Lecture 3: Method Specifications and Testing

CS 2110, Matt Eichhorn and Leah Perlmutter

September 2, 2025

Announcements

In recognition of labor day

- Gratitude to union workers for the 40 hour work week
- Tech Workers' Coalition
  - techworkerscoalition.org/
  - What is TWC? -- "Guided by our vision for an inclusive & equitable tech industry, TWC organizes to build worker power through rank & file self-organization and education."
  - September news

Labor Day is a US holiday to celebrate the contributions
- of rank and file workers toward workers' rights and the meaning of work culture

It's similar to the May 1 holiday International Workers Day
- But for some reason the United States recognizes it in September instead of May

I also want to introduce you to a labor oriented community in our discipline

Tech Workers Coalition
- a professional community centered around tech workers
- You can find them online at (link)
- In their own words... (read quote)

Announcements

- It's normal for strong students to struggle (productively) in CS 2110
- Women in Computing at Cornell (WICC) Partner Finding Social Friday 9/5 at 4pm in Gates G01
  - Social event for everyone studying computing

Announcements

- Assignment 1 (A1) due Wednesday 9/3 at 10 pm
  - see course website --> Assignments
  - virtual tour: Checking test output on gradescope
- Office hours will now be in Malott 301H
  - see course website --> About --> Office Hours

Announcements

- Poll Everywhere: let's try again!
  - Don't stream youtube or video games or download torrents
  - Try texting in your answer (need to register – see ed post)
  - Spread out across both wifi networks and mobile data

Streaming is distracting
- I don't have to tell you that it distracts you from your own learning
- But more importantly, it also distracts the people sitting behind you
- And one streamer could take the bandwidth for 50 pollev participants
- Media intensive web browsing such as instagram also snatches lots of bandwidth

If you must stream, pause it for each pollev question
Hold your peers accountable – if you see somebody streaming
- You have my encouragement to gently tap their shoulder and ask them to stop

Poll Everywhere

PollEv.com/2110fa25 text `2110fa25` to `22333`

Name one place you enjoy traveling to!

Today's Learning Outcomes

1. Write detailed JavaDoc specifications for a method given its signature and an English description of its behavior.
2. Describe the client and implementer roles in software development.
3. Identify the pre-conditions, post-conditions, and side-effects of a method given its specification and signature.
4. Incorporate defensive programming practices such as pre-condition checks and invariant assertions into code.
5. Write comprehensive unit tests for a method given only its specifications and signature.
6. Explain the process of test-driven development and identify its potential benefits.
7. Describe the differences between black-box and glass-box testing.

Lots of learning objectives today!
For many of these, it is the first day (and not the last) that we will cover them

