Even More Exceptions, More JUnit

(All interleaved)

How To Do Good Testing

- Unit Tests should test a single unit of code at a time.
- Unit Tests should typically have one or few assertions per test
    - Otherwise, your test is probably testing more than one unit of code!
- Make sure your unit tests have inputs that test all possible outputs.
    - Black box view
- Make sure your unit tests have inputs that test all possible paths that your code can take
    - White box view

What kinds of test can you write?

- Basic assertions:
  - `assertEquals([message], expected, actual)`
    - Check that two inputs are equal
  - `assertTrue([message], expression)`
    - Check that some boolean expression is true
  - `assertNull([message], expression)`
    - Check that some value is null
  - `assertArrayEquals([message], expectedArr, actualArr)`
    - Check if two arrays have the same elements

What kinds of test can you write?

Does my code return the right value?

```
@Test
public void testBadStringMissingCIT() {
    String input = "001591CIS777776312345";
    boolean expected = false;
    boolean actual = PartTwo.isValidProductKey(input);
    assertEquals(expected, actual);
}
```

(From the Exam autograder)

What kinds of test can you write?

Does my code return a value that has a certain property?

```java
@Test
public void testRunExperiment100000Times() {
    double threshold = 0.04332;
    double error = Math.abs(result100000 - Math.PI);
    assertTrue(error < threshold);
}
```

(From the HW3 autograder)

What kinds of test can you write?

Does my code ever accidentally deal with a null value?

```java
@Test
public void testAllCardsInDeckInitialized() {
    Deck d = new Deck();
    while (!d.isEmpty()) {
        Card c = d.deal();
        assertNotNull(c);
    }
}
```

(From my HW4 testing. This is not a great test, BTW--way too many asserts per test!)

What kinds of test can you write?

Timeout Assertions: It's not just important that your code is correct; it should also run quickly.

```
import static java.time.Duration.ofMillis;

@Test
public void testCodeRunsQuickly() {
    String input = "Sorta long string for testing example";
    assertTimeout(ofMillis(3000), () -> {
        runExpensiveOperation(input);
    });
}
```

Passes when `runExpensiveOperation` finishes in under 3000ms.

What kinds of test can you write?

This test will let `runExpensiveOperation` go as long as it needs and tell you how long it took. Could be a problem with infinite loops...

```java
import static java.time.Duration.ofMillis;

@Test
public void testCodeRunsQuickly() {
    String input = "Sorta long string for testing example";
    assertTimeout(ofMillis(3000), () -> {
        runExpensiveOperation(input);
    });
}
```

What kinds of test can you write?

This test will let `runExpensiveOperation` go only for 3000ms and fail if it's not finished. Less information provided, but more robust in case of infinite loop.

```java
import static java.time.Duration.ofMillis;

@Test
public void testCodeRunsQuickly() {
    String input = "Sorta long string for testing example";
    assertTimeoutPreemptively(ofMillis(3000), () -> {
        runExpensiveOperation(input);
    });
}
```

What kinds of test can you write?

Throwing Assertions: testing to make sure that your code throws exceptions when you expect it to.

```
@Test
public void testCodeThrowsAnException() {
    Exception e = assertThrows(FileNotFoundException.class, () -> {
        FileInputStream fs = new FileInputStream("sdjflksjdlfksjdf.fssdf");
    });
}
```

What kinds of test can you write?

Once the exception is thrown, you can also test that it contains the right message, too:

```
@Test
public void testCodeThrowsAnException() {
    Exception e = assertThrows(FileNotFoundException.class, () -> {
        FileInputStream fs = new FileInputStream("sdjdf.ff");
    });
    assertEquals("sdjdf.ff (No such file or directory)", e.getMessage());
}
```

Virtues of Black Box Testing

Black Box Testing is the process of writing tests just based on inputs and outputs, not concerning yourself with how the unit of code is written.

- This lets you write tests before you write your code (TEST DRIVEN DEVELOPMENT)
  - 🚨 CIT 594 🚨
- Verifies that you actually know what your code is supposed to be doing

Drawbacks of Black Box Testing

It's very hard to know when you're done!

How do you know when you've covered every possible output? Every possible input? Is that ever even possible? (not really)

Alternative: Coverage-Based Testing

Once you've written your code, make sure you write enough tests so that every line of your code actually gets tested.

Testing Coverage is the measure of how many lines of your program are actually executed when running the tests you've written.

