Unit Testing with JUnit 5

Get more out of your unit tests with Hamcrest and Mockito!

Luigi Libero Lucio STARACE
luigiliberolucio.starace@unina.it

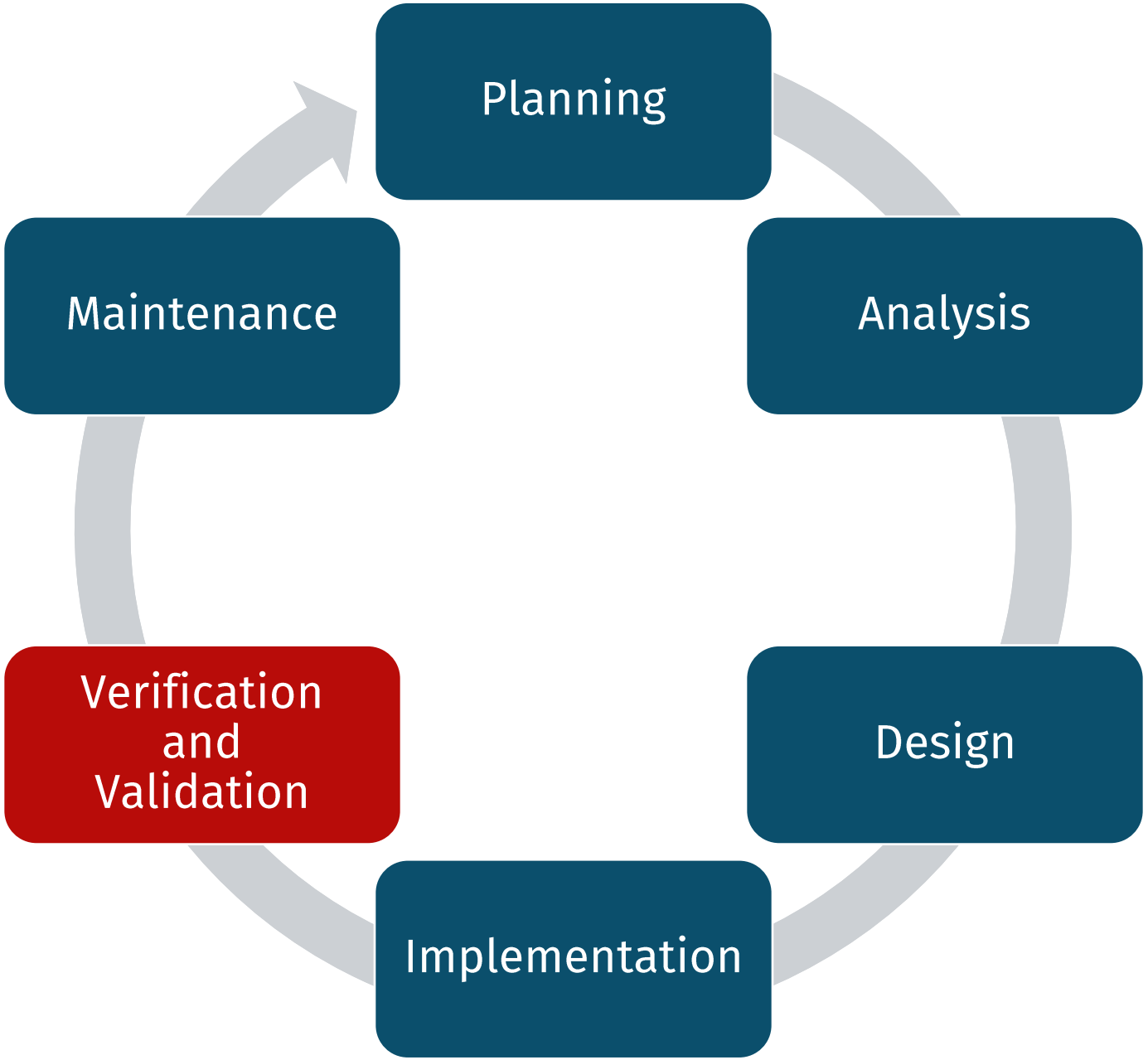
April 28, 2020
University of Naples, Federico II

What’s the plan

1. Unit testing
2. JUnit 5
3. Assertion Frameworks
4. Testing in Isolation

Unit Testing

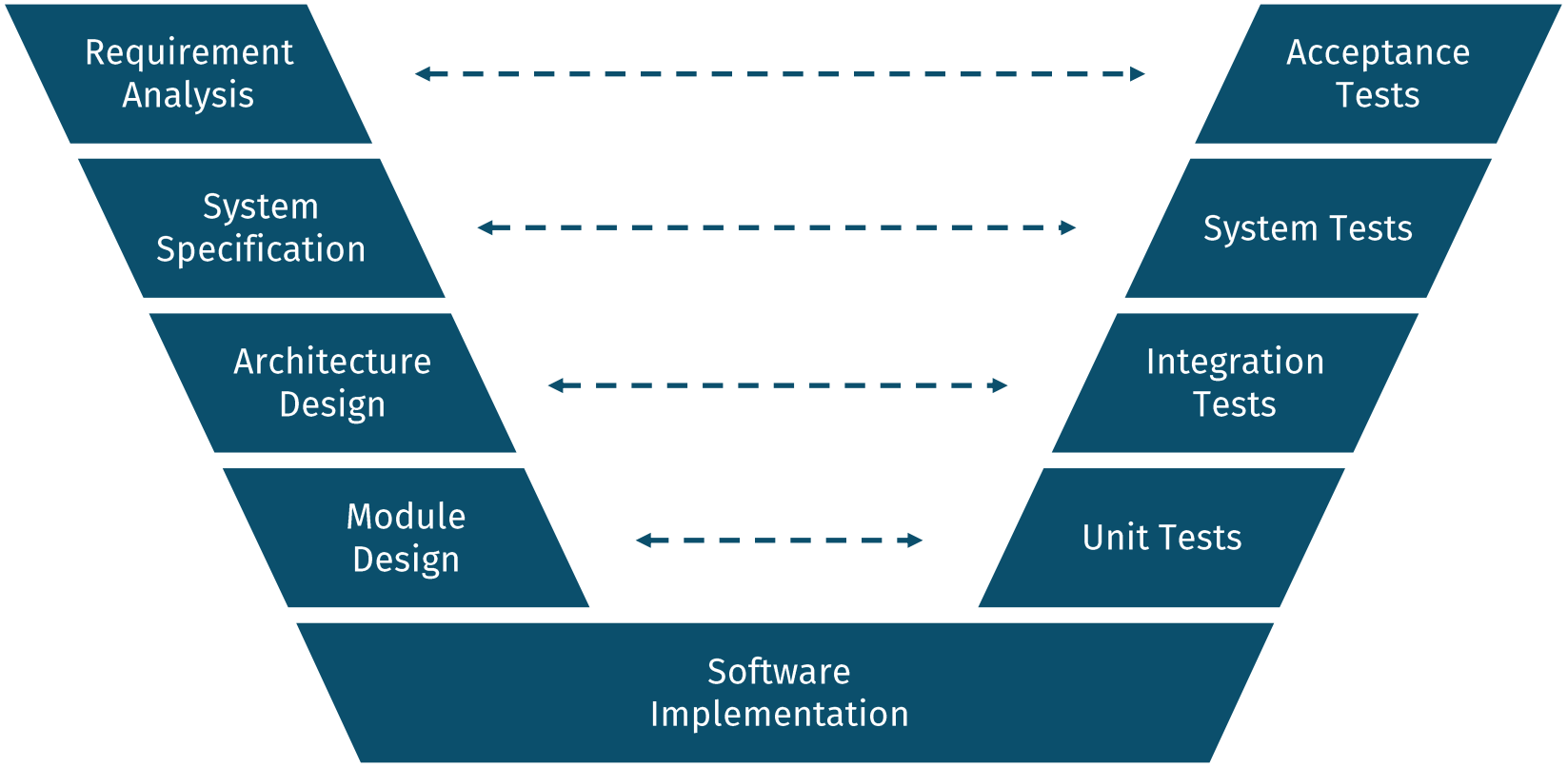
The Software Life-cycle



Software Verification

- How can we increase the confidence that our software works as intended?
- Static verification:
  - No code execution involved;
  - Code reviews, inspections with checklists, static analysis;
- Dynamic verification:
  - Testing!

The V-Model

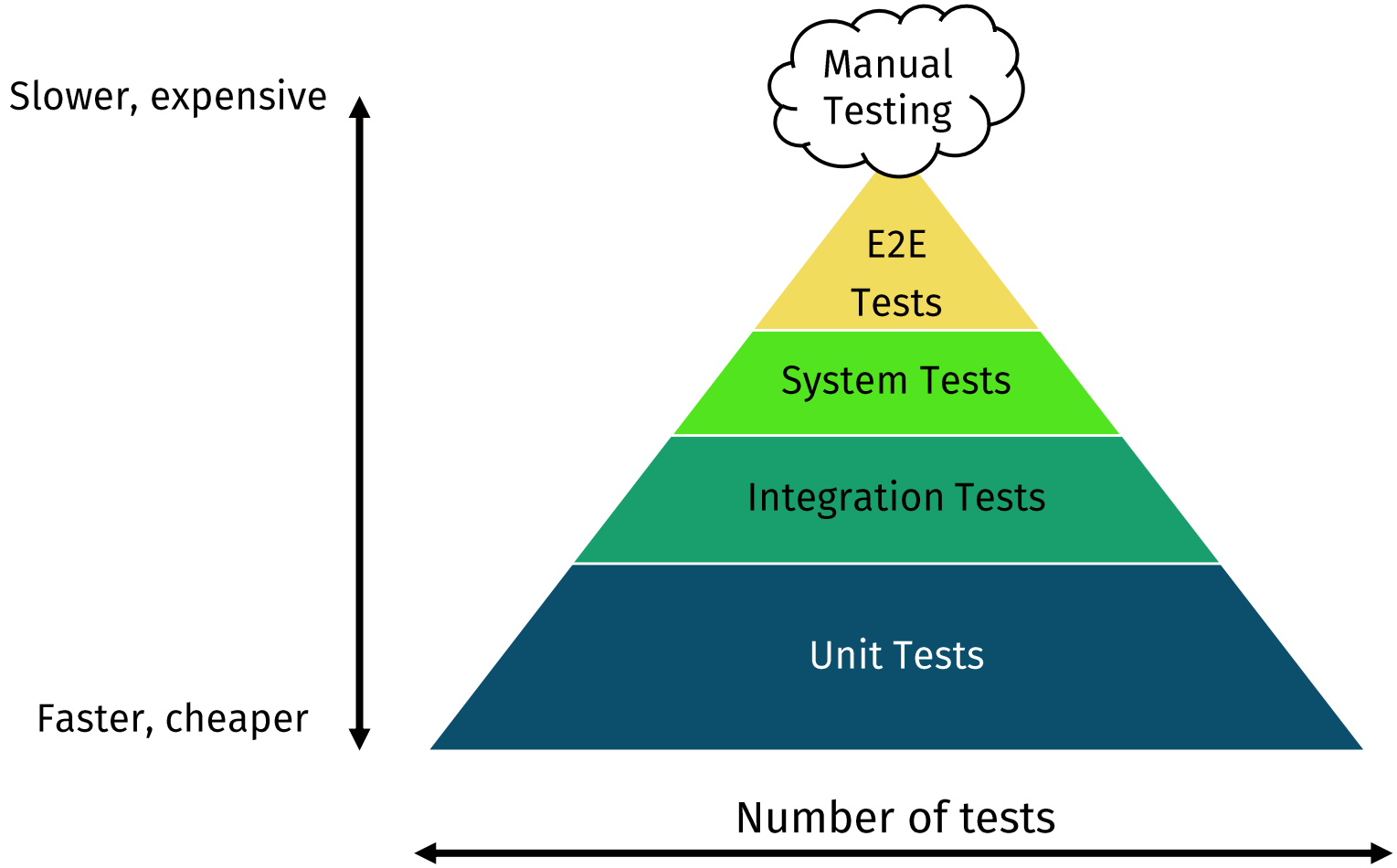

Requirement Analysis
System Specification
Architecture Design
Module Design
Software Implementation
Unit Tests
Integration Tests
System Tests
Acceptance Tests

