159 XTENDED

A WEEKLY REVIEW

WEEK 3 REVIEW, TESTING

Pull up **Socrative.com**

and join room **XTEND159**

WHAT IS A BUG?

Answer individually first, then discuss as a group.

TESTING TERMINOLOGY

Definitions from Dr. David Bernstein's CS 159 Lecture Slides on Testing.

- Failure

  A deviation between a product's actual behavior and intended behavior.

- Fault

  A defect that could (or does) give rise to a failure.

- Trigger Condition

  A condition that cause a fault to result in a failure

- Symptom

  A characteristic of a failure (that helps you recognize that a failure has occurred)

ACTIVITY 1: SPOTTING A BUG

```
public static int divides(int a, int b)
{
    int result = a/b;
    System.out.println(result);
    return result;
}
```

What input would break this method?

How will the method behave in this instance?

ACTIVITY 1: SPOTTING A BUG

```
public static int divides(int a, int b)
{
    int result = a/b;
    System.out.println(result);
    return result;
}
```

What input would break this method?

B having the value 0.

How will the method behave in this instance?

An ArithmeticException will be thrown.

ACTIVITY 1: SPOTTING A BUG

```
public static int divides(int a, int b)
{
    int result = a/b;
    System.out.println(result);
    return result;
}
```

What input wouldn’t break this method?

How will the method behave in this instance?

ACTIVITY 1: SPOTTING A BUG

```
public static int divides(int a, int b)
{
     int result = a/b;
     System.out.println(result);
     return result;
}
```

What input wouldn't break this method?

A any value, B any value other than 0.

How will the method behave in this instance?

It will print and return the result.

ACTIVITY 1: SPOTTING A BUG

```
public static int divides(int a, int b)
{
    int result = a/b;
    System.out.println(result);
    return result;
}
```

Failure: Method throws an ArithmeticException instead of returning a number.

Fault: Not checking if the denominator is zero before dividing.

Trigger Condition: B having the value 0.

BEFORE WRITING TESTS, TAKE A MOMENT TO THINK THROUGH THE FUNCTIONALITY OF YOUR CODE.

TESTING TERMINOLOGY

Definitions from Dr. David Bernstein's
CS 159 Lecture Slides on Testing.

- Unit Testing ← JUnit Testing

  A component is tested on it's own.

  Testing a method.

- Integration Testing

  Components are tested in combination.

  Testing the entire program.

TESTING TERMINOLOGY

Definitions from Dr. David Bernstein's CS 159 Lecture Slides on Testing.

- Test Case
    - A particular choice of inputs (and expected outputs)
        - A single assertEquals, assertTrue, or assertFalse test.
- Test Suite
    - A collection of tests.
        - A test class for a particular class.

TESTING TERMINOLOGY

Definitions from Dr. David Bernstein's CS 159 Lecture Slides on Testing.

- Line Coverage

  Make sure every statement in your code is executed.

- Branch Coverage

  Make sure every "branch" (resulting from conditionals) is executed.