Example

```
public class Coverage {
        public static boolean m(int p) {
                if (p < 0) {
                        p = 1;
                        return p > 1;
                }
                return p == 1 || p == 9;
        }
        public static boolean mm() {
                return false;
        }
}
```

Borrowed from Cornell CIT 591/594 equivalent

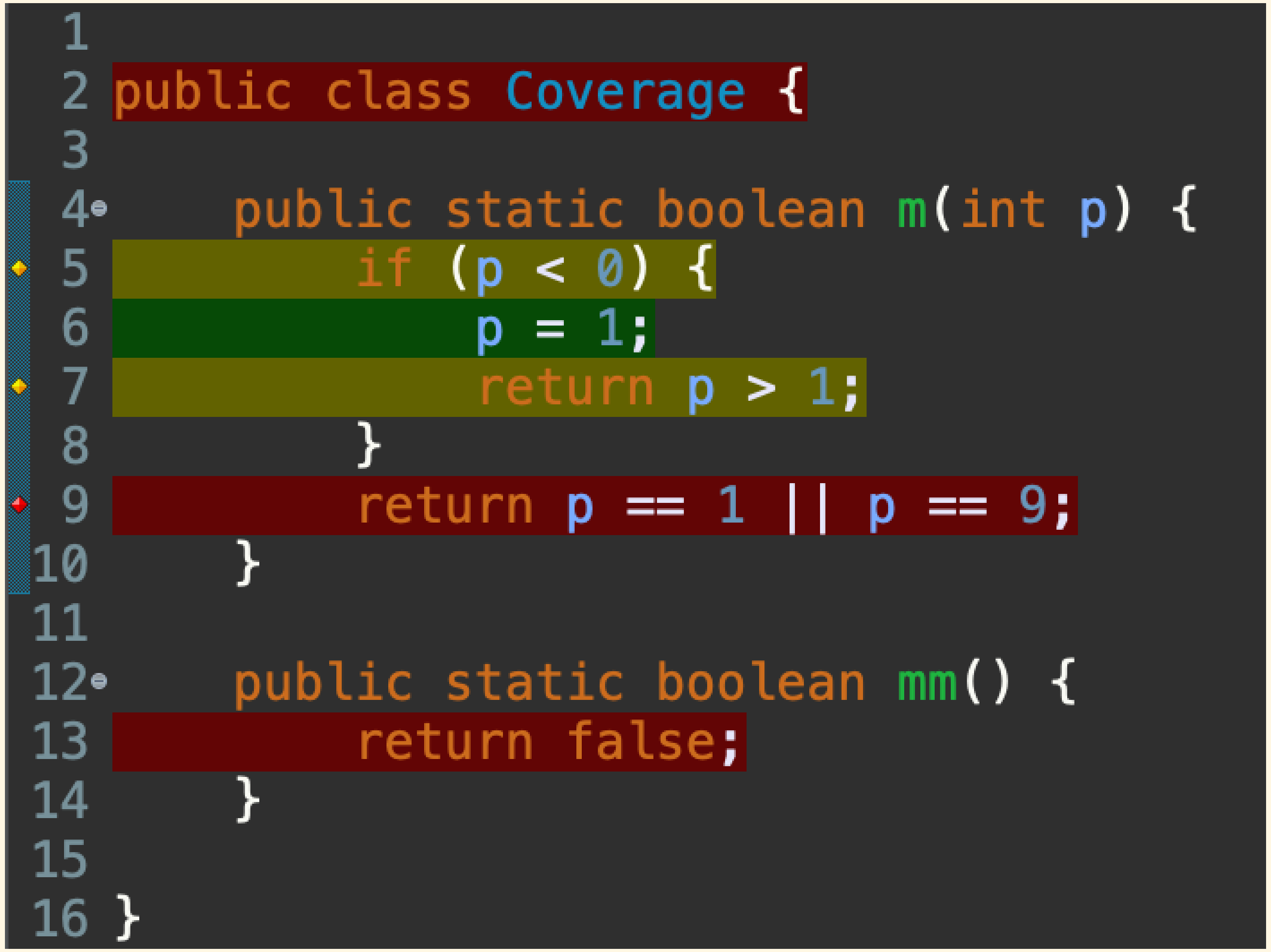
One Test Case

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

class CoverageTest {
    @Test
    void test() {
        assertEquals(false, Coverage.m(-1));
    }
}
```

One test is usually not enough, but with Eclipse Coverage Testing, we get an idea of how lacking it is!

Coverage



```

public class Coverage {

    public static boolean m(int p) {
        if (p < 0) {
            p = 1;
            return p > 1;
        }
        return p == 1 || p == 9;
    }

    public static boolean mm() {
        return false;
    }

}
```

Coverage

- Green:
    - This line of code is executed and all paths branching from it were started.
- Yellow:
    - This line of code is executed at least once, but not all paths branching from it were started.
- Red:
    - This line of code was never reached, and not all paths branching from it were started.

Activity: Cover this code!