Software Testing

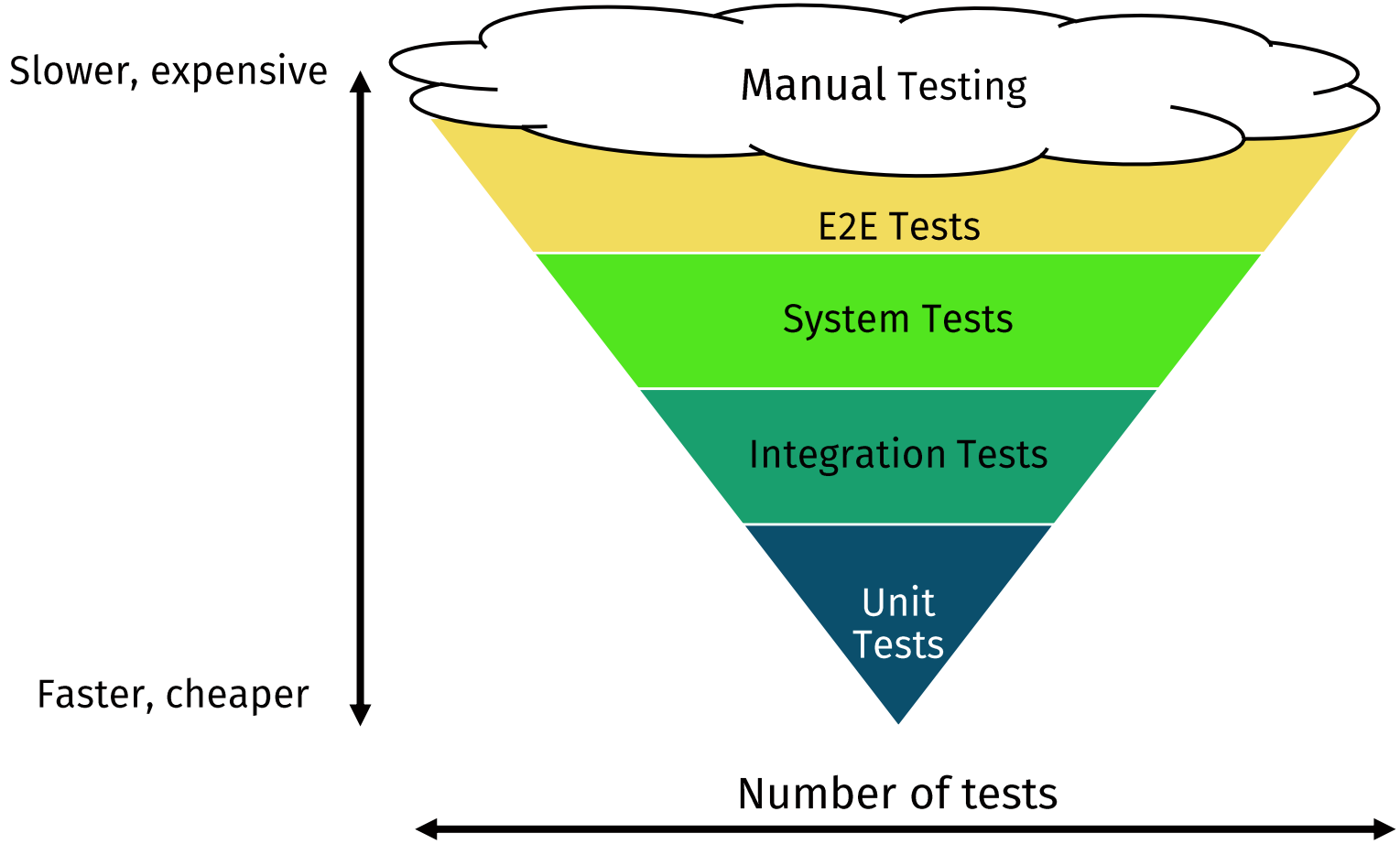
Can be performed at different levels:

- Unit testing
  - Tests a single *unit* (e.g. a class, or a method)
- Integration testing
  - Checking that different units work together as intended
- System testing
  - Targets the system as a whole, against its specification
- Acceptance – End-to-End (E2E) testing
  - Is the product acceptable for delivery?

The Testing Pyramid



The Testing Ice-cream Cone Anti-pattern



What's Unit Testing?

A method by which individual *units* of source code are tested to determine whether they work as intended.

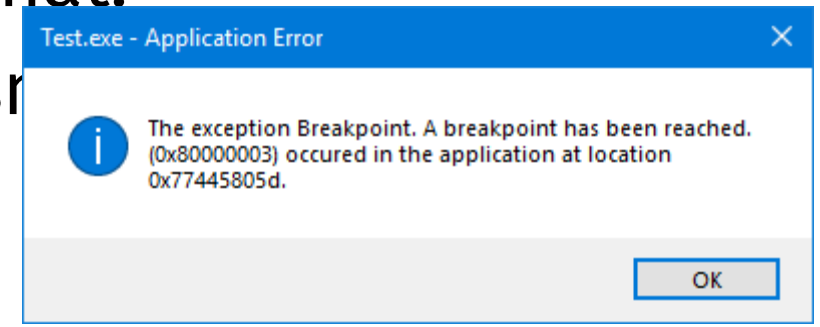
- So, what's a source code unit?
  - In object-oriented design, usually it's a class;
  - In procedural and functional approaches, it might be a single function;
  - The developer team decides what makes sense to be a unit.
- Unit tests can be organized in:
  - **Test cases**, which verify a certain behaviour of a single unit;
  - **Test suites**, i.e. group of test cases insisting on related behaviours.

Reasons to Unit Test

- Early error detection;
- Supports maintenance (helps avoiding regressions);
- Improves design;
- Helps determining specifications;
- Product documentation.

Excuses not to Unit Test

- I *never* make mistakes!
- Ain’t got time for that!
- Management doesn



Properties of a good Unit Test

- It is consistent, repeatable;
- It’s fast;
- It’s ***clean***!

> *What makes a clean test? Three things. Readability, readability, and readability. Readability is perhaps even more important in unit tests than it is in production code.*
> *What makes tests readable? The same thing that makes all code readable: clarity, simplicity, and density of expression. In a test you want to say a lot with as few expressions as possible.*

— Robert Martin, Clean Code


JUnit
5

JUnit

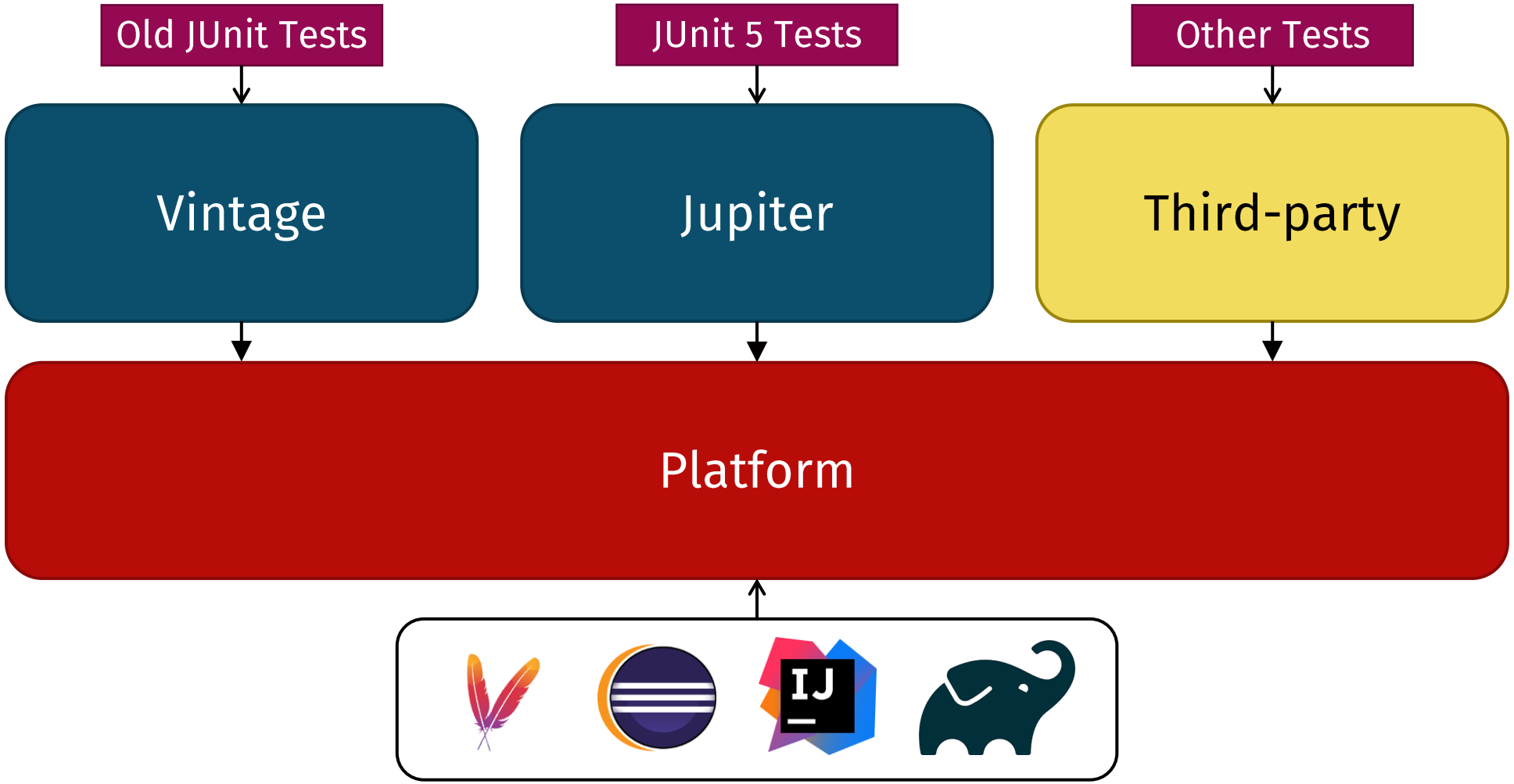
- Open-source framework for writing and running tests in Java;
- Originally written by *Erich Gamma* and *Kent Back;*
- One of many frameworks in the xUnit family:
  - nUnit, RUnit, PHPUnit, CppUnit, and more.
- Used in 30.7% of the top 10,000 Java projects on GitHub[1].

