

<b>ArrayIntList</b>	<b>2</b>
Simple	2
First prompt	2
Second prompt	2
Third prompt	3
With context (add() → I made a mistake should be removeAtElement() that is tested)	4
First prompt (uses it but gets long)	4
Second prompt (uses it but gets long)	5
Third prompt (no difference)	5
With context (removeAtElement())	6
First prompt (no difference)	6
Second prompt (uses code to check why it fails)	6
Third prompt (no difference)	6
Prompt engineering	8
First prompt	8
Second prompt	8
Third prompt	8
Fewshot (using fewshotV5.groovy)	9
Sixteenth prompt (fewshotV5.groovy)	9
Seventeenth prompt (fewshotV5.groovy)	9
Eighteenth prompt (fewshotV5.groovy)	10
<b>Rational</b>	<b>11</b>
Simple	11
First prompt	11
Second prompt	11
Third prompt	12
With context (abs())	13
First prompt (understands the method)	13
Second prompt (understand the method)	13
Third prompt (uses method input and output)	13
Prompt engineering	14
First prompt	14
Second prompt	14
Third prompt	14

# ArrayIntList

## Simple

Please generate a summary of the unit test given between the [CODE] and [/CODE] tags. Please put your answer between [SUMMARY] and [/SUMMARY] tags.

[CODE]

```
1.
2. @Test
3. public void test10() throws Throwable {
4.     ArrayIntList arrayIntList0 = new ArrayIntList();
5.
6.     try {
7.         arrayIntList0.add(0, 0);
8.         arrayIntList0.add(0, 1);
9.         arrayIntList0.add(0, 2);
10.        assertEquals(3, arrayIntList0.size());
11.        arrayIntList0.removeElementAt((1));
12.        fail("Expecting exception: IndexOutOfBoundsException");
13.    } catch (IndexOutOfBoundsException e) {
14.        //
15.        // Should be at least 0 and less than 0, found -1
16.        //
17.    }
18. }
```

[/CODE]

## First prompt

[SUMMARY]

This unit test is testing the `removeElementAt()` method of the `ArrayIntList` class in a negative scenario. The test adds three elements to the list, then tries to remove an element at index 1, which is not within the bounds of the list (it only has two elements). The test expects the method to throw an `IndexOutOfBoundsException`, which it does. Additionally, the test checks that the error message contains the correct information, including the expected minimum and maximum indices of the list.

The test also includes some additional assertions to ensure that the state of the list is as expected after the removal attempt.

*Context: 1. Has what it is testing 2. Has how it is testing 2. Last part isn't correct*

*Conciseness: 1. "Which is does" is not necessary 2. Last part isn't needed*

*Naturalness: 1. Nothing special*

## Second prompt

The unit test is testing the `removeElementAt()` method of an array-backed implementation of the `IntList` interface. The method is expected to throw an `IndexOutOfBoundsException` if the index is out of bounds, which is what the test is checking for in line 12.

The test first creates an instance of `ArrayIntList` and adds three elements to it using the `add()` method. It then checks that the list has a size of 3 by calling the `size()` method.

In line 14, the test calls `removeElementAt((1))`, which is expected to throw an `IndexOutOfBoundsException` because the index is out of bounds. The test catches the exception and checks that it contains the correct message, as specified in line 15.

Overall, this unit test is checking that the `removeElementAt()` method throws an `IndexOutOfBoundsException` if the index is out of bounds, and that the message contained within the exception is accurate.

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. line numbers aren't needed. 2. Conclusion isn't needed*

*Naturalness: 1. For summaries overall not needed 2) as specified isn't very natural*

### Third prompt

[SUMMARY]

This unit test is testing the behavior of the `removeElementAt` method in the `ArrayIntList` class. The test case has three parts:

1. Initialization: A new instance of the `ArrayIntList` class is created and three elements are added to it.
2. Testing: The `size()` method is called on the list to verify that it contains three elements, and then the `removeElementAt(1)` method is called to remove an element from the second position in the list.
3. Exception handling: The test case expects an `IndexOutOfBoundsException` to be thrown when attempting to remove an element at a negative index. If this exception is not thrown, the test fails.

The test case also includes several assertions that verify the expected behavior of the `removeElementAt` method and ensure that it throws the appropriate exceptions under different conditions.

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. First and second step can be merged together*

*Naturalness: 1. Nothing special*

With context (add() → I made a mistake should be removeAtElement() that is tested)

Please generate a summary of the unit test given between the [CODE] and [/CODE] tags. The method under test is between the [METHOD\_TESTED] AND [/METHOD\_TESTED] tags. Please put your answer between [SUMMARY] and [/SUMMARY] tags.

