Poll Everywhere

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

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

```
static void shuffle(int[] arr) {
    int temp = arr[0];
    arr[0] = arr[2];
    arr[2] = temp;
    arr = new int[]{arr[1], arr[2], arr[0]};
}

public static void main(String[] args) {
    int[] nums = {1, 2, 3};
    shuffle(nums);
}
```

`{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

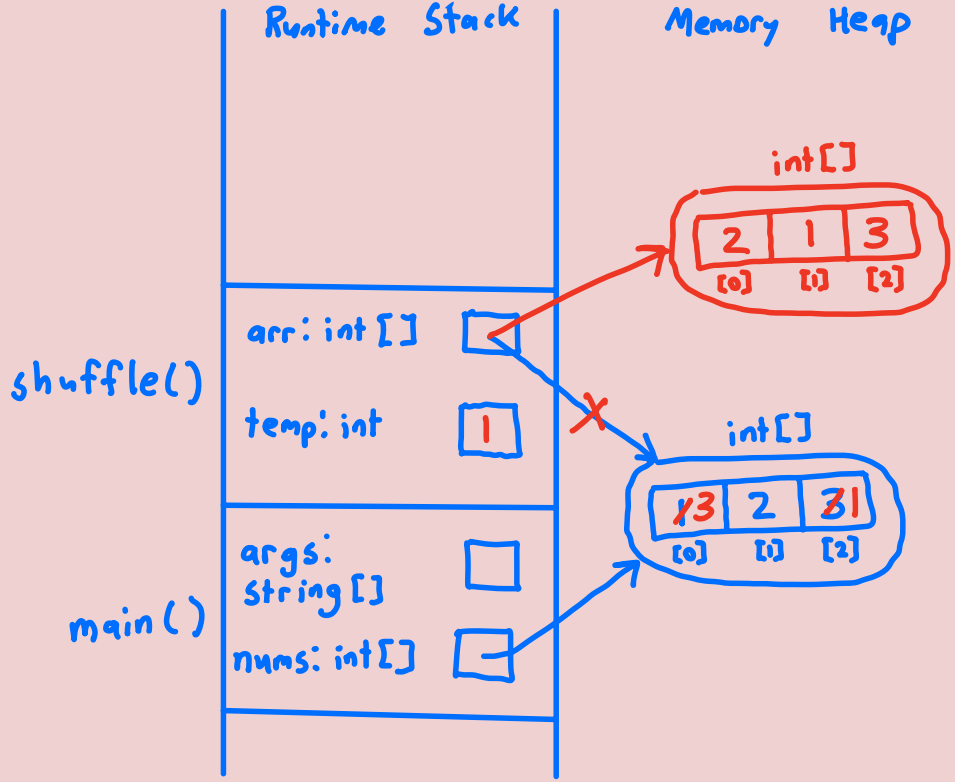
What is the state of nums at the end of `main()`?

```
static void shuffle(int[] arr) {
    int temp = arr[0];
    arr[0] = arr[2];
    arr[2] = temp;
    arr = new int[]{arr[1], arr[2], arr[0]};
}

public static void main(String[] args) {
    int[] nums = {1, 2, 3};
    shuffle(nums);
}
```

swap entries a[0] and a[2]

reassign refer



Lecture 3: Specifications and Testing

CS 2110

January 27, 2026

Today's Learning Outcomes

12. Write detailed Javadoc specifications for a method given its signature and an English description of its behavior.

13. Describe the *client* and *implementer* roles in software development.

14. Identify the pre-conditions, post-conditions, and side effects of a method given its specifications and signature.

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

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

19. Describe the differences between *black-box* and *glass-box* testing.

Client and Implementer Roles

Large, long-lasting software systems require coordination between developers; different people write different parts of the code, which must integrate together

A single developer interacts with a module (class, package, method) in one of two ways:

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

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

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

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 inputs (parameters)

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

Post-conditions: Properties that will be true when a method returns* if the pre-conditions were met

- interpretation / assertions about return value
- side effects (modifications to objects referenced by parameters)

Specifying Pre-conditions

Use a "Requires..." clause

```
/**
 * 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) { ... }
```

(so that i is a valid index in the string)

Note:

As an implicit (always present unless specifically noted) pre-condition, we assume reference type parameters are not null

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, there is no guarantee of its behavior. Any behavior meets the spec.

Easiest approach to pre-condition violations: Ignore them

- 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 using Java assert statements
- actively check pre-conditions, helpful for debugging

Coding Demo: Defensive Programming

(Bad) Pre-condition Violation Handling

Throw a "more appropriate" exception

```
/** 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));
}
```

If you're throwing an exception in certain cases, this should be reflected in the specs.

* We'll talk more about exceptions soon...

(Bad) Pre-condition Violation Handling

"Correct" the inputs

```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) {
    i = Math.max(i, 0);
    i = Math.min(i, str.length() - 1);
    return Character.isUpperCase(str.charAt(i));
}
```

can cause strange behaviors "downstream"

If you're doing extra work, might as well update specs to make these inputs valid.

Unit Testing

Way for implementers to check whether their code conforms to the specifications.

(we write tests looking at specs, not source code)

Separate unit tests for each method/class allow us to more easily diagnose issues. Keep each test small, include many tests.

Each test:
1. sets up input arguments
2. calls the method under test
3. compares output to (hard-coded) expected output
   or checks for expected side effects

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 sufficiently many unit tests to "cover" all possible inputs to the unit under test

Any small modification to a correct implementation that causes it to violate 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!! What is the "expected" behavior?!	(B)
I'm not sure	(C)

Testing Methods with 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) { ... }
```

"inspect" state of objects using one or more assertions

later: use "accessor" methods to learn about state

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:

```
/** 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) { ... }
```

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

```
@Test
void testMultipleMaxima() {
   int[] a = {1, 2, 4, 3, 6, 5, 1};
   assertEquals(2, findLocalMax(a));
}
```

Underspecification

Sometimes, there might be multiple different return values (or side effect behaviors) that all conform to the specs.

In this case, a sound unit test must pass for <u>all</u> of them

** 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

```
/** 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) { ... }
```

Account for all possibilities

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

Check the post-condition property

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

Black Box vs. Glass Box Testing

Black-Box Testing: Write tests without referring to the source code, only its specifications

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

Glass-Box Testing: Use implementation details to inform design of tests

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

Test-Driven Development

Using specs to write unit tests first, then use tests as a tool to guide development

- 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