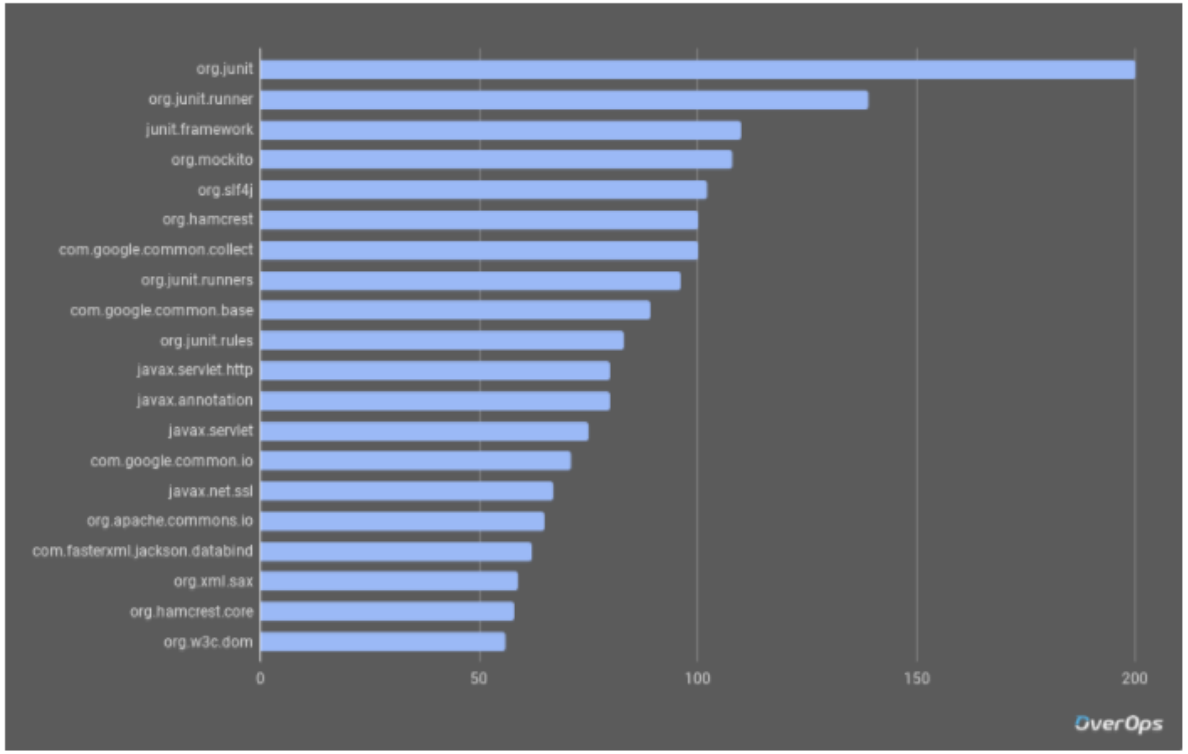
JUnit 5

Junit

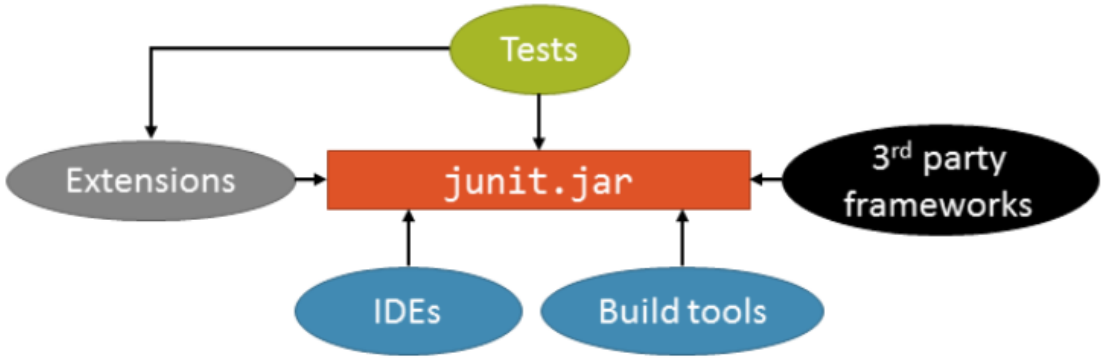
- JUnit is the most important testing framework for the JVM and one of the most influential in software engineering in general.

