UNIT TESTING AND JUNIT

SOFTWARE QUALITY ASSURANCE AND TESTING

UNIT TESTING

UNIT TESTING

Purpose:

- **Validation:** Verify that each unit or component of the software performs as designed.
- **Isolation:** Test individual units in isolation to ensure their behavior independent of the rest of the system.

Characteristics:

- **Granularity:** Focus on testing the smallest testable parts of the application, usually individual functions or methods.
- **Automation:** Unit tests are typically automated and are run frequently during development.

Testing of an individual software unit

usually a class & its helpers

Focus on the functions of the unit

functionality, correctness, accuracy

Usually carried out by the developers of the unit

can use black-box and white-box techniques to design test cases

UNIT TESTING

Advantages:

- **Early Detection:** Identifies defects early in the development process.
- **Isolation of Issues:** Helps pinpoint the source of issues to specific units or components.
- **Documentation:** Serves as executable documentation for how units are expected to function.

Components:

- **Test Cases:** Individual scenarios or conditions that are tested.
- **Test Fixtures:** The initial setup and configuration needed for a set of test cases.
- **Assertions:** Statements that assert the expected behavior or outcomes of the unit under test.

Frameworks:

- **JUnit (Java):** A widely used testing framework for Java applications.
- **Pytest (Python):** A testing framework for Python applications.
- **Mocha (JavaScript):** A testing framework for JavaScript applications.
- **NUnit (.NET):** A testing framework for .NET applications.

UNIT TESTING

Unit testing: Looking for errors in a subsystem in isolation.

- "subsystem" means a particular class or object.
- The Java library JUnit helps us to easily perform unit testing.

The basic idea:

- For a given class Foo, create another class FooTest to test it, containing various "test case" methods to run.
- Each method looks for particular results and passes / fails.

JUnit provides "assert" commands to help us write tests.

- The idea: Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail.

JUNIT

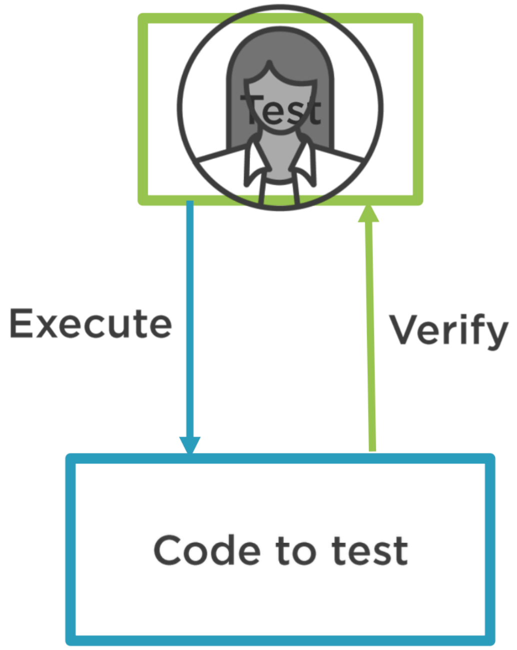
Test is just code

Executes target code

Class, method, or function

Check to see if the results are correct

Done manually before test libraries appeared



Benefits of Unit Testing

Quick feedback

Automated regression checking

Design aid

Improves confidence

Documentation

BENEFITS

BEST PRACTICES:

- **Isolation:** Test each unit in isolation to avoid dependencies on external factors.
- **Independence:** Ensure tests are independent of each other to facilitate parallel execution.
- **Repeatable:** Tests should produce the same results consistently when run multiple times.
- **Fast:** Unit tests should execute quickly to allow for frequent runs during development.
- Continuous Integration:
    - **Integration with CI/CD:** Incorporate unit tests into continuous integration and deployment pipelines.
    - **Automated Build:** Trigger automated builds and tests whenever changes are committed to version control.

JUNIT

WHY JUNIT

- **Automated Testing:**
  - JUnit facilitates the creation of automated tests, allowing for the quick and repeatable execution of test cases.
- **Test Framework:**
  - JUnit is a widely adopted testing framework for Java applications.
  - It provides annotations and assertions that simplify the process of writing and organizing tests.
- **Test Organization:**
  - JUnit supports the organisation of tests into test classes, making it easy to group related test cases.
  - Annotations like `@Test`, `@Before`, `@After`, `@BeforeClass`, and `@AfterClass` help define the test structure.
- **Test Execution:**
  - JUnit provides test runners and test engines for discovering and executing tests.
  - It supports parallel execution of tests, enabling faster feedback during development.
- **Assertion Methods:**
  - JUnit offers a rich set of assertion methods through the `Assertions` class, making it easy to check expected outcomes in tests.
  - Assertions help verify that the application behaves as expected.
- **Test Coverage:**
  - JUnit aids in achieving high test coverage by allowing developers to write comprehensive test suites.
  - Test coverage helps identify areas of code that are not exercised by tests.