[1] https://blog.overops.com/githubs-10000-most-popular-java-projects-here-are-the-top-libraries-they-use/

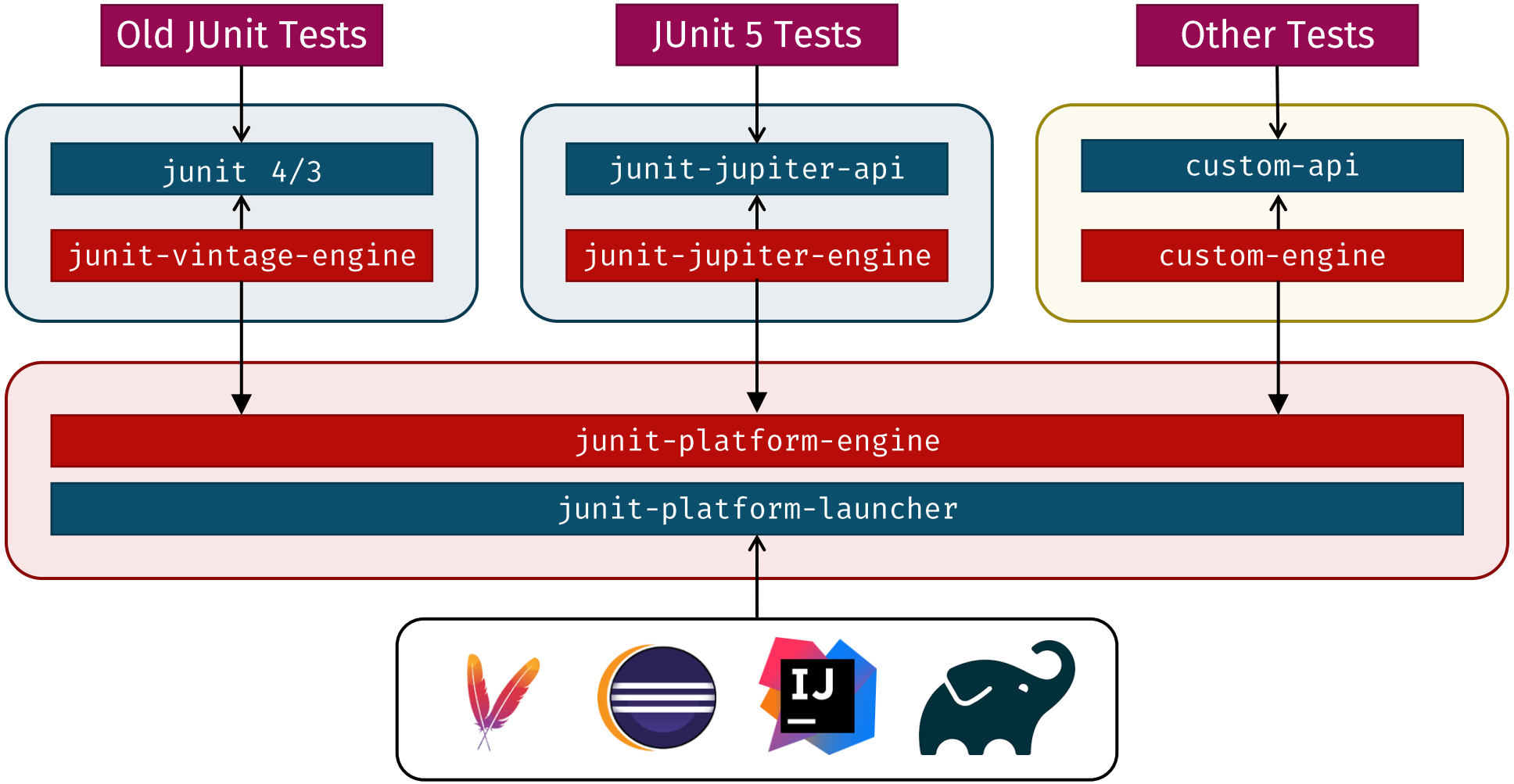
On JUnit

> *Never in the field of software development have so many owed so much to so few lines of code.*
>
> — Martin Fowler

The JUnit 5 Architecture



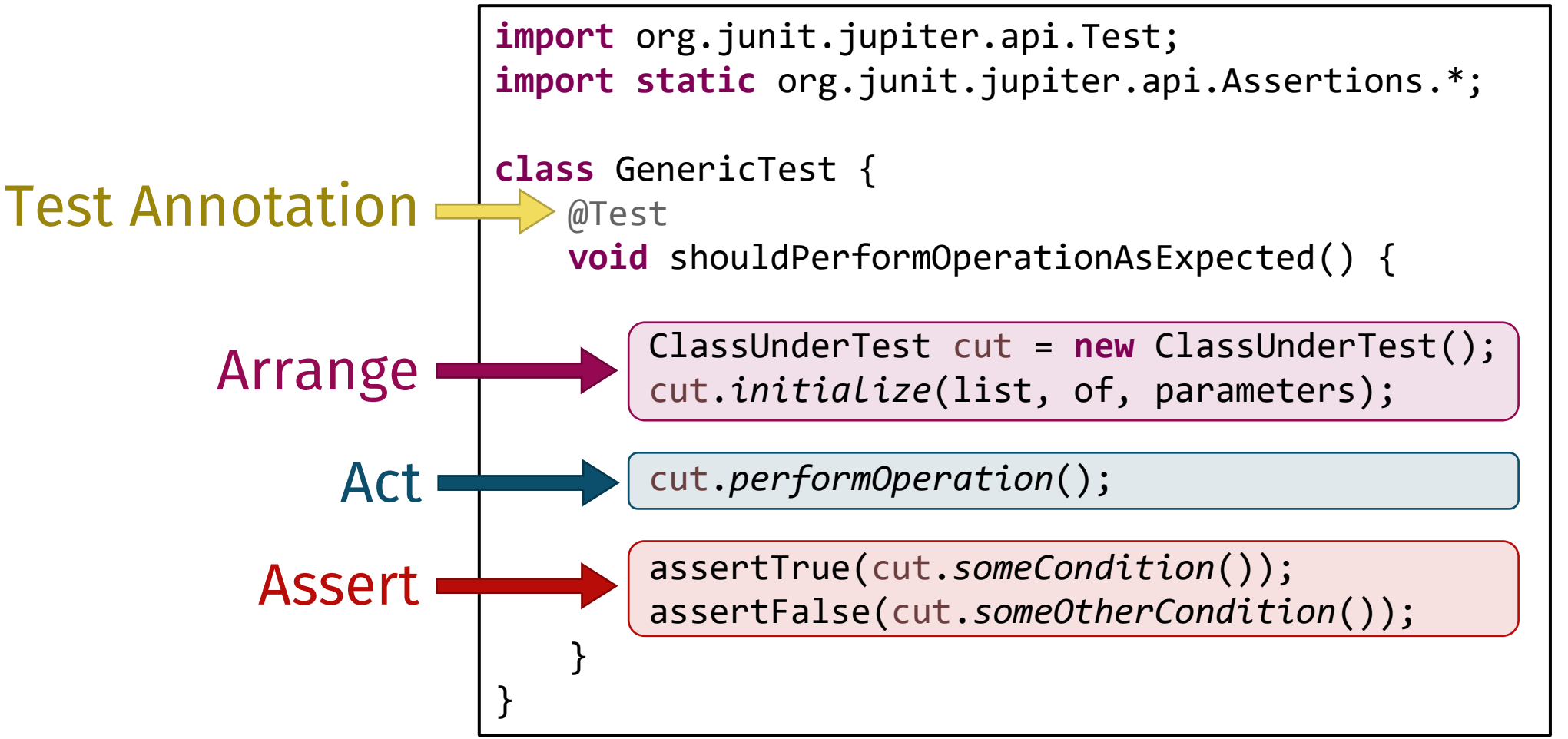
The JUnit 5 Architecture



Installing JUnit 5

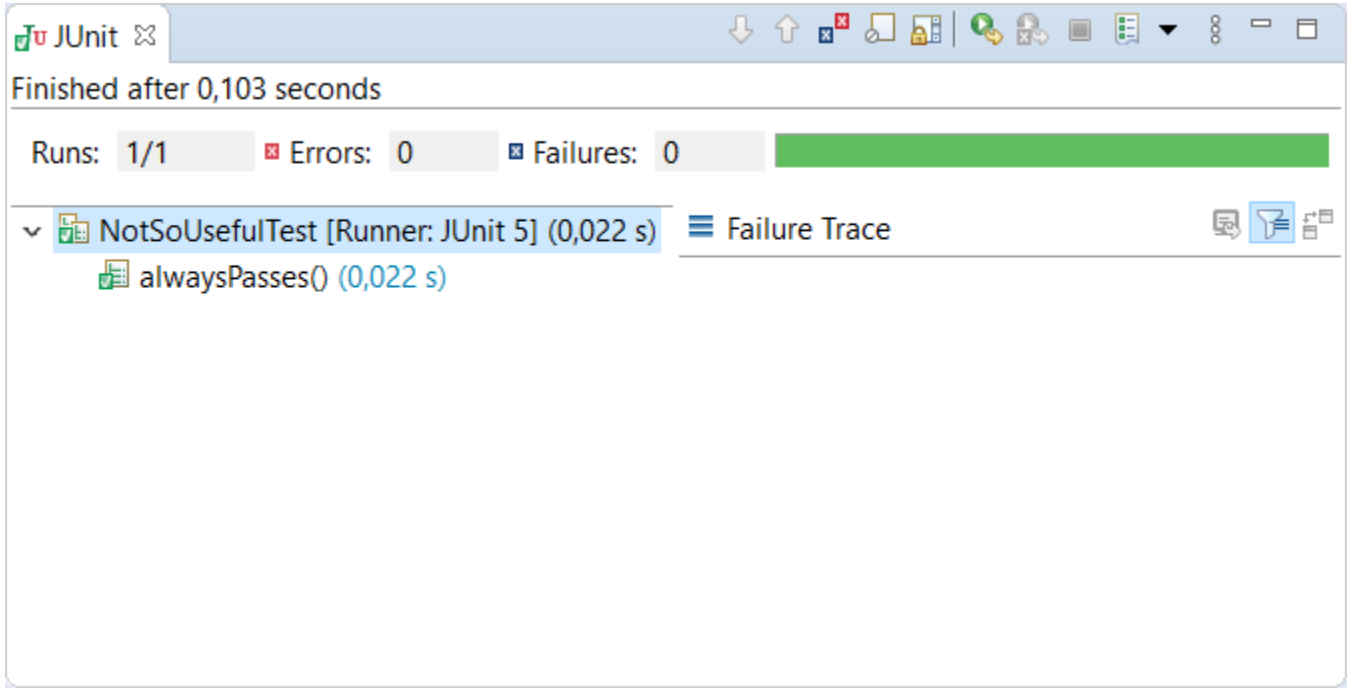
- Out-of-the-box support in many IDEs:
  - Eclipse, IntelliJ IDEA, NetBeans, VS Code
- Through build-tools like Maven, Ant, Gradle;
- Manually downloading the required artifacts.

