Meme Credit: Twitter user @PinkFreudHokie

Lab 2: Javadoc, JUnit, & I/O

CS 2112 Fall 2025

September 8 / 10, 2025

Enable Assertions

- Choose Run → Edit Configurations
- Click "Edit configuration templates"
- Choose "Application" then "Modify options"
- Click "Add VM options"
- Type "-ea" into the VM arguments field and click OK

Javadoc Overview

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

Doc Comments

Doc comments start with `/**` and end with `*/`.

```
/**
 * 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 signature
  - Bad: The first argument is an integer
- Document preconditions and invariants
- If helpful, use `@param` and `@return` for organization
- Cover edge cases

Writing Good Docs

The ultimate goal is to provide useful, non-obvious information

```java
// set x to 3
int x = 3
```

Comments like the above are useless and do not add value.
It's common to make the mistake that more documentation is better documentation, but it's easy to write a lot of words that don't say anything. Even Java's own docs do this sometimes - notice how the linked method specs do not explain any edge cases at all or how to use the method.
Documentation is written for people to read. Don't waste people's time.

Javadoc and IntelliJ

- IntelliJ autocompletes Javadoc comments if the method signature is already written
- Type in `/**`, then `Enter` to generate a Javadoc template
- Hover over a method to see its Javadoc in the tooltip

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.

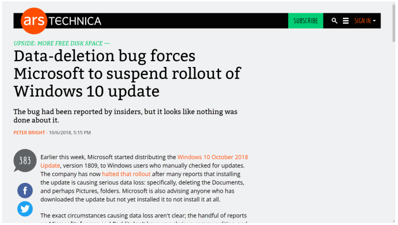
This way, if a test fails, you know exactly what is broken.

- 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?



UPSIDE: MORE FREE DISK SPACE —

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

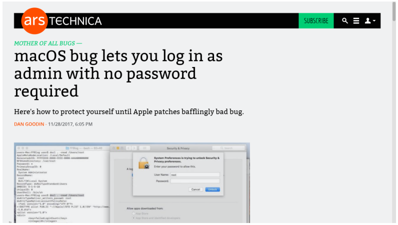
The bug had been reported by insiders, but it looks like nothing was done about it.

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

383

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

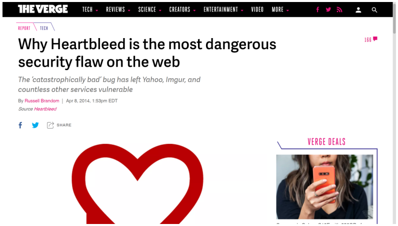


MOTHER OF ALL BUGS —

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



REPORT \ TECH

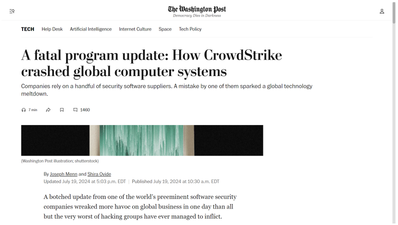
Why Heartbleed is the most dangerous security flaw on the web

166

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

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

Source Heartbleed



TECH Help Desk Artificial Intelligence Internet Culture Space Tech Policy

A fatal program update: How CrowdStrike crashed global computer systems

Companies rely on a handful of security software suppliers. A mistake by one of them sparked a global technology meltdown.

7 min 1460

(Washington Post illustration; shutterstock)

By Joseph Menn and Shira Ovide
Updated July 19, 2024 at 5:03 p.m. EDT | Published July 19, 2024 at 10:30 a.m. EDT

A botched update from one of the world's preeminent software security companies wreaked more havoc on global business in one day than all but the very worst of hacking groups have ever managed to inflict.

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 it is frequently skipped or put off
  - Automated test cases can be run automatically and as frequently as needed

Edge Cases

When writing tests, ensure that you not only cover the common case, but also have ample coverage of corner / edge cases, which is where most bugs occur.
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.

Black 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.

```java
/**
 * Compute whether one date occurs before another
 * @param d1 The first date
 * @param d2 The second date
 * @return Whether d1 occurs on or before d2
 */
public boolean dateComp( Date d1, Date d2 )
```

In this example, valid black box tests would check cases where the first date occurs before the second, when it occurs after the second, and also when they appear on the same day (edge case).

Black Box Exercise

Given the following method, what cases would you test?

```java
/**
 * Finds the index of target inside a sorted array arr.
 *
 * Requires: arr be sorted, target be present
 *
 * @param   arr      Sorted array to search
 * @param   target   Value to search for
 * @return  index such that arr[index] == target
 */
public int findIndex(int[] arr, int target);
```

White Box Testing

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.

```java
/**
* Compute whether one date occurs on or before... etc.
*/
public boolean dateComp( Date d1, Date d2 ) {
  if (d1.year != d2.year) { return d1.year < d2.year; }
  if (d1.month != d2.month) {
    return d1.month < d2.month;
  }
  return d1.day <= d2.day;
}
```

In this example, glass box tests would deliberately test dates with different years, then different months, then different days to ensure each branch functions correctly.

Glass Box Exercise

Given the same method, what cases would you add?

