JUnit vs. TestNG:
Which Framework Fits Your Testing Strategy?

JavaCro'25
Rovinj, Croatia
October 14, 2025

Boni García
https://bonigarcia.dev/

Introduction

> *A **unit testing framework** is a tool that provides structure and reusable components to write, organize, and run automated tests for given pieces of code, ensuring they behave as expected.*

JUnit

TestNG

Introduction – JUnit

- Originally created by Kent Beck and Erich Gamma in 1999
- JUnit 4 was the most popular dependency in Java
  - Based on Java annotations (e.g., `@Test`)
- JUnit 5 (first released on 2017) was a key milestone:
  - JUnit Platform, providing the launching infrastructure for running tests and defining the API for test engines (like JUnit Jupiter) to discover and execute tests
- JUnit 6 has been released on September 30, 2025
  - No general API changes
  - Support for Java 17 and remove deprecated APIs

https://docs.junit.org/current/user-guide/

Introduction – TestNG

- Created by Cédric Beust in 2004
- Inspired by JUnit but designed to overcome its limitations
  - Also based on Java annotations (e.g., `@Test`)
- Key features:
  - Built-in support for grouping and parameterized tests
  - Native support for parallel test execution
- Still actively maintained
  - Current stable release: TestNG 7.11.0 (Java 17+)

TestNG

https://testng.org/

Introduction – JUnit vs. TestNG

- This talk compares different features of JUnit and TestNG:

Test lifecycle (basics)	Parameterized tests	Categorizing and filtering tests
Conditional test execution	Ordering tests	Parallel test execution
	Advanced test lifecycle	

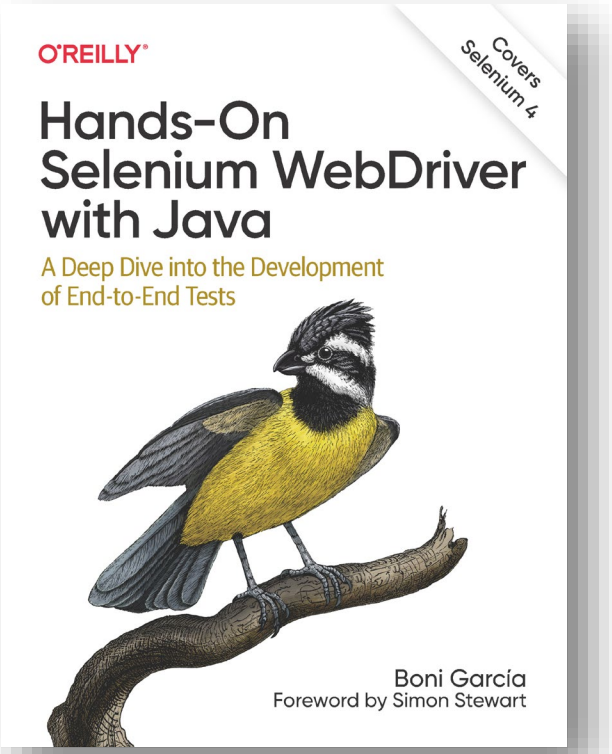
Introduction – Examples

- As a real use case, I will use **Selenium** to illustrate the key differences between JUnit and TestNG

> *Selenium is a **browser automation library**, not a **testing framework**.*

https://selenium.dev/

Introduction – Examples



https://github.com/bonigarcia/mastering-junit5

https://github.com/bonigarcia/selenium-webdriver-java

Test lifecycle (basics)

