FUNDAMENTAL PROGRAMMING TECHNIQUES

ASSIGNMENT 1 – SUPPORT PRESENTATION (PART 3)

Outline

- Unit Testing with JUnit
- Regular expressions and pattern matching

Unit Testing with JUnit

Unit Testing with JUnit

- Configure Maven to work with Junit – add the Junit dependency in pom.xml

```
…
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.9.2</version>
    <scope>test</scope>
</dependency>
<dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
</dependency>
…
```

```
…
<build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.0.0-M7</version>
        <dependencies>
          <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.4.0</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
</build>
…
```

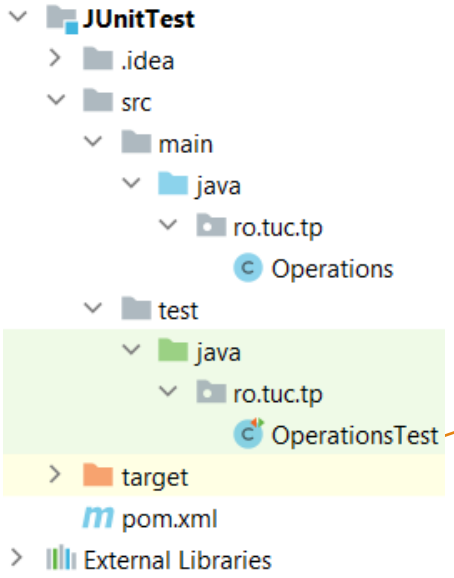
Unit Testing with JUnit

- Consider the class *Operations* that defines methods for adding/subtracting/multiplying two numbers

```
public class Operations {
    public static int add(int firstNumber, int secondNumber) {
        return firstNumber + secondNumber;
    }
    public static int subtract(int firstNumber, int secondNumber) {
        return firstNumber - secondNumber;
    }
    public static int multiply(int firstNumber, int secondNumber) {
        return firstNumber * secondNumber;
    }
}
```

Unit Testing with JUnit

- Create the test class
  - Create a java test class named *OperationsTest.java* and place it in src/main/test
  - Implement a test method named *addTest* in your test class
  - Specify the annotation @Test to the method *addTest*()
  - Implement the test condition and check the condition using *assertEquals* API of JUnit

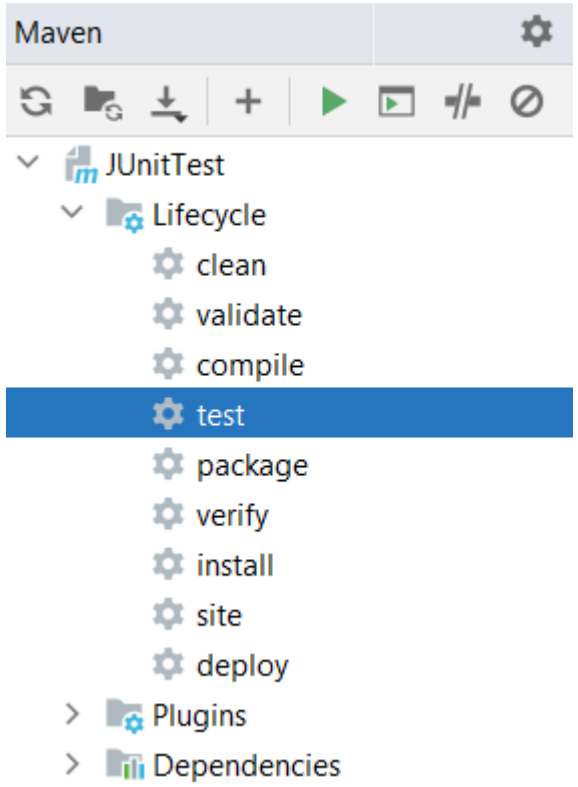


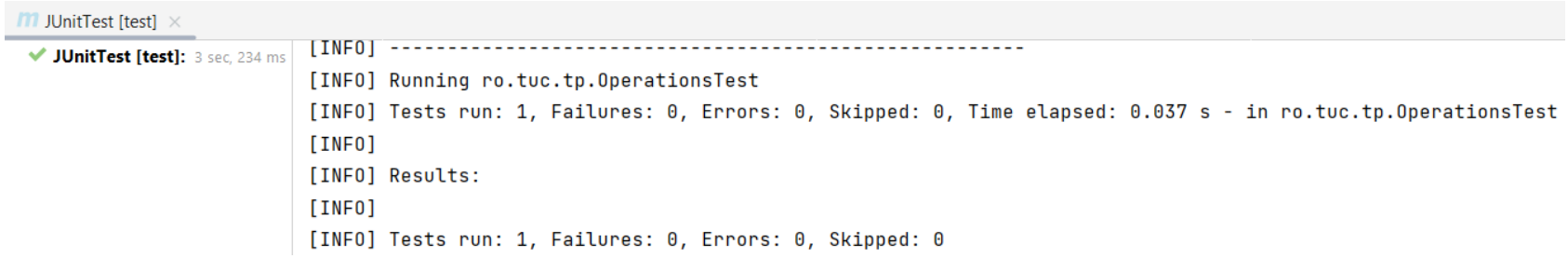

```
package ro.tuc.tp;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class OperationsTest {
    @Test
    public void addTest(){
        assertEquals(Operations.add(2,3), 5);
    }
}
```