```java
/** Finds the index of target inside a sorted array arr.
 *  Requires: arr be sorted, target be present */
public int findIndex(int[] arr, int target) {
        int mid = arr.length / 2;
        if (target < arr[mid]) {
                for (int i = 0; i < mid; i++) {
                        if (arr[i] == target) {
                                return i;
                        }
                }
        } else {
                for (int i = arr.length - 1; i > mid; i--) {
                        if (arr[i] == target) {
                                return i;
                        }
                }
        }
        return -1;
}
```

Glass Box vs. Spec

It may be tempting on the previous slide to write a test case that an array missing the target would return -1, but that may not be a good idea!
A glass box test looks at the implementation as a way to target test cases towards high risk code paths (like different if/else branches, or loop boundaries). The tests it produces should still comply with the spec. This way, if the implementation changes in the future, even if the changes makes the glass box tests redundant, they won't fail. This creates less work for the programmer, and also means the existing test suite can be used to help ensure the new implementation is still correct.

IntelliJ: Creating a Test Class

Setting up JUnit with IntelliJ is fairly simple. IntelliJ should have come with JUnit support.

After creating a project, place the caret at the class that you want to create a test for, then press **Alt + Enter/Opt + Enter**, and click **Create Test**. Click **OK** when it asks whether to create test in the same source root. If JUnit5 library not found in module, click **Fix**.

Note that there are options in the dialog for automatically generating test stubs for existing methods.

In the test file, if there is red text, hover over it, and select **Add library to classpath**.

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

Basic Test Case

```java
@Test
void basicTest() {
    Calculator calc = new Calculator();
    assertEquals(4, calc.multTwo(2));
}
```

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 `assertEquals`. Pass the expected value first and the actual test second into `assertEquals`.

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

```java
@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

```java
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

- Test fixtures are used to prevent repetition of setup/cleanup code.
- Uses `@BeforeEach`, `@AfterEach`, `@BeforeAll`, and `@AfterAll` annotations.
- Class members are used to store environment variables and make them available to the tests.

Example: BasicTestFixture

```java
import org.junit.jupiter.api.BeforeEach;

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.

I/O Handout

A detailed reference on I/O can be found in the I/O handout on the course webpage:
https://courses.cs.cornell.edu/cs2112/2025fa/handouts/IO.pdf

Paths

A path represents the location of a file, typically on your computer.

e.g.,: `C:\Users\andru\Documents\CS 2112\Lab 2.tex`

Types of Paths

There are two types of paths: absolute and relative.

Absolute Paths

- Starting at root, full path of file
- Usually only works on your machine
- e.g.,: `C:\Users\andru\Documents \CS 2112\Lab 2.tex`

Relative Paths

- Relative to current directory
- In IntelliJ, project folder
- Typically used when programming
- e.g.,: `Documents\CS 2112\Lab 2.tex` (if we're in the `andru` directory)

Using Paths in Java

You can call `Paths.get(...)` with a relative path to acquire a `Path` object, which represents the location of a file.

```java
Path p = Paths.get("res", "map1.xml");
```

The above code returns a reference to the relative path `res/map1.xml`.
Note you can seperate directories as separate arguments, or pass an entire relative path in.

Files

Once you have a path to a file, Java provides many methods that allow you to operate on it, listed under the `Files` class.

eg: `exists(Path p)`, `isReadable(Path p)`, `createFile(Path p)`, `delete(Path p)`, `isWritable(Path p)`, `size(Path p)`, and more.

Check the official documentation for more:
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/nio/file/Files.html

Streams

A **stream** is a sequence of data being processed (read / written) from beginning to end.
Input streams are data coming into a program (for example, reading from a file).
Output streams are data leaving a program (for example, writing to a file).

Types of Streams

- Byte Stream
- Character Stream
- Raw Stream
- Blocking Stream
- Buffered Stream
- NIO Stream
- Object Stream
- etc.

Basic Streams

Reads one byte at a time.

```java
InputStream is = Files.newInputStream(p);
is.read(); // Gets the next byte in the file
```

We can use a **Buffered Stream** to get more than one byte at a time, for convenience.
Remember to always **close** a stream when finished working with it.

Buffered Readers

```
InputStream is = Files.newInputStream(p);
BufferedReader br = new BufferedReader(is);
                    // or
BufferedReader br = Files.newBufferedReader(p);

// read whole line (or null if empty)
String s = br.readLine();
br.close(); // close stream
```

Buffered Writers

```java
BufferedWriter bw = Files.newBufferedWriter(p);
// Overwrites p if exists, creates if not

bw.write("..."); // No newline
bw.close(); // Don't forget
```

Use a `PrintWriter` to write non-String objects and get additional methods.

```java
PrintWriter pw =
    new PrintWriter(Files.newBufferedWriter(p));
pw.println(6); // Includes newline
```

Standard Streams

Your OS provides every program with three “standard” I/O streams. These streams have defaults, but can be changed per program. For example, a user may want to redirect standard error into a log file instead of showing it in the console.

Standard Input: What the user types into your program, typically in the console.

Standard Output: What your program shows to the user, typically in the console.

Standard Error: Error messages from your program, typically in red in the console.

Standard Streams in Java

Java exposes each of the standard streams to the programmer as fields in the `System` class: `System.in`, `System.out`, and `System.err`.

Standard input is an `InputStream`, and the other two are `PrintWriter`.

Thus, `System.out.println("")` is calling the `println("")` method on a `PrintWriter` that just happens to be standard output.

Character Encoding

Character encoding defines how characters we recognize get stored to disk as individual bytes.

For this class, use Unicode **UTF-8**.

I/O Exercise

Linked under today's lab on the schedule is `FileCopier.java`. This class has a single static method that copies a file from one location to another.

Write test cases for this method. At least one test case should ensure that the copied output has the exact same bytes as the original file.

(Hint: you may find this code helpful when testing RSA in A1)

Feel free to reference the IO handout:

https://courses.cs.cornell.edu/cs2112/2025fa/handouts/IO.pdf