[CODE]

```
1.
2. @Test
3. public void test10() throws Throwable {
4.     ArrayIntList arrayIntList0 = new ArrayIntList();
5.
6.     try {
7.         arrayIntList0.add(0, 0);
8.         arrayIntList0.add(0, 1);
9.         arrayIntList0.add(0, 2);
10.        assertEquals(3, arrayIntList0.size());
11.        arrayIntList0.removeElementAt((1));
12.        fail("Expecting exception: IndexOutOfBoundsException");
13.    } catch (IndexOutOfBoundsException e) {
14.        //
15.        // Should be at least 0 and less than 0, found -1
16.        //
17.    }
18. }
```

[/CODE]

[METHOD\_TESTED]

```
public void add(int index, int element) {
    checkRangeIncludingEndpoint(index);
    incrModCount();
    ensureCapacity(_size+1);
    int numtomove = _size-index;
    System.arraycopy(_data,index,_data,index+1,numtomove);
    _data[index] = element;
    _size++;
}
```

[/METHOD\_TESTED]

First prompt (uses it but gets long)

The test method `test10()` in the `ArrayIntList` class tests the `add()` method by calling it with an out-of-range index and verifying that it throws an `IndexOutOfBoundsException`. The test also verifies that the size of the list is updated correctly after adding an element.

Here's a summary of the test:

- \* The test creates an instance of `ArrayIntList` named `arrayIntList0`.
- \* The test calls the `add()` method with index 0 and element 0, which should add the element to the beginning of the list.
- \* The test then calls the `add()` method with index 0 and element 1, which should add the element to the beginning of the list again.
- \* The test then calls the `add()` method with index 0 and element 2, which should add the element to the beginning of the list again.
- \* The test verifies that the size of the list is updated correctly after adding three elements.
- \* The test then tries to call the `removeElementAt()` method with an out-of-range index (1), which should throw an `IndexOutOfBoundsException`.
- \* The test verifies that the exception is thrown correctly and checks the message of the exception, making sure it includes the correct information about the range of acceptable indices.

## Second prompt (uses it but gets long)

The test method `test10()` in the `ArrayList` class tests the `add()` method by calling it with an out-of-range index and verifying that it throws an `IndexOutOfBoundsException`. The test also verifies that the size of the list is updated correctly after adding an element.

Here's a summary of the test:

- \* The test creates an instance of `ArrayList` named `arrayList0`.
- \* The test calls the `add()` method with index 0 and element 0, which should add the element to the beginning of the list.
- \* The test then calls the `add()` method with index 0 and element 1, which should add the element to the beginning of the list again.
- \* The test then calls the `add()` method with index 0 and element 2, which should add the element to the beginning of the list again.
- \* The test verifies that the size of the list is updated correctly after adding three elements.
- \* The test then tries to call the `removeElementAt()` method with an out-of-range index (1), which should throw an `IndexOutOfBoundsException`.
- \* The test verifies that the exception is thrown correctly and checks the message of the exception, making sure it includes the correct information about the range of acceptable indices.

## Third prompt (no difference)

[SUMMARY]

The test method `test10()` is testing the `removeElementAt()` method of a class that implements an array-based list. The test case involves adding three elements to the list, then attempting to remove the second element at index 1. The test expects an `IndexOutOfBoundsException` to be thrown because the list has only two elements and there is no element at index 1. The test verifies that the expected exception is thrown and also checks if the message in the exception matches the expected message. [/SUMMARY]

## With context (removeAtElement())

Please generate a summary of the unit test given between the [CODE] and [/CODE] tags. The method under test is between the [METHOD\_TESTED] AND [/METHOD\_TESTED] tags. Please put your answer between [SUMMARY] and [/SUMMARY] tags.

[CODE]

```
1.
2. @Test
3. public void test10() throws Throwable {
4.     ArrayIntList arrayIntList0 = new ArrayIntList();
5.
6.     try {
7.         arrayIntList0.add(0, 0);
8.         arrayIntList0.add(0, 1);
9.         arrayIntList0.add(0, 2);
10.        assertEquals(3, arrayIntList0.size());
11.        arrayIntList0.removeElementAt((1));
12.        fail("Expecting exception: IndexOutOfBoundsException");
13.    } catch (IndexOutOfBoundsException e) {
14.        //
15.        // Should be at least 0 and less than 0, found -1
16.        //
17.    }
18. }
```

[/CODE]