```java
public class Coverage {
        public static boolean m(int p) {
                if (p < 0) {
                        p = 1;
                        return p > 1;
                }
                return p == 1 || p == 9;
        }
        public static boolean mm() {
                return false;
        }
}
```

Write tests that will cover this program entirely.

Activity: Cover this code!

`isValidProductKey` (see calendar tab of course website)

Now: HashMaps

Common problem: how to lookup some value based on another value (e.g. a student's grade based on their name)

Maps allow us to associate *keys* with *values*.

- keys are what you use to loopup values.
- not vice versa!

Maps at a High Level

I want to map State names (keys) to Populations (values).

So, perhaps:

```
CA --> 38332521
AZ --> 6626624
MA --> 6692824
```

> The state name "CA" maps to a population of 38,332,521.

Maps at a High Level

I want to map State names (keys) to Populations (values).

So, perhaps:

```
CA --> 38332521
AZ --> 6626624
MA --> 6692824
```

> The state name "PA" maps to ???.

If a key is missing, there's no associated value.

Maps at a High Level

I want to map State names (keys) to Populations (values).

So, perhaps:

```
CA --> 38332521
AZ --> 6626624
MA --> 6692824
CA --> 38349391
```

> The state name "CA" maps to ???

Repeating States doesn't make much sense.

Making a HashMap

```java
HashMap<K, V> mapping = new HashMap<K, V>();
```

Creates a new variable called `mapping` that stores a `HashMap<K, V>` (a mapping from keys of type `K` to values of type `V`). The initial value of the variable is set to be a new empty `HashMap<K, V>`.

e.g.,

```java
HashMap<String, Double> gradeMap = new HashMap<String, Double>();
```

Remember, No Primitive Types!

- `Integer`, not `int`
- `Double`, not `double`
- `Boolean`, not `boolean`
- `Character`, not `char`

Two Fundamental Operations of a HashMap

`put(K key, V value)` and `get(K key)`

- `put` introduces a new mapping from `key --> value`
- `get` returns the value that is mapped to the provided `key`

The Invariants of a HashMap

Keys are all unique!

- Any `HashMap` can only associate one particular key with one value.
- A missing key implicitly means that the key maps to the value `null`.
    - Careful!!

Keys are not stored in any particular order

- `ArrayLists` have values set in a particular order, accessible by indices.
- No such ordering for a `HashMap`. (There is for a `TreeMap`...)

Put and Get as Example

```java
HashMap<String, Integer> statePopulation = new HashMap<String, Integer>();

// 2013 population data
statePopulation.put("CA", 38332521);
statePopulation.put("AZ", 6626624);
statePopulation.put("MA", 6692824);

System.out.printf("Population of Arizona in 2013 is %d.%n", statePopulation.get("AZ"));
```

⬇️

```
Population of Arizona in 2013 is 6626624.
```

Replacing Keys

```java
HashMap<String, Integer> statePopulation = new HashMap<String, Integer>();
statePopulation.put("CA", 38332521);
statePopulation.put("AZ", 6626624);
statePopulation.put("MA", 6692824);

System.out.printf("Population of Arizona in 2013 is %d.%n", statePopulation.get("AZ"));

statePopulation.put("AZ", 6871809);

System.out.printf("Population of Arizona in 2014 is %d.%n", statePopulation.get("AZ"));
```

⬇

```
Population of Arizona in 2013 is 6626624.
Population of Arizona in 2014 is 6871809.
```

Getting Keys That Aren't Present

```java
HashMap<String, Integer> statePopulation = new HashMap<String, Integer>();

// 2013 population data
statePopulation.put("CA", 38332521);
statePopulation.put("AZ", 6626624);
statePopulation.put("MA", 6692824);

System.out.printf("Population of Pennsylvania in 2013 is %d.%n", statePopulation.get("PA"));
```

⬇

```
Population of PA in 2013 is null.
```

Getting Keys That Aren't Present

```
HashMap<String, Integer> statePopulation = new HashMap<String, Integer>();

// 2013 population data
statePopulation.put("CA", 38332521);
statePopulation.put("AZ", 6626624);
statePopulation.put("MA", 6692824);

int x = statePopulation.get("PA");
```

⬇️

`NullPointerException`!

(What's the difference between this and the last example?)

Also Useful: `containsKey()`

`mapping.containsKey(K key)` returns `true` when `key` is present in `mapping` as a key, and `false` otherwise.