- According to several studies (2017) , JUnit 4 was the most used library for Java projects. For instance, *The Top 100 Java libraries on GitHub*.



JUnit4

- JUnit 4 was released back in 2006.
- Not up to date with the newest testing patterns
- Not up to date with newest Java language features (build in Java5)
- Monolithic Architecture

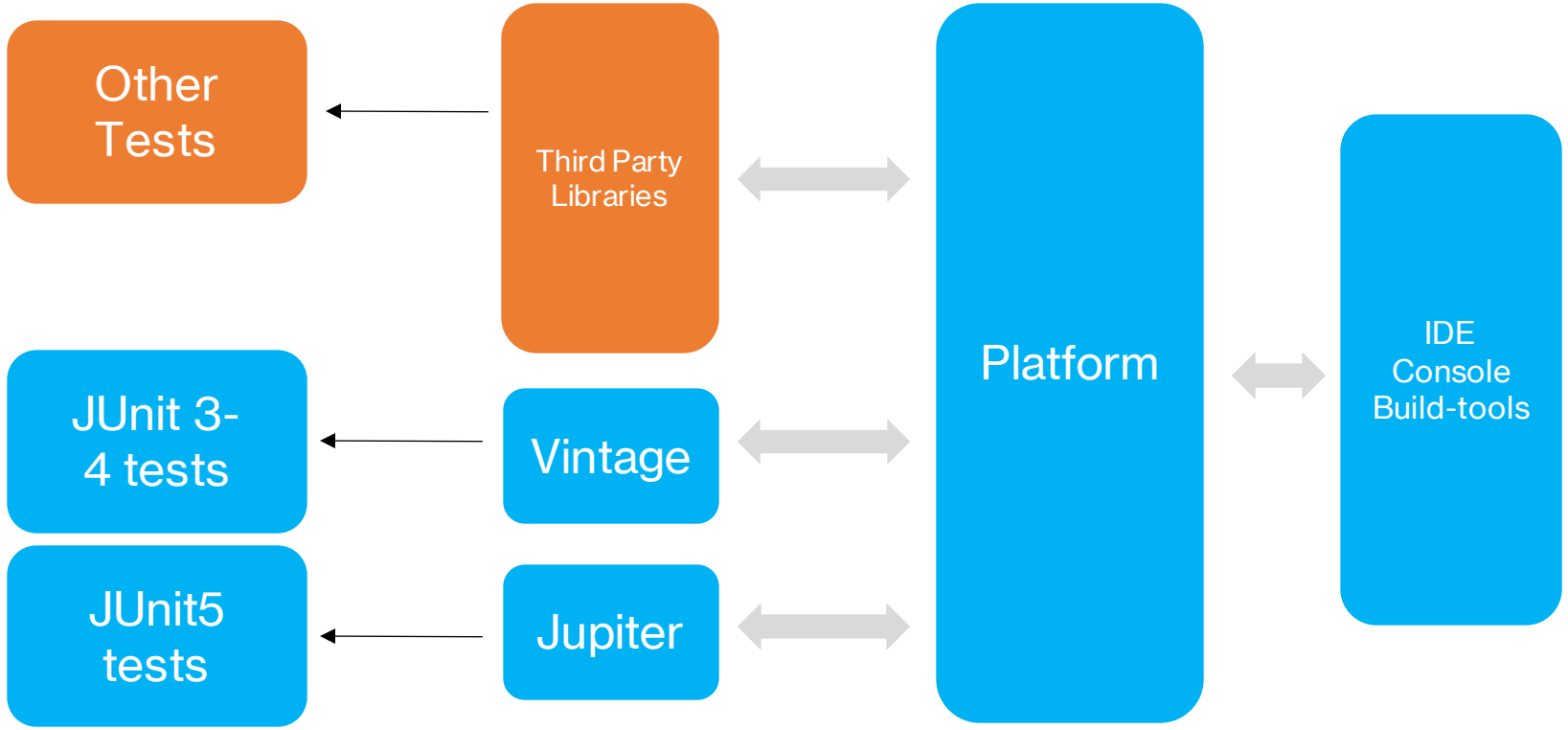


The JUnit 4 Architecture

Road to Junit5

- In July 2015 Johannes Link and Marc Philipp started a crowdfunding campaign (http://junit.org/junit4/junit-lambda-campaign.html) on Indiegogo under the name JUnit Lambda
- Run from July to October 2015 and raised over 54k euros from individuals and companies
- 4 companies donated 6 week of their own developer time

Junit5 Architecture



Setting up JUnit5

pom.xml JUnit 5 dependencies

```
<dependency>
     <groupId>org.junit.jupiter</groupId>
     <artifactId>junit-jupiter-api</artifactId>
     <version>5.6.0</version>
     <scope>test</scope>
</dependency>
<dependency>
     <groupId>org.junit.jupiter</groupId>
     <artifactId>junit-jupiter-engine</artifactId>
     <version>5.6.0</version>
     <scope>test</scope>
</dependency>
```

Jupiter API

JUnit Platform

Maven Surefire plugin configuration in pom.xml

```
<build>
    <plugins>
         <plugin>
             <artifactId>maven-surefire-plugin</artifactId>
             <version>2.22.2</version>
        </plugin>
    </plugins>
</build>
```

JUnit Jupiter Programming Model

`org.junit.jupiter.api`

- Core Assertions
- Conditional tests execution
- Annotations and meta-annotation
- Custom display names
- Tagging and Filtering
- Nested Test classes
- Parameterized Tests
- Dynamic tests

Core Assertions

- assertEquals() , assertNotNull() etc
- assertThrows()
- assertTimeout()
- assertAll()
- for more power AssertJ,Hamcrest etc

Conditional test execution

- assumeTrue() / assumeFalse()
- @EnabledOnOs() / @DisabledOnOs()
- @EnabledOnJre() / @DisabledOnJre()
- @EnabledIfSystemProperty()
- @EnabledIfEnvironmentVariable()
- @EnabledIf() / @DisableIf()
- @Disable

