**Unit test for class called ErrorReport which implements the Comparable interface.**

package com.ch794.marker.configuration;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class ErrorReportTest {

@Test

public void testGetScore() {

ErrorReport errorReport1 = new ErrorReport("file1", 1, 1, "violation1", "type1");

ErrorReport errorReport2 = new ErrorReport("file2", 2, 2, "violation2", "type2");

ErrorReport errorReport3 = new ErrorReport("file3", 3, 3, "violation3", "type3");

// Set up the violation scoring for the Marker.currentMarkingConfiguration object

Marker.currentMarkingConfiguration.violationScoring.put("type1", 10);

Marker.currentMarkingConfiguration.violationScoring.put("type2", 5);

// Test getScore() method

assertEquals(10, errorReport1.getScore());

assertEquals(5, errorReport2.getScore());

assertEquals(1, errorReport3.getScore()); // Default score when violation type is not found in configuration

}

@Test

public void testCompareTo() {

ErrorReport errorReport1 = new ErrorReport("file1", 1, 2, "violation1", "type1");

ErrorReport errorReport2 = new ErrorReport("file1", 1, 1, "violation2", "type2");

ErrorReport errorReport3 = new ErrorReport("file2", 1, 1, "violation3", "type3");

// Test compareTo() method

assertEquals(-1, errorReport1.compareTo(errorReport2)); // Expected result: errorReport1 is less than errorReport2

assertEquals(1, errorReport2.compareTo(errorReport1)); // Expected result: errorReport2 is greater than errorReport1

assertEquals(0, errorReport2.compareTo(errorReport2)); // Expected result: errorReport2 is equal to errorReport2

assertEquals(-1, errorReport2.compareTo(errorReport3)); // Expected result: errorReport2 is less than errorReport3

}

}

In this unit test, the getScore() and compareTo() are tested. Three instances are created with different values for the instance vaiables. In the testGetScore() method, the violationScoring configuration is set for the Marker.currentMarkingConfiguration object and tests if the getScore() method reurnns the expected scores for each ErrorReport instance. The testCompareTo() method, the compareTo() method is tested by comparing the three ErrorReport instances. It checks to see if the method returns the exected results for each comparison.

**Java Unit Test for the FileErrorReport class**

package com.ch794.marker.configuration;

import org.junit.Test;

import static org.junit.Assert.\*;

public class FileErrorReportTest {

@Test

public void testAddReport() {

FileErrorReport report = new FileErrorReport("test-file");

ErrorReport error1 = new ErrorReport("test-file", 1, 1, "violation1", "type1");

report.addReport(error1);

ErrorReport error2 = new ErrorReport("test-file", 2, 2, "violation2", "type2");

report.addReport(error2);

assertEquals(2, report.getReports().size());

assertEquals("test-file", report.getFile());

assertEquals(error1, report.getReports().get(0));

assertEquals(error2, report.getReports().get(1));

}

@Test

public void testGetScore() {

FileErrorReport report = new FileErrorReport("test-file");

ErrorReport error1 = new ErrorReport("test-file", 1, 1, "violation1", "type1");

report.addReport(error1);

ErrorReport error2 = new ErrorReport("test-file", 2, 2, "violation2", "type2");

report.addReport(error2);

assertEquals(2, report.getScore());

}

@Test

public void testGetPrettyPrint() {

FileErrorReport report = new FileErrorReport("test-file");

ErrorReport error1 = new ErrorReport("test-file", 1, 1, "violation1", "type1");

report.addReport(error1);

ErrorReport error2 = new ErrorReport("test-file", 2, 2, "violation2", "type2");

report.addReport(error2);

String expectedOutput = "Results for test-file: Score (2)\n" +

"test-file 1:1 (+1) [type1] violation1\n" +

"test-file 2:2 (+1) [type2] violation2\n" +

"\n";

assertEquals(expectedOutput, report.getPrettyPrint());

}

}

This unit test has 3 methods

* testAddReport() tests that FIleErrorReport can correctly add new ErrorReport objects to its list of reports.
* testGetScore() tests that FileErrorReport can correctly calculate the score of its reports.
* testGetPrettyPrint() tests that FileErrorReport can correctly generate a pretty-printed summary of its reports.

**Unit Test for the MarkingConfiguration class**

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

import java.io.ByteArrayOutputStream;

import java.io.PrintStream;

import java.util.Arrays;

import java.util.List;

import java.util.Map;

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

class MarkingConfigurationTest {

private final ByteArrayOutputStream outContent = new ByteArrayOutputStream();

private final ByteArrayOutputStream errContent = new ByteArrayOutputStream();

private final PrintStream originalOut = System.out;

private final PrintStream originalErr = System.err;

private MarkingConfiguration configuration;

@BeforeEach

void setUp() {

// Redirect standard output and error to streams for testing

System.setOut(new PrintStream(outContent));

System.setErr(new PrintStream(errContent));

// Create a new configuration with some sample values for testing

configuration = new MarkingConfiguration(

"test",

".\\tests\\examples\\%username%\\hello",

".\\config.XML",

".\\config\\template.txt",

Arrays.asList("blacklist1.java", "blacklist2.java"),

Map.of("violation1", 3, "violation2", 1)

);

}

@AfterEach

void tearDown() {

// Restore standard output and error

System.setOut(originalOut);

System.setErr(originalErr);

}

@Test

void testGetPlaceholderCollection() {

PlaceholderCollection collection1 = configuration.getPlaceholderCollection("user1");

PlaceholderCollection collection2 = configuration.getPlaceholderCollection("user2");

assertNotNull(collection1);

assertNotNull(collection2);

assertNotSame(collection1, collection2);

}

@Test

void testGetUserErrorReport() {

UserErrorReport report1 = configuration.getUserErrorReport("user1");

UserErrorReport report2 = configuration.getUserErrorReport("user2");

assertNotNull(report1);

assertNotNull(report2);

assertNotSame(report1, report2);

}

@Test

void testReportAuditEvent() {

AuditEvent event = new AuditEvent("file.java", "violation", null, null, null);

// Report the event and verify that the corresponding error report was created

configuration.currentUsername = "user1";

configuration.reportAuditEvent(event);

assertTrue(configuration.errorReports.containsKey("user1"));

assertEquals(1, configuration.errorReports.get("user1").getErrorReports().size());

assertEquals("file.java", configuration.errorReports.get("user1").getErrorReports().get(0).getFileName());

assertEquals("violation", configuration.errorReports.get("user1").getErrorReports().get(0).getViolation());

assertEquals(null, configuration.errorReports.get("user1").getErrorReports().get(0).getLine());

assertEquals(null, configuration.errorReports.get("user1").getErrorReports().get(0).getColumn());

assertEquals(null, configuration.errorReports.get("user1").getErrorReports().get(0).getKey());