WHY JUNIT

- **Continuous Integration:**
  - JUnit is commonly integrated into continuous integration (CI) systems such as Jenkins, Travis CI, and others.
  - Automated testing with JUnit is an essential part of the CI/CD (Continuous Integration/Continuous Deployment) pipeline.
- **Rapid Feedback:**
  - Automated tests with JUnit provide rapid feedback to developers by quickly identifying regressions or issues.
  - Fast feedback accelerates the development process and helps maintain code quality.
- **Regression Testing:**
  - JUnit is effective for regression testing, ensuring that new changes do not introduce unintended side effects or break existing functionality.
- **Ecosystem Support:**
  - JUnit has a strong and active community, and it is well-supported in the Java development ecosystem.
  - Many integrated development environments (IDEs) provide built-in support for running and debugging JUnit tests.
- **Extensibility:**
  - JUnit 5 introduces a flexible extension model, allowing developers to extend and customize the behavior of the testing framework.
  - Custom extensions can be used for various purposes, such as parameter resolution, lifecycle callbacks, and more.
- **Standardized Testing Practices:**
  - JUnit helps standardize testing practices across development teams, making it easier for team members to understand and contribute to the testing efforts.

JUNIT

- JUnit is a framework for writing unit tests
  - A unit test is a test of a single class
    - A test case is a single test of a single method
    - A test suite is a collection of test cases
- Unit testing is particularly important when software requirements change frequently
  - Code often has to be refactored to incorporate the changes
  - Unit testing helps ensure that the refactored code continues to work

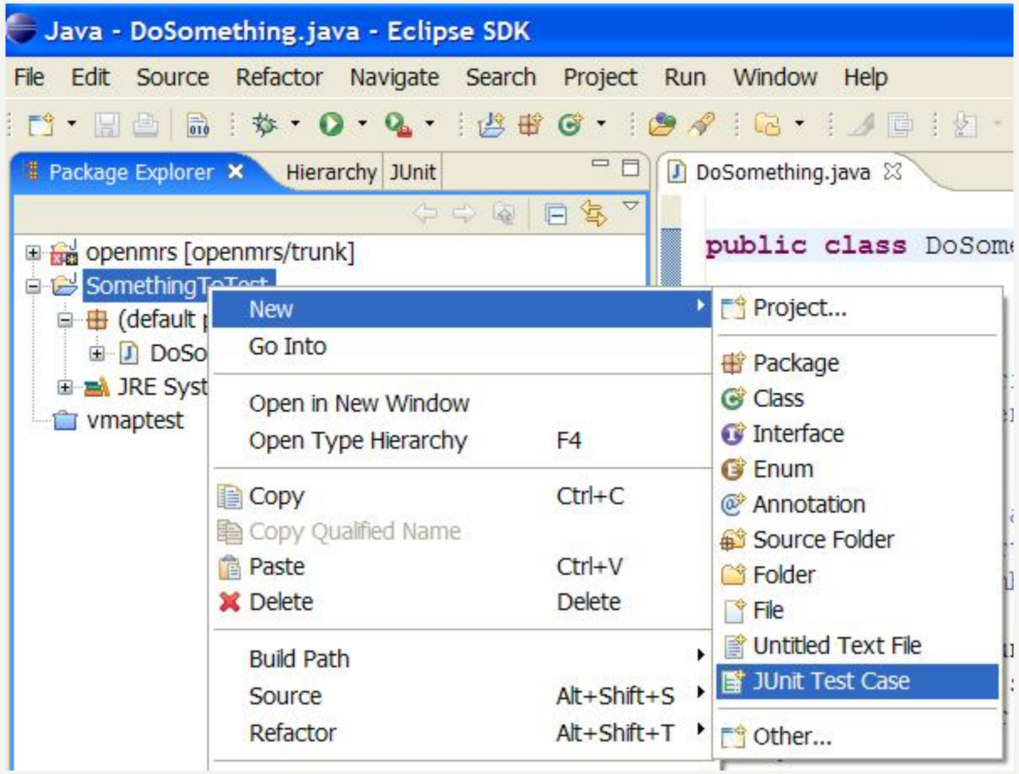
INSTALL

Sumit Ghosh

JUNIT AND ECLIPSE

- To add JUnit to an Eclipse project, click:
  - **Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 4 → Finish**

- To create a test case:
  - right-click a file and choose **New → Test Case**
  - or click **File → New → JUnit Test Case**
  - Eclipse can create stubs of method tests for you.



```
public class DoSome
```

JUNIT ARCHITECTURE

- Test Classes and Methods
- Annotations
- Test Runners
- JUnit Platform
- Test Engine
- Test Execution
- Extension Model
- Assertions
- Test Lifecycle

A JUNIT TEST CLASS

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

class name {
    ...

    @Test
        void name() {  // a test case method
        ...
    }
}
```

– A method with `@Test` is flagged as a JUnit test case.

- All `@Test` methods run when JUnit runs your test class.

THE STRUCTURE OF A TEST METHOD

- A test method doesn’t return a result
- If the tests run correctly, a test method does nothing
- If a test fails, it throws an `AssertionFailedError`
- The JUnit framework catches the error and deals with it; you don’t have to do anything

A SIMPLE JUNIT TEST CASE

```
/** Test of setName() method, of class Value  */
@Test
public void createAndSetName() {
    Value v1 = new Value();

    v1.setName("Y");

    String expected = "Y";
    String actual = v1.getName();

    Assert.assertEquals(expected, actual);
}
```

A SIMPLE JUNIT TEST CASE

Identify this Java method as a test case

```
/** Test of setName() method, c
@Test
public void createAndSetName() {
    Value v1 = new Value();

    v1.setName("Y");

    String expected = "Y";
    String actual = v1.getName();

    Assert.assertEquals(expected, actual);
}
```

A SIMPLE JUNIT TEST CASE

```
/** Test of setName() method, of class Value  */
@Test
public void createAndSetName() {
    Value v1 = new Value();

    v1.setName("Y");

    String expected = "Y";
    String actual = v1.getName();

