Unit Testing in Java	start time:

This activity will introduce key ideas & issues in testing individual methods.
We will use the Java language, the BlueJ environment, and the JUnit test framework,
but the same concepts apply to many other languages, environments, & frameworks.

Before you start, complete the form below to assign a role to each member.
If you have 3 people, combine Speaker & Reflector.

Team	**Date**
Team Roles	**Team Member**
Recorder: records all answers & questions, and provides copies to team & facilitator.	
Speaker: talks to facilitator and other teams.	
Manager: keeps track of time and makes sure everyone contributes appropriately.	
Reflector: considers how the team could work and learn more effectively.	

Notes

1. When a question asks you to **explain** something, always answer in **complete sentences**.
2. **Recorder**: Write legibly & neatly so that everyone can read & understand your responses.
    a. Note the time whenever your team starts a new section or question.
3. You will need a computer with BlueJ open and ready to use below.
    a. In BlueJ, go to Tools > Preferences > Editor to adjust the font size.
    b. Consider using an external monitor or projector so everyone can see BlueJ easily.

(12 min) I. Test a Method Body	start time:

```
public class GC {

  /**
   * Converts score to letter grade.
   * @param score   percentage score (0-100)
   * @return        letter grade (e.g. A, B, C, ...)
   */
  public static String sToG( double score ) {
    if      ( score >= 90.0 ) return "A";
    else if ( score >= 80.0 ) return "B";
    else if ( score >= 70.0 ) return "C";
    else if ( score >= 60.0 ) return "D";
    else                      return "F";
  }

} // end class GC
```

1. (2 min) Consider the Java method **`sToG()`** above.
Note that it includes a **signature** (name, parameters, return type), and a **body**.
Complete the table below to show what results you would expect for each input:

	a	**b**	**c**	**d**
test input:	100	90	89	80
expected output:				

2. (1 min) Suppose another developer calls **`sToG()`** as shown below.

```
  /**
   * Compute letter grade from score.
   * @param   correct number correct
   * @param   total   total possible
   * @return          letter grade (e.g. A, B, C, ...)
   */
  public static String getGrade(double correct, double total) {
    return GC.sToG( correct / total );
  }
```

What grade(s) will be returned by:

a. `getTestGrade(9.0, 10.0)` b. `getTestGrade(4.0, 4.0)`

3. (1 min) In complete sentences, describe the risks of using `sToG()` in a larger program, and how `sToG()` could be modified to avoid these risks.

4. (2 min) Explain why we should test methods with a variety of **expected inputs** (like those listed above).

5. (1 min) Explain why we should test methods with a variety of **unexpected inputs**, such as -10, +110, -1000, +1000, Double.MIN_VALUE, and Double.MAX_VALUE. (The last two are defined Java constants for the smallest and largest possible double values).

6. (3 min) Copy the above code for `sToG()` into BlueJ and make sure it compiles and runs. Measure the total time to run `sToG()` on each input below, and note the output.

expected	**a**	**b**	**c**	**d**
input:	`100`	`90`	`89`	`80`
output:				

unexpected	**e**	**f**	**g**	**h**	**i**	**j**
input:	`-10`	`+110`	`-1000`	`1000`	`Double. MIN_VALUE`	`Double. MAX_VALUE`
output:						

7. (1 min) Running each test by hand is called **manual testing**.
   a. How many seconds **total** did it take to **run all of the test cases**? ______
   b. How many seconds **per test case**? (divide total by number of tests) ______

Before you continue, review progress with the facilitator.

## *(16 min) II. Test a Method Signature*	start time:

1. (2 min) In BlueJ, **do not delete** `sToG()`, and **add** the code below for `gToP()`,
which has a **signature** but is **not implemented** (it a **stub** that allows the code to compile).

```
public class GC {

  public static String sToG(double score) { ... }

  //********** ADD THIS METHOD **********
  /**
   * Converts letter grade to quality points.
   * @param  grade   letter grade (A-F)
   * @return         quality points (e.g. 4.0, 3.0, ...)
   */
  public static double gToP(String grade) {
    return 0.0; // URGENT: this method is INCOMPLETE
  }

} // end class GC
```

2 (2 min) Method `gToP()` is not defined correctly, but **if it was**,
what **expected inputs** would you use to test it, and what outputs would you expect?

expected					
input:	"A"				
output:	4.0				

3. (2 min) What are some **unexpected inputs** you would use to test `gToP()`?

unexpected					
input:	"A-"				
output:	---	---	---	---	---

4. (1 min) In **black box testing**, the testers **do not look** at details of the code being tested.
In **white box testing** (also called **clear box** and **glass box testing**)
the testers **do look** at details of the code being tested.
Which approach (**black box** or **white box**) was used in section I? In section II?
Explain your answers in complete English sentences.

