DIT636/DAT560 - Unit Testing Exercise

You have been hired to test our new calendar app! Congratulations!(?)
This program allows users to book meetings, adding those meetings to calendars maintained for rooms and employees. It will actively prevent multiple bookings, and will manage the busy and open status for employees and rooms.

The system enables the following high-level functions:

- Booking a meeting
- Booking vacation time
- Checking availability for a room
- Checking availability for a person
- Printing the agenda for a room
- Printing the agenda for a person

The main interface (user input through the command line) is provided by the **`Main`** method in the **`PlannerInterface`** class. During system testing, a tester would focus on testing through that interface.

Your goal in this exercise is to **perform unit testing of the low-level classes** that are integrated by this interface.

As a tester, you have full access to the source code, available at:
https://chalmers.instructure.com/courses/22401/files/folder/Resources?preview=2601854
(it was originally set up as an Eclipse project, but you can import it into the IDE of your choice).
IDE Configuration Instructions:
https://junit.org/junit5/docs/current/user-guide/#running-tests

You are to do the following:

1. **Formulate an informal test plan.**
    a. Given the above features and the code documentation, plan out a series of unit tests to ensure that these features can be performed without error by the classes.
        i. Make sure you think about both the normal execution and illegal inputs and actions that could be performed. Think of as many things that could go wrong as you can! For instance, you will probably be able to add a normal meeting, but can you add a meeting for February 35th? Try it out.
2. **Write tests in the jUnit framework.**
    a. If a test is supposed to cause an exception to be thrown. Make sure you check for that exception.
    b. Make sure that your expected output is detailed enough to ensure that - if something is supposed to fail - that it fails for the correct reasons. Use appropriate assertions.

jUnit Basics

JUnit is a Java-based toolkit for writing executable tests.

- Choose a target from the code base.

```
public class Calculator {
  public int evaluate (String expression) {
    int sum = 0;
    for (String summand: expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

- Write a “testing class” containing a series of unit tests centered around testing that target.
    - Tests and code classes are generally kept in separate folders with parallel class structure.
        - For example, if Calculator is in package cse.dit635.calculator, the code would generally found in the folder **src/main/java/cse/dit635/calculator/Calculator.java**
        - Your tests would then be in the folder **src/test/java/cse/dit635/calculator/CalculatorTest.java**. This test class is also in package cse.dit635.calculator.
    - Each test is denoted **@test**

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class CalculatorTest {
  @Test
  public void testEvaluate_normal() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
  }
}
```

- @BeforeEach annotation defines a common test initialization method:

```
@BeforeEach
public void setUp() throws Exception
{
        this.registration = new Registration();
        this.registration.setUser(“ggay”);
}
```

- @AfterEach annotation defines a common test tear down method:

```
@AfterEach
public void tearDown() throws Exception
{
        this.registration.logout();
        this.registration = null;
}
```

- @BeforeClass defines initialization to take place before any tests are run.

```
@BeforeClass
  public static void setUpClass() {
        myManagedResource = new
                ManagedResource();
  }
```

- @AfterClass defines tear down after all tests are done.

```
 @AfterClass
  public static void tearDownClass() throws IOException {
    myManagedResource.close();
    myManagedResource = null;
  }
```

- Assertions are a "language" of testing - constraints that you place on the output.
    - assertEquals, assertArrayEquals
        - Compares two items for equality.
        - For user-defined classes, relies on `.equals` method.
            - Compare field-by-field
            - `assertEquals(studentA.getName(), studentB.getName())` rather than `assertEquals(studentA, studentB)`

```
@Test
public void testAssertEquals() {
   assertEquals("failure - strings are not equal", "text", "text");
}
```

    - assertArrayEquals compares arrays of items.

```
@Test
public void testAssertArrayEquals() {
        byte[] expected = "trial".getBytes();
        byte[] actual = "trial".getBytes();
        assertArrayEquals("failure - byte arrays
                not same", expected, actual);
}
```

    - assertFalse, assertTrue
        - Take in a string and a boolean expression.
        - Evaluates the expression and issues pass/fail based on outcome.
        - Used to check conformance of solution to expected properties.

```
@Test
public void testAssertFalse() {
   assertFalse("failure - should be false", (getGrade(studentA,
“CSCE747”).equals(“A”));
```

```
}
@Test
public void testAssertTrue() {
        assertTrue("failure - should be true", (getOwed(studentA) > 0));
}
```

- assertNull, assertNotNull
  - Take in an object and checks whether it is null/not null.
  - Can be used to help diagnose and void null pointer exceptions.

```
@Test
public void testAssertNotNull() {
   assertNotNull("should not be null", new Object());
}

@Test
public void testAssertNull() {
   assertNull("should be null", null);
}
```

- assertSame,assertNotSame
  - Checks whether two objects are clones.
  - Are these variables aliases for the same object?
    - assertEquals uses .equals().
    - assertSame uses ==

```
@Test
public void testAssertNotSame() {
   assertNotSame("should not be same Object", studentA, new Object());
}
@Test
public void testAssertSame() {
    Student studentB = studentA;
   assertSame("should be same", studentA, studentB);
}
```

- We can use assertions to verify that expected exceptions are thrown:

```
@Test
void exceptionTesting() {
    Throwable exception = assertThrows(
        IndexOutOfBoundsException.class,
        () -> {
            new ArrayList<Object>().get(0);
         });
         assertEquals("Index:0, Size:0",
             exception.getMessage());
}
```

  - **assertThrows** checks whether the code block throws the expected exception.
  - **assertEquals** can be used to check the contents of the stack trace.