Specifications

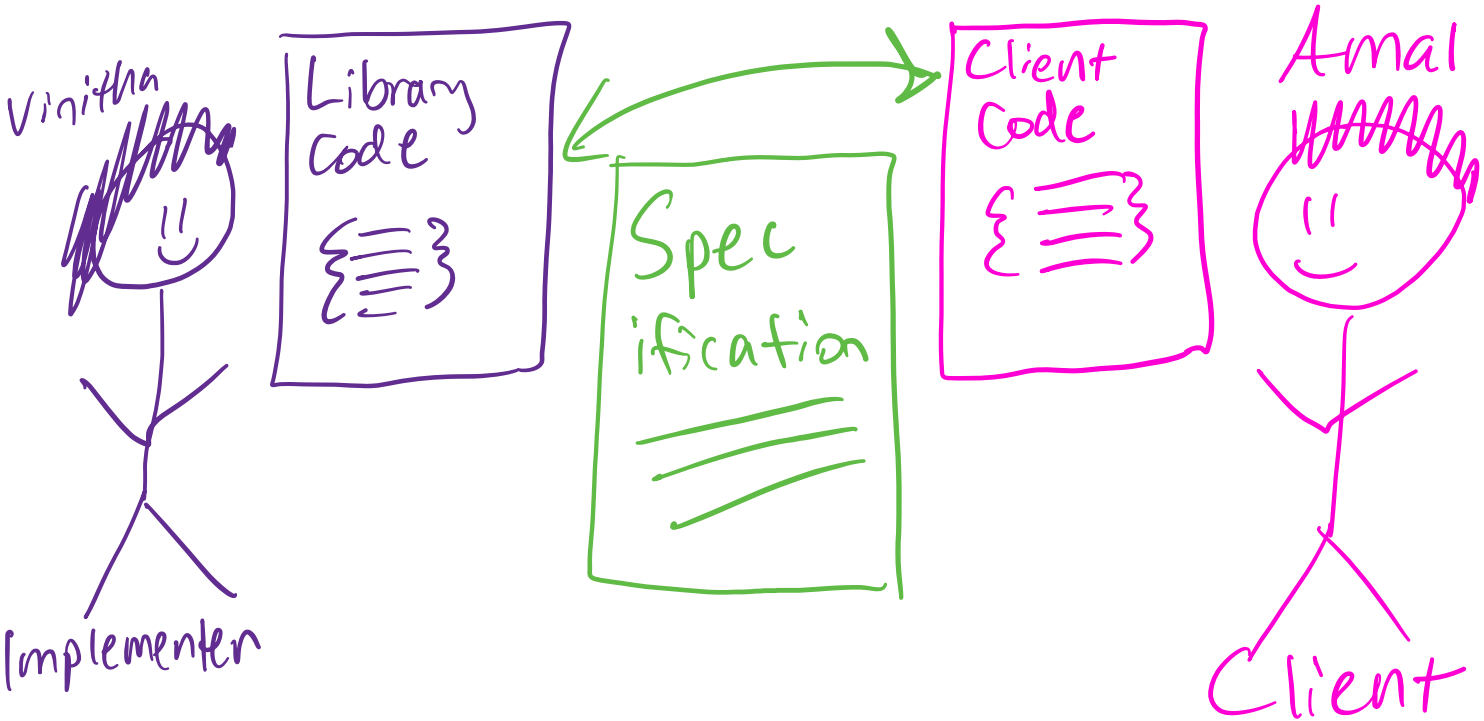
- The *specifications* of a unit of code (for example, a *method*, a *class*, or a *field*) are a precise description of the intended behavior of that code.

Virtual tour: JavaDocs IRL

Let's look at the specification for **`substring(int beginIndex, int endIndex)`** in two places!

- Java Library Documentation for java String
- Source code for java String

Why Specifications? The Client and the Implementer



Why Specifications? The Client and the Implementer

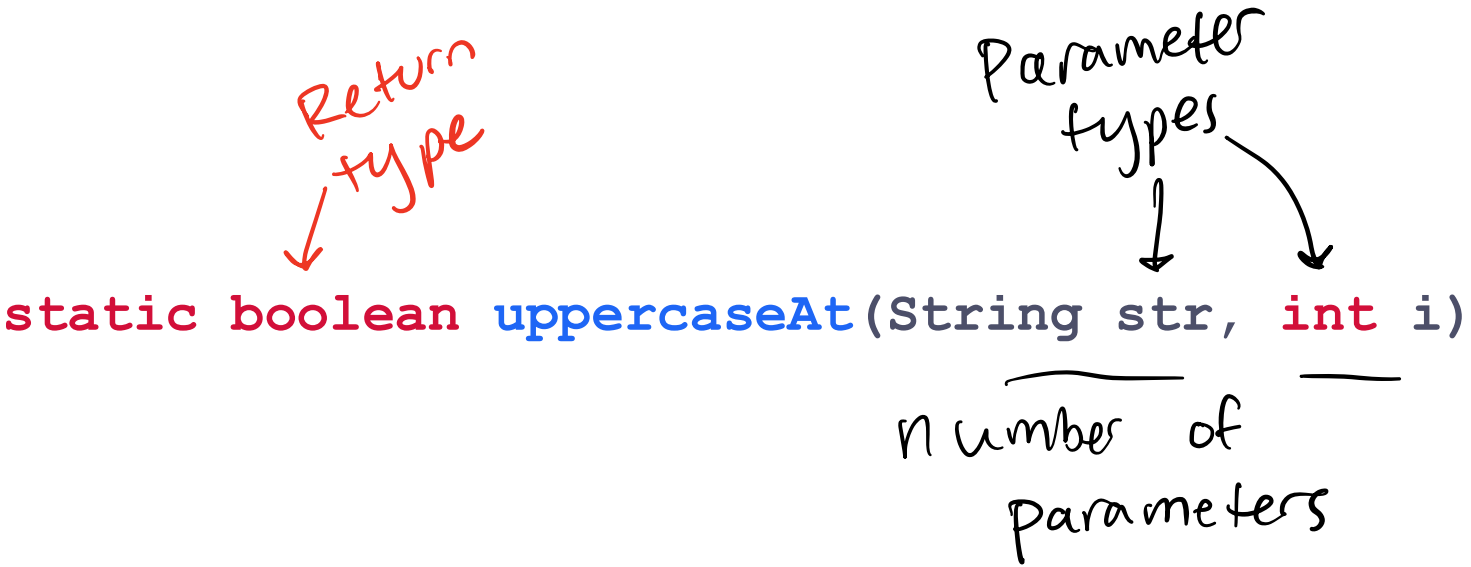
Draw a picture of two people
Implementer - Vinitha
Client – Amal

