Java

Testing

FSR Informatik

21. Januar 2021

Java-Kurs

Overview

Switch

Differentiate

If you need to differentiate multiple possible execution paths, working with if/ else can be impracticable.

```
1  public static void main (String[] args) {
2
3      int address = 2;
4
5      if (address == 1) {
6          System.out.println("Dear Sir,");
7      } else if (address == 2) {
8          System.out.println("Dear Madam,");
9      } else if (address == 4) {
10         System.out.println("Dear Friend,");
11     } else {
12         System.out.println("Dear Sir/Madam,");
13     }
14 }
```

Better use a Switch statement!

Differentiate with Switch

Don't forget the break at the end of each case!

```
1  public static void main (String[] args) {
2
3      int address = 2;
4
5      switch(address) {
6          case 1:
7              System.out.println("Dear Sir,");
8              break;
9          case 2:
10             System.out.println("Dear Madam,");
11             break;
12         case 4:
13             System.out.println("Dear Friend,");
14             break;
15         default:
16             System.out.println("Dear Sir/Madam,");
17             break;
18     }
19 }
```

Differentiate with Switch

Depending on a variable you can switch the execution paths using the keyword **switch**. This works with `int`, `char` and `String`.

The variable is compared with the value following the keyword case. If they are equal, the program will enter the corresponding case block. If nothing fits, the program will enter the default block.

```
1  public static void main (String[] args) {
2      switch(intVariable) {
3          case 1:
4              doSomething();
5              break;
6          default:
7              doOtherThings();
8              break;
9      }
10 }
```

Break

After the last command of the case block, you can tell the program to leave using **break**.
Without **break** the program will continue regardless of whether a new case started, like in the example below.

```
1  public static void main (String[] args) {
2
3      switch( 1 ) {
4          case 1:
5              System.out.println("enter case 1");
6          case 2:
7              System.out.println("enter case 2");
8              break;
9          default:
10             System.out.println("enter default case");
11             break;
12     }
13 }
```

Enum Types

Enum

An enum is a special data type that enables for a variable to be a set of predefined constants.

```
1 public enum Day{
2     SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
3     THURSDAY, FRIDAY, SATURDAY
4 }
```

Switch with Enum

Using the enum from the previous slide. You can write a switch statement as follows:

```
1  public void tellItLikeItIs(Day day) {
2      switch (day) {
3          case FRIDAY:
4              System.out.println("Fridays are good.");
5              break;
6
7          case SATURDAY: case SUNDAY:
8              System.out.println("Weekends are best.");
9              break;
10
11         default:
12             System.out.println("Midweek days are so-so.");
13             break;
14     }
15 }
```

Testing

Testing

Testing helps to minimize bugs in your program.
A good program should be tested.

But: Tests never show the absence of bugs.

Test Case

A test case is a check if a program functions behaves and functions properly with the specified input. It always includes an expected result. It can succeed or fail.

After a tested part is altered the test case can be reused to examine if the new part still works as intended.

Covering the Cases

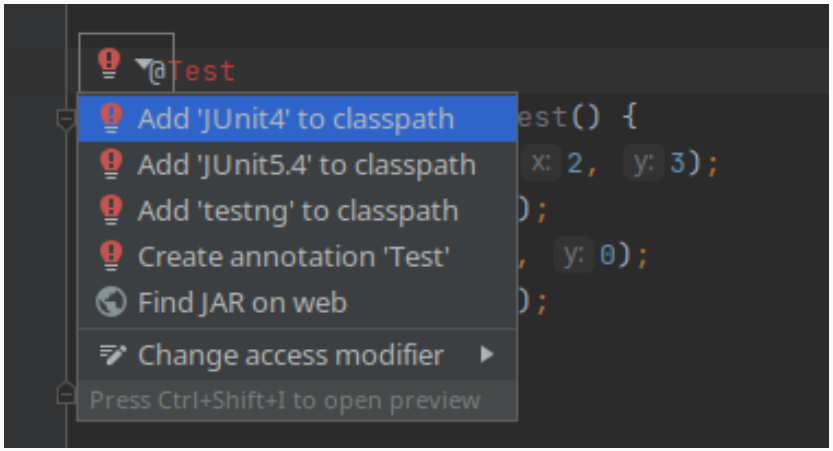
A test case includes the normal case.
Every type of possible input should be tested. But do not write tests for every possible input.

