

Unit testing and automation

Tools and Techniques for Software Testing - Barbara Russo

SwSE - Software and Systems Engineering group

Debugging

Tools and Techniques for Software Testing - Barbara Russo
SwSE - Software and Systems Engineering group

Debugging

- Synonym of testing for bugs in code
- java has a set of commands that help find an error
- **javac** to find compiler bugs
- **jdb** to inspect the code for logic bugs

Compiler errors

- ‘javac’ on the DatesBuggy.java class

```
[BarbaraMini-568:CourseLatexNotes barbaramini$ javac DatesBuggy.java
DatesBuggy.java:6: error: illegal start of expression
    if (month == 9) || (month == 4) || (month == 6) || (month == 11)) {
                   ^
DatesBuggy.java:6: error: not a statement
    if (month == 9) || (month == 4) || (month == 6) || (month == 11)) {
                                                         ^
DatesBuggy.java:6: error: ';' expected
    if (month == 9) || (month == 4) || (month == 6) || (month == 11)) {
                                                         ^
DatesBuggy.java:9: error: 'else' without 'if'
    else if (month == 2)
    ^
DatesBuggy.java:27: error: not a statement
    for (aMonth = 0, aMonth < someMonth; aMonth = aMonth + 1) {
                                   ^
DatesBuggy.java:27: error: ';' expected
    for (aMonth = 0, aMonth < someMonth; aMonth = aMonth + 1) {
                                                         ^

6 errors
[BarbaraMini-568:CourseLatexNotes barbaramini$
```

Logic errors

Execute the class between 13th Jan and 4th March:
March: `java DatesW.java 1 13 3 4`

```
[BarbaraMini-568:CourseLatexNotes barbaramini$ java Dates 1 13 3 4  
The difference in days between 1/13 and 3/4 is:  
19
```

Failure: number of days is wrong. Correct output is 50. Where is the error in the code?

- Recompile the program with the ``-g'` option to tell the compiler to provide information that jdb can use to display local (stack) variables
- Then use `'jdb'` on the compiled class

```
[BarbaraMini-568:CourseLatexNotes barbaramini$ javac -g Dates.java  
[BarbaraMini-568:CourseLatexNotes barbaramini$ jdb Dates 1 13 3 4  
Initializing jdb ...  
> █
```

Breakpoints

- At this point, jdb has invoked the Java interpreter, the Dates.class is loaded, and the interpreter stops before entering main()
- Give the command **'stop in DatesBuggy.main'** and then **'run'** and the interpreter will continue executing for a very short time until just after it enters main();
- same applies for any other method (e.g., Dates.daysInMonth)

Breakpoints

```
BarbaraMini-568:CourseLatexNotes barbaramini$ jdb Dates 1 13 3 4
Initializing jdb ...
> stop in Dates.main
Deferring breakpoint Dates.main.
It will be set after the class is loaded.
> run
run Dates 1 13 3 4
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint Dates.main

Breakpoint hit: "thread=main", Dates.main(), line=15 bci=0
15          someMonth = Integer.parseInt(args[0]);

main[1] █
```


Inspect

- Type **'list'** to see the source code for the instructions that are about to execute, or you can type **'print args'** to see the value of the variable called **'args'** or **'locals'** to see all variables

```
main[1] list
11   public static void main (String[] args) {
12       int someMonth, someDay;
13       int laterMonth, laterDay;
14       int aMonth;
15 =>   someMonth = Integer.parseInt(args[0]);
16       someDay = Integer.parseInt(args[1]);
17       laterMonth = Integer.parseInt(args[2]);
18       laterDay = Integer.parseInt(args[3]);
19       /* Used to record what day in the year the first day */
20       /* of someMonth and laterMonth are. */
main[1] █
```

More on Breakpoints

- ‘**stop in**’ means set a breakpoint and then ‘**run**’ executes the program until there
- Continue to examine the program's behaviour as it executes by setting further breakpoints, or using ‘**step**’ to execute one instruction at a time
- At each breakpoint, use the ‘**print**’ or ‘**locals**’ command to examine the values of program variables, until the bug is isolated

Finding the error

- The error in DatesBuggy can be corrected by changing only ONE line.
- When the error is found: change the file and correct the error
- Type **'exit'** to exit the debugging
- Recompile and execute it to see if the problem is solved
- Then **commit** the file with git

