



"Of the 15 engineers on my team, a third are from BairesDev"
Nishant R. - **Pinterest**

500+ companies trust us with their software development needs.

Java Unit Testing With JUnit 5: Best Practices & Techniques Explained

Master Java Unit Testing: Dive into tools, best practices, and techniques to ensure robust code. Enhance software reliability and deliver flawlessly!

SOFTWARE DEVELOPMENT

10 MIN READ



Article Contents



Looking to boost your Java development efforts? This guide explores the world of Java testing, covering foundational concepts and advanced techniques. You'll learn about the importance of Test Driven Development (TDD), JUnit's setup and use, assertions for validating behavior, and best practices for writing high-quality tests. Whether you're a beginner looking to grasp the basics or an expert aiming to refine your skills, you'll find valuable insights into Java tests.

What is Java Unit Testing?

The goal of unit testing is to isolate “units” of code and test them to make sure they are working as intended. A “unit” is the smallest testable part of an application, typically a single method or class. That way, when a test fails, it's easy to pinpoint which part or “unit” is not working as intended.

But before diving into the specific steps involved in unit testing, let's look at why we should create unit tests.

Why Write Unit Tests?

Java developers often need to manually test code to see if it works as intended.

Writing unit tests helps you automate this process and ensures that the same tests run in the same environment under the same initial conditions.

Unit tests have a number of advantages, including:

1. **Easy troubleshooting:** JUnit tests will reveal when your code is not working as expected. This makes it easier for you to identify major bugs or issues before they escalate and creep into your production builds.
2. **Enable code refactoring:** Unit tests provide a safety net when your code changes so you can refactor and modify it with the confidence that it won't introduce new bugs into your software.



3. Improve code quality: Unit tests encourage developers to write more modular, testable, and maintainable code.

While writing unit tests might be time-consuming initially, it can ultimately reduce overall development time by reducing the effort spent fixing bugs and reworking code later in the development process.

Test Driven Development

Test Driven Development is a software development practice where developers write test methods before writing code. The idea is to assess for the intended behavior first. This, in many cases, makes it easier to implement the actual behavior. It's also more difficult to introduce bugs. You can fix any bugs that have appeared by writing additional tests that expose the defective code behavior.

The TDD process typically involves three steps:

- **Write failing tests:** Describe the intended behavior of your application and write test cases based on that. The tests are expected to fail.
- **Write code:** The next step is to write some code to make the tests pass. The code is written only to meet the requirements of the test and nothing more.
- **Refactoring:** Look for ways to improve the code while still maintaining its functionality. This could include simplifying the code, removing duplication, or improving its performance.

Installation of JUnit 5

Now that we have covered the importance and process of Test Driven Development, we can explore how to set up JUnit 5, one of the most popular Java testing frameworks.



Maven

To install JUnit 5 in Maven, add the following dependencies in the pom.xml file.

```
<dependencies>
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-api</artifactId>
<version>5.9.2</version>
<scope>test</scope>
</dependency><!-- For running parameterized tests -->
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-params</artifactId>
<version>5.9.2</version>
<scope>test</scope>
</dependency>
</dependencies>
```

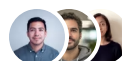
Gradle

To install and set up JUnit 5 in Gradle, add the following lines in your build.gradle file.

```
test { useJUnitPlatform() }

dependencies {
testImplementation 'org.junit.jupiter:junit-jupiter-api:5.9.2'
testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.9.2'
}
```

JUnit Packages



Both `org.junit` and `org.junit.jupiter.api` are Java unit testing packages that provide support for writing and running tests.

But `org.junit` is the older testing framework, which was introduced with JUnit 4, while `org.junit.jupiter.api` is the newer Java software testing framework introduced with JUnit 5. The latter builds upon JUnit 4 and adds new features and functionalities. The JUnit 5 framework has support for parameterized tests, parallel tests, and lambdas, among other features. For our purposes, we are going to use JUnit 5.

How to Write Unit Tests

We can mark a method as a test by adding `@Test` annotation. The method marked for testing should be public.

