

# Announcements

---

The slides for this lecture are particularly unhelpful without consulting either the textbook or the web videos. If you feel like you're missing something you are, see those resources instead.

# Pre-Announcement

---

Project Manager at Autonomous Underwater Vehicles at Berkeley here to say  
“Join us for sea robots.”

- Interest in AI, Machine Learning, etc.
- Email [auvs.at.berkeley@gmail.com](mailto:auvs.at.berkeley@gmail.com)
-

# Quick Note on Autograders / Debugging

---

Autograder is not intended to be your debugging tool.

- It is a bad habit to rely on teacher provided tools for correctness.

Example: Your LinkedListDeque is failing in the autograder, but works fine when you run the provided are passing LinkedListDequeTest.java file.

- Approach #0: Assume the autograder is broken (unlikely but possible).
- Approach #1: Visually inspect your code for errors. Sometimes works.
- Approach #2: Ask for help in office hours / Piazza. OK, but slow!
- Approach #3: Look inside LinkedListDequeTest.java to see how it works. Add your own tests that fail. Set breakpoints and use the visual debugger to figure out what's going wrong.
  - We'll be discussing this sort of approach today.

# CS61B: 2018

---

## Lecture 7: Testing

- A Simple JUnit test
- Testing Philosophy
- Selection Sort
- Simpler JUnit Tests



# How Does a Programmer Know That Their Code Works?

---

What evidence did you have for Project 0 in 61B?

- We gave you some tests.
- Running main and seeing if the planets move around in a proper planetary way.
- The real MVP: Autograder.

In the real world, programmers believe their code works because of **tests they write themselves**.

- Knowing that it works for sure is usually impossible.
- This will be our new way.

# How Does a Program

Know That It's Not Working?

What evidence did you

- We gave you some
- Running main and
- way.
- The real MVP: Aut

In the real world, programs  
**write themselves.**

- Knowing that it w
- This will be our ne



# Sorting: The McGuffin for Our Testing Adventure

---

To try out this new way™, we need a task to complete.

- Let's try to write a method that sorts arrays of Strings.

`x = {"he", "is", "the", "agoyatis", "of", "mr.", "conchis"}`



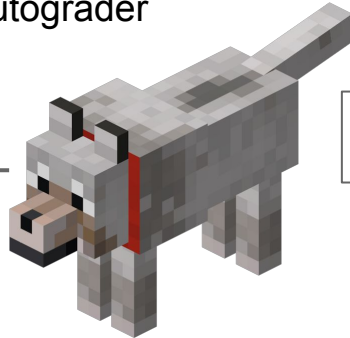
```
public static void  
sort(String[] x)
```



`x = {"agoyatis", "conchis", "he", "is", "mr.", "of", "the"}`

# The Old Way

Autograder



```
public static void  
sort(String[] x)
```

gradescope

Back to Course

Project 0 : NBody

- Configure Autograder
- Manage Submissions
- Review Grades
- Settings

Autograder Results

Results Code

Autograder Output (hidden from students)

... (truncated)

Results of the entire autograder run using autograder version 0.23 beta.

Velocity Limiting (0.0/0.0)

To keep you from getting too reliant on the autograder submission to the ad all you use one "token". You have after any submission that consumes a token, will abort, and you'll have to wait until your me

Hi Quan Sun. You have 1 tokens remaining out of a

- Submission at Friday January 27, 00:35:11 PST
- Submission at Friday January 27, 00:20:20 PST

If this current submission scores 0 points, you a Tokens recharge every 1200 seconds. Your next rec

File Checking (0.0/0.0)

Verifying that all required files have been submi

- ./NBody.java (ok)
- ./Planet.java (ok)

Compilation

GROUP

+Add Group Member

AUTOGRADE SCORE

19.667 / 25.0

FAILED TESTS

NBody: Test readPlanets (0.0/1.333)

Planet: body.txt (using our NBody) (0.0/1.333)

Planet: 9body.txt (using our NBody) (0.0/1.333)

NBody: Test NBody Textual Output (0.0/1.333)