Implementer is an implementer of a library
Client is a user of the library

You can think of a specification as a contract between the client and the implementer

- They both have to agree on what the library code is supposed to do
- And communicate about that expectation
- The formal way for doing that is with a document called a specification

Specification: Method Signature



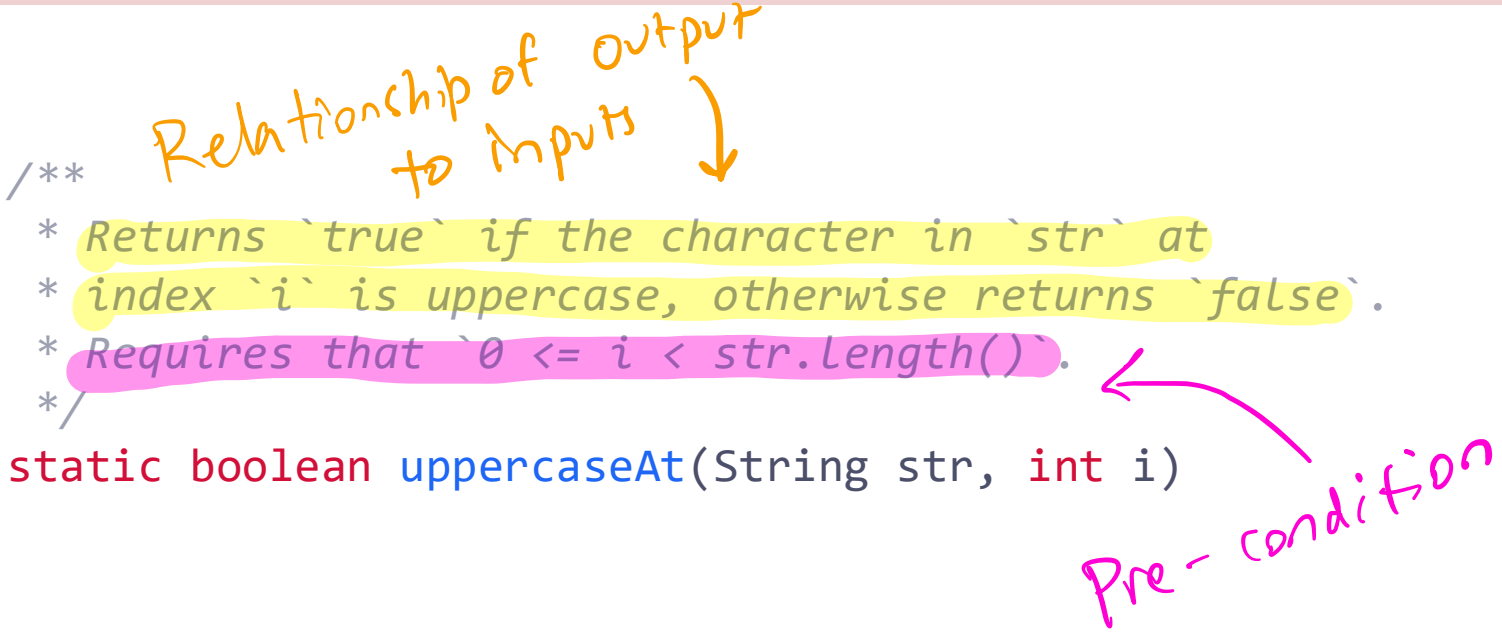
Specification: Method Signature

```
static boolean uppercaseAt(String str, int i)
```


- Return value must be a boolean
- First argument must be a string
- Second argument must be an int
- There must be exactly 2 arguments

These are all enforced by the compiler

Specification: Pre-conditions & Post-Conditions



Specification: Pre-conditions & Post-Conditions

```
/**
 * Returns `true` if the character in `str` at
 * index `i` is uppercase, otherwise returns `false`.
 * Requires that `0 <= i < str.length()`.
 */
static boolean uppercaseAt(String str, int i)
```


In our code we write the specification using this special comment

- that starts with /**
- and ends with */

Questions So Far.