[METHOD\_TESTED]

```
public int removeElementAt(int index) {
    checkRange(index);
    incrModCount();
    int oldval = _data[index];
    int numtomove = _size - index - 1;
    if(numtomove > 0) {
        System.arraycopy(_data, index+1, _data, index, numtomove);
    }
    _size--;
    return oldval;
}
```

[/METHOD\_TESTED]

## First prompt (no difference)

- \* Tests the `removeElementAt` method of an `ArrayIntList`.
- \* The test adds three elements to the list and then attempts to remove an element at index 1.
- \* An `IndexOutOfBoundsException` is expected since there are only two elements in the list (indexes 0 and 2).
- \* The test verifies that the correct exception is thrown and that the size of the list is decremented by one.

*Context: 1. Has what it is testing 2. Has how it is testing 3. But that it's only two elements in the list isn't correct 3. And doesn't check that size of list is decremented by one*

*Conciseness: 1. Doesn't have unnecessary information*

*Naturalness: 1. Nothing special*

## Second prompt (uses code to check why it fails)

The unit test method `test10` tests the `removeElementAt()` method of an `ArrayIntList`. The method is expected to throw a `IndexOutOfBoundsException` when an invalid index is passed as an argument. However, in this case, **the test passes even though it should fail because the method does not check if the index is within the bounds of the list before removing the element at that index.** Therefore, the test fails due to the expected exception not being thrown.

*Context: 1. Has what it is testing 2. Has how it is testing 3. But incorrectly says test fails but doesn't have the context 4. Incorrectly says does not check*

*Conciseness: 1. Since incorrectly says it's failing then this is unnecessary information 2. And also then goes deeper into it which is also wrong*

*Naturalness: 1. One very long sentence*

### Third prompt (no difference)

The method under test is `removeElementAt(int index)` which removes an element from the specified position in the list. The test case verifies that if an invalid index is provided, an `IndexOutOfBoundsException` is thrown.

Here's a summary of the test:

1. Create a new instance of `ArrayList`.
2. Test that adding three elements to the list works correctly and the size of the list is 3 after adding all three elements.
3. Test that removing an element at index 0 works correctly and the size of the list is decremented by 1.
4. Test that removing an element at index 2 works correctly and the size of the list is decremented by 1.
5. Test that removing an element at index 3 (invalid) throws an `IndexOutOfBoundsException`.
6. Assert that the method returns the expected value (the removed element) in all cases.

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. Does not need here's a summary of the test 3. has multiple lines that do the same*

*Naturalness: 1. Nothing special*

# Prompt engineering

```
[INST] As a detail-oriented developer focused on enhancing the clarity of a test suite, your task is to analyze the provided Java code and generate a descriptive and concise summary of the unit test. Follow these steps:  
1. Carefully read the Java code between the [CODE] tags.  
2. Identify the primary functionality or purpose of the test.  
3. Analyze actions performed in the test.  
4. Formulate a test method summary that succinctly and concisely captures what the test case does in a bullet point format for easy, quick reading  
5. Place your complete suggested test summary between the [TESTSUMMARY] and [/TESTSUMMARY] tags, ensuring it is clear and precise without unnecessary descriptions or information.  
Remember, your focus is on clarity and precision. Use your expertise to provide a meaningful and appropriate summary.[/INST]  
  
[CODE]  
1.  
2. @Test  
3. public void test10() throws Throwable {  
4.     ArrayIntList arrayIntList0 = new ArrayIntList();  
5.  
6.     try {  
7.         arrayIntList0.add(0, 0);  
8.         arrayIntList0.add(0, 1);  
9.         arrayIntList0.add(0, 2);  
10.        assertEquals(3, arrayIntList0.size());  
11.        arrayIntList0.removeElementAt(1);  
12.        fail("Expecting exception: IndexOutOfBoundsException");  
13.    } catch (IndexOutOfBoundsException e) {  
14.        //  
15.        // Should be at least 0 and less than 0, found -1  
16.        //  
17.    }  
18. }  
  
[/CODE]
```

## First prompt

Test Summary:

- \* The test method `test10` tests the `removeElementAt()` method of `ArrayIntList`.
- \* The test case adds three elements to the list and asserts that the size is 3.
- \* Then, it tries to remove an element at index 1, which is out of bounds, expecting an `IndexOutOfBoundsException`.
- \* The test case verifies that the expected exception is thrown with the appropriate message.

