



Object-Oriented Design and Programming

Chouki Tibermacine

Chouki.Tibermacine@umontpellier.fr



Assertions & Introduction to Java Unit Testing

Outline

1. Assertions
2. Automatic Unit Testing in Java
 - 2.1 Introduction
 - 2.2 Testing Code
 - 2.3 Writing Good Tests
 - 2.4 Parameterized Tests and Test Suites

Outline

1. Assertions

2. Automatic Unit Testing in Java

2.1 Introduction

2.2 Testing Code

2.3 Writing Good Tests

2.4 Parameterized Tests and Test Suites

Design by Contract

- In Java, there are several techniques for implementing **contracts** in classes :
 1. with **Exceptions** : **preconditions** of methods, by making different if-tests (checking parameters of a method, for ex.) and throwing exceptions if something is wrong
 2. with **Assertions** : **postconditions** of methods and **invariants**
- Assertions enable to check properties in programs
- Programs are interrupted in case an assertion is not respected
- Assertions can be inhibited
- They can help in debugging

Assertions

Syntax

```
assert conditionWhichMustBeTrue;  
assert conditionWhichMustBeTrue: anObject;
```

`anObject` is converted to a `String` and displayed

Example

```
assert (offset==sizeOfFile);  
assert (offset==sizeOfFile):  
    "The file has not been parsed entirely";
```

Good Practices

Assertions are used as :

- Invariants of an algorithm :
 - At the end of an iteration to search the min value, the obtained value must be less than all values processed before
- Stream invariants :
 - This level of the program must not be reached. Example : a switch-case statement, with a default that must never be reached or a loop that must never end
- Postconditions :
 - After parsing a file, the offset must reach the end of the file
- Class invariants :
 - A Person must always have an age between 0 and 140

Bad Practices

Assertions must **not** be used for :

- checking the parameters of a method because they come from elsewhere and may be incorrect (rather, test and treat the problem, possibly by throwing exceptions to the caller)
- checking the results of interactions with a user (which can include errors that must be treated)
- creating side effects : changing the state of a program with assertions in the two expressions that compose them

Important Notes

- Assertions can be enabled or disabled while compiling and running programs
- By default, they are disabled
- Don't assume that they will be always executed
- Good tool when developing, testing and debugging (they enable to set up guards)
- An assertion throws an `AssertionError` (inherits from `Error` and then from `Throwable`), which must not be caught or declared to be thrown
- They are never executed in production code

How to compile and run programs with assertions? with command-line

- Compile a program by enabling assertions :

```
javac -source <version> MyClass.java
```

where <version> must be replaced by you JDK version

- Example :

```
javac -source 14 MyClass.java
```

(run `java -version` to know the version of your JDK)

- Run a program by enabling assertions :

```
java -ea MyClass
```

ea stands for “enable assertions”

- Compile and run the program by enabling assertions :

```
java -ea MyClass.java
```

How to compile and run programs with assertions? with command-line

- We can enable assertions for some packages or classes and disable (-da option) them for others
- Run the command :
man java
then read carefully the part of the manual about -da and -ea options

How to compile and run programs with assertions? with command-line

Exercise

- Write a class named Say with a `main(args...)` method that prints one or several messages received as argument(s) (in args array).
- Put an assertion that checks if at least one argument is received by this method
- Compile and run your program with and without arguments and options :

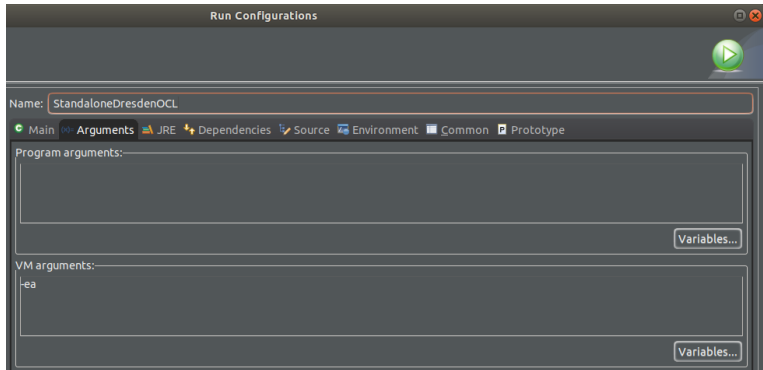
```
java Say
```

```
java -ea Say
```

```
java -ea Say "Hello World!"
```

How to compile and run programs with assertions? In IDEs

- In Eclipse, Menu Run then Run Configurations (see Figure)



- Tell us on Slack how to do it in your favorite IDE

Outline

1. Assertions

2. Automatic Unit Testing in Java

2.1 Introduction

2.2 Testing Code

2.3 Writing Good Tests

2.4 Parameterized Tests and Test Suites

Why to Automate Tests ?