In JUnit 5, there are two ways to use assertion methods like `assertEquals`, `assertTrue`, and so on: static import and regular import.

A static import allows you to use only static members (such as methods) of a class without specifying the class name. In JUnit, static imports are commonly used for assertion methods. For example, instead of writing `Assert.assertEquals(expected, actual)`, you can use `assertEquals(expected, actual)` directly after using a static import statement.

```
import static org.junit.jupiter.api.Assert.*;
public class MainTest {
    @Test
    public void twoPlusTwoEqualsFalse() {
        int result = 2 + 2;
        assertEquals(4, result);
    }
}
```



```
}  
}
```

Assertions

JUnit 5 provides several built-in assertion methods that can be used to verify the behavior of the code under test. An assertion is simply a method that compares the output of a test unit with an expected result.

Throughout this article, we will be looking at several test methods. Keeping the ideas of test-driven development in mind, we will not look at the code implementation for any of these cases. Instead, we will discuss the intended behavior and edge cases (if any) and write JUnit tests based on that.

Assert.assertEquals() and Assert.assertNotEquals()

The assertEquals method is used to check whether two values are equal or not. The test passes if the expected value equals the actual value.

In the example, we are testing an “add” method, which takes two integers and returns their sum.

```
@Test  
void threePlusFiveEqualsEight() {  
    Calculator calculator = new Calculator();// syntax: assertEquals(expected  
    value, actual value, message);  
    assertEquals(8, calculator.add(3, 5));  
}
```



When comparing objects, the `assertEquals` method uses the “equals” method of the object to determine if they are equal. If the “equals” method is not overridden, only then will it perform a reference comparison. For example, calling `assertEquals` on two strings will call the `string.equals(string)` method.

Keep this in mind because arrays do not override the “equals” method. Calling `array1.equals(array2)` will only compare their references. Hence, you should not use `assertEquals` to compare arrays or any object which does not override the equals method. If you want to compare arrays, use `Arrays.equals(array1, array2)`, and if you want to test array equality, use the `assertArrayEquals` method.

Assert.assetSame()

This method compares the references of two objects or values. The test passes when the two objects have the same references. Otherwise, it fails.

Assert.assertTrue() and Assert.assertFalse()

The `assertTrue` method checks whether a given condition is true or not. The test passes only if the condition is true. Here, we are testing the `mod()` method, which returns the modulus of a number.

```
@Test
void mustGetPositiveNumber() {
    // syntax: assertTrue(condition)
    assertTrue(calculator.mod(-32) >= 0)
}
```

Similarly, the `assertFalse` method passes the test only when the condition is false.



Assert.assertNull() and Assert.assertNotNull()

As you might have guessed, the `assertNull` method expects a null value. Likewise, the `assertNotNull` method expects any value that is not null.

Assert.assertEquals()

Previously, we mentioned that using `assertEquals` on arrays does not produce the intended result. If you want to compare two arrays element by element, use `assertArrayEquals`.

Test Fixtures

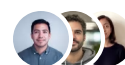
JUnit test fixtures are a set of objects, data, or code used to prepare a testing environment and provide a known starting point for testing. That includes the preparation and cleanup tasks necessary for testing a particular unit of code.

BeforeEach and AfterEach

The `@BeforeEach` annotation in JUnit is used to mark a method that should be executed before each test in a test class. The `@BeforeEach` annotation is used to prepare the test environment or set up any necessary resources before each test case is executed.

In the previous examples, instead of instantiating the `Calculator` object inside every test method, we can instantiate it in a separate method which will be called before the test runner executes a test.