    Assert.assertEquals(expected, actual);
}
```

Confirm that setName saves the specified name in the Value object

A SIMPLE JUNIT TEST CASE

```
/** Test of setName() method, of cla
@Test
public void createAndSetName() {
    Value v1 = new Value();

    v1.setName("Y");

    String expected = "Y";
    String actual = v1.getName();

    Assert.assertEquals(expected, actual);
}
```

Check to see that the Value object really did store the name

A SIMPLE JUNIT TEST CASE

```
/** Test of setName() method, of class Value  */
@Test
public void createAndSetName() {
    Value v1 = new Value();

    v1.setName("Y");

    String expected = "Y";
    String actual = v1.getName();

    Assert.assertEquals(expected, actual);
}
```

Assert that the expected and actual should be equal. If not, the test case should fail.

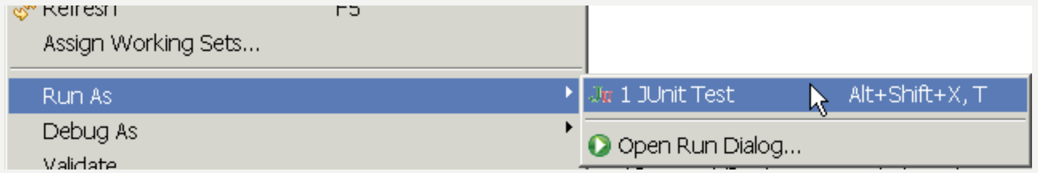
RUNNING A TEST

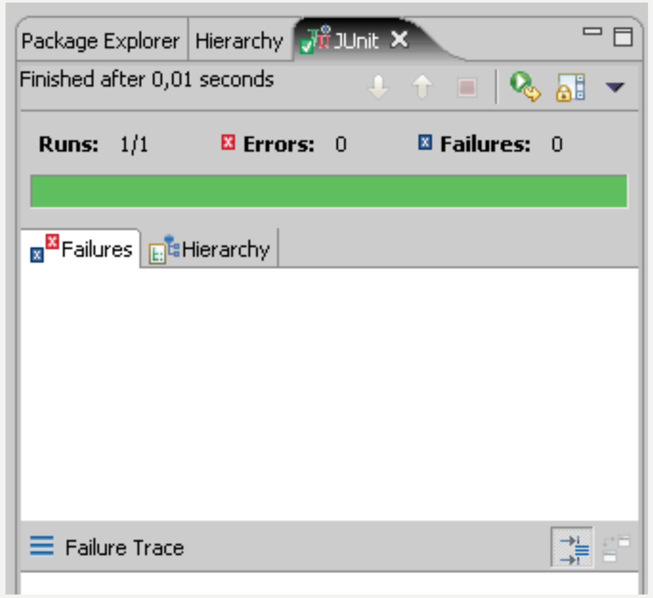
- Right click it in the Eclipse Package Explorer at left; choose:

  Run As → JUnit Test

- The JUnit bar will show **green** if all tests pass, **red** if any fail.

- The Failure Trace shows which tests failed, if any, and why.





Finished after 0,01 seconds

Runs: 1/1 Errors: 0 Failures: 0

Sumit Ghosh

ASSERTION METHODS

ASSERTIONS IN TEST CASES

During execution of a test case:

- If an assertion is **true**,
  - Execution continues
- If any assertion is **false**,
  - Execution of the test case stops
  - The test case fails
- If an *unexpected* exception is encountered,
  - The verdict of the test case is an error.
- If all assertions were true,
  - The test case *passes*.

ASSERTION METHODS: BOOLEAN CONDITIONS

- Static methods defined in org.junit.Assert
- Assert a Boolean condition is true or false

  assertTrue(*condition*)

  assertFalse(*condition*)
- Optionally, include a failure message

  assertTrue(*condition*, *message*)
  assertFalse(*condition*, *message*)
- Examples

  assertTrue(search(a, 3) == 1);

  assertFalse(search(a, 2) >= 0,“Failure: 2 is not in array.”);

ASSERTION METHODS: NULL OBJECTS

- Assert an object references is null or non-null

  assertNull(*object*)

  assertNotNull(*object*)

- With a failure message

  assertNull(*object, message*)

  assertNotNull(*object, message*)

- Examples

```
assertNotNull(new Object(),”Should not be null.");
assertNull(null,”Should be null.");
```

ASSERTION METHODS: OBJECT IDENTITY

- Assert two object references are identical

  assertSame(*expected, actual*)

  – True if: expected == actual

  assertNotSame(*expected, actual*)

  – True if: expected != actual

- The order does not affect the comparison,
  - But, affects the message when it fails

- With a failure message

  assertSame(*expected, actual, message*)

  assertNotSame(*expected, actual, message*)

ASSERTION METHODS: OBJECT IDENTITY

- Examples

```
assertNotSame(new Object(), new Object(),"Should not be same.");

Integer num1 = Integer.valueOf(2013);
assertSame(num1, num1,"Should be same.”);

Integer num2 = Integer.valueOf(2014);
assertSame(num1, num2, "Should be same.");
```

```
java.lang.AssertionError:
Should be same. expected same:<2013> was not:<2014>
```

ASSERTION METHODS: OBJECT EQUALITY

- Assert two objects are equal:

  assertEquals(expected, actual)

  - True if: expected.equals(actual)
  - Relies on the equals() method
  - Up to the class under test to define a suitable equals() method.

- With a failure message

  assertEquals(*expected, actual,message*)

ASSERTION METHODS: OBJECT EQUALITY

- Examples