Parameterized tests

Categorizing and filtering tests

Conditional test execution

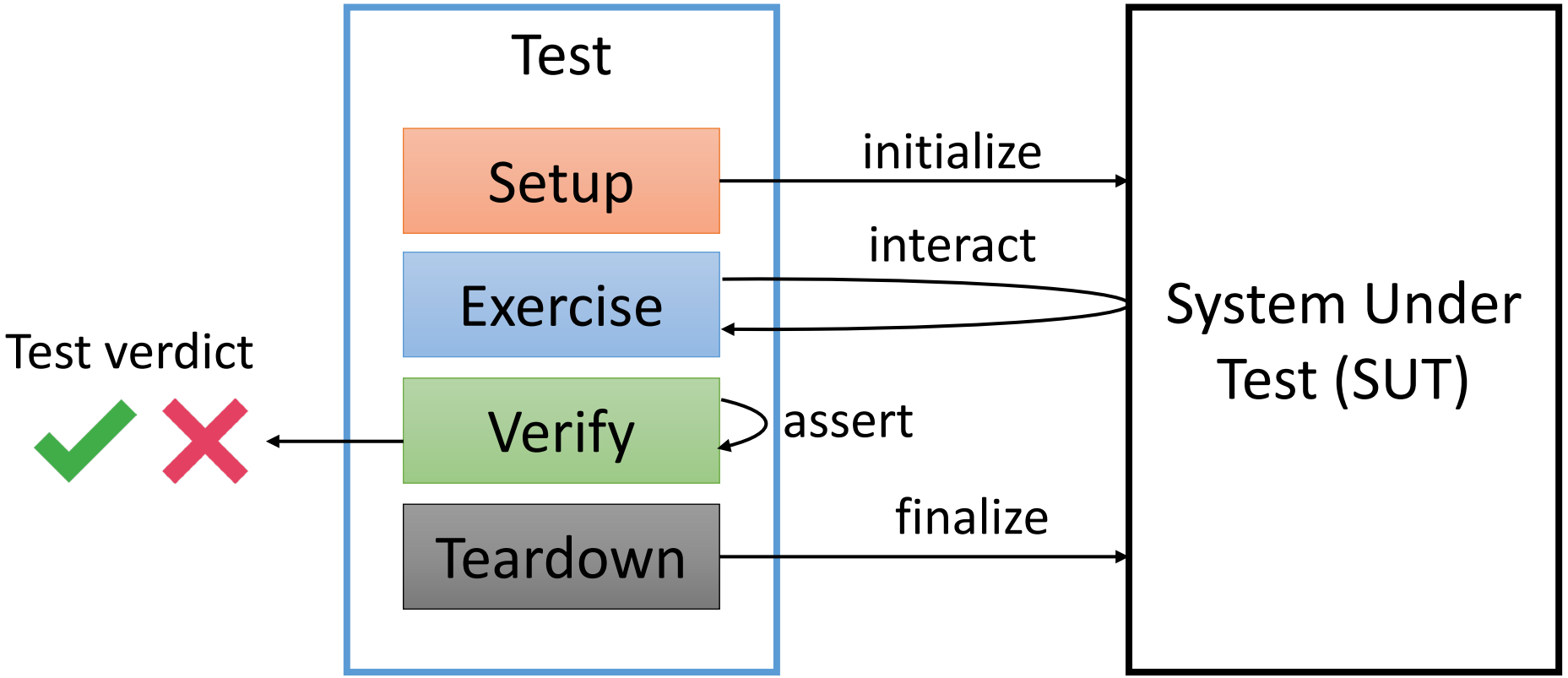
Ordering tests

Parallel test execution

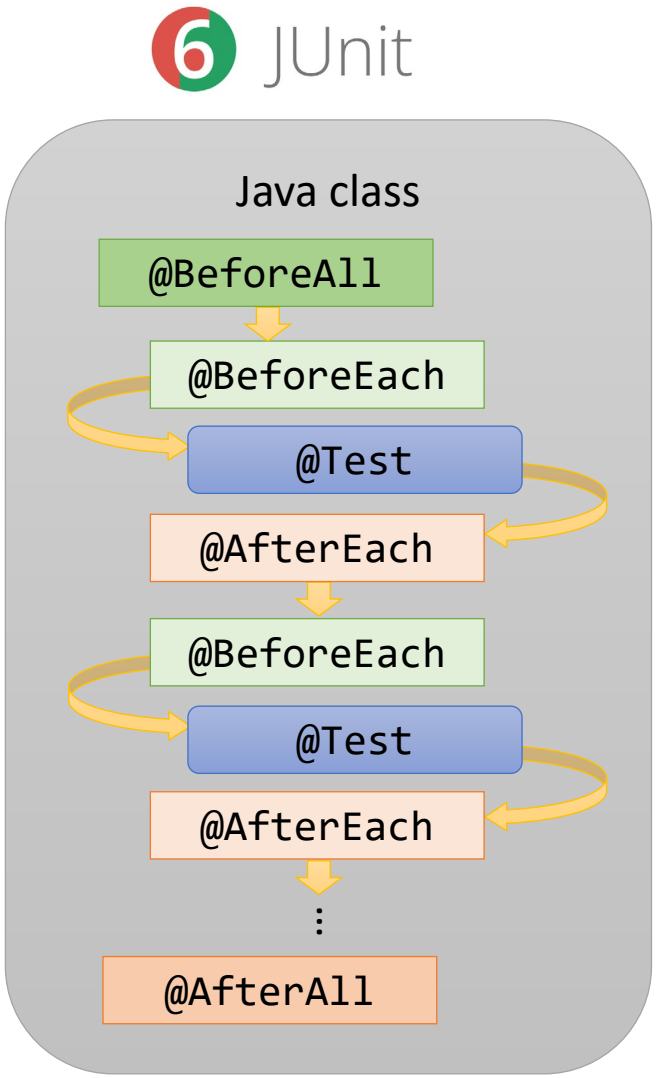
Advanced test lifecycle

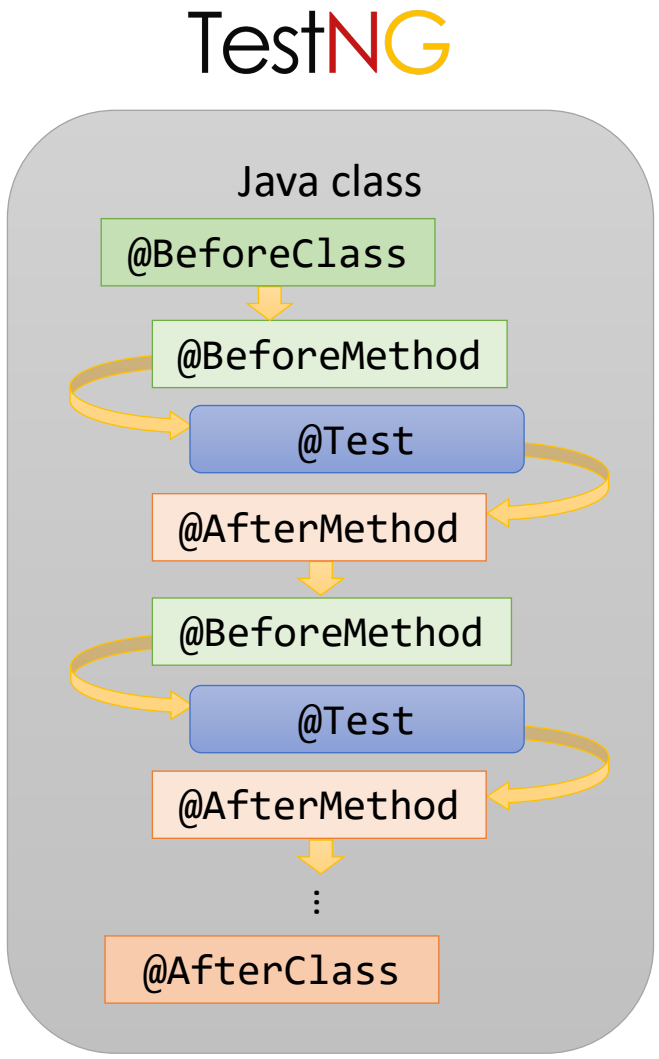
Test Lifecycle (Basics)

> *The **test lifecycle** is the sequence of steps a testing framework follows to set up the test fixture (initial state), execute the test(s), and clean up afterward*



Test Lifecycle (Basics)





Test Lifecycle (Basics) – JUnit vs. TestNG

- Example #1: basic test with Selenium

```java
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

class HelloWorldSeleniumJupiterTest {

    WebDriver driver;

    @BeforeEach
    void setup() {
        driver = new ChromeDriver();
    }

    @Test
    void test() {
        // Test logic
    }

    @AfterEach
    void teardown() {
        driver.quit();
    }
}
```

JUnit

```java
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class HelloWorldSeleniumNGTest {

    WebDriver driver;

    @BeforeMethod
    public void setup() {
        driver = new ChromeDriver();
    }

    @Test
    public void test() {
        // Test logic
    }

    @AfterMethod
    public void teardown() {
        driver.quit();
    }
}
```

TestNG

Test Lifecycle (Basics) – JUnit vs. TestNG

```java
import org.junit.jupiter.api.Test;

class BasicSeleniumJupiterTest extends BrowserParent {

    @Test
    void test() {
        // Test logic
    }
}
```

```java
import org.testng.annotations.Test;

public class BasicSeleniumNGTest extends BrowserParent {

    @Test
    void test() {
        // Test logic
    }
}
```

```java
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

class BrowserParent {

    WebDriver driver;

    @BeforeEach
    void setup() {
        driver = new ChromeDriver();
    }

    @AfterEach
    void teardown() {
        driver.quit();
    }
}
```

JUnit

```java
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;

public class BrowserParent {

    WebDriver driver;

    @BeforeMethod
    public void setup() {
        driver = new ChromeDriver();
    }

    @AfterMethod
    void teardown() {
        driver.quit();
    }
}
```

TestNG

Test lifecycle (basics)

Parameterized tests

Categorizing and filtering tests

Conditional test execution

Ordering tests

Parallel test execution

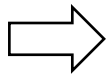
Advanced test lifecycle

Parameterized Tests

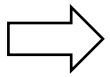
> *A **parameterized test** is a test that runs multiple times with different input values, allowing us to reuse the same logic across varied datasets*

Dataset

Param1	Param2
Value1	Value2
Value3	Value4
Value5	Value6



```
@Test

Param1
Param2
```



```
@Test

Value1
Value2
```

```
@Test

Value3
Value4
```

```
@Test

Value5
Value6
```

Parameterized Tests – JUnit

- To implement a parameterized test in JUnit we need to:
  - Use `@ParameterizedTest` (instead of `@Test`) or `@ParameterizedClass` (in addition to `@Test`)
  - Use an argument provider (to define the dataset)

Argument provider	Description
`@ValueSource`	Arrays of `String`, `int`, `long`, or `double`
`@EnumSource`	Enumerated types (`java.lang.Enum`)
`@MethodSource`	Static method that provides an `Stream` of values
`@CsvSource`	Comma-separated values
`@CsvFileSource`	CSV file in the classpath
`@ArgumentsSource`	Class implementing `org.junit.jupiter.params.provider.ArgumentsProvider`
`@FieldSource`	Values (`Collection`, `Iterable`, array, streams) from an static field

Parameterized Tests – TestNG

- There are two ways of implementing parameterized tests in TestNG:
  1. Using `@DataProvider` (most common and scalable)
  2. Using `@Parameters` + `testng.xml` (dataset in `<test></test>`)

```java
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class ParameterizedXmlNGTest {

    @Test
    @Parameters({ "label", "amount" })
    public void test(String label, int amount) {
        System.out.println("[XML] Label: " + label +
                " -- Amount: " + amount);
    }

}
```

```xml
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="TestSuite" verbose="1" parallel="false">
    <test name="ParameterizedTest-1">
        <parameter name="label" value="hello" />
        <parameter name="amount" value="10" />
        <classes>
            <class name="io.github.bonigarcia.testng.parameterized.ParameterizedXmlNGTest" />
        </classes>
    </test>
    <test name="ParameterizedTest-2">
        <parameter name="label" value="world" />
        <parameter name="amount" value="20" />
        <classes>
            <class name="io.github.bonigarcia.testng.parameterized.ParameterizedXmlNGTest" />
        </classes>
    </test>
</suite>
```