- Identify problems in the code the earliest possible and fix them before it becomes too expensive to fix them
- Repeat them in time
- More generally :
 - Achieve stakeholders goals
 - Meet functional requirements
 - Correctly handle corner/edge cases
 - **Continuously deliver reliable code**
 - **Remove fear from change**

Testing Hierarchy

- Three types of tests, from most specific (and simplest) to most general (and most complex) :
 1. Unit Tests (object of this course)
 2. Aggregation/Integration Tests (test components of the system)
 3. System Tests (end-to-end tests)
- Unit testing : test a single unit of functionality (a method, a class, or a small module), **a single unit of behavior**

A Tool for writing and executing Tests : JUnit

- Origins : Xtreme Programming (Test-First Development) and Agile Methods
- Test Framework written in Java by E. Gamma and K. Beck
- Open Source Project : <http://www.junit.org>
- There are other frameworks, but this is the most popular one

A First Example (from R. Warburton)

- Let us write an app which represents a Cafe that stores beans and milk and enables to brew coffee to produce different kinds of coffees
- 3 classes :
 - CoffeeType : Enum of different kinds of Coffee
 - Coffee : a class for a single cup of coffee
 - Cafe : a class for brewing coffee

A First Example : Coffee class

```
1 package fr.igpolytech.mtp.oodp1;
2
3 public class Coffee {
4     private final CoffeeType type;
5     private final int beans;
6     private final int milk;
7     public Coffee(CoffeeType type, int beans, int milk) {
8         this.type = type;
9         this.beans = beans;
10        this.milk = milk;
11    }
12    public CoffeeType getType() {
13        return type;
14    }
15    public int getBeans() {
16        return beans;
17    }
18    public int getMilk() {
19        return milk;
20    }
21    @Override
22    public String toString() {
23        return "Coffee{" +
24            "type=" + type +
25            ", beans=" + beans +
26            ", milk=" + milk;
27    }
28 }
29
```

A First Example : Cafe class

```
1 package fr.igpolytech.mtp.oodp1;
2
3 public class Cafe {
4     private int beansInStock = 0;
5     private int milkInStock = 0;
6     public Coffee brew(CoffeeType coffeeType) {return brew(coffeeType,1);}
7     public Coffee brew(CoffeeType coffeeType,int quantity) {
8         requirePositive(quantity);
9         int requiredBeans = coffeeType.getRequiredBeans() * quantity;
10        int requiredMilk = coffeeType.getRequiredMilk() * quantity;
11        if(requiredBeans > beansInStock || requiredMilk > milkInStock) {
12            throw new IllegalStateException("Insufficient milk or beans");
13        }
14        beansInStock -= requiredBeans;
15        milkInStock -= requiredMilk;
16        return new Coffee(coffeeType,requiredBeans,requiredMilk);
17    }
18    public void restockBeans(int weightInGrams) {
19        requirePositive(weightInGrams);
20        beansInStock += weightInGrams;
21    }
22    public void restockMilk(int weightInGrams) {
23        requirePositive(weightInGrams);
24        milkInStock += weightInGrams;
25    }
26    private void requirePositive(int value) {
27        if(value < 0) {
28            throw new IllegalArgumentException();
29        }
30    }
31    public int getBeansInStock() {return beansInStock;}
32    public int getMilkInStock() {return milkInStock;}
33 }
```

A First Example : CoffeeType enum

```
1 package fr.igpolytech.mtp.oodpl;
2
3 public enum CoffeeType {
4     Espresso(7,0),
5     Latte(7,227),
6     FilterCoffee(10,0);
7     private final int requiredBeans;
8     private final int requiredMilk;
9     CoffeeType(int requiredBeans, int requiredMilk){
10         this.requiredBeans = requiredBeans;
11         this.requiredMilk = requiredMilk;
12     }
13     public int getRequiredBeans() {
14         return requiredBeans;
15     }
16     public int getRequiredMilk() {
17         return requiredMilk;
18     }
19 }
```