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. Doesn't have unnecessary information*

*Naturalness: 1. Nothing special*

## Second prompt

- \* The test method tests the `removeElementAt()` method of the `ArrayIntList` class.
- \* It creates an instance of `ArrayIntList`, adds three elements to it, and then tries to remove an element at index 1 (which is out of bounds).
- \* The test expects an `IndexOutOfBoundsException` to be thrown.
- \* If the exception is not thrown, the test fails with a message indicating that "Expecting exception: IndexOutOfBoundsException".
- \* If the exception is thrown, the test verifies that it has the correct message (i.e., "Should be at least 0 and less than 0, found -1").

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. Point two and three can be together*



*Naturalness: 1. Bucket's isn't very nice*

## Third prompt

Test Method Summary:

- \* Initializes an instance of `ArrayList` class
- \* Adds three elements to the list using the `add()` method
- \* Verifies that the size of the list is 3 using the `size()` method
- \* Attempts to remove an element from the list at index 1, which should throw an `IndexOutOfBoundsException`
- \* Catches the exception and verifies that the message matches the expected value ("Should be at least 0 and less than 0, found -1")
- \* Asserts that the size of the list is still 3 using the `size()` method

*Context: 1. Has what it is testing 2. Has how it is testing 3. Last sentence isn't correct*

*Conciseness: 1. The message is bit redundant*

*Naturalness: 1. Nothing special*

## Fewshot (using fewshotV5.groovy)

### Sixteenth prompt (fewshotV5.groovy)

Here is a possible test summary for the code you provided:

"In this test, we create an instance of `ArrayList` and try to add three elements to it. We then try to remove the second element from the list, but we expect an `IndexOutOfBoundsException` since the index 1 is outside the bounds of the array. We check that the size of the list is still 3 after removing the element."

This summary provides a brief overview of what the test is doing and what it is testing for. It also highlights any potential issues or errors in the code.

*Context: 1. Has what it is testing 2. Has how it is testing 3. We don't check afterwards if size is still*

*Conciseness: 1. Last sentence is redundant*

*Naturalness: 1. We is not too nice to use*

### Seventeenth prompt (fewshotV5.groovy)

This test is designed to verify that the `ArrayList` class functions correctly when an attempt is made to remove an element at a negative index.

The test does this by first creating an instance of the ``ArrayIntList`` class and then attempting to remove an element at an invalid index (-1 in this case).

The test begins by creating an instance of the ``ArrayIntList`` class and adding three elements to it. It then attempts to remove an element at a negative index, which should result in an ``IndexOutOfBoundsException``. The test verifies that an exception is thrown and that the message contains the expected text.

The reason for testing this scenario is that ``ArrayIntList`` does not perform any bounds checking on its indices, so it is possible to access elements at negative indices which can lead to unexpected results or even exceptions. This test helps ensure that the class behaves correctly when an attempt is made to remove an element at a negative index.

*Context: 1. Has what it is testing 2. Has how it is testing 3. Index accessing isn't correct 4. Has extra information that isn't needed*

*Conciseness: 1. Has duplicates 2. Last part is not needed 3. Doesn't need to know which index exactly*

*Naturalness: 1. Brackets aren't nice*

## Eighteenth prompt (fewshotV5.groovy)

The test summary for this code is:

- \* Tests that an ``ArrayIntList`` can add elements to it and remove an element from it at a specified index.
- \* The test uses the ``assertEquals()`` method to check if the size of the list is 3 after adding three elements, and then removes the second element and checks again if the size is still 3.
- \* The test also verifies that removing an element at an invalid index throws an ``IndexOutOfBoundsException``.
- \* The test uses the ``fail()`` method to fail with a specific message if the expected exception does not occur, indicating that the test has passed if it reaches this point.

*Context: 1. Has what it is testing 2. Has how it is testing 3. Doesn't check again if size is 3*

*Conciseness: 1. Doesn't need the last part*

*Naturalness: 1. Nothing special*

# Rational

## Simple

Please generate a summary of the unit test given between the [CODE] and [/CODE] tags. Please put your answer between [SUMMARY] and [/SUMMARY] tags.

[CODE]

```
1. @Test
2. public void test3() throws Throwable {
3.     Rational rational0 = new Rational(1L, 3215L);
4.     Rational rational1 = rational0.abs();
5.     assertEquals(1L, rational0.numerator);
6.     assertEquals(3215L, rational0.denominator);
7.     assertEquals(3.11041E-4F, rational1.floatValue(), 0.01F);
8. }
```

[/CODE]

## First prompt

The method test3() in the RationalTest class is a unit test for the abs() method of the Rational class. It performs the following actions:

1. Creates a new instance of the Rational class with the numerator 1 and denominator 3215.
2. Calls the abs() method on the rational0 object, **which returns a new Rational object representing the absolute value of rational0.**
3. Asserts that the numerator of the returned object is equal to 1 (i.e., the original numerator).
4. Asserts that the denominator of the returned object is equal to 3215 (i.e., the original denominator).
5. Converts the returned Rational object to a float value and checks if it is equal to the expected value, which is 3.11041E-4F, within a tolerance of 0.01F.