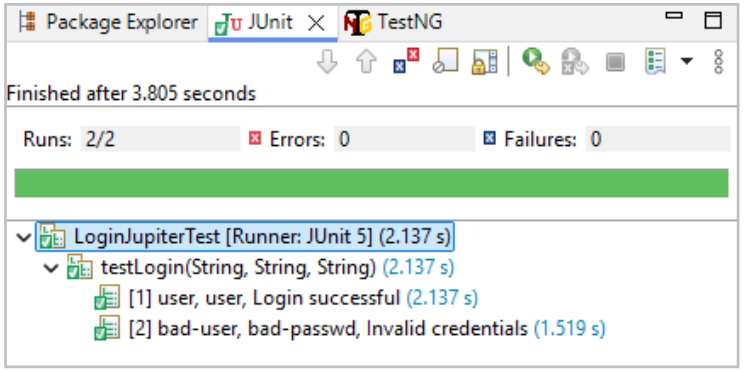
```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running TestSuite
[XML] Label: hello -- Amount: 10
[XML] Label: world -- Amount: 20
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.772 s -- in TestSuite
```

Parameterized Tests – JUnit vs. TestNG

- Example #2: data-driven test case with Selenium

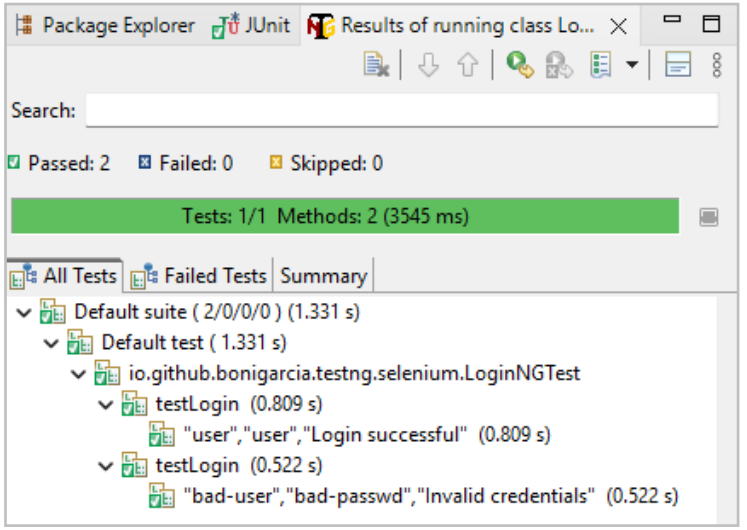
```java
class LoginJupiterTest extends BrowserParent {

    static Stream<Arguments> loginData() {
        return Stream.of(Arguments.of("user", "user", "Login successful"),
                Arguments.of("bad-user", "bad-passwd", "Invalid credentials"));
    }

    @ParameterizedTest
    @MethodSource("loginData")
    void testLogin(String username, String password, String expectedText) {
        // Test logic
    }
}
```

JUnit




```java
public class LoginNGTest extends BrowserParent {

    @DataProvider(name = "loginData")
    public static Object[][] data() {
        return new Object[][] { { "user", "user", "Login successful" },
                { "bad-user", "bad-passwd", "Invalid credentials" } };
    }

    @Test(dataProvider = "loginData")
    public void testLogin(String username, String password, String expectedText) {
        // Test logic
    }

}
```

TestNG



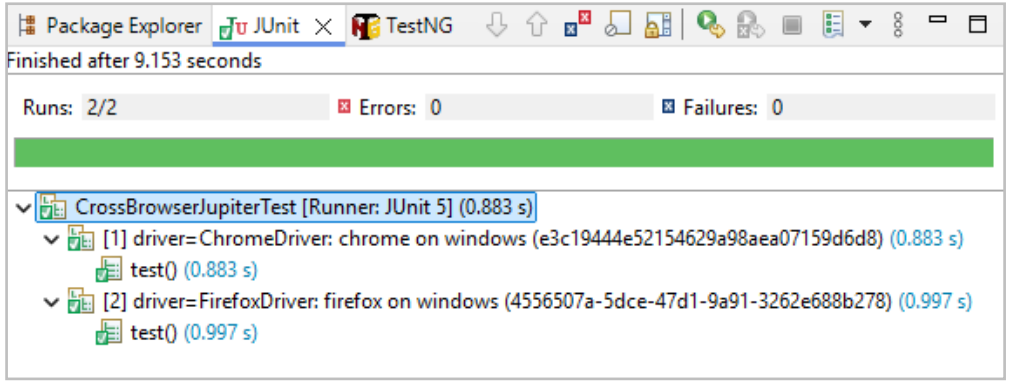
Parameterized Tests – JUnit vs. TestNG

- Example #3: cross-browser testing with Selenium

```java
@ParameterizedClass
@ArgumentsSource(CrossBrowserProvider.class)
class CrossBrowserParent {

    @Parameter
    WebDriver driver;

    @AfterEach
    void teardown() {
        driver.quit();
    }

}
```

```java
class CrossBrowserProvider implements ArgumentsProvider {

    @Override
    public Stream<? extends Arguments> provideArguments(
            ExtensionContext context) {
        ChromeDriver chrome = new ChromeDriver();
        FirefoxDriver firefox = new FirefoxDriver();

        return Stream.of(Arguments.of(chrome), Arguments.of(firefox));
    }

}
```

```java
class CrossBrowserJUnitTest extends CrossBrowserParent {

    @Test
    void test() {
        // Test logic
    }
}
```

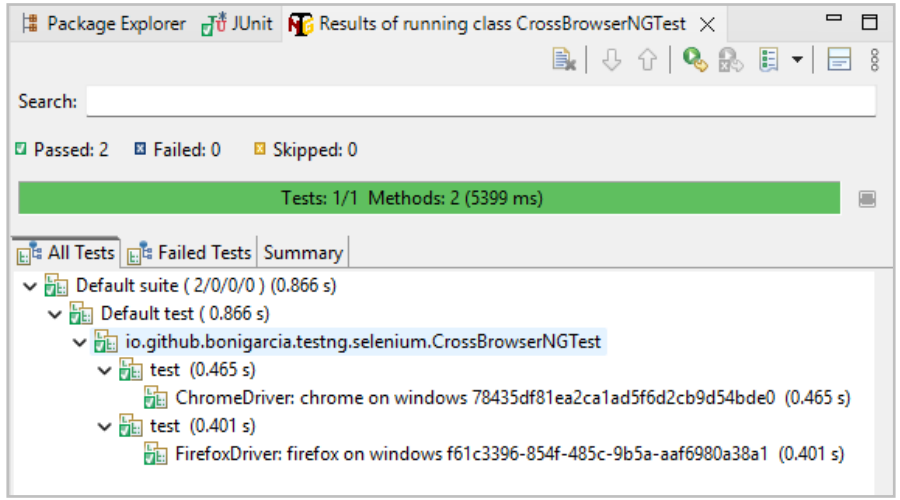


Parameterized Tests – JUnit vs. TestNG

- Example #3: cross-browser testing with Selenium

```java
public class CrossBrowserNGTest extends CrossBrowserParent {

    @Test(dataProvider = "browserProvider")
    public void test(WebDriver driver) {
        this.driver = driver;

        // Test logic
    }

}
```

TestNG




```java
public class CrossBrowserParent {

    WebDriver driver;

    @DataProvider(name = "browserProvider")
    public static Object[][] data() {
        ChromeDriver chrome = new ChromeDriver();
        FirefoxDriver firefox = new FirefoxDriver();

        return new Object[][] { { chrome }, { firefox } };
    }

    @AfterMethod
    void teardown() {
        driver.quit();
    }

}
```

Test lifecycle (basics)

Parameterized tests

Categorizing and filtering tests

Conditional test execution

Ordering tests

Parallel test execution

Advanced test lifecycle

Categorizing and Filtering Tests

> ***Categorizing and filtering*** *allows us to group tests into categories and run only the ones that match specific criteria.*

- In JUnit, test classes and methods can be tagged in JUnit using `@Tag`
- In TestNG, test methods can be grouped using the attribute `groups` in `@Test`
- Those categories (tags or groups) can later be used to filter test discovery and execution