Unit Testing

- Making sure a method conforms to its specification
- "unit" refers to a unit of code

We don't test all the code at one time, we test it bit by bit

Test Suite: A Set of Input/Output Pairs

```java
/**
 * Returns `true` if the character in `str` at
 * index `i` is uppercase, otherwise returns `false`.
 * Requires that `0 <= i < str.length()`.
 */
static boolean uppercaseAt(String str, int i)
```

Input (str)	Input (i)	Expected output
"Hello"	0	true
"Hello"	5	Error
"Untitled"	5	false
"333"	2	???
"CHEESE yay"	4	true

We do that by writing a test suite for each unit of code

A test suite consists of input/output pairs
- That is, what do we want to pass in to the method to exercise it in different ways
- And for each set of inputs, what output do we expect?

Think: write 3 input/output pairs
Pair: with a neighbor, compare i/o pairs and come up with a pair that's substantially different from the ones you came up with before
Share out some pairs

~~~
[Introduce hand-up for silence technique]
~~~

Share out

Emphasis on

- Positive cases (output is true)
- Negative cases (make sure to also test situations where the output is false)

Test Suite: Edge Cases

```
/**
 * Returns `true` if the character in `str` at
 * index `i` is uppercase, otherwise returns `false`.
 * Requires that `0 <= i < str.length()`.
 */
static boolean uppercaseAt(String str, int i)
```

Input (str)	Input (i)	Expected output
"CHEESE yay"	6	false
""	0	???? *****
"LA CANCIÓN"	8	true
"Σ'αγαπώ"	0	true

If not enough variety, TPS again
If there was enough variety, emphasize

- Empty string
- String with non-alphabetic characters (numbers, spaces, punctuation)
- String with alphabetic, non-ASCII characters (LA CANCIÓN)
- String with alphabetic, non-Latin characters -- Σ'αγαπώ (greek for I love you, the first letter is an uppercase sigma)

Coding Demo: Unit Testing

- Source code –
- Test code –
- JUnit –
- Annotations –
- Junit Assertions –

Now we're going to put this into practice with a code demo.

During demo
- Specs.java (in src folder)
- Source Code is what we might think of as the functionality of our program

- SpecsTest.java (in test folder)
- Test Code tests our source code

- Import statements (JUnit is a test runner)

- Test Annotations (what are annotations)
- Decorating the method definition
- Test tells JUnit to run it as a test
- DisplayName (and how it appears in IntelliJ when running)

- Call to assertTrue (assertions make tests pass or fail)

- Mess up the test and see it fail, read the error
- Note that the output of the tests is just plain text, but IntelliJ displays it intelligently

Let's implement some of our test cases from the tables! (go easy, just 1-2 for time)

Coding Demo: Unit Testing

- Source code – what we often think of as the functionality of our program, found in **`src`** directory
- Test code – code that tests the source code, found in **`tests`** directory
- JUnit – a java library for writing and running unit tests
- Annotations – like labels for our code
- Junit Assertions – methods used to verify things in JUnit tests

After demo: review the definitions

What about invalid inputs?

```
/**
* Returns `true` if the character in `str` at
* index `i` is uppercase, otherwise returns `false`.
* Requires that `0 <= i < str.length()`.
*/
static boolean uppercaseAt(String str, int i)
```

Input (str)	Input (i)	Expected output
"Hi!"	100	???
"rainbows"	-3	???
""	0	???

[Introduce think/poll/pair]

Think: What should the program do in these examples? And why?
Poll: Put your answer in pollev
Pair: Discuss with partner and revise answer if desired

THINK

What about invalid inputs?

PollEv.com/2110fa25
text 2110fa25 to 22333

```
/**
* Returns `true` if the character in `str` at
* index `i` is uppercase, otherwise returns `false`.
* Requires that `0 <= i < str.length()`.
*/
static boolean uppercaseAt(String str, int i)
```

Input (str)	Input (i)	Expected output
"Hi!"	100	???
"rainbows"	-3	???