PASSED TESTS

Velocity Limiting (0.0/0.0)

File Checking (0.0/0.0)

API (5.0/5.0)

Planet: Constructor test (1.333/1.333)

Planet: calcDistance test (1.333/1.333)

Planet: calcForceExertedBy tests (1.333/1.333)

Planet: calcForceExertedByX and calcForceExertedByY tests (1.333/1.333)

Planet: calcNetForceExertedByXY (1.333/1.333)

Planet: calcNetForceExertedByX 3body.txt (1.333/1.333)

Planet: calcNetForceExertedByY 3body.txt (1.333/1.333)

Planet: update() test (1.333/1.333)

NBody: Test readRadius (1.333/1.333)

Planet: planets.txt (using our NBody) (1.333/1.333)

Planet: planets.txt (dt not equal to 25000) (using our NBody) (1.333/1.333)

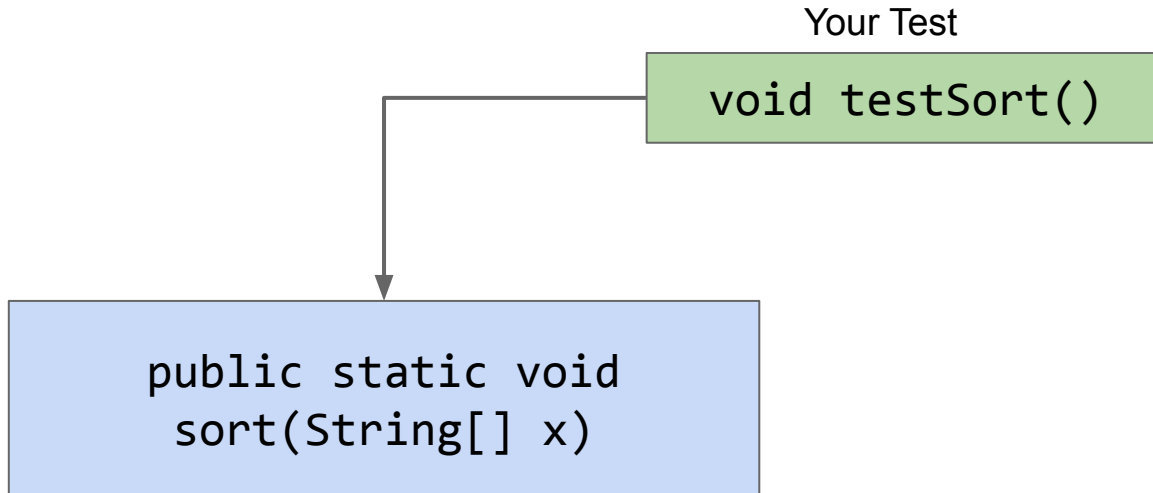


# The New Way

---

In this lecture we'll write `sort`, as well as our own test for `sort`.

- Even crazier idea: We'll start by writing `testSort` first!



# Ad Hoc Testing vs. JUnit

# Ad-Hoc Testing is Tedious

```
public class TestSort {  
    /** Tests the sort method of the Sort class. */  
    public static void testSort() {  
        String[] input = {"beware" , "of", "falling", "rocks"};  
        String[] expected = {"beware" , "falling", "of", "rocks"};  
        Sort.sort(input);
```

JUnit saves us the trouble of  
writing code like this (and more!).

```
        for (int i = 0; i < input.length; i += 1) {  
            if (!input[i].equals(expected[i])) {  
                System.out.println("Mismatch at position " + i + ", expected: '" + expected[i] +  
                                    "', but got '" + input[i] + "'");  
                return;  
            }  
        }  
    }  
}
```

```
    public static void main(String[] args) {  
        testSort();  
    }  
}
```

## JUnit: A Library for Making Testing Easier (example below)

```
public class TestSort {  
    /** Tests the sort method of the Sort class. */  
    public static testSort() {  
        String[] input = {"cows", "dwell", "above", "clouds"};  
        String[] expected = {"above", "cows", "clouds", "dwell"};  
        Sort.sort(input);  
  
        org.junit.Assert.assertArrayEquals(expected, input);  
    }  
  
    public static void main(String[] args) {  
        testSort();  
    }  
}
```