```
assertEquals("JUnit", “JUnit”, "Should be equal.");

assertEquals("JUnit", “Java”, "Should be equal.");
```

```
org.junit.ComparisonFailure:
Should be equal. expected:<J[Unit]> but was:<J[ava]>
```

ASSERTION METHODS: EQUALITY OF ARRAYS

- Assert two arrays are equal:

  assertArrayEquals(expected, actual)

  - arrays must have same length
  - Recursively check for each valid index i,

    assertEquals(expected[i],actual[i])

    or

    assertArrayEquals(expected,actual)

- With a failure message

  assertArrayEquals(*expected, actual,message*)

ASSERTION METHODS: EQUALITY OF ARRAYS

- Examples

```
int[] a1 = { 2, 3, 5, 7 };
int[] a2 = { 2, 3, 5, 7 };
assertArrayEquals(a1, a2, "Should be equal");

int[][] a11 = { { 2, 3 }, { 5, 7 }, { 11, 13 } };
int[][] a12 = { { 2, 3 }, { 5, 7 }, { 11, 13 } };
assertArrayEquals(a11, a12,"Should be equal");
```

ASSERTION METHODS: FLOATING POINT VALUES

- For comparing floating point values (double or float)
  - assertEquals requires an additional parameter **delta**.

  assertEquals(*expected, actual, delta*)

  assertEquals(*expected, actual, delta, message*)
- The assertion evaluates to true if

  Math.abs(expected – actual) <= delta
- Example:

```
double d1 = 100.0, d2 = 99.99995;
assertEquals(d1, d2, 0.0001,“Should be equal within delta.”);
```

EXCEPTION TESTING

- A.k.a., robustness testing
- The expected outcome of a test is an exception.

```
public static int checkedSearch(int[] a, int x) {
  if (a == null || a.length == 0)
    throw
    new IllegalArgumentException("Null or empty array.");
  …
}
```

```
checkedSearch(null, 1);
```

EXCEPTION TESTING: SPECIFY THE EXCEPTED EXCEPTION

- Specify an expected exception in a test case
  - A particular class of exception is expected to occur

```
@Test
   void exceptionTesting() {
       Exception exception = assertThrows(ArithmeticException.class, () ->
     calculator.divide(1, 0));
       assertEquals("/ by zero", exception.getMessage());
   }
```

- The verdict
  - Pass: if the expected exception is thrown
  - Fail: if no exception, or an unexpected exception

EXCEPTION TESTING: THE FAIL ASSERTION

- Assertion methods
  - fail()
  - fail(*message*)
- Unconditional failure
  - i.e., it always fails if it is executed
- Used in where it should not be reached
  - e.g., after a statement, in which an exception should have been thrown.

EXCEPTION TESTING: USE FAIL() ASSERTION

- Catch exceptions, and use fail() if not thrown

```
@Test
public void testCheckedSearch3() {
  try {
       checkedSearch(null, 1);
       fail("Exception should have occurred");
  } catch (IllegalArgumentException e) {
       assertEquals(e.getMessage(), "Null or empty array.");
  }
}
```

- Allows
  - inspecting specific messages/details of the exception
  - distinguishing different types of exceptions

JUNIT ASSERTION METHODS

`assertTrue(`**test**`)`	fails if the boolean test is `false`
`assertFalse(`**test**`)`	fails if the boolean test is `true`
`assertEquals(`**expected**`, `**actual**`)`	fails if the values are not equal
`assertSame(`**expected**`, `**actual**`)`	fails if the values are not the same (by ==)
`assertNotSame(`**expected**`, `**actual**`)`	fails if the values *are* the same (by ==)
`assertNull(`**value**`)`	fails if the given value is *not* `null`
`assertNotNull(`**value**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:
  - e.g. `assertEquals("`**message**`", `**expected**`, `**actual**`)`
  - Why is there no `pass` method?

ARRAYINTLIST JUNIT TEST

```
import org.junit.*;
import static org.junit.Assert.*;

public class TestArrayIntList {
    @Test
    public void testAddGet1() {
        ArrayIntList list = new ArrayIntList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));
    }

    @Test
    public void testIsEmpty() {
        ArrayIntList list = new ArrayIntList();
        assertTrue(list.isEmpty());
        list.add(123);
        assertFalse(list.isEmpty());
        list.remove(0);
        assertTrue(list.isEmpty());
    }
    ...
```

Use various assertion methods provided by JUnit 5 to validate the expected outcomes.

```
public class StringUtils {

    public String reverse(String input) {
        // Reverses the characters in the input string
        // Example: "hello" -> "olleh"
        return new StringBuilder(input).reverse().toString();
    }

    public boolean isPalindrome(String input) {
        // Checks if the input string is a palindrome
        // Example: "radar" -> true, "hello" -> false
        String reversed = reverse(input);
        return input.equals(reversed);
    }

    public String concatenate(String str1, String str2) {
        // Concatenates two strings
        // Example: concatenate("Hello", "World") -> "HelloWorld"
        return str1 + str2;
    }

    public int countOccurrences(String input, char target) {
        // Counts the occurrences of a specific character in the input string
        // Example: countOccurrences("hello", 'l') -> 2
        int count = 0;
        for (char c : input.toCharArray()) {
            if (c == target) {
                count++;
            }
        }
        return count;
    }
}
```

Use various assertion methods provided by JUnit 5 to validate the expected outcomes.

