Black-box and Glass-box Testing

Table of contents

1. Black box testing 1
2. Glass box testing 4
3. Ascending 5
4. Maximum — bonus 8
5. Index Of — bonus 9

Applied class for 2026-04-16-solid-principles-ii[1] (Week 7). Applies the black-box/white-box testing concepts from java-testing[2] (Week 5, 2026-03-26-testing[3]) to a worked example. Most of these tasks ask for *a* set of inputs achieving some coverage goal — many different answers are equally correct, so answers here are reveal-only worked examples rather than checkable exercises.

1. Black box testing

The following class tracks how many passengers are on a bus. Only the specification is given — no bodies — because black-box tests are written against the specification[4], not the implementation.

```
public class Bus {
    /**
     * Construct a new instance of the bus with a specified capacity.
     *
     * @param capacity The maximum passengers at any one time.
     */
    public Bus(int capacity) {...}
```

[1] courses/csse2002/2026-04-16-solid-principles-ii.pdf
[2] courses/csse2002/java-testing.pdf
[3] courses/csse2002/2026-03-26-testing.pdf
[4] courses/csse2002/java-specification.pdf

```
    /**
     * Calculate how many passengers are currently on the bus.
     *
     * @returns The number of passengers on the bus.
     */
    public int getCurrent() {...}

    /**
     * Calculate the average number of passengers on the bus between stops.
     *
     * @returns The average passengers on the bus.
     */
    public int getAverageCount() {...}

    /**
     * Exchange passengers at a bus stop.
     * Passengers get off before new passengers get on.
     *
     * @param on The number of passengers boarding the bus.
     * @param off The number of passengers disembarking the bus.
     */
    public void stop(int on, int off) {...}
}
```

The class above spells the tag `@returns`, not the standard `@return`. The course is inconsistent about specification-tag spellings (see csse2002-diagram-notation[5]).

Task 0 — write a smoke test

> **Working**
>
> There is a huge range of smoke tests you could come up with. Your smoke test should utilize the `Bus` class under normal/expected operation:
>
> ```
> Bus bus = new Bus(10);
> bus.stop(5, 0);
> bus.stop(3, 1);
> assertEquals(7, bus.getCurrent());
> assertEquals(6, bus.getAverageCount());
> ```
>
> The Javadoc says passengers get off **before** new passengers get on, so `stop(3, 1)` on a bus holding 5 goes $5 - 1 = 4$, then $4 + 3 = 7$; and the average between the two stops is $(5 + 7)/2 = 6$.

[5] courses/csse2002/csse2002-diagram-notation.pdf

See java-junit[6] for `assertEquals` and week8-lab-junit-testing[7] for writing these as real tests.

Task 1 — develop boundary scenarios (with justification)

Develop some boundary scenarios/inputs to use for testing. For each scenario you should justify why it is a good boundary case. You are **not** expected to know the expected output for each scenario.

i Working

There are many different situations to try; the following are just some examples.

Scenario	Justification
Construct a bus with capacity 0	A bus that can have no passengers is an uncommon scenario we may expect bugs from.
Construct a bus with a negative capacity	It is not clear how a bus with a negative capacity should behave; we should aim to clarify its functionality and test it.
Ask for the average number of passengers after no stops	An average often involves division by the number of times — we want to ensure that asking for the average after zero stops doesn't create an exception.
Negative values for `on` and `off`	Behaviour for negative values is unclear — will a negative number of `on` passengers result in passengers getting off?
More passengers get off at the stop than are currently on the bus	Results in undefined behaviour.
More passengers on the bus than capacity	Behaviour is not specified, so we wish to see how `Bus` behaves.

Task 2 — how can we make `Bus`'s behaviour fully specified?

Many of the boundary scenarios/inputs above are a result of unspecified or unclear behaviour of the bus class. How can we fix `Bus` to make all behaviour specified and clear?

[7] courses/csse2002/java-junit.pdf

[7] courses/csse2002/week8-lab-junit-testing.pdf

Working

If we specify that `getCurrent() <= capacity` is an **invariant** of the `Bus` class (see java-specification[8]), then we must protect the invariant throughout by either throwing exceptions or adding preconditions:

1. Add a `throws IllegalArgumentException` to the constructor of `Bus` when `capacity < 0`.
2. Add a precondition to `stop` that both `on` and `off` are greater than or equal to zero.
3. Add a `throws IllegalStateException` to the `stop` method if `getCurrent() + on - off` is less than zero or greater than capacity.