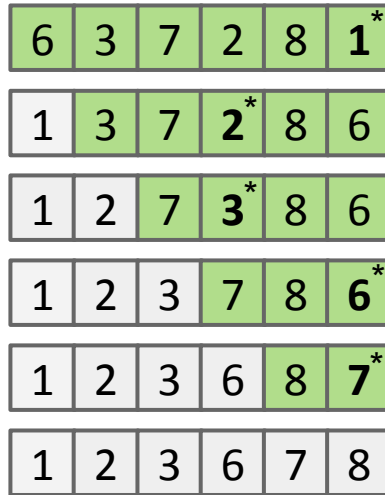
# Selection Sort

# Back to Sorting: Selection Sort

---

Selection sorting a list of N items:

- Find the smallest item.
- Move it to the front.
- Selection sort the remaining N-1 items (without touching front item!).



As an aside: Can prove correctness of this sort using invariants.

# Back to Sorting: Selection Sort

---

Selection sorting a list of N items:

- Find the smallest item.
- Move it to the front.
- Selection sort the remaining N-1 items (without touching front item!).

**Let's try implementing this.**

- I'll try to simulate as closely as possible how I think students might approach this problem to show how TDD helps.

Not shown in details in these slides. See lecture video.

6	3	7	2	8	1*
---	---	---	---	---	----

1	3	7	2*	8	6
---	---	---	----	---	---

1	2	7	3*	8	6
---	---	---	----	---	---

1	2	3	7	8	6*
---	---	---	---	---	----

1	2	3	6	8	7*
---	---	---	---	---	----

1	2	3	6	7	8
---	---	---	---	---	---

# The Evolution of Our Design

Created testSort:

```
testSort()
```

Created a sort skeleton:

```
sort(String[] inputs)
```

Created testFindSmallest:

```
testFindSmallest()
```

Created findSmallest:

```
String findSmallest(String[] input)
```

Created testSwap:

```
testSwap()
```

Created swap:

```
swap(String[] input, int a, int b)
```

Changed findSmallest:

```
int findSmallest(String[] input)
```

Used Google to  
figure out how to  
compare strings.

Used debugger to fix.

Now we have all the **helper methods** we need, as well as **tests** that make us pretty sure that they work! All that's left is to write the sort method itself.



# Very Tricky Problem

method signature

Without changing the signature of `public static void sort(String[] a)`, how can we use recursion? What might the recursive call look like?

```
public static void sort(String[] x) {  
    int smallest = findSmallest(x);  
    swap(inputs, 0, smallest);  
    // recursive call??  
}
```

## Very Tricky Problem: Bad But Tempting Solution

method signature

Without changing the signature of `public static void sort(String[] a)`, how can we use recursion? What might the recursive call look like?

```
public static void sort(String[] x) {  
    int smallest = findSmallest(x);  
    swap(inputs, 0, smallest);  
    sort(x[1:]); ← Would be nice, but not possible!  
}
```

Some languages support sub-indexing into arrays. Java does not.

- Bottom line: No way to get address of the middle of an array.

## Very Tricky Problem: Good Solution

method signature

Without changing the signature of `public static void sort(String[] a)`, how can we use recursion? What might the recursive call look like?

```
public static void sort(String[] x) {  
    sort(x, 0);  
}
```

```
/** Destructively sorts x starting at index k */  
public static void sort(String[] x, int k) {  
    ...  
    sort(x, k + 1);  
}
```

# Major Design Flaw in findSmallest

---

We didn't properly account for how `findSmallest` would be used.

- Example: Want to find smallest item from among the last 4: 

1	2	7	3*	8	6
---	---	---	----	---	---
- We need another parameter so that it's actually useful for sorting.