```
HashMap<String, Integer> statePopulation = new HashMap<String, Integer>();

// 2013 population data
statePopulation.put("CA", 38332521);
statePopulation.put("AZ", 6626624);
statePopulation.put("MA", 6692824);

System.out.println(statePopulation.containsKey("PA"));
System.out.println(statePopulation.containsKey("CA"));
```

➡️ `false, true`

Thinking with HashMaps

How could you use a HashMap to model recitation groups?

- What would be the keys?
- What would be the values?

Can you think of a couple ways to do it?

Idea 1

How could you use a HashMap to model recitation groups?

- Have the keys be student names
- Have the values be the group numbers.

Easy to answer the question "what recitation group is this student in?"

Are there questions that are harder to answer?

Idea 2

How could you use a HashMap to model recitation groups?

- Have the keys be group numbers
- Have the values be an array or an ArrayList of students in that recitation

Easy to answer the question "Who is in this particular recitation group?"

Are there questions that are harder to answer?

Takeaway:

There are multiple ways to tse a HashMap to organize the same data.

Some implementations have different virtues compared to others.

Worked Example

Return a `HashMap` mapping each student to their highest assignment score.

Find `assignments.zip` on the course website.

```java
public static HashMap<String, Double> findBestScores(String[] filenames) {
    return null;
}

public static void main(String[] args) {
    String[] filenames = {"hw1.txt", "hw2.txt", "hw3.txt"};
    HashMap<String, Double> output = findBestScores(filenames);
}
```

Solution

```java
public static HashMap<String, Double> findBestScores(String[] filenames) throws IOException {
    HashMap<String, Double> mapping = new HashMap<String, Double>();
    for (String filename : filenames) {
        FileInputStream gradeFile = new FileInputStream(filename);
        Scanner gradeScanner = new Scanner(gradeFile);
        while (gradeScanner.hasNextLine()) {
            Scanner line = new Scanner(gradeScanner.nextLine());
            String key = line.next();
            double newGrade = line.nextDouble();
            if (!mapping.containsKey(key) || mapping.get(key) < newGrade) {
                mapping.put(key, newGrade);
            }
        }
        gradeScanner.close();
    }
    return mapping;
}
```

Other Useful Methods

Signature	Purpose
putIfAbsent(K key, V value)	Only add the mapping from key to value if the key isn't already in the mapping.
containsValue(V value)	Returns `true` if there's a key that maps to `value`; `false` otherwise.
remove(K key)	Removes the entry corresponding to this key
clear()	Removes all entries
keySet() and values()	Returns a set of keys or a collection of values, correspondingly

Hashing

🚨 The next few slides don't need to be memorized closely. This is just some (hopefully interesting) background. 🚨

Hashing

Of course, we could implement something similar to a HashMap just using an ArrayList.

How would we approach this?

Hashing

Of course, we could implement something similar to a HashMap just using an ArrayList.

How would we approach this?

- Maybe create a new class for the Entry (e.g. something with a field for the `String` key and a field for the `int` value)
- Put a bunch of those objects in an ArrayList and iterate over the list to find/add things

Problem: Finding the value mapped to a key this way takes a lot of looking.

Here's a basic approach to looking over the Mappings in an ArrayList, iterating over them one at a time.

```java
for (int i = 0; i < myMap.size(); i++) { // if a "hash map" were just an ArrayList
  Entry current = myMap.get(i);
  if (current.getKey().equals("Harry")) {
    return current.getValue();
  }
}
```

What's the value associated with key "Harry"?

How many iterations of the for-loop do we go through if `entryOne` has the key `"Harry"`?