The two exception types do different jobs (see java-exceptions[9]): the argument itself is nonsense in case 1, whereas in case 3 the arguments are individually fine but would drive the object into a state the invariant forbids.

2. Glass box testing

Glass-box testing helps us cover edge cases of program control flow. This is possible as we are allowed to use the internal structure (control-flow) of a method to construct our test cases.

There are different levels of *code coverage*; in this course we focus on three:

1. **Statement coverage**
2. **Branch coverage**
3. **Path coverage**

Consider what inputs would achieve each level of code coverage for `countOccurrences`:

```java
public static int countOccurrences(int[] array, int target) {
    int count = 0;
    for (int num : array) {
        if (num == target) {
            count++;
        }
    }
    return count;
}
```

[9] courses/csse2002/java-specification.pdf

[9] courses/csse2002/java-exceptions.pdf

Working

- **Statement coverage** — every statement must run at least once, and the only one that isn't run unconditionally is `count++`. So the array must contain the target at least once: `[2]` with target `2`.
- **Branch coverage** — the `if` must additionally be false once, and the `for` must additionally terminate (it always does). `[2, 3]` with target `2` is enough on its own; so is `[2, 3, 2]`.
- **Path coverage** — approximated as 0, 1 and 2 iterations crossed with the `if` outcome each time, exactly the seven-row shape used for `ascending`. Because there is no early `return`, all seven are reachable.

3. Ascending

```
/**
 * Returns true if and only if numbers is an array of ascending
 * integers.
 *
 * @require numbers != null
 * @ensure \result is true iff for all indices j such that
 * 0 <= j < numbers.length - 1, numbers[j] <= numbers[j + 1]
 */
public boolean ascending(int[] numbers) {
    boolean result = true;
    for (int i = 0; i < numbers.length - 1; i++) {
        // invariant: for all j, 0 <= j < i, numbers[j] <= numbers[j + 1]
        if (numbers[i] > numbers[i + 1]) {
            result = false;
        }
    }
    return result;
}
```

The `// invariant:` line is the loop invariant — see java-loop-invariants[10]. It plays no part in the coverage answers below, which turn on the loop's shape: the body has no early exit, so every combination of iteration count and `if` outcome is reachable.

The three coverage levels are questions about the method's **control-flow graph**, so it helps to have it in front of you:

[10] courses/csse2002/java-loop-invariants.pdf

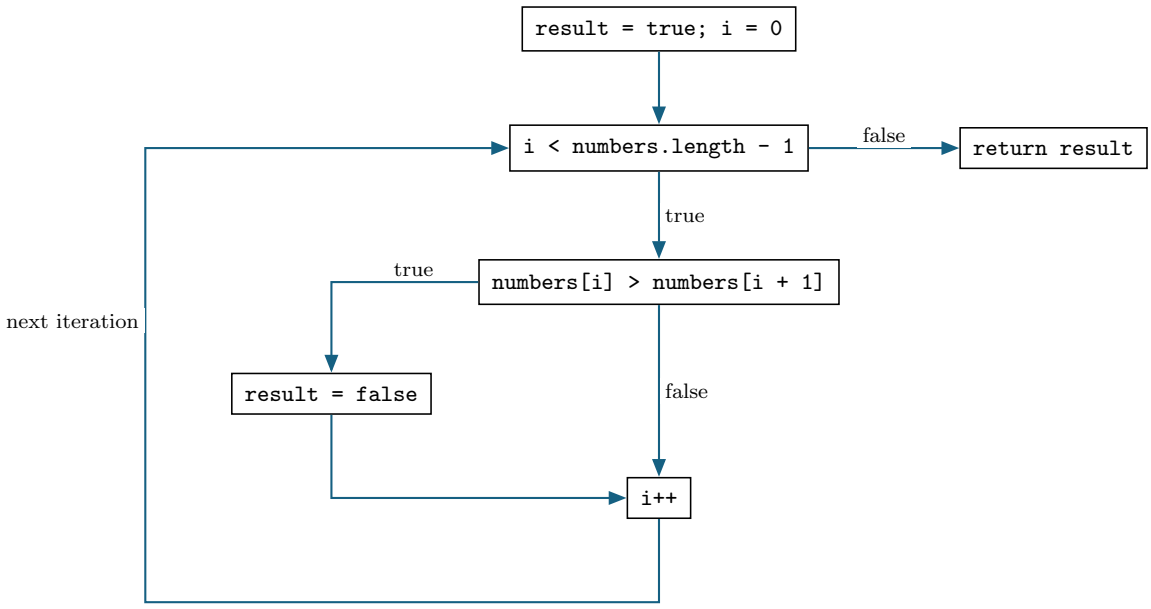

Figure 1: Control-flow graph of `ascending`. Every statement lies on some path; the two labelled edges out of `numbers[i] > numbers[i + 1]` are the only branch in the loop body, and the back edge is what makes exhaustive path coverage infeasible and forces the 0/1/2-iteration approximation. Statements and decisions are both drawn as plain rectangles; the teal arrows are the flow of control.

Task 3 — statement coverage

Provide a minimal set of inputs that achieve *statement* coverage.

> **Working**
>
> `[2, 1]`
> One iteration, with the `if` true, is enough to touch `result = false` — the only statement not on the unconditional path.

Task 4 — branch coverage

Provide a minimal set of inputs that achieve *branch* coverage.

> **Working**
>
> `[2, 1, 2]` **or** the pair `[2, 1]` `[1, 2]`
> Either way the `if` has to come out true once and false once.

Task 5 — path coverage

Provide a minimal set of inputs that achieve *path* coverage.

> **Working**
>
> | Input | Justification |
> |---|---|
> | `[]` or `[3]` | 0 times through loop |
> | `[3, 4]` | 1 time through loop, if false |
> | `[4, 3]` | 1 time through loop, if true |
> | `[3, 4, 5]` | 2 times through loop, if false, false |
> | `[3, 4, 2]` | 2 times through loop, if false, true |
> | `[4, 3, 5]` | 2 times through loop, if true, false |
> | `[5, 4, 3]` | 2 times through loop, if true, true |
>
> Writing the justification for each row helps ensure no cases are missed.
> Both `[]` and `[3]` give zero iterations, because the loop bound is `numbers.length - 1`.
> The 0/1/2-iteration approximation is the one from java-testing[11] — exhaustive path coverage of a loop is infeasible.

[11]

4. Maximum — bonus

Returns the maximum number within an array of integers, if one exists, else returns `Integer.MIN_VALUE`.

```
/**
 * Returns the maximum number within an array of integer,
 * if one exists, else returns Integer.MIN_VALUE.
 *
 * @require numbers != null
 * @ensure numbers.length > 0 ==> (
 *     (\forall int i; 0 <= i < numbers.length ==> \result >= numbers[i]) &&
 *     (\exists int i; 0 <= i < numbers.length && \result == numbers[i])
 * )
 * @ensure numbers.length == 0 ==> \result = Integer.MIN_VALUE
 */
public int maximum(int[] numbers) {
    int currentMaximum = Integer.MIN_VALUE;
    for (int number : numbers) {
        if (number > currentMaximum) {
            currentMaximum = number;
        }
    }
    return currentMaximum;
}
```

Task 6 — statement coverage

> **Working**
>
> `[1]`

Task 7 — branch coverage

> **Working**
>
> `[2, 1]`
> The first element beats `Integer.MIN_VALUE` (if true), the second does not beat `2` (if false).

courses/csse2002/java-testing.pdf

Task 8 — path coverage

i Working

Input	Justification
`[]`	0 times through loop
`[Integer.MIN_VALUE]`	1 time through loop, if false
`[1]`	1 time through loop, if true
`[Integer.MIN_VALUE, Integer.MIN_VALUE]`	2 times through loop, if false, false
`[Integer.MIN_VALUE, 1]`	2 times through loop, if false, true
`[2, 1]`	2 times through loop, if true, false
`[1, 2]`	2 times through loop, if true, true

`Integer.MIN_VALUE` is doing real work here: it is the only value for which `number > currentMaximum` can be false on the *first* iteration, because the accumulator starts there and the comparison is strict.