```
public class EmailValidator {

    public boolean isValidEmail(String email) {
        // Validates whether the provided email address is valid
        // Example: isValidEmail("user@example.com") -> true
        //          isValidEmail("invalid_email") -> false
        // Implement a simple validation logic for demonstration purposes
        return email != null && email.matches("[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}");
    }

    public boolean isCorporateEmail(String email) {
        // Checks if the provided email address is a corporate email (ends with a specific domain)
        // Example: isCorporateEmail("user@example.com") -> false
        //          isCorporateEmail("employee@company.com") -> true
        return email != null && email.endsWith("company.com");
    }
}
```

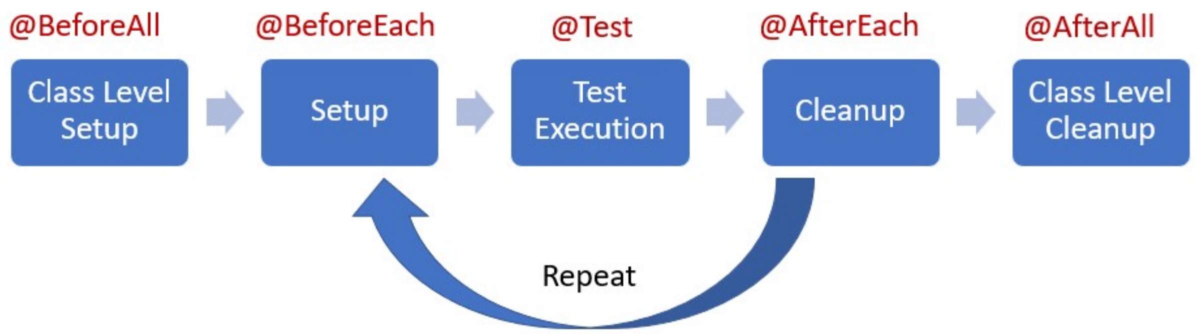
Use various assertion methods provided by JUnit 5 to validate the expected outcomes.

```
public class Book {

    private String title;
    private String author;
    private boolean available;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
        this.available = true;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public boolean isAvailable() {
        return available;
    }

    public void setAvailable(boolean available) {
        this.available = available;
    }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class LibraryCatalog {

    private List<Book> books;

    public LibraryCatalog() {
        this.books = new ArrayList<>();
    }

    public void addBook(Book book) {
        books.add(book);
    }

    public List<Book> getBooks() {
        return new ArrayList<>(books);
    }

    public void borrowBook(String title) {
        for (Book book : books) {
            if (book.getTitle().equals(title) && book.isAvailable()) {
                book.setAvailable(false);
                return;
            }
        }
    }

    public void returnBook(String title) {
        for (Book book : books) {
            if (book.getTitle().equals(title) && !book.isAvailable()) {
                book.setAvailable(true);
                return;
            }
        }
    }
}
```

JUNIT LIFE CYCLE

LIFE CYCLE



Setup (@BeforeEach):

- The method annotated with `@BeforeEach` is executed before each test method.It serves the same purpose as the `@Before` method in JUnit 4.

Teardown (@AfterEach):

- The method annotated with `@AfterEach` is executed after each test method.
- It serves the same purpose as the `@After` method in JUnit 4.

Class Cleanup (@AfterAll):

- The method annotated with `@AfterAll` is executed once after all test methods have run.
- It serves the same purpose as the `@AfterClass` method in JUnit

`@BeforeAll`

The method executed once before all test methods in a test class and typically used for setup operations that need to be performed once for the entire test class.

RUNNING ALL TOGETHER

```java
public class AppTest {

    @BeforeAll
    static void setup(){
        System.out.println("@BeforeAll executed");
    }

    @BeforeEach
    void setupThis(){
        System.out.println("@BeforeEach executed");
    }

    @Test
    void testCalcOne()
    {
        System.out.println("======TEST ONE EXECUTED=======");
        Assertions.assertEquals( 4 , Calculator.add(2, 2));
    }

    @Test
    void testCalcTwo()
   {
        System.out.println("======TEST TWO EXECUTED=======");
        Assertions.assertEquals( 6 , Calculator.add(2, 4));
    }

    @AfterEach
    void tearThis(){
        System.out.println("@AfterEach executed");
    }

    @AfterAll
    static void tear(){
        System.out.println("@AfterAll executed");
    }
}
```

TEST SUITE

TEST SUITES

- In practice, you want to run a group of related tests (e.g. all the tests for a class)
- @SelectPackages
- @SelectClasses
- @ExcludePackages
- @IncludeTags
- @ExcludeTags

MAVEN DEPENDENCY

```xml
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-suite</artifactId>
    <version>1.10.1</version>
    <scope>test</scope>
</dependency>
```

TEST SUITES

- **Test suite**: One class that runs many JUnit tests.
  - An easy way to run all of your app's tests at once.