# The Evolution of our Design

Created testSort:

```
testSort()
```

Created a sort skeleton:

```
sort(String[] inputs)
```

Created testFindSmallest:

```
testFindSmallest()
```

Created findSmallest:

```
String findSmallest(String[] input)
```

Created testSwap:

```
testSwap()
```

Created swap:

```
swap(String[] input, int a, int b)
```

Changed findSmallest:

```
int findSmallest(String[] input)
```

Added helper method:

```
sort(String[] inputs, int k)
```

Used debugger to realize fundamental design flaw in findSmallest

Used Google to figure out how to compare strings.

Used debugger to fix.

# The Evolution of our Design

Created testSort:

```
testSort()
```

Created a sort skeleton:

```
sort(String[] inputs)
```

Created testFindSmallest:

```
testFindSmallest()
```

Created findSmallest:

```
String findSmallest(String[] input)
```

Created testSwap:

```
testSwap()
```

Created swap:

```
swap(String[] input, int a, int b)
```

Changed findSmallest:

```
int findSmallest(String[] input)
```

Added helper method:

```
sort(String[] inputs, int k)
```

Used debugger to realize fundamental design flaw in findSmallest

Modified findSmallest:

```
int findSmallest(String[] input, int k)
```

Used Google to  
figure out how to  
compare strings.

Used debugger to fix.

# And We're Done!

---

Often, development is an incremental process that involves lots of task switching and on the fly design modification.

Tests provide stability and scaffolding.

- Provide confidence in basic units and mitigate possibility of breaking them.
- Help you focus on one task at a time.

In larger projects, tests also allow you to safely **refactor**! Sometimes code gets ugly, necessitating redesign and rewrites (see project 2).

One remaining problem: Sure was annoying to have to constantly edit which tests were running. Let's take care of that.

# Simpler JUnit Tests

(using two new syntax tricks)



# Simple JUnit

---

New Syntax #1: `org.junit.Assert.assertEquals(expected, actual);`

- Tests that `expected` equals `actual`.
- If not, program terminates with verbose message.

We've already seen this throughout today.

JUnit does much more:

- Other methods like `assertEquals` include `assertFalse`, `assertNotNull`, etc., see <http://junit.org/junit4/javadoc/4.12/org/junit/Assert.html>
- Other more complex behavior to support more sophisticated testing.
- See lab3.


# Better JUnit

---

The messages output by JUnit are kind of ugly, and invoking each test manually is annoying.

Yes this is weird, as it implies you'd be instantiating `TestSort.java`. In fact, JUnit runners do this. I don't know why.

New Syntax #2 (just trust me):

- **Annotate** each test with `@org.junit.Test`.
- Change all test methods to non-static. 
- Use a JUnit runner to run all tests and tabulate results.
  - IntelliJ provides a default runner/renderer. OK to delete `main`.
  - If you want to use the command line instead, see the `jh61b` runner in the lab 3 supplement. Not preferred.
  - Rendered output is easier to read, no need to manually invoke tests!

**There is a lot of black magic happening here! Just accept it all for now.**

## Even Better JUnit

---

It is annoying to type out the name of the library repeatedly, e.g. **org.junit.Test** and **org.junit.Assert.assertEquals**.

New Syntax #3: To avoid this we'll start every test file with:

```
import org.junit.Test;  
import static org.junit.Assert.*;
```

This will magically eliminate the need to type '**org.junit**' or '**org.junit.Assert**' (more after the midterm on what these imports really mean).

# Testing Philosophy (Web Video Only)

# Correctness Tool #1: Autograder

---

Idea: Magic autograder tells you code works.