Categorizing and Filtering Tests

- Example #5: grouping Selenium tests

```java
class CategoriesJUnitTest extends BrowserParent {

    @Test
    @Tag("WebForm")
    void testCategoriesWebForm() {
        // Test logic
    }

    @Test
    @Tag("HomePage")
    void testCategoriesHomePage() {
        // Test logic
    }

}
```

JUnit

```java
public class CategoriesNGTest extends BrowserGroupsParent {

    @Test(groups = { "WebForm" })
    public void testCategoriesWebForm() {
        // Test logic
    }

    @Test(groups = { "HomePage" })
    public void tesCategoriestHomePage() {
        // Test logic
    }
}
```

TestNG

```
mvn test -Dgroups=HomePage

gradle test -Pgroups=HomePage

mvn test -Dtest=CategoriesNGTest -DexcludedGroups=HomePage

gradle test --tests CategoriesNGTest -PexcludedGroups=HomePage
```

```java
public class BrowserGroupsParent {

    WebDriver driver;

    @BeforeMethod(alwaysRun = true)
    public void setup() {
        driver = new ChromeDriver();
    }

    @AfterMethod(alwaysRun = true)
    void teardown() {
        driver.quit();
    }
}
```

Test lifecycle (basics)

Parameterized tests

Categorizing and filtering tests

Conditional test execution

Ordering tests

Parallel test execution

Advanced test lifecycle

Conditional Test Execution

> ***Conditional test execution*** *allows us to enable or skip tests based on predefined conditions.*

- JUnit provides a rich set of built-in annotations for skipping tests (`@Disabled` and others)
  - Also, we can use `Assumptions` to disable tests in runtime
- TestNG provides the annotation `@Ignore` and attributes in `@Test` (e.g., `enabled=false`) to run conditionally
  - Also, we can use `SkipException` to disable tests in runtime

Conditional Test Execution – JUnit

- JUnit annotations for disabling tests:

Annotation(s)	Description
`@Disabled`	To disable test class or method
`@DisabledOnJre` `@EnabledOnJre`	To disable/enable depending on the Java version
`@DisabledOnJreRange` `@EnabledOnJreRange`	To disable/enable depending on a range of Java versions
`@DisabledOnOs` `@EnabledOnOs`	To disable/enable depending on the operating system (e.g., Windows, Linux, macOS, etc.)

`@DisabledIfSystemProperty`
`@DisabledIfSystemProperties`
`@EnabledIfSystemProperty`
`@EnabledIfSystemProperties`
	To disable/enable depending on the value of system properties

`@DisabledIfEnvironmentVariable`
`@DisabledIfEnvironmentVariables`
`@EnabledIfEnvironmentVariable`
`@EnabledIfEnvironmentVariables`
	To disable/enable depending on the value of an environment variable

`@DisabledIf` `@EnabledIf`	To disable/enable based on the boolean return of a custom method

Conditional Test Execution – JUnit vs. TestNG

- Example #6: skipping tests (I)

```java
class DisabledJupiterTest {

    @Disabled("Optional reason for disabling")
    @Test
    public void testDisabled1() {
        // Test logic
    }

    @DisabledOnJre(JAVA_17)
    @Test
    public void testDisabled2() {
        // Test logic
    }

    @EnabledOnOs(MAC)
    @Test
    public void testDisabled3() {
        // Test logic
    }

}
```


```java
public class DisabledNGTest {

    @Ignore("Optional reason for disabling")
    @Test
    public void testDisabled1() {
        // Test logic
    }

    @Test(enabled = false)
    public void testDisabled2() {
        // Test logic
    }

}
```

TestNG

Conditional Test Execution – JUnit vs. TestNG

- Example #7: skipping tests (II)

```java
class ConditionalJupiterTest {

    @Test
    public void testConditional() {
        boolean condition = false; // runtime condition
        Assumptions.assumeTrue(condition);

        // Test logic
    }
}
```


```java
public class ConditionalNGTest {

    @Test
    public void testConditional() {
        boolean condition = false; // runtime condition
        if (!condition) {
            throw new SkipException("Skipping test");
        }

        // Test logic
    }

}
```

Test lifecycle (basics)

Parameterized tests

Categorizing and filtering tests

Conditional test execution

Ordering tests

Parallel test execution

Advanced test lifecycle

Ordering Tests

> ***Ordering tests*** *is used to control the sequence in which tests are executed.*

- The default order for test execution are:
  - In JUnit, tests are run in an unspecified order (not guaranteed, deterministic algorithm that is but intentionally nonobvious meant to be independent)
  - In TestNG, tests are run in alphabetical order by method name
- To change this behavior:
  - In JUnit, we use `@TestMethodOrder` with `@Order`
  - In TestNG, we use `priority` and `dependsOnMethods` in `@Test`, or class order in `testng.xml`

Ordering Tests – JUnit vs. TestNG

- Example #8: reuse the same browser to run tests in a given order

```java
@TestInstance(Lifecycle.PER_CLASS)
@TestMethodOrder(OrderAnnotation.class)
class OrderJunitTest {

    WebDriver driver;

    @BeforeAll
    void setup() {
        driver = new ChromeDriver();
    }

    @Test
    @Order(1)
    void testA() {
        // Test logic
    }

    @Test
    @Order(2)
    void testB() {
        // Test logic
    }

    @AfterAll
    void teardown() {
        driver.quit();
    }
}
```

JUnit

```java
public class OrderNGTest {

    WebDriver driver;

    @BeforeClass
    public void setup() {
        driver = new ChromeDriver();
    }

    @Test(priority = 1)
    public void testA() {
        // Test logic
    }

    @Test(priority = 2)
    public void testB() {
        // Test logic
    }

    @AfterClass
    public void teardown() {
        driver.quit();
    }

}
```

TestNG

Test lifecycle (basics)

Parameterized tests

Categorizing and filtering tests

Conditional test execution

Ordering tests

Parallel test execution

Advanced test lifecycle

Parallel Test Execution

> ***Parallel test execution*** *allows us to run multiple tests simultaneously to speed up execution.*

- JUnit provides different configuration parameters to tests in parallel
  - `junit.jupiter.execution.parallel.enabled` (to enable test parallelism)
  - `junit.jupiter.execution.parallel.mode.classes.default` (to run test classes in parallel)
  - `junit.jupiter.execution.parallel.mode.default` (to run test methods in parallel)
  - These parameters can be specified using a configuration file or in runtime trough annotations (`@Execution`)
- TestNG enables parallelism using the `testng.xml` config file

Parallel Test Execution – JUnit vs. TestNG

- Example #9: run Selenium tests in parallel

```
junit.jupiter.execution.parallel.enabled = true
junit.jupiter.execution.parallel.mode.default = concurrent
junit.jupiter.execution.parallel.mode.classes.default = same_thread
```


```
@Execution(ExecutionMode.CONCURRENT)
class Parallel1JupiterTest extends BrowserParent {

    @Test
    void testParallel1() {
        // Test logic
    }

}
```

```
@Execution(ExecutionMode.CONCURRENT)
class Parallel2JupiterTest extends BrowserParent {

    @Test
    void testParallel2() {
        // Test logic
    }

}
```