POLL – When ready, please silently answer this poll question about the first input/output pair

Think: What should the program do in these examples? And why?
Poll: Put your answer in pollev
Pair: Discuss with partner and revise answer if desired

Poll Everywhere

PollEv.com/2110fa25 text `2110fa25` to `2233`

What is the correct behavior for

```
Specs.uppercaseAt("Hi!", 100);
```

return `true` **(A)**	return `false` **(B)**
error **(C)**	any of the above **(D)**

SKIP THIS SLIDE

Think: What should the program do in these examples? And why?
Poll: Put your answer in pollev
Pair: Discuss with partner and revise answer if desired

What about invalid inputs?

```
/**
* Returns `true` if the character in `str` at
* index `i` is uppercase, otherwise returns `false`.
* Requires that `0 <= i < str.length()`.
*/
static boolean uppercaseAt(String str, int i)
```

Input (str)	Input (i)	Expected output
"Hi!"	100	???
"rainbows"	-3	???

PAIR & REPOLL – Now please discuss with a neighbor. You may revise your poll answer as needed.

SHARE: WHY?

Think: What should the program do in these examples? And why?
Poll: Put your answer in pollev
Pair: Discuss with partner and revise answer if desired

Dealing with Pre-condition violations

- When a pre-condition is violated, a method's behavior is undefined
- Technically, any behavior would conform to the spec
- There are good behaviors and bad behaviors (style-wise)

Undefined = It can do anything
When behavior is undefined, any behavior complies with the spec

Dealing with Pre-condition violations

```java
/**
 * Returns `true` if the character in `str` at
 * index `i` is uppercase, otherwise returns `false`.
 * Requires that `0 <= i < str.length()`.
 */
static boolean uppercaseAt(String str, int i) {
  if (i < 0) {
    i = 0;
  } else if (i >= str.length()) {
    i = str.length() - 1;
  }
  return Character.isUpperCase(str.charAt(i));
}
```


Make it succeed – BAD

Technically this follows the spec (any behavior is allowed when preconditions are violated)
But, it violates the principle of least surprise

- Because it modifies the inputs provided by the caller

Dealing with Pre-condition violations

```java
/**
 * Returns `true` if the character in `str` at
 * index `i` is uppercase, otherwise returns `false`.
 * Requires that `0 <= i < str.length()`.
 */
static boolean uppercaseAt(String str, int i) {
  if (i < 0 || i >= str.length()) {
    throw new IllegalArgumentException(
       "`i` must be between 0 and `str.length()-1`.");
  }
  return Character.isUpperCase(str.charAt(i));
}
```


Again, technically allowed because when preconditions are not met, the program can do anything

- Given that the spec does not specify an exception, we're probably doing too much here
- If the spec specified this exception, then this would be the right thing to do

Dealing with Pre-condition violations

```java
/**
 * Returns `true` if the character in `str` at
 * index `i` is uppercase, otherwise returns `false`.
 * Requires that `0 <= i < str.length()`.
 */
static boolean uppercaseAt(String str, int i) {
  return Character.isUpperCase(str.charAt(i));
}
```


Just let it fail!!! -- good

Follows the KISS principle (Keep It Simple Silly!)
- Simpler code is easier to write, read, and maintain!
- str.charAt(i)

Dealing with Pre-condition violations

```java
/**
 * Returns `true` if the character in `str` at
 * index `i` is uppercase, otherwise returns `false`.
 * Requires that `0 <= i < str.length()`.
 */
static boolean uppercaseAt(String str, int i) {
  assert 0 <= i && i < str.length();
  return Character.isUpperCase(str.charAt(i));
}
```


Defensive programming
This can be useful for debugging
Caution: assert statements can add confusion because you need to enable them if you want them to run

What about invalid inputs?

```
/**
* Returns `true` if the character in `str` at
* index `i` is uppercase, otherwise returns `false`.
* Requires that `0 <= i < str.length()`.
*/
static boolean uppercaseAt(String str, int i)
```

Input (str)	Input (i)	Expected output
"Hi!"	100	???
"rainbows"	-3	???

To revisit our earlier question, what should tests do
- When preconditions are violated?