The code is available on Moodle

Testing the Cafe class

- The main functionality to test is associated to brew method
- In Java, by convention, for a given class we write a Testing class : a CafeTest class in our example
- In the testing class, we write a series of methods each of which corresponds to a behavior (each method is a **Test Case**) :
 - can we brew some Espresso ?
 - ...
- Each method should have a name that corresponds to the test made (don't use testBrew1, testBrew2, ...)
- In JUnit, methods must be annotated @Test so that the framework will invoke them in a certain way

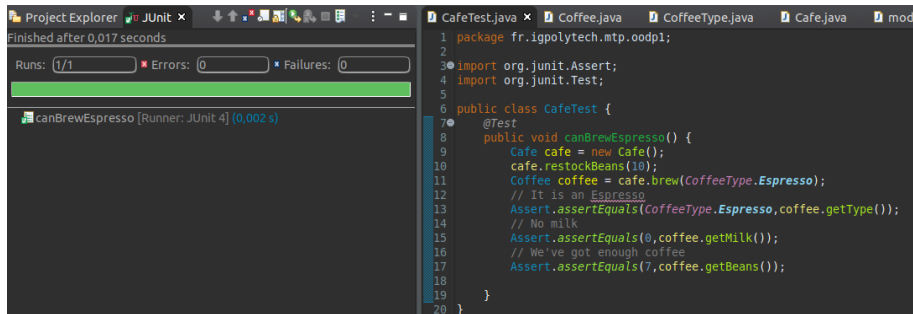
A First Test Example : CafeTest class

```
1 package fr.igpolytech.mtp.oodp1;
2
3 import org.junit.Assert;
4 import org.junit.Test;
5
6 public class CafeTest {
7     @Test
8     public void canBrewEspresso() {
9         Cafe cafe = new Cafe();
10        cafe.restockBeans(7);
11        Coffee coffee = cafe.brew(CoffeeType.Espresso);
12        // It is an Espresso
13        Assert.assertEquals(CoffeeType.Espresso, coffee.getType());
14        // No milk
15        Assert.assertEquals(0, coffee.getMilk());
16        // We've got enough coffee
17        Assert.assertEquals(7, coffee.getBeans());
18    }
19 }
```

Plenty of assertions : `assertTrue(...)`, `assertFalse(...)`,
`assertNull(...)`, `assertSame(...)`, ...

A First Test Example : Running the Test

- On Eclipse, right-click on the method `canBrewEspresso()`, then choose menu item “Run As”, and at last “JUnit Test”



```
1 package fr.igpolytech.mtp.oodp1;
2
3 import org.junit.Assert;
4 import org.junit.Test;
5
6 public class CafeTest {
7     @Test
8     public void canBrewEspresso() {
9         Cafe cafe = new Cafe();
10        cafe.restockBeans(10);
11        Coffee coffee = cafe.brew(CoffeeType.Espresso);
12        // It is an Espresso
13        Assert.assertEquals(CoffeeType.Espresso, coffee.getType());
14        // No milk
15        Assert.assertEquals(0, coffee.getMilk());
16        // We've got enough coffee
17        Assert.assertEquals(7, coffee.getBeans());
18    }
19 }
20 }
```

- Try it and try to break the tests. What happens?
- Did you notice that we do not invoke test methods? JUnit do it
- Don't forget to add JUnit Library in the build-path of your IDE Project

Test Results

- Pass/Run (green) : no errors and failures
- Errors (red) : tests do not run properly
- Failures (red) : tests run properly but the tested behavior does not execute properly

A First Test Example : Running the Test with Failure

- After having changed Line 15 (replaced 0 by 1)

The screenshot displays an IDE with two main panes. The left pane shows the JUnit test runner results. It indicates that the test 'canBrewEspresso' failed after 0.015 seconds. The failure trace shows a `java.lang.AssertionError` with the message 'expected:<1> but was:<0>' at line 15 of `CafeTest.java`. The right pane shows the source code of `CafeTest.java`. The code defines a `Cafe` class and a `canBrewEspresso` method. Line 15, which is highlighted, contains the assertion `Assert.assertEquals(1, coffee.getMilk());`. The code also includes imports for `org.junit.Assert` and `org.junit.Test`, and a `restockBeans` method.

```
1 package fr.igpolytech.mtp.oodp1;
2
3 import org.junit.Assert;
4 import org.junit.Test;
5
6 public class CafeTest {
7     @Test
8     public void canBrewEspresso() {
9         Cafe cafe = new Cafe();
10        cafe.restockBeans(10);
11        Coffee coffee = cafe.brew(CoffeeType.Espresso);
12        // It is an Espresso
13        Assert.assertEquals(CoffeeType.Espresso, coffee.getType());
14        // No milk
15        Assert.assertEquals(1, coffee.getMilk());
16        // We've got enough coffee
17        Assert.assertEquals(7, coffee.getBeans());
18    }
19 }
20 }
```

