JUnit Testing

Table of contents

Test Driven Development (TDD) 1
`DistinctCounter` 2
`PalindromeCounter` 10

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.

Step 3's refactoring can be of the implementation under test **or of the test code itself**.

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' value such as `0` for integers, reference types normally return `null`. This lets you write an outline of a class that compiles but may not work.

> In pure test driven development, not compiling is considered a test failure, so a test is written for a class **before it exists**, and creating the class is considered a part of making a test pass.

[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, which may be added to the collection and retrieved 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"}
```

`"Hello"` and `"Hello "` are distinct — the trailing space matters — and `"Hello "` sorts before `"Z"`.

Task 0 — the stub

```
class DistinctCounter {
    public DistinctCounter() {}
    void add(String element) {}
    int getDistinctCount() {
        return 0;
    }
    String[] getStrings() {
        return null;
    }
}
```

Task 1 — the JUnit 4 test class

```
import org.junit.Test;

import static org.junit.Assert.*;

class DistinctCounterTest {
```

```
    @Test
    public void testEmpty() {}
}
```

Task 2 — the test cases

Write each of the following, and after each one write just enough code to make it pass, then refactor. The count cases:

1. An empty counter has a distinct count of zero.
2. A counter with one element has a count of one.
3. A counter with two distinct elements has only a count of two.
4. A counter with two identical elements has a count of one.
5. A counter with two identical elements and one other unique element has a count of two.
6. A counter with N distinct elements has a counter of N for $N > 4$.

The array cases mirror those six, plus three extra cases that ask specifically for a **sorted** array:

7. An empty counter has an empty array of strings.
8. A counter with one element has an array with that element.
9. A counter with two distinct elements has an array of those two elements.
10. A counter with two distinct elements has a **sorted** array of those two elements.
11. A counter with two identical elements has an array of one element.
12. A counter with two identical elements and one other unique element has an array of two elements.
13. A counter with two identical elements and one other unique element has a **sorted** array of two elements.
14. A counter with N distinct elements has an array of N elements for $N > 4$.
15. A counter with N distinct elements has a **sorted** array of N elements for $N > 4$.

> **i** Working
>
> There are many different ways to implement `DistinctCounter`, so it becomes an **interface** with four different implementations behind it — which is exactly the point of writing the tests first: all four pass the same fifteen cases of Task 2.

```
public interface DistinctCounter {
    void add(String value);
    int getDistinctCount();
    String[] getStrings();
}
```

1 — a hash set. A set won't add duplicates, so `add` is trivial and the sorting happens on the way out:

```
public class DistinctCounterHashSet implements DistinctCounter {
    private final Set<String> distinct;

    public DistinctCounterHashSet() {
        distinct = new HashSet<>();
    }

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

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

    public String[] getStrings() {
        String[] elements = new String[distinct.size()];
        int index = 0;
        for (String element : distinct) {
            elements[index] = element;
            index++;
        }
        Arrays.sort(elements);
        return elements;
    }
}
```

2 — a tree set, which automatically keeps elements in sorted order, so `getStrings()` needs no `Arrays.sort` at all:

```java
public class DistinctCounterTreeSet implements DistinctCounter {
    /** TreeSet maintains Strings in lexicographical order **/
    private final Set<String> distinct;

    public DistinctCounterTreeSet() {
        distinct = new TreeSet<>();
    }

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

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

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

3 — an array list, sorted on get. Without a set, `add` must check `contains` itself:

```java
public class DistinctCounterArrayListOnGet implements DistinctCounter {
    private final List<String> distinct;

    public DistinctCounterArrayListOnGet() {
        distinct = new ArrayList<>();
    }

    public void add(String word) {
        if (distinct.contains(word)) {
            return;
        }
        distinct.add(word);
    }

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

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

4 — `DistinctCounterArrayListOnAdd`, an array list sorted on add — an insertion sort, walking to the first element that sorts after `word` and inserting there; if no such element exists, `word` sorts after everything already in the list and is appended. Because the list is already ordered, this class's `getStrings()` is the bare `return distinct.toArray(new String[]{});` — no `Arrays.sort`, unlike implementation 3. Only `add` is given here:

```
public void add(String word) {
    if (distinct.contains(word)) {
        return;
    }
    boolean added = false;
    for (int i = 0; i < distinct.size(); i++) {
        if (distinct.get(i).compareTo(word) > 0) {
            distinct.add(i, word);
            added = true;
            break;
        }
    }
    if (!added) {
        distinct.add(word);
    }
}
```

The tests. `@Before` constructs an instance of `DistinctCounter` rather than duplicating the creation in every test method. The `new DistinctCounter()` in `setup()` below no longer compiles once `DistinctCounter` is the interface; in practice you construct whichever of the four implementations you are testing.

