Unit Testing Code

Summary & Reference

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

public class TestingExamples {

    @Test
    public void testingExampleOne() {
        String inputToTest = "example";
        int otherInputToTest = 4;

        double expected = 14.5;
        double actual = functionToTest(inputToTest, otherInputToTest);

        assertEquals(expected, actual);
    }

    @Test
    public void testingExampleTwo() {
        int[] input = { 3, 4, 5 };
        int result = countEvens(input);
        assertTrue(result < 3);

    }
}
```

Quick Reference for when you will write your own tests. ☺

Testing a unit of code

```
int findMax(int a, int b, int c) {
            if (a > b) {
                  if (a > c) return a;
                  else return c;
            }
            else {
                  if (b > c) return b;
                  else return a;          // should be c
            }
```

How do we test this code?

Testing a unit of code

```
int findMax(int a, int b, int c) {
            if (a > b) {
                  if (a > c) return a;
                  else return c;
            }
            else {
                  if (b > c) return b;
                  else return a;          // should be c
            }
```

1. Identify the **INPUT**, possibly including any state variables
2. Generate, manually or through means OUTSIDE of your code an **EXPECTED OUTPUT**
3. Execute your code to get an **ACTUAL OUTPUT**

4. Compare the expected and actual output

Test Case

- Comprised of:
  - An Input
  - An **EXPECTED** output (Usually manually coded in)
  - And an **ACTUAL** output. (generated by the code we are testing)
- If an expected output doesn't match the actual output, one of the two is wrong
  - Usually, but not necessarily, the actual output is wrong

Testing a unit of code

```
int findMax(int a, int b, int c) {
            if (a > b) {
                  if (a > c) return a;
                  else return c;
            }
            else {
                  if (b > c) return b;
                  else return a;          // should be c
            }
```

Test Case #1: Input = {3,2,1}; Expected output = 3; Actual output = 3

PASS!!!

Test Case #2: Input = {1,2,3}; Expected output = 3; Actual output = 1

FAIL!!!

Testing is like potato chips

- They both contribute to my overall poor health*

- Additionally, you can’t have just one
    - One test passing may have no bearing on another test passing

*credit to Will McBurney (?) for this good joke

Why does Test 1 pass and not Test 2?

- Test1 does not cover/execute the underlying **FAULT** in the code.
- A ***fault*** is a static defect in the code, or “bug”

Test Case #1: Input = {3,2,1}; Expected output = 3; Actual output = 3

PASS!!!

Test Case #2: Input = {1,2,3}; Expected output = 3; Actual output = 1

FAIL!!!

JUnit

- An automatic testing tool that allows you to write tests once and continue to use them again and again
- In this way, if you change something later that breaks code that worked previously, you will immediately know because your tests fail
- Technically not built into Java

Writing a JUnit Test

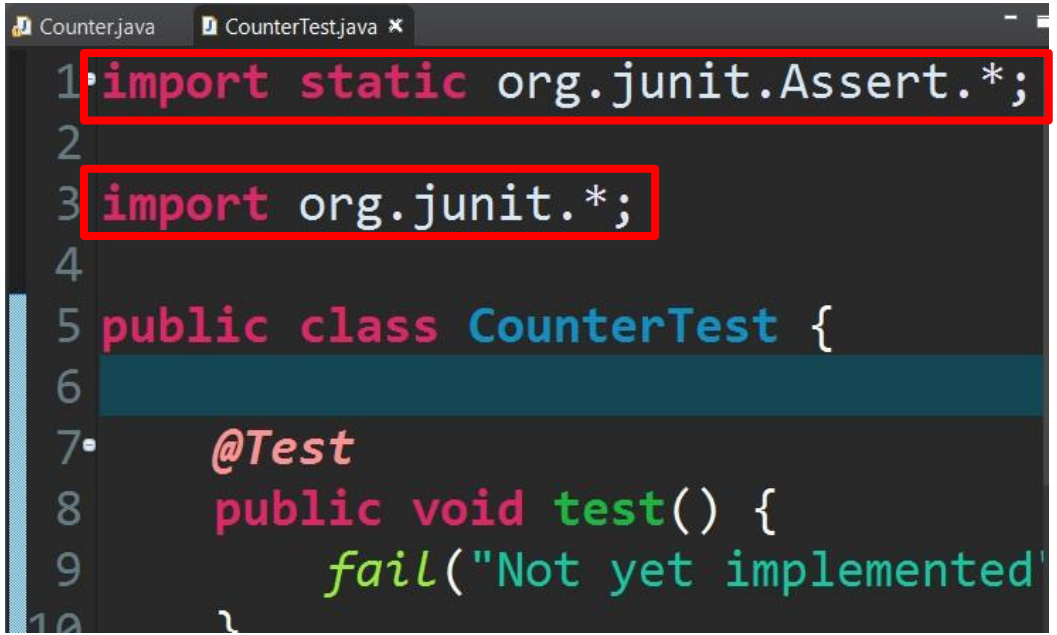
```
@Test // This must be before every test function
public void testFindMax0() {      // Notice - no static keyword
    // inputs
    int a = 3;
    int b = 2;
    int c = 1;
    // expected - generated manually
    int expected = 3;
    // actual - Execute the code with the above input
    int actual = findMax(a, b, c);
    // Assertion - if the two things below aren't equal, the
    //             test fails. Always put expected argument first.
    assertEquals(expected, actual);
}
```

Writing a JUnit Test (with an error message)

