

---

# JAVA / UNIT TESTING

## EXERCISES

---

[home](#) / [exercises](#) / [unit-testing](#)

[#exercises](#)

[#java](#)

[#unit-testing](#)


[#oop](#)

---

## JAVA / UNIT TESTING

---

### 1. SETUP PROJECT

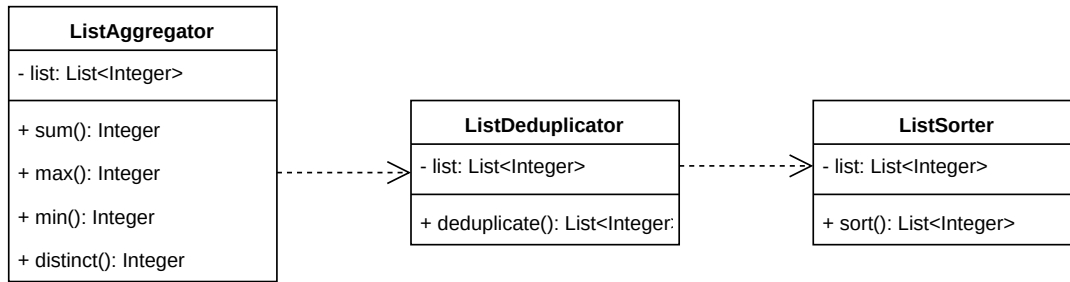
- **Download** and **unzip** the following  **project** into a suitable folder on your computer.
- Open **IntelliJ** and **import** the project:
  - **Step 1:** Import Project;
  - **Step 2:** Select Folder;
  - **Step 3:** Import from external model (Gradle);
  - **Step 4:** Select "Gradle" and click "Finish".
- Run all tests by selecting "**src/test/java**", **right-clicking** and selecting "**Run All Tests**".
- Verify that all tests **pass** (you might need to turn on showing tests that are passing by selecting the checkmark icon).

### 2. ANALYZE PROJECT

The project contains three classes:

- **ListAggregator:** Contains several methods that calculate values from lists of integers (sum, min, max and distinct). The **distinct()** method returns the number of distinct numbers in the list.
- **ListDeduplicator:** Is capable of removing duplicates from a list of integers.

- **ListSorter**: Is capable of sorting a list of integers.



As you can see, the **distinct()** method in the **ListAggregator** class, depends on the **ListDeduplicator** class in order to calculate the number of unique elements in a list.

Also, the **ListDeduplicator** class depends on the **ListSorter** class as it is much easier to remove duplicates in an already sorted list.

### 3. SIMPLIFY TEST SETUP

Take a moment to notice that our test methods are **organized** along **three** different **phases** (the 3 As):

- **Arrange** - Where the test is setup and the data is arranged.
- **Act** - Where the the actual method under test is invoked.
- **Assert** - Where a single logical assert is used to test the outcome.

Notice that the setup for the **ListAggregator** tests is always the same:

```

List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(4);
list.add(2);
list.add(5);
  
```

Do **one of two** things:

- Create a **helper** method, that gets **called** from **each one** of the tests, setting up the list.

- Create a **helper** method, having a **@Before annotation**, setting up the list as an **attribute**. Methods with a **@Before** annotation are called before **each test**.

Do the same for the **other test classes** making sure that all tests **still pass**.

## 4. CORNER CASES

You received a **bug report**:

### Bug report #7263

Created a list with values "-1, -4 and -5".

Tried to calculate the maximum of these values but got 0 instead of -1.

- Create a **test** that **confirms** the **bug**.
- Observe that the test **fails**.
- **Fix** the **code** so the test **passes**.

## 5. DISTINCT

You received a **bug report**:

### Bug report #8726

Created a list with values "1, 2, 4 and 2".

Tried to calculate the number of distinct values in the list but got 4 instead of 3.

- Start by creating a **test** that **confirms** the **bug**.
- Observe that the test **fails**.

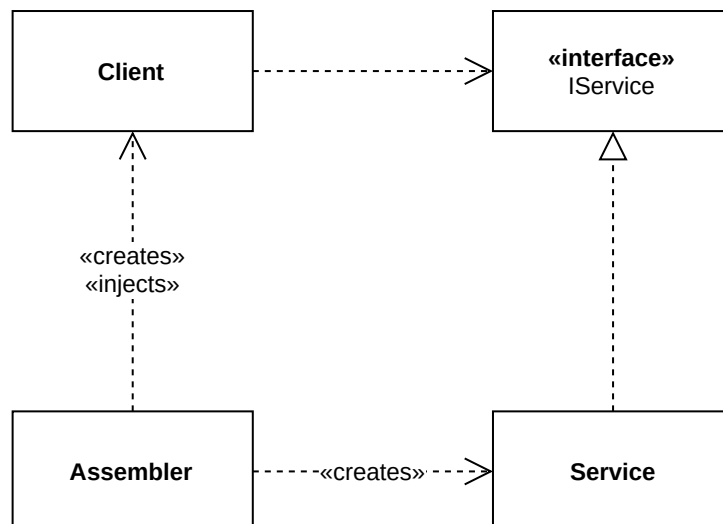
- Then, **look** into the **ListAggregator.distinct()** method code. **Spoiler alert:** you won't find anything wrong...