}

@Test

void testGetUsernames() {

// Create some test directories and files

new File(".\\tests\\examples\\user1\\hello\\test.java").getParentFile().mkdirs();

new File(".\\tests\\examples\\user2\\hello\\test.java").getParentFile().mkdirs();

new File(".\\tests\\examples\\user1\\hello\\test.java").mkdirs();

new File(".\\tests\\examples\\user2\\hello\\test.java").mkdirs();

List<String> usernames = configuration.getUsernames();

assertEquals(2, usernames.size());

assertTrue(usernames.contains("user1

Inside the testMyMethod() method, there are a few lines of code:

MyClass myClass = new MyClass(); - This creates an instance of the MyClass class that we want to test.

int result = myClass.myMethod(2 , 3); - This calls the myMethod() method of the MyClass instance with argguments 2 and 3 and stores the result in an interger variable called result.

assertEquals(5, result); - This is an assertion that checks whether the value of the result variable is equal to 5. If the assertion fails, the test will fail. To summarise this unit test is testing the myMethod() method of the MyClass class by creating an instance of the class, calling the method with some arguments, and then checking whether the results is what is expected. If the result is not what is expected, the test will fail.

**Unit Test for UserErrorReport class**

import com.ch794.marker.configuration.\*;

import com.ch794.marker.enums.\*;

import com.ch794.marker.placeholder.\*;

import com.ch794.marker.placeholder.types.\*;

import org.junit.Test;

import static org.junit.Assert.\*;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class UserErrorReportTest {

@Test

public void testUserErrorReport() {

UserErrorReport userReport = new UserErrorReport("testUser");

FileErrorReport fileReport1 = new FileErrorReport("file1.txt");

ErrorReport errorReport1 = new ErrorReport("error1", 10);

ErrorReport errorReport2 = new ErrorReport("error2", 20);

fileReport1.addReport(errorReport1);

fileReport1.addReport(errorReport2);

userReport.addErrorReport(fileReport1);

FileErrorReport fileReport2 = new FileErrorReport("file2.txt");

ErrorReport errorReport3 = new ErrorReport("error3", 30);

fileReport2.addReport(errorReport3);

userReport.addErrorReport(fileReport2);

assertEquals(60, userReport.getScore());

assertEquals("testUser", userReport.getUsername());

Map<String, FileErrorReport> reports = userReport.getReports();

assertEquals(2, reports.size());

assertTrue(reports.containsKey("file1.txt"));

assertTrue(reports.containsKey("file2.txt"));

assertEquals(fileReport1, reports.get("file1.txt"));

assertEquals(fileReport2, reports.get("file2.txt"));

List<String> fileInfo = userReport.getDetailedFileInfo(EnumSortMode.ASC);

assertEquals(2, fileInfo.size());

assertEquals("file1.txt (30 Points)", fileInfo.get(0));

assertEquals("file2.txt (30 Points)", fileInfo.get(1));

List<String> infractionInfo = userReport.getDetailedInfractionInfo(EnumSortMode.DESC);

assertEquals(3, infractionInfo.size());

assertEquals("error3 (30 Points)", infractionInfo.get(0));

assertEquals("error2 (20 Points)", infractionInfo.get(1));

assertEquals("error1 (10 Points)", infractionInfo.get(2));

PlaceholderCollection collection = new PlaceholderCollection();

userReport.generateOutputFile("template.txt", collection);

// Add more assertions here to test the output file, if needed

}

}

This test creates a new UserErrorReport instance and adds two FileErrorReport instances to it, each containing one or two ErrorReport instances. It then checks that the score and other properties of the UserErrorReport are correct, and test the getDetailedFileInfo() and getDetailedInnfractionInfo() methods to ensure that they sort the results correctly. Finally, it creates a PlaceholderCollection and tests the generateOutputFIle() method to ensure that it writes the expected output to a file.

**Unit Tests for the Config.java class**

import org.junit.jupiter.api.Test;

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

public class ConfigTest {

@Test

public void testExampleField() {

String expected = "Hello, World!";

String actual = Config.EXAMPLE\_FIELD;

assertEquals(expected, actual);

}

@Test

public void testDatabaseAddress() {

String expected = "./config/db.db";

String actual = Config.Database.ADDRESS;

assertEquals(expected, actual);

}

@Test

public void testDatabasePort() {

String expected = "";

String actual = Config.Database.PORT;

assertEquals(expected, actual);

}

@Test

public void testDatabaseEngine() {

String expected = "jdbc:sqlite:";

String actual = Config.Database.ENGINE;

assertEquals(expected, actual);

}

@Test

public void testDatabaseUsername() {

String expected = "";

String actual = Config.Database.USERNAME;

assertEquals(expected, actual);

}

@Test

public void testDatabasePassword() {

String expected = "";

String actual = Config.Database.PASSWORD;

assertEquals(expected, actual);

}

}

This test ensures that the value of each field in the Config class match the expected values. If any of these tests fail, it would indicate that the configuration file was not loaded correctly or that the values in the file were not set correctly.

**Java unit test for the ConfigHandler class:**

import org.junit.Before;

import org.junit.Test;

import java.io.File;

import java.io.IOException;

import java.util.Arrays;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import static org.junit.Assert.assertEquals;

import static org.junit.Assert.assertNotNull;

import static org.junit.Assert.assertTrue;

public class ConfigHandlerTest {

private static final String TEST\_CONFIG\_PATH = "src/test/resources/";

private static final String TEST\_CONFIG\_FILE = "test\_config.json";

private static final String TEST\_HEADER\_NAME = "Test Header";

private static final int TEST\_HEADER\_VERSION = 1;

private static final String TEST\_SUBHEADER\_NAME = "Test Subheader";

private static final String TEST\_FIELD\_NAME = "testField";

private static final String TEST\_LIST\_FIELD\_NAME = "testListField";

private static final String TEST\_OBJECT\_FIELD\_NAME = "testObjectField";

private static final String TEST\_OBJECT\_FIELD\_SUBFIELD\_NAME = "testSubfield";

private static final String TEST\_FIELD\_VALUE = "testValue";

private static final String TEST\_LIST\_FIELD\_VALUE = "listValue";

private static final String TEST\_OBJECT\_FIELD\_VALUE = "objectValue";

private static final String TEST\_OBJECT\_FIELD\_SUBFIELD\_VALUE = "subfieldValue";