```
@Test  // This must be before every test function
public void testFindMax0() {       // Notice - no static keyword
    String message = "ERROR: findMax(3,2,1) returned an incorrect result";
    // expected - generated manually
    int expected = 3;
    // actual - Execute the code with the above input
    int actual = findMax(3, 2, 1);
    // Assertion - now prints out an error message if the assert fails
    assertEquals(message, expected, actual);
}
```

Import Statements

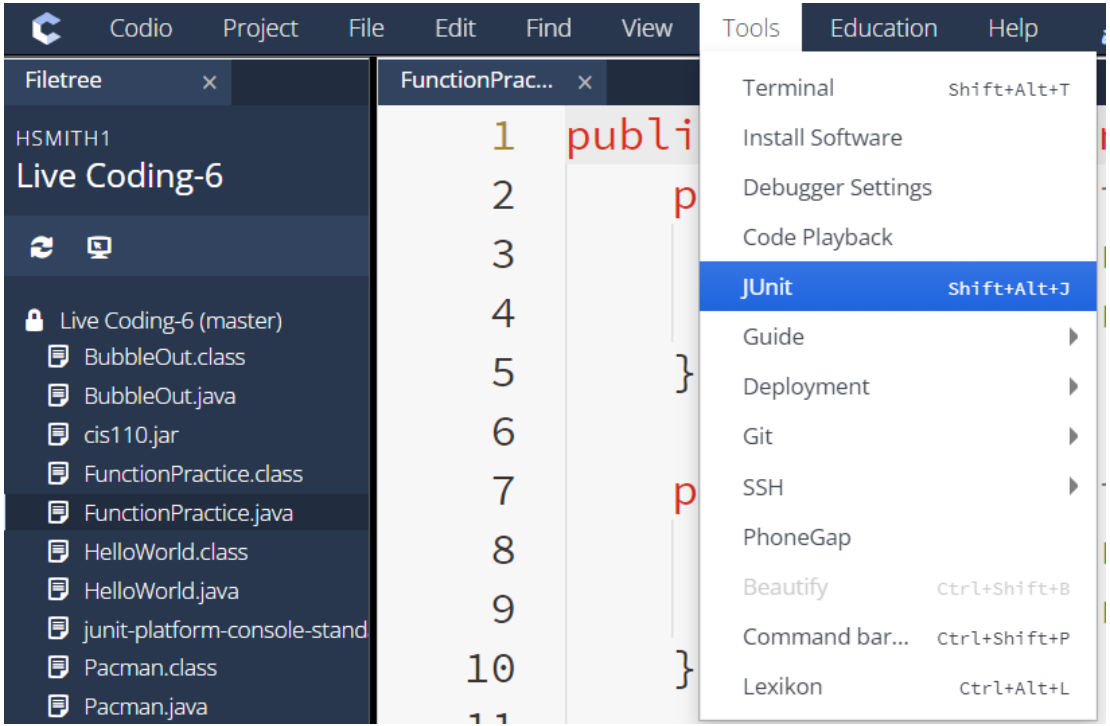
Start all Test files with the two important statements below.



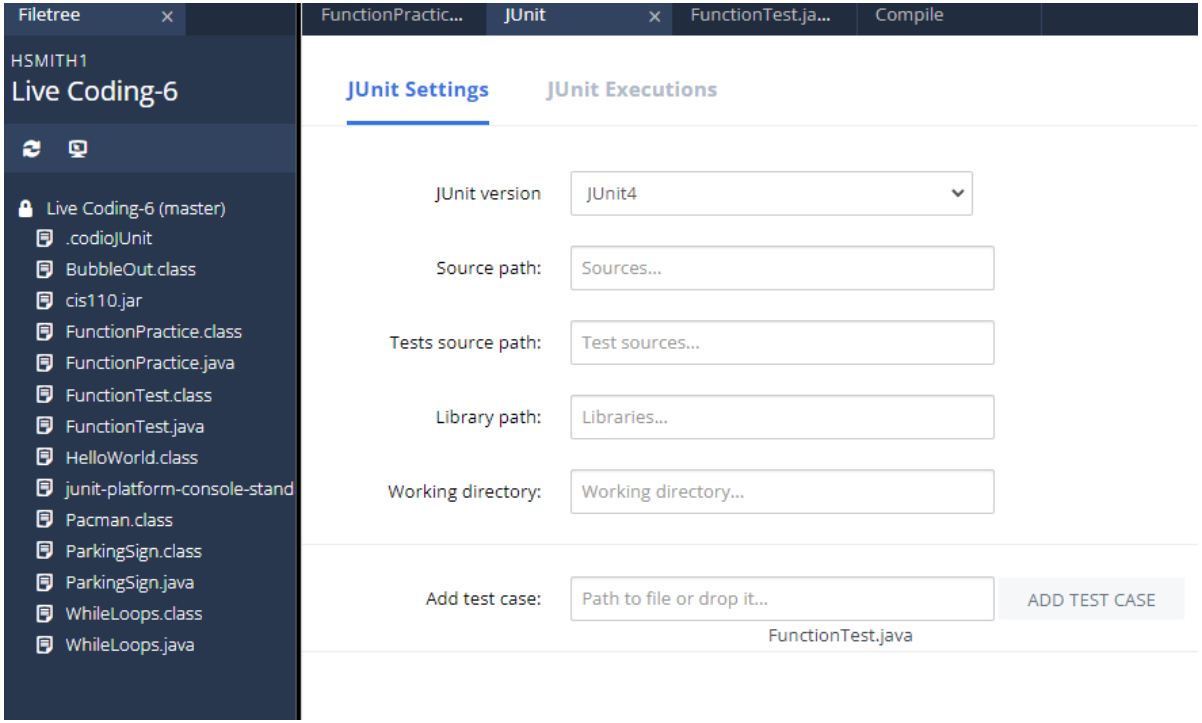
Writing Junit (Demo)