5. Index Of — bonus

```
/**
 * Returns the index of the given number within the array of numbers,
 * or -1 if number is not in numbers.
 *
 * @require numbers != null
 * @ensure numbers[\result] = number ||
 *     (\result == -1 &&
 *      \forall int j; 0 <= j < numbers.length ==> numbers[j] != number)
 */
public int indexOf(int[] numbers, int number) {
    for (int i = 0; i < numbers.length; i++) {
        // invariant: \forall int j; 0 <= j < i ==> numbers[j] != number
        if (numbers[i] == number) {
            return i;
        }
    }
    return -1;
}
```

This one differs from the other two in exactly one structural way: the `if` body **returns**, so the true branch leaves the method instead of continuing round the loop.

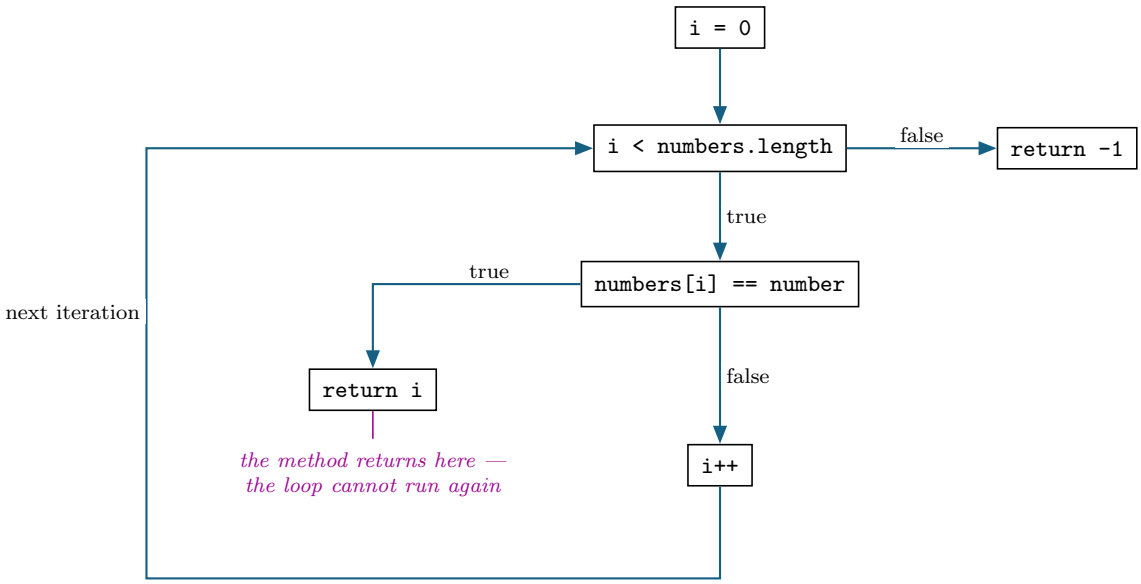

Figure 2: Control-flow graph of `indexOf`. Unlike `ascending`, the true branch of the `if` is an exit, not a step — so no path can take the `if` true and then iterate again, which is why two of the seven path-coverage rows in Task 11 read `Cannot be tested`. The exit is labelled in magenta.

Task 9 — statement coverage

Working

`[] 1` and `[1] 1`
Two calls are needed: `return i` and `return -1` are both statements, and no single input reaches both.

Task 10 — branch coverage

Working

`[] 1` and `[2, 1] 1`
`[2, 1]` with target `1` takes the `if` false then true; `[]` takes the loop condition false immediately.

Task 11 — path coverage

Working

Input	Justification
`[] 42`	0 times through loop
`[1] 2`	1 time through loop, if false
`[1] 1`	1 time through loop, if true
`[1, 2] 3`	2 times through loop, if false, false
`[1, 2] 2`	2 times through loop, if false, true
Cannot be tested.	2 times through loop, if true, false
Cannot be tested.	2 times through loop, if true, true

The last two rows are unreachable for the reason drawn in Figure 2: taking the `if` true on the first iteration returns immediately, so there is no second iteration to have an outcome at all.