able to run any Java class that is properly set up as a JUnit test class. You may need to right-click on the file and select **Run As** → **JUnit Test**.

The JUnit website is: `http://junit.org/`

Creating a Test Class

You can create a JUnit test for a specific class by right-clicking a class in the package explorer, and then choosing **New** → **JUnit Test Case**.



Basic Test Case

```java
@Test
void basicTest() {
    assertTrue(true, "true is true");
}
```

Any method that is preceded by `@Test`, returns `void` and has no arguments will be run as a test case. The actual testing in the test case is done using assertion statements such as `assertTrue`.

Useful Assertion Statements

- `assertFalse(boolean cond)`
- `assertNotNull(Object o)`
- `assertNull(Object o)`
- `assertTrue(boolean cond)`
- `fail(String msg)`

`fail(String msg)` is useful when you have code that is supposed to be unreachable, e.g. if you expect an exception to be thrown.

Each of these methods can also take a description as the second argument: `assertTrue(boolean cond, String msg)`

For a complete list, go to:

`https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html`

Testing Exceptions

```
@Test
public void exceptionTest() {
    try {
        Integer.parseInt("error");
    } catch (NumberFormatException e) {
        // Desired exception, so test passes
    }
}
```

If you want to test if a particular method throws an exception, you can use try/catch blocks with the appropriate assert statements to make the test fail if the appropriate exception isn't thrown.

Testing Exceptions

```java
@Test
void exceptionTest() {
    assertThrows(NumberFormatException.class, () -> {
        Integer.parseInt("error")
    });
}
```

A cleaner way to test if an exception should be thrown is to use the new assertThrows method in JUnit 5. Now the test case will be considered to have passed if that exception was thrown, and failed if a different exception or no exception was thrown.

Example: BasicTestClass

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class BasicTestClass {
    @Test
    public void basicTest() {
      assertTrue(true, "true is true");
    }

    @Test
    public void exceptionTest() {
      assertThrows(NumberFormatException.class,
      () -> Integer.parseInt("error"));
    }
}
```

Ignoring Tests

If you want to temporarily disable a test case (this might come up if you have test cases for parts of your program that aren't fully implemented yet, for instance), you can do so by putting `@Disabled` above `@Test`. When running tests, JUnit will distinguish between tests that pass, tests that fail due to an assertion, tests that fail due to an unexpected and uncaught exception, and tests that were ignored.

Example: IgnoredTestClass

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

public class IgnoredTestClass {
    @Test
    public void basicTest() {
        assertFalse(false, "false is false");
    }

    @Disabled
    @Test
    public void ignoredTest() {
        fail("ignore me");
    }
}
```

Test Fixtures

Sometimes you have a specific setup that you want to run several tests on. Rather than copy and paste the code to recreate that setup, you can create a test fixture.

A test fixture is just an ordinary test class, but with some special functions that are called at certain times: before and after each test, and also before and after the entire set of tests. Instead of being labeled with `@Test`, these special functions are labeled with `@BeforeEach`, `@AfterEach`, `@BeforeAll`, and `@AfterAll`, respectively. Class members are used to store environment variables and make them available to the tests.

Test Fixture Special Methods

Just like a normal test method, the `@BeforeEach` and `@AfterEach` methods are expected to be `public`, `void`, and have no arguments. The `@BeforeAll` and `@AfterAll` methods are expected to be `public static void` and have no arguments.

The purpose of the `@BeforeEach` method is to set up the environment for each test case. The purpose of the `@AfterEach` method is to clean up any external resources used by the test, such as deleting temporary files. `@BeforeAll` and `@AfterAll` are used for one-time setup and cleanup for the entire test fixture. If your test fixture does not require some of these methods, you can just leave them out.

Example: BasicTestFixture

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

public class BasicTestFixture {
    private int[] x;

    @BeforeEach
    public void setup () {
        x = new int[1];
    }

    @Test
    public void sumTest () {
        x[0] = 4;
        assertEquals(4, x[0]);
    }
}
```

Test Suites

Suppose you have a lot of separate test classes for each piece of your program. How do you run all of them at the same time?

A test suite is just a list of test classes which all get run together. Test suites require different imports from test classes and test fixtures. To write a test suite, start with an empty Java class, and put the following above the class definition:

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestClass1.class,
    TestClass2.class,
    ...
})
```

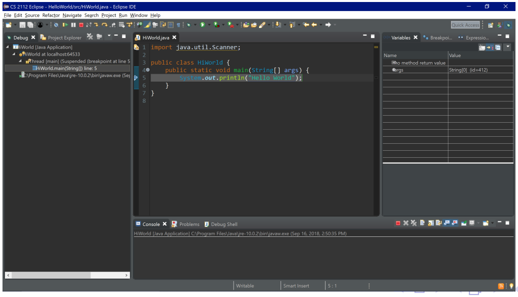
Example: BasicTestSuite

```java
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    BasicTestClass.class,
    IgnoredTestClass.class,
    BasicTestFixture.class
})

public class BasicTestSuite {}
```

In addition to running test classes and test fixtures, test suites can also run other test suites.

Debug View

You can enter debug view by clicking the bug icon next to Run.
Debug view allows you to add breakpoints, step through individual lines of code, inspect variable contents, and test expressions to see their results.
To exit debug view, choose Window → Perspective → Open Perspective → Java



```
HiWorld [Java Application]
  HiWorld at localhost:64533
    Thread [main] (Suspended (breakpoint at line 5
      HiWorld.main(String[]) line: 5
  C:\Program Files\Java\jre-10.0.2\bin\javaw.exe (Sep
```

```
1 import java.util.Scanner;
2
3 public class HiWorld {
4     public static void main(String[] args) {
5         System.out.println("Hello World");
6     }
7 }
8
```

Name	Value
no method return value	
args	String[0] (id=412)

```
HiWorld [Java Application] C:\Program Files\Java\jre-10.0.2\bin\javaw.exe (Sep 16, 2018, 2:50:35 PM)
```

Using Debug View

Double click to the left of a line number to set or unset a breakpoint. Execution will pause when it reaches a breakpoint.

Use the “Variables” pane to inspect the values of variables in scope.

Use the step buttons left of the run button to move through your code. Step Into will go into the highlighted method call, Step Over will run the selected method and pause on the next line, and Step Return will run until the current method has returned.

Use the “Expressions” pane to test the value of various expressions.

Exercise

Download Lab02.java from the course website.

The `Room` class represents a classroom that tracks the students inside for contact tracing reasons.

- Fill in the missing Javadoc for two methods
- Write black-box and glass-box test cases
- Debug and fix the class