LEC 16

JUnit Testing

Questions during Class?

Raise hand or send here

sli.do #cse122

BEFORE WE START

Slido vote & chat with neighbors:

What is your favorite bug (insect or otherwise) and why?

Music: 122 26sp Lecture Jams 🎧

Instructor: Elba Garza

TAs:

David	Caleb	Cole	Yang
William	Neha	Blake R.	Cady
Dani	Wesley	Carson	Diya
Rohan	Isis	Sushma	
Andrew	Colin	Connor	
Ava	Naomi	Mahima	
Shreyank	Hanna	Nicolae	
Nicole	Blake P.	Ivory	

Lecture Outline

- Announcements
- Importance of Testing
- JUnit
- Testing Tips

Announcements

- Tomorrow (Thursday, May 28): Programming Assignment 3 (P3) due!
- Resubmission 5 **extended deadline!**
  - Now due tonight, 11:59 PM
- Resubmission 6 opens tomorrow
- Creative Project 3 (C3) will be released Friday, May 29
- Final Exam: **Tuesday, June 9th 2:30-4:20 PM**
  - KNE 120 (here)
- JUnit “virtual section” for today’s content, linked from this Ed module

Lecture Outline

- Announcements
- **Importance of Testing**
- JUnit
- Testing Tips

Importance of Testing

Software, written by people, controls more and more of our day-to-day lives.

Bugs (just like the ones we all write) are just as easy to write in this software.

Stakes can be quite high so bugs can have catastrophic effects

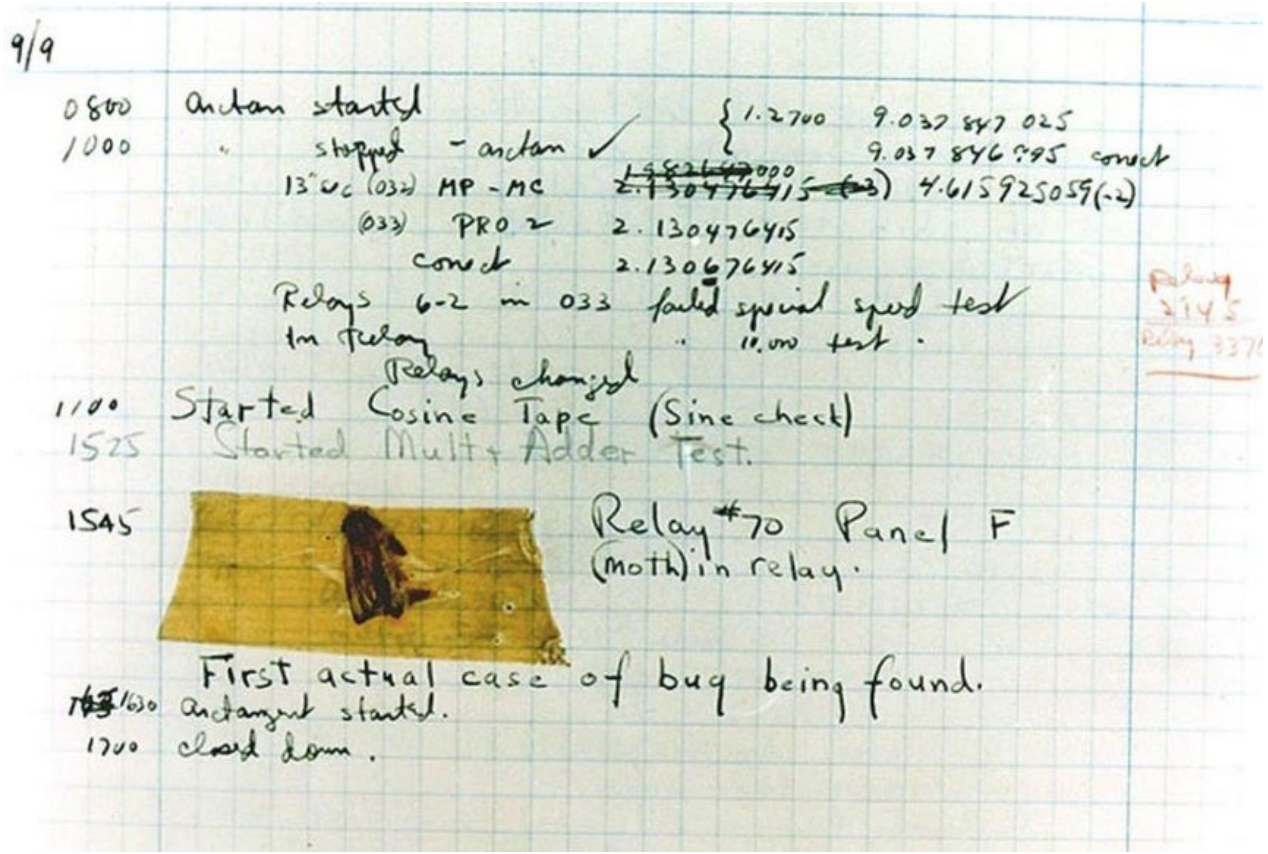
Source: Hackaday

The Horizon IT System for The UK Post Office

Source: Fujitsu.com

Fun Aside – what was the first computer bug?