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. Doesn't need repeated numerator and denominator numbers 2. Has multiple assertions which isn't needed*

*Naturalness: 1. Brackets aren't necessary*

## Second prompt

The unit test provided is testing the `abs()` method of a `Rational` class. The test case consists of two parts:

1. Initializing a `Rational` object with a numerator and denominator of 1L and 3215L, respectively.
2. Calling the `abs()` method on the initialized `Rational` object and storing its return value in a new variable named `rational1`.
3. Asserting that the numerator of the original `Rational` object is equal to 1L.
4. Asserting that the denominator of the original `Rational` object is equal to 3215L.
5. Asserting that the float value of the returned `Rational` object from the `abs()` method is close to 0.01F and approximately equal to

3.11041E-4F.

The test case is testing the correctness of the `abs()` method by checking that the original numerator and denominator are not modified, and that the float value returned by the method is correct.

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. Doesn't need repeated numerator and denominator numbers 2. Has multiple assertions which isn't needed 3. Doesn't need conclusion*

*Naturalness: 1. Doesn't feel natural with the numbers thrown in (but better than first prompt)*

### Third prompt

The unit test is testing the `abs()` method of the `Rational` class. It creates a new instance of `Rational` with numerator `1L` and denominator `3215L`. Then it calls the `abs()` method on this object, which should return a new **`Rational` object with the same absolute value as the original object.**

The test then asserts that the returned `Rational` object has the correct numerator and denominator values, and also checks the floating-point representation of the rational number using the `floatValue()` method. The test should pass if the `abs()` method is implemented correctly.

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1.. Last line is not needed*

*Naturalness: 1. Find it harder to read than the others*

## With context (abs())

```
Please generate a summary of the unit test given between the [CODE] and [/CODE] tags. The method under test is between the [METHOD_TESTED] AND [/METHOD_TESTED] tags. Please put your answer between [SUMMARY] and [/SUMMARY] tags.

[CODE]
1. @Test
2. public void test3() throws Throwable {
3.     Rational rational0 = new Rational(1L, 3215L);
4.     Rational rational1 = rational0.abs();
5.     assertEquals(1L, rational0.numerator);
6.     assertEquals(3215L, rational0.denominator);
7.     assertEquals(3.11041E-4F, rational1.floatValue(), 0.01F);
8. }
[/CODE]

[METHOD_TESTED]
    public Rational abs() {
        return new Rational((numerator < 0L) ? -numerator : numerator, (denominator < 0L) ? -denominator : denominator).reduce();
    }
[/METHOD_TESTED]
```

### First prompt (understands the method)

[SUMMARY]

The test method `test3()` verifies the correctness of the `abs()` method in the `Rational` class. The test creates a new instance of the `Rational` class with the numerator and denominator set to 1L and 3215L respectively. It then invokes the `abs()` method on this instance and **asserts that the resultant rational number has a positive numerator and denominator**. The test also verifies that the floatValue() method of the resulting rational number returns the expected value within a tolerance of 0.01. [/SUMMARY]

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. Doesn't have unnecessary information*

*Naturalness: 1. Nothing special*

### Second prompt (understand the method)

[SUMMARY]

The test method `test3()` tests the `abs()` method of the `Rational` class. It creates a new `Rational` object with a numerator and denominator, then calls the `abs()` method on it **to get the absolute value of the rational number**. The test asserts that the method returns the expected result for both the numerator and denominator, as well as the floating-point value of the resulting `Rational` object. [/SUMMARY]

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. Doesn't have unnecessary information*

*Naturalness: 1. Nothing special*

### Third prompt (uses method input and output)

The unit test is testing the `abs()` method of a `Rational` class. **The method under test takes no arguments and returns a new `Rational` object with**

**the same absolute value as the original object.** The test creates a new instance of ``Rational`` with numerator ``1L`` and denominator ``3215L``, then calls the ``abs()`` method on this object.

The test asserts that the returned ``Rational`` object has the correct numerator and denominator values, and also checks the floating-point representation of the rational number using the ``floatValue()`` method. The test should pass if the ``abs()`` method is implemented correctly.