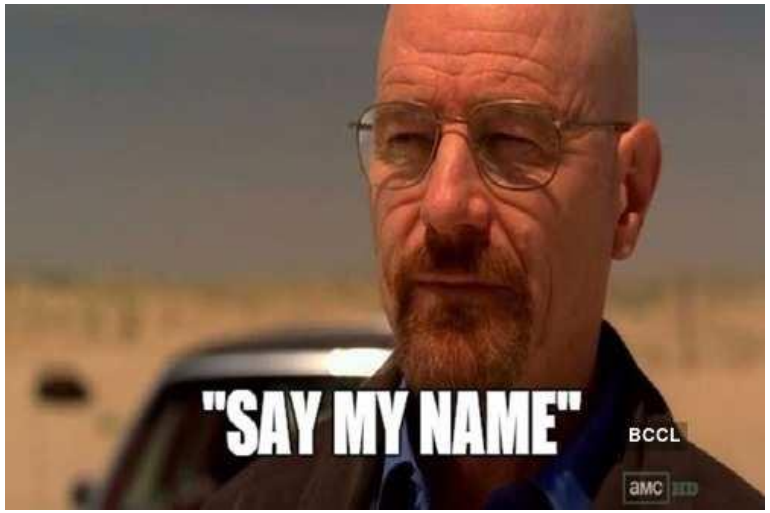
Custom display names

@DisplayName

```java
@Test
@DisplayName("Custom test name containing spaces")
void testWithDisplayNameContainingSpaces() {
}

@Test
@DisplayName("╯°□°）╯")
void testWithDisplayNameContainingSpecialCharacters() {
}

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



Display Name Generators

- JUnit Jupiter supports custom display name generators that can be configured via the **@DisplayNameGeneration** annotation.
- Values provided via **@DisplayName** annotations always take precedence over display names generated by a **DisplayNameGenerator**.

DisplayNameGenerator	Behavior
`Standard`	Matches the standard display name generation behavior in place since JUnit Jupiter 5.0 was released.
`Simple`	Removes trailing parentheses for methods with no parameters.
`ReplaceUnderscores`	Replaces underscores with spaces.
`IndicativeSentences`	Generates complete sentences by concatenating the names of the test and the enclosing classes.

Tagging and Filtering

- Tags are a JUnit Platform concept for marking and filtering tests.
- Test classes and methods can be tagged via the @Tag annotation.
- For filtering JUnit tests within the various phases of the Maven build, we can use the Maven Surefire plugin. The Surefire plugin allows us to include or exclude the tags in the plugin configuration:

```java
@IntegrationTest
@SpringBootTest
public class DummyIntegrationTest {


    @Tag("TST")
    @Test
    void test_executed_on_tst() { System.out.println("This is an integration test"); }


