LEC 16

CSE 122

JUnit Testing

Questions during Class?

Raise hand or send here

sli.do #cse122

BEFORE WE START

Talk to your neighbors:

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

Music: 122 26Wi Lecture Tunes

Instructor: Adrian Salguero

TAs:

Ava	Dalton	Neal	Shreyank
Blake R	Dani	Neha	Sthiti
Blake P	David	Nicolae	Sushma
Cady	Diya	Nicole	Suyash
Caleb	Hanna	Rio	TJ
Cole	Ivy	Rohan	Wesley
Colin	Mahima	Saachi	Yang
Connor	Medha	Shreya	

Lecture Outline

- **Announcements**
- Importance of Testing
- JUnit
  - How Much Testing is Enough?
- Example: Brainstorm Test Cases (Card & BattleManager)
- Challenge: Floating Point Precision

Announcements

- Tomorrow (Thursday, March 5): Programming Assignment 3 (P3) due!
- Resubmission 6 opens tomorrow
  - Eligible: P2 and C2
- Creative Project 3 (C3) will be released Friday, March 6
- Final Exam: **Tuesday, March 17th 12:30-2:20**
  - **Section A:** KNE 130
  - **Section B:** KNE 220
- JUnit "virtual section" for today's content, linked from this Ed module

Lecture Outline

- Announcements
- **Importance of Testing**
- JUnit
  - How Much Testing is Enough?
- Example: Brainstorm Test Cases (Card & BattleManager)
- Challenge: Floating Point Precision

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

Practice : Pair

sli.do #cse122

Bugs you've experienced

Can you think of a bug(s) you've experienced or heard of that have had serious effects?

If you can't, can you think of any absurd bugs you've seen?

Lecture Outline

- Announcements
- Importance of Testing
- **JUnit**
  - How Much Testing is Enough?
- Example: Brainstorm Test Cases (Card & BattleManager)
- Challenge: Floating Point Precision

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

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`
  - …

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

public class ArrayListTest {
    @Test
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add("Nicole");
        list.add("Dani");
        list.add("Cady");

        assertEquals("Nicole", list.get(0));
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add("Nicole");
        list.add("Dani");
        list.add("Cady");

        assertEquals("Nicole", list.get(0));
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add("Nicole");
        list.add("Dani");
        list.add("Cady");

        assertEquals("Nicole", list.get(0));       // TRUE!!
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add("Nicole");
        list.add("Dani");
        list.add("Cady");

        assertEquals("Nicole", list.get(0));
        assertEquals("Dani", list.get(2));
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add("Nicole");
        list.add("Dani");
        list.add("Cady");

        assertEquals("Nicole", list.get(0));
        assertEquals("Dani", 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
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add("Nicole");
        list.add("Dani");
        list.add("Cady");

        assertEquals("Nicole", list.get(0));
        assertEquals("Cady", list.get(2));
    }
}
```

JUnit Testing

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

public class ArrayListTest {
    @Test
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add("Nicole");
        list.add("Dani");
        list.add("Cady");

        assertEquals("Nicole", list.get(0));
        assertEquals("Cady", 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
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add("Nicole");
        list.add("Dani");
        list.add("Cady");

        assertEquals("Nicole", list.get(0));
        assertEquals("Cady", 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
    public void testAddAndGet() {
        List<String> list = new ArrayList<>();
        list.add("Nicole");
        list.add("Dani");
        list.add("Cady");

        assertEquals("Nicole", list.get(0));
        assertEquals("Cady", list.get(2));

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

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
- Write a test method per distinct case
  - Test that different states of input don't break your code (empty or null params)
  - Test that code correctly enters all boolean checks (loops, if/else)
- 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!

In-Class: Remix – Courses Taught

How Many Test Cases Is Enough?

- In general, more *diverse* tests –> more confidence!
- Try to think adversarially and try to break your own code with tests, How do you "user-proof" your code?
- **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)
  - **Clear-box Testing** you do *after* you've written your solution.
- Test a wide variety of different cases
  - Think about **boundary or "edge" cases** in particular, where the behavior should change

Lecture Outline

- Announcements
- Importance of Testing
- JUnit
  - How Much Testing is Enough?
- **Example: Brainstorm Test Cases (Card & BattleManager)**
- Challenge: Floating Point Precision

Card Class

- Each card has a suit (♠ ♣ ♦ ♥) and a value (e.g., 2, 3, 10, J, Q, K)
  - Note: value represented as an `int`

Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King
1	2	3	4	5	6	7	8	9	10	11	12	13

  - For example, for the Queen of Hearts card
    - The suit is hearts ♥
    - The value is Queen (represented as 12)

Card Class

- `public Card(int value, String suit)`
  - Throws an `IllegalArgumentException` if `value` or `suit` is invalid
- `public String getSuit()`
- `public int getValue()`
- `public String toString()`
- `public boolean equals(Object other)`

BattleManager Class

- Assumes two players
- Setup: 52-card deck is split between the two players evenly
- Each round:
  - Each player flips their top card
  - The player with the higher *value* card takes both cards
    - Aces are considered "high" – they beat all other values
  - If the cards have the same value, "battle"
    - Each player places 3 cards face down, then flips a new card, and the player with the higher value card takes all cards
      - (If this is another battle, repeat previous process)
- Goal: one player has all 52 cards

BattleManager Class

- `public BattleManager()`
- `public BattleManager(Queue<Card> deck1, Queue<Card> deck2)`
- `public void deal()`
- `public boolean gameOver()`
- `public int getPlayer1DeckSize()`
- `public int getPlayer2DeckSize()`
- `public void play()`

Practice : Pair

sli.do #cse122

What test cases can you test for the Card class?
What about the BattleManager class?

Spend 2 minutes brainstorming specification testing

Then 2 minutes brainstorming clear-box testing

Lecture Outline

- Announcements
- Importance of Testing
- JUnit
  - How Much Testing is Enough?
- Example: Brainstorm Test Cases (Card & BattleManager)
- **Challenge: Floating Point Precision**

Challenge: Floating Point Numbers

- Another name for `doubles` are floating point numbers
- Floating point numbers are nice, but imprecise
  - Computers can only store a certain amount of precision (can't store 0.3333333333 repeating forever)
  - Finite precision can lead to slightly incorrect calculations with floating point numbers

```
0.7 + 0.1
0.7999999999999999
```

- Take-away: Essentially can never rely on == for doubles. Instead, must define some notion of how far away they can be to be tolerated as the same
  - JUnit: `assertEquals(expected, actual, delta)`