JUnit 5 – Arrange, Act, Assert



How do we run the tests? (1 of 3)

- In Eclipse, right click on the Test class, then *Run as* > *JUnit test*



How do we run the tests? (2 of 3)

- JUnit test can be ran also from build tools
  - E.g., for Maven, check out the official `maven-surefire-plugin`
  - Useful in CI/CD;

How do we run the tests? (2 of 3)

```
[INFO] --- maven-surefire-plugin:2.22.2:test (default-test) @ test ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running test.NotSoUsefulTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.016 s - in test.NotSoUsefulTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.386 s
[INFO] Finished at: 2020-04-11T18:46:42+02:00
[INFO] ------------------------------------------------------------------------
```

How do we run the tests? (3 of 3)

- The `ConsoleLauncher` is a command-line application to run tests.
- You can download it as an executable jar with all dependencies included.

How do we run the tests? (3 of 3)

```
$> java -jar junit-platform-console-standalone-1.6.2.jar -cp build/classes/java/test --scan-classpath

└─ JUnit Jupiter
   └─ NotSoUsefulTest
      └─ alwaysPasses() ✔

Test run finished after 24 ms
[         0 containers found      ]
[         0 containers skipped    ]
[         0 containers started    ]
[         0 containers aborted    ]
[         0 containers successful ]
[         0 containers failed     ]
[         1 tests found           ]
[         0 tests skipped         ]
[         1 tests started         ]
[         0 tests aborted         ]
[         1 tests successful      ]
[         0 tests failed          ]
```

JUnit 5 – Lifecycle Annotations

- `@Test`
  - Used to mark test methods;
- `@BeforeAll / @AfterAll`
  - Denotes methods that should be executed before / after all test methods in the test class;
- `@BeforeEach / @AfterEach`
  - Denotes methods that should be executed before / after each test methods;

JUnit 5 – Test Instance Lifecycle

- By default, JUnit creates a new instance of each test class before running each test method.
- To execute all test cases on the same test instance, you can annotate the test class with:

```
@TestInstance(Lifecycle.PER_CLASS)
```

JUnit 5 – Assertions

- Assertions are utility methods we can use to declaratively assert conditions in tests;
- They're defined as static methods of the `Assertions` class;
- Failed assertions usually throw `AssertionFailedError`.

JUnit 5 – Assertion examples

`assertTrue()` and `assertFalse()`

```
int number = 0; boolean condition = true; String message = "ABCDEF";

// passes
assertTrue(condition);

// fails, with custom error message
assertFalse(number < 1, "Number shouldn't be less than 1.");

// fails, with error message supplier
assertTrue(message.contains("Z"),() ->
    String.format("Message (%s) should contain \"Z\"",message)
);
```

JUnit 5 – Assertion examples

`Assert[Not]Equals()` and `assert[Not]Same()`

```
Sheep dolly = new Sheep("Dolly");
Sheep otherDolly = dolly.clone();

assertEquals(dolly, otherDolly);
assertNotSame(dolly, otherDolly);
```

JUnit 5 – Assertion examples

- assertNull() and assertNotNull()
- assertArrayEquals()
- assertIterableEquals()
- assertTimeout() and assertTimeoutPreemptively()
- assertThrows() and assertDoesNotThrow()

```
assertTimeoutPreemptively(Duration.ofMillis(300), () -> { object.doStuff(); });

assertThrows(RuntimeException.class, ()-> {
    throw new RuntimeException();
});
```

JUnit 5 – Assertion examples

AssertAll()

```
Person grandPa = getRandomGrandPa();

assertAll(
    () -> assertNotNull(grandPa.getName()    , "Name should not be null"),
    () -> assertTrue   (grandPa.getAge() > 50, "Should be older than 50"),
    () -> assertEquals ("M", grandPa.getSex(), "Should be male")
);

// Same as this? No! AssertAll always executes all the assertions!
assertNotNull(grandPa.getName()    , "Name should not be null");
assertTrue   (grandPa.getAge() > 50, "Should be older than 50");
assertEquals ("M", grandPa.getSex(), "Should be male");
```

JUnit 5 – Test Naming

Test classes and methods can declare custom names with the `@DisplayName` annotation:

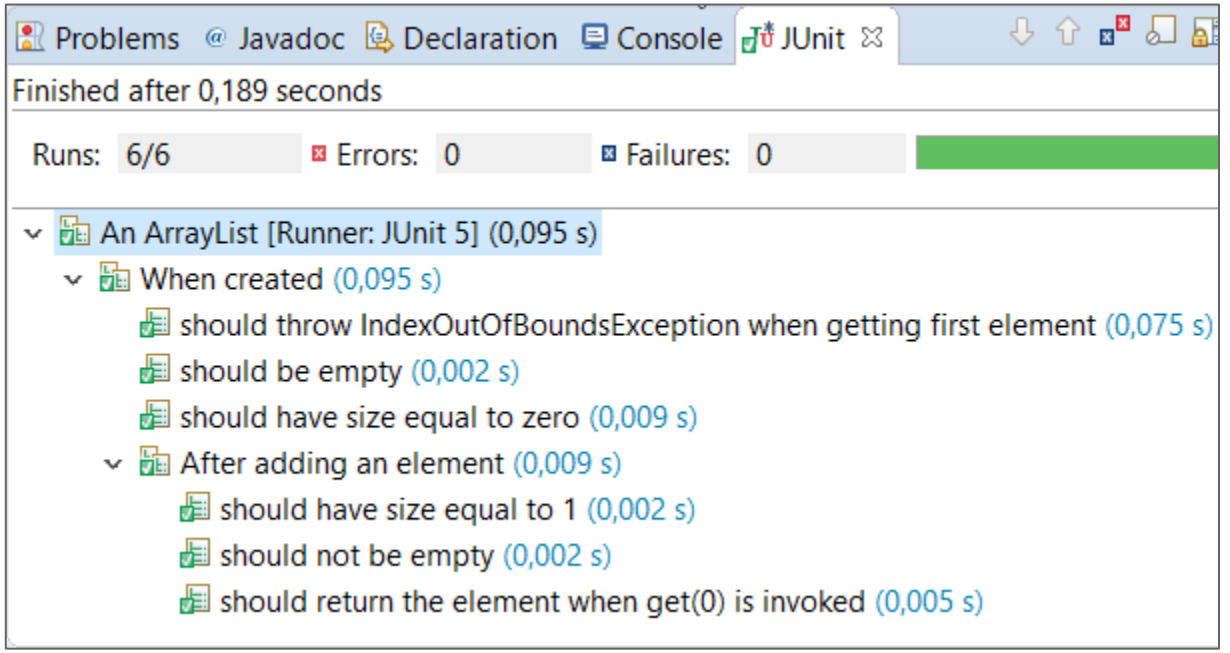
- Can contain spaces;
- Special characters;
- And even emojis (some men just want to watch the world burn).

```java
@DisplayName("A special test case")
class DisplayNameDemo {

    @Test
    @DisplayName("Custom name")
    void testWithCustomName() {}

    @Test
    @DisplayName("ヽ°□°) ノ")
    void testWithSpecialName() {}

    @Test
    @DisplayName("😱")
    void testWithEmojiName() {}
}
```

JUnit 5 – Nested Test Classes

With `@Nested` test classes, you can express relationship among different group of tests and better organize your test code.

```
@DisplayName("An ArrayList")
class NamingExample {
  @Nested
  @DisplayName("When created")
  public class TestEmptyArrayList {
    @Test
    @DisplayName("should be empty")
    void shouldBeEmpty() {}
  }
}
```



JUnit 5 – Parametrized Tests

Sometimes, test cases differ only for their inputs and expected results.

How can we make our code **DRY**er?

```
@Test
void shouldSumTwoPositiveNumbers(){
    int a = 4, b= 6, expected = 10;
    int actual = Calculator.sum(a, b);
    assertEquals(actual, expected);
}

@Test
void shouldSumTwoNegativeNumbers(){
    int a = -3, b= -5, expected = -8;
    int actual = Calculator.sum(a, b);
    assertEquals(actual, expected);
}

//and so on...
```

JUnit 5 – Parametrized Tests

This is more compact, but…

- Not very readable!
- Stops at the first failed assertion!
- It could be hard to find which assertion failed!

```
@Test
void shouldSumCorrectly(){
    int[][] inputs = {{4,6,10},{-3,-5,-8}};
    for(int[] input : inputs) {
        int a = input[0], b = input[1],
            expected = input[2];
        int actual = Calculator.sum(a, b);
        assertEquals(actual, expected);
    }
}
```

JUnit 5 – Parametrized Tests

This evaluates all assertions, but…

- Still not very readable!
- Still, It could be hard to find which assertion failed!

```
@Test
void shouldSumCorrectlyV2(){
    int[][] inputs = {{4,6,10},{-3,-5,-8}};
    List<Executable> assertions =
        new ArrayList<Executable>();
    for(int[] input : inputs) {
        int a = input[0], b = input[1],
            expected = input[2];
        int actual = Calculator.sum(a, b);
        assertions.add(() ->
            assertEquals(actual, expected));
    }
    assertAll(assertions);
}
```

JUnit 5 – Parametrized Tests

Parametrized tests make running a test multiple times with different inputs possible.

A parametrized test method

- Is annotated with `@ParametrizedTest`
- Takes at least one argument
- Is provided with at least one data source

JUnit will run the parametrized method with each of the inputs provided by the data sources.