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="parallel-suite" parallel="classes" thread-count="2">
    <test name="parallel-tests">
        <classes>
            <class name="io.github.bonigarcia.testng.selenium.HelloWorldSeleniumNGTest" />
            <class name="io.github.bonigarcia.testng.selenium.BasicSeleniumNGTest" />
        </classes>
    </test>
</suite>
```

TestNG

Test lifecycle (basics)

Parameterized tests

Categorizing and filtering tests

Conditional test execution

Ordering tests

Parallel test execution

Advanced test lifecycle

Advanced test lifecycle – JUnit

- In JUnit 5+, the **extension model** provides comprehensive capabilities to customize and hook into the test lifecycle at various points

Category	Description	Extension Point(s)

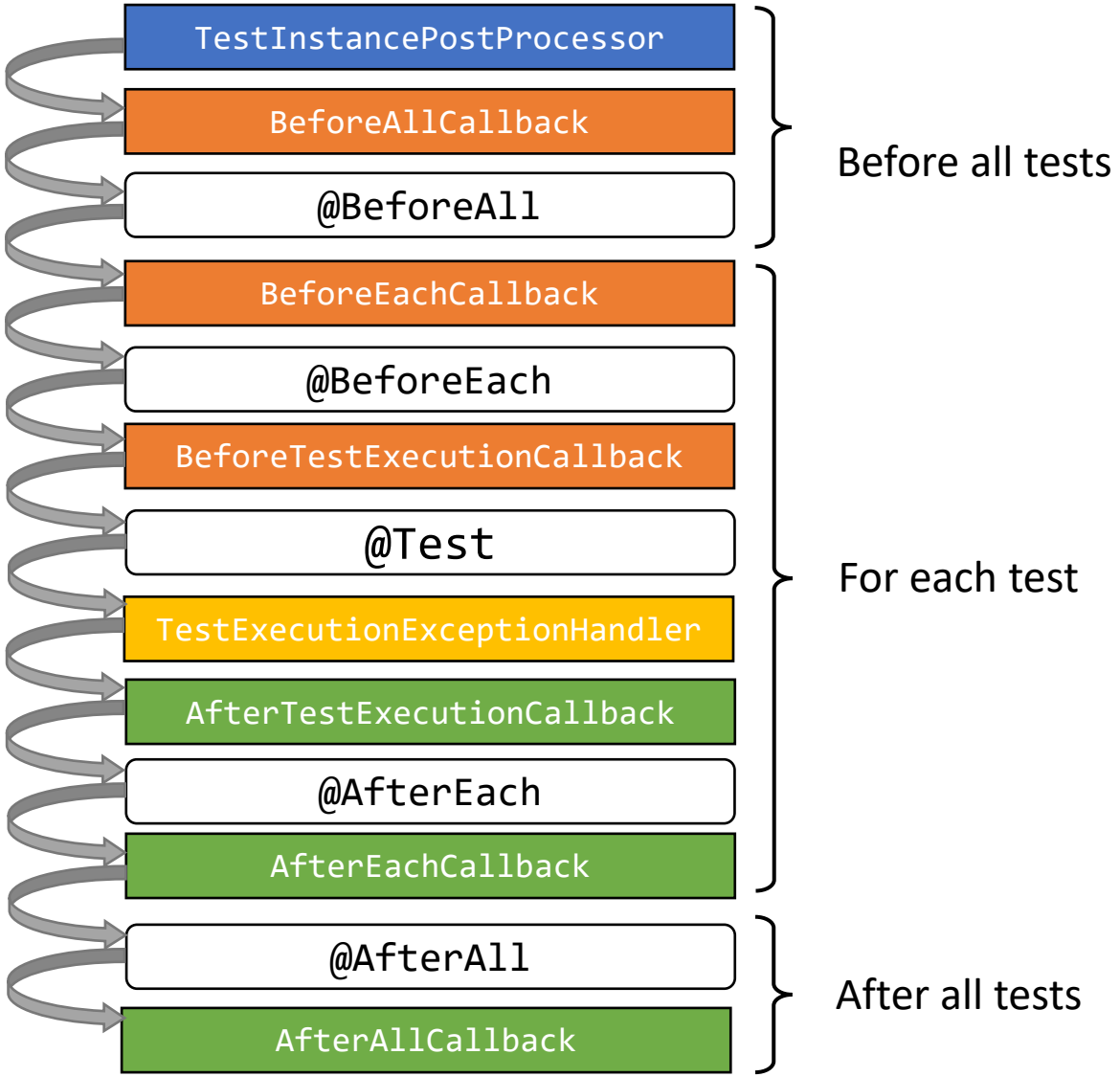
Test lifecycle callbacks	To include custom logic during the test lifecycle	`BeforeAllCallback`
		`BeforeEachCallback`
		`BeforeTestExecutionCallback`
		`AfterTestExecutionCallback`
		`AfterEachCallback`
		`AfterAllCallback`

Parameter resolution	To inject parameters in test methods or constructors	`ParameterResolver`
Test templates	To implement tests using @TestTemplate	`TestTemplateInvocationContextProvider`
Conditional test execution	To enable or disable tests depending on custom conditions	`ExecutionCondition`
Exception handling	To handle exceptions during the test and its lifecycle	`TestExecutionExceptionHandler` `LifecycleMethodExecutionExceptionHandler`

Test instance	To create and process test class instances	`TestInstanceFactory`
		`TestInstancePostProcessor`
		`TestInstancePreDestroyCallback`

Intercepting invocations	To intercept calls to test code (and proceed or not)	`InvocationInterceptor`

Advanced test lifecycle – JUnit



JUnit

Advanced test lifecycle – TestNG

- TestNG provides a rich set of **listeners** to intercept lifecycle events for tests and suites

Listener Interface	Description	Key Methods TestNG
`ITestListener`	Hooks into the lifecycle of individual test methods	`onTestStart, onTestSuccess, onTestFailure, onTestSkipped, onTestFailedButWithinSuccessPercentage`
`ISuiteListener`	Hooks into the start and finish of a test suite	`onStart(ISuite suite), onFinish(ISuite suite)`
`IInvokedMethodListener`	Intercepts every method invocation, including configuration methods (`@Before*`, `@After*`)	`beforeInvocation, afterInvocation`
`IReporter`	Generates custom reports after the suite execution	`generateReport(List<XmlSuite> xmlSuites, List<ISuite> suites, String outputDirectory)`
`IAnnotationTransformer`	Allows modification of test annotations at runtime	`transform(ITestAnnotation annotation, Class testClass, Constructor testConstructor, Method testMethod)`
`IHookable`	Intercepts the execution of test methods	`run(IHookCallBack callBack, ITestResult testResult)`
`IExecutionListener`	Hooks into the start and finish of the entire test execution (suite run)	`onExecutionStart(), onExecutionFinish()`
`IDataProviderListener`	Monitors data provider usage for test methods	`beforeDataProviderInvocation, afterDataProviderInvocation`
`IRetryAnalyzer`	Re-run a failed test a number of times	`boolean retry(ITestResult result)`

Advanced test lifecycle – JUnit vs. TestNG

- Example #10: retrying Selenium tests (to detect flakiness)

```java
@ExtendWith(RetryExtension.class)
class RandomCalculatorJupiterTest extends BrowserParent {

    @Test
    void testRandomCalculator() {
        // Test logic
    }

}
```

```java
class RandomCalculatorJupiterTest extends BrowserParent {

    @RegisterExtension
    Extension failureWatcher = new RetryExtension(5);

    @Test
    void testRandomCalculator() {
        // Test logic
    }

}
```


```java
public class RetryExtension implements TestExecutionExceptionHandler {

    static final int DEFAULT_MAX_RETRIES = 3;

    final AtomicInteger retryCount = new AtomicInteger(1);
    final AtomicInteger maxRetries = new AtomicInteger(DEFAULT_MAX_RETRIES);

    public RetryExtension() {
        // Default constructor
    }

    public RetryExtension(int maxRetries) {
        this.maxRetries.set(maxRetries);
    }

    @Override
    public void handleTestExecutionException(ExtensionContext extensionContext,
            Throwable throwable) throws Throwable {
        // Manage throwable depending on the retry count
    }

}
```

Advanced test lifecycle – JUnit vs. TestNG

- Example #10: retrying Selenium tests (to detect flakiness)

```java
public class RandomCalculatorNGTest extends BrowserParent {

    @Test(retryAnalyzer = RetryAnalyzer.class)
    @Retry(5)
    public void testRandomCalculator() {
        // Test logic
    }

}
```

```java
@Retention(RetentionPolicy.RUNTIME)
public @interface Retry {
    int value();
}
```

TestNG

```java
public class RetryAnalyzer implements IRetryAnalyzer {

    static final int DEFAULT_MAX_RETRIES = 3;

    final AtomicInteger retryCount = new AtomicInteger(1);