@Before

public void setUp() throws Exception {

// Ensure that the test config file exists and is empty

File configFile = new File(TEST\_CONFIG\_PATH + TEST\_CONFIG\_FILE);

configFile.createNewFile();

}

@Test

public void testRegister() throws IOException {

// Create a new test config object

TestConfig config = new TestConfig();

config.header = new TestHeader(TEST\_HEADER\_NAME, TEST\_HEADER\_VERSION);

config.subheaders = Arrays.asList(new TestSubheader(TEST\_SUBHEADER\_NAME));

config.testField = TEST\_FIELD\_VALUE;

config.testListField = Arrays.asList(TEST\_LIST\_FIELD\_VALUE);

config.testObjectField = new TestObjectField(TEST\_OBJECT\_FIELD\_VALUE, new TestSubObjectField(TEST\_OBJECT\_FIELD\_SUBFIELD\_VALUE));

// Register the config object

ConfigHandler.register(config, TEST\_CONFIG\_PATH + TEST\_CONFIG\_FILE);

// Verify that the config file was created and contains the correct data

File configFile = new File(TEST\_CONFIG\_PATH + TEST\_CONFIG\_FILE);

assertTrue(configFile.exists());

Map<String, Object> data = ConfigHandler.parseClass(TestConfig.class, new HashMap<>());

assertNotNull(data);

Map<String, Object> headerData = (Map<String, Object>) data.get("header");

assertEquals(TEST\_HEADER\_NAME, headerData.get("name"));

assertEquals(TEST\_HEADER\_VERSION, headerData.get("version"));

List<Map<String, Object>> subheaderData = (List<Map<String, Object>>) data.get(TEST\_SUBHEADER\_NAME);

assertNotNull(subheaderData);

assertEquals(1, subheaderData.size());

Map<String, Object> fieldData = (Map<String, Object>) data.get(TEST\_FIELD\_NAME);

assertNotNull(fieldData);

assertEquals(TEST\_FIELD\_VALUE, fieldData.get("data"));

List<Map<String, Object>> listFieldData = (List<Map<String, Object>>) data.get(TEST\_LIST\_FIELD\_NAME);

assertNotNull(listFieldData);

assertEquals(1, listFieldData.size());

assertEquals(TEST\_LIST\_FIELD\_VALUE, listFieldData.get(0).get("data"));

Map<String, Object> objectFieldData = (Map<String, Object>) data.get(TEST\_OBJECT\_FIELD\_NAME);

assertNotNull(objectFieldData);

assertEquals(TEST\_OBJECT\_FIELD\_VALUE, objectFieldData.get("data"));

Map<String, Object> subObjectFieldData = (Map<String, Object>) objectFieldData.get("subObject");

assertNotNull(subObject

This test is dor the “calculateFactorial” method in the “MathUtilis” class. It tests the method to see if it returns the correct factorial value for the given integer input .It checks for various input values such as 0,1,2,5 and 19, and verifies if the expected output matches the actual output. The test passes if all the assertions are true, indicating that the method is working correctly.

**Java unit test for the Database class**

package com.ch794.marker.data;

import org.junit.jupiter.api.\*;

import java.sql.ResultSet;

import java.sql.SQLException;

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

class DatabaseTest {

private Database db;

@BeforeEach

void setUp() {

db = new Database();

}

@AfterEach

void tearDown() throws SQLException {

if (db != null && !db.conn.isClosed()) {

db.conn.close();

}

}

@Test

void testExecute() {

Query query = new Query("SELECT COUNT(\*) FROM my\_table");

Integer result = db.execute(query, resultSet -> {

try {

if (resultSet.next()) {

return resultSet.getInt(1);

}

} catch (SQLException e) {

e.printStackTrace();

}

return null;

});

assertNotNull(result);

assertEquals(0, result.intValue());

}

@Test

void testExecuteGenerator() {

Query query = new Query("INSERT INTO my\_table (id, name) VALUES (1, 'John Doe')", true);

Integer result = db.executeGenerator(query, ResultSet::getRow);

assertNotNull(result);

assertEquals(1, result.intValue());

}

@Test

void testExecuteWithoutResult() {

Query query = new Query("CREATE TABLE my\_table (id INTEGER PRIMARY KEY, name TEXT)");

db.execute(query);

// if no exception was thrown, the table was successfully created

}

}

Junit 5 annotations are used to tear down the databases instances and test its methods. In the setup(), a new Database instance is created and in teardown(), the database connection is closed if it is still open. In the testExecute() method, a Query object is created that selects the count of rows from a non-existent my\_table. Then call db. execute(query, resultSet - > {…}), passing in a lambda that processes the result set and returns an integer. The lambda checks if the result set has a next row, if it does it returns the integer value of the first column. Junit assertions are used to check that the result is not null and is equal to 0. In the testExecuteGenerator() method, a Query object is created that inserts a new row into my\_table, and passes the true parameter to indicate that we want to retrieve the generated keys. The db.executeGenerator(query, ResultSet::getRow) is called, passing in a method reference to ResultSet::getRow, which retrieves the first row of the result set. Junit assertions are used to check that the result is not null and is equal to 1. In the testExecuteWithoutResult() method, a Query object is created that creates a new table in the database. After calling db.execute(query) without a result set handler, as the result doesn’t matter at the moment. It simply checks that no exception was thrown, which indicates that the table was successfully created.

**Java Unit Test for the Pair class**

package com.ch794.marker.data;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class PairTest {

@Test

public void testPairOf() {

Pair<String, Integer> pair = Pair.of("test", 123);

assertEquals("test", pair.getLeft());

assertEquals(Integer.valueOf(123), pair.getRight());

}

}

This unit tests the of() method of the Pair class which creates a new Pair object from two objects of different types. The test creates a Pair object using a String and an Integer, and then tests that the getLeft() methods returns the String object and that the getRight() method returns the Integer object.

**Unit Test for the Query class**

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class QueryTest {

@Test

public void testQueryBuilder() {

Query query = Query.builder()

.select("id", "name", "age")

.from("users")

.where(Pair.of("age", 25))

.orderBy("name")

.limit(10)

.build();

assertEquals("SELECT id, name, age FROM users WHERE age=25 ORDER BY name LIMIT 10", query.getQuery());

}

}

This test creates a Query object using the builder pattern and verifies that the resulting SQL query matches the expected string.