Unit Testing

This lecture tools

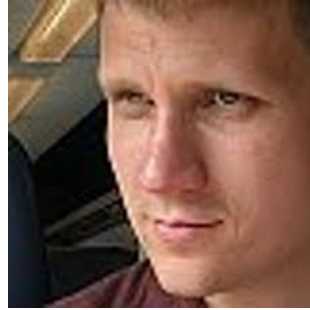
- Eclipse/IntellJ IDEA
- JUnit 5-4
- Maven (little)

Unit testing

- Each time you write a code module, you should write test cases for it
 - A possible exception: accessor methods (i.e., getters and setters)
 - Generally, accessor methods will be written error-free

Unit testing

- It focuses on faults within modules and code that could easily be broken



Test with annotation

JUnit 4 / 5

Developers: Kent Beck, Erich Gamma, David Saff, Kris Vasudevan

JUnit 5

- **JUnit 5 = Platform + Jupiter + Vintage**
- **Platform** launches testing frameworks on the JVM
 - It also provides a *Console Launcher* to launch the platform from the command line and a *JUnit 4 based Runner* for running any TestEngine on the platform in a JUnit 4 based environment
- **Jupiter** is new model for writing tests and extensions in JUnit 5
 - Jupiter provides a TestEngine for running Jupiter based tests
- **Vintage** provides a TestEngine for running JUnit 3 and JUnit 4 based tests

Annotations

- Test annotations characterize methods as test methods
- Annotations are **strongly typed**, so the compiler will flag any mistakes right away
- Test classes no longer need to extend anything (such as TestCase for JUnit 3)
- One can pass additional parameters to annotations

Runners

- We use JUnit **Runners** to execute the test methods
- The Runners can be configured in Eclipse
 - for all project
 - for a single class
 - for a single method

JUnit5

- <https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations>
- all core annotations are located in the **org.junit.jupiter.api**

Maven

- We use it to build and test java projects
- In particular, it provides
 - Dependency list
 - Unit test reports including coverage
- Maven has a central repository for jar and dependencies
 - <https://maven.apache.org/repository/>

The POM

- Project's configuration file in Maven
- XML structure
- It contains the majority of information required to build and test a project
 - It contains info on dependencies
- It automatizes the build and test process

Let's have a look at it

The screenshot shows the Eclipse IDE interface. The main editor displays the contents of a `pom.xml` file. The file is a Maven project configuration for `TTST-demo`, version `0.0.1-SNAPSHOT`. It includes dependencies for `junit` (version `4.12`) and `org.junit.jupiter` (version `5.5.1`).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>it.unibz</groupId>
8     <artifactId>TTST-demo</artifactId>
9     <version>0.0.1-SNAPSHOT</version>
10
11     <dependencies>
12     <dependency>
13         <groupId>junit</groupId>
14         <artifactId>junit</artifactId>
15         <version>4.12</version>
16         <scope>test</scope>
17     </dependency>
18
19     <dependency>
20         <groupId>org.junit.jupiter</groupId>
21         <artifactId>junit-jupiter-engine</artifactId>
22         <version>5.5.1</version>
23         <scope>test</scope>
24     </dependency>
25
26     <dependency>
27         <groupId>org.junit.jupiter</groupId>
28         <artifactId>junit-jupiter</artifactId>
29         <version>5.5.1</version>
30         <scope>test</scope>
31     </dependency>
32 </dependencies>
33 </project>
```

The console output at the bottom shows the Maven build process:

```
<terminated> TTST-demo [Maven Build] /Library/Java/JavaVirtualMachines/jdk-12.0.1.jdk/Contents/Home/bin/java (Oct 14, 2019, 8:35:57 AM)
[INFO] Scanning for projects...
[INFO] -----< it.unibz:TTST-demo >-----
[INFO] Building TTST-demo 0.0.1-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ TTST-demo ---
[INFO] Deleting /Users/barbaramini/unibz/Dropbox/Courses2019_2020/workspace/TTST-demo/target
```

How to create a Maven project

- Let' watch it, step by step

<https://www.youtube.com/watch?v=sNEcpw8LPpo>

and more recent instructions

<https://www.vogella.com/tutorials/EclipseMaven/article.html>

- First create dependencies with JUnit 5 components
- Then create your first class and test class named “App” and “AppUnitTest” in the package “it.unibz”

Run Maven

The screenshot shows the Eclipse IDE interface. The main editor displays a Maven `pom.xml` file with the following content:

```
9 <version>0.0.1-SNAPSHOT</version>
10 <packaging>jar</packaging>
11
12 <dependencies>
13
14 </dependencies>
15
16 <properties>
17
18 </properties>
19
20 <build>
21   <plugins>
22     <plugin>
23     <plugin>
24     <plugin>
25   </plugins>
26   <testResources>
27     <testResource>
28       <filtering>true</filtering>
29       <directory>src/test/resources</directory>
30     </testResource>
31   </testResources>
32   <plugins>
33     <plugin>
34       <groupId>org.apache.maven.plugins</groupId>
35       <artifactId>maven-jar-plugin</artifactId>
36       <configuration>
37         <useManifestMerging>true</useManifestMerging>
38       </configuration>
39     </plugin>
40     <plugin>
41       <groupId>org.apache.maven.plugins</groupId>
42       <artifactId>maven-surefire-plugin</artifactId>
43       <configuration>
44         <test>it.unibz.CalculatorTest</test>
45         <testFailureIgnore>true</testFailureIgnore>
46       </configuration>
47     </plugin>
48     <plugin>
49       <groupId>org.apache.maven.plugins</groupId>
50       <artifactId>maven-failsafe-plugin</artifactId>
51       <configuration>
52         <test>it.unibz.CalculatorTest</test>
53         <testFailureIgnore>true</testFailureIgnore>
54       </configuration>
55     </plugin>
56   </plugins>
57 </build>
58 </pom.xml>
```

A context menu is open over the `<plugin>` tag at line 22. The menu items are:

- New
- Show In
- Open
- Open With
- Copy
- Copy Qualified Name
- Paste
- Delete
- Remove from Context
- Mark as Landmark
- Build Path
- Move...
- Rename...
- Import...
- Export...
- Refresh
- Coverage As
- Run As** (highlighted)
 - m2 1 Maven build
 - m2 2 Maven build...
 - m2 3 Maven clean
 - m2 4 Maven generate-sources
 - m2 5 Maven install
 - m2 6 Maven test
 - Run Configurations...
- Debug As
- Profile As
- Maven
- Team
- Compare With
- Replace With
- Source
- Validate
- Properties

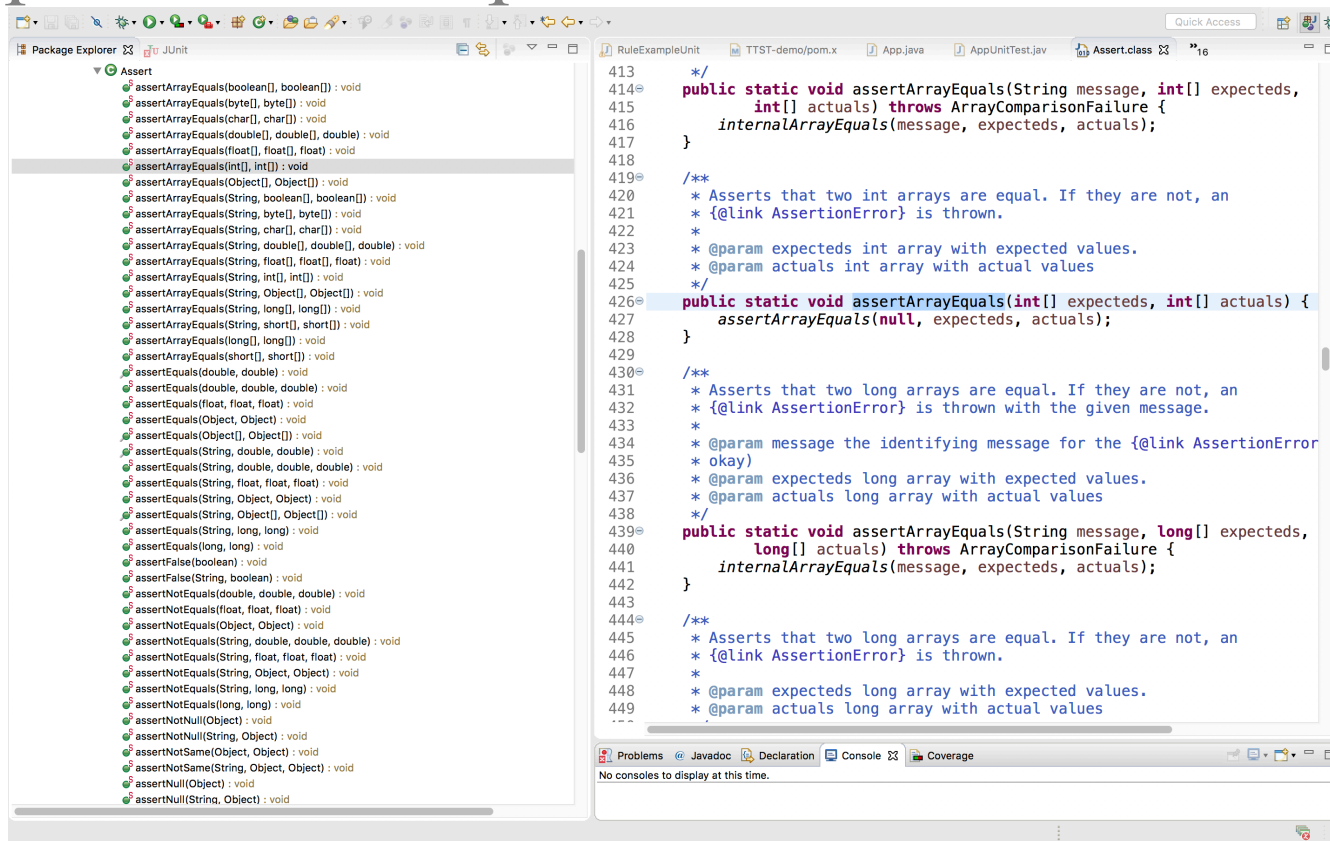
s, re-run Maven with the `-e` switch.
le full debug logging.
d possible solutions, please read the f
<https://maven.apache.org/display/MAVEN/ MojoFailureException>

MVN repository

- <https://mvnrepository.com>
- copy and past the Maven XML node in the dependency you selected
- A new dependency in the Maven folder appears

Maven dependencies

- Open Maven dependencies



Code used

- Calculator.java
- CalculatorUnitTest.java

First example: use @Test

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

Example

Here we are using the legacy with JUnit4