    @Override
    public boolean retry(ITestResult result) {
        Method method = result.getMethod().getConstructorOrMethod().getMethod();
        int maxRetries = DEFAULT_MAX_RETRIES;
        if (method.isAnnotationPresent(Retry.class)) {
            Retry retry = method.getAnnotation(Retry.class);
            maxRetries = retry.value();
        }
        if (retryCount.get() <= maxRetries) {
            logError(result.getThrowable());
            retryCount.incrementAndGet();
            return true;
        }
        return false;
    }

    private void logError(Throwable e) {
        System.err.println("Attempt test execution #" + retryCount.get()
                + " failed (" + e.getClass().getName() + "thrown):  "
                + e.getMessage());
    }

}
```

Advanced test lifecycle – JUnit vs. TestNG

- Example #11: gather data (e.g., browser screenshot) if test fails

```
@ExtendWith(FailureWatcher.class)
class FailureJupiterTest extends BrowserParent {

    @Test
    void testFailure() {
        // Test logic
        fail("Forced error");
    }

}
```


```
public class FailureWatcher implements TestExecutionExceptionHandler {

    @Override
    public void handleTestExecutionException(ExtensionContext context,
            Throwable throwable) throws Throwable {

        context.getTestInstance().ifPresent(testInstance -> {
            WebDriver driver = (WebDriver) SeleniumUtils
                    .getFieldFromTestInstance(testInstance, "driver");
            SeleniumUtils.getScreenshotAsFile(driver, context.getDisplayName());
        });

        throw throwable;
    }

}
```

Advanced test lifecycle – JUnit vs. TestNG

- Example #11: gather data (e.g., browser screenshot) if test fails

```
public class FailureNGTest {

    WebDriver driver;

    @BeforeMethod
    public void setup() {
        driver = new ChromeDriver();
    }

    @AfterMethod
    public void teardown(ITestResult result) {
        if (result.getStatus() == ITestResult.FAILURE) {
            SeleniumUtils.getScreenshotAsFile(driver, result.getName());
        }

        driver.quit();
    }

    @Test
    public void testFailure() {
        // Test logic
        fail("Forced error");
    }

}
```

TestNG

Advanced test lifecycle – JUnit vs. TestNG

- Example #12: reporting test suite

```java
@ExtendWith(Reporter.class)
class Report1JupiterTest extends BrowserParent {

    @Test
    void testReport1() {
        // Test logic
    }

}
```

```java
@ExtendWith(Reporter.class)
class Report2JupiterTest extends BrowserParent {

    @Test
    void testReport2() {
        // Test logic
    }

}
```


```java
public class Reporter implements BeforeAllCallback, BeforeEachCallback,
        AfterTestExecutionCallback {
    static final String REPORT_NAME = "report-junit.html";
    ExtentReports report;
    ExtentTest test;

    @Override
    public void beforeAll(ExtensionContext context) throws Exception {
        Store store = context.getRoot()
                .getStore(ExtensionContext.Namespace.create(STORE_NAMESPACE));
        report = store.get(STORE_NAME, ExtentReports.class);
        if (report == null) {
            report = new ExtentReports();
            store.put(STORE_NAME, report);

            Runtime.getRuntime().addShutdownHook(new Thread(report::flush));
        }
        ExtentSparkReporter htmlReporter = new ExtentSparkReporter(REPORT_NAME);
        report.attachReporter(htmlReporter);
    }

    @Override
    public void beforeEach(ExtensionContext context) throws Exception {
        test = report.createTest(context.getDisplayName());
    }

    @Override
    public void afterTestExecution(ExtensionContext context) throws Exception {
        context.getTestInstance().ifPresent(testInstance -> {
            // Take screenshot
            test.addScreenCaptureFromBase64String(screenshot);
        });
    }

}
```

Advanced test lifecycle – JUnit vs. TestNG

- Example #12: reporting test suite

```java
@Listeners(Reporter.class)
public class Report1NGTest extends BrowserParent {

    @Test
    public void testReport1() {
        // Test logic
    }

}
```

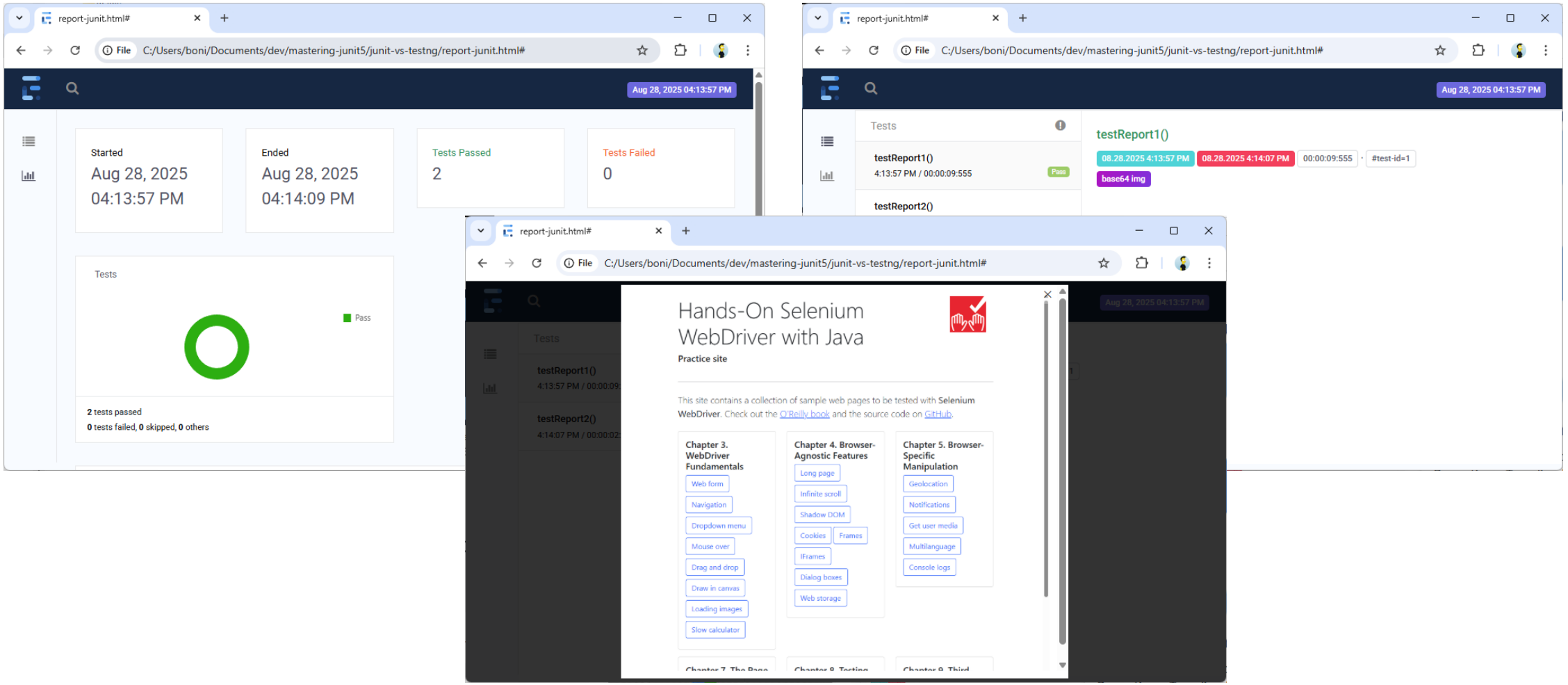
```java
@Listeners(Reporter.class)
public class Report2NGTest extends BrowserParent {

    @Test
    public void testReport2() {
        // Test logic
    }

}
```

TestNG


```java
public class Reporter implements ITestListener {

    static final String REPORT_NAME = "report-testng.html";
    ExtentReports report;
    ExtentTest test;

    @Override
    public void onStart(ITestContext context) {
        ITestListener.super.onStart(context);
        report = new ExtentReports();
        ExtentSparkReporter htmlReporter = new ExtentSparkReporter(REPORT_NAME);
        report.attachReporter(htmlReporter);
    }