The problem is that when we are testing the **distinct()** method, we are also testing the **ListDeduplicator.deduplicate()** code. **Before** fixing the bug, **lets fix the test**.

To test the **distinct()** and the **deduplicate()** methods independently from each other, we must go from a design that looks like this:



Where our client (the **ListAggregator**) depends directly on its service (the **ListDeduplicator**). To something like this:



Where the **client depends** on an **interface** (lets call it **IService**) instead, and some **Assembler** class (the **ListAggregatorTest**) is responsible for creating the **concrete service** (the **ListDeduplicator**) and **injecting** it into the client (the **ListAggregator**).

Like this:

```
int distinct = aggregator.distinct(new ListDeduplicator(list));
```

This is what is called **Dependency Injection** and it allows our test to inject into the **ListAggregator** any list deduplicator **service**. Even one that always responds with the same canned answer (a **Stub**).

To remove the dependency between the **ListAggregatorTest** and the **ListDeduplicator** class using a stub, we first need to:

- Create a **IListDeduplicator** interface containing only the definition of the **deduplicate()** method.
- Modify the **ListAggregator.distinct()** method so that it can receive a class that implements this **IListDeduplicator** interface.
- Make **ListDeduplicator** implement this interface.
- Change the tests so that a **ListDeduplicator** is injected into the **distinct** method.

And then create the stub:

- Create a stub that always returns the correct answer for the data we are testing as a **inner-class** inside our **ListAggregatorTest.distinct()** method.
- Modify both **distinct()** tests so that they inject this **stub** class. This should make **both tests pass**.

This did **not fix** any bug, we simply **corrected** the failing **test** as it should not be the one failing. To **fix** our code we still have to:

- Create tests for **sorting** and **deduplicating** using these same values: "1, 2, 4 and 2".
- Make sure to use **Dependency Injection** in the **deduplicator()** method as it also depends on the **sort()** method.
- **Fix** the code that **needs fixing**. Only **one** of the tests should be failing now and that should point you in the **correct direction**.

## 6. MOCKITO

Redo the previous exercise but this time use [Mockito](#)  to create the stubs.

To use **Mockito**, you must first add this to the dependencies on your **build.gradle** file:

```
testCompile group: 'org.mockito', name: 'mockito-core', version: '2.25.0'
```

Creating a deduplicator using Mockito, should look like this:

```
IListDeduplicator deduplicator = Mockito.mock(IListDeduplicator.class);
```

Making the stub return the correct list can then be done like this:

```
Mockito.when(deduplicator.deduplicate()).thenReturn(deduplicated);
```

Where **deduplicated** is the list that we want the method to return.

## 7. COVERAGE

- Run all tests again, but this time **right-click** on "**src/test/java**", and select "**Run All Tests with Coverage**".

The report should appear on the right side of the screen.

Enter inside the **com** package, then inside the **aor** and **numbers** packages and verify if all classes, methods and lines are covered by your tests. If not add more tests until they are.

## 8. FILTERS

Create a new class **ListFilterer** that will be capable of **filtering** a **list of numbers**. This class should have a **constructor** that **receives a list** and single method called **filter** with the following **signature**:

```
public List<Integer> filter(IListFilter filter);
```

As you can see, this method **returns** a **list of numbers** that have been **filtered** by a certain **filter** (**Dependency Injection** again).

The **IListFilter** interface, should have only one method that returns true if a certain number should be accepted for that filter and false otherwise:

```
public boolean accept(Integer number);
```

Create **two classes** that follow this interface: **PositiveFilter** (that accepts only **positive** numbers) and **DivisibleByFilter** (that receives an **integer** upon construction and accepts only numbers **divisible** by that number).

- Create tests for all these classes (**ListFilterer**, **PositiveFilter** and **DivisibleByFilter**).
- Use **stubs** when necessary.
- Verify the test **coverage** again.

## 9. MUTATION TESTING

Test coverage allows us to **access** the **percentage** of lines **covered** by our **tests** but it doesn't verify the **quality** of those tests.

**Mutation** testing tries to **mitigate** this problem by creating **code mutations** (that should not pass the tests) and **verifying** if any of those mutations **survive** our test suite.

To use **PIT** (a **test mutation system** for **Java**) we must first add the following line to the **plugin section** of our **build.gradle** file:

```
id 'info.solidsoft.pitest' version '1.4.6'
```

By default, **PIT** runs all tests **under** the **package** with the **same name** as the **group** defined in your **build.gradle** file. So if all your classes and tests are under the **com.aor.numbers** package, no other configuration should be necessary.

**PIT** should have **automatically** created a **gradle task** called **pitest** that you can execute by doing (or using the IntelliJ gradle panel):

```
./gradlew pitest
```

This will run **PIT** and create a **report** under **"build/reports/pitest/<date>"**. You can open this report using your **browser** and check if any mutations **survived**.

Try **improving** your tests so **all mutations die**.

## 10. HERO TESTING

- With your new found knowledge, create **tests** for the **Hero** code you created last class.
- Try using **Dependency Injection** to **remove** the **dependency** between your **Element** classes and the **lanterna** library.
- Try using **Mocks**, with **Mockito**, to test if the correct **lanterna functions** are being **called** by your code.
- Verify the **coverage** of your tests.
- Try **mutation** testing and **improve** the results by writing **more** and **better** tests.

Copyright © André Restivo