```
@Suite
@SelectClasses({DivisionDemoTest.class,
        MultiplicationDemoTest.class})
public class MyTestSuite_class
{
}
```

TEST SUITE EXAMPLE

```java
package test_division.test_suite;


import org.junit.platform.suite.api.SelectPackages;
import org.junit.platform.suite.api.Suite;

@Suite
@SelectPackages("org.calculator.demos.multiplication")
public class MyTestSuite_package
{
}
```

OTHER FEATURES

CONDITIONAL TEST

ARCH

OS

JRE

CUSTOM

DISABLE TEST

ANNOTATION:
@DISABLED

ORDER OF EXECUTION

- Numerical Order
- Method Name

PARAMETER

- String & Integer
- From CSV

REPEAT

REPEATING TEST CASES

JUNIT BEST PRACTICES

Each test case should be independent.

Test cases should be independent of execution order.

No dependencies on the state of previous tests.

JUNIT TEST FIXTURES

- The context in which a test case is executed.
- Typically include:
  - Common objects or resources that are available for use by any test case.
- Activities to manage these objects
  - Set-up: object and resource allocation
  - Tear-down: object and resource de-allocation

CODE COVERAGE & JACOCO

CODE COVERAGE

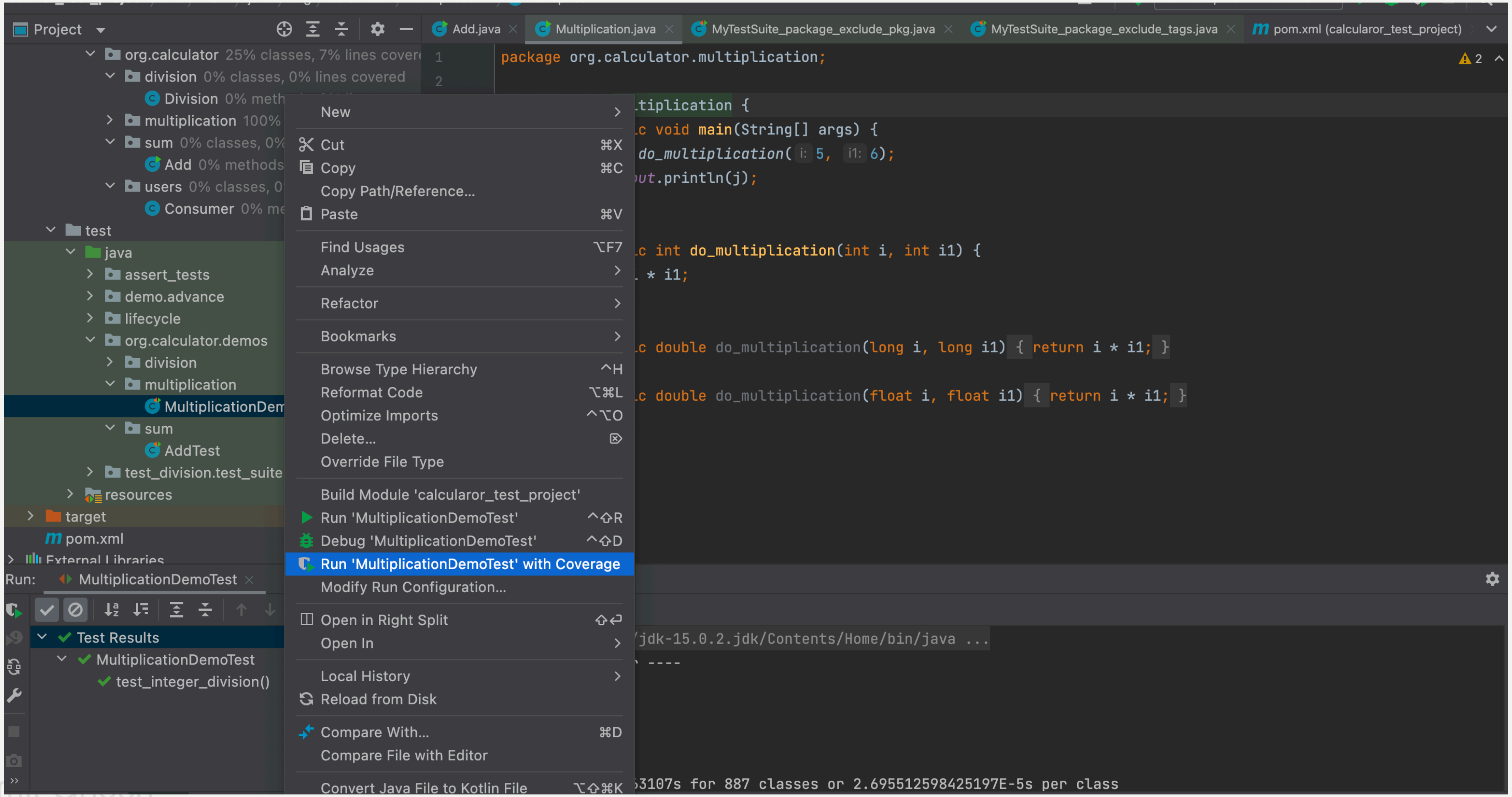
Code coverage is a software metric used to measure how many lines of our code are executed during automated tests.

Statement Coverage

Branch Coverage

Function Coverage

IN-BUILD COVERAGE TOOL



