
Java Unit Test

Sebelum Belajar Materi Ini

- Java Dasar
- Java Object Oriented Programming
- Java Generic
- Java Collection
- Java Lambda
- Apache Maven

Agenda

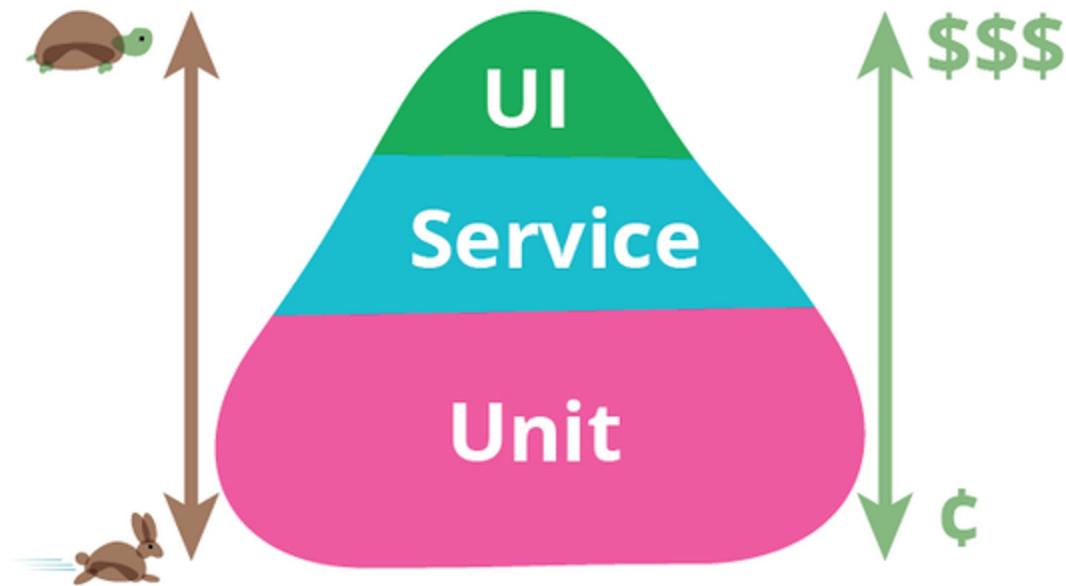
- Pengenalan Software Testing
- Pengenalan JUnit
- Membuat Test
- Menggunakan Assertions
- Menggunakan Assumptions
- Dependency Injection di Test
- Mocking
- Dan -lain-lain

Pengenalan Software Testing

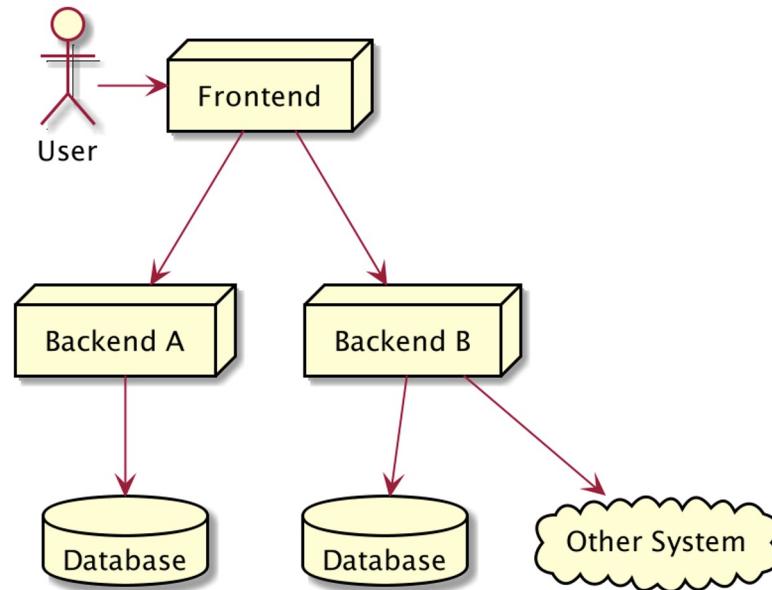
Pengenalan Software Testing

- Software testing adalah salah satu disiplin ilmu dalam software engineering
- Tujuan utama dari software testing adalah memastikan kualitas kode dan aplikasi kita baik
- Ilmu untuk software testing sendiri sangatlah luas, pada materi ini kita hanya akan fokus