5. (2 min) Study the code below, and assume that `sToG()` and `gtoP()` are unchanged.
Describe what will be printed by:
a. `gToP_A()` b. `gToP_B()` c. `gToP_all()`

```
public class GC {

  public static String sToG(double score) { ... }

  public static double gToP(String grade) { ... }


  public static void gToP_A() {
    System.out.println("gToP A: " + (4.0 == gToP("A")) );
  }
  public static void gToP_B() {
    System.out.println("gToP B: " + (3.0 == gToP("B")) );
  }
  ...
  public static void gToP_all() {
    gToP_A(); gtoP_B(); ...
  }

} // end class
```

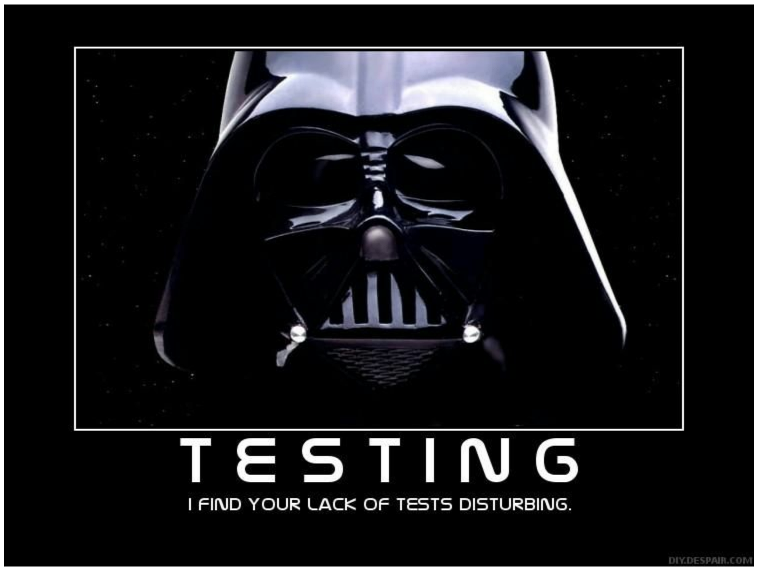
6. (4 min) Edit class `GC` to add one test method for each of your 5 expected test cases above, using the examples above. (**Do not delete existing code** since you may need it later.) Measure the time it takes you to write the test cases, and the time it takes you to run them. Each test method should call `gToP()` with one test input & check the result.
This is known as **automated testing**.

a. How many seconds **total** did it take to **write all 5 test cases**? ______
b. How many seconds **per test case**? (divide total by number of tests) ______

c. How many seconds **total** did it take to **run all 5 test cases**? ______
d. How many seconds **per test case**? (divide total by number of tests) ______

7. (1 min) All of the `gToP()` tests have a similar structure. Describe this structure, and explain why unit tests for many other methods might have similar structures.

8. (1 min) Give a **copy** of `class GC` to the facilitator to review (printout or email).
If you are editing this document electronically,
paste your Java code below or at the end of the document.



(12 min) III. Structure of Test Code	start time:

1. (2 min) Each set of test input values is a **test case**.
A **test method** can test 1 or more test cases. How many test cases are tested by method:

a. `gToP_A()` ____ b. `gToP_all()` _____

2. (1 min) In general, it is easier to debug a short method or a long method? _____

3. (2 min) A set of related test methods is called a **test suite.**
A Java class that contains a test suite (and not much else) is called a **test class**.
Explain why, in general, it is better to have a test suite with 10 test methods,
each with 2 test cases, rather than a test suite with 2 test methods, each with 10 test cases.

4. (2 min) In your automated tests, how do you know when a test **passes**? When a test **fails**?

5. (2 min) If you had a test suite with 1000 (or 100,000) test methods,
explain why it is better to see a list of **failed tests** than a list of **passed tests**.
(Hint: Which list is longer? Which list is more useful if you need to fix problems?)

6. (1 min) If you had to test 100+ methods, and run all of the tests 100+ times,
would you prefer manual testing or automated testing?