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
class CalculatorTest {    Calculator calculator;
```




```
@BeforeEach
void setUp() {
    calculator = new Calculator();
}
@Test
void twoPlusTwoEqualsFour() {
    assertEquals(4, calculator.add(2, 2));
}
}
```

The `@AfterEach` annotation in JUnit is used to mark a method that should be executed after each test in a test class. The `@AfterEach` annotation can be used to clean up any resources (like databases or network connections) or reset states that were created during the execution of the test case.

BeforeAll and AfterAll

The `@BeforeAll` and `@AfterAll` annotations in JUnit are used to mark methods that **should be executed once** before and after all the test cases executed.

The primary use case of the `@BeforeAll` method is to set up any global resources or initialize any shared state that needs to be available to all the test cases in the class. For example, if a test class requires a database connection, the `@BeforeAll` method can be used to create a single database connection that can be shared by all the test cases.

Some More Assertions

After covering annotations like `@AfterEach`, `@BeforeAll`, and `@AfterAll`, we can now dive into some advanced assertions in JUnit, beginning with techniques for testing for exceptions.



Testing for exceptions

In order to check if a piece of code will throw an exception or not, you can use the `assertThrows`, which takes the class reference of the expected exception as the first argument and the piece of code you want to test as the second argument.

Now, let's say we want to test the "divide" method of our `Calculator` class, which takes two integers, divides the first number by the second number and returns a double value. However, it throws an exception (`ArithmeticException`) if the second argument is zero. We can test that by using the `assertThrows` method.

```
@Test
void testDivision() {
    assertThrows(RuntimeException.class, () -> calculator.divide(32, 0));
}
```

If you run the above test, you will notice that the test passes. As mentioned previously, the divide method will return an `ArithmeticException`, but we are not checking for that. The above code works because the `assertThrows` only checks for an exception to be thrown irrespective of the type.

Use `assertThrowsExactly` to expect an error of a fixed type. In this case, it's better to use `assertThrowsExactly`.



Blog

```
void testDivision() {
    assertThrowsExactly(ArithmeticException.class, () ->
        calculator.divide(32, 0));
}
```



The `assertNotThrows` method takes in an executable code as an argument and tests if the code throws any exception. The test passes if no exception is thrown.

Testing for Timeouts

The `assertTimeout` method allows you to test whether a block of code completes within a specified time limit. Here's the syntax of `assertTimeout`:

```
assertTimeout(Duration timeout, Executable executable)
```

`assertTimeout` executes the code under test in a separate thread, and if the code completes within the specified time limit, the assertion passes. If the code takes longer than the specified time limit to complete, the assertion fails and the test is marked as a failure.

```
@Test
void testSlowOperation() {
    assertTimeout(Duration.ofSeconds(1), () -> {
        Thread.sleep(500); // simulate a slow operation
    });
}
```

The `assertTimeoutPreemptively` assertion is a method that allows you to test if a block of code completes within a specified just like the `assertTimeout`. The only difference is that `assertTimeoutPreemptively` interrupts the execution when the time limit is exceeded.

Dynamic Tests



Dynamic tests in JUnit are tests that are generated at runtime instead of being predefined. They allow developers to generate tests programmatically based on input data. Dynamic tests are implemented using the `@TestFactory` annotation. The method annotated `@TestFactory` must return a `Stream`, `Collection`, or `Iterator` of generic type `DynamicTest`.

Here, we are testing a `subtract` method that returns the difference between the first argument and the second.

```
@TestFactory
Stream<DynamicTest> testSubtraction() {
    List<Integer> numbers = List.of(3, 7, 14, 93);

    return numbers.stream().map((number) ->
DynamicTest.dynamicTest("Test: " + number, () ->
        assertEquals(number - 4, calculator.subtract(number, 4)));
    );
}
```

Parameterized Tests

Parameterized tests allow you to write a single test method and run it multiple times with different arguments. This can be useful when you want to test a method with different input values or combinations of values.

To create a parameterized test in JUnit 5, you can use the `@ParameterizedTest` annotation and provide arguments using annotations like `@ValueSource`, `@CsvSource`, `@MethodSource`, `@ArgumentsSources`, and so on.

Passing One Argument



The `@ValueSource` annotation takes in an array of single values of any type. In the example below, we are testing a function that checks if a given number is odd or not. Here, we are using the `@ValueSource` annotations to take a list of arguments. The test runner runs the test for each value provided.