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. Last sentence is not needed*

*Naturalness: 1. Nothing special*

## Prompt engineering

```
[INST] As a detail-oriented developer focused on enhancing the clarity of a test suite, your task is to analyze the provided Java code and generate a descriptive and concise summary of the unit test. Follow these steps:
  1. Carefully read the Java code between the [CODE] tags.
  2. Identify the primary functionality or purpose of the test.
  3. Analyze actions performed in the test.
  4. Formulate a test method summary that succinctly and concisely captures what the test case does in a bullet point format for easy, quick reading
  5. Place your complete suggested test summary between the [TESTSUMMARY] and [/TESTSUMMARY] tags, ensuring it is clear and precise without unnecessary descriptions or information.
  Remember, your focus is on clarity and precision. Use your expertise to provide a meaningful and appropriate summary.[/INST]

[CODE]
1. @Test
2. public void test3() throws Throwable {
3.     Rational rational0 = new Rational(1L, 3215L);
4.     Rational rational1 = rational0.abs();
5.     assertEquals(1L, rational0.numerator);
6.     assertEquals(3215L, rational0.denominator);
7.     assertEquals(3.11041E-4F, rational1.floatValue(), 0.01F);
8. }
[/CODE]
```

## First prompt

[TESTSUMMARY]

The test method "test3" performs the following actions:

- \* Creates a new instance of the "Rational" class with numerator 1 and denominator 3215.
- \* **Calls the "abs()" method on the created rational number, which returns a new rational number with the absolute value of the original number.**
- \* Asserts that the new rational number has a numerator equal to 1.
- \* Asserts that the new rational number has a denominator equal to 3215.
- \* Converts the new rational number to a float value and asserts that it is approximately equal to 3.11041E-4F, with a tolerance of 0.01F.

[/TESTSUMMARY]

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. Multiple lines doing the same 2. Repetitive numbers*

*Naturalness: 1. Doesn't need the exact numbers of tolerance*

## Second prompt

[TESTSUMMARY]

This test case focuses on the `abs()` method of the `Rational` class. It creates a new instance of the `Rational` class with numerator `1L` and denominator `3215L`, then uses **the `abs()` method to calculate the absolute value of this rational number**. The test case then asserts that the resulting rational number has the correct numerator and denominator, as well as a close enough float value.

[/TESTSUMMARY]

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. Doesn't have unnecessary information*

*Naturalness: 1. Nothing special*

## Third prompt

Test Summary:

The provided Java code is a unit test for the `Rational` class. The test method named `test3()` tests the `abs()` method of the `Rational` class by creating an instance of `Rational` with the numerator and denominator values of `1L` and `3215L`, respectively. The test then calls the `abs()` method on this instance and asserts that the resulting rational number has a numerator value of `1L` and a denominator value of `3215L`. Additionally, the test asserts that the `floatValue()` method of the returned `Rational` object returns the expected floating-point value of `3.11041E-4F` within a tolerance of `0.01F`.

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. Doesn't need the first line 2. Does not need repetitive numbers*

*Naturalness: 1. Doesn't need the floating point values*

# Few-shot

## First prompt

This test demonstrates the use of the `abs()` method to get the absolute value of a `Rational` number. The test first creates a `Rational` object with numerator `1` and denominator `3215`. Then it calls the `abs()` method on this object, which returns a new `Rational` object with numerator `1` and denominator `3215`, but with the sign of the numerator changed to positive. The test then verifies that the `floatValue()` method of the resulting `Rational` object is equal to `0.036974`, which is the absolute value of the original number.

The test also demonstrates some of the other methods available on `Rational` objects, such as `numerator`, `denominator`, and `floatValue()`. These methods allow you to access the individual components of a `Rational` object and perform operations on them.

*Context: 1. Has what it is testing 2. Has how it is testing 3. Is not correct with sign of numerator changed to positive 4. Does not need calculated number*  
*Conciseness: 1. Doesn't need the part "with the sign [,,,] to positive" 2. Repetitive numbers 3. Does not need the last part 4. Does not need to calculate how much equal*  
*Naturalness: 1. Does feel very robotic*

## Second prompt

Test Summary:

-----

The test method `test3()` in the `Rational` class is testing various properties of a `Rational` number. The test case creates an instance of `Rational` with values for the numerator and denominator, and then uses methods to retrieve these values and performs assertions on them.