    @Tag("DEV")
    @Test
    void test_executed_on_dev() { System.out.println("This is an integration test"); }
}
```

```xml
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
    <configuration>
        <groups>UnitTest|IntegrationTest</groups>
    </configuration>
</plugin>
```

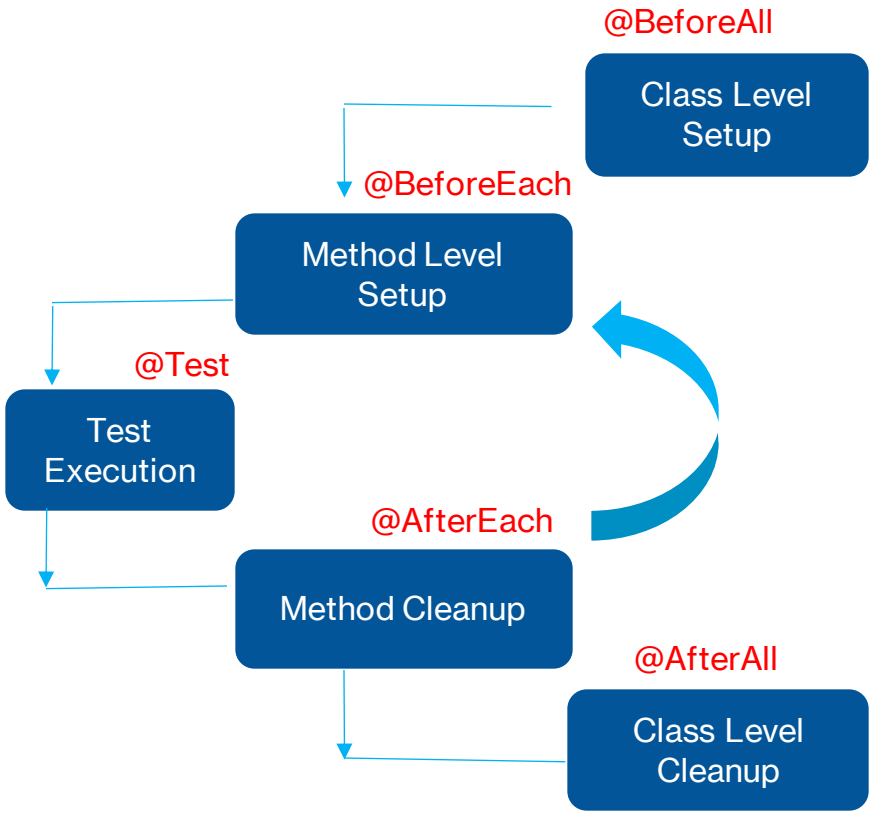
Test Instance Lifecycle

By default, JUnit creates a new instance of the test class before executing each test method. This helps us to run individual test methods in isolation and avoids unexpected side effects.

Setup: This phase puts the test infrastructure in place. JUnit provides class level setup (**@BeforeAll**) and method level setup (**@BeforeEach**).

Test Execution: In this phase, the test execution and assertion happen. The execution result will signify a success or failure.

Cleanup: This phase is used to cleanup the test infrastructure setup in the first phase. Just like setup, teardown also happen at class level (**@AfterAll**) and method level (**@AfterEach**).



Nested Tests

- @Nested tests give the test writer more capabilities to express the relationship among several groups of tests.
- Such nested tests make use of Java's nested classes and facilitate hierarchical thinking about the test structure.
- Only non-static nested classes (i.e. inner classes) can serve as @Nested test classes. Nesting can be arbitrarily deep, and those inner classes are subject to full lifecycle support with one exception: @BeforeAll and @AfterAll methods do not work by default