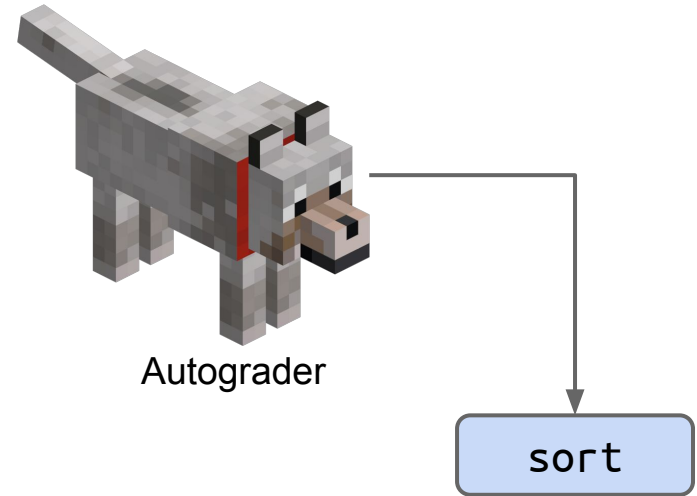
- We use JUnit + jh61b libraries.

Why?

- Less time wasted on “boring” stuff.
- Determines your grade.
- Gamifies correctness.

Why not?

- Autograders don't exist in real world.
- Errors may be hard to understand.
- Slow workflow.
- No control if grader breaks / misbehaves.



# Autograder Driven Development (ADD)

---

The worst way to approach programming:

- Read and (mostly) understand the spec.
- Write entire program.
- Compile. Fix all compilation errors.
- Send to autograder. Get many errors.
- Until correct, repeat randomly:
  - Run autograder.
  - Add print statements to zero in on the bug.
  - Make changes to code to try to fix bug.

```
[63, 12, 91, 5, 0]
got to this spot, It is: 1
got to this spot, It is: 2
got here!
[63, 12, 0, 5, 91]
got to this spot, It is: 3
got to this spot, It is: 4
got here!
[5, 12, 0, 63, 91]
Test Failed. Expected: ...
```

This workflow is slow and unsafe!

Note: Print statements are not inherently evil. While they are a weak tool, they are very easy to use.

## Correctness Tool #2: Unit Tests

Idea: Write tests for every “unit”.

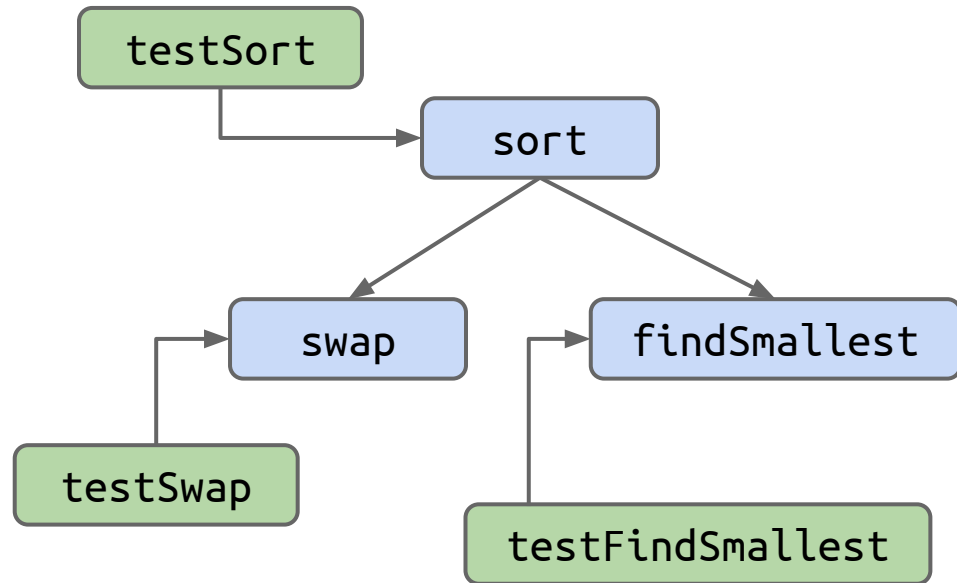
- JUnit makes this easy!

Why?

- Build confidence in basic modules.
- Decrease debugging time.
- Clarify the task.

Why not?

- Building tests takes time.
- May provide false confidence.
- Hard to test units that rely on others.
  - e.g. how do you test `addFirst`?