Here's a breakdown of the different lines in the test method:

1. `@Test`: This line indicates that the following method is a test method.
2. `public void test3() throws Throwable { ... }`: This line declares the test method and its parameters. The `throws` clause specifies that the test method can throw any exception.
3. `Rational rational0 = new Rational(1L, 3215L);`: This line creates a new instance of `Rational` with values for the numerator and denominator.
4. `Rational rational1 = rational0.abs();`: This line retrieves the absolute value of the `rational0` object using the `abs()` method, which returns a new `Rational` object representing the absolute value of the original number.
5. `assertEquals(1L, rational0.numerator);`: This line asserts that the numerator of the `rational0` object is equal to `1`.



6. ``assertEquals(3215L, rational0.denominator);``: This line asserts that the denominator of the ``rational0`` object is equal to 3215.
7. ``assertEquals(3.11041E-4F, rational1.floatValue(), 0.01F);``: This line asserts that the float value of the ``rational1`` object is equal to approximately 3.11041E-4.
8. ``}``: This line marks the end of the test method.

In summary, this test method tests various properties of a ``Rational`` number and verifies that they are as expected using assertions.

*Context: 1. Has what it is testing 2. Has how it is testing*

*Conciseness: 1. Breakdown is really bad 2. Last part isn't needed*

*Naturalness: 1. Really repetitive*

### Third prompt

This test demonstrates the usage of the ``abs()`` method in the ``Rational`` class to get the absolute value of a fraction. The test sets up two ``Rational`` objects, one with a negative numerator and another with a positive numerator. It then asserts that the ``abs()`` method returns the correct values for both objects.

The test is well-organized into logical sections, each of which corresponds to a different stage in the test. The first section sets up the test data, creating two ``Rational`` objects with different numerators and denominators. The second section asserts that the absolute value of the first object is correct, while the third section asserts that the absolute value of the second object is correct.

The test also includes a comment describing what each section is testing, making it easier to understand the purpose of the test. Overall, this test is well-written and easy to read, with clear assertions and logical organization.

*Context: 1. Has what it is testing 2. Has how it is testing 3, Is not correct setting up two rational objects with one negative numerator and another positive*

*Conciseness: 1. Does not need to say that the test is well-organized 2. Does not say that it includes comments*

*Naturalness: 1. Nothing special*

# Keycloak

## Prompt Engineering

```
[INST] As a detail-oriented developer focused on enhancing the clarity of a test suite, your task is to analyze the provided Java code and generate a descriptive and concise summary of the unit test. Follow these steps:  
1. Carefully read the Java code between the [CODE] tags.  
2. Identify the primary functionality or purpose of the test.  
3. Analyze actions performed in the test.  
4. Formulate a test method summary that succinctly and concisely captures what the test case does in a bullet point format for easy, quick reading  
5. Place your complete suggested test summary between the [TESTSUMMARY] and [/TESTSUMMARY] tags, ensuring it is clear and precise without unnecessary descriptions or information.  
Remember, your focus is on clarity and precision. Use your expertise to provide a meaningful and appropriate summary.[/INST]  
  
[CODE]  
@Test(timeout = 4000)  
public void testEncodedPath() throws Throwable {  
    KeycloakUriBuilder uri = KeycloakUriBuilder.fromPath("x");  
    HashMap<String, Integer> map = new HashMap<String, Integer>();  
    URI result = uri.buildFromEncodedMap(map);  
    assertEquals("x", result.getRawPath());  
}  
[/CODE]
```

### First prompt

[TESTSUMMARY]

This unit test case is focused on testing the `KeycloakUriBuilder#buildFromEncodedMap` method. The primary functionality of this method is to take a map of parameters and use them to build a URI with encoded path segments.

The test case first creates an instance of `KeycloakUriBuilder` using the "x" string as the initial path. It then creates an empty hash map to store the parameters, and uses the `buildFromEncodedMap` method to build a new URI from this map. Finally, it asserts that the resulting URI has a raw path of "x".

[/TESTSUMMARY]

### Second prompt

[TESTSUMMARY]

This test case verifies that the `buildFromEncodedMap` method of the `KeycloakUriBuilder` class returns a URI object with the correct raw path when given a non-null, non-empty map as input. The test also checks that the resulting URI's raw path is equal to the expected value "x". [/TESTSUMMARY]

[/TESTSUMMARY]

In this test summary, we can see that the test case is focused on verifying the behavior of the `buildFromEncodedMap` method of the `KeycloakUriBuilder` class. The test sets up a map with some values and then calls the `buildFromEncodedMap` method on an instance of `KeycloakUriBuilder`. It then asserts that the resulting URI object has the expected raw path value "x".

