Testing Methods

Writing effective unit tests is an essential skill for software developers. In this activity, you will learn the basics of designing tests for methods.

Manager:	Recorder:
Presenter:	Reflector:

Content Learning Objectives

After completing this activity, students should be able to:

- Describe one benefit and one difficulty of test-driven development.
- Select input and expected output values to use as tests for a method.
- Correctly compare values based on type (integer, double, and string).

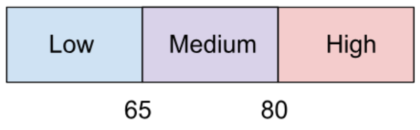
Process Skill Goals

During the activity, students should make progress toward:

- Hand-tracing code and predicting output. (Information Processing)

Model 1   Method Specifications

A program about the weather includes a class named `Temperature`. A method named `report` takes an integer parameter named `temp` and returns a `String`. The return value is `"High"` if `temp` is 80 or above, `"Medium"` if `temp` is 65 or above, and `"Low"` otherwise.



Example Input and Output

Input: 60	Input: 65	Input: 70	Input: 85
Output: `"Low"`	Output: `"Medium"`	Output: `"Medium"`	Output: `"High"`

Questions (15 min)   Start time:

1. Based on the specification:

a) What is the class name?

b) What is the method name?

c) What is the return type?

d) What is the parameter name?

e) What is the parameter type?

2. What should the return value be if the argument passed to the parameter is. . . ?

a) 90   b) 32   c) 80

3. As a team, discuss what each pair of input values has in common:

a) 32 and 60

b) 60 and 90

c) 65 and 80

4. Organize the example inputs from the model and questions #2 and #3 into the following categories. (You should have seven integers listed.)

"Low"	Boundary	"Medium"	Boundary	"High"

5. Which category might benefit from having an additional example? Explain your answer.

6. If 65 and 80 were not included in the examples, would writing the method (based on the specifications) be easier or more difficult? Explain your answer.

7. What is the smallest number of tests you would choose to ensure the code for this specification is correct? Explain your answer.

8. (Optional) What are the most extreme input values that could be used to test a method like `Temperature.report()`?

The process of writing tests before writing code is called **test-driven development** *(TDD). By using TDD, a programmer has a better idea of what the code should do before writing the code.*

The examples of 65 and 80 are called **boundary conditions** *(or edge cases). Programming mistakes are more likely to occur on the boundary where the output may change.*

Model 2 Expected vs Actual

Proposed Implementation

```
public static String report(int temp) {
    if (temp > 80) {
        return "High";
    }
    if (temp > 65) {
        return "Medium";
    }
    return "Low";
}
```

Input	Expected Output	Actual Output
60	"Low"	
65	"Medium"	
70	"Medium"	
80	"High"	
85	"High"	

Questions (10 min) Start time:

9. Trace the code by hand, and fill in the Actual Output column of the table. Put an asterisk (*) next to any actual value that doesn't match the expected.

10. Explain how to fix the code so that the actual values match the expected values.

11. Consider a method named `average` that takes three parameters (representing temperature values) and returns the average: $(x + y + z) \div 3$. Each parameter is a `double`, and the return type is `double`. Write three examples of inputs and expected output values that would do the *best possible job* of testing the `average` method.

a) Inputs: Output:

b) Inputs: Output:

c) Inputs: Output:

12. What is different or unique about each of your tests?

By comparing actual results against expected results, one can test if code is correct. When designing tests, the inputs should be **categorically different** *to ensure that all cases are being tested.*

Model 3 Comparing Values

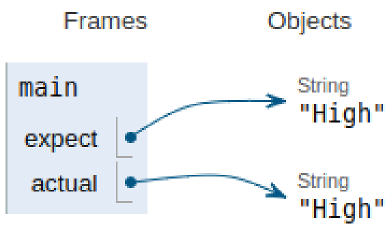
The == operator can be used to compare two integers:

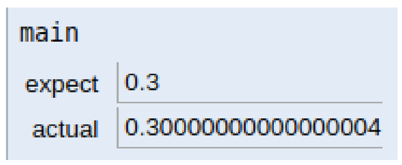
```
int expect = 5;
int actual = Math.abs(-5);

if (expect == actual) {
    System.out.println("Pass!")
} else {
    System.err.println("Fail!")
}
```

However, not all data types can be compared using ==. Care must be taken when comparing strings and doubles. The following examples would print *Fail!* if compared using ==.

```
expect = "High";
actual = Temperature.report(90);
```

```
expect = 0.3;
actual = 0.1 + 0.1 + 0.1;
```





Figures drawn with Java Visualizer: https://cscircles.cemc.uwaterloo.ca/java_visualize/

Questions (20 min) Start time:

13. Consider the first example in the model, where `expect` and `actual` are integers. What is the output of the if statement?

14. Consider the `String` example in the model (bottom left).

a) Do the strings in the diagram have the same contents?

b) Do `expect` and `actual` refer to the same string object?

15. Explain why `expect == actual` can be `false` when comparing strings, even if the strings have the same contents.

16. Consider the `double` example in the model (bottom right). What is the boolean result of `expect == actual`?

17. Using a web browser, open the FloatConverter link below. Enter each input value, one at a time, into the box labeled "You entered." Record the "Value actually stored" in the table.

https://www.h-schmidt.net/FloatConverter/IEEE754.html

Input Value	**Actual Value**
0	
0.1	
0.3	
0.5	

18. Explain why 0.1 + 0.1 + 0.1 (in the model) results in the value 0.30000000000000004 (instead of the correct value 0.3).

19. Why does using the == operator not always work when comparing doubles?

20. Rewrite the code "`expect == actual`" (without using the == operator) to test if the two variables are close enough to be considered equal. Assume that "close enough" means that the values are within 0.0001 of each other.

In the last question, 0.0001 is called a **delta** *(or tolerance). Another way to write 0.0001 in Java is* `1e-4` *(which means* 10^{-4}*). A delta of* `1e-4` *tests whether the first four decimal places of* `expect` *and* `actual` *are the same. Other common delta values include* `1e-9` *and 0.01, depending on how much precision is needed.*