    @Override
    public void onTestStart(ITestResult result) {
        ITestListener.super.onTestStart(result);
        test = report.createTest(result.getName());
    }

    @Override
    public void onTestSuccess(ITestResult result) {
        ITestListener.super.onTestSuccess(result);
        // Take screenshot
        test.addScreenCaptureFromBase64String(screenshot);
    }

    @Override
    public void onFinish(ITestContext context) {
        ITestListener.super.onFinish(context);
        report.flush();
    }

}
```

Advanced test lifecycle – JUnit vs. TestNG

- Example #12: reporting test suite



Conclusions

- Both JUnit and TestNG provide a comprehensive programming model for developing advanced tests in Java
- Similar aspects in JUnit and TestNG
  - Basic test lifecycle
  - Categorizing and filtering tests
  - Ordering tests
  - Parallel test execution
- Strong aspects in JUnit:
  - Parameterized tests
  - Conditional test execution
  - Extension model
- Strong aspects in TestNG:
  - Test listeners

JUnit vs. TestNG:
Which Framework Fits Your Testing Strategy?

Thank you so much!

Get these slides at:

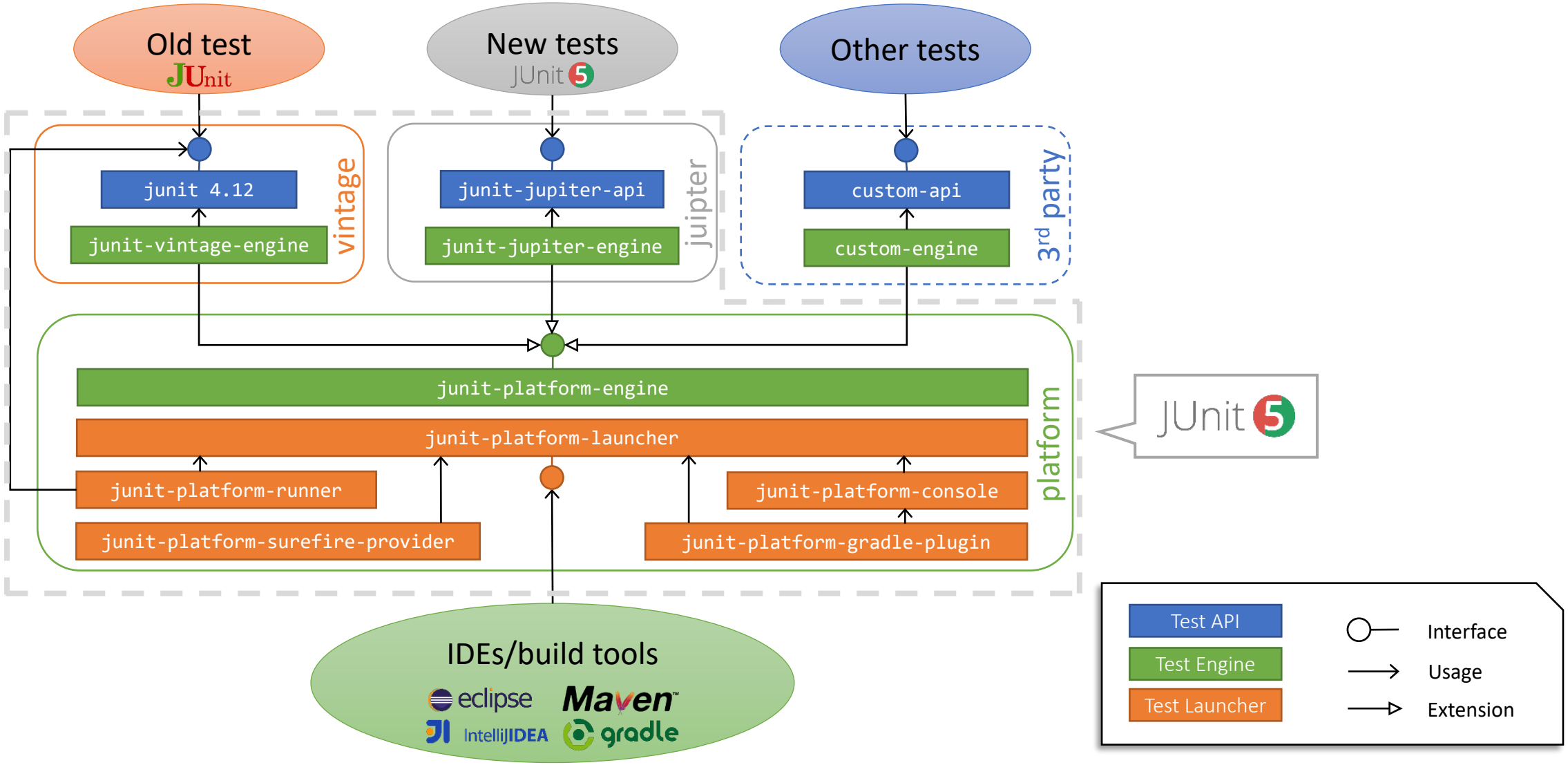
https://bonigarcia.dev/

Boni García
boni.garcia@uc3m.es

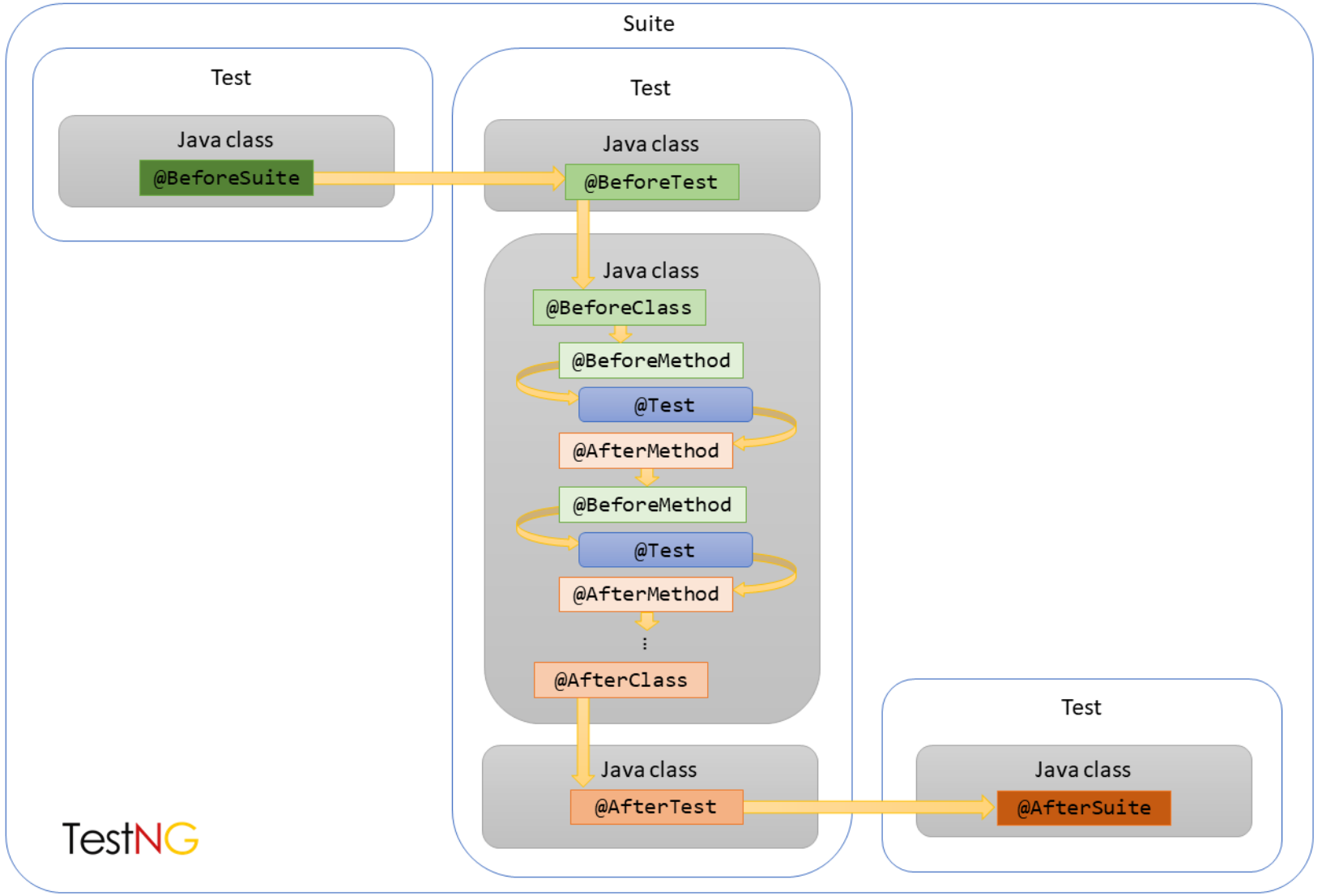
Read this story at:

https://medium.com/@boni.gg

JUnit Architecture



TestNG Lifecycle – Suites



Parameterized Tests – JUnit