[short pause]
[next slide]

What about invalid inputs?

```
/**
* Returns `true` if the character in `str` at
* index `i` is uppercase, otherwise returns `false`.
* Requires that `0 <= i < str.length()`.
*/
static boolean uppercaseAt(String str, int i)
```

Input (str)	Input (i)	Expected output
"Hi!"	100	UNDEFINED
"rainbows"	-3	UNDEFINED

DO NOT TEST!!

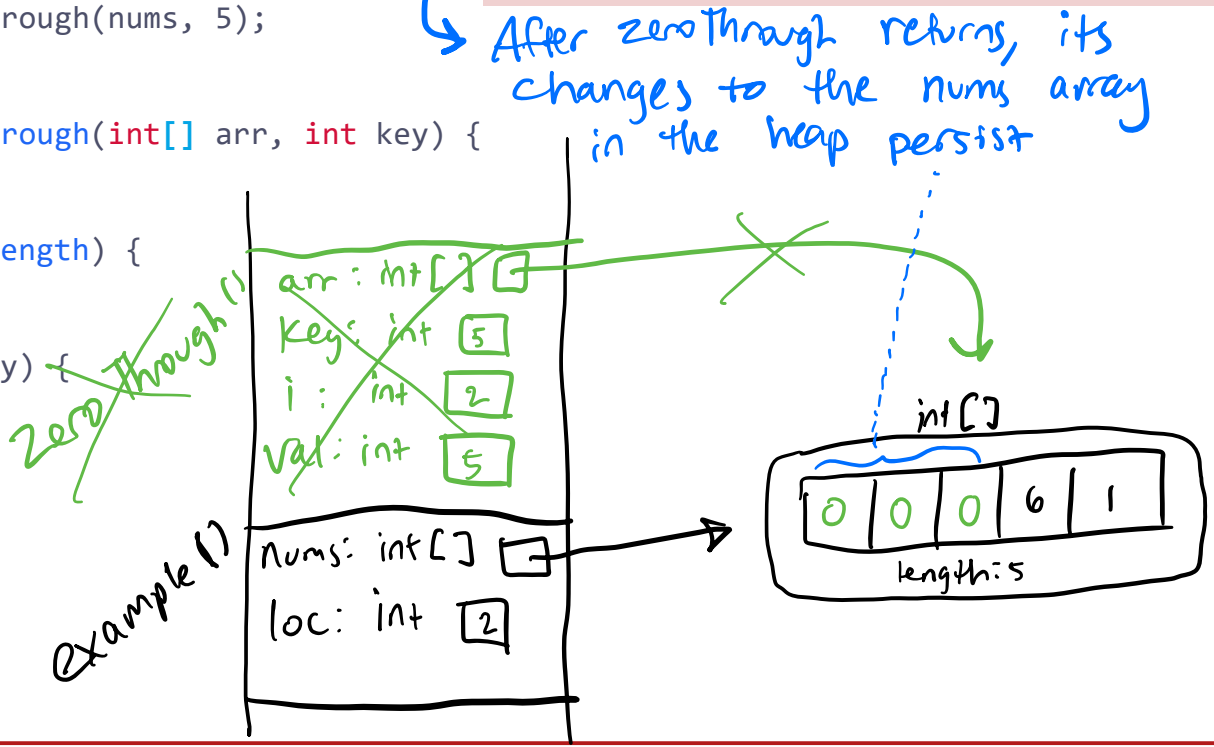
What Are Side Effects?

```
/**
 * Zeroes-out all entries before and including the first
 * instance of `key` in `arr` and returns the index where`key`
 * was found. If `key` is not present in `arr`, then all
 * indices of `arr` are zeroed out, and `arr.length` is returned.
 */
static int zeroThrough(int[] arr, int key) {
 int i = 0;
 int val;
 while(i < arr.length) {
   val = arr[i];
   arr[i] = 0;
   if (val == key) {
     return i;
   }
   i++;
 }
 return i;
}
```

What Are Side Effects?

```
static void example() {
 int[] nums = {4, 3, 5, 6, 1};
 int loc = zeroThrough(nums, 5);
}

static int zeroThrough(int[] arr, int key) {
  int i = 0;
  int val;
  while(i < arr.length) {
    val = arr[i];
    arr[i] = 0;
    if (val == key) {
      return i;
    }
    i++;
  }
  return i;
}
```