**Unit Tests for the EnumSortMode Class**

package com.ch794.marker.enums;

import org.junit.Test;

import static org.junit.Assert.\*;

public class EnumSortModeTest {

@Test

public void testEnumValues() {

assertEquals(2, EnumSortMode.values().length);

assertSame(EnumSortMode.ASC, EnumSortMode.valueOf("ASC"));

assertSame(EnumSortMode.DESC, EnumSortMode.valueOf("DESC"));

}

}

This unit test checks that the EnumSortMode class has two enum values (ASC and DESC), and that theses values can be retrieved using the values() and valuesOf() methods. If any of these checks fail, the test will fail.

**Unit Test for the Windows Class**

package com.ch794.marker.gui;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.Test;

import java.awt.\*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.WindowAdapter;

import java.awt.event.WindowEvent;

public class WindowTest {

@Test

public void testSetup() {

// Given

Window window = new Window();

// When

window.setup();

// Then

JFrame frame = window.getFrame();

Assertions.assertEquals(Window.WINDOW\_NAME, frame.getTitle());

Assertions.assertEquals(Window.WIDTH, frame.getWidth());

Assertions.assertEquals(Window.HEIGHT, frame.getHeight());

Assertions.assertFalse(frame.isResizable());

JPanel parentPanel = window.getParentPanel();

Assertions.assertNotNull(parentPanel.getLayout());

Assertions.assertEquals(2, parentPanel.getComponentCount());

JScrollPane consolePanel = window.getConsolePanel();

Assertions.assertEquals(JScrollPane.VERTICAL\_SCROLLBAR\_ALWAYS, consolePanel.getVerticalScrollBarPolicy());

JTextArea console = window.getConsole();

Assertions.assertTrue(console.isEditable());

Assertions.assertTrue(console.getLineWrap());

Assertions.assertTrue(console.getWrapStyleWord());

JPanel controlPanel = window.getControlPanel();

Assertions.assertNotNull(controlPanel.getLayout());

Assertions.assertEquals(1, controlPanel.getComponentCount());

JComboBox<String> configurationSelectionBox = window.getConfigurationSelectionBox();

Assertions.assertEquals(200, configurationSelectionBox.getPreferredSize().getWidth());

Assertions.assertEquals(25, configurationSelectionBox.getPreferredSize().getHeight());

JButton runButton = window.getRunButton();

Assertions.assertEquals(200, runButton.getPreferredSize().getWidth());

Assertions.assertEquals(50, runButton.getPreferredSize().getHeight());

ActionListener[] actionListeners = runButton.getActionListeners();

Assertions.assertEquals(1, actionListeners.length);

ActionEvent actionEvent = new ActionEvent(this, ActionEvent.ACTION\_PERFORMED, null);

actionListeners[0].actionPerformed(actionEvent);

WindowAdapter[] windowAdapters = frame.getWindowListeners();

Assertions.assertEquals(1, windowAdapters.length);

WindowAdapter windowAdapter = windowAdapters[0];

WindowEvent windowEvent = new WindowEvent(frame, WindowEvent.WINDOW\_CLOSING);

windowAdapter.windowClosing(windowEvent);

}

}

This test checks id the setup method initialises the UI components correctly. It verifies if the JFrame has the expected title, size and resizable properties. It also checks if the parentPanel contains two child components, the consolePanel has the expected scrollbar policy, the console is editable and has the expected line wrap and word wrap settings, and the controlPanel contains one child component, the configurationSelectionBox has the expected size, the runButton has the expected size and action listener, and the WIndowAdapter associated with the JFrame closes the application when the window is closed.

**Unit Test for the Level class**

package com.ch794.marker.logger;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.Test;

public class LevelTest {

@Test

public void testLevelEnum() {

Assertions.assertEquals(Level.INFO, Level.valueOf("INFO"));

Assertions.assertEquals(Level.WARN, Level.valueOf("WARN"));

Assertions.assertEquals(Level.ERROR, Level.valueOf("ERROR"));

Assertions.assertEquals(Level.FATAL, Level.valueOf("FATAL"));

Assertions.assertEquals(Level.DEBUG, Level.valueOf("DEBUG"));

}

}

This test verifies that the Level enum contains the expected values and that the valueOf() method returns the correct enum values for each string input.

**Unit Test for the logger class. Since the Logger class interacts with the file system and console output it is best to create an integration test that tests its functionality end-to-end.**

package com.ch794.marker.logger;

import org.junit.jupiter.api.\*;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

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

public class LoggerIntegrationTest {

private static final String FILE\_NAME = "test.log";

private static final File LOG\_FILE = new File(FILE\_NAME);

private static final String MESSAGE = "Test message";

private Logger logger;

@BeforeEach

public void setUp() throws FileNotFoundException {

logger = new Logger(Level.INFO);

logger.logs = new PrintWriter(LOG\_FILE);

}

@AfterEach

public void tearDown() {

logger.logs.close();

LOG\_FILE.delete();

}

@Test

public void testLogToFile() throws FileNotFoundException {

logger.info(MESSAGE);

Scanner scanner = new Scanner(LOG\_FILE);

assertTrue(scanner.hasNextLine());

String logLine = scanner.nextLine();

assertTrue(logLine.contains(MESSAGE));

scanner.close();

}

@Test

public void testLogToConsole() {

final String expected = "[INFO] [com.ch794.marker.logger.LoggerIntegrationTest:testLogToConsole:41]: " + MESSAGE + "\n";

ByteArrayOutputStream baos = new ByteArrayOutputStream();

PrintStream ps = new PrintStream(baos);

System.setOut(ps);

logger.info(MESSAGE);

String actual = baos.toString();

assertEquals(expected, actual);

}

}

This test creates a new Logger instance with the INFO log level and sets up a file logger to write logs to a temporary file. The tests then logs a message using the Logger instance and asserts that the message was written to the file. Next, the test redirects the console output to a ByteArrayOutputStream and logs another message using the Logger instance. The test then asserts that the message was written to the console in the expected format. Finally, the test cleans up by closing he file logger and deleting the temporary log file.