JUnit 5 – Parametrized Test Data Sources

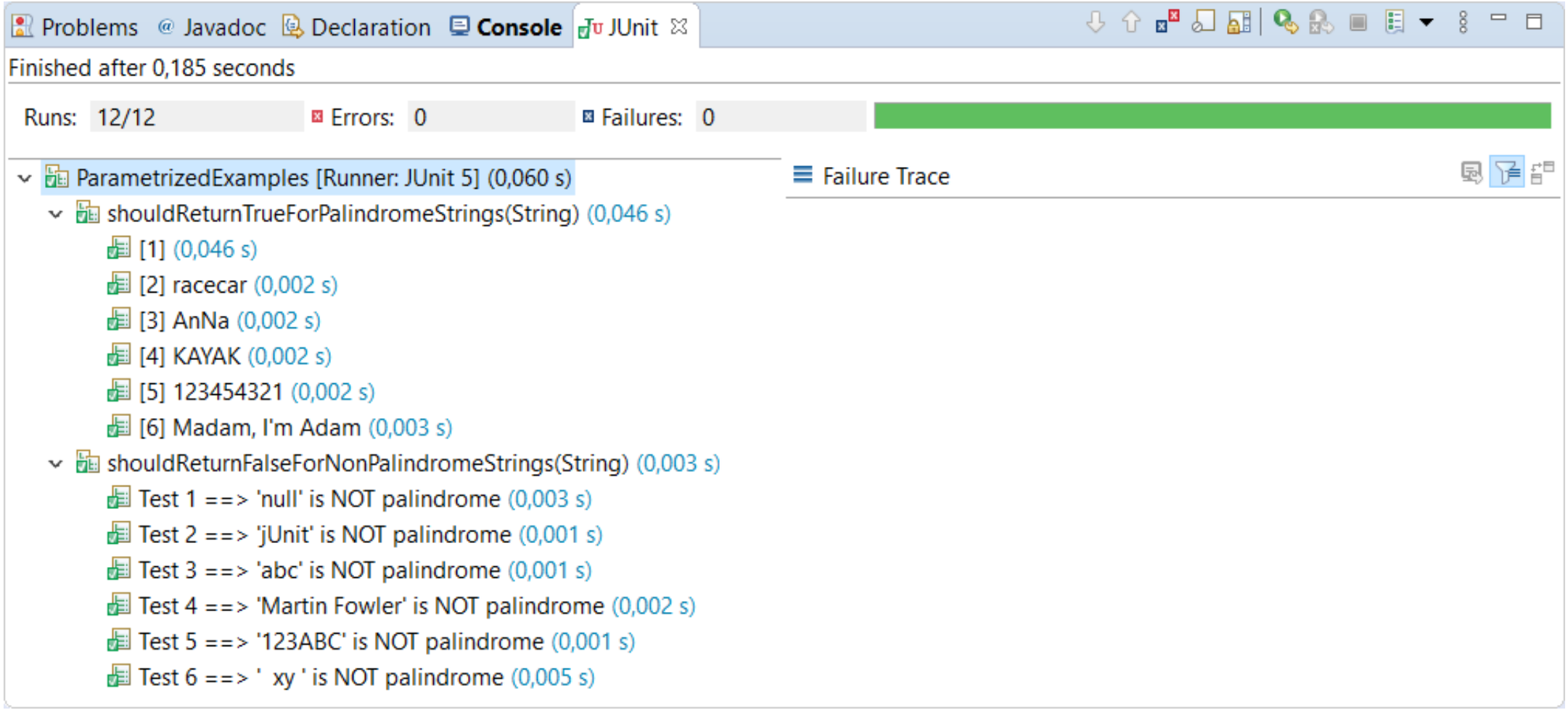
Data sources in Junit 5 include:

- `@NullSource`: provides one single null argument. Cannot be used with primitive parameters;
- `@EmptySource`: provides a single empty argument for a single parameter of type String, Collections, or primitive array.
- `@ValueSource`: lets you specify an array of literal values to be used as arguments parameterized test invocation;
- `@MethodSource`: lets you specify a method that provides a `Stream<Arguments>` to be used as inputs.

JUnit 5 – Parametrized Test Examples

```
@ParameterizedTest
@EmptySource
@ValueSource(strings = {"racecar", "AnNa", "KAYAK", "123454321", "Madam, I'm Adam"})
void shouldReturnTrueForPalindromeStrings(String string) {
    assertTrue(PalindromeChecker.isPalindrome(string));
}

@ParameterizedTest(name = "Test {index} ==> ''{0}'' is NOT palindrome")
@NullSource
@ValueSource(strings = {"jUnit", "abc", "Martin Fowler","123ABC","  xy "})
void shouldReturnFalseForNonPalindromeStrings(String string) {
    assertFalse(PalindromeChecker.isPalindrome(string));
}
```

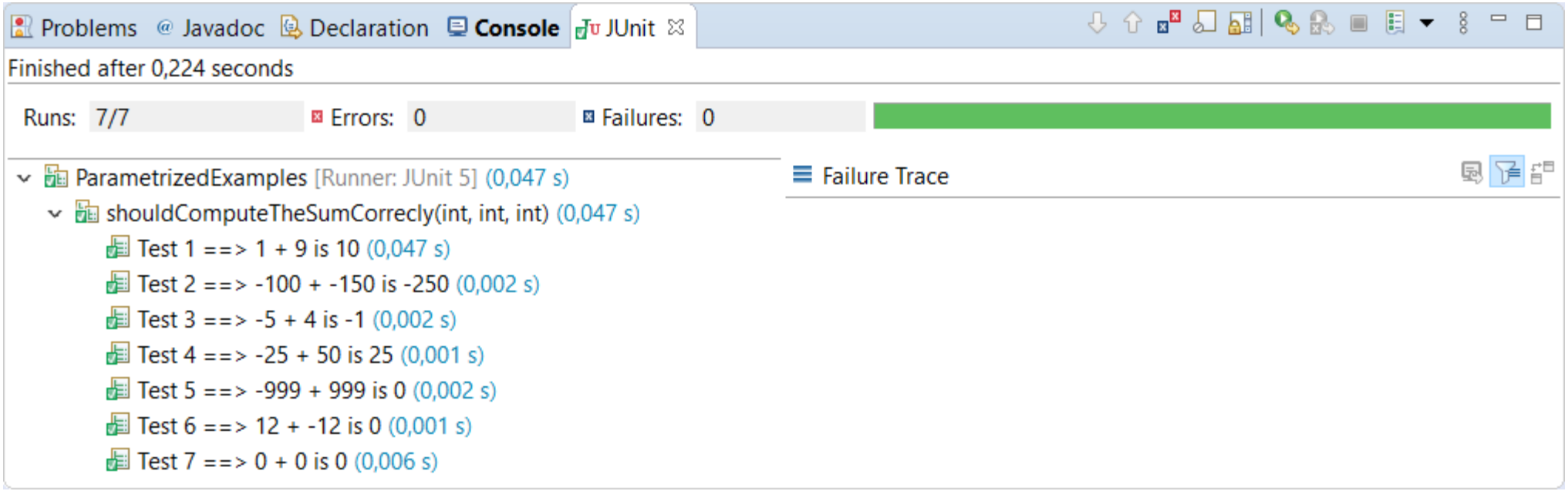
JUnit 5 – Parametrized Test Examples



JUnit 5 – Parametrized Test Examples

```
@ParameterizedTest(name = "Test {index} ==> {0} + {1} is {2}")
@MethodSource("addInputProvider")
void shouldComputeTheSumCorrecly(int a, int b, int expected) {
    int actual = Calculator.sum(a, b);
    assertEquals(expected, actual);
}

static Stream<Arguments> addInputProvider() {
    return Stream.of(
        Arguments.of(1,9,10),
        // more arguments..
        Arguments.of(0,0,0)
    );
}
```

JUnit 5 – Parametrized Test Examples



JUnit 5 – Conditional Test Execution

If we don’t want a test method to run, we can disable it.
Disabled test methods will be skipped by the JUnit runner.

A test method can be disabled *unconditionally*

```
@Test
@Disabled
void disabledTest() {}

@Test
@Disabled("Disabled until F23 is implemented")
void disabledTest2() {}
```

JUnit 5 – Conditional Test Execution

If we don’t want a test method to run, we can disable it.
Disabled test methods will be skipped by the JUnit runner.

A test method can be disabled *based on conditions*

```
@Test
@DisabledOnOs(OS.WINDOWS)
void testDisabledOnWindows() {}

@Test
@EnabledOnOs({OS.WINDOWS, OS.MAC})
void testEnabledOnWindowsAndMac() {}
```

JUnit 5 – Conditional Test Execution

If we don’t want a test method to run, we can disable it.
Disabled test methods will be skipped by the JUnit runner.

A test method can be disabled *based on conditions*

```
@Test
@EnabledForJreRange(min = JAVA_8)
void fromJava8toCurrentJavaFeatureNumber() {}

@Test
@DisabledOnJre(JAVA_9)
void notOnJava9() {}
```

JUnit 5 – Assumptions

- Assumptions are utility methods we can use to conditionally execute tests (or parts of tests);
- They’re defined as static methods of the `Assumptions` class;
- If an assumption fails, the current test (or part thereof) is skipped.

```
@Test
void testOnlyOnDeveloperWorkstation() {
    assumeTrue("DEV".equals(System.getenv("ENV")),
        "Aborting test: not on dev environment");
    assertSomething(/*...*/); // only in dev environment
}
```

JUnit 5 – Assumptions

- Assumptions are utility methods we can use to conditionally execute tests (or parts of tests);
- They’re defined as static methods of the `Assumptions` class;
- If an assumption fails, the current test (or part thereof) is skipped.

```
@Test
void testInAllEnvironments() {
    assumingThat("PROD".equals(System.getenv("ENV")),() -> {
        assertSomething(/*...*/); // only in PROD environment
    });
    assertSomething(/*...*/); // always
}
```

JUnit 5 – Conditional Test Execution

JUnit 5 – Extension Model

The behaviour of test classes and test methods can be furtherly extended by registering Extensions.

Extensions are related to certain phases in the lifecycle of a test, and their methods are called by the JUnit engine when the corresponding phase is reached.

Common extension points are

- Test Instance Post-processing (e.g. to initialize parameters)
- Parameter resolution (e.g. to add parameter sources)
- Lifecycle callbacks (e.g. run a custom method after each test)

JUnit 5 – Extension Model

```
@ExtendWith(DatabaseExtension.class)
class AnIntegrationTest {

    @InitDatabase          // custom annotation
    DatabaseDAO database; // automatically initialized

    // test methods..
}
```

Assertion Frameworks

Assertion Frameworks

Assertion frameworks allow developers to write better assertions, and thus better tests, without re-inventing the wheel!

- More readable
- More flexible
- More informative error messages

Notable examples include: Hamcrest, Google Truth, AssertJ

What's wrong with standard JUnit assertions?

