

LEC 16

CSE 122

JUnit Testing

BEFORE WE START

Talk to your neighbors:

*What is your favorite “classic” mobile game?
(e.g. candy crush, temple run, fruit ninja, 2048, etc)*

Instructor Melissa Lin

TAs	Poojitha Arangam	Audrey Lin
	Darel Gunawan	Di Mao
	Colton Harris	Steven Nguyen
	Atharva Kashyap	Ben Wang
	Eesha Kunisetty	Jaylyn Zhang


Questions during Class?

Raise hand or send here

sli.do #cse122



Lecture Outline

- **Announcements** 
- Optional
- Importance of Testing
- JUnit
- Example: Tic Tac Toe

Announcements

- Final Exam on Wed + Fri @ 10:50 – 11:50am PCAR 192
 - Review session on Monday Aug 14
 - More resources/policies on [course website](#)
- Programming Assignment 3 due Sunday, Aug 13
- Resub 6 due Tuesday, Aug 15

Exam Format

- 6 questions in total, each will receive one ESN grade
 - Some questions might have sub-parts
 - Reminder: Quiz and Exam grades are all mixed into the same bucket
- General format
 - 3 Questions: Mix of Conceptual, Mechanical/Tracing, Debugging Problems
 - 3 Questions: Programming Problems
 - Wednesday – 3 questions
 - Friday – 3 questions
- See sections 11, 13, and 14 for practice handwriting problems
- Practice finals posted on course website

Exam Logistics

Most important bits

- Wednesday and Friday @ 10:50am – 11:50am in PCAR 192
- Seat assignments
- Don't cheat
 - Only have the exam open during the time (don't start early; don't work after)
 - No electronic devices
- You can bring one 8.5x11 inch paper with notes (front and back)
 - Will also provide a reference sheet (see course website)
- Bring husky card + pencil

Questions? Raise hand or ask on sli.do (#cse122)

Lecture Outline

- Announcements
- **Optional** ◀
- Importance of Testing
- JUnit
- Example: Tic Tac Toe

Optional

`Optional` is a Java class that is used to handle situations where a value is *sometimes* there.

You give `Optional` a type to hold (or potentially not hold) when you are referring to its type.

e.g., `Optional<String>`, `Optional<Integer>`, `Optional<Point>`

Optional Methods

Method	Description
<code>Optional.empty()</code>	Creates an empty <code>Optional</code> object
<code>Optional.of(...)</code>	Creates an <code>Optional</code> object holding the object it's given
<code>isEmpty()</code>	Returns true if there <i>is no</i> value stored, and false otherwise
<code>isPresent()</code>	Returns true if there <i>is a</i> value stored, and false otherwise
<code>get()</code>	Returns the stored object from the <code>Optional</code> (if one is stored; otherwise throws a <code>NoSuchElementException</code>)

The `Optional` class has more than just these methods, but these are what you'll need to focus on for this class!


Optional Methods

`isEmpty()`, `isPresent()`, and `get()` are called like normal instance methods (on an actual instance of `Optional`).

`Optional.of(...)` and `Optional.empty()` are called differently

(Like the `Math` class methods)

Lecture Outline

- Announcements
- Optional
- **Importance of Testing** 
- JUnit
- Example: Tic Tac Toe

(PCM) 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





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
- Optional
- Importance of Testing
- **JUnit** 
- Example: Tic Tac Toe

JUnit Basics

- `import` statements to give you access to JUnit method annotations and assertion methods!
- Method Annotations
 - `@Test`
 - `@DisplayName`
 - ...
- Assertion Methods
 - `assertEquals(expected, actual)`
 - `assertTrue(boolean)`
 - `assertFalse(boolean)`
 - ...