- See the view at the left. What is in there?

Running Tests with Command Line

- This is possible, but we need to solve many dependencies (a cumbersome task when done manually)
- This is made easy by the well-known **Build Tools** like Maven or Gradle
- Next year, we'll have a course on these tools, and you'll see how it is easy to work on large Java projects with multiple dependencies and many build steps (download dependencies like JUnit, compile, test, build JAR, generate documentation, ...)

Common Structure for Tests

Three clauses in a Test :

- **Given** clause : What should the world looks like when the behavior happens ? (the preconditions)
- **When** clause : What is being tested ? (a simple isolated behavior)
- **Then** clause : What are the changes that happened ? (the postcondition)

Common Structure for Tests on our Example

```
1 package fr.igpolytech.mtp.oodpl;
2
3 import org.junit.Assert;
4 import org.junit.jupiter.api.Test;
5
6 public class CafeTest {
7     @Test
8     public void canBrewEspresso() {
9         Cafe cafe = new Cafe();
10        Given cafe.restockBeans(10);
11        When Coffee coffee = cafe.brew(CoffeeType.Espresso);
12        // It is an Espresso
13        Assert.assertEquals(CoffeeType.Espresso, coffee.getType());
14        // No milk
15        Then Assert.assertEquals(0, coffee.getMilk());
16        // We've got enough coffee
17        Assert.assertEquals(7, coffee.getBeans());
18    }
19 }
20 }
```

Exercise

- Write a second test method for ensuring that brewing espresso consumes beans in the stock
- The same Given and When clauses

Exceptions, Failures and Errors

- Sometimes an exception is the correct result (a wrong input from a user)
- If we want to check the throwing of the exception in the code :

```
// Then clause
@Test(expected=IllegalArgumentException.class)
public void lattesRequireMilk() {
    // Given clause
    Cafe cafe = new Cafe();
    cafe.restockBeans(7);
    // When clause
    cafe.brew(CoffeeType.Latte);
}
```

Without the parameter of the `@Test` annotation, we'll get a failure. Why?

Exceptions, Failures and Errors

- Sometimes an exception is the correct result (a wrong input from a user)
- If we want to check the throwing of the exception in the code :

```
// Then clause
@Test(expected=IllegalArgumentException.class)
public void lattesRequireMilk() {
    // Given clause
    Cafe cafe = new Cafe();
    cafe.restockBeans(7);
    // When clause
    cafe.brew(CoffeeType.Latte);
}
```

Without the parameter of the `@Test` annotation, we'll get a failure. Why? **Because the When clause produces an exception which is not handled**

Exceptions, Failures and Errors – Ctd

- Failures vs Errors :
 - An error : a problem with the code of the test (must be fixed so that the test checks the behavior properly)
 - A failure : the test runs correctly but the tested code does not provide the expected behavior
 - » It is what we get when we tried to break the tests in the previous example (replacing 0 by 1 in the second Then clause)
 - » The assertions executed by JUnit throw an AssertionError with details about the test (expected and actual values)

```
! java.lang.AssertionError: expected:<1> but was:<0>  
at Cafe/fr.igpolytech.mtp.oodp1.CafeTest.canBrewEspresso(CafeTest.java:15)
```

Good Practices

- Give good names to tests : important for readability and thereby help in maintenance/evolution
 - Be descriptive : the name should reflect when and then clauses, even if it is a too long name
 - Use app domain terminology and natural language in naming
- Test the behavior not the implementation
- DRY : Don't Repeat Yourself (avoid duplication when writing tests)
- Diagnostics (output) should help in identifying the problem to fix