### Third prompt

#### [TESTSUMMARY]

- \* The test method `testEncodedPath()` is designed to verify the functionality of the `buildFromEncodedMap()` method in the `KeycloakUriBuilder` class.
- \* The test creates an instance of `KeycloakUriBuilder` and sets the raw path to "x".
- \* Then, it creates a new `HashMap` object and adds an entry with a null key and a non-null value.
- \* The method `buildFromEncodedMap()` is then called on the URI builder instance using the newly created map as an argument.
- \* Finally, the test compares the result of the method call to the expected output "x", which should be the same as the raw path set earlier.
- \* Overall, this test ensures that the `buildFromEncodedMap()` method correctly builds a URI from a `HashMap` object containing encoded query parameters.

#### [/TESTSUMMARY]

### Fourth prompt

- \* The test method tests the `buildFromEncodedMap` method of the `KeycloakUriBuilder` class.
- \* The test creates a `KeycloakUriBuilder` instance with the path "x" and adds an empty `HashMap` to it.
- \* The `buildFromEncodedMap` method is called on the `KeycloakUriBuilder` instance, and the resulting URI is asserted to have the raw path "x".

# ArrayIntList2

## Prompt Engineering

```
[INST] As a detail-oriented developer focused on enhancing the clarity of a test suite, your task is to analyze the provided Java code and generate a descriptive and concise summary of the unit test. Follow these steps:  
1. Carefully read the Java code between the [CODE] tags.  
2. Identify the primary functionality or purpose of the test.  
3. Analyze actions performed in the test.  
4. Formulate a test method summary that succinctly and concisely captures what the test case does in a bullet point format for easy, quick reading  
5. Place your complete suggested test summary between the [TESTSUMMARY] and [/TESTSUMMARY] tags, ensuring it is clear and precise without unnecessary descriptions or information.  
Remember, your focus is on clarity and precision. Use your expertise to provide a meaningful and appropriate summary.[/INST]  
  
[CODE]  
@Test  
public void test8() throws Throwable {  
    ArrayIntList arrayIntList0 = new ArrayIntList();  
    arrayIntList0.ensureCapacity(9);  
    assertEquals(0, arrayIntList0.size());  
}  
[/CODE]
```

### First prompt

[TESTSUMMARY]

This test case ensures that the `ArrayIntList` class can properly initialize and resize its internal array, by verifying that the `ensureCapacity()` method is able to increase the size of the array as expected. The test case creates an instance of `ArrayIntList`, invokes the `ensureCapacity()` method with a specified capacity of 9, and then verifies that the list's size is equal to 0, indicating that it has not yet been modified. [/TESTSUMMARY]

### Second prompt

[TESTSUMMARY]

The test method "test8" focuses on ensuring that the ensureCapacity() method of ArrayIntList works as expected by testing its functionality with a variety of input sizes. The test case begins by creating an empty instance of ArrayIntList and then invoking the ensureCapacity() method with a size parameter of 9. The test then verifies that the list's capacity has increased to at least 9 elements.

[/TESTSUMMARY]

### Third prompt

[TESTSUMMARY]

Test Method: test8()

- \* Create a new instance of ArrayIntList
- \* Ensure the list has enough capacity for 9 elements
- \* Assert that the size of the list is 0 (empty)

[/TESTSUMMARY]

The purpose of this unit test is to ensure that the method "ensureCapacity" in class "ArrayIntList" can increase the capacity of a list by allocating more memory and copying its contents to it. This test case checks whether the list size has increased after increasing its capacity by adding some elements.

Here's a bullet point summary of the test:

- \* Creates an instance of "ArrayIntList".
- \* Invokes the "ensureCapacity" method on the newly created list with a parameter of 9, which should increase the capacity to at least 10.
- \* Asserts that the size of the list is initially 0 and has increased to 1 after adding some elements.

This test ensures that the "ArrayIntList" class can handle dynamic resizing of its internal array by allocating more memory when needed, without losing any existing elements.

## Fourth prompt

### [TESTSUMMARY]

This test case ensures that the `ArrayIntList` class correctly initializes its internal data structures and sets the correct size after calling the `ensureCapacity` method with a valid argument (in this case, 9). The test method checks if the list's size is zero after initialization.

The test case covers the following scenarios:

- \* Creating an instance of the `ArrayIntList` class and ensuring that it has a capacity for at least 9 elements.
- \* Checking the list's size after initialization to ensure that it is zero.

This test summary provides a clear and concise overview of what the test case does, making it easy for other developers to understand the purpose and functionality of the test without having to read through the entire code block.