```
1  import static org.junit.Assert.*;
2  import org.junit.*;
3
4  public class FunctionTest {
5
6      @Test
7      public void testMean() {
8          double a = 5;
9          double b = 7;
10         double expected = 19;
11         double actual = FunctionPractice.mean(a, b);
12
13         assertEquals(expected, actual, 0.001);
14     }
15 }
```

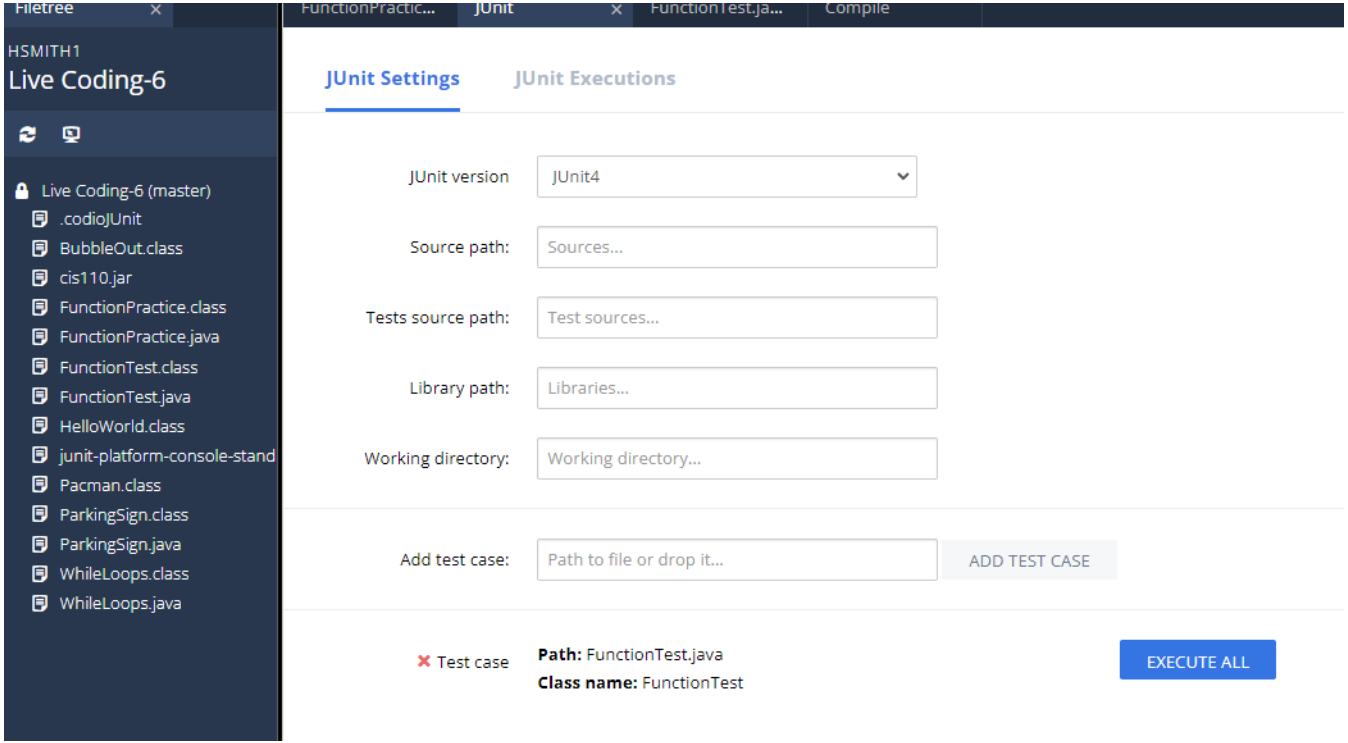
How to find JUnit



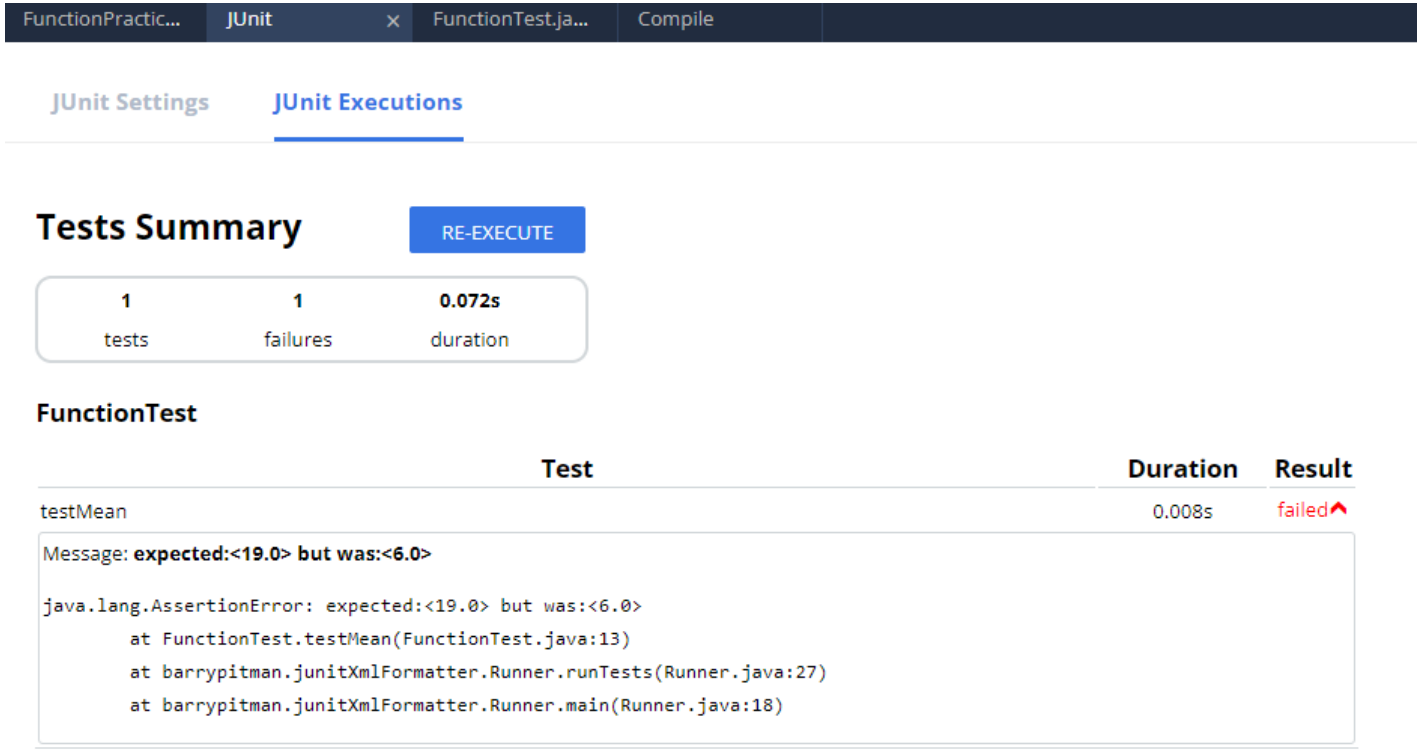
How to set up JUnit



How to run tests



Failure!


FunctionPractic...
JUnit
FunctionTest.ja...
Compile
JUnit Settings
JUnit Executions
Tests Summary
RE-EXECUTE
1
tests
1
failures
0.072s
duration
FunctionTest
Test
Duration
Result
testMean
0.008s
failed
Message: expected:<19.0> but was:<6.0>
java.lang.AssertionError: expected:<19.0> but was:<6.0>
at FunctionTest.testMean(FunctionTest.java:13)
at barrypitman.junitXmlFormatter.Runner.runTests(Runner.java:27)
at barrypitman.junitXmlFormatter.Runner.main(Runner.java:18)

Success!

Tests Summary

RE-EXECUTE

1	0	0.23s
tests	failures	duration

FunctionTest

Test	Duration	Result
testMean	0.001s	passed

What a test failing means

- A test failing doesn’t always mean the code has a bug
  - The test could be written wrong (that is, the test writer came up with the wrong expected output)
- A test passing doesn’t mean there is no bug
  - The test code not execute a buggy statement
  - The test could execute a buggy statement in a way that a failure doesn’t manifest

Consider these test cases