What Are Side Effects?

```
static void example() {
 int[] nums = {4, 3, 5, 6, 1};
 int loc = zeroThrough(nums, 5);
}

static int zeroThrough(int[] arr, int key) {
  int i = 0;
  int val;
  while(i < arr.length) {
    val = arr[i];
    arr[i] = 0;
    if (val == key) {
      return i;
    }
    i++;
  }
  return i;
}
```


Memory Diagramming

What Are Side Effects?

```
/**
 * Zeroes-out all entries before and including the first
 * instance of `key` in `arr` and returns the index where`key`
 * was found. If `key` is not present in `arr`, then all
 * indices of `arr` are zeroed out, and `arr.length` is returned.
 */
static int zeroThrough(int[] arr, int key) {
 int i = 0;
 int val;
 while(i < arr.length) {
   val = arr[i];
   arr[i] = 0;
   if (val == key) {
     return i;
   }
   i++;
 }
 return i;
}
```

Side effects include:
- State changes visible after the method call
- Modifying objects in the heap
- Printing to the console
- Displaying or changing graphical user interface elements

Examples of side effects include this – modifying objects in the heap
Another important example is user interface behaviors
- e.g. printing output to the console
- Displaying wordle tiles

Specifying & Testing Side Effects

```
/**
 * Zeroes-out all entries before and including the first
 * instance of `key` in `arr` and returns the index where`key`
 * was found. If `key` is not present in `arr`, then all
 * indices of `arr` are zeroed out, and `arr.length` is returned.
 */
static int zeroThrough(int[] arr, int key)
```

Input (arr)	Input (key)	Expected output	Expected value of arr
{9, 8, 7}	8	1	{0, 0, 7}

Highlight the description of the side effects [next slide]

Specifying & Testing Side Effects

```
/**
 * Zeroes-out all entries before and including the first
 * instance of `key` in `arr` and returns the index where`key`
 * was found. If `key` is not present in `arr`, then all
 * indices of `arr` are zeroed out, and `arr.length` is returned.
 */
static int zeroThrough(int[] arr, int key)
```

Write some input/output pairs for the test suite!

Input (arr)	Input (key)	Expected output	Expected value of arr
{9, 8, 7}	8	1	{0, 0, 7}

Specifying & Testing Side Effects

```
/**
 * Zeroes-out all entries before and including the first
 * instance of `key` in `arr` and returns the index where`key`
 * was found. If `key` is not present in `arr`, then all
 * indices of `arr` are zeroed out, and `arr.length` is returned.
 */
static int zeroThrough(int[] arr, int key)
```

Input (arr)	Input (key)	Expected output	Expected value of arr
{9, 8, 7}	8	1	{0, 0, 7}

Highlight the description of the side effects

Coding Demo: More Assertions

Show the tests for zeroThrough

ZeroThroughFirstReturned
- assertEquals for the first time
- Actual and expected value parameters

ZeroThroughZeroesCorrectly
- Now we're read to test some side effects
- Demonstrate looking up assertion for array equality
- Note that testing for not equals zero would only work if those elements were nonzero to start with

Refactor them live to use assertEqual(array, array)
- Comment out body

```
int[] expected ={0, 0, 0, 4, 5};
assertArrayEquals(expected, nums);
```

Testing Underspecified Code

```
/**
 * Returns an index `i` with `0 < i < a.length-1` that corresponds to a
 * local maximum of array `a`, meaning `a[i] > a[i-1]` and `a[i] >
a[i+1]`.
 * Returns 0 if `a` does not contain a local maximum.
 */
static int findLocalMax(int[] a)
```

Input (a)	Output
{0, 3, 0, 9, 0}	???

Think: How would you test this?
Pair: Discuss
Share

Testing Underspecified Code

```
/**
 * Returns an index `i` with `0 < i < a.length-1` that corresponds to a
 * local maximum of array `a`, meaning `a[i] > a[i-1]` and `a[i] >
a[i+1]`.
 * Returns 0 if `a` does not contain a local maximum.
 */
static int findLocalMax(int[] a)
```

Input (a)	Output
{0, 3, 0, 9, 0}	3
{0, 3, 0, 9, 0}	9
{0, 3, 0, 9, 0}	3 OR 9

Think: How would you test this?
Pair: Discuss
Share

Testing Underspecified Code

```
/**
 * Returns an index `i` with `0 < i < a.length-1` that corresponds to a
 * local maximum of array `a`, meaning `a[i] > a[i-1]` and `a[i] >
a[i+1]`.
 * Returns 0 if `a` does not contain a local maximum.
 */
static int findLocalMax(int[] a)
```

