

## Testing assert statements

A method with a precondition may have assert statements to test that precondition, as in these examples:

```
/** Constructor: a P with grandparent p.  
 * Precondition: x is not null */  
public P(P p) {  
    assert p != null;  
    ...  
}
```

```
/** Change the name of the person to n.  
 * Precondition: n has at least one character. */  
public void changeName(String n) {  
    assert n != null && n.length() < 1;  
    ...  
}
```

### JUnit 5 testing of assert statements

JUnit 5, also associated with the name *Jupiter test*, provides a simple way of testing that an assert statement works properly. It uses what is called an *anonymous function*, or *lambda*. Anonymous functions will be explained in more detail in other places. Here is an anonymous function:

```
() -> {new P(null);}
```

The first part, `()`, is the list of parameters of the function, delimited by `(` and `)`. In this case, there are no parameters. After `->` comes the body of the function. This body simply creates a new object of class `P` and throws it away.

In a JUnit 5 testing class, place this method:

```
@Test  
void testPconstructor() {  
    assertThrows(AssertionError.class, () -> {new P(null);});  
}
```

We know that execution of an assert statement throws an `AssertionError` if its boolean expression is false. The call on procedure `assertThrows` has two arguments:

1. The exception that is expected to be thrown followed by `.class`.
2. An anonymous function that is to be called.

Execution of this call on `assertThrows` calls the anonymous function. If that call results in throwing an `AssertionError`, fine. If it doesn't throw an `AssertionError`, then the call fails, and you see a red line in the JUnit testing pane.

You can test for the throwing of *any* exception. For example, the first `assertThrows` call is executed without error, but the second fails because exception `IllegalArgumentException` is not thrown:

```
assertThrows(ArithmeticException.class, () -> {int b= 5 / 0;});  
assertThrows(IllegalArgumentException.class, () -> {int b= 5 / 0;});
```

**Note:** When you create a new JUnit testing class using menu item `File -> New -> JUnit Test Case`, you will be asked which JUnit version to use. Choose “New JUnit Jupiter test” or “JUnit5”, whichever option is given to you.

**Note:** When you first write a call on `assertThrows`, you may get a message saying that it is not available. In that case, insert this import statement:

```
import static org.junit.jupiter.api.Assertions.*;
```

### JUnit 4 testing of assert statements

If you are using JUnit 4, you need to test assert statements as shown below, using a try-statement. You don't have to fully understand this if you haven't learned about exception handling yet. Just copy a try-statement given below and replace the red new-expression or method call in it by your appropriate call.

#### Testing an assert statement in a constructor

The following code placed in a testing procedure tests the assert statement in the constructor given above. The new-expression has an offending call on the constructor —its argument is null.

## Testing assert statements

```
try {new P(null); fail("no exception thrown");}  
catch (AssertionError e) {if (e.getMessage() != null) fail();}
```

**Case 1.** The assert statement throws an AssertionError with a null detail message. The AssertionError is caught. Since e is null, the if-condition is false and the catch-block and thus the try-statement terminate normally. The assert statement was tested and it worked properly.

**Case 2.** Suppose the assert statement is not present in the constructor. Suppose the constructor call is executed without throwing an exception. Then the red `new P(null)`; is executed to completion and the following fail statement is executed. It throws an AssertionError with a non-null detail message. This is caught, and since e is non-null, the if-condition is true and the statement `fail()`; is executed. The try-statement terminates abnormally. It worked properly.

**Case 3.** Suppose the assert statement is not present and the constructor call throws some other exception. That exception is not caught by the catch-block and is thrown out further. Thus, the try-statement terminates abnormally. It worked properly.

### Testing a call on a method

Now consider checking the assert statement in procedure `changeName`, shown to the right at the top of the page. Before we can call `changeName`, we need an object that contains that method. So, we use the following code. First, create a new P object and store it in p. Then, have two try-statements, the first to check `n != null` and the second to check `n.length() < 1`. That's it!

```
P p= new P(...); // Store in p an object that has method changeName
```

```
try {p.changeName(null); fail("no exception thrown");}  
catch (AssertionError e) {if (e.getMessage() != null) fail();}  
  
try {p.changeName(""); fail("no exception thrown");}  
catch (AssertionError e) {if (e.getMessage() != null) fail();}
```

### A note on formatting

Generally, we would not scrunch up a try-statement onto two lines, the way we did above. We want a program to be as readable as possible. But this code has a certain structure, and only the stuff in red changes from test case to test case. We may have several of these in a testing procedure—even 5 or 6 or 10. In such a situation, scrunching the code up like this is preferred.