Before you continue, review progress with the facilitator.

(12 min) IV. Numbers of Test Cases	start time:

1. (5 min) Study the handout with methods from Java's **Math** and **String** classes. For each method below, list input values (or sets of inputs) you might use to prove whether it works correctly.

Java API method & example use	**test case #1**	**test case #2**	**test case #3**	**test case #4**
a. `Math.abs(double a)` `x = Math.abs(-30);`	-0.01	0.01	0	
b. `Math.round(double a)` `x = Math.round(3.01);`				
c. `Math.sqrt(double a)` `x = Math.sqrt(2);`				
d. `s.charAt(int index)` `y = "dog".charAt(1);`				
e. `s.endsWith(String suffix)` `z = "dog".endsWith("g");`				

2. (4 min) Based on question 1, for each parameter type below, list some likely test input values. Remember that you want to test how a method works under **all** circumstances.

a. int b. double c. String d. boolean

3. (1 min) Estimate a typical number of **test cases** for any method with 1 parameter, based on your answers to the previous question. Explain your estimate.

4. (2 min) Estimate a typical number or range of **test cases** for any method with 2, 4, and 8 input parameters. Explain your estimate.

(8 min) V. Development Practices	start time:

1. (2 min) Explain why it tends to be easier to test methods
that are **short (~5 lines)** rather than **long (50+ lines)**.

2. (2 min) Explain why it tends to be easier to write **test methods**
than **methods being tested**.

3. (3 min) Some developers use the following approach to development:

i. Check that the entire test suite **passes**, and everything works correctly.
ii. Write a new test case for a problem to be fixed or a feature to be added.
iii. Check that the new test **fails**. so that something must be changed.
iv. Write code that should cause the new test to pass.
v. Check that the entire test suite (including the new test) **passes**,
which shows that the change was successful, and that nothing else was broken.
vi. If appropriate, rewrite the code to make it better without changing what it does.
This is called **refactoring**.
vii. Check that the entire test suite **passes**, which shows that
the refactoring was successful, and that nothing else was broken.

This approach is **Test-Driven Development (TDD) .** In TDD:

a. Which is **written first** - the **test case** for a feature or the **feature**?
b. When a **new test** is run for the first time, should it **pass or fail**?
c. **How long** should it take to find out if a change in the code causes a test to **fail**?

Handout: Selected Methods from Math & String

Class Math	Selected Methods	

`static double`
`static float`
`static int`
`static long`
	`abs(double a)`
	`abs(float a)`
	`abs(int a)`
	`abs(long a)`
		Returns the absolute value of a value.

`static double`
`static float`
`static int`
`static long`
	`max(double a, double b)`
	`max(float a, float b)`
	`max(int a, int b)`
	`max(long a, long b)`
		Returns the greater of two values.

`static double`
`static float`
`static int`
`static long`
	`min(double a, double b)`
	`min(float a, float b)`
	`min(int a, int b)`
	`min(long a, long b)`
		Returns the smaller of two values.

`static double`	`pow(double a, double b)`	Returns a raised to the power of b.

`static long`
`static int`
	`round(double a)`
	`round(float a)`
		Returns the closest integer to the argument.

`static double`	`sqrt(double a)`	Returns the correctly rounded positive square root of a value.

Class Math	Selected Methods	
`char`	`charAt(int index)`	Returns the char value at an index.
`boolean`	`endsWith(String suffix)`	Tests if string ends with a suffix.
`boolean`	`equalsIgnoreCase(String str)`	Compares string to another String, ignoring case.

`int`
`int`
`int`
`int`
	`indexOf(int ch )`
	`indexOf(int ch , int i)`
	`indexOf(String s )`
	`indexOf(String s , int i)`
		Returns the index within string of the **first** occurrence of ch or str, searching forward starting at the specified index.

`int`
`int`
`int`
`int`
	`lastIndexOf(int ch )`
	`lastIndexOf(int ch ,int i)`
	`lastIndexOf(String s )`
	`lastIndexOf(String s ,int i)`
		Returns the index within string of the **last** occurrence of ch or str, searching backward starting at the specified index.

`boolean`	`startsWith(String prefix)`	Tests if string starts with a prefix.

`String`
`String`
	`substring(int beginIndex)`
	`substring(int beginIndex,`
	`int endIndex)`
		Returns a new string that is a substring of this string.

`static` `String`	`valueOf(/* any */ a)`	Returns the string representation of the argument.