How to TDD a List Implementation in Java

https://www.baeldung.com/java-test-driven-list

1. Overview

In this tutorial, we'll walk through a custom *List* implementation using the Test-Driven Development (TDD) process.

This is not an intro to TDD, so we're assuming you already have some basic idea of what it means and the sustained interest to get better at it.

Simply put, **TDD is a design tool, enabling us to drive our implementation with the help of tests**.

A quick disclaimer – we're not focusing on creating efficient implementation here – just using it as an excuse to display TDD practices.

2. Getting Started

First, let's define the skeleton for our class:

```
public class CustomList<E> implements List<E>
{
    private Object[] internal = {};
    // empty implementation methods
}
```

The *CustomList* class implements the *List* interface, hence it must contain implementations for all the methods declared in that interface.

To get started, we can just provide empty bodies for those methods. If a method has a return type, we can return an arbitrary value of that type, such as *null* for *Object* or *false* for *boolean*.

For the sake of brevity, we'll omit optional methods, together with some obligatory methods that aren't often used.

3. TDD Cycles

Developing our implementation with TDD means that we need to **create test cases first**, thereby defining requirements for our implementation. Only **then we'll create or fix the implementation code**to make those tests pass.

In a very simplified manner, the three main steps in each cycle are:

1. **Writing tests –** define requirements in the form of tests
2. **Implementing features –** make the tests pass without focusing too much on the elegance of the code
3. **Refactoring –** improve the code to make it easier to read and maintain while still passing the tests

We'll go through these TDD cycles for some methods of the *List* interface, starting with the simplest ones.

4. The *isEmpty* Method

The *isEmpty* method is probably the most straightforward method defined in the *List* interface. Here's our starting implementation:

```
@Override
public boolean isEmpty()
{
    return false;
}
```

This initial method definition is enough to compile. The body of this method will be “forced” to improve when more and more tests are added.

4.1. The First Cycle

Let’s write the first test case which makes sure that the *isEmpty* method returns *true* when the list doesn’t contain any element:

```
@Test
public void givenEmptyList_whenIsEmpty_thenTrueIsReturned()
{
    List<Object> list = new CustomList<Object>();

    assertTrue(list.isEmpty());
}
```

The given test fails since the *isEmpty* method always returns *false*. We can make it pass just by flipping the return value:

```
@Override
public boolean isEmpty()
{
    return true;
}
```

4.2. The Second Cycle

To confirm that the *isEmpty* method returns *false* when the list isn’t empty, we need to add at least one element:

```
@Test
public void givenNonEmptyList_whenIsEmpty_thenFalseIsReturned()
{
    List<Object> list = new CustomList<>();
    list.add(null);

    assertFalse(list.isEmpty());
}
```

An implementation of the *add* method is now required. Here’s the *add* method we start with:

```
@Override
public boolean add(E element)
{
    return false;
}
```

This method implementation doesn’t work as no changes to the internal data structure of the list are made. Let’s update it to store the added element:

```
@Override
public boolean add(E element)
{
    internal = new Object[] { element };
    return false;
}
```

Our test still fails since the *isEmpty* method hasn't been enhanced. Let's do that:

```
@Override
public boolean isEmpty()
{
    if (internal.length != 0)
   {
        return false;
    }
   Else
  {
        return true;
    }
}
```

The non-empty test passes at this point.

4.3. Refactoring

Both test cases we've seen so far pass, but the code of the *isEmpty* method could be more elegant.

Let's refactor it:

```
@Override
public boolean isEmpty()
{
    return internal.length == 0;
}
```

We can see that tests pass, so the implementation of the *isEmpty* method is complete now.

5. The *size* Method

This is our starting implementation of the *size* method enabling the *CustomList* class to compile:

```
@Override
public int size()
{
    return 0;
}
```

5.1. The First Cycle

Using the existing *add* method, we can create the first test for the *size* method, verifying that the size of a list with a single element is *1*:

```
@Test
public void givenListWithAnElement_whenSize_thenOneIsReturned()
{
    List<Object> list = new CustomList<>();
    list.add(null);

    assertEquals(1, list.size());
}
```

The test fails as the *size* method is returning *0*. Let's make it pass with a new implementation:

```
@Override
public int size()
{
    if (isEmpty())
   {
```

```
        return 0;
    }
    Else
    {
        return internal.length;
    }
}
```

5.2. Refactoring

We can refactor the *size* method to make it more elegant:

```
@Override
public int size()
{
   return internal.length;
}
```

The implementation of this method is now complete.

6. The *get* Method

Here's the starting implementation of *get*:

```
@Override
public E get(int index)
{
    return null;
}
```

6.1. The First Cycle

Let's take a look at the first test for this method, which verifies the value of the single element in the list:

```
@Test
public void givenListWithAnElement_whenGet_thenThatElementIsReturned()
{
    List<Object> list = new CustomList<>();
    list.add("baeldung");
    Object element = list.get(0);

    assertEquals("baeldung", element);
}
```

The test will pass with this implementation of the *get* method:

```
@Override
public E get(int index)
{
    return (E) internal[0];
}
```

6.2. Improvement

Usually, we'd add more tests before making additional improvements to the *get* method. Those tests would need other methods of the *List* interface to implement proper assertions.

However, these other methods aren't mature enough, yet, so we break the TDD cycle and create a complete implementation of the *get* method, which is, in fact, not very hard.

It's easy to imagine that *get* must extract an element from the *internal* array at the specified location using the *index* parameter:

```
@Override
public E get(int index)
{
    return (E) internal[index];
}
```

7. The *add* Method

This is the *add* method we created in section 4:

```
@Override
public boolean add(E element)
{
    internal = new Object[] { element };
    return false;
}
```

7.1. The First Cycle

The following is a simple test that verifies the return value of *add*:

```
@Test
public void givenEmptyList_whenElementIsAdded_thenGetReturnsThatElement()
{
    List<Object> list = new CustomList<>();
    boolean succeeded = list.add(null);

    assertTrue(succeeded);
}
```

We must modify the *add* method to return *true* for the test to pass:

```
@Override
public boolean add(E element)
{
    internal = new Object[] { element };
    return true;
}
```

Although the test passes, the *add* method doesn't cover all cases yet. If we add a second element to the list, the existing element will be lost.

7.2. The Second Cycle

Here's another test adding the requirement that the list can contain more than one element:

```
@Test
public void givenListWithAnElement_whenAnotherIsAdded_thenGetReturnsBoth() {
    List<Object> list = new CustomList<>();
    list.add("baeldung");
    list.add(".com");
    Object element1 = list.get(0);
    Object element2 = list.get(1);

    assertEquals("baeldung", element1);
    assertEquals(".com", element2);
}
```

The test will fail since the *add* method in its current form doesn't allow more than one element to be added.

Let's change the implementation code:

```
@Override
public boolean add(E element)
{
    Object[] temp = Arrays.copyOf(internal, internal.length + 1);
    temp[internal.length] = element;
    internal = temp;
    return true;
}
```

The implementation is elegant enough, hence we don't need to refactor it.

8. Conclusion

This tutorial went through a test-driven development process to create part of a custom *List*implementation. Using TDD, we can implement requirements step by step, while keeping the test coverage at a very high level. Also, the implementation is guaranteed to be testable, since it was created to make the tests pass.

Note that the custom class created in this article is just used for demonstration purposes and should not be adopted in a real-world project.

The complete source code for this tutorial, including the test and implementation methods left out for the sake of brevity, can be found over on GitHub.