Unit Testing with JUnit

- Run the test





Unit Testing with JUnit

- **Basic Annotations** (Link)

Annotation	Description
@Test	Denotes that a method is a test method.
@ParameterizedTest	Denotes that a method is a parameterized test.
@RepeatedTest	Denotes that a method is a test template for a repeated test.
@BeforeEach	Denotes that the annotated method should be executed before each @Test, @RepeatedTest, @ParameterizedTest, method in the current class.
@AfterEach	Denotes that the annotated method should be executed after each @Test, @RepeatedTest, @ParameterizedTest method in the current class.
@BeforeAll	Denotes that the annotated method should be executed before all @Test, @RepeatedTest, @ParameterizedTest methods in the current class;
@AfterAll	Denotes that the annotated method should be executed after all @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class.
…	…

Unit Testing with JUnit

- **Assertions** are static methods defined in the org.junit.jupiter.api.Assertions class: assertEquals, assertAll, assertNotEquals, assertTrue, etc. - check (Link) for more examples
  - In case the assertion facilities provided by JUnit Jupiter are not sufficient enough, third party libraries can be used (e.g. AssertJ, Hamcrest, etc.)

Unit Testing with JUnit

- **Parameterized Tests** [Link] - make it possible to run a test multiple times with different arguments
  - Must declare at least one source that will provide the arguments for each invocation and then consume the arguments in the test method

```
package ro.tuc.tp;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;
import java.util.ArrayList;
import java.util.List;
import static junit.framework.TestCase.assertEquals;

public class ParameterizedTestClass {
    @ParameterizedTest
    @MethodSource("provideInput")
    void testAdditions(int firstNumber, int secondNumber, int expectedResult){
      assertEquals(expectedResult, Operations.add(firstNumber, secondNumber)); }

    private static List<Arguments> provideInput(){
        List<Arguments> arguments = new ArrayList<>();
        arguments.add(Arguments.of(2, 3, 5));
        arguments.add(Arguments.of(4, 6, 10));
        arguments.add(Arguments.of(12, 23, 35));
        return arguments;
    }
}
```

Note:

1) Add the following dependency

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.9.2</version>
    <scope>test</scope>
</dependency>
```

2) The method providing the arguments must be static

Regular expressions and pattern matching

Regular expressions and pattern matching

- **java.util.regex package** [Ref]
  - Contains classes used for pattern matching with regular expressions
  - Regular expression = sequence of characters defining a search pattern
  - Result of matching a regular expression against a text
    - True/false result -> specifies if the regular expression matched the text
    - Set of matches – one match for every occurrence of the regular expression found in the text
  - Consists of the classes:

Class	Description
Pattern	• **Pattern object** = compiled representation of a regular expression • **compile() methods** - accept a regular expression as the first argument, to return a **Pattern object**

Matcher	• **Matcher object** = engine that interprets the pattern and performs match operations against an input string
	• **matcher() method** – invoked on a Pattern object to obtain a Matcher object
	• Other methods
	  • **Index methods** (start, end) – show where the match was found in the input string
	  • **Study methods** (lookingAt, find, matches) – review the input string and return a Boolean indicating whether or not the pattern is found
	  • **Replacement methods** (appendReplacement, appendTail, replaceAll, replaceFirst, quoteReplacement) – replace text in an input string

PatternSyntaxException	**PatternSyntaxException object** – unchecked exception indicating syntax error in a regular expression pattern

Regular expressions and pattern matching

- **Constructs**

Category	Construct	Matches
Character classes	[abc]	a, b, or c (simple class)
	[^abc]	Any character except a, b, or c (negation)
	[a-zA-Z]	a through z or A through Z, inclusive (range)
	[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
	[a-z&&[def]]	d, e, or f (intersection)
	[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
	[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)
Predefined character classes	.	Any character
	\d	A digit: [0-9]
	\D	A non-digit: [^0-9]
	\s	A whitespace character: [\t\n\x0B\f\r]
	\S	A non-whitespace character: [^\s]
	\w	A word character: [a-zA-Z_0-9]
	\W	A non-word character: [^\w]
Logical operators	*XY*	*X* followed by *Y*
	X|*Y*	Either *X* or *Y*
	(*X*)	X, as a capturing group

Regular expressions and pattern matching

- **Constructs**

Category	Construct	Matches
Greedy quantifiers	*X*?	*X*, once or not at all
	*X**	*X*, zero or more times
	X+	*X*, one or more times
	X{*n*}	*X*, exactly *n* times
	X{*n*,}	*X*, at least *n* times
	X{*n*,*m*}	*X*, at least *n* but not more than *m* times
Reluctant quantifiers	*X*??	*X*, once or not at all
	*X**?	*X*, zero or more times
	X+?	*X*, one or more times
	X{*n*}?	*X*, exactly *n* times
	X{*n*,}?	*X*, at least *n* times
	X{*n*,*m*}?	*X*, at least *n* but not more than *m* times
	X??	*X*, once or not at all
Possessive quantifiers	*X*?+	*X*, once or not at all
	*X**+	*X*, zero or more times
	X++	*X*, one or more times
	X{*n*}+	*X*, exactly *n* times
	X{*n*,}+	*X*, at least *n* times
	X{*n*,*m*}+	*X*, at least *n* but not more than *m* times

Quantifiers allow users to specify the number of occurrences to match against [Link]:

- **Greedy Quantifier (Default)**
  - Try to match the longest text that matches a given pattern
  - Work by first reading the entire string before trying any match
  - If the whole text doesn't match, remove the last character and try again, repeating the process until a match is found.
- **Reluctant Quantifier (Appending a ? after quantifier)**
  - Uses an approach that is the opposite of greedy quantifiers
  - It starts with the first character and processes one character at a time
- **Possessive Quantifier (Appending a + after quantifier)**
  - Matches as many characters as possible, like a greedy quantifier
  - But if the entire string doesn't match, then it doesn't try removing characters from the end

Regular expressions and pattern matching

Greedy Quantifiers - Example

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
class Test
{
    public static void main(String[] args)
    {
    Pattern p = Pattern.compile("g+");
    Matcher m = p.matcher("ggg");
         while (m.find())
          System.out.println("Pattern found from " + m.start() +
              " to " + (m.end()-1));
     }
}
```

Output: Pattern found from 0 to 2

Reluctant Quantifiers - Example

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
class Test
{
    public static void main(String[] args)
    {
        Pattern p = Pattern.compile("g+?");
         Matcher m = p.matcher("ggg");
          while (m.find())
            System.out.println("Pattern found from " + m.start() +
                          " to " + (m.end()-1));
      }
}
```

Output: Pattern found from 0 to 0
Pattern found from 1 to 1
Pattern found from 2 to 2

Regular expressions and pattern matching

Possessive Quantifiers - Example

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
class Test
{
    public static void main(String[] args)
    {
      Pattern p = Pattern.compile("g++");
      Matcher m = p.matcher("ggg");
        while (m.find())
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));
    }
```

Output: Pattern found from 0 to 2

Explanation: In this example we get the same output as Greedy because the whole text matches the pattern.

Regular expressions and pattern matching

- **Example -** Create a regular expression for validating Romanian mobile phone numbers. A valid mobile phone number should contain 10 digits, out of which the first 2 should be 07, and the rest from 0 to

```
…
String PHONE_PATTERN = "07[0-9]{8}";
String PHONE_EXAMPLE = "1711123456";
Pattern pattern = Pattern.compile(PHONE_PATTERN);
Matcher matcher = pattern.matcher(PHONE_EXAMPLE);
if(matcher.matches()){
     System.out.println("The phone is valid");
}
else {
     System.out.println("The phone is not valid");
}
…
```

-

To test your regular expressions check this [link]

Regular expressions and pattern matching

- **Capturing groups**
  - Are a way to treat multiple characters as a single unit
  - Are created by placing the characters to be grouped inside a set of parentheses – example: (ABC)
  - Are numbered by counting their opening parenthesis from left to right – check the example below

The **expression ((A)(B(C)))** contains 4 groups

Group number	Matching
1	((A)(B(C)))
2	(A)
3	(B(C))
4	(C)

- **Example**

```
String text = "John writes about this, and John Doe writes about that," +
                       " and John Wayne writes about everything.“;
String patternString1 = "((John) (.+?)) ";
Pattern pattern = Pattern.compile(patternString1);
Matcher matcher = pattern.matcher(text);
while(matcher.find()) {
    System.out.println("found: <"  + matcher.group(1) +
                    "> <"       + matcher.group(2) +
                    "> <"       + matcher.group(3) + ">");
        }
```

```
found: <John writes> <John> <writes>
found: <John Doe> <John> <Doe>
found: <John Wayne> <John> <Wayne>
```

Sources link1 and link2