Let's try our hand with an assertion

Suppose you need to test the `getDivisors()` method:

```
class MathUtils {

    // Returns a list of divisors of number
    static List<Integer> getDivisors(int number){
        List<Integer> list = new ArrayList<Integer>();
        for(int i = 1; i <= number; i++) {
            if(number % i == 0) {
                list.add(i);
            }
        }
        return list;
    }
}
```

Let's try our hand with an assertion (v. 1)

```
@Test
void testDivisorsOfEight() {
    List<Integer> divisors = MathUtils.getDivisors(8);
    List<Integer> expected = Arrays.asList(1,2,4,8);
    assertEquals(expected, divisors);
}
```

Quite brittle! We're over-specifying

- The test depends on the implementation, not on the public interface
- We don't really need the divisors to be in ascending order
- A small change to the implementation could break the test
  - E.g. adding `1` and `number` to the list before the loop, since both are sure to be divisors!

Let's try our hand with an assertion (v. 1)

```
@Test
void testDivisorsOfEight() {
    List<Integer> divisors = MathUtils.getDivisors(8);
    List<Integer> expected = Arrays.asList(1,2,4,8);
    assertEquals(expected, divisors);
}
```

Error message might not be very informative!

- With two `List<Integer>` you should get something like:
  `expected: <[1,2,4,8]> but was: <[2,4,8]>`
  That's not too bad, but what if there were 1000 elements?
- With two `List<Object>` you would have gotten something on the lines of `expected: <java.lang.ArrayList@6adbdf5> but was <java.lang.ArrayList@5ajrd98>`
  which is not very helpful!

Let’s try our hand with an assertion (v. 2)

```
@Test
void testDivisorsOfEight() {
    List<Integer> divisors = MathUtils.getDivisors(8);
    List<Integer> expected = Arrays.asList(1,2,4,8);
    assertTrue(divisors.containsAll(expected));
}
```

Too loose! We’re under-specifying

- The test would pass when `divisors` contains more numbers!

Let's try our hand with an assertion (v. 3)

```
@Test
void testDivisorsOfEight() {
    List<Integer> divisors = MathUtils.getDivisors(8);
    List<Integer> expected = Arrays.asList(1,2,4,8);
    assertAll(
        () -> assertTrue(expected.containsAll(divisors)),
        () -> assertTrue(divisors.containsAll(expected))
    );
}
```

Solved the over/under-specification issues, but..

- Made it a little harder to read (think if we couldn't use `containsAll()`)
- The error message we get is something like: `Multiple Failures (1 failure)`
  `expected: <true> but was: <false>`
  which is, again, not very helpful!

What's wrong with JUnit Assertions?

- Can lead to hard-to-read tests
- Can lead to over/under-specification
  - Resist the temptation of `assertEquals()` when it's not what we *really* need
  - Do not settle with loose specifications for the sake of convenience
- Error messages not always immediate to understand
  - There exist ways to achieve better error messages in standard JUnit assertions (e.g. the `Supplier<String>` parameter), but why should we re-invent the wheel?

Getting to know Hamcrest

Hamcrest allows developers to declaratively define

Matchers that can be combined to create flexible expressions of intent.

- Not only for testing (UI Validation, data filtering, …)
- Matcher objects represent conditions, and they can be combined declaratively to create natural-language like (fluent) assertions

Testing with Hamcrest

The entry point for Hamcrest in our tests is the method

```
<T> void assertThat(T actual, Matcher<? super T> matcher)
```

Testing with Hamcrest

The entry point for Hamcrest in our tests is the method

```
<T> void assertThat(T actual, Matcher<? super T> matcher)
```

```
// these express the same assertion
assertEquals(expected, actual);
assertThat(actual, equalTo(expected));
assertThat(actual, is(expected));
assertThat(actual, is(equalTo(expected)));

// and so do these                    // and these
assertTrue(condition);                assertNotNull(actual);
assertThat(condition, is(true));      assertThat(actual, is(not(nullValue())));
```

Hamcrest – Core Matchers

- `equalTo(o)` given an `Object`, returns a Matcher that matches with any object which is equal to o (according to `o.equals()`);
- `not(m)` returns a Matcher that complements the given Matcher `m`;
- `is(k)` if given an object, returns `equalTo(k)`. If given a Matcher, returns the matcher without changing its behaviour;

```
assertThat("foo", is(not(equalTo("bar"))));
```

Hamcrest – String Matchers

Take a `String s` as input and return a suitable Matcher

- `containsString(s)` and `containsStringIgnoringCase(s)`
- `startsWith(s)` and `startsWithIgnoringCase(s)`
- `endsWith(s)` and `endsWithIgnoringCase(s)`

```
assertThat("Hamcrest", startsWith("Ham")); // passes
assertThat("Hamcrest", startsWith("ham")); // fails
assertThat("Hamcrest", containsStringIgnoringCase("CRE")); //passes
assertThat("Hamcrest", endsWithIgnoringCase("REST")); //passes
```

Hamcrest – Comparable Matchers

Take a `Comparable c` as input and return a suitable Matcher

- `greaterThan(c)` and `greaterThanOrEqualTo(c)`
- `lessThan(c)` and `lessThanOrEqualTo(c)`
- `closeTo(c,delta)` with `delta` being the admissible error

```
assertThat(10, is(greaterThanOrEqualTo(10)));
assertThat(10, is(lessThan(100)));
double a = 1.001, b = 1.002, delta = 0.002;
assertThat(a, is(closeTo(b, delta)));
```

Hamcrest – Logical Matchers

- `allOf(m1,…)` returns a matcher that matches the examined object only if it matches all the given matchers.
- `anyOf(m1,…)` returns a matcher that matches the examined object only if it matches one of the given matchers.
- `both(m1).and(m2)` and `either(m1).or(m2)` build matchers that match only if both (resp. either) `m1` and (resp. or) `m2` match.

```
assertThat("Hamcrest", allOf(startsWith("Ham"), endsWith("crest")));
assertThat(123, anyOf(is(lessThan(50)), is(greaterThan(120))));
assertThat(123, either(is(lessThan(50))).or(is(greaterThan(130))));
```

Hamcrest – Object Matchers

- hasProperty(s) matches with all objects with an «s» property
- hasProperty(s, m) matches with all objects whose «s» field matches the given matcher

```
Car car = new Car("Stelvio", 280, "Gasoline"); //model, hp, fuel
assertThat(car, hasProperty("fuel"));
assertThat(car, hasProperty("hp", is(greaterThan(200))));
```

Hamcrest – Collection Matchers: Iterables

- `hasItem(m)` matches with Iterables that contain an item matching with `m`
- `contains(k1,…)` matches only if each n-th element of the examined Iterable matches with the corresponding n-th matcher/object
- `containsInAnyOrder(k1,…)` matches only if each element of the examined Iterator matches with exactly one of the matchers/objects (same length required).

```
List<String> s = Arrays.asList("Anna","Bob","Carl");
assertThat(s, hasItem(both(startsWith("A")).and(endsWith("a"))));
assertThat(s, contains(startsWith("A"),startsWith("B"),startsWith("C")));
assertThat(s, containsInAnyOrder("Bob","Anna","Carl"));
```

Hamcrest – Collection Matchers: Maps

- `hasKey(m)` matches with Maps that contain a key matching with `m`
- `hasValue(m)` matches with Maps containing a value matching with m
- `hasEntry(m1,m2)` matches with Maps that contain a key matching with m1 whose value matches with m2.

```
Map<String,Integer> map = new HashMap<String,Integer>();
map.put("Anna", 1); map.put("Bob", 3); map.put("Carl", 3);
assertThat(map, hasKey(containsString("nn")));
assertThat(map, hasValue(is(greaterThan(2))));
assertThat(map, hasEntry(startsWith("A"), is(lessThan(2))));
```

Hamcrest – There's much more!

- Check out the docs if you want to know more!
- For a starter, you can take a look at the `Matchers` class.

Let's get back at our example

```
@Test
void testDivisors() {
    List<Integer> divisors = MathUtils.getDivisors(8);
    List<Integer> expected = Arrays.asList(1,2,4,8);
    assertAll( // v3
        () -> assertTrue(expected.containsAll(divisors)),
        () -> assertTrue(divisors.containsAll(expected))
    );
    // with Hamcrest
    assertThat(divisors, containsInAnyOrder(1,2,4,8));
}
```

And here's what an error message looks like with Hamcrest:

```
Expected: iterable with items [<1>, <2>, <4>, <8>] in any order
     but: no item matches: <1> in [<2>, <4>, <8>]
```

Another example

We need to check that in a `List<Car>` there is at least one `Car` such that its fuel is `"Electric"` or `"Hydrogen"`.

Another example – The classic way

We need to check that in a `List<Car>` there is at least one `Car` such that its fuel is `"Electric"` or `"Hydrogen"`.

```
boolean found = false;
for(Car c: Car.getCars()) {
    if(c.getFuel().equals("Electric") || c.getFuel().equals("Hydrogen")) {
        found = true;
        break;
    }
}
assertTrue(found);

> Error message:
Expected: <true> but was: <false>
```

Another example – The Hamcrest way

We need to check that in a `List<Car>` there is at least one `Car` such that its fuel is `"Electric"` or `"Hydrogen"`.

```
assertThat(Car.getCars(), hasItem(
    hasProperty("fuel", either(is("Electric")).or(is("Hydrogen")))
));


> Error message:
Expected: a collection containing
          hasProperty("fuel", (is "Electric" or is "Hydrogen"))
     but: mismatches were: [ property 'fuel' was "Hidrogen",
          property 'fuel' was "Gas", property 'fuel' was "Gas"]
```

Writing our own Matcher

Suppose we need to check that a `List<Student>` contains at least one `Student` whose name is not a palindrome.

Yes, we *really* do need that! 😐

```
assertThat(list, hasItem(hasProperty("firstName", is(not(palindrome())))));
```

Hamcrest features a lot of built-in Matchers, but none to match palindrome strings, unfortunately.

Not too bad, we'll write it ourselves!

Writing our own Matcher

To create a new matcher, one needs to implement the Matcher interface:

```
public interface Matcher<T> {
    boolean matches(Object actual);
    void describeMismatch(Object actual, Description mismatchDescription);
}
```

Often, anyway, it is more practical to extend one of the abstract classes `BaseMatcher<T>` or `TypeSafeMatcher<T>`, which implement `Matcher<T>`.