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("Hunter Schafer");
```

```
        list.add("Miya Natsuhara");
```

```
        list.add("CSE 122");
```

```
        assertEquals("Hunter Schafer", list.get(0));
```

```
        assertEquals("Miya Natsuhara", list.get(1));
```

```
        assertEquals("CSE 122", list.get(2));
```

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

```
    }
```

```
}
```

} put object into some expected state

} Use assert statements to check if observed state is what we expect

Using JUnit

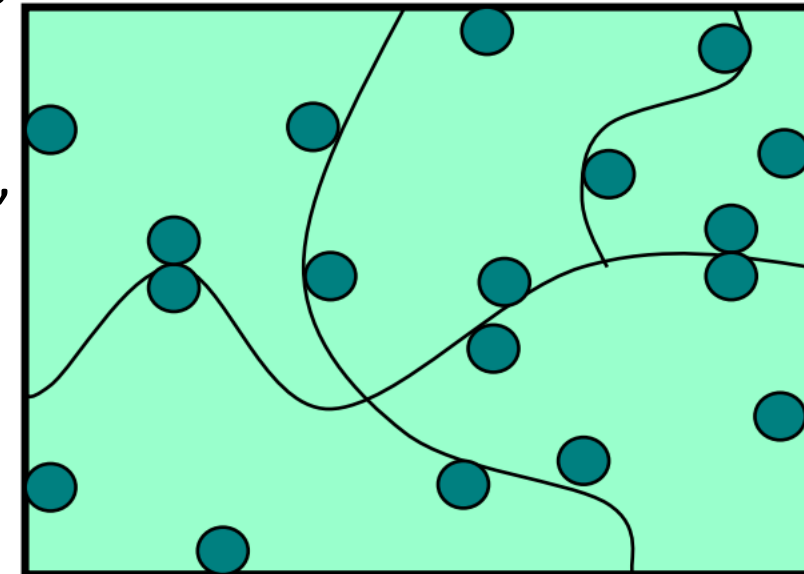
- Each `@test` method should be independent
 - ie. set up its own state, make all relevant assertions
- An `@test` fails if any `assert` statement fails
- JUnit executes `@test` methods in an arbitrary order

Using JUnit - Tips


- one `@test` method per distinct case (i.e., empty case, one element, even, odd, some edge case, ...)
 - Might also want to test calling multiple methods to check that they work together as expected
- `assertEquals(expected, actual, message)` can provide a description of what the line is testing
- Good coding practices still apply
 - Eg. you can write helper methods in your test file

(PCM) How Many Test Cases Is Enough?

- In general, more tests → more confidence!
- Try to think adversarially and try to break your own code with tests
- 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!
 - 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
- Optional
- Importance of Testing
- JUnit
- **Example: Tic Tac Toe** 



Practice : Pair

sli.do[#cse122](https://twitter.com/cse122)

What test cases can you think of for the TicTacToe spec?

Closed or open box tests?

Closed box testing - write tests based on a **specification** independent of any implementation.

Open box testing - write tests for a particular implementation.

Test Driven Development - write tests *before* the implementation

Part B: Consider the following method. For each of the following commented Points, fill in the table for which conditions are always true (under any circumstance), only sometimes true, or never true at each comment. You can abbreviate **A**=always, **S**=sometimes and **N**=never.

```
public Set<String> mystery(Set<String> s, int n) {
    Set<String> s2 = new HashSet<>();
    // Point A
    if (n <= 0) {
        throw new IllegalArgumentException();
    }
    // Point B
    if (!s.isEmpty()) {
        Iterator<String> it = s.iterator();
        // Point C
        while (it.hasNext()) {
            // Point D
            if (it.next().length() > n) {
                s2.add(it.remove());
                n--;
            }
        }
    }
    // Point E
    return s2;
}
```

An explanation for Point A's answers

s.isEmpty(): S
True for `mystery([], 3)` and false for `mystery(["hi"], 3)`. Therefore sometimes true.

s2.isEmpty(): A
`s2` has just been created with no elements and no other methods have been called on it.

n > 0: S
True for `mystery(["hi"], 3)` and false for `mystery(["hi"], -19)`. Therefore sometimes true.

	Point A	Point B	Point C	Point D	Point E
<code>s.isEmpty()</code>	S				
<code>s2.isEmpty()</code>	A				
<code>n > 0</code>	S				