```
@ParameterizedTest
@ValueSource(ints = {3, 9, 77, 191})
void testIfNumbersAreOdd(int number) {
    assertTrue(calculator.isOdd(number), "Check: " + number);
}
```

Passing Multiple Arguments

The `@CsvSource` annotation takes in a comma-separated list of arguments as input, where each row represents a set of inputs for the test method. Here, in the example provided below, we are testing a multiplication method which returns the product of two integers.

```
@ParameterizedTest
@CsvSource({"3,4", "4,14", "15,-2"})
void testMultiplication(int value1, int value2) {
    assertEquals(value1 * value2, calculator.multiply(value1,
value2));
}
```

Passing null and empty values

The `@NullSource` annotation provides a single null argument. The test method is executed once with a null argument.



The `@EmptySource` annotation provides an empty argument. For strings, this annotation will provide an empty string as an argument.

Additionally if you want to use both null and empty arguments, use the `@NullAndEmptySource` annotation.

Passing Enums

When the `@EnumSource` annotation is used with a parameterized test method, the method is executed once for each specified enum constant.

In the example given below, the test runner runs the `testWithEnum` method for each value of the enum.

```
enum Color {  
    RED, GREEN, BLUE  
}  
  
@ParameterizedTest  
@EnumSource(Color.class)  
void testWithEnum(Color color) {  
    assertNotNull(color);  
}
```

By default, `@EnumSource` includes all the constants defined in the specified enum type. You can also customize the list of constants by specifying one or more of the following attributes.

The `name` attribute is used to specify the constant names to include or exclude, and the `mode` attribute is used to specify whether to include or exclude the names.



```
enum ColorEnum {
    RED, GREEN, BLUE
}

@ParameterizedTest
@EnumSource(value = ColorEnum.class, names = {"RED", "GREEN"}, mode =
EnumSource.Mode.EXCLUDE)
void testingEnums(ColorEnum colorEnum) {
    assertNotNull(colorEnum);
}
```

In the above example, the test case will run only once (for ColorEnum.BLUE).

Passing Arguments from File

In the example below, @CsvFileSource is used to specify a CSV file (test-data.csv) as an argument source for the testWithCsvFileSource() method. The CSV file contains three columns, which correspond to the three parameters of the method.

```
// Contents of the .csv file
// src/test/resources/test-data.csv
// 10, 2, 12
// 14, 3, 17
// 5, 3, 8

@ParameterizedTest
@CsvFileSource(resources = "/test-data.csv")
void testWithCsvFileSource(String input1, String input2, String expected)
{
    int iInput1 = Integer.parseInt(input1);
    int iInput2 = Integer.parseInt(input2);
    int iExpected = Integer.parseInt(expected);
```



```
        assertEquals(iExpected, calculator.add(iInput1, iInput2));
    }
}
```

The `resources` attribute specifies the path to the CSV file relative to the `src/test/resources` directory in your project. You can also use an absolute path if necessary.


Note that the values in the CSV file are always treated as strings. You may need to convert them to the appropriate types in your test method.

Passing Values from a Method

The `@MethodSource` annotation is used to specify a method as a source of argument for a parameterized test method. This can be useful when you want to generate test cases based on a custom algorithm or data structure.

In the example below, we are testing the `isPalindrome` method which takes an integer as input and checks if the integer is palindrome or not.

```
static Stream<Arguments> generateTestCases() {
    return Stream.of(
        Arguments.of(101, true),
        Arguments.of(27, false),
        Arguments.of(34143, true),
        Arguments.of(40, false)
    );
}

@ParameterizedTest
@MethodSource("generateTestCases")
void testWithMethodSource(int input, boolean expected) {
    // the isPalindrome(int number) method checks  given
    // input is palindrome or not
}
```



```
    assertEquals(expected, calculator.isPalindrome(input));  
}
```

Custom Arguments

The `@ArgumentsSource` (not to be confused with `ArgumentsSources`) is an annotation that can be used to specify a custom argument provider for a parameterized test method. The custom annotation provider is a class that provides arguments for the test method. The class must implement the `ArgumentsProvider` interface and override its `provideArguments()` method.

Consider the following example:

```
static class StringArgumentsProvider implements ArgumentsProvider {  
    String[] fruits = {"apple", "mango", "orange"};  
  
    @Override  
    public Stream<? extends Arguments>  
provideArguments(ExtensionContext extensionContext) throws Exception {  
    return Stream.of(fruits).map(Arguments::of);  
}  
}  
  
@ParameterizedTest  
@ArgumentsSource(StringArgumentsProvider.class)  
void testWithCustomArgumentsProvider(String fruit) {  
    assertNotNull(fruit);  
}
```

In this example, `StringArgumentsProvider` is a custom argument provider that provides strings as test arguments. The provider implements the



ArgumentsProvider interface and overrides its provideArguments() method to return a stream of Arguments.

You can use the @ArgumentsSources annotation to specify multiple argument sources for a single parameterized test method.

Nested Tests

In JUnit 5, nested test classes are a way of grouping related tests and organizing them in a hierarchical structure. Each nested test class can contain its own setup, teardown, and tests.

To define a nested test class, use the @Nested annotation before an inner class. The inner should not be static.

```
class ExampleTest {  
  
    @BeforeEach  
    void setup1() {}  
  
    @Test  
    void test1() {}  
  
    @Nested  
    class NestedTest {  
  
        @BeforeEach  
        void setup2() {}  
  
        @Test  
        void test2() {}  
  
        @Test  
        void test3() {}  
    }  
}
```



```
}  
}
```

The code will execute in the following order.

```
setup1() -> test1() -> setup1() -> setup2() -> test2() -> setup1() ->  
setup2() -> test3()
```

Just like how a test class can contain nested test classes, a nested test class can also contain its own nested test classes. This allows you to create a hierarchical structure for your tests, making it easier to organize and maintain your test code.

Test Suite

JUnit Test Suites are a way to organize your tests. Although nested tests are a great way of organizing tests, as the complexity of a project grows, it becomes harder and harder to maintain them. Moreover, before running any nested test method, all the test fixtures are run first, which may be unnecessary. Hence, we use test suites to organize our tests regularly.

In order to use the JUnit test suites, first create a new class (say `ExampleTestSuite`). Then, add the `@RunWith(Suite.class)` annotation to tell the JUnit test runner to use `Suite.class` to run the tests. The `Suite.class` runner in JUnit allows you to run multiple test classes as an entire test suite. Then, specify the classes you want to run using the `@SuiteClasses` annotation.



```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({
    CalculatorTest.class,
    CalculatorUtilsTest.class
})
public class CalculatorTestSuite {}
```

Best Practices for Writing Better Tests

Now that we've explored specific assertions, we should cover best practices that can maximize the efficacy of tests. A foundational guideline is keeping tests simple and focused, but there are additional considerations. Let's dive into some key principles to follow when writing robust, efficient unit tests.

- **Write tests that are simple and focused:** Unit tests should be simple and focused on testing one aspect of the code at a time. It should be easy to understand and maintain and should provide clear feedback on what is being tested.
- **Use descriptive test names:** Test names should be descriptive and should provide clear information about what is being tested. This helps to make the test suite more readable and understandable. To name a test, use the `@DisplayName` annotation.


```
@Test
@DisplayName("Checking nine plus seven equals sixteen")
void twoPlusTwoEqualsFour() {
```