Writing our own Matcher

```
public class IsPalindrome extends TypeSafeMatcher<String>{

    @Override
    public void describeTo(Description description) {
        description.appendText("a palindrome string");
    }

    @Override
    protected boolean matchesSafely(String item) {
        StringBuilder sb = new StringBuilder(item).reverse();
        return sb.toString().equalsIgnoreCase(item);
    }

    public static IsPalindrome palindrome() {
        return new IsPalindrome();
    }
}
```

Generates a description of the object we match

matchesSafely is called by TypeSafeMatcher .matches()

Static method to access our matcher, so we can use it as the built-in ones

Using our own matcher

```
import static org.hamcrest.Matchers.*;
import static it.unina.computerscience.softeng.junitdemo.examples.IsPalindrome.*;

// ...

assertThat(list, hasItem(hasProperty("firstName", is(not(palindrome())))));


> Error message:
Expected: a collection containing
          hasProperty("firstName", is not a palindrome string)
     but: mismatches were: [ property 'firstName' was "Anna",
          property 'firstName' was "Bob",  property 'firstName' was "Otto"]
```

Other Java Assertion frameworks

Other well-known Java assertion frameworks include:

- AssertJ (web)
- Google Truth (web)

They're less general than Hamcrest, and are based on method chaining.

```
// AssertJ
assertThat(gamma.getName()).startsWith("Er").endsWith("ich");
assertThat(theGangOfFour).contains(gamma,helm).doesNotContain(fowler);

// Truth
assertThat(theGangOfFour).containsExactly(gamma,helm,vliss,johnson).inOrder();
```

AssertJ / Truth vs Hamcrest

AssertJ / Truth

- Based on method chaining
  - No LISP-like parentheses hell
  - IDE auto-complete
- Generally can provide better error messages

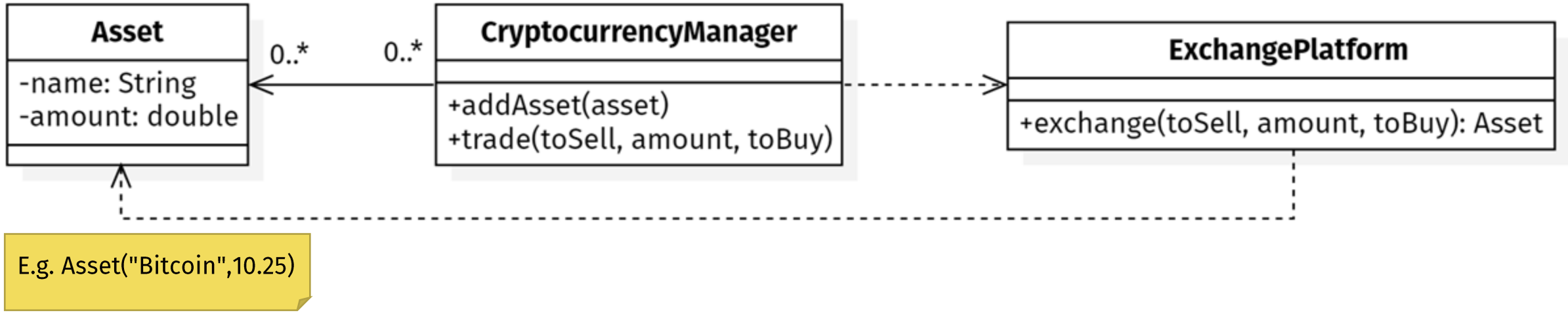
Hamcrest

- More general
  - Applications outside of testing
  - Can be used to set expectations when mocking

Testing in Isolation

Dependencies and Testing

A class may depend on other helper classes (e.g. DAOs, APIs)



Suppose we need to test the `trade()` method.

Dependencies and Testing

```java
@Test
void testTrading() {
    // arrange
    CryptocurrencyManager c = new CryptocurrencyManager();
    c.addAsset(new Asset("BTC",10.0));
    c.addAsset(new Asset("EUR",10000.0));
    // act (1 BTC = 7000.0 EUR)
    c.trade("EUR", 7000.0, "BTC");
    // assert
    assertTrue(c.getAsset("EUR").get().getAmount() == 3000.0);
    assertTrue(c.getAsset("BTC").get().getAmount() == 11.0);
}
```

We're using the real ExchangeAPI, trading **real** Euros!

The exchange rate might change tomorrow!

See anything wrong with this?

Dependencies and Testing

Using real production objects as helpers in our unit tests has a few drawbacks and may not always be possible:

- When a test fails, we don't know if the bug is in the unit we're testing or in its dependencies;
- The introduction of a bug in a highly-used helper object can cause a ripple of failing tests all across the system;
- Sometimes, we cannot afford to use real production objects! Think of a `BankTransactionsDAO`, or a `MissileLauncher`!

How can we alleviate the above issues and still test our units?

Test Doubles

Test Doubles are replacements for production objects that are typically used in tests.

Test Doubles can be classified as follows:

- **Fakes:** objects that have working implementations, but are not suitable for production because of limitations**;**
- **Stubs:** replace a real component and return pre-defined answers;
- **Mocks:** more advanced, configurable test stubs that can also record the indirect outputs of the unit under test.

(Lack of) Inversion of Control

Suppose our CryptocurrencyManager was like this:

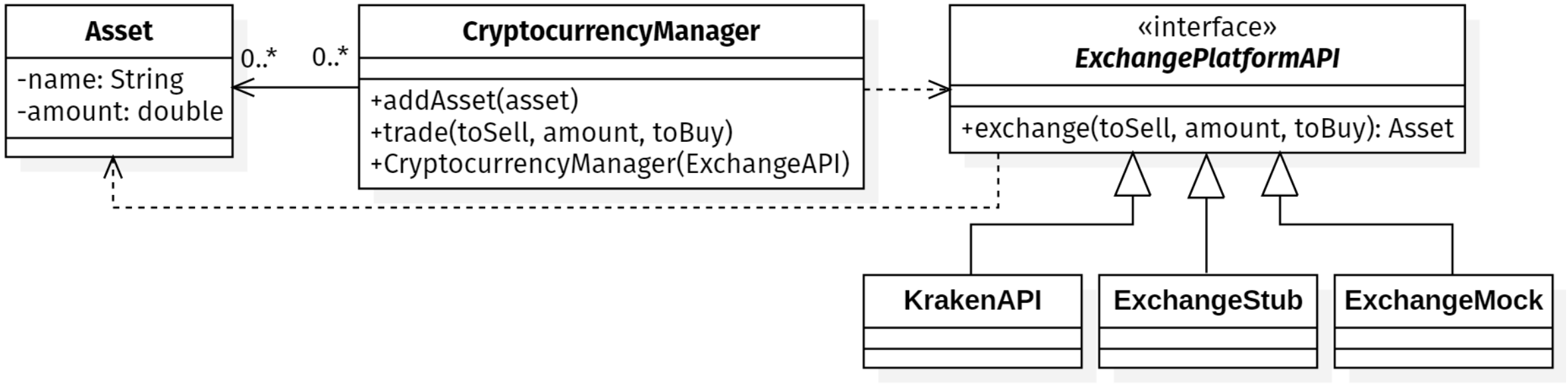
```
public class CryptocurrencyManager {
  private ExchangePlatform exchangeAPI = new ExchangePlatform();
  CryptocurrencyManager(){/*...*/}
  public void trade(String toSell, double amountToSell, String toBuy) {/*...*/}
}
```

That's really BAD design!

- What if we want to support more than one exchange platform?
- There's no way we can use a test double in place of the real deal!

Inversion of Control

- Client classes should be able to provide dependencies
  - Through constructors, public setters, builders, method parameters…
- Inversion of control is fundamental to make units testable



Dependencies and Testing

```java
@Test
void testTrading() {
    // arrange
    CryptocurrencyManager c = new CryptocurrencyManager(new ExchangeStub());
    c.addAsset(new Asset("BTC",10.0));
    c.addAsset(new Asset("EUR",10000.0));
    // act (1 BTC = 7000.0 EUR)
    c.trade("EUR", 7000.0, "BTC");
    // assert
    assertTrue(c.getAsset("EUR").get().getAmount() == 3000.0);
    assertTrue(c.getAsset("BTC").get().getAmount() == 11.0);
}
```

We're not trading **real** Euros anymore!

The exchange rate won't change if the stub doesn't!

Our very simple Exchange Platform Stub

```
class ExchangeStub implements ExchangeAPI {
    @Override
    public Asset exchange(String toSell, double amount, String toBuy) {
        return new Asset("BTC",1.0);
    }
}
```

Don’t be fooled by this tiny example! Writing mocks and stubs can be very tedious!

Think of multiple methods, with many conditions each!

- E.g.: when amount < 1000, then return X, when >2000 …
- And you might not be able to re-use it in the next test!

Mocking frameworks

Writing stubs and mocks is boring and takes time.

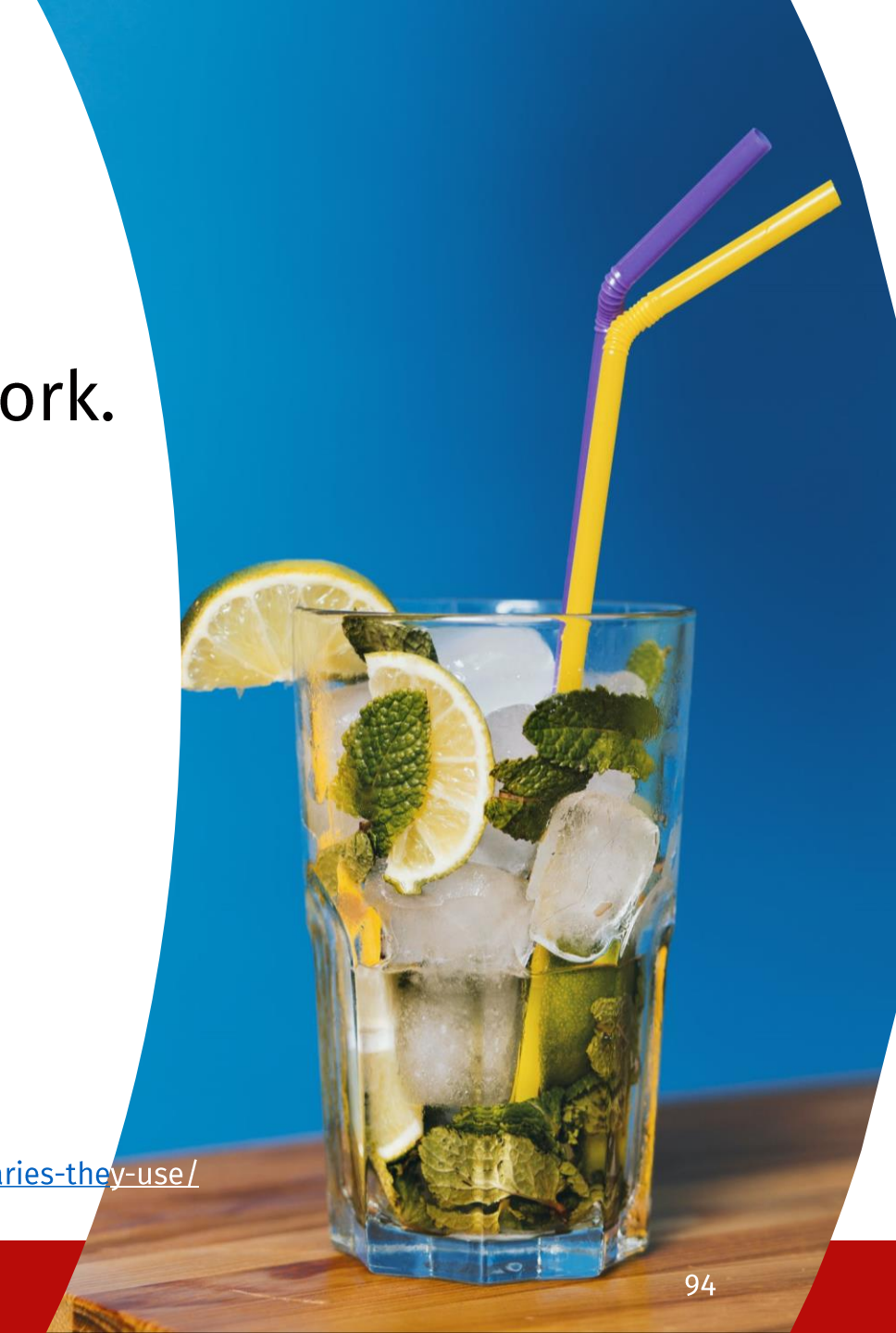
Mocking frameworks allow us to easily create mocks and stubs to use in our tests.