- 1947 Harvard "computers" find moth trapped in the Mark II



Lecture Outline

- Announcements
- Importance of Testing
- JUnit
- Testing Tips

Using a Testing Framework

- `Unit Test` – a method that compares what your code does against what you *expect* it to do
- `Testing Framework` – a library of code that gives you special tags and key words for your unit tests so that you can click the "test" button instead of the "run" button and you get a list of tests with info like green check mark passes or error messages

Like a music tuner! Technology specifically built to compare what your instrument sounds like against what it's expected to sound like

Writing a Unit Test

- Break your code into cases, what should occur in each case?

```
public static int max(int x, int y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```

- **Specification Testing** (based on the spec) vs. **Clear-box Testing** (based on how you know your implementation works)
  - **Specification Testing** you can do *before* writing your solution! (Test Driven Development)

```
// Returns the maximum of two integers.
public static int max(int x, int y) {
```

  - **Clear-box Testing** you do *after* you've written your solution.

In-Class: Remix – Courses Taught

Practice : Pair

sli.do #cse122

What test cases can you test for the coursesTaught method?

Spend 1 minute brainstorming specification testing

Then 1 minute brainstorming clear-box testing

JUnit Basics

- `JUnit` – a unit testing framework for the Java language
  - `import` statements to give you access to JUnit method annotations and assertion methods!
- Method Annotations
  - `@Test`
  - `@DisplayName`
  - …
- Assertion Methods
  - `assertEquals`
  - `assertTrue`
  - `assertFalse`
  - …

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

public class ArrayListTest {
    @Test
    @DisplayName("Add works alongside Get")
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add("Nicolae");
        list.add("Isis");
        list.add("Shreyank");

        assertEquals("Isis", list.get(1));
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    @DisplayName(“Add works alongside Get")
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add(“Nicolae");
        list.add(“Isis");
        list.add(“Shreyank");

        assertEquals(“Isis", list.get(1));
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    @DisplayName(“Add works alongside Get")
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add(“Nicolae");
        list.add(“Isis");
        list.add(“Shreyank");

        assertEquals(“Isis", list.get(1));       // TRUE!!
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    @DisplayName(“Add works alongside Get")
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add(“Nicolae");
        list.add(“Isis");
        list.add(“Shreyank");

        assertEquals(“Isis", list.get(1));
        assertEquals(“Nicolae", list.get(2));
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    @DisplayName(“Add works alongside Get")
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add(“Nicolae");
        list.add(“Isis");
        list.add(“Shreyank");

        assertEquals(“Isis", list.get(1));
        assertEquals(“Nicolae", list.get(2));            // FALSE!!
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    @DisplayName(“Add works alongside Get")
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add(“Nicolae");
        list.add(“Isis");
        list.add(“Shreyank");

        assertEquals(“Isis", list.get(1));
        assertEquals(“Shreyank", list.get(2));
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    @DisplayName(“Add works alongside Get")
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add(“Nicolae");
        list.add(“Isis");
        list.add(“Shreyank");

        assertEquals(“Isis", list.get(1));
        assertEquals(“Shreyank", list.get(2));          // TRUE!!!
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    @DisplayName(“Add works alongside Get")
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add(“Nicolae");
        list.add(“Isis");
        list.add(“Shreyank");

        assertEquals(“Isis", list.get(1));
        assertEquals(“Shreyank", list.get(2));

        assertTrue(list.size() == 3);
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    @DisplayName(“Add works alongside Get")
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add(“Nicolae");
        list.add(“Isis");
        list.add(“Shreyank");

        assertEquals(“Isis", list.get(1));
        assertEquals(“Shreyank", list.get(2));

        assertTrue(list.size() == 3);            // TRUE!!
    }
}
```

In-Class: Remix – Courses Taught (Testing time!)

Lecture Outline

- Announcements
- Importance of Testing
- JUnit
- **Testing Tips**

Testing Tips

- Write many tests for each method
  - Test that your method does what you want it to do
  - Test combinations of your method being used with other methods
- Use `assertEquals(expected, actual, message)` to provide a description of what case that line is testing
- Testing code is just code. Use good coding practices (e.g., helper methods to reduce redundancy) to help you write code.
  - It can take time, but if you do it well, developing your solution can be a breeze!

How Many Test Cases Is Enough?

- In general, more *diverse* tests –> more confidence!
- Test a wide variety of different cases
  - Think about **boundary or "edge" cases** in particular, where the behavior should change

```
public static int max(int x, int y) {
    if (x >= y) {
        return x;
    } else {
        return y;
    }
}
```

  - Empty objects, null, negative numbers
- Try to think adversarially and try to break your own code with tests, how do you “user-proof” your code?