Focus on edge cases and corner cases of your software.
Always ask yourself: *How do I expect my program to handle defective input.*

JUnit

Importing JUnit

When you use the annotation @Test for the first time, Intellij will propose to import and add it to the classpath. Just press enter, and you are good to go.



Example Class

First we need a class we want to test.

```
1 public class Calc {
2
3     public int add(int x, int y) {
4             return x + y;
5     }
6 }
```

Create a test class stub automatically via `Alt+Ins` >`Test` and select the methods you want to test.

The *Name* is **CalcTest** and the *class being tested* is **Calc**.

Test Class

This is an empty test class. In the method `void add()` the tests for the method `int add(int a, int b)` can be implemented. The method is annotated with **@Test**.

```
1  import org.junit.jupiter.api.Test;
2
3  import static org.junit.jupiter.api.Assertions.*;
4
5  class CalcTest {
6
7      @Test
8      void add() {
9      //Not yet implemented
10     }
11 }
```

You can run the test by clicking the green arrow next to the declaration of the function.

The first Test

`assertEquals()` checks if the two parameter are equal.

`assertTrue()` checks if the condition is true.

`assertFalse()` checks if the condition is false.

```
1  import org.junit.jupiter.api.Test;
2
3  import static org.junit.jupiter.api.Assertions.*;
4
5  class CalcTest {
6
7      @Test
8      void add() {
9          Calc calc = new Calc();
10         assertEquals(calc.add(3, 5), 8);
11         assertFalse(calc.add(40, 2) = 42);
```

If and only if all assert methods pass the test will succeed.

Multiple Tests

You can put multiple tests in one test class.

```
1  public class CalcTest {
2
3      @Test
4      public void testNormalAddition() {
5          Calc calc = new Calc();
6          assertEquals(8, calc.add(3, 5));
7      }
8
9      @Test
10     public void testNegativeAddition() {
11         Calc calc = new Calc();
12         assertEquals(-2, calc.add(3, -5));
13     }
14 }
```

Annotation - Before

A method annotated with **@Before** will run before the test methods.

Use a method to set up an environment for the tests.

Now the tests will become smaller and easier to read.

```
1  public class CalcTest {
2
3      private static Calc calc;
4
5      @BeforeAll
6      public static void init() {
7          this.calc = new Calc();
8      }
9
10     @Test
11     public void testNormalAddition() {
12         assertEquals(8, this.calc.add(3, 5));
13     }
14 }
```

Annotation - After

A method annotated with **@AfterAll** will run after the test methods.
Use the method if there are things to tidy up:

- close database connections
- close file streams
- delete test databases or test files

Other Annotations

For example:

- @BeforeEach Method is executed before each @Text
- @AfterEach Method is executed after each @Text
- @TimeOut Test fails if execution exceeds a given duration

Have a look here for even more annotations, and more details:
`https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations`

Example Bookshelf

```
1   public class Bookshelf {
2
3       private String[] books;
4
5       public Bookshelf () {
6           this.books = new String[10];
7           this.books[0] = "Harry Potter";
8       }
9
10      public String getBook(int number) {
11         return this.books[number];
12      }
13  }
14
```

Test Bookshelf

If your software throws exceptions you have to verify the proper throwing of these exceptions.

```
1  public class BookshelfTest {
2      private static BookShelf bookshelf;
3
4      @BeforeAll
5      static void init() {
6          bookshelf = new BookShelf();
7      }
8
9      @Test
10     public void testOutOfBounds() {
11         assertThrows(ArrayIndexOutOfBoundsException.class, () ->
       bookshelf.getBook(15));
12     }
13 }
14
```

Test Bookshelf - in Detail

```
1 @Test
2 public void testOutOfBounds() {
3     assertThrows(ArrayIndexOutOfBoundsException.class, () ->
      bookshelf.getBook(15));
4 }
```

The `first parameter` has to be a subclass of the interface *Throwable*. `ArrayIndexOutOfBoundsException.class` shows the class of the exception the test expects to be thrown.

To generate the `second parameter` a lambda expression is used. We will introduce them in the next lesson. For now, you can call the function you want to test with the shown syntax.

Why use unit tests?

Unit tests do not need human interaction. The automation of testing needs less time than testing by hand.

It is easy to rerun the tests when software is altered.

Links

JUnit Javadoc:
`https://junit.org/junit5/docs/current/user-guide/#overview`

An extensive JUnit tutorial:
`http://www.vogella.com/tutorials/JUnit/article.html`