```
import static org.junit.Assert.assertEquals;  
import org.junit.Test;
```

@Test

```
public void evaluatesExpression() {  
    Calculator calculator = new Calculator();  
    int sum = calculator.evaluate("1+2+3");  
    assertEquals(6, sum);  
}
```

P/F criterion

Universität Bozen
Università di Bolzano
Università Ljedia de Bulsan

Oracle or expected output

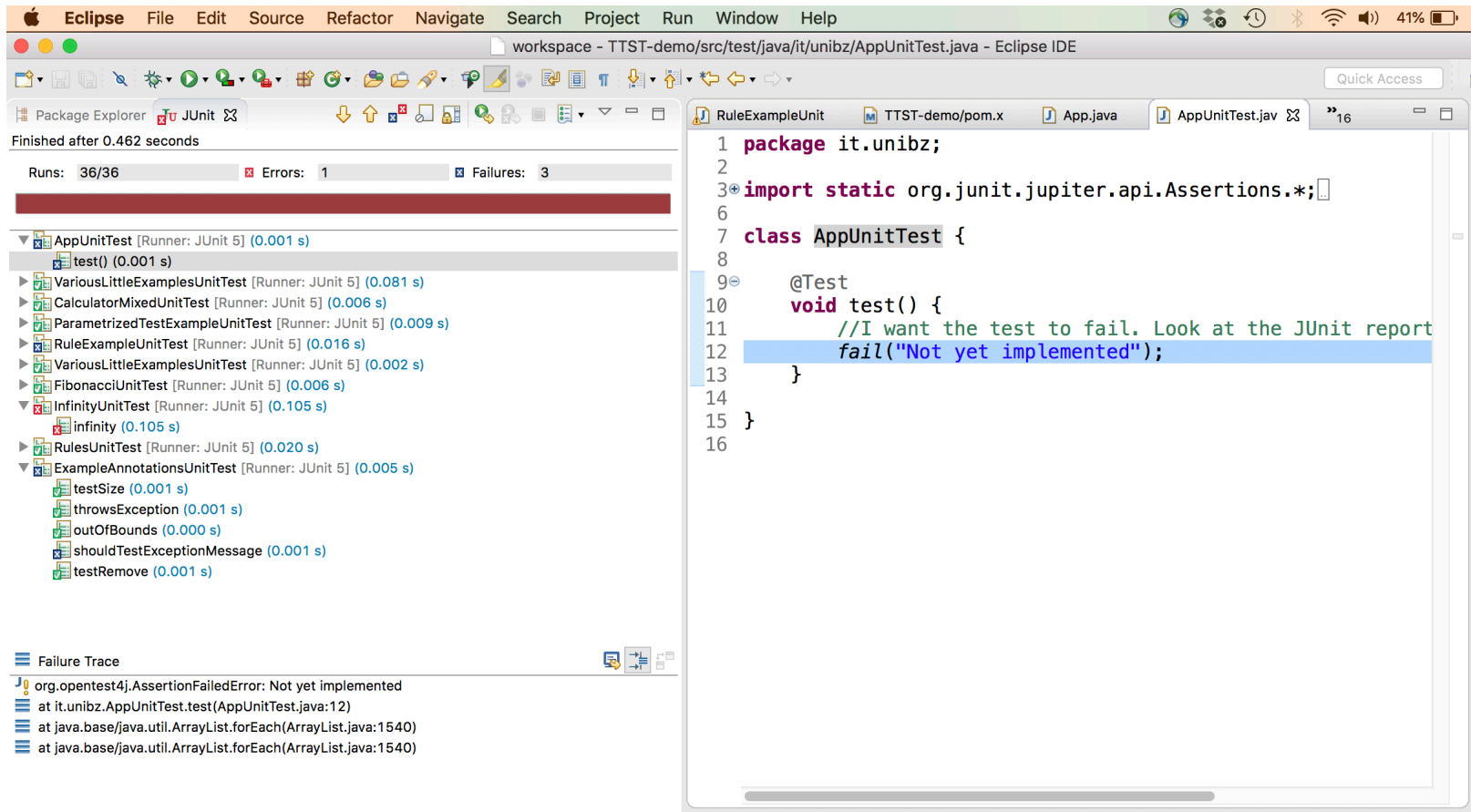
@Test

- It tags **public method that returns void** to run as a test method
 - JUnit first constructs a new instance of the class then invokes the annotated method
- Any *expected exceptions* thrown by the test will be *reported as a error*
- Any *bug* is reported as *failure*
- If *no exceptions/bugs* are thrown, the *test succeeds*

Code used

- App.java
- AppUnitTest.java

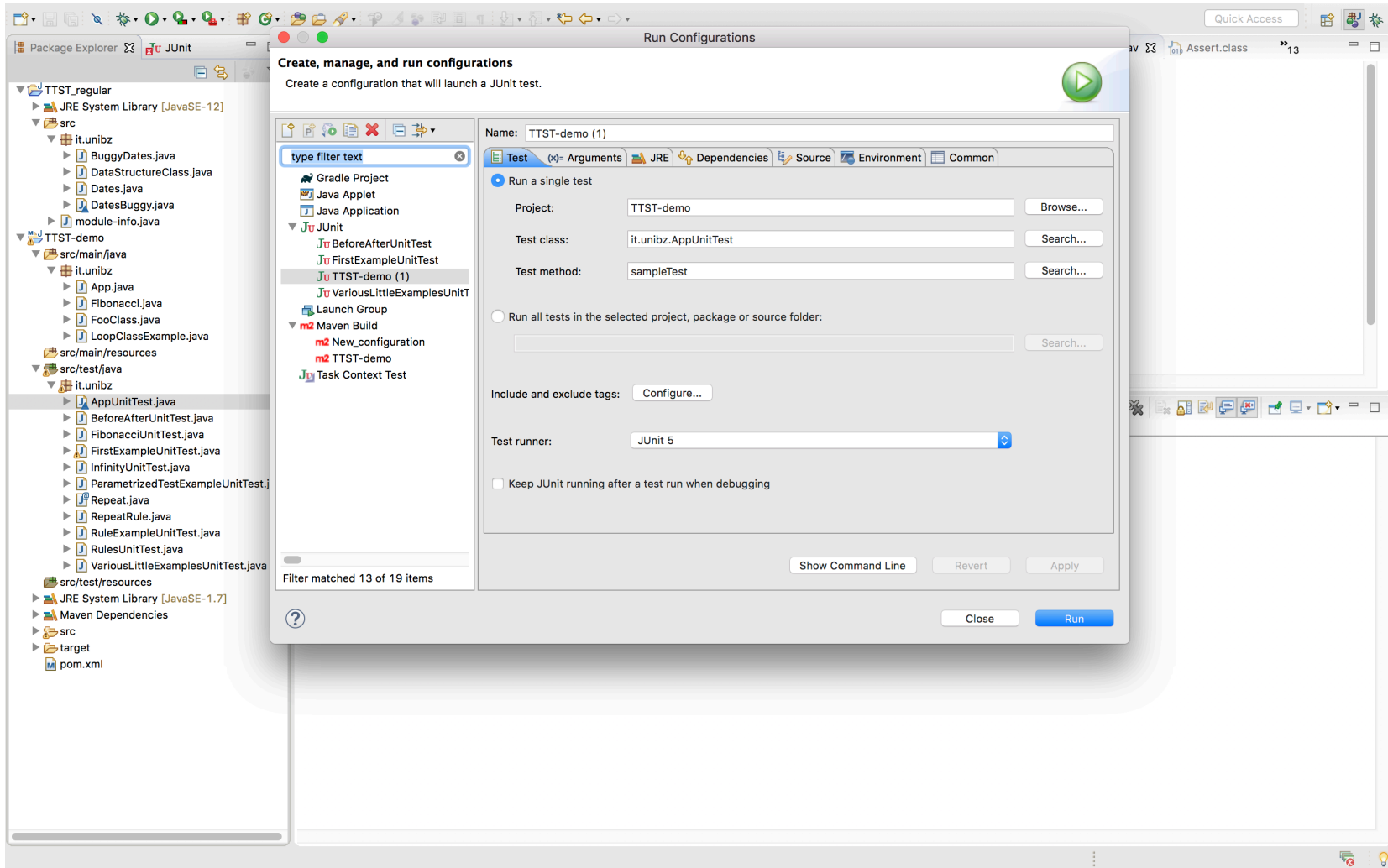
Run with JUnit configuration



The screenshot shows the Eclipse IDE interface. The top menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The title bar indicates the workspace is 'workspace - TTST-demo/src/test/java/it/unibz/AppUnitTest.java - Eclipse IDE'. The Package Explorer on the left shows a project structure with a 'JUnit' folder. Below it, a summary bar indicates 'Finished after 0.462 seconds', 'Runs: 36/36', 'Errors: 1', and 'Failures: 3'. A tree view shows the test results for 'AppUnitTest' and its sub-classes, with 'infinity' and 'ExampleAnnotationsUnitTest' showing failures. The 'Failure Trace' at the bottom left shows the error: 'org.opentest4j.AssertionFailedError: Not yet implemented' at 'it.unibz.AppUnitTest.test(AppUnitTest.java:12)'. The main editor window displays the source code for 'AppUnitTest.java', which includes a package declaration, an import for JUnit assertions, and a test method that calls 'fail("Not yet implemented");'.