```
import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.*;

public class DistinctCounterTest {
    private DistinctCounter counter;

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

Some insist that each test method should have **only one assertion**. The tests here do that because it corresponds to the test cases listed for the exercise, but it is better to refactor tests to have two assertions than to duplicate the initialisation code.
That gives twelve test methods, not fifteen. The six count cases map one-to-one, but the nine array cases collapse to six methods, because each "…has a **sorted** array of…" case is already pinned down by the expected array in the case above it.

```java
@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 testEmptyArray() {
    assertArrayEquals("Empty counter does not have an empty array",
            new String[]{}, counter.getStrings());
}

@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());
}

@Test
public void testNDinstinctElementsArray() {
    counter.add("E"); counter.add("A"); counter.add("B"); counter.add("A");
    counter.add("A"); counter.add("B"); counter.add("D"); counter.add("A");
    counter.add("A"); counter.add("C"); counter.add("B");
    assertArrayEquals("Counter with 5 distinct elements does not have an array
     ↪  of five",
            new String[]{"A", "B", "C", "D", "E"}, counter.getStrings());
}
```

In the last one the elements are added deliberately out of order and with repeats, so the expected array simultaneously pins down *distinctness*, *count*, and *sorted order*.

Four implementations, one interface, one shared test suite — and, looking ahead, one of those four is the class that `PalindromeCounter` actually extends:

