

# Hands-On: Testing

---

This activity is designed to help improve your testing skills. Testing is an important, although not sufficient, tool used to ensure the correctness of the software that you write.

## Linear Search and the Importance of Testing

---


Linear search is a search algorithm based on the *linear scan* solution strategy. As we know, there are many different ways of *expressing* this in code. Some are better or worse in subjective ways (loop form, style, etc.) and some are better or worse in objective ways (correctness, efficiency, etc.).

Here's an expression of linear search to consider.

```
public int search(int[] a, int target) {
    int i = 0;
    while ((a[i] != target) && (i < a.length)) {
        i++;
    }
    if (a[i] == target) {
        return i;
    }
    else {
        return -1;
    }
}
```

You may see the problem right away, but this activity is designed to illustrate how testing can expose errors such as this one. We will be using the [JUnit](#) testing framework, so you must have that configured on the machine you're using in order to complete this lab.

**Note:** These instructions are for the [jGRASP](#) IDE. You can replicate this in most any IDE that you may be using.

1. Download the following files to a suitable directory on your local machine.
  - [LinearSearch1.java](#)
  - [LinearSearch1Test.java](#)
2. Open `LinearSearch1.java` in jGRASP and compile it.
3. Click on the *Create JUnit test file* button () in the toolbar and select the *Use existing file* option. This will automatically select `LinearSearch1Test.java` as a JUnit test file for the `LinearSearch1.java` source file.

- (Note for personal use: If you have multiple test files for a source file or if the test file doesn't follow the `*Test` naming convention, you should use the [Projects](#) feature in jGRASP.)

4. Examine the three test methods in `LinearSearch1Test`.

- Note that a "test method" begins with the `@Test` annotation. Any method without this annotation is not considered a test method and will not be automatically called by a JUnit runner.
- Each test method is structured according to the same pattern:

```
int[] a = // a sample array to be searched through
int target = // a sample value to be searched for
int expected = // the correct value that should be returned
int actual = LinearSearch1.search(a, target) // the actual value that is returned
assertEquals(expected, actual); // check if the method returned the expected value
```

- There isn't just one way to structure and write a test method, but following this pattern is a good idea - especially when you're just starting out. Following this pattern offers a few advantages:
  - It's simple and easy to remember.
  - It reinforces the one-to-one correspondence of a "test method" and a "test case".
  - It focuses our thoughts on the purpose of a test case: To verify that the code under test performs the expected action in a *specific scenario*.

5. In the editor window for either file, run the tests (🚀).

- Note that running a JUnit test file is different than running a source file. A class that is internal to JUnit and known as a "runner" is actually what is being executed. The runner calls the test methods inside the test file.

6. Explore the JUnit test window that pops up, and notice that all tests pass.

7. Of course we know already that there is an error in the linear search implementation, so let's pause to consider exactly what testing can tell us. As [Edsger Dijkstra](#) wrote in [Notes on Structured Programming](#)

Program testing can be used to show the presence of bugs, but never to show their absence!

So the usefulness of tests is directly proportional to their strength, rigor, and comprehensiveness. If a set of tests is weak and superficial, then passing 100% of them means very little. But passing a set of very well thought-out and rigorous tests means much more. We can never demonstrate through testing that a program is *correct*, but we can increase our confidence in the program and its likelihood of being correct.

8. Add test methods to `LinearSearch1Test` until you have exposed the error in the `search` method of `LinearSearch1`.
9. Now that you have exposed the error, be sure that you can *describe it in words*. Observing that a test case failed isn't enough. Stating that "searching for 5 caused an `IndexOutOfBoundsException`" isn't enough. Before you can hope to repair the error, you have to *understand the underlying cause* of test case failure. What is it about 5, for example, that makes the test case fail?
  - This is a good opportunity to discuss this with someone else. Try to describe the error to each other and see if you agree with one another's description.
10. Based on your understanding of the problem, modify the `search` method in `LinearSearch1` to eliminate the error.
  - There are many ways to correct the error, including a total re-write of the method. For now, let's avoid that. Try to correct the error by relying on the [short-circuit evaluation](#) of the `&&` operator in Java.
  - It's worth noting that relying on short-circuit evaluation [might not be the best solution](#).
11. By now you will have noticed that the same error is made at two different locations in the method. So, you will have to make at least two changes to the `search` method to eliminate the error.

## Linear Search Reconsidered

---

Linear search is a particular instance of the *linear scan* solution strategy. In our particular case we are performing a linear scan of an array. In given programming languages, there are common [idioms](#) for doing things like scanning an array. These idioms, or common and accepted ways of expressing things, have developed over long use by practicing professionals. They're not only the expected way of expressing something, but they often can be less error-prone than alternative expressions.

The most basic form of a linear scan on an array in Java uses the `foreach` loop.


```
for (int elem : a) {  
    // do something with this element  
}
```

This form is not appropriate if the index of the element is needed. In that case, the following form is appropriate.

```
for (int i = 0; i < a.length; i++) {  
    // do something with element a[i]  
}
```

One or both of these forms are appropriate for many linear scans of an array, but not all. **Think of an example where this form would not be a good choice for performing a linear scan of an array.**

Since the version of linear search we're writing needs to return the location (index) of the target element in a, the second form is the one we will choose to use.

1. Download the following files to a suitable directory on your local machine.
  - [LinearSearch2.java](#)
  - [LinearSearch2Test.java](#)
2. Open `LinearSearch2.java` in jGRASP and compile it.
  - Note that the `search` method has been rewritten using the conventional `for` loop form of the linear scan solution strategy.
3. Click on the *Create JUnit test file* button () in the toolbar and select the *Use existing file* option. This will automatically select `LinearSearch2Test.java` as a JUnit test file for the `LinearSearch2.java` source file.
4. Not only has the `search` method been rewritten, but the test cases have been expanded as well. The test cases have been grouped into *boundary* cases, *typical* cases, and *special* cases.
5. Run these tests on the revised linear search implementation. Make changes in the code under test and the tests themselves until you are familiar and comfortable with writing and working with JUnit tests.

## Testing a min method

---

1. Download the following file to a suitable directory on your local machine.
  - [MinOfThree.java](#)
2. Open `MinOfThree.java` and compile it.
  - Read and study the two implementations of minimum: `min1()` and `min2()`. Think about the following questions.
    - a. Which method is easier to understand?
    - b. Which method do you think stands a better chance of being correct, just from its appearance?
    - c. Are there any errors that are obvious just from reading the code?
3. Create a JUnit test file named `MinOfThreeTest.java` for this class.
4. Write a set of test methods that will help you decide on the correctness of both the `MinOfThree.min1()` method and the `MinOfThree.min2()` method.
5. Correct all errors that your testing exposes.

6. Once you are satisfied with the results of your testing and debugging, reflect on your answers to the three questions asked above in light of the work you've done.

## Submission

---

The submission page for this activity asks you to demonstrate the result of your testing and debugging work by submitting a corrected version of `MinOfThree.java` for a grade.