```java
import static org.junit.jupiter.api.Assertions.assertNotNull;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

class ValueSourceParameterizedTest {

    @ParameterizedTest
    @ValueSource(strings = { "Hello", "World" })
    void testWithStrings(String param) {
        System.out.println("String parameter: " + param);
        assertNotNull(param);
    }

    @ParameterizedTest
    @ValueSource(ints = { 0, 1 })
    void testWithInts(int param) {
        System.out.println("int parameter: " + param);
        assertNotNull(param);
    }

    @ParameterizedTest
    @ValueSource(booleans = { true, false })
    void testWithBooleans(boolean param) {
        System.out.println("boolean parameter: " + param);
        assertNotNull(param);
    }

}
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.ValueSourceParameterizedTest
String parameter: Hello
String parameter: World
int parameter: 0
int parameter: 1
boolean parameter: true
boolean parameter: false
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.117 s -- in io.github.bonigarcia.ValueSourceParameterizedTest
```

Parameterized Tests – JUnit


```java
import static org.junit.jupiter.api.Assertions.assertNotNull;
import java.util.concurrent.TimeUnit;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.EnumSource;

class EnumSourceParameterizedTest {

    @ParameterizedTest
    @EnumSource(TimeUnit.class)
    void testWithEnum(TimeUnit param) {
        System.out.println("TimeUnit parameter: " + param);
        assertNotNull(param);
    }

}
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.EnumSourceParameterizedTest
TimeUnit parameter: NANOSECONDS
TimeUnit parameter: MICROSECONDS
TimeUnit parameter: MILLISECONDS
TimeUnit parameter: SECONDS
TimeUnit parameter: MINUTES
TimeUnit parameter: HOURS
TimeUnit parameter: DAYS
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.124 s -- in io.github.bonigarcia.EnumSourceParameterizedTest
```

Parameterized Tests – JUnit

```java
import static org.junit.jupiter.api.Assertions.assertNotEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import java.util.stream.Stream;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

class MethodSourceMixedTypesParameterizedTest {

    static Stream<Arguments> stringAndIntProvider() {
        return Stream.of(Arguments.of("Hello", 10), Arguments.of("World", 20));
    }

    @ParameterizedTest
    @MethodSource("stringAndIntProvider")
    void testWithMultiArgMethodSource(String first, int second) {
        System.out.println(String.format("String: %s -- int: %d", first, second));
        assertNotNull(first);
        assertNotEquals(0, second);
    }

}
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running
io.github.bonigarcia.MethodSourceMixedTypesParameterizedTest
String: Hello -- int: 10
String: World -- int: 20
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.106 s -- in
io.github.bonigarcia.MethodSourceMixedTypesParameterizedTest
```

Parameterized Tests – JUnit

```java
import static org.junit.jupiter.api.Assertions.assertFalse;
import java.util.List;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.FieldSource;

class FieldSourceParameterizedTest {

    static List<String> cities = List.of("Madrid", "Rome", "Paris", "London");

    @ParameterizedTest
    @FieldSource("cities")
    void testCities(String city) {
        System.out.println("Testing city: " + city);
        assertFalse(city.isBlank());
    }

}
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.FieldSourceParameterizedTest
Testing city: Madrid
Testing city: Rome
Testing city: Paris
Testing city: London
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.129 s -- in io.github.bonigarcia.FieldSourceParameterizedTest
```

Parameterized Tests – JUnit

```java
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ArgumentsSource;

class ArgumentSourceParameterizedTest {

    @ParameterizedTest
    @ArgumentsSource(CustomArgumentsProvider1.class)
    void testWithArgumentsSource(String first, int second) {
        System.out.println(String.format("String: %s -- int: %d", first, second));
        assertNotNull(first);
        assertTrue(second > 0);
    }

}
```

```java
import java.util.stream.Stream;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.ArgumentsProvider;

public class CustomArgumentsProvider1 implements ArgumentsProvider {

    @Override
    public Stream<? extends Arguments> provideArguments(
            ExtensionContext context) {
        return Stream.of(Arguments.of("hello", 1), Arguments.of("world", 2));
    }

}
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running
io.github.bonigarcia.ArgumentSourceParameterizedTest
String: hello -- int: 1
String: world -- int: 2
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.104 s -- in
io.github.bonigarcia.ArgumentSourceParameterizedTest
```

Parameterized Tests – JUnit

```java
import static org.junit.jupiter.api.Assertions.assertNotNull;
import java.util.stream.Stream;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.Parameter;
import org.junit.jupiter.params.ParameterizedClass;
import org.junit.jupiter.params.provider.MethodSource;

@ParameterizedClass
@MethodSource("myProvider")
class ParameterizedClassTest {

    @Parameter
    String param;

    static Stream<String> myProvider() {
        return Stream.of("hello", "world");
    }

    @Test
    void test() {
        System.out.println("String parameter: " + param);
        assertNotNull(param);
    }

}
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.ParameterizedClassTest
String parameter: hello
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.047 s -- in io.github.bonigarcia.ParameterizedClassTest
[INFO] Running io.github.bonigarcia.ParameterizedClassTest
String parameter: world
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.003 s -- in io.github.bonigarcia.ParameterizedClassTest
```

Parameterized Tests – TestNG

- There are two ways of implementing parameterized tests in TestNG:
  1. Using @DataProvider (most common and scalable)

```java
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class ParameterizedDataProviderNGTest {

    @DataProvider(name = "myData")
    public static Object[][] data() {
        return new Object[][] { { "hello", 10 }, { "world", 20 } };
    }

    @Test(dataProvider = "myData")
    public void testParameterized(String label, int amount) {
        System.out.println(
            "[DataProvider] Label: " + label + " -- Amount: " + amount);
    }
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running
io.github.bonigarcia.testng.parameterized.ParameterizedDataPro
viderNGTest
[DataProvider] Label: hello -- Amount: 10
[DataProvider] Label: world -- Amount: 20
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.631 s -- in
io.github.bonigarcia.testng.parameterized.ParameterizedDataPro
viderNGTest
```

Parameterized Tests – TestNG

2. Using @Parameters + testng.xml (dataset in <test></test>)

```java
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class ParameterizedXmlNGTest {

    @Test
    @Parameters({ "label", "amount" })
    public void test(String label, int amount) {
        System.out.println("[XML] Label: " + label + " -- Amount: " + amount);
    }

}
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running TestSuite
[XML] Label: hello -- Amount: 10
[XML] Label: world -- Amount: 20
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.772 s -- in TestSuite
```

```xml
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="TestSuite" verbose="1" parallel="false">
    <test name="ParameterizedTest-1">
        <parameter name="label" value="hello" />
        <parameter name="amount" value="10" />
        <classes>
            <class name="io.github.bonigarcia.testng.parameterized.ParameterizedXmlNGTest" />
        </classes>
    </test>
    <test name="ParameterizedTest-2">
        <parameter name="label" value="world" />
        <parameter name="amount" value="20" />
        <classes>
            <class name="io.github.bonigarcia.testng.parameterized.ParameterizedXmlNGTest" />
        </classes>
    </test>
</suite>
```