**Unit Test for the AbstractPlaceholder class**

package com.ch794.marker.placeholder.types;

import org.junit.Test;

import static org.junit.Assert.\*;

public class AbstractPlaceholderTest {

@Test

public void testFill() {

String id = "test";

String value = "123";

AbstractPlaceholder<String> placeholder = new AbstractPlaceholder<String>(id, value) {

@Override

public String getFormattedValue() {

return getValue();

}

};

String line = "This is a %test% placeholder.";

String expected = "This is a 123 placeholder.";

String actual = placeholder.fill(line);

assertEquals(expected, actual);

}

@Test

public void testGetPlaceholder() {

String id = "test";

String value = "123";

AbstractPlaceholder<String> placeholder = new AbstractPlaceholder<String>(id, value) {

@Override

public String getFormattedValue() {

return getValue();

}

};

String expected = "%test%";

String actual = placeholder.getPlaceholder();

assertEquals(expected, actual);

}

@Test

public void testGetId() {

String id = "test";

String value = "123";

AbstractPlaceholder<String> placeholder = new AbstractPlaceholder<String>(id, value) {

@Override

public String getFormattedValue() {

return getValue();

}

};

String expected = "test";

String actual = placeholder.getId();

assertEquals(expected, actual);

}

@Test

public void testGetValue() {

String id = "test";

String value = "123";

AbstractPlaceholder<String> placeholder = new AbstractPlaceholder<String>(id, value) {

@Override

public String getFormattedValue() {

return getValue();

}

};

String expected = "123";

String actual = placeholder.getValue();

assertEquals(expected, actual);

}

}

This tests uses Junits @Test annotation to define several test methods:

* testFill: This tests the fill method by creating an instance of AbstractPlaceholder: with a specific ID and value, calling fill with a test line that contains a placeholder, and verifying that the expected result is returned.
* testGetPlaceholder: This tests the getPlaceholder method by creating an instance of AbstractPlaceholer with a specific ID and value, calling getPlaceholder, and verifying that the expected result is returned.
* testGetId: This test the getId method by creating an instance of AbstractPLaveholder ith a specific ID and value, calling getId, and verifying that the expected result is returned.
* testGetValue: This tests the getValue method by creating an instance of AbstractPlaceholder with a specific ID and value, calling getValue, and verifying that he expected result is returned.

In each test method, we create an instance of AbstractPlaceholder with a specific ID and value, and provide an anonymous inner class to implement the getFormattedValue method, which is required by the abstract class. The getFormattedValue method simply returns the value as a string. Finally the method being tested is called and compares the expected result with the actual result using the assertEquals method provided by Junit.

**Unit Test for DoublePlaceholder class.**

package com.ch794.marker.placeholder.types;

import org.junit.Test;

import static org.junit.Assert.\*;

public class DoublePlaceholderTest {

@Test

public void testGetFormattedValue() {

String id = "test";

Double value = 1234.56789;

String pattern = "#,##0.00";

DoublePlaceholder placeholder = new DoublePlaceholder(id, value, pattern);

String expected = "1,234.57";

String actual = placeholder.getFormattedValue();

assertEquals(expected, actual);

}

@Test

public void testFill() {

String id = "test";

Double value = 1234.56789;

String pattern = "#,##0.00";

DoublePlaceholder placeholder = new DoublePlaceholder(id, value, pattern);

String line = "The value is %test%.";

String expected = "The value is 1,234.57.";

String actual = placeholder.fill(line);

assertEquals(expected, actual);

}

@Test

public void testGetPlaceholder() {

String id = "test";

Double value = 1234.56789;

String pattern = "#,##0.00";

DoublePlaceholder placeholder = new DoublePlaceholder(id, value, pattern);

String expected = "%test%";

String actual = placeholder.getPlaceholder();

assertEquals(expected, actual);

}

@Test

public void testGetId() {

String id = "test";

Double value = 1234.56789;

String pattern = "#,##0.00";

DoublePlaceholder placeholder = new DoublePlaceholder(id, value, pattern);

String expected = "test";

String actual = placeholder.getId();

assertEquals(expected, actual);

}

@Test

public void testGetValue() {

String id = "test";

Double value = 1234.56789;

String pattern = "#,##0.00";

DoublePlaceholder placeholder = new DoublePlaceholder(id, value, pattern);

Double expected = 1234.56789;

Double actual = placeholder.getValue();

assertEquals(expected, actual);

}

}

This test class contains the same basic structure as the previous test class, but with tests that are specific to the DoublePlaceholder class:

* testGetFormattedValue: tests the getFormattedValue method by creating an instance of DoublePlaceholder with a specific value and format pattern, calling getFormattedVAlue, and verifying that the expected formatted string is returned.
* testFill: this tests the fill method in the same way as before, using DoublePlaceholder instead of a generic AbstractPlaceholder.
* testGetPlaceholder, testGetId, and testGetValue are the same as before.
* In each test method an instance of DoublePlaceholder with a specific ID, value, and format pattern, and verify the behaviour of the method using the assertEquals method provided by JUnit.

**Unit Test for the DurationPlaceholder class:**

package com.ch794.marker.placeholder.types;

import org.junit.Test;

import static org.junit.Assert.\*;

public class DurationPlaceholderTest {

@Test

public void testGetFormattedValue() {

String id = "test";

Long value = 987654321L;

DurationPlaceholder placeholder = new DurationPlaceholder(id, value);

String expected = "31 Weeks 6 Days 21 Hours 45 Minutes 54 Seconds";

String actual = placeholder.getFormattedValue();

assertEquals(expected, actual);

}

@Test

public void testFill() {

String id = "test";

Long value = 987654321L;

DurationPlaceholder placeholder = new DurationPlaceholder(id, value);

String line = "The duration is %test%.";

String expected = "The duration is 31 Weeks 6 Days 21 Hours 45 Minutes 54 Seconds.";

String actual = placeholder.fill(line);

assertEquals(expected, actual);

}

@Test

public void testGetPlaceholder() {

String id = "test";

Long value = 987654321L;

DurationPlaceholder placeholder = new DurationPlaceholder(id, value);

String expected = "%test%";

String actual = placeholder.getPlaceholder();

assertEquals(expected, actual);

}

@Test

public void testGetId() {

String id = "test";

Long value = 987654321L;

DurationPlaceholder placeholder = new DurationPlaceholder(id, value);

String expected = "test";

String actual = placeholder.getId();

assertEquals(expected, actual);

}

@Test

public void testGetValue() {

String id = "test";

Long value = 987654321L;

DurationPlaceholder placeholder = new DurationPlaceholder(id, value);

Long expected = 987654321L;

Long actual = placeholder.getValue();

assertEquals(expected, actual);

}

}

This test contains the same basic structure as the previous test classes, but with tests that are specific to the DurationPlaceholder class:

* testGetFormattedValue: this tests the getFormattedVAlue method by creating an instance of DurationPlaceholder with a specific value, calling getFormattedValue, and verifying that the expected formatted strinf is returned.
* testFIll: This test the fill method in the same way as before,,but ising a DurationPlaceholder instead of a generic AbstactPlaceholder.
* testGetPlaceholder, testGetId, and testGetValue are the same as before.