```
1 package it.unibz;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5
6
7 class AppUnitTest {
8
9     @Test
10    void test() {
11        //I want the test to fail. Look at the JUnit report
12        fail("Not yet implemented");
13    }
14
15 }
16
```

Run a single test class or method



Run it with Maven

The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer displays the project structure for 'TTST_demo', including source files and test files. The 'Run Configurations' dialog is open, showing a configuration for 'OneClassMethodTest' with the goal '-Dtest=AppUnitTest#sampleTest test'. The console at the bottom shows the execution of the test, which fails with an AssertionError.

```
<terminated> OneClassMethodTest [Maven Build] /Library/Java/JavaVirtualMachines/jdk-12.0.1.jdk/Contents/Home/bin/java (Oct 14, 2019, 12:04:35 PM)
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running it.unibz.AppUnitTest
[ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.026 s <<< FAILURE! - in it.unibz.AppUnitTest
[ERROR] sampleTest Time elapsed: 0.02 s <<< FAILURE!
java.lang.AssertionError: sample
    at it.unibz.AppUnitTest.sampleTest(AppUnitTest.java:19)
```

Run it with Maven

- You need to set the goal
- `-DTest=<className>#<methodName> test`
- eventually use full path to package:
`<packageName>.<className>`

Code used

- FirstExampleUnitTest.java

Optional parameters of @Test

- **expected and timeout (JUnit4)**
- **expected:** checks a test method throws the expected exception
 - If it *does not throw* an exception or if it *throws a different* exception than the one declared, the *test fails (it returns an error)*
 - If *no expected exception parameter* and an exception is thrown, the *test fails (it returns an error)*

Example: test succeeds

```
@Test(expected=IndexOutOfBoundsException.class)
public void outOfBounds() {
    new ArrayList<Object>().get(0);
}
```

Package Explorer | JUnit

Finished after 0.039 seconds

Runs: 5/5 | Errors: 1 | Failures: 1

```

it.unibz.FirstExampleUnitTest [Runner: JUnit 4] (0.000 s)
  ✖ outOfBounds1 (0.000 s)
  ✔ throwsException (0.000 s)
  ✔ outOfBounds (0.000 s)
  ✔ evaluatesExpression (0.000 s)
  ✔ shouldTestExceptionMessage (0.000 s)

```

Failure Trace

```

java.lang.Exception: Unexpected exception, expected<java.lang.ArithmeticException> but was<java.lang.IndexOutOfBoundsException>
Caused by: java.lang.IndexOutOfBoundsException: Index 0 out of bounds for length 0
    at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
    at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
    at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:248)
    at java.base/java.util.Objects.checkIndex(Objects.java:372)
    at java.base/java.util.ArrayList.get(ArrayList.java:458)
    at it.unibz.FirstExampleUnitTest.outOfBounds1(FirstExampleUnitTest.java:37)
... 17 more

```

BeforeAfterUnit | FirstExampleUnit | AppUnitTest.jav | Assert.class | 17