# Test-Driven Development (TDD)

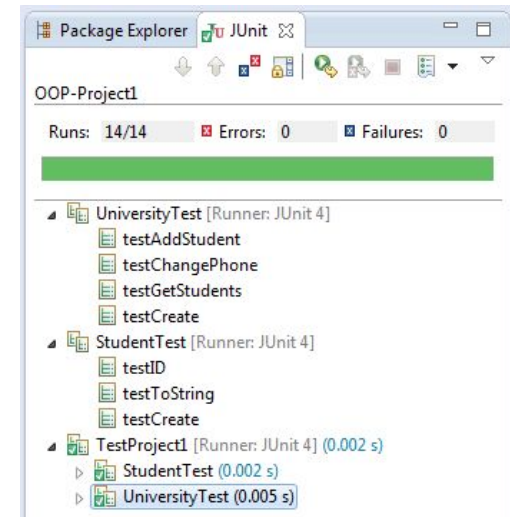
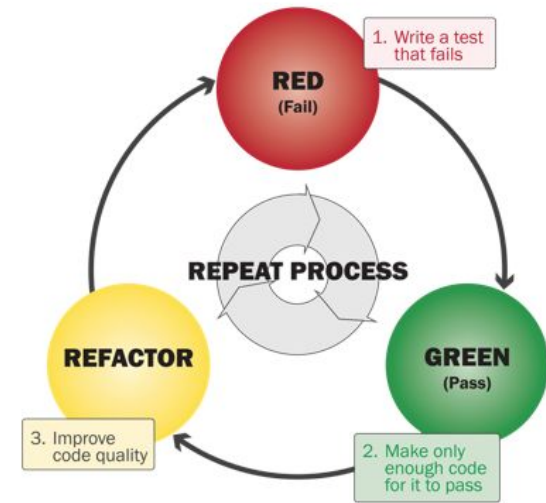
Steps to developing according to TDD:

- Identify a new feature.
- Write a unit test for that feature.
- Run the test. It should fail. **(RED)**
- Write code that passes test. **(GREEN)**
  - Implementation is certifiably good!
- Optional: Refactor code to make it faster, cleaner, etc.

Not required in 61B. You might hate this!

- But testing is a good idea.

Interesting perspective: [Red-Shirt, Red, Green, Refactor](#).



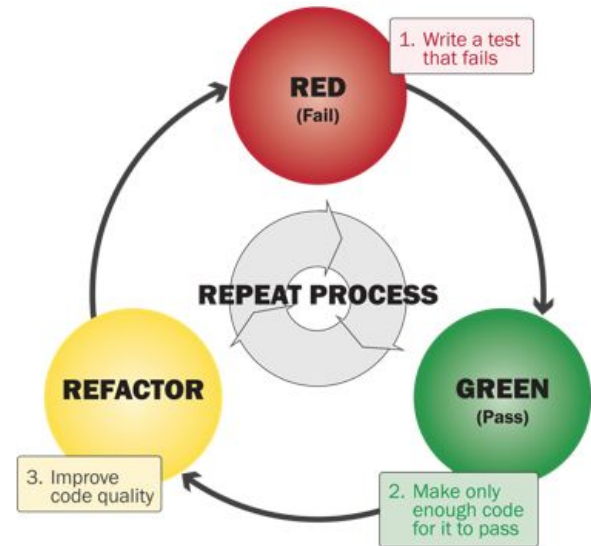


# A Tale of Two Workflows

TDD is an extreme departure from the naive workflow.

- What's best for you is probably in the middle.

```
$ python sort.py
[63, 12, 91, 5, 0]
got to this spot, lt is: 1
got to this spot, lt is: 2
got here!
[63, 12, 0, 5, 91]
got to this spot, lt is: 3
got to this spot, lt is: 4
got here!
[5, 12, 0, 63, 91]
```



# Correctness Tool #3: Integration Testing

Idea: Tests cover many units at once.