```
package org.calculator.multiplication;

tiplication {
c void main(String[] args) {
 do_multiplication( i: 5,  i1: 6);
ut.println(j);

c int do_multiplication(int i, int i1) {
 * i1;

c double do_multiplication(long i, long i1) { return i * i1; }

c double do_multiplication(float i, float i1) { return i * i1; }
```

```
/jdk-15.0.2.jdk/Contents/Home/bin/java ...
 ----

3107s for 887 classes or 2.695512598425197E-5s per class
```

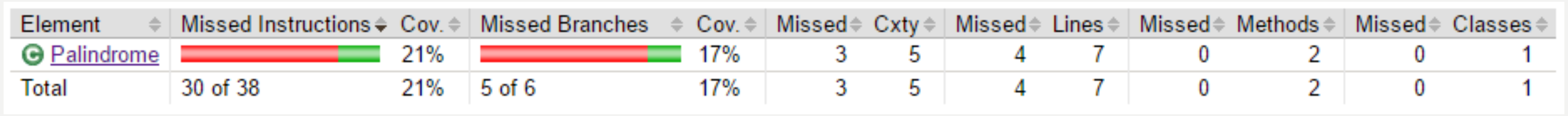
JACOCO

- **Lines coverage** reflects the amount of code that has been exercised based on the number of Java byte code instructions called by the tests.
- **Branches coverage** shows the percent of exercised branches in the code, typically related to *if/else* and *switch* statements.
- **Cyclomatic complexity** reflects the complexity of code by giving the number of paths needed to cover all the possible paths in a code through linear combination.

MAVEN CONFIGURATION

```xml
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.7.7.201606060606</version>
    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

REPORT



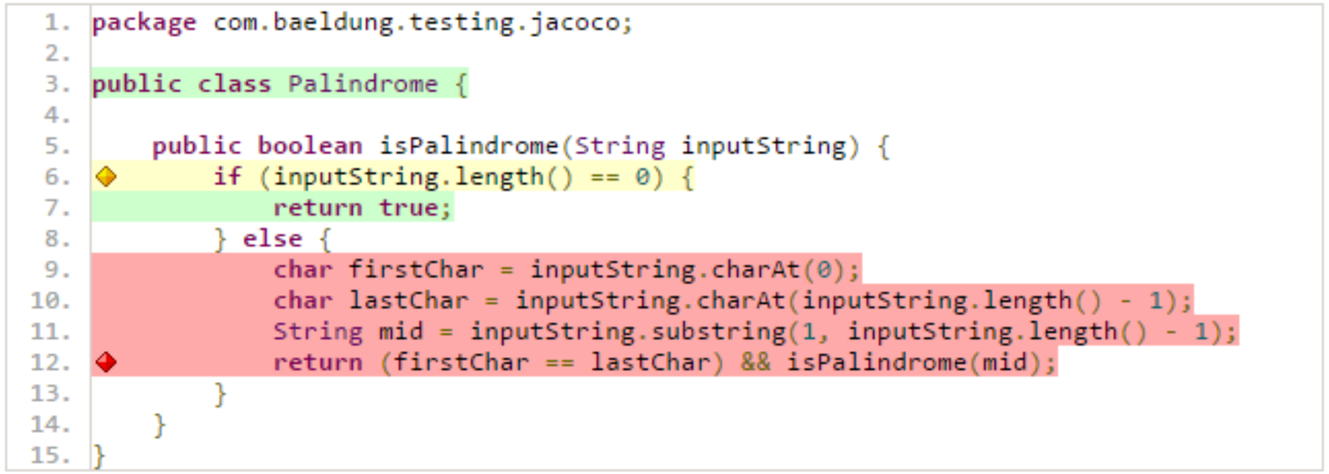
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Palindrome	🟥🟩	21%	🟥🟩	17%	3	5	4	7	0	2	0	1
Total	30 of 38	21%	5 of 6	17%	3	5	4	7	0	2	0	1

REPORT ANALYSIS

•**Red diamond** means that no branches have been exercised during the test phase.

•**Yellow diamond** shows that the code is partially covered – some branches have not been exercised.

•**Green diamond** means that all branches have been exercised during the test.



```
package com.baeldung.testing.jacoco;

public class Palindrome {

    public boolean isPalindrome(String inputString) {
        if (inputString.length() == 0) {
            return true;
        } else {
            char firstChar = inputString.charAt(0);
            char lastChar = inputString.charAt(inputString.length() - 1);
            String mid = inputString.substring(1, inputString.length() - 1);
            return (firstChar == lastChar) && isPalindrome(mid);
        }
    }
}
```

SETUP COVERAGE THRESHOLDS

Define the code coverage thresholds for different categories (e.g., overall coverage, class coverage, method coverage) in your build configuration.

This example sets a minimum coverage ratio of 80% for all packages.

