Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

What is the state of `nums` at the end of `main()`?

```
1  static void shuffle(int[] arr) {
2    int temp = arr[0];
3    arr[0] = arr[2];
4    arr[2] = temp;
5    arr = new int[]{arr[1], arr[2], arr[0]};
6  }
7
8  public static void main(String[] args) {
9    int[] nums = {1, 2, 3};
10   shuffle(nums);
11 }
```

`{1, 2, 3}`	(A)
`{2, 1, 3}`	(B)
`{2, 3, 1}`	(C)
`{3, 2, 1}`	(D)

Poll Everywhere

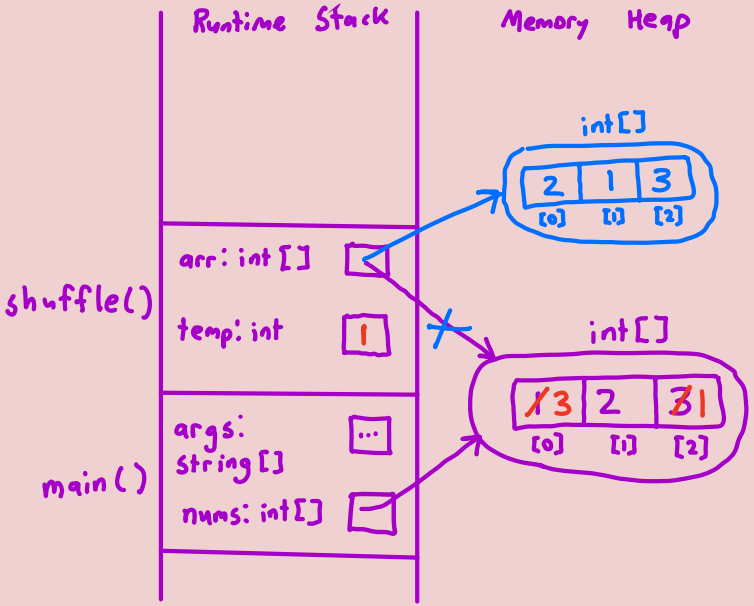
PollEv.com/javabear text **javabear** to 22333

What is the state of nums at the end of main()?

```
1  static void shuffle(int[] arr) {
2    int temp = arr[0];
3    arr[0] = arr[2];
4    arr[2] = temp;
5    arr = new int[]{arr[1], arr[2], arr[0]};
6  }
7
8  public static void main(String[] args) {
9    int[] nums = {1, 2, 3};
10   shuffle(nums);
11 }
```

swap entries a[0] and a[2]

reassign reference



Lecture 3: Specifications and Testing

CS 2110

September 1, 2026

Today's Learning Outcomes

12. Identify the pre-conditions, post-conditions, and side effects of a method and use these to write detailed Javadoc specifications.

14. Write comprehensive unit tests for a method given only its specifications and signature.

16. Explain the process of *test-driven development* and identify its potential benefits.

17. Differentiate between *black-box* and *glass-box* testing and identify scenarios where each is appropriate.

Client and Implementer Roles

Large, long-lasting software systems require coordination between developers.

Different people write different parts of the code, which must later be integrated.

A single developer interacts with a code module (a class, package, or **method**) from one (or both) of two possible perspectives:

1. Client ← Need way to coordinate → 2. Implementer

- caller of module
- needs to know how to use it, but not necessarily how it works (abstraction)
- actually run the code in their application

- author of module
- Knows all "internal details"
- may not ever "run" their code in a real system

Method Specifications

What information does a client need to (properly) use a method in their code?

- method name
- # of arguments
- type of each argument
- order of arguments

} method signature
how does the client call the method?

- interpretation of each argument
- conditions / requirements of inputs

} how do they call it correctly?

- return type
- interpretation of return value

} what will it do?

⋮

Pre-conditions and Post-conditions

Pre-conditions: Properties that must be true when a method is called

Today (for static methods): requirements on parameters

- allowed / disallowed values (valid ranges)
- required state of reference types (e.g., order of array entries)
- relationships between parameters

Post-conditions: Properties that will be true when a method returns **<u>if</u>** the pre-conditions were met.

- interpretation / assertions about <u>return value</u>
- <u>side effects</u>: modifications to the program state
  - state of shared objects
  - print output

Specifying Pre-conditions

/** ... */ for multiline Javadoc comments
built-in support to generate specs documentation from these

```
1  /**
2   * Returns `true` if the character in `str` at index `i` is uppercase; otherwise
3   * returns `false`. Requires that 0 ≤ i < str.length() .
4   */
5  static boolean uppercaseAt(String str, int i) { ... }
```

Predicate clauses

↑ ; must be valid index into str

Use "Requires..." clauses to specify pre-conditions.