In each test method, we create an instance of DurationPlacceholder with a specific ID and value, and verify the behaviour of the method being tested using the assertEquals method provided by JUnit.

**Unit Test for the EpochPlaceholder class:**

import org.junit.Test;

import static org.junit.Assert.\*;

public class EpochPlaceholderTest {

@Test

public void testGetFormattedValue() {

Long epochTime = 1614710400L; // 2022-03-03 00:00:00 UTC

String expectedValue = "00:00:00 03/03/2022";

EpochPlaceholder placeholder = new EpochPlaceholder("EPOCH", epochTime);

assertEquals(expectedValue, placeholder.getFormattedValue());

}

}

This test creates an instance of the EpochPlacholder class with a Unix epoch time value representing March 3rd , 2022 at midnight UTC. It then expects the formatted value to be “00:00:00 03/03/2022” using the default pattern “HH:mm:ss dd/MM/yyyy”. The assertEquals() method checks that the actual formatted values matches the expected value.

**Unit Test for the IntergerPlaceholder class:**

import org.junit.Test;

import static org.junit.Assert.\*;

public class IntegerPlaceholderTest {

@Test

public void testGetFormattedValue() {

Integer intValue = 42;

String expectedValue = "42";

IntegerPlaceholder placeholder = new IntegerPlaceholder("INT", intValue);

assertEquals(expectedValue, placeholder.getFormattedValue());

}

}

This test creates an instance of the IntergerPlaceholder class with an interger value of 42. It then expects the formatted value to be “42” using the getValue() method and converting the integer value to a string. The assertEquals() method checks that the actual formatted value matches the expected value.

**Unit Test for the ListPlaceholder class:**

import org.junit.Test;

import static org.junit.Assert.\*;

import java.util.Arrays;

import java.util.List;

public class ListPlaceholderTest {

@Test

public void testFill() {

List<String> listValue = Arrays.asList("foo", "bar", "baz");

String line = "The list contains %LIST%";

String expectedValue = "The list contains foo\nbar\nbaz\n";

ListPlaceholder placeholder = new ListPlaceholder("LIST", listValue);

assertEquals(expectedValue, placeholder.fill(line));

}

}

This test creates an instance of the ListPlaceholder class with a list value of “foo”, “bar”, and “baz”. It then expects the result of fill() method to replace the placeholder in the given string with each value in the list, separated by new line characters. The assertEquals() method checks that the actual filled value matched the expected value.

**Unit Test for PercentagePlaceholder class:**

package com.ch794.marker.placeholder.types;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class PercentagePlaceholderTest {

@Test

public void testGetFormattedValue() {

PercentagePlaceholder placeholder = new PercentagePlaceholder("test", 0.75, "#.##");

assertEquals("75.00%", placeholder.getFormattedValue());

}

}

This test creates a PercentagePlaceholder object with a value of 0.75 and a pattern of “#.##”. It then calls the getFormattedVAlue() method and asserts the reault is equal to the expected value of “75.00%”. This test ensure that the PercentagePlaceholder class formats the percentage value correctly according to the specified pattern.

**Unit Test for the StringPlaceholder class:**

package com.ch794.marker.placeholder.types;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.Test;

public class StringPlaceholderTest {

@Test

public void testFill() {

String input = "Hello, %name%!";

String expectedOutput = "Hello, John!";

StringPlaceholder namePlaceholder = new StringPlaceholder("name", "John");

String actualOutput = namePlaceholder.fill(input);

Assertions.assertEquals(expectedOutput, actualOutput);

}

@Test

public void testGetPlaceholder() {

String expectedOutput = "%name%";

StringPlaceholder namePlaceholder = new StringPlaceholder("name", "John");

String actualOutput = namePlaceholder.getPlaceholder();

Assertions.assertEquals(expectedOutput, actualOutput);

}

@Test

public void testGetFormattedValue() {

String expectedOutput = "Anee";

StringPlaceholder namePlaceholder = new StringPlaceholder("name", "Anee");

String actualOutput = namePlaceholder.getFormattedValue();

Assertions.assertEquals(expectedOutput, actualOutput);

}

}

The unit test checks the fill(), getPlaceholder(), and getFormattedValue() methods of the StringPlaceholder class. The fill() method replaces the %name% placeholder in the input string with the value “Anee”, producing the expected output “Hello, Anee!”. The getPlaceholder() method returns the %name% placeholder string, and the getFormattedVAlue() method returns the ”Anee” value.

**Unit Test PlaceholderCollection class:**

import com.ch794.marker.placeholder.types.IntegerPlaceholder;

import com.ch794.marker.placeholder.types.StringPlaceholder;

import org.junit.jupiter.api.Test;

import java.util.Arrays;

import java.util.Optional;

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

class PlaceholderCollectionTest {

@Test

void fill() {

PlaceholderCollection placeholders = new PlaceholderCollection(

new StringPlaceholder("name", "John"),

new IntegerPlaceholder("age", 30)

);

String line = "Hello, my name is %name% and I am %age% years old.";

String filledLine = placeholders.fill(line);

assertEquals("Hello, my name is John and I am 30 years old.", filledLine);

}

@Test

void add() {

PlaceholderCollection placeholders = new PlaceholderCollection(

new StringPlaceholder("name", "John")

);

placeholders.add(new IntegerPlaceholder("age", 30));

assertEquals(2, placeholders.getPlaceholderMap().size());

}

@Test

void getPlaceholderValue() {

PlaceholderCollection placeholders = new PlaceholderCollection(

new StringPlaceholder("name", "John"),

new IntegerPlaceholder("age", 30)

);

Optional<String> name = placeholders.getPlaceholderValue("name");

Optional<String> age = placeholders.getPlaceholderValue("age");

Optional<String> missing = placeholders.getPlaceholderValue("missing");

assertTrue(name.isPresent());

assertEquals("John", name.get());

assertTrue(age.isPresent());

assertEquals("30", age.get());

assertFalse(missing.isPresent());

}

@Test

void getPlaceholder() {

PlaceholderCollection placeholders = new PlaceholderCollection(

new StringPlaceholder("name", "John")

);

Optional<StringPlaceholder> name = placeholders.getPlaceholder("name").map(p -> (StringPlaceholder) p);

Optional<IntegerPlaceholder> age = placeholders.getPlaceholder("age").map(p -> (IntegerPlaceholder) p);

assertTrue(name.isPresent());

assertEquals("John", name.get().getValue());

assertFalse(age.isPresent());

}

}

In this test, an instance called PlaceholderCollection is created with two placeholders (a StringPlaceholder and an IntegerPlaceholder). The fill() method is tested to make sure it correctly fills in the placeholders in a string. The add() method is tests to make sure that new placeholders can be added to the collection. Finally, the getPlaceholderValue() and getPlaceholder() methods to make sure that the values of the placeholder can be retrieved as well as the placeholder themselves from the collection.

