JUnit Testing

Table of contents

Test Driven Development (TDD) 1
`DistinctCounter` 2
`PalindromeCounter` 5

Practical for 2026-04-23-java-io[1] (Week 8). Extends java-junit[2] (Week 5) with a full Test Driven Development (TDD) worked example.

Test Driven Development (TDD)

Cycle (see java-testing[3]):

1. Write a test for some piece of functionality.
2. Write just enough functionality for the test to pass.
3. Refactor implementation while preserving the passing test.
4. Repeat.

Stubs: a stub class has all public members, but methods return dummy values based on their return type — `void` methods are empty, primitives return a default (e.g. `0`), reference types normally return `null`. This lets you write an outline of a class that compiles but doesn't yet work — in pure TDD, not compiling counts as a test failure, so a stub is often the very first thing written to make a not-yet-existing class's test compile.

[1] courses/csse2002/2026-04-23-java-io.pdf
[2] courses/csse2002/java-junit.pdf
[3] courses/csse2002/java-testing.pdf

`DistinctCounter`

Tracks a collection of distinct strings, retrievable in lexicographical order:

- `DistinctCounter()`
- `void add(String element)`
- `int getDistinctCount()`
- `String[] getStrings()` — in lexicographical order

```
DistinctCounter distinct = new DistinctCounter();
distinct.add("Z");
distinct.add("Hello");
distinct.add("Z");
distinct.add("Hello ");
distinct.getDistinctCount(); // 3
distinct.getStrings();       // {"Hello", "Hello ", "Z"}
```

> **i** Working
>
> **Stub**:
>
> ```
> class DistinctCounter {
>     public DistinctCounter() {}
>     void add(String element) {}
>     int getDistinctCount() { return 0; }
>     String[] getStrings() { return null; }
> }
> ```
>
> **JUnit 4 test class skeleton**:
>
> ```
> import org.junit.Test;
> import static org.junit.Assert.*;
>
> class DistinctCounterTest {
>     @Test
>     public void testEmpty() {}
> }
> ```
>
> **Implementation** (one of several equally valid designs — a `HashSet` naturally rejects duplicates, so `getStrings()` just needs to sort on the way out):

```java
public class DistinctCounterHashSet implements DistinctCounter {
    private final Set<String> distinct = new HashSet<>();

    public void add(String word) { distinct.add(word); } // set won't add
     ↪ duplicates

    public int getDistinctCount() { return distinct.size(); }

    public String[] getStrings() {
        String[] elements = distinct.toArray(new String[]{});
        Arrays.sort(elements);
        return elements;
    }
}
```

(Other equally valid implementations: a `TreeSet`, which keeps elements sorted automatically without an explicit `Arrays.sort`; or an `ArrayList`-backed version that checks `contains()` before adding, sorting either on every `getStrings()` call or by inserting in sorted position on every `add()`.)

Representative tests (`@Before` constructs a fresh `counter` before each test, avoiding duplicated setup code in every method):

```java
public class DistinctCounterTest {
    private DistinctCounter counter;

    @Before
    public void setup() { counter = new DistinctCounter(); }

    @Test
    public void testEmptyCounterCount() {
        assertEquals("Empty counter does not have a count of zero", 0,
         ↪  counter.getDistinctCount());
    }

    @Test
    public void testTwoIdenticalCount() {
        counter.add("A");
        counter.add("A");
        assertEquals("Counter with two identical elements does not have count of
         ↪  one",
                1, counter.getDistinctCount());
    }

    @Test
    public void testTwoDistinctArray() {
        counter.add("A");
        counter.add("B");
        assertArrayEquals("Counter with two distinct elements does not have an
         ↪  array of two",
                new String[]{"A", "B"}, counter.getStrings());
    }
}
```

The full test suite (not reproduced in full here) mirrors this pattern across every case worth naming: empty / one element / two distinct / two identical / two identical + one other / N distinct (for both `getDistinctCount()` and `getStrings()`, plus that the returned array is sorted). TDD like this tends to produce a very thorough test suite, but one that mostly targets "happy path" execution — it's still worth adding boundary cases (see java-testing[4]) on top, e.g.:

```java
@Test
public void testNullStringCount() {
    counter.add(null);
    assertEquals(0, counter.getDistinctCount());
}
```

[4] courses/csse2002/java-testing.pdf

`PalindromeCounter`

Extends `DistinctCounter` with:

1. `int getPalindromeCount()` — number of distinct palindromes
2. `String[] getPalindromes()` — distinct palindromes
3. `String[] getNonPalindromes()` — distinct non-palindromes

Working

```
public class PalindromeCounter extends DistinctCounterTreeSet {
    private static boolean isPalindrome(String word) {
        if (word.length() < 2) {
            return true;
        }
        if (word.charAt(0) != word.charAt(word.length() - 1)) {
            return false;
        }
        return isPalindrome(word.substring(1, word.length() - 1));
    }

    public int getPalindromeCount() { return getPalindromes().length; }

    public String[] getPalindromes() {
        List<String> palindromes = new ArrayList<>();
        for (String word : getStrings()) {
            if (isPalindrome(word)) {
                palindromes.add(word);
            }
        }
        return palindromes.toArray(new String[]{});
    }

    public String[] getNonPalindromes() {
        List<String> distincts = new ArrayList<>(Arrays.asList(getStrings()));
        ↪  // wrap to allow removeAll
        distincts.removeAll(Arrays.asList(getPalindromes()));
        return distincts.toArray(new String[]{});
    }
}
```

Representative test (built up incrementally via TDD, one case at a time — empty, single palindrome, single non-palindrome, one of each, then several of each):

```java
public class PalindromeCounterTest {
    private PalindromeCounter counter;

    @Before
    public void setup() { counter = new PalindromeCounter(); }

    @Test
    public void testManyOfEachCounter() {
        counter.add("car");
        counter.add("racecar");
        counter.add("mamma mia");
        counter.add("AbbA");
        counter.add("palindrome");
        counter.add("rufus");
        assertEquals("Counter with two palindromes has incorrect count",
                2, counter.getPalindromeCount());
        assertArrayEquals("Counter with multiple palindromes does not have all
         ↪  in array",
                new String[]{"AbbA", "racecar"}, counter.getPalindromes());
        assertArrayEquals("Counter with multiple non-palindromes does not have
         ↪  all in array",
                new String[]{"car", "mamma mia", "palindrome", "rufus"},
                 ↪  counter.getNonPalindromes());
    }
}
```

Note `isPalindrome` treats strings shorter than 2 characters as palindromes by definition (the base case), and is case-sensitive (`"AbbA"` is a palindrome as written — comparing first/last characters directly, not after case-folding).