```

1  b
2  7 import org.hamcrest.CoreMatchers;
3  8 import org.junit.Rule;
4  9 import org.junit.Test;
5  10 import org.junit.rules.ExpectedException;
6  11
7  12 public class FirstExampleUnitTest {
8  13     ArrayList<Integer> myList;
9  14
10 15     public int evaluate(String expression) {
11 16         int sum = 0;
12 17         for (String summand: expression.split("\\+"))
13 18             sum += Integer.valueOf(summand);
14 19         return sum;
15 20     }
16 21
17 22     @Test
18 23     public void evaluatesExpression() {
19 24         FirstExampleUnitTest calculator = new FirstExampleUnitTe
20 25         int sum = calculator.evaluate("1+2+3");
21 26         assertEquals(6, sum);
22 27
23 28     @Test(expected=IndexOutOfBoundsException.class)
24 29     // @Test
25 30     public void outOfBounds() {
26 31         new ArrayList<Integer>().get(0);
27 32
28 33         //Check whether the exception is the one expected
29 34         // @Test(expected=IndexOutOfBoundsException.class)
30 35     @Test(expected=ArithmeticException.class)
31 36     public void outOfBounds1() {
32 37         new ArrayList<Object>().get(0);
33 38         myList.get(1);
34 39     }
35 40
36 41     @Rule
37 42     public ExpectedException thrown = ExpectedException.none();
38 43

```

Problems | Javadoc | Declaration | Console | Coverage

<terminated> FirstExampleUnitTest [JUnit] /Library/Java/JavaVirtualMachines/jdk-12.0.1.jdk/Contents/Home/bin/java (Oct 14, 2019)

Optional parameters of @Test

- **timeout** causes a test to fail if it takes longer than a specified amount of clock time (measured in milliseconds)
- The test execution returns a time-out **error**

```
@Test(timeout=100)
public void infinity() {
    while(true);
}
```

Test Fixtures

- A **test fixture** is a fixed state of a set of objects used as a baseline for running tests
- JUnit provides annotations so that test classes can have fixture run **before or after** tests

Test Fixtures

- When a test class contains multiple methods to test, you can define **two void methods** that initialize and release respectively the common objects used in all tests
- You can call them *setup()* and *tearDown()*
- Use the tag `@BeforeAll` and `@AfterAll` to identify them

Example

```
ArrayList<Integer> myList;
```

```
@BeforeAll
```

```
public void initialize() {  
    myList= new ArrayList<Integer>();  
}
```

```
@Test
```

```
public void testSize() {  
    System.out.println(myList+" uses sizeList");  
}
```

Example

```
public class Example {
    List myList;
    @BeforeAll
    public void setUp() {
        myList= new ArrayList();
    }
    @Test
    public void testSize() {
        System.out.println{myList + "it uses sizeList"};
    }
    @Test
    public void testRemove() {
        System.out.println{"it uses removeList"};
    }
}
```

Annotations with JUnit 5

ANNOTATION	DESCRIPTION
<code>@BeforeEach</code>	The annotated method will be run before each test method in the test class.
<code>@AfterEach</code>	The annotated method will be run after each test method in the test class.
<code>@BeforeAll</code>	The annotated method will be run before all test methods in the test class. This method must be static.
<code>@AfterAll</code>	The annotated method will be run after all test methods in the test class. This method must be static.
<code>@Test</code>	It is used to mark a method as junit test
<code>@DisplayName</code>	Used to provide any custom display name for a test class or test method
<code>@Disable</code>	It is used to disable or ignore a test class or method from test suite.
<code>@Nested</code>	Used to create nested test classes
<code>@Tag</code>	Mark test methods or test classes with tags for test discovering and filtering
<code>@TestFactory</code>	Mark a method is a test factory for dynamic tests

Code Used

- ExampleUnitTestSuite.java
- MySecondClassUnitTest.java
- MyFirstClassUnitTest.java
- Test.java

@SuiteClasses

- The @SuiteClasses annotation specifies the classes to be executed when a class annotated with @RunWith(Suite.class) is run

Create Test Suite and Test Runner

- Step 1) Create a simple test class (e.g. MyFirstClassTest) and add a method annotated with `@Test`
- Step 2) Create another test class to add (e.g. MySecondClassTest) and create a method annotated with `@Test`

Create Test Suite and Test Runner

- Step 3) To create a testSuite you need to first annotate the class with `@RunWith(Suite.class)` and `@SuiteClasses(class1.class, class2.class, ...)`