```
public class BankResourceTest {


    @Nested
    @DisplayName("GET methods")
    class GET {...}

    @Nested
    @DisplayName("POST methods")
    class POST {...}

    @Nested
    @DisplayName("DELETE methods")
    class DELETE {...}

}
```

Repeated Tests

- JUnit Jupiter provides the ability to repeat a test a specified number of times by annotating a method with **@RepeatedTes**t and specifying the total number of repetitions desired. Each invocation of a repeated test behaves like the execution of a regular @Test method with full support for the same lifecycle callbacks and extensions.

```
@RepeatedTest(10)
void repeatedTest() {
        System.out.println("Test is executed");
}

@RepeatedTest(5)
void repeatedTestWithRepetitionInfo(RepetitionInfo repetitionInfo) {
      System.out.println(repetitionInfo.getCurrentRepetition());
      assertEquals(5, repetitionInfo.getTotalRepetitions());
 }
```

Parameterized Tests

- Parameterized tests make it possible to run a test multiple times with different arguments. They are declared just like regular @Test methods but use the @ParameterizedTest annotation instead. In addition, you must declare at least one source that will provide the arguments for each invocation and then consume the arguments in the test method

```
@ParameterizedTest
@ValueSource(strings = {"race", "taxi", "1234"})
void parameterized_test_1(String candidate) {
assertEquals(4, candidate.toCharArray().length);
}
```

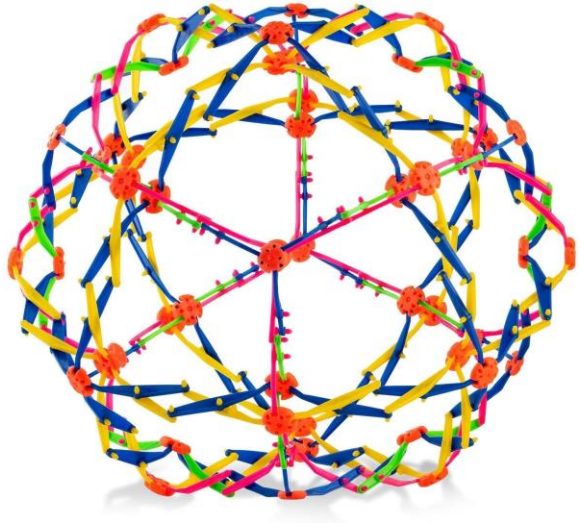
JUnit 5 Extension Model

The purpose of Junit 5 extensions is to extend the behavior of test classes or methods, and these can be reused for multiple tests.

Before Junit 5, the JUnit 4 used two types of components for extending a test: test runners and rules. By comparison, JUnit 5 simplifies the extension mechanism by introducing a single concept: the *Extension* API.

Declaratively - **@ExtendWith**
Programmatically - **@RegisterExtension**

- Conditional test execution
- Lifecycle callbacks
- Parameter resolution
- Exception handling
- Test instance post-processing



Conditional Test Execution

- JUnit 5 provides a type of extension that can control whether or not a test should be executed.
- This is defined by implementing the **ExecutionCondition** interface.

