Talk to your neighbors:
What plans do you have for after this course ends?

Instructor: Ido Avnon
TAs: Abby Williams, Chloë Mi Cartier, Connor Sun, Cynthia Pan, Katharine Zhang, Marcus Sanches, Rohini Arangam
Lecture Outline

• Announcements

• Optional

• Collections

• Importance of Testing/Bugs

• JUnit

• Example
Announcements

• Programming Assignment 3 (P3) due tomorrow!
  - Due August 8th by 11:59 PM
• Quiz 2 TOMORROW in section!
• Reminder on Final Exam
  - Part 1: In section August 15th
  - Part 2: in lecture August 16th
Final Exam Part 2 (Friday August 16th)

- Slightly longer quiz (60 minutes, 4 questions)
- In lecture (this classroom PCCAR 192)
  - Please come 5-10 minutes early
  - Bring: pencil, ID card, notes
- Unlimited paper notes
- Paper exam, no electronics
- Resources: out Friday
Final Exam Part 1

• NOT A PAPER EXAM!
• A one-on-one presentation of your culminating project with your TA in section!
  - Run-through of your code
  - Reflection on your code and process
• Worth two ESN grades (given by the TA you present to)
• More info and resources: Friday!
Lecture Outline

• Announcements

• Optional

• Collections

• Importance of Testing/Bugs

• JUnit

• Example
Optional

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

- A variable that can sometimes be initialized
- Optional<String> keepPlaying = Optional.empty();
- Optional<Integer> maxValue = Optional.of(-1);

Like a collection, Optional uses <> to denote the type it contains..
- e.g., Optional<String>, Optional<Integer>, Optional<Point>
Optional Methods

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

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)
Why Optional?

Using Optional can help programmers avoid NullPointerExceptions by making it explicit when a variable may or may not contain a value.

• Remember – null refers to the absence of an object!

There are other Optional methods (that you should explore in your own time if you’re interested) that can be really useful to cleanly work with data that may or may not be present.
Lecture Outline

• Announcements

• Optional

• **Collections**

• Importance of Testing/Bugs

• JUnit

• Example
Collections: What *classes* have we seen so far?

... 

Array, ArrayList, LinkedList, Stack, 
HashSet & HashMap, TreeSet & TreeMap
Collections: What *interfaces* have we seen so far?

... 
Set, 
Queue, 
List, 
Comparable
Java Collection

- An *extremely* general interface that *every data structure* we have talked about indirectly implements

- Methods in the interface
  - add
  - remove
  - contains
  - isEmpty
  - size
  - And more...

- Map’s `values()` method returns a Collection !!!
Lecture Outline

• Announcements

• Optional

• Collections

• Importance of Testing/Bugs

• JUnit

• Example
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.
A - What bugs have you experienced?

Start presenting to display the poll results on this slide.
Lecture Outline

• Announcements
• Optional
• Collections
• Importance of Testing/Bugs
• JUnit
• Example
Using a Testing Framework

• Unit Test – a method that compares what your codes 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`
  - ...

JUnit Testing

```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("Ido Avnon");
        list.add("And his amazing TAs");
        list.add("CSE 122");

        assertEquals("Ido Avnon", list.get(0));
    }
}
```
JUnit Testing

```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("Ido Avnon");
        list.add("And his amazing TAs");
        list.add("CSE 122");

        assertEquals("Ido Avnon", 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("Ido Avnon");
        list.add("And his amazing TAs");
        list.add("CSE 122");

        assertEquals("Ido Avnon", list.get(0));
        assertEquals("And his amazing TAs", list.get(2));
    }
}
JUnit Testing

```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("Ido Avnon");
        list.add("And his amazing TAs");
        list.add("CSE 122");

        assertEquals("Ido Avnon", list.get(0));
        assertEquals("And his amazing TAs", list.get(2)); //FALSE
    }
}
```
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!
How Many Test Cases Is Enough?

- In general, more *diverse* tests $\rightarrow$ 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
• Optional
• Collections
• Importance of Testing/Bugs
• JUnit
• Example
Card Class

- Each card has a suit (♠️, ♣️, ♦️, ❤️) and a value (e.g., 2, 3, 10, J, Q, K)
  - Note: value represented as an int
  
<table>
<thead>
<tr>
<th>Ace</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>Jack</th>
<th>Queen</th>
<th>King</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

- 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()
What test cases can you think of for the Card class?
What test cases can you think of for the BattleManager class?
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)