```
int findMax(int a, int b, int c) {
            if (a > b) {
                  if (a > c) return a;
                  else return c;
            }
            else {
                  if (b > c) return b;
                  else return a;          // should be c
            }
```

Test Case #3: Input = {1,1,1}; Expected output = 1; Actual output = 1

PASS!!!

Test Case #4: Input = {4,5,6}; **Expected output = 4**; Actual output = 4

PASS!!!

Errors in the test case

- Encountering the fault does **not** mean your test will fail.
- Your test could be erroneous!
    - In this case, test #4 is a false positive

Test Case #3: Input = {1,1,1}; Expected output = 1; Actual output = 1

PASS!!!

Test Case #4: Input = {4,5,6}; **Expected output = 4**; Actual output = 4

PASS!!!

False Negatives

- If your test is erroneous, you could also get a false negative.
- This test DOESN'T cover the fault, but still fails, due to erroneous testing

Test Case #4: Input = {9,8,7}; **Expected output = 7**; Actual output = 9

FAIL!!!

Testing Strategies

- Exhaustive Testing
  - Attempt a test with every possible input
  - Not even remotely feasible in most cases
- Random Testing
  - Select random inputs
  - Likely to miss narrow inputs that are special cases (example, dividing by zero)

Testing Strategies

- Black-box Testing
  - Select inputs based on the specification space
  - “Assume the code can’t be seen”
  - We focus on this one
- White-box Testing
  - Select inputs based on the code itself
  - Have every line of code covered by at least one test

The need for automatic testing