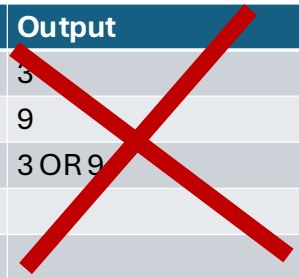
Input (a)	Output	
{0, 3, 0, 9, 0}	3	✖
{0, 3, 0, 9, 0}	9	✖
{0, 3, 0, 9, 0}	3 OR 9	MEH

Think: How would you test this?
Pair: Discuss
Share

Testing Underspecified Code

```
/**
 * Returns an index `i` with `0 < i < a.length-1` that corresponds to a
 * local maximum of array `a`, meaning `a[i] > a[i-1]` and `a[i] >
a[i+1]`.
 * Returns 0 if `a` does not contain a local maximum.
 */
static int findLocalMax(int[] a)
```

Input (a)	Output
{0, 3, 0, 9, 0}	3
{0, 3, 0, 9, 0}	9
{0, 3, 0, 9, 0}	3 OR 9



We won't test the output for a specific value

- But rather that it complies with the postconditions

Testing Underspecified Code

```
/**
 * Returns an index `i` with `0 < i < a.length-1` that corresponds to a
 * local maximum of array `a`, meaning `a[i] > a[i-1]` and `a[i] > a[i+1]`.
 * Returns 0 if `a` does not contain a local maximum.
 */
static int findLocalMax(int[] a)

-----------------------

@Test
void testMultipleMaxima() {
  int[] a = {1, 2, 4, 3, 6, 5, 1};
  int i = findLocalMax(a);
  assertTrue(a[i] > a[i-1]);
  assertTrue(a[i] > a[i+1]);
}
```


We won't test the output for a specific value

- But rather that it complies with the postconditions

i.e. Assertions don't have to check the return value for equality

- They can test properties of the return value
- And its relation to other data

Testing Underspecified Code

```java
/**
 * Returns an index `i` with `0 < i < a.length-1` that corresponds to a
 * local maximum of array `a`, meaning `a[i] > a[i-1]` and `a[i] > a[i+1]`.
 * Returns 0 if `a` does not contain a local maximum.
 */
static int findLocalMax(int[] a)

-----------------------

@Test
void testMultipleMaxima() {
  int[] a = {1, 2, 4, 3, 6, 5, 1};
  int i = findLocalMax(a);
  assertTrue(a[i] > a[i-1]);
  assertTrue(a[i] > a[i+1]);
}
```

Unit tests don't have to check the return value for equality
- They can test properties of the return value
- And its relation to other data

We won't test the output for a specific value
- But rather that it complies with the postconditions

Testing Underspecified Code

```
/**
 * Returns an index `i` with `0 < i < a.length-1` that corresponds to a
 * local maximum of array `a`, meaning `a[i] > a[i-1]` and `a[i] > a[i+1]`.
 * Returns 0 if `a` does not contain a local maximum.
 */
static int findLocalMax(int[] a)

-----------------------

@Test
void testMultipleMaxima() {
  int[] a = {1, 2, 4, 3, 6, 5, 1};
  int i = findLocalMax(a);
  assertTrue(a[i] > a[i-1]);
  assertTrue(a[i] > a[i+1]);
}
```

Unit tests don't have to check the return value for equality
- They can test properties of the return value
- And its relation to other data

Well specified code is more easily testable!

Another insight

Test Driven Development

1. Write specification
2. Write a test suite
3. Implement code for the specification
4. Run tests & revise code until it passes tests

Why Test Driven Development?

- …

Think/Pair/Share

(or think/share if not time)

Black Box vs Glass Box

Black box testing – tests based on specification and NOT code (test-driven development is an example!)

Glass box testing – implementation details inform test design

Think/Pair/Share

Metacognition

- Take 1 minute to write down a brief summary of what you have learned today

closing announcements to follow…

Announcements

- Assignment 1 (A1) due Wednesday 9/3 at 10 pm
  - see course website --> Assignments
  - virtual tour: Checking test output on gradescope
- Office hours will now be in Malott 301H
  - see course website --> About --> Office Hours