```
public class EnvironmentExecutionCondition implements ExecutionCondition {

@Override
public ConditionEvaluationResult evaluateExecutionCondition(ExtensionContext context) {

return extensionContext.getTags().contains("ACC") ?
      enabled("Enabled") :
       disabled("Disabled");
}
```

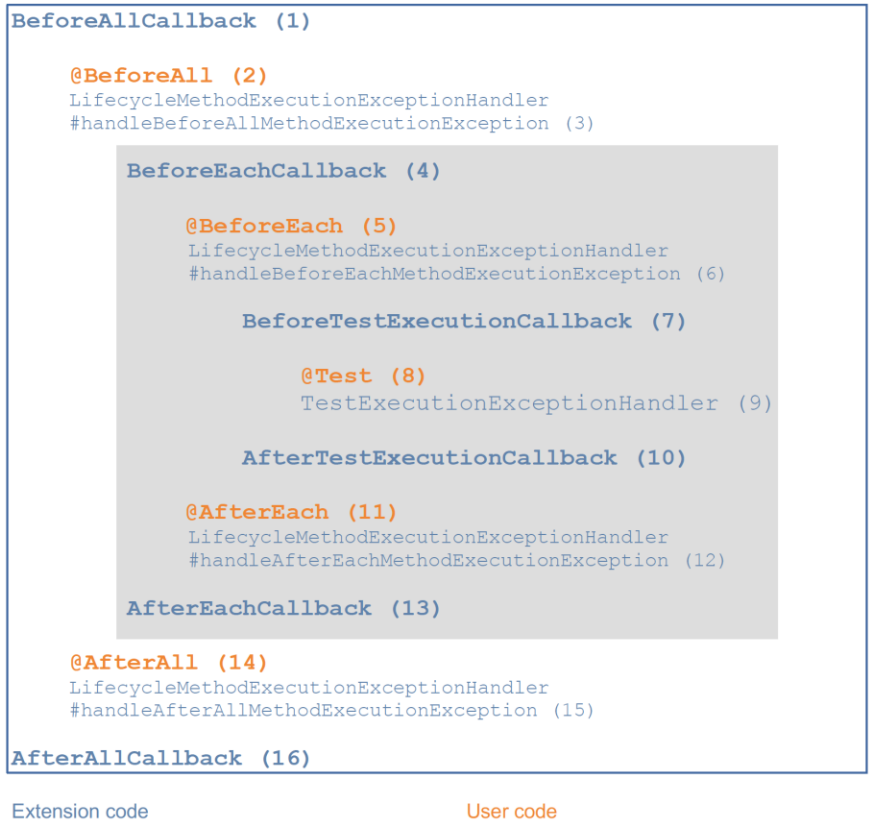
```
@Test
@ExtenhWith(EnvironmentExecutionCondition.class)
@Tag("ACC")
void regression_test() {
System.out.println("Run regression test");
}
```

Lifecycle callbacks

The following interfaces define the APIs for extending tests at various points in the test execution lifecycle.

- ***BeforeAllCallback*** and ***AfterAllCallback*** – executed before and after all the test methods are executed
- ***BeforeEachCallBack*** and ***AfterEachCallback*** – executed before and after each test method
- ***BeforeTestExecutionCallback*** and ***AfterTestExecutionCallback*** – executed immediately before and immediately after a test method

Each of these interfaces contains a method we need to override.



Parameter resolution

By implementing *ParameterResolver* you can serve up objects of any type to your test methods

ParameterResolver – has two methods:

- **supportsParameter()** – returns true if the parameter's type is supported (TestRestTemplate in this example)
- **resolveParamater()** – serves up an object of the correct type (new TestRestTemplate), which will then be injected in your test method

```
public class TestRestTemplateParameterResolver implements ParameterResolver {

@Override
public boolean supportsParameter(ParameterContext parameterContext, ExtensionContext extensionContext) throw
// Check if the parameter provided is valid
}


@Override
public Object resolveParameter(ParameterContext parameterContext, ExtensionContext extensionContext) throws {
//Instantiate the parameter
}
```

```
@TestRestTemplateResolver
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class LifecycleTest {

@Test
public void test_endpoint(TestRestTemplate template) throws URISyntaxException {
    template.exchange(request, new ParameterizedTypeReference<Object>() {
});
}
```

The End