```
3+ import org.junit.runner.RunWith;
6
7 @RunWith(Suite.class)
8 @SuiteClasses({ MyFirstClassTest.class, MySecondClassTest.class })
9 public class TestSuiteExample {
10
11 //Code goes Here...
12
13 }
```

Imports

```
import org.junit.runner.RunWith;  
import org.junit.runners.Suite;  
import org.junit.runners.Suite.SuiteClasses;
```

```
@RunWith(Suite.class)
```

```
@SuiteClasses(ATest.class, BTest.class, CTest.class)
```

```
public class ABCSuite {  
}
```

Test Runner

- Step 4) Create a Test Runner class to run the test suite

```
3+ import org.junit.runner.JUnitCore;
6
7 public class Test {
8-   public static void main(String[] args) {
9       Result result = JUnitCore.runClasses(TestSuiteExample.class);
10      for (Failure failure : result.getFailures()) {
11          System.out.println(failure.toString());
12      }
13      System.out.println(result.wasSuccessful());
14  }
15 }
```

Used code

- `ParametrizedTestExampleUnitTest.java`

Annotating a class with `@RunWith`

- When a class is annotated with `@RunWith` or extends a class annotated with `@RunWith`, JUnit will invoke the class it references to run the tests in that class instead of the runner built into JUnit

Example

```
package it.unibz;

/*
 * Simple class that uses the Parameterized runner. It runs the test 10 times.
 * No expected value is foreseen (second entry of Object has length 0).
 */
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

@RunWith(Parameterized.class)
public class ParametrizedTestExampleUnitTest {

    @Parameterized.Parameters
    public static Object[ ][ ] data() {
        return new Object[10][0];
    }

    public void runTenTimes() {

    }

    @Test
    public void runsTenTimesTest() {
        System.out.println("run");
    }

}
```

Parameterized tests w. JUnit5

- In order to use parameterized tests in JUnit5 you need to add a dependency on the *junit-jupiter-params* artifact
- <https://www.baeldung.com/parameterized-tests-junit-5>

Rule

```
package it.unibz;
/*
 * Simple rule on how to create a new folder any time the test method is executed
 */

import static org.junit.Assert.assertTrue;
import java.io.File;
import java.io.IOException;

import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TemporaryFolder;
public class RulesUnitTest {

    @Rule
    public TemporaryFolder tempFolder = new TemporaryFolder();

    @Test
    public void testRule() throws IOException {
        File newFolder = tempFolder.newFolder("Temp Folder");
        assertTrue(newFolder.exists());
    }
}
```

Code used

- ExampleBeforeEachUnitTest.java

Display name

```
@Test
@DisplayName("Hello World")
void test01(){
    System.out.println("Test Hello World is Invoked");
}
```

```
@DisplayName("My Test Name")
@RepeatedTest(value = 5, name = "{displayName} - repetition {currentRepetition} of {totalRepetitions}")
void addNumber(TestInfo testInfo){
    System.out.println("Hello World");
}
```

Disable

```
// a method
@Test
@Disabled("Do not run this test")
void test01(){
    System.out.println("Hello World");
}

// a test class

@Disabled
public class AppTest{
    @Test
    void test01(){
        System.out.println("Hello World");
    }
}
```

Used code

- `VariousLittleExamplesUnitTest.java`

Used code

- Use of `@Parameterized` runner
- `FooClass.java`
- `Foo1UnitTest.java`
- `FooUnitTest.java`

- `Fibonacci.java`
- `FibonacciUnitTest.java`

Exercise

- Design a unit test with parametrized and params
- to test a Fibonacci function

```
public class Fibonacci {  
    public static int compute(int n){  
        int result;  
        if (n<=1){  
            result=n;  
        }else{  
            result = compute(n-1)+compute(n-2);  
        }  
        return result;  
    }  
}
```


Fitnessse and unit tests

- Fitnessse is a black box instrument
 - It shows I/O per method
 - It is a way to automate Acceptance Test!
 - It requires some code development beforehand
 - Developers need to write fixtures to run the decision tables, but
 - Fixtures are a thin form of “drivers” implemented by delegating the behavior of the original code; they do not contain any test logic (e.g., no assertions, no annotation, no dynamic test)
 - Based on the decision tables of Fitnessses you can drive the implementation of the Unit Tests (as a black-box tool that is used to design the white box test)

Exercise

- Build test suite for class Auction