* In CS2110, we'll add an implicit (unwritten) non-null pre-condition to all reference type parameters, unless otherwise specified

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

What should a method do if its pre-conditions are not met by its client?

"undefined behavior"

- Try to meet the post-conditions anyway (A)
- Return an obviously wrong value to "get back" at the client ? (B)
- Throw an exception to report the issue to the client (C)
- Crash the program (D)

Handling Pre-condition Violations

When the client violates a method's pre-conditions, it does not offer any guarantees about its behavior. **Any** behavior satisfies its specifications!

Easiest Approach: Ignore pre-condition violations

- write the code to work under the assumption of pre-conditions with no extra checks

"Best" Approach: Defensive programming

- Turn pre-condition violations into runtime errors with Java assert statements
- actively check pre-conditions, helpful for debugging

Coding Demo: Defensive Programming

* Remember to enable assertions!
  they're ignored by default
  instructions in IntelliJ guide

(Bad) Pre-condition Violation Handling

Do something special without including it in the specs

Throwing a "more appropriate" exception

```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));
}
```

"Correcting" the inputs

```java
static boolean uppercaseAt(String str, int i) {
  return Character.isUpperCase(Math.clamp(i, 0, str.length() - 1));
}
```

Unit Testing

Unit tests check whether a method/class *conforms to its specifications*.

not that method works correctly, does what its code says

* we write tests looking at specs, not source code

Many unit tests follow the same basic structure:

1. Set up inputs (arguments) to method "WHEN..."
2. Call method we're testing
3. Compare output to hard-coded expected output "THEN..."
   or check for expected side effects

Writing many separate unit tests for each method/class allows us to more easily diagnose issues as we code.

Coding Demo: JUnit

Good Unit Test Suites

Soundness: Unit tests should all pass for any implementation that meets the specs

** not just your particular implementation

Coverage: There are enough unit tests to "cover" the possible scenarios the module we're testing can encounter

Any small deviation from the specs should cause at least one test to fail.

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

Should a good unit test suite include tests that commit pre-condition violations?

Yes	(A)
No !! can't assert anything about undefined behavior	(B)
I'm not sure	(C)

Testing Methods with Side Effects

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

- can't just "assertEquals()" the return value
- instead, "inspect" state of objects using one or more assertions
- later for testing classes: use "accessor" methods

* Need to think of tests beyond "input, output" pairs to ensure good coverage

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

Consider the following method specifications:

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

What's wrong with the following JUnit test for this method?

```
1  @Test
2  void testMultipleMaxima() {
3    int[] a = {10, 12, 15, 13, 16, 14, 11};
4    assertEquals(2, findLocalMax(a));
5  }
```

Underspecification

Sometimes, there can be multiple different return values (or side effect behaviors) that all conform to a method's specifications.

In this case, we say that the method is **underspecified**.

Sound unit tests need to pass when run against all of these possible implementations

** Most common source of errors in CS2110 testing. Don't assume your approach is the only way to meet the specs.

Don't "overspecify" your assertions.

Testing with Underspecification

```java
/** Returns an index `i` with `0 < i < a.length-1` corresponding 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) { ... }
```

Account for all possibilities

```java
@Test
void testMultipleMaxima() {
  int[] a = {10, 12, 15, 13, 16, 14, 11};
  int i = findLocalMax(a);
  assertTrue(i==2 || i==4);
}
```

Check the post-condition property

```java
@Test
void testMultipleMaxima() {
  int[] a = {10, 12, 15, 13, 16, 14, 11};
  int i = findLocalMax(a);
  assertTrue(a[i] > a[i-1] &&
             a[i] > a[i+1]);
}
```

Test-Driven Development

= Use specs to write unit tests **first**, then use tests to guide development.

Many Benefits:

- focus code on specs
- think about edge cases upfront - often leads to more elegant code (fewer "patches")
- clear progress measure during development
- separation of responsibilities
- test soundness

But coming up with tests can be tricky... It takes some practice.

Black Box vs. Glass Box Testing

Black Box Testing: Tests are written without access to the source code. These tests are based on only specs, not any implementation details.

- naturally happens for test-driven development
- good for ensuring soundness
- good for dividing work after agreeing on specs

Glass Box Testing: Use implementation details to inform test suite.

- good for identifying corner cases of your implementation, achieving line coverage
- still need to ensure soundness