Popular frameworks include: Mockito, EasyMock, JMockit.

Introducing Mockito

Mockito is a well-known mocking framework.
Top 10 Java library across all libraries[1] .

Plus, it tastes really good!



[1] https://blog.overops.com/githubs-10000-most-popular-java-projects-here-are-the-top-libraries-they-use/

Mockito – Creating Mocks

Mocks can be created with the static method

`<T> T mock(Class<T> classToMock)`

By default, each method of the mock will return the default value for the corresponding type.

```
List mockedList = mock(ArrayList.class); // with classes
Map mockedMap   = mock(Map.class);       // and with interfaces too

mockedMap.get("Foo"); // returns null
mockedList.get(2);    // returns null
mockedList.isEmpty(); // returns false
mockedList.size();    // returns 0
```

Mockito – Configuring Mocks

We can configure the behaviour of our mock as follows:

```
List mockedList = mock(ArrayList.class); // with classes
Map mockedMap   = mock(Map.class);       // and with interfaces too

// configuration
when(mockedMap.get("Foo")).thenReturn("Bar");
doReturn("Bar").when(mockedList).get(2); // alternative way
when(mockedList.size()).thenReturn(99);

mockedMap.get("Foo"); // returns "Bar"
mockedList.get(2);    // returns "Bar"
mockedList.isEmpty(); // returns false as before
mockedList.size();    // returns 99
```

Mockito – Argument matchers

Often, we won't need to be so specific about input arguments.

In fact, being more generic could help us write less configuration code and more flexible tests.

```
// configuration: order matters!
when(mockedList.get(anyInt())).thenReturn("Mockito!");
when(mockedList.get(lt(0))).thenThrow(IllegalArgumentException.class);
when(mockedList.get(-25)).thenReturn("MOCKITO!");

mockedList.get(1988);  // returns "Mockito!"
mockedList.get(-25);   // returns "MOCKITO!"
mockedList.get(-2);    // throws IllegalArgumentException
```

Mockito – Configuring Mocks

When we invoke a method on mock with a given list of parameters, Mockito tries to find the latest configuration rule that «matches», and applies that.

If no configuration rule is found, it defaults to default values.

Therefore, configuration order matters, and more general rules should be declared before the more specific ones.

Mockito – Argument matchers

Mockito features a lot of built-in argument matchers, that can often also be combined in a Hamcrest-like fashion.

E.g.: `any()`, `anyString()`, `anyCollection()`, `leq()`, `isNull()`, …

We won't examine them in detail today, but if you're interested you can check out the ArgumentMatchers and the AdditionalMatchers classes from the docs.

It's also possible to use Hamcrest matchers as Mockito argument matchers with the MockitoHamcrest class!

Mockito – Arg. matchers with Hamcrest

```
import static org.mockito.Mockito.*;
import static org.mockito.hamcrest.MockitoHamcrest.argThat;
import static org.hamcrest.Matchers.*;
import static it.unina.computerscience.softeng.junitdemo.examples.IsPalindrome.*;

// configuration - argThat is a static method from the MockitoHamcrest class
when(mockedList.add(argThat(
        is(palindrome()) // Hamcrest Matcher
))).thenReturn(true);

mockedList.add("Abe");  // returns false
mockedList.add("Bob");  // returns true
mockedList.add("Otto"); // returns true
```

Mockito – Configuring Answers

When mocking with Mockito, you're not limited to pre-determined return values.

- With the generic interface Answer, you can customize the behaviour of your mocks by specifying an action to be executed to compute the desired output.

Mockito – Configuring Answers

```
List<Integer> myMock = mock(ArrayList.class);

when(myMock.get(anyInt())).thenAnswer(
  new Answer() {
    public Object answer(InvocationOnMock invocation) {
      return ((Integer)invocation.getArgument(0)) + 1;
    }
  });

//same as the above
when(myMock.get(anyInt())).thenAnswer(
  invocation -> ((Integer)invocation.getArgument(0)) + 1
);

myMock.get(42); // returns 43
myMock.get(99); // returns 100
```

Mockito – Verifying behaviour

Usually, in the assert phase of our tests, we focus on checking that the final state of the Class Under Test is the one we expect.

This is testing by side-effects, and often it is not enough!

```
// assert
assertTrue(c.getAsset("EUR").get().getAmount() == 3000.0);
assertTrue(c.getAsset("BTC").get().getAmount() == 11.0);
// hopefully, a call was made to ExchangeAPI.exchange() with the right params!
```

Mockito – Verifying behaviour

After you're done with your mock, you can further inspect it to check if and how it's been used!

Out-of-the-box, with no configuration required, a Mockito mock keeps track of every method call it receives.

You can verify a mock's past behaviour with the `verify()` method and its variants.

Mockito – Verifying behaviour: examples

```
// check that myMock.size() was called exactly once
verify(myMock).size();
// check that myMock.get(19) was called exactly once
verify(myMock).get(19);
// check that myMock.get was called exactly three times with any int arg.
verify(myMock, times(3)).get(anyInt()); // with Mockito matcher
// check that myMock.get was called no more than three times with any even arg.
verify(myMock, atMost(3)).get(argThat(is(even()))); // with Hamcrest matcher
// check that myMock.get(0) was called at least two time
verify(myMock, atLeast(2)).get(0);
// check that no interaction occurred with mockedStack
verifyNoInteractions(mockedStack);
// check that myMock.addAll was never called
verify(myMock, times(0)).addAll(anyCollection());
```

Mockito – Verifying behaviour: examples

```
// check that method calls happened in a precise order
InOrder inOrder = inOrder(myMock);
inOrder.verify(myMock).get(19);
inOrder.verify(myMock).get(42);
inOrder.verify(myMock).get(99);

// check that no interaction happened after myMock.get(99)
verifyNoMoreInteractions(myMock);
```

Mockito – Verifying behaviour: errors

Here's what a `verify()` error message looks like:

```
> Error message:

org.mockito.exceptions.verification.NoInteractionsWanted:
No interactions wanted here:
-> at CryptocurrencyManagerTest.java:153
But found this interaction on mock 'arrayList':
-> CryptocurrencyManagerTest.java:138
```

Getting back at our example

```
@Test
void testTrading() {
    // create and configure mock
    ExchangePlatformAPI api = mock(ExchangePlatformAPI.class);
    when(api.exchange("EUR", 7000.0, "BTC")).thenReturn(new Asset("BTC",1.0));
    // arrange
    CryptocurrencyManager c = new CryptocurrencyManager(api);
    c.addAsset(new Asset("BTC",10.0)); c.addAsset(new Asset("EUR",10000.0));
    // act
    c.trade("EUR", 7000.0, "BTC");
    //assert
    assertTrue(c.getAsset("EUR").get().getAmount() == 3000.0);
    assertTrue(c.getAsset("BTC").get().getAmount() == 11.0);
    verify(api).exchange("EUR",7000.0,"BTC"); // API were actually called
}
```

Mockito and JUnit 5

There's an official mockito-junit-jupiter extension for JUnit 5.

```
@ExtendWith(MockitoExtension.class) // register the extension
class CryptocurrencyManagerTest {

    @Mock ExchangePlatformAPI api;  // parameter initialization

    @Test
    void testTrading() {/*...*/}

    @Test
    void testWithMockParam(@Mock List<Integer> list) {/*...*/} //param. resolution

}
```

Mockito – Limitations

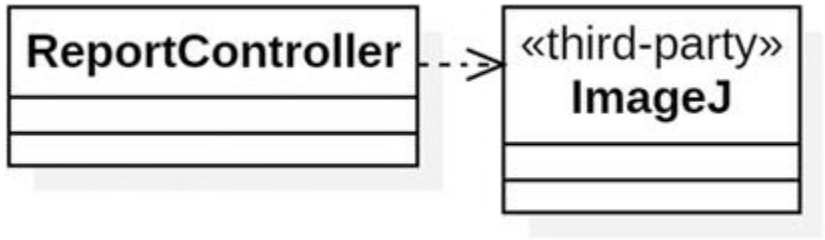
With Mockito you cannot mock:

- constructors;
- static methods;
- equals(), hashCode();

If you need to, you can either consider refactoring, or checkout more "invasive" tools like PowerMock, which uses custom classloaders and bytecode manipulation.

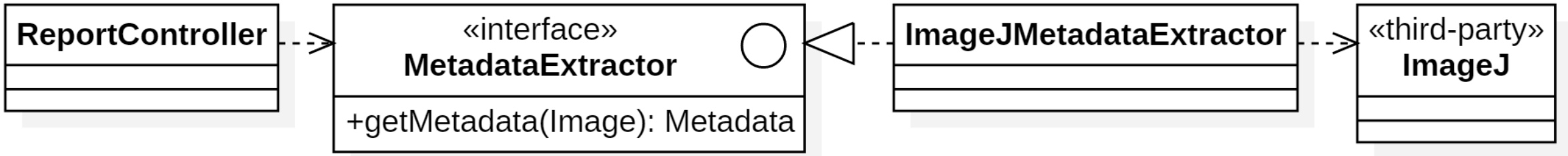
Rules to mock by

1. Mock behaviour, not data!
   - Just instantiate your POJOs, use builders if you need to.
2. Do not mock types you don't own (e.g. third-party libraries)
   - It might cover up bugs!
   - You should put up an adapter layer between your units and 3rd-party libraries, and mock the adapter



Rules to mock by

1. Mock behaviour, not data!
   - Just instantiate your POJOs, use builders if you need to.
2. Do not mock types you don't own (e.g. third-party libraries)
   - It might cover up bugs!
   - You should put up an adapter layer between your units and 3rd-party libraries, and mock the adapter



References

- https://martinfowler.com/bliki/UnitTest.html
- https://martinfowler.com/bliki/TestDouble.html
- http://xunitpatterns.com/Test%20Double.html
- https://martinfowler.com/articles/mocksArentStubs.html
- https://junit.org/junit5/docs/current/user-guide/
- http://hamcrest.org/JavaHamcrest/
- https://site.mockito.org/