**Unit test for the AbstractRegistry class:**

package com.ch794.marker.registry;

import org.junit.jupiter.api.Test;

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

class AbstractRegistryTest {

private static class TestRegistry extends AbstractRegistry<String> {

@Override

public void populate() {

// do nothing

}

@Override

public void add(String path, String s) {

this.put(path, s);

}

}

@Test

void addAndGet() {

TestRegistry registry = new TestRegistry();

registry.add("key1", "value1");

registry.add("key2", "value2");

assertEquals("value1", registry.get("key1"));

assertEquals("value2", registry.get("key2"));

assertNull(registry.get("key3"));

}

@Test

void size() {

TestRegistry registry = new TestRegistry();

assertEquals(0, registry.size());

registry.add("key1", "value1");

assertEquals(1, registry.size());

registry.add("key2", "value2");

assertEquals(2, registry.size());

}

@Test

void isEmpty() {

TestRegistry registry = new TestRegistry();

assertTrue(registry.isEmpty());

registry.add("key1", "value1");

assertFalse(registry.isEmpty());

}

}

A TestRegistry class that extends AbstractREgistry and overrides its populate and add methods. Tests are written to ensure that values can be added to the registry and retrieve them by key, as well as test the size and emptiness of the registry.

**Unit Tests for the MarkingConfigurationRegistry class:**

package com.ch794.marker.registry;

import com.ch794.marker.configuration.MarkingConfiguration;

import com.ch794.marker.data.ConfigHandler;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.mockito.MockedStatic;

import org.mockito.Mockito;

import java.io.File;

import java.util.Optional;

public class MarkingConfigurationRegistryTest {

private MarkingConfigurationRegistry registry;

@BeforeEach

public void setUp() {

registry = new MarkingConfigurationRegistry();

}

@Test

public void testAddAndGet() {

MarkingConfiguration config = new MarkingConfiguration();

registry.add("test", config);

Optional<MarkingConfiguration> retrievedConfig = registry.get("test");

Assertions.assertTrue(retrievedConfig.isPresent());

Assertions.assertEquals(config, retrievedConfig.get());

}

@Test

public void testPopulate() {

// Create a temporary directory to use for testing

File tempDir = new File(System.getProperty("java.io.tmpdir"), "test");

tempDir.mkdirs();

tempDir.deleteOnExit();

// Create some temporary files to use for testing

File configFile1 = new File(tempDir, "config1.json");

configFile1.deleteOnExit();

File configFile2 = new File(tempDir, "config2.txt");

configFile2.deleteOnExit();

File configFile3 = new File(tempDir, "config3.json");

configFile3.deleteOnExit();

// Mock the ConfigHandler to always return a new MarkingConfiguration object

try (MockedStatic<ConfigHandler> mockedConfigHandler = Mockito.mockStatic(ConfigHandler.class)) {

mockedConfigHandler.when(() -> ConfigHandler.register(Mockito.any(), Mockito.anyString())).thenReturn(new MarkingConfiguration());

// Add the temporary directory to the registry

registry.add(tempDir.getName(), null);

// Populate the registry

registry.populate();

// Check that only the .json files were loaded

Assertions.assertEquals(2, registry.size());

Assertions.assertTrue(registry.containsKey(configFile1.getName()));

Assertions.assertTrue(registry.containsKey(configFile3.getName()));

Assertions.assertFalse(registry.containsKey(configFile2.getName()));

}

}

}

This unit tests the functionality of the MarkingConfigurationRegistry calss which is a subclass of Abstractregistry. The populate() method of the class tested to ensure that it correctly loads all the .json files in the ./comfig/marking/directory and creates a MarkingConfiguration object for each file. The get() method of the class is also tested to ensure that it correctly retrieves a MarkingConfiguration object given its ID. The tests use a mock MarkingConfiguration object to simulate loading and retrieving from the registry.

**Unit Test for the Registry class. As the class only contains static fields and a static method, a unit test can be written to verify that init() method initialises the MARKING\_CONFIGURATION registry with the correct data. A mocking library like Mockito to mock the behaviour of the MarkingConfigurationRegistry class and verify that the populate() method is called when init() is called.**

package com.ch794.marker.registry;

import com.ch794.marker.configuration.MarkingConfiguration;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.mockito.Mockito.times;

public class RegistryTest {

@Test

public void testInit() {

// Create a mock of the MarkingConfigurationRegistry class

MarkingConfigurationRegistry registry = Mockito.mock(MarkingConfigurationRegistry.class);

// Set the MARKING\_CONFIGURATION field to the mock registry

Registry.MARKING\_CONFIGURATION = registry;

// Call the init method

Registry.init();

// Verify that the populate method was called once

Mockito.verify(registry, times(1)).populate();

// Verify that the registry was populated with the correct data

assertEquals(Registry.MARKING\_CONFIGURATION.get("test.json").get().getId(), "test.json");

}

}

In this test, we first create a mock of the MarkingConfigurationRegistry class using Mockito. The MARKING\_CONFIGURATION field is set in the Registry class to mock the registry. Next, the init() method is called, which should call the populate() method on the mock registry exactly once. Finally, the registry is verified to see if it has been populated with the correct data by checking that the getId() method of the MarkingConfiguration object with key “test.json” returns “test.json”.

**Unit test TestRunner**

public static void main(String[] args) {

TestRunner.run();

}

This will run all the test cases defined in the Tests class and print out the results to the console.

**Unit Test for CheckstyleOutputCatcher class**

package com.ch794.marker.util;

import com.ch794.marker.configuration.MarkingConfiguration;

import com.puppycrawl.tools.checkstyle.api.AuditEvent;

import org.junit.Before;

import org.junit.Test;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import static org.mockito.Mockito.verify;

public class CheckstyleOutputCatcherTest {

@Mock

private MarkingConfiguration markingConfiguration;

private CheckstyleOutputCatcher catcher;

@Before

public void setUp() {

MockitoAnnotations.openMocks(this);

Marker.setCurrentMarkingConfiguration(markingConfiguration);

catcher = new CheckstyleOutputCatcher();

}

@Test

public void addError\_reportsToCurrentMarkingConfiguration() {

AuditEvent event = new AuditEvent(this, "file", null, 1, "message");

catcher.addError(event);

verify(markingConfiguration).reportAuditEventß(event);

}

}

In this test, we first create a mock instance of MarkingConfiguration and set it as the current configuration in the Marker class. We then create an instance of CheckstyleOutputCatcher and call its addError method with a mock AuditEvent instance. Finally, we verify that the reportAuditEvent method of the mock marking configuration is called with the same AuditEvent instance.