```
    assertEquals(16, calculator.add(9,7));  
}
```

- **Using random values at runtime:** Generating random values at runtime is not recommended for unit testing. Using random values can help ensure that the code being tested is robust and can handle a wide variety of inputs. Random values can help reveal edge cases and other scenarios that might not be apparent from a static test case. However, using random values can also make tests less reliable and repeatable. If the same test is run multiple times, it may produce different results each time, which can make it difficult to diagnose and fix problems. If random values are used, it is important to document the seed used to generate them so that the tests can be reproduced.
- **Never test the implementation details:** Unit tests should focus on testing the behavior of a unit or component, not how it is implemented. Testing implementation details can make tests brittle and difficult to maintain.
- **Edge Cases:** Edge cases are cases where your code might fail. For example, if you are dealing with objects, one common edge case is when the object is null. Make sure to cover all the edge cases while writing tests.
- **Arrange-Act-Assert (AAA) pattern:** The AAA pattern is a useful pattern for structuring tests. In this pattern, the Arrange phase sets up the test data and context, the Act phase performs the operation being tested, and the Assert phase verifies that the expected results are obtained.

Mockito

Mockito is an open-source Java mocking framework that allows you to create and use mock objects in unit tests. Mock objects are used to  imitate real objects

in the system that are difficult to test in isolation.

Installation

To add mockito to your project add the following dependency in the pom.xml.

```
<!-- https://mvnrepository.com/artifact/org.mockito/mockito-core -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.3.0</version>
    <scope>test</scope>
</dependency>
```

If you are using Gradle, add the following to your build.gradle.

```
repositories { mavenCentral() }
dependencies { testImplementation "org.mockito:mockito-core:3.+" }
```

Using Mock Objects

In unit testing, we want to test the behavior of a unit of code independently of the rest of the system. However, sometimes a code module depends on other modules or some external dependencies that are difficult or impossible to test in isolation. In this case, we use mock objects to simulate the behavior of these dependencies and isolate the module under test.

In the example given below, we have a User class which we want to test. The User class depends on a UserService class which is responsible for fetching data



from a database. The UserService class has a method called getUserById which fetches information about an user from a database and returns it.

```
public class User {
    private final int id;
    private final UserService userService;

    public User(int id, UserService userService) {
        this.userService = userService;
        this.id = id;
    }

    public String getName() {
        UserInfo info = userService.getUserById(id);
        return info.getName();
    }
}

public class UserService {
    public UserInfo getUserById(int id) {
        // retrieve user information from a database
    }
}
```

To unit test the getName() method of the User class, we need to test it in isolation from the UserService class and the database.

One way to do this is to use a mock object to simulate the behavior of the UserService class. Here's an example of how to do this using Mockito:

```
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
```

```
@Test
```



```

public void testGetName() {
    UserService userService = Mockito.mock(UserService.class);

    // Create a mock UserEntity object
    UserEntity entity = new UserEntity(123, "John");
    Mockito.when(userService.getUserById(123)).thenReturn(entity);

    User user = new User(123, userService);

    String name = user.getName();
    assertEquals("John", name);

    Mockito.verify(userService).getUserById(123);
}

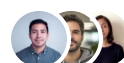
```

In the above example, we're creating a mock object for the `UserService` class using the `Mockito.mock()` method. We're then defining the behavior of the mock object using the `Mockito.when()` method, which specifies that when the `getUserById()` method is called with the argument `123`, the mock object should return a `UserEntity` object with the name "John."

We then create a `User` object with the mock `UserService` and test the `getName()` method. Finally, we verify that the mock object was used correctly using the `Mockito.verify()` method, which checks that the `getUserById()` method was called with the argument `123`.

Using a mock object in this way allows us to test the behavior of the `getName()` method in isolation from the `UserService` class and the database, ensuring that any errors or bugs are related only to the behavior of the `User` class itself.

Java Testing Frameworks



JUnit is by far the most popular choice when it comes to testing frameworks.

However, there are a lot of other options. Here are some of them:

1. **TestNG:** TestNG is another popular Java testing framework that supports a wide range of testing scenarios, including unit testing, functional testing, and integration testing. It provides advanced features like parallel testing, test dependencies, and data-driven testing.
2. **AssertJ:** AssertJ is a Java assertion library that provides a fluent API for defining assertions. It provides a wide range of assertions for testing different types of objects and supports custom assertions.
3. **Hamcrest:** Hamcrest is a Java assertion library that provides a wide range of matchers for testing different types of objects. It allows developers to write more expressive and readable tests by using natural language assertions.
4. **Selenium:** Selenium is a Java testing framework for testing web applications. It allows developers to write automated tests for web applications using a variety of programming languages, including Java.
5. **Cucumber:** Cucumber is a Java testing framework that allows developers to write automated tests in a behavior-driven development (BDD) style. It provides a simple, natural language syntax for defining tests, making it easier to write tests that are easy to read and understand.

Conclusion

In this article, we covered everything you need to know to get started with unit testing using JUnit and Mockito. We also discussed the principles of test-driven development and why you should follow it.

By adopting a test-driven development approach, you can ensure that your code behaves as intended. But like any software development practice, TDD has its pros and cons, and its effectiveness will depend on the specific project and



team. For larger projects, engaging [Java development services](#) can provide testing expertise to properly implement TDD based on your needs.

Ultimately, the decision to use TDD should factor in the project goals, team skills, and whether outside Java testing resources may be beneficial. With the right understanding of TDD tradeoffs, even inexperienced teams can reap the quality benefits of a test-first methodology.

If you enjoyed this, be sure to check out some of our other Java articles.

- [8 Best Java IDEs & Text Editors](#)
- [6 Best Java GUI Frameworks](#)
- [7 Best Java Machine Learning Libraries](#)
- [Top 5 Java Build Tools Compared](#)
- [9 Best Java Static Code Analysis Tools Listed](#)
- [Java Concurrency: Master the Art of Multithreading](#)

FAQ

How do you handle dependencies when setting up unit tests in Java?

Dependencies are automatically managed by [build tools](#) like Maven and Gradle. Hence, it is highly recommended to use them.

Can you use JUnit to test non-Java code, such as JavaScript or Python?

No, you cannot use JUnit to test non-Java code. However, languages like Javascript and Python have their own frameworks for unit testing. For example,



Javascript(React) has Jest and Python has PyTest for unit testing.

What are some common pitfalls to avoid when writing unit tests and how can you mitigate them?

While writing unit tests, make sure your tests are simple and focused on testing one aspect of the code at a time. Use descriptive names and group similar tests. Try to cover all the edge cases.



Article tags: [java](#)



By [BairesDev Editorial Team](#)

Founded in 2009, BairesDev is the leading nearshore technology solutions company, with 4,000+ professionals in more than 50 countries, representing the top 1% of tech talent. The company's goal is to create lasting value throughout the entire digital transformation journey.



[Previous article](#)

[Java Integration Testing Explained With Examples](#)

[Next article](#)



Java Unit Testing With JUnit 5: Best Practices & Techniques Explained

Hiring engineers?

We provide nearshore tech talent to companies from startups to enterprises like Google and Rolls-Royce.



Alejandro D.
Sr. Full-stack Dev.



Gustavo A.
Sr. QA Engineer



Fiorella G.
Sr. Data Scientist

[About Us](#)

[Our Services](#)

[Our Clients](#)

Related articles

Why Scaling DevOps Feels Like Herding Cats

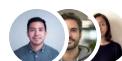
Automating Quality Assurance: Essential Strategies for Growth Without Sacrificing Stability

Top 5 Software Anti Patterns to Avoid for Better Development Outcomes

The Essential Guide to Software Scalability for Growing Applications

Swift: A Beginner's Guide to iOS Development

Swift Protocols Will Change How You Think About Code





Discover BairesDev. Resources.

About Us	Case Studies
Methodologies	Blog
Technologies	Press
Certifications	Software Development Insights
Our Services	Technologies Insights
Dedicated Teams	Industries Insights
Staff Augmentation	Technology Resource Center
Software Outsourcing	Client Referral Program
Expertise	
Diversity	
Social Responsibility	
Senior Advisor Program	

Careers.

Job Opportunities
Talent Referrals

[Privacy Policy](#) | [Terms of Service](#) | [Do Not Sell My Personal Information](#)

BairesDev 2009 - 2025. All rights reserved.

Get insights from the experts on building and scaling technology teams.

Your e-mail address

Subscribe 



By subscribing I accept the [Privacy Policy](#).

Get in touch.

Contact Us

Schedule a Call

+1 (408) 478-2739

Follow us.

