More Exceptions, JUnit

Exceptions

We've seen exceptions before:

- `ArrayIndexOutOfBoundsException` when you ask for an index outside of an Array
- `InputMismatchException` when you try to use a `Scanner` to read an `int` from something that doesn't represent an `int`
- `NullPointerException` when you call a method from an object that hasn't been properly initialized

Unchecked Exceptions

"You messed up."

These are the exceptions we've seen before. It's possible to write code that compiles but doesn't behave correctly. These bugs are your fault as a programmer.

Not everything is your fault, though

Checked Exceptions are those resulting from errors outside of your control.

- A file you're trying to read from just isn't there
- The operating system refuses to help you close a file
- The file ended while you were expecting there to be more to read

These have to be handled.

Handling Exceptions

- Easy way: just don't!
  - Tag the method that could throw a checked exception with `throws <Exception-Name>`
  - Let the method that called this method worry about. *Don't say we didn't warn ya!*
- Safer way: `try/catch`
  - Specify behavior in the case that specific exceptions are thrown.

(It is actually often better to take the easier route)

Try / Catch

Syntax for trying to run some code and providing a response in the case that an Exception is thrown.

```
try {
    // dangerous statement(s)
} catch (IOException e) {
    // error handling statement(s)
} catch (NumberFormatException e) {
    // error handling statement(s)
}
```

If an exception is thrown, run the code in the first `catch` block with a matching exception.

Try / Catch / Finally

Code in `finally` will be run once the `try` block is entered, no matter what.

```
PrintWriter out = new PrintWriter("file.txt");
try {
    writingMethod(out);
} catch (NumberFormatException e) {
    System.out.println("Can't do that!");
} finally {
    out.close()
    // Make sure that out gets closed no matter what!!!
}
```

Throwing Exceptions

Sometimes you want to throw an exception yourself!

```
throw new Exception("message describing error");
```

e.g.

```
throw new IllegalStateException("You cannot accelerate if the car is in park");
throw new IllegalArgumentException("Can't deposit a negative value into your bank account");
throw new ArithmeticException("This field does not have a zero element");
```

Throwing Practice

Fix this program so that it throws an exception with a meaningful message when you try to double the value at an invalid position in the array.

```
public static void doubleElement(int i, int[] arr) {
    arr[i] *= 2;
}
```

Throwing Your Own Exceptions

Basically, create a new class that follows exactly this template. We'll learn what most of these new words (`extends` , `super`) refer to soon.

```
public class CustomException extends Exception {

        public CustomException(String message) {
                super(message);
        }

}
```

Basically nothing to it!

Throwing Practice

Fix this program again so that it throws an `IllegalDoublingIndexException` exception with a meaningful message when you try to double the value at an invalid position in the array. (You'll need to write the `IllegalDoublingIndexException`).

```
public static void doubleElement(int i, int[] arr) {
    arr[i] *= 2;
}
```

CopyFile.java Example

Copy a chosen file to a new destination, but overwriting the old file shouldn't be allowed!

```
public class CopyFile {
    public static void main(String[] args) {
        Scanner scnr = new Scanner(System.in);
        String source;
        String destination;
        System.out.println("Provide the source filename.");
        source = scnr.next();
        System.out.println("Provide the destination filename.");
        destination = scnr.next();
        copyFile(source, destination);
    }
}
```

Testing

Testing Helps By...

- Identifying when your new code is more or less correct (unit testing)
- Checking to see that new changes to your code base haven't broken anything (regression testing)

Unit Tests

Every Unit of code that you write (typically every method for every class counts as a unit) should be tested with unit tests.

Design a number of unit tests per unit of code that check how the unit of code behaves for a particular category of input.

Black Box Testing

Unit Testing is a type of *black box testing*; we provide some inputs and check that the output is correct.

- Pretend that you don't know how the method is written
- Matches how end users actually interact with the classes you write

Example: you can test that a score for a game is calculated without knowing how the method is actually implemented.

```
public double calculateScore(Board b) {
    // Hidden!
}
```

Generating Test Cases

Unit tests consist of:

- A chosen input, to see how the unit behaves when given
- An expected output, determined to be what the unit should return given the particular input
- An actual output, calculated by running the unit of code on the chosen input
- An assertion statement that compares the *actual* to the *expected* output and indicates whether or not the test passed.

Generating Test Cases

Example:

```
public int maxOfThree(int a, int b, int c)
```

Input	Expected	Actual
3, 4, 5	5	maxOfThree(3, 4, 5)
3, 5, 4	5	maxOfThree(3, 5, 4)
-1, -2, -3	-1	maxOfThree(-1, -2, -3)

The assertion for all is the same: `assertEquals(expected, actual)`.

JUnit

JUnit is a testbench package, or a package used to write Java programs that can be run to test the behavior of your code.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
class HoursMinsToMinutesTest {

        @Test
        void testZeroHoursZeroMinutes() {
                int inputHours = 0;
                int inputMinutes = 0;
                int expected = 0;
                int actual = HoursMinsToMinutes.hrMinToMin(inputHours, inputMinutes);
                assertEquals(expected, actual);
        }
    ...
```

JUnit

Very specific syntax:

- Every test case is a function, annotated with `@Test`
  - The function should be `public void`
  - The function name should describe what the test is checking
- The test should have the inputs, expected output, and actual output represented somewhere inside
- The test usually ends with a special method called `assert...`

JUnit

- assertEquals()
  - Check that two inputs are equal
- assertTrue()
  - Check that some boolean expression is true
- assertNull()
  - Check that some value is null
- assertArrayEquals()
  - Check if two arrays have the same elements