```
myMap = < entryOne, entryTwo, entryThree, entryFour, ... entry10000 >
```

```
for (int i = 0; i < myMap.size(); i++) { // if a "hash map" were just an ArrayList
  entry current = myMap.get(i);
  if (current.getKey().equals("Harry")) {
    return current.getValue();
  }
}
```

What's the value associated with key "Harry"?

How many iterations of the for-loop do we go through if `entry10000` has the key `"Harry"`?

```
myMap = < entryOne, entryTwo, entryThree, entryFour, ... entry10000 >
```

```
for (int i = 0; i < myMap.size(); i++) { // if a "hash map" were just an ArrayList
  Mapping current = myMap.get(i);
  if (current.getKey().equals("Harry")) {
    return current.getValue();
  }
}
```

Linear Searching Has Linear Costs

Searching index by index (*a linear search*) in an ordered sequence like an array or ArrayList requires more effort when...

- There are more elements in the collection, and
- The element that you're looking for is located towards the end

Linear Searching Has Linear Costs

Not really a problem now, but imagine if Facebook had to use this strategy to find how many friends a user has

- Way too many users
- Devastating results if you're looking for a user at the end of the list.

Hashing: almost magical

Imagine if, given a very big ArrayList of **values**, and some particular **key**, you could ~instantly calculate the index where its value would live.

- Calculate the index where the value should live (~instant)
- Ask for the value stored at that index from the ArrayList (~instant)
- If the value is null, then you know the key doesn't have a value mapped to it.
- If the value is not null, then you know you have the value mapped to the key.

Hashing: almost magical

Finding the last name as a value associated with a first name key:

```
values-> ["Fouh", "Yang", ..., "Smith", "Wu", ..., "Mammadov"];
(indices) 0       1       ...   2928     2929 ...  389293

key -> "Harry"

crystalBall(key) -> The value associated with key "Harry" would live at index 2928.

values.get(2928) -> "Smith"
```

✅ "Harry" maps to "Smith"

Not actually magic! (Obviously)

Carefully chosen functions called *Hash Functions* can (almost uniquely) map a key to one index over a specified range.

These indices can be used to indicate where the corresponding value should be stored in an underlying table.

> *Excuse me, can you tell me where to find the coffee?*

> *Yes, aisle three.*

Not actually magic! (Obviously)

More on Hashing in future courses here (excited for CIT 594?)

Sets

HashMaps are useful for asking questions about data associations.

- What's this person's student ID?
- How many students are in the group 8?
- What's the population of this state?

Sometimes we don't need to worry about data association, only membership.

Questions of Membership

- Is John in class today?
- Has this assignment been added to the "graded" group?
- While iterating over this collection, have I seen this particular element yet?

These are questions that can be answered by maintaining a Set of elements that meet a certain criteria.

Motivating a Set

Imagine that we have a requirement that each student attend at least one "Code Review" throughout the semester. A student can attend more than one, but at least one is required for credit.

Idea:

- Keep a Set of students
- Whenever a student attends a code review, add them to the Set
- At the end of the semester, for each student in the course, assign them +10pts if they are in the set of students who have attended a code review.

Motivating a Set

```
codeReviewAttendance <- {} // the empty set

// after code review 1:
codeReviewAttendance.add("Vivian")
codeReviewAttendance.add("Jintong")
print(codeReviewAttendance)-> {"Jintong", "Vivian"}


// after code review 2:
codeReviewAttendance.add("Vivian")
codeReviewAttendance.add("Dana")
print(codeReviewAttendance)-> {"Jintong", "Vivian", "Dana"}
```

HashSet

An **unordered** collection of elements that supports exceedingly quick performance on the operations `add`, `remove`, `contains`, and `size`.

Any element can only be present once in a HashSet.

Making a HashSet

```
HashSet<E> set = new HashSet<E>();
```

Creates a new variable called `set` that stores a `HashSet<K, E>` (an unordered collection of elements of type `E`). The initial value of the variable is set to be a new empty `HashSet<E>`.

e.g.,

```
HashSet<String> attendance = new HashSet<String>();
```

Other Useful Methods

Signature	Purpose
add(E e)	Put `e` in the set
contains(E e)	Returns `true` if `e` is present in the Set, `false` otherwise.
remove(Object o)	Removes `o` from the Set.
size()	Number of entries in the set

`add` and `remove` return `true` when the operation changes the set, and `false` otherwise.

Set Practice:

Union

- Given two sets, return a set containing all elements present in either of the sets.

Intersection

- Given two sets, return a set containing all elements present in both sets.