

The
Pragmatic
Programmers

Pragmatic Unit Testing in Java with JUnit

Third Edition



Jeff Langr

Foreword by Dave Thomas

edited by Kelly Talbot

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit <https://www.pragprog.com>.

Copyright © The Pragmatic Programmers, LLC.

Organizing Related Tests into Nested Classes

As your classes grow by taking on more behaviors, you'll need more and more tests to describe the new behaviors. Use your test class size as a hint: if you declare several dozen tests in one test source file, chances are good that the class under test is too large. Consider splitting the production class up into two or more classes, which also means you'll want to split the test methods across at least two or more test classes.

You may still end up with a couple dozen test methods in one test class. A larger test class can not only be daunting from a navigational sense, but it can also make it harder to find all tests that relate to each other.

To help group related tests, you might consider starting each related test's name with the same thing. Here are three tests describing how withdrawals work in the Account class:

```
@Test void withdrawalReducesAccountBalance() { /* ... */ }
@Test void withdrawalThrowsWhenAmountExceedsBalance() { /* ... */ }
@Test void withdrawalNotifiesIRSWhenAmountExceedsThreshold() { /* ... */ }
```

A better solution, however, is to group related tests within a JUnit `@Nested` class:

```
@Nested
class Withdrawal {
    @Test void reducesAccountBalance() { /* ... */ }
    @Test void throwsWhenAmountExceedsBalance() { /* ... */ }
    @Test void notifiesIRSWhenAmountExceedsThreshold() { /* ... */ }
}
```

You can create a number of `@Nested` classes within your test class, similarly grouping all methods within it. The name of the nested class, which describes the common behavior, can be removed from each test name.

You can also use `@Nested` classes to group tests by context—the state established by the arrange part of a test. For example:

```
class AnAccount
    @Nested
    class WithZeroBalance {
        @Test void doesNotAccrueInterest() { /* ... */ }
        @Test void throwsOnWithdrawal() { /* ... */ }
    }
}
```

```

@Nested
class WithPositiveBalance {
    @BeforeEach void fundAccount() { account.deposit(1000); }
    @Test void accruesInterest() { /* ... */ }
    @Test void reducesBalanceOnWithdrawal() { /* ... */ }
}
}

```

Tests are split between those needing a zero-balance account (`WithZeroBalance`) and those needing a positive account balance (`WithPositiveBalance`).

Observing the JUnit Lifecycle

You've learned about using before and after hooks and how to group related tests into nested classes. Using a skeleton test class, let's take a look at how these JUnit elements are actually involved when you run your tests.

`AFundedAccount` contains six tests. Per its name, all tests can assume that an account exists and has a positive balance. An account object gets created at the field level and subsequently funded within a `@BeforeEach` method. Here's the entire `AFundedAccount` test class, minus all the intricate details of each test.

```

utj3-junit/01/src/test/java/scratch/AFundedAccount.java

```

```

import org.junit.jupiter.api.*;

class AFundedAccount {
    Account account = new Account("Jeff");
    AFundedAccount() {
        // ...
    }

    @BeforeEach
    void fundAccount() {
        account.deposit(1000);
    }

    @BeforeAll
    static void clearAccountRegistry() {
        // ...
    }

    @Nested
    class AccruingInterest {

        @BeforeEach
        void setInterestRate() {
            account.setInterestRate(0.027d);
        }

        @Test
        void occursWhenMinimumMet() {
            // ...
        }
    }
}

```

```

@Test
void doesNotOccurWhenMinimumNotMet() {
    // ...
}

@Test
void isReconciledWithMasterAccount() {
    // ...
}
}

@Nested
class Withdrawal {

    @Test
    void reducesAccountBalance() {
        // ...
    }

    @Test
    void throwsWhenAmountExceedsBalance() {
        // ...
    }

    @Test
    void notifiesIRSWhenAmountExceedsThreshold() {
        // ...
    }
}
}

```

While you could choose to instantiate the account field in a `@BeforeEach` method, there's nothing wrong with doing field-level initialization, particularly if there's not much going on. The field declaration in `AFundedAccount` initializes an account with some arbitrary name, so it's not interesting enough to warrant a `@BeforeEach` method. But if your common initialization is at all interesting or requires a series of statements, you'd definitely want it to occur within a `@BeforeEach` method.

The use of `@Nested` makes for well organized test results when you run your tests:

Test Name	Execution Time
Test Results	25 ms
AFundedAccount	25 ms
Withdrawal	20 ms
notifiesIRSWhenAmountExceedsThreshold	16 ms
reducesAccountBalance	2 ms
throwsWhenAmountExceedsBalance	2 ms
AccruingInterest	5 ms
occursWhenMinimumMet	1 ms
isReconciledWithMasterAccount	2 ms
doesNotOccurWhenMinimumNotMet	2 ms

You can clearly see the grouping of related tests, which makes it easier to find what you're looking for. The visual grouping also makes it easier to spot the glaring absence of necessary tests as well as review their names for consistency—with other tests or with your team's standards for how tests are named.

I instrumented each of the `@BeforeEach` methods, the `@Test` methods, and the constructors (implicitly defined in the listing) with `System.out` statements. Here's the output when the tests are run:

```
@BeforeAll::clearAccountRegistry
AFundedAccount(); Jeff balance = 0
    Withdrawal
        @BeforeEach::fundAccount
            notifiesIRSWhenAmountExceedsThreshold
AFundedAccount(); Jeff balance = 0
    Withdrawal
        @BeforeEach::fundAccount
            reducesAccountBalance
AFundedAccount(); Jeff balance = 0
    Withdrawal
        @BeforeEach::fundAccount
            throwsWhenAmountExceedsBalance
AFundedAccount(); Jeff balance = 0
    Accruing Interest
        @BeforeEach::fundAccount
        @BeforeEach::setInterestRate
            occursWhenMinimumMet
AFundedAccount(); Jeff balance = 0
    Accruing Interest
        @BeforeEach::fundAccount
        @BeforeEach::setInterestRate
            accruesNoInterestWhenMinimumMet
AFundedAccount(); Jeff balance = 0
    Accruing Interest
        @BeforeEach::fundAccount
        @BeforeEach::setInterestRate
            doesNotOccurWhenMinimumNotMet
```

The static `@BeforeAll` method executes first.

The output shows that a new instance of `AFundedAccount` is constructed for each test executed. It also shows that the account is, as expected, properly initialized with a name and zero balance.

Creating a new instance for each test is part of JUnit's deliberate design. It helps ensure each test is isolated from side effects that other tests might create.



JUnit creates a new instance of the test class for each test method that runs.

The `@BeforeEach` method `fundAccount`, declared within the top-level scope of the `AFundedAccount` class, executes prior to each of all six tests.

The `@BeforeEach` method `setInterestRate`, declared within the scope of `AccruingInterest`, executes only prior to each of the three tests defined within that nested class.