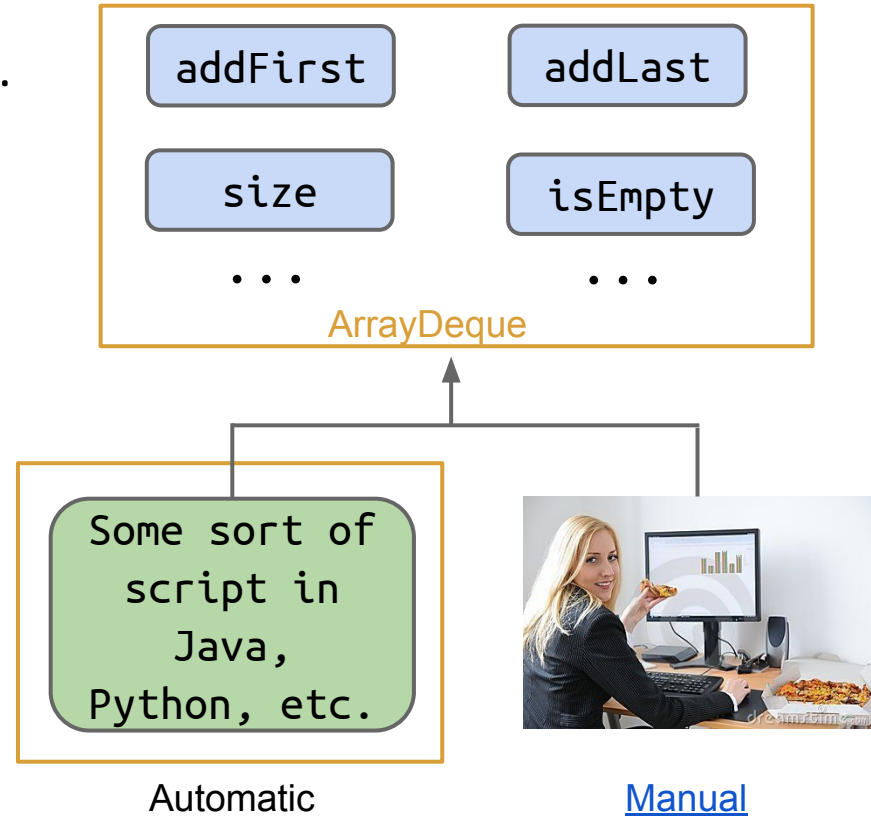
- Not JUnit's focus, but JUnit can do this.

Why?

- Unit testing is not enough to ensure modules interact properly or that system works as expected.

Why not?

- Can be tedious to do manually.
- Can be challenging to automate.
- Testing at highest level of abstraction may miss subtle or rare errors.



# Parting Thoughts

---

- JUnit makes testing easy.
- You should write tests.
  - But not too many.
  - Only when they might be useful!
  - Write tests first when it feels appropriate [I do this a lot].
  - Lab 3, Project 1B, and Project 2 will give you practice!
  - Most of the class won't require writing lots of tests (to save you time).
- Some people really like TDD. Feel free to use it in 61B.
  - See today's optional reading for thoughts from the creator of Ruby on Rails and others.

# More On JUnit (Extra)

## Bonus Slide: What is an Annotation?

---

Annotations (like `org.junit.Test`) don't do anything on their own.

```
@Test
public void testSort() {
    ...
}
```

Runner uses reflections library to iterate through all methods with “Test” annotation. Pseudocode on next slide.

## Sample Runner Pseudocode

---

Runner uses reflections library to iterate through all methods with “Test” annotation.

```
List<Method> L = getMethodsWithAnnotation(TestSort.class,  
                                           org.junit.Test);  
  
int numTests = L.size();  
int numPassed = 0;  
for (Method m : L) {  
    result r = m.execute();  
    if (r.passed == true) { numPassed += 1; }  
    if (r.passed == false) { System.out.println(r.message); }  
}  
System.out.println(numPassed + "/" + numTests + " passed!");
```

# Citations

---

Training montage: Wet Hot American Summer

Creepy hand picture (title slide):

<http://www.automatedtestinginstitute.com/home/images/stories/Functional.jpg>

Red-Green-Refactor image courtesy of a guy who has had issues with TDD:

<http://ryantablada.com/post/red-green-refactor---a-tdd-fairytale>

# How It Usually Goes...