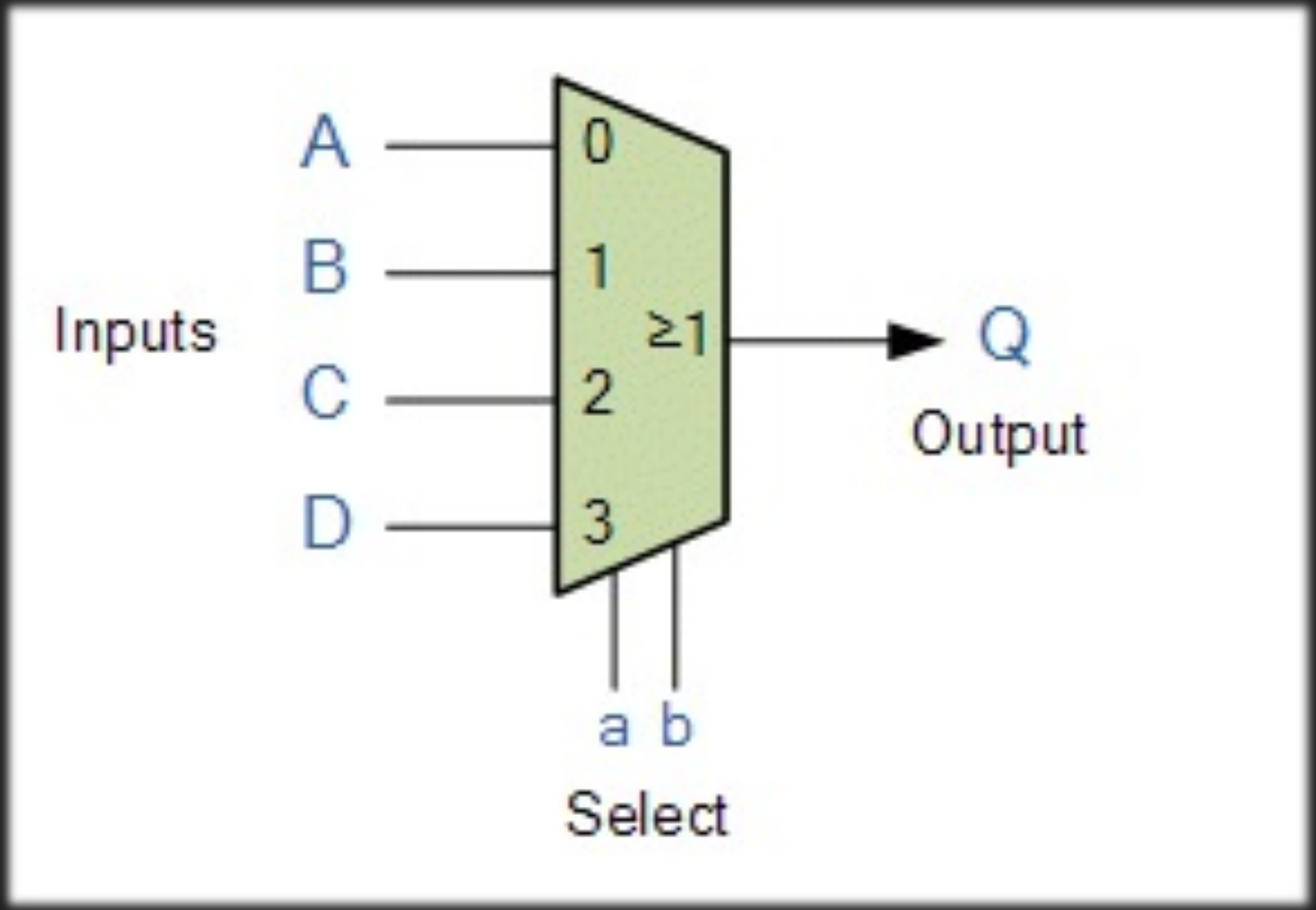
ACTIVITY 2: WRITING TESTS

Small preview of CS 261 :P

```
public static int multiplexor(int a, int b, int s)
{
    if (s < 0 || s > 1)
        return 0;

    if (s == 0)
        return a;

    if (s == 1)
        return b;
}
```



If the selector is 0, `a` should be returned. If the selector is 1, `b` should be returned. Any other value for selector is invalid, so 0 should be returned. Write a test suite that provides 100% line and branch coverage for this method.

TESTING REVIEW

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

import org.junit.jupiter.api.Test;

public class multiplexorTest
{
    @Test
    public void goodSelectorTest()
    {
        assertEquals(2, multiplexor(2, 3, 0));
        assertEquals(5, multiplexor(0, 5, 1));
    }

    @Test
    public void badSelectorTest()
    {
        assertEquals(0, multiplexor(1, 1, -5));
        assertEquals(0, multiplexor(1, 1, 10));
    }
}
```

TESTING TERMINOLOGY

Definitions from Dr. David Bernstein's CS 159 Lecture Slides on Testing.

- What type of testing approach was this?

TESTING TERMINOLOGY

Definitions from Dr. David Bernstein's CS 159 Lecture Slides on Testing.

- What type of testing approach was this?
- White Box Testing

  The tester knows the internal details of the component being tested.

  When you can see the code while writing tests.

TESTING TERMINOLOGY

Definitions from Dr. David Bernstein's CS 159 Lecture Slides on Testing.

- What type of testing approach was this?
- White Box Testing

  The tester knows the internal details of the component being tested.

  When you can see the code while writing tests.
- Let's look at an alternative approach…
- Black Box Testing

  The tester has information about the form of the inputs and outputs, but not about the "internals" of the component being tested.

  When you can't see the code while writing tests.

ACTIVITY 3: BLACK BOX TESTING

`calculator` receives two ints `a` and `b` and returns a String.
If either of the parameters are greater than 100, it should return “Overload!”.
If either of the parameters are less than zero, it should return “Underload!”.
If both of parameters are within the range of 0 to 100, it should return a String representation of the sum of the values.

Write a test suite that provides 100% line and branch coverage for this method.

THE CODE:

```
public static String calculator(int a, int b)
{
    if (a < 0 || b < 0)
        return "Underload!";

    if (a > 100 || b > 100)
        return "Overload!";

    return a + b;
}
```

THE TESTS:

```
assertEquals("Underload!", calculator(1, -6));
assertEquals("Underload!", calculator(-6, 1));
assertEquals("Overload!", calculator(101, 1));
assertEquals("Overload!", calculator(1, 101));
assertEquals("10", calculator(5, 5));
```

THE CODE:

```
public static String calculator(int a, int b)
{
    if (a < 0 || b < 0)
        return “Underload!”;

    if (a > 100 || b > 100)
        return “Overload!”;

    return a + b;
}
```

THE TESTS:

```
assertEquals(“Underload!”, calculator(1, -6));
assertEquals(“Underload!”, calculator(-6, 1));
assertEquals(“Overload!”, calculator(101, 1));
assertEquals(“Overload!”, calculator(1, 101));
assertEquals(“10”, calculator(5, 5));
```

Notice how if a is greater than 100,
and b is greater than 0,
we don’t know whether to return
“Overload!” or “Underload!”

Writing tests before code
helps us think about how
our code is supposed to behave.