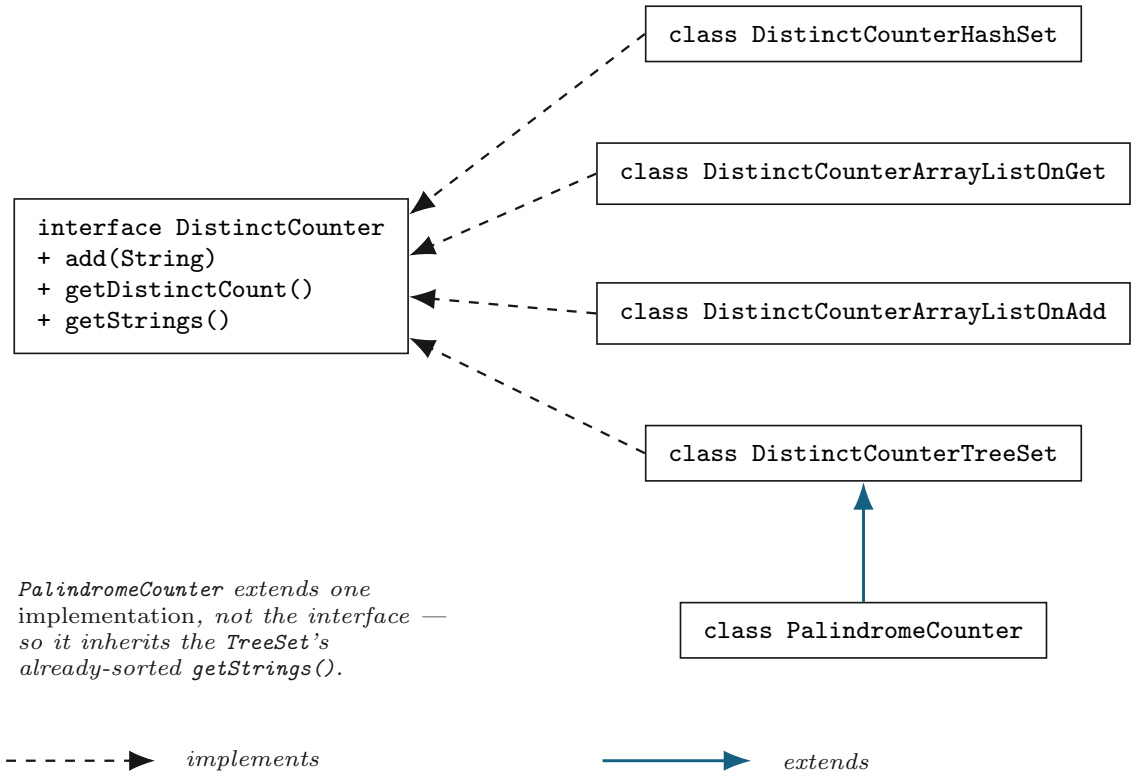

Figure 1: The four interchangeable implementations of one `DistinctCounter` interface, and — below them — the one of the four that `PalindromeCounter` actually extends. The dashed line running from implementer to interface is the *implements* arrow. In the course's dependency diagrams `Uses` is a solid line and `Implements` a dashed line, **both** with the same small **filled** arrowhead; this course does *not* use UML's hollow realisation triangle. The solid teal arrow into the superclass is the generalisation (*extends*) connector.

Task 3 — a boundary case

Test driven development will result in a highly comprehensive test suite, with every component rigorously tested. One possible downside is that it **tends to target "happy path" execution** — so it is worth adding boundary cases (see java-testing[4]) deliberately.

i Working

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

@Test
public void testNullStringArray() {
    counter.add(null);
    assertArrayEquals(new String[]{}, counter.getStrings());
}
```

The exercise asks *how does your implementation handle this case?* Only `DistinctCounterTreeSet` **throws**, and it throws on `add` itself, because a `TreeSet` has to compare the incoming element against what it holds. The other three accept `null`, so the count becomes 1 and both assertions simply **fail**. In particular `DistinctCounterArrayListOnAdd` never reaches `compareTo` here — on a fresh counter the list is empty, so its insertion loop has nothing to walk and the `null` is appended. (It, and the two implementations that call `Arrays.sort` on the way out, would hit a `NullPointerException` on a `null` only once the counter already holds another element to compare against.) Deciding which behaviour is correct is a specification question, not an implementation one — see java-specification[5].

`PalindromeCounter`

`PalindromeCounter` inherits from `DistinctCounter` and extends it with:

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

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

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

Tasks 4 and 5 repeat Tasks 0 and 1 for this class: write the stub, then create the JUnit 4 test class, following the `DistinctCounter` pattern. **Task 6** then builds up test cases and functionality together, TDD-style.

i Working

It extends `DistinctCounterTreeSet` — one concrete implementation, not the interface — which is why `getStrings()` is already in lexicographical order and `getPalindromes()` can simply filter it in place:

```
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() {
        // wrapping in new array list required to allow removeAll
        List<String> distincts = new ArrayList<>(Arrays.asList(getStrings()));
        List<String> palindromes = Arrays.asList(getPalindromes());
        distincts.removeAll(palindromes); // removes the palindromes
        return distincts.toArray(new String[]{});
    }
}
```

Two details in `isPalindrome` worth pinning down: a string shorter than two characters is a palindrome **by definition** (the recursive base case), and the comparison is **case-**

sensitive on the characters as written — `"AbbA"` is a palindrome because `'A' == 'A'` and `'b' == 'b'`, no case-folding involved.
`getNonPalindromes` has to wrap `Arrays.asList(...)` in a `new ArrayList<>(...)`: `Arrays.asList` returns a fixed-size view, which does not support `removeAll`.
The tests, built up one case at a time — empty, a single palindrome, a single non-palindrome, one of each, then several of each. These are only a subset of the tests you could implement using test driven development; three of the five methods are given here:

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

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

    @Test
    public void testEmptyCounter() {
        assertEquals("Empty counter does not have a count of zero",
                0, counter.getPalindromeCount());
        assertArrayEquals("Empty counter does not have no palindromes",
                new String[]{}, counter.getPalindromes());
        assertArrayEquals("Empty counter does not have no non palindromes",
                new String[]{}, counter.getNonPalindromes());
    }

    @Test
    public void testSingleNonPalindromeCounter() {
        counter.add("car");
        assertEquals("Counter with one non-palindrome does not have count of
        ↪  zero",
                0, counter.getPalindromeCount());
        assertArrayEquals("Counter with one non-palindrome does not have empty
        ↪  palindromes",
                new String[]{}, counter.getPalindromes());
        assertArrayEquals("Counter with one non-palindrome does not have a
        ↪  non-palindromes array of that element",
                new String[]{"car"}, counter.getNonPalindromes());
    }

    @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());
    }
}
```

> These test methods carry **three** assertions each — the refactoring recommended for the `DistinctCounter` tests but not done there, now that the three getters are all views of the same added words.