- Automatic testing (such as JUnit) allows for testing rapidly after each update
- If an update breaks a test, a commit can be rejected
- Ensure you don’t break something that already worked
    - Not fool proof

Searching

Overview

- We often need to search for an item in a collection
- In this module, we will learn about how to search for an element in an array
- Example:
    - Find the cat named Garfield inside an array named *shelter*

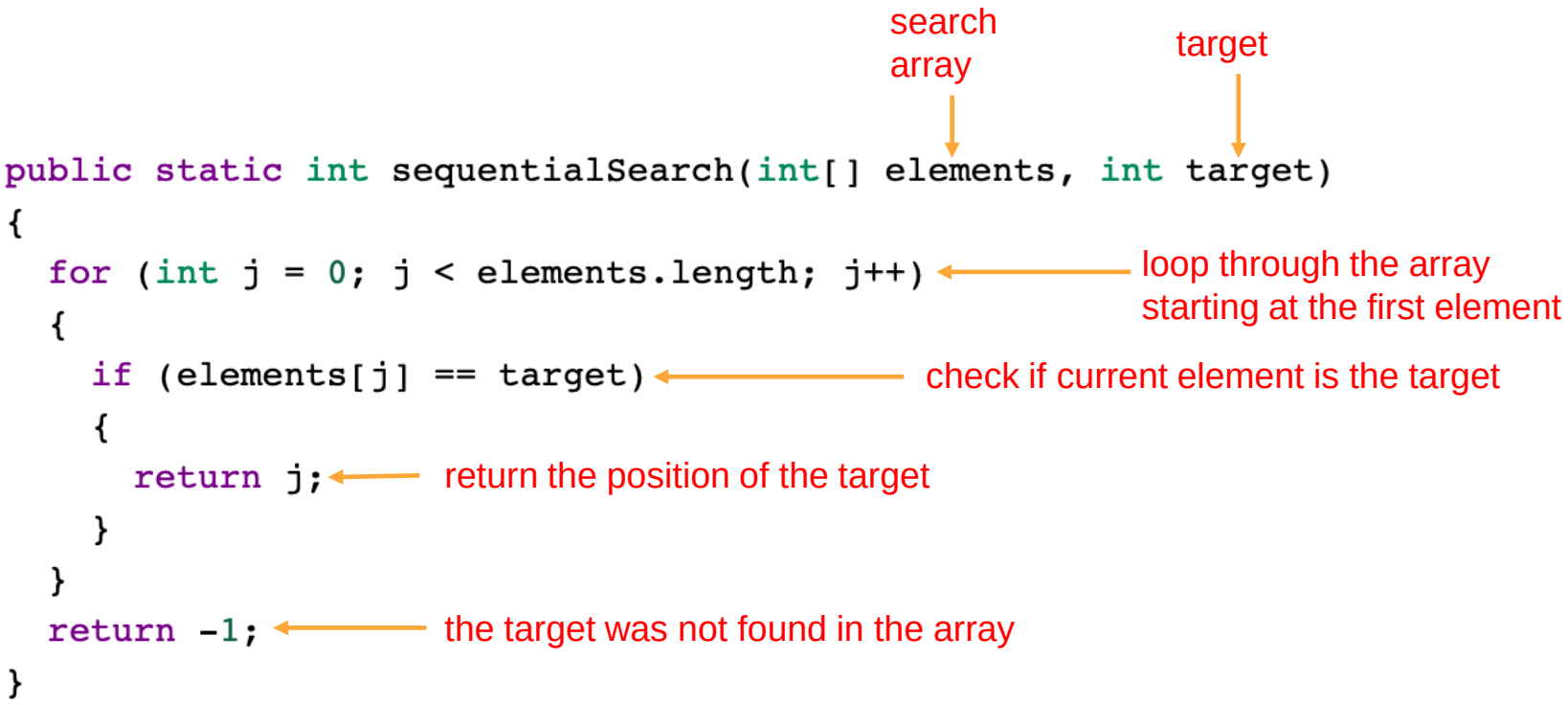
Learning Objectives

- **To be able to use linear search to find an element inside an array**
- To be able to use binary search to find an element inside an array
- To be able to know when to use linear search and when to use binary search

Linear Search

- Used to search for a value (the target) in an **unsorted array**
- It uses a loop to iterate over the values
- Starts at the first element and move to the next element until the target is found
- Returns the position of the target if it was found in the array
- Returns -1 if the target was not found in the array

Linear Search: array



Learning Objectives

- To be able to use linear search to find an element inside an array or an ArrayList
- **To be able to use binary search to find an element inside an array**
- **To be able to know when to use linear search and when to use binary search**

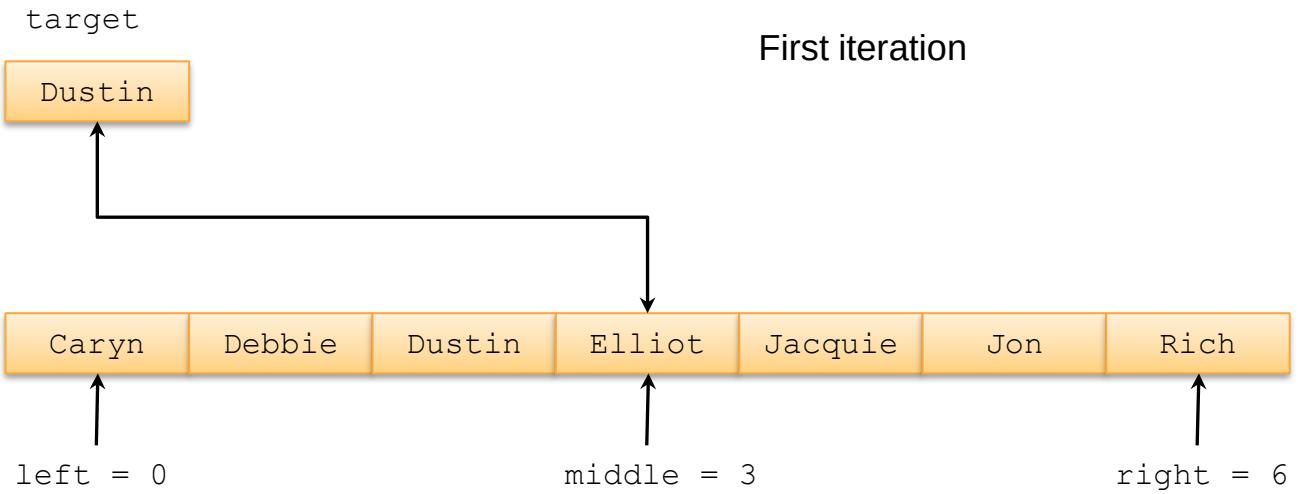
Binary Search

- Used to search for a value (the target) in a **sorted array**
- Keeps dividing the array in half
- Compares the target with the value at the middle index (middle element)
- If the target is less than the middle element, then we search the target in the **left half of the array** (the positions before the middle element)