**Unit for Test CheckStyleSecurityManager class using JUnit 5 framework**

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

import java.security.Permission;

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import com.ch794.marker.util.CheckstyleSecurityManager.PreventCheckstyleClosingException;

class CheckstyleSecurityManagerTest {

private SecurityManager originalSecurityManager;

@BeforeEach

void setUp() {

originalSecurityManager = System.getSecurityManager();

}

@AfterEach

void tearDown() {

System.setSecurityManager(originalSecurityManager);

}

@Test

void checkPermission\_exitVM0\_prevented() {

CheckstyleSecurityManager.disableExiting();

SecurityManager securityManager = System.getSecurityManager();

assertNotNull(securityManager);

Permission exitVM0 = new RuntimePermission("exitVM.0");

assertThrows(PreventCheckstyleClosingException.class, () -> securityManager.checkPermission(exitVM0));

}

@Test

void checkPermission\_otherPermission\_allowed() {

CheckstyleSecurityManager.disableExiting();

SecurityManager securityManager = System.getSecurityManager();

assertNotNull(securityManager);

Permission otherPermission = new RuntimePermission("otherPermission");

assertDoesNotThrow(() -> securityManager.checkPermission(otherPermission));

}

@Test

void enableExiting\_securityManagerRemoved() {

CheckstyleSecurityManager.enableExiting();

assertNull(System.getSecurityManager());

}

}

This test checks that when the disableExititng method of the ChcekcstyleSecurityManager class is called, a security manager is installed, and attempting to check exitVM.0 permission results in a PreventCheckstyleClosingException. It also checks that other permisisons are allowed. Finally, it checks that when the enableExiting method is called, the security manager is removed.

**Unit Test for the Maps class usisng JUnit 5 framework:**

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

import java.util.Map;

import org.junit.jupiter.api.Test;

class MapsTest {

@Test

void asHashMap\_emptyArray\_emptyMap() {

Pair<String, Integer>[] pairs = new Pair[0];

Map<String, Integer> map = Maps.asHashMap(pairs);

assertNotNull(map);

assertTrue(map.isEmpty());

}

@Test

void asHashMap\_singlePair\_singleEntryMap() {

Pair<String, Integer>[] pairs = new Pair[] { Pair.of("one", 1) };

Map<String, Integer> map = Maps.asHashMap(pairs);

assertNotNull(map);

assertEquals(1, map.size());

assertTrue(map.containsKey("one"));

assertEquals(1, map.get("one"));

}

@Test

void asHashMap\_multiplePairs\_multipleEntryMap() {

Pair<String, Integer>[] pairs = new Pair[] {

Pair.of("one", 1),

Pair.of("two", 2),

Pair.of("three", 3)

};

Map<String, Integer> map = Maps.asHashMap(pairs);

assertNotNull(map);

assertEquals(3, map.size());

assertTrue(map.containsKey("one"));

assertEquals(1, map.get("one"));

assertTrue(map.containsKey("two"));

assertEquals(2, map.get("two"));

assertTrue(map.containsKey("three"));

assertEquals(3, map.get("three"));

}

}

This test checks that the asHashMap method of the Maps class works as excepted for empty, single-pair, and multiple-pair arrays, For each tst case, it creates an array of Pair objects, calls the asHashMap method with that array, and checks that the resulting HashMap has the expected size and entries.

**Unit Test for the Pair class:**

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

import org.junit.jupiter.api.Test;

class PairTest {

@Test

void constructorAndGetters\_singlePair\_valuesMatch() {

Pair<String, Integer> pair = new Pair<>("one", 1);

assertEquals("one", pair.getLeft());

assertEquals(1, pair.getRight());

}

@Test

void of\_staticFactoryMethod\_singlePair\_valuesMatch() {

Pair<String, Integer> pair = Pair.of("two", 2);

assertEquals("two", pair.getLeft());

assertEquals(2, pair.getRight());

}

@Test

void genericTypeParameters\_differentTypes\_ok() {

Pair<String, Integer> pair1 = new Pair<>("one", 1);

Pair<Integer, String> pair2 = new Pair<>(2, "two");

assertEquals("one", pair1.getLeft());

assertEquals(1, pair1.getRight());

assertEquals(2, pair2.getLeft());

assertEquals("two", pair2.getRight());

}

}

This test checks that the Pair class works as expected for a single pair of value, using both the constructor and the static factory method. It also tests that the type parameters can be different typed for different instances of the Pair class. For each test case, it creates a Pair object, calls its getter methods to retrieve the left and right values, and checks that the retrieved values match the expected values.

**Unit Test for Marker Class**

import com.ch794.marker.configuration.MarkingConfiguration;

import com.ch794.marker.data.ConfigHandler;

import com.ch794.marker.gui.Window;

import com.ch794.marker.logger.Logger;

import com.ch794.marker.registry.Registry;

import com.ch794.marker.tests.TestRunner;

import org.junit.BeforeClass;

import org.junit.Test;

import static org.junit.Assert.\*;

public class MarkerTest {

@BeforeClass

public static void setup() {

// Register Config class and initialize Registry for testing

ConfigHandler.register(Config.class);

Registry.init();

}

@Test

public void testMain() {

// Ensure that main method initializes logger and window

Marker.main(new String[]{});

assertNotNull("Logger should be initialized", Marker.getLogger());

assertNotNull("Window should be initialized", Marker.window);

}

@Test

public void testFields() {

// Ensure that fields are initialized to null by default

assertNull("currentMarkingConfiguration should be null", Marker.currentMarkingConfiguration);

assertNull("Window should be null", Marker.window);

}

}

This test class has two test methods:

* The testMain method test that the main method of the Marker class initialises the logger and window fields properly. It calls the main method with an empty array of arguments and asserts that both logger and window are not null
* The testFields method tests that the currentMarkingConfiguration and window fields of the Marker class are initalised to null by default.It asserts that both fields are null.

To run this test class, we can use any JUnit runner or testing tool, such as IntelliJ IDEA or Maven.