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.10</version>
    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>test</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
        <execution>
            <id>jacoco-check</id>
            <goals>
                <goal>check</goal>
            </goals>
            <configuration>
                <rules>
                    <rule>
                        <element>PACKAGE</element>
                        <limits>
                            <limit>
                                <counter>LINE</counter>
                                <value>COVEREDRATIO</value>
                                <minimum>0.8</minimum>
                            </limit>
                        </limits>
                    </rule>
                </rules>
            </configuration>
        </execution>
    </executions>
</plugin>
```

FAIL THE BUILD ON VIOLATION

In Maven configure the jacoco-maven-plugin to fail the build on coverage violations

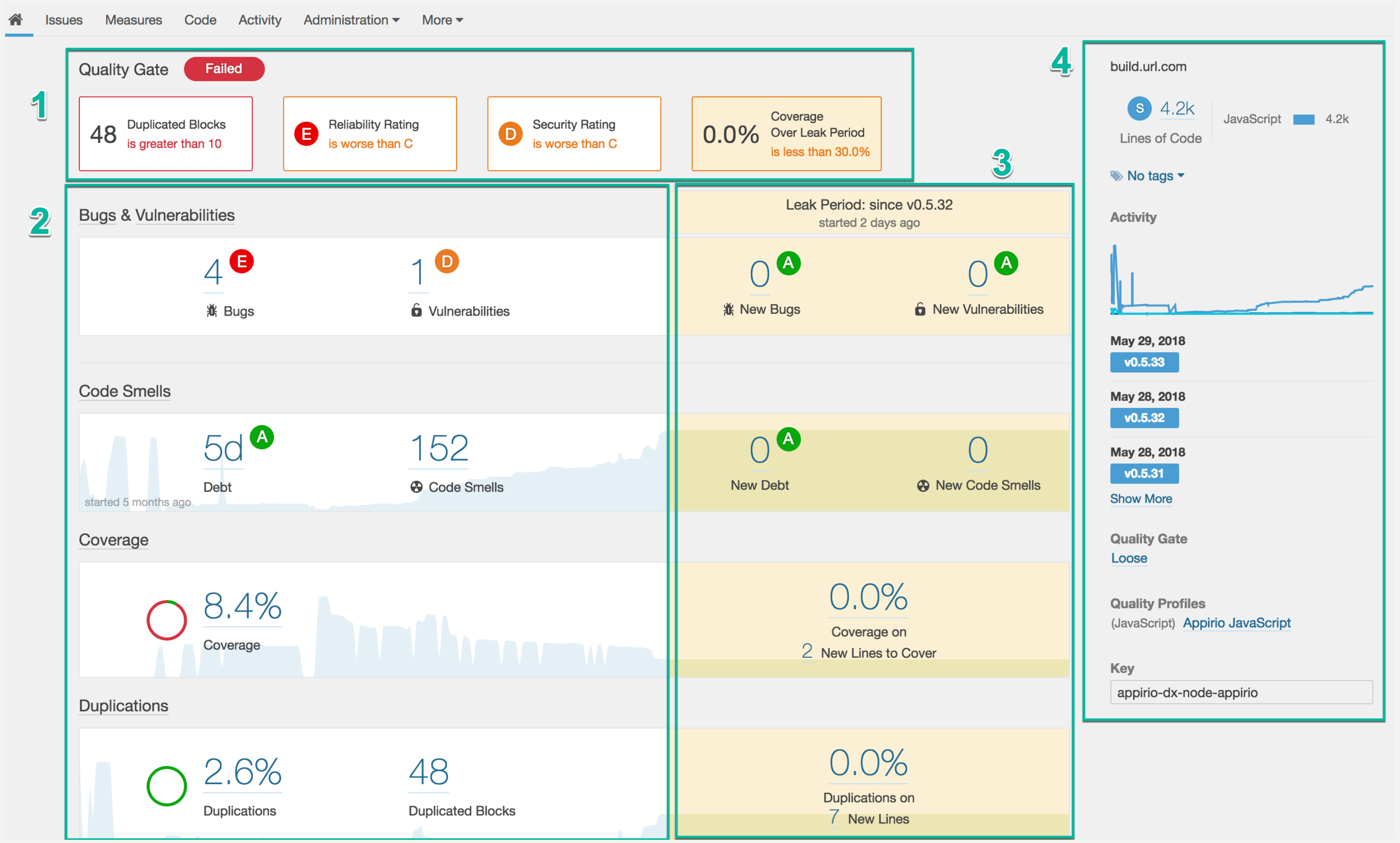
```
<build>
    <plugins>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.8.7</version>
            <executions>
                <execution>
                    <id>check</id>
                    <goals>
                        <goal>check</goal>
                    </goals>
                    <configuration>
                        <!-- other configurations -->
                        <haltOnFailure>true</haltOnFailure>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

SONAR QUBE

SonarQube is an open-source platform for continuous inspection of code quality.

It provides a set of tools for code quality management, static code analysis, and continuous inspection of code to identify and fix issues early in the development process.

SonarQube is widely used in software development teams to improve code quality, reduce technical debt, and enhance overall software maintainability.



Quality Gate Failed

48 Duplicated Blocks is greater than 10
E Reliability Rating is worse than C
D Security Rating is worse than C
0.0% Coverage Over Leak Period is less than 30.0%

Bugs & Vulnerabilities
4 E Bugs
1 D Vulnerabilities

Code Smells
5d A Debt
152 Code Smells
started 5 months ago

Coverage
8.4% Coverage

Duplications
2.6% Duplications
48 Duplicated Blocks

Leak Period: since v0.5.32
started 2 days ago
0 A New Bugs
0 A New Vulnerabilities
0 A New Debt
0 New Code Smells
0.0% Coverage on 2 New Lines to Cover
0.0% Duplications on 7 New Lines

build.url.com
4.2k Lines of Code
JavaScript 4.2k
No tags
Activity
May 29, 2018 v0.5.33
May 28, 2018 v0.5.32
May 28, 2018 v0.5.31
Show More
Quality Gate
Loose
Quality Profiles
(JavaScript) Appirio JavaScript
Key
appirio-dx-node-appirio

OUTLINE

- Unit Testing and JUnit
- Assertion Methods
- JUnit Best Practices

- **JUnit documentation**
  - http://junit.org/junit5
  - https://junit.org/junit5/docs/snapshot/user-guide/
- **An introductory tutorial**
  - http://www.vogella.com/tutorials/JUnit/article.html
- **Using JUnit in Eclipse**
  - https://www.eclipse.org/community/eclipse_newsletter/2017/october/article5.php
  - https://www.educative.io/courses/java-unit-testing-with-junit-5/B892KY261z2
- **How to use JUnit with NetBeans**
  - https://testingandlearning.home.blog/2019/01/30/how-to-use-junit-with-netbeans/