- If the target is greater than the middle element, then we search the target in the **right half of the array** (the positions before the middle element)

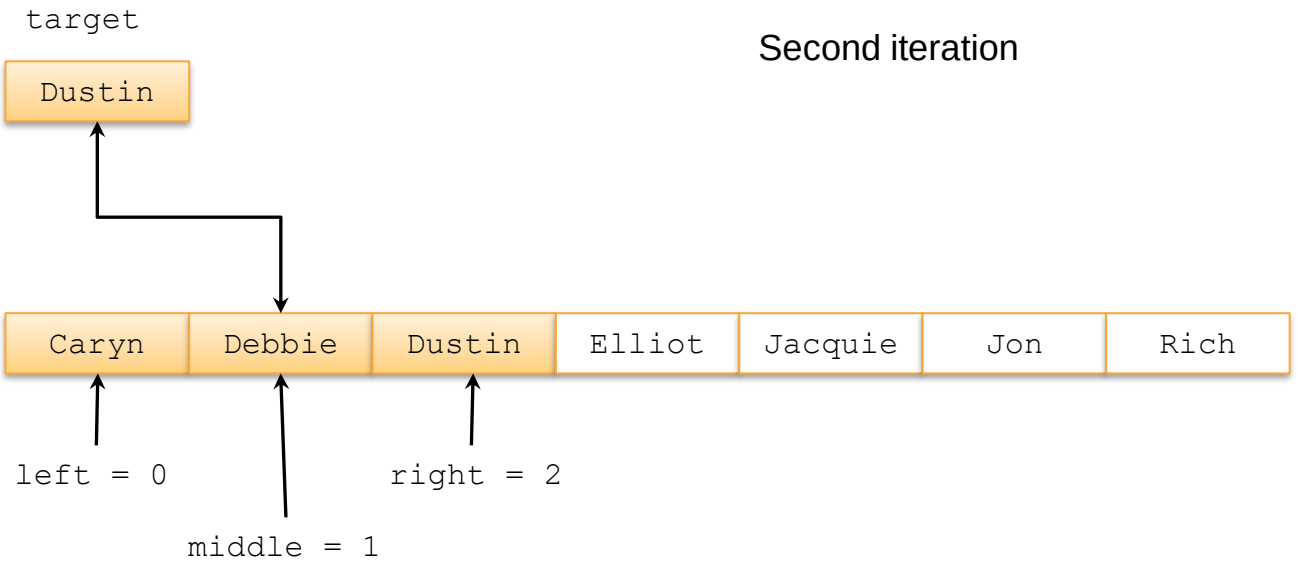
Binary Search

- Returns the position of the middle element if it is equal to the target
- Returns -1 if the target was not found in the array

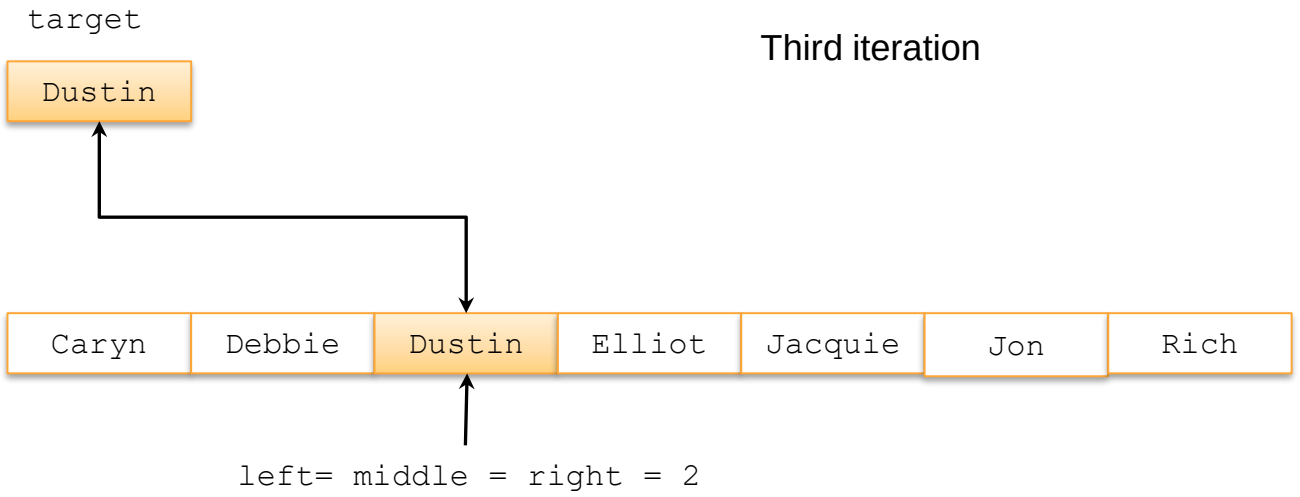
Binary Search



Binary Search



Binary Search



Binary Search

search array

target

```
public static int binarySearch(String[] elements, String target) {
   int left = 0;
   int right = elements.length - 1;
   while (left <= right)
   {
      int middle = (left + right) / 2;
      if (target.compareTo(elements[middle]) < 0)
      {
         right = middle - 1;
      }
      else if (target.compareTo(elements[middle]) > 0)
      {
         left = middle + 1;
      }
      else {
         return middle;
      }
    }
    return -1;
}
```

keep searching until no space left

compute middle position

move right before middle when target < middle element

move left after middle when target > middle element

Return middle when target == middle element

the target was not found in the ArrayList

Linear Search vs. Binary Search

- Binary search is faster than linear search
- Binary search runs on sorted data
- Linear search runs on unsorted data

Linear Search vs. Binary Search

- **Runtime analysis**: how many comparisons will it take to determine that the target is not in the array?

Length of the array	Linear Search	Binary Search
2	2	2
4	4	3
8	8	4
16	16	5
100	100	7