Behavior Not Implementation

- Test only the public API (public methods of a class)
- Do not trespassing private members of a class
- Do not expose private members to test them
- Even if we change the implementation of a tested method (use a different data structure, for example), the test should continue to run properly
- Test cases should be independent the ones from the others
- Tests cases are defined without parameters and have void as a return type

DRY : Don't Repeat Yourself

- In our previous two test cases, the Given and When clauses are repeated. This is an anti-pattern (*Duplication Anti-pattern*)
- This produces several places to change
- These two clauses should be put in a single place
 - Do it in your copy of the code
- The value 7 used in the Given clause should be replaced by a constant (ESPRESSO_BEANS). The same for value 0 (NO_MILK constant). This is more readable and prevents duplication of these values

Diagnostics

- If we want to compare values in a test, we have to expose values, by privileging `assertEquals(...)` to `assertTrue()`
- The latter assertion produces only an assertion error. Example

```
assertTrue(aList.size() == 1);  
// Should be replaced by:  
assertEquals(1,aList.size());  
// This will print: expected value=1, but the actual value=0  
// for example in a failing test  
// We can go further by writing  
assertEquals("Wrong quantity of coffee",1,aList.size());
```

- Go back to our example and add String messages like above

Before and after tests

- JUnit provides annotations that can be used for indicating code parts that must be executed before and after test running
- Annotations used for methods :
 - `@Before` : Before each test method runs
 - `@After` : After each test method runs
 - `@BeforeClass` : Before all tests in the class (used to annotate static methods)
 - `@AfterClass` : After all tests in the class (used to annotate static methods)
- Use the `@Before` annotation to annotate a method that you have to write and which contains the creation of the object Cafe needed in each test (change the test cases)
- `@Test` annotation can have a `timeout` parameter :
`@Test(timeout=10)`

The test fails if the test executes in more than 10ms

Testing with JUnit 5

- Assertion methods (`assertEquals(...)`, ...) are provided by class `Assertions` from `org.junit.jupiter.api`

```
import org.junit.jupiter.api.Assertions;  
...  
Assertions.assertEquals(...)
```

```
// or a static import of all methods from Assertions:  
import static org.junit.jupiter.api.Assertions.*;  
...  
assertEquals(...)
```

- `@Before` → `@BeforeEach`, ...
- More details in : <https://junit.org/junit5/docs/current/api/>

AssertThat (since JUnit 4.4)

- `assertThat([value], [matcher statement]);`
- Examples :
 - `assertThat(x, is(3));`
 - `assertThat(x, is(not(4)));`
 - `assertThat(responseString, either(containsString("color")).or(containsString("colour")));`
 - `assertThat(myList, hasItem("3"));`
- `not(s)`, `either(s).or(ss)`, `each(s)`
- Diagnostic messages are more clear
- Use `org.hamcrest.MatcherAssert.assertThat(...)` instead of `Assert.assertThat(...)` (the latter is deprecated)
- All (static) methods `is()`, `not()`, ... can be imported statically :

```
import static org.hamcrest.CoreMatchers.*;
```


Parameterized Test

- Goal : reuse a test method (case) with different datasets
- Test datasets :
 - returned by a method annotated @Parameters
 - This method returns a collection of arrays containing the data and possibly the expected result
- Test Class :
 - annotated @RunWith(Parameterized.class)
 - contains methods which must be executed with each dataset
- For each data, the class is instantiated and the test methods are executed

Parameterized Tests : the Requirements

- A Public constructor which uses the parameters (the dataset)
- The method which returns the parameters (datasets) must be static

Example of a Parameterized Test – to be tested

```
import org.junit.Test; import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*; import java.util.*;
@RunWith(Parameterized.class)
public class TestParamSum {
    private int x; private int y; private int res;
    public TestParamSum(int x, int y, int res) {
        this.x = x; this.y = y; this.res = res;
    }
    @Parameters
    public static Collection testData() {
        return Arrays.asList(new Object[][] {
            { 0, 0, 0 }, { 1, 1, 2 }, { 2, 1, 3 }, {10, 9, 19}});
    }
    @Test
    public void sumCalculatesAddition() {
        assertEquals(res, new Sum().sum(x,y));
    }
}
```

Test Suites

- Group test cases in order to chain their execution
- i.e. group the execution of test classes
- Example : – to be tested (by replacing TestClass1 and TestClass2 by the previous two classes, CafeTest and TestParamSum)

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestClass1.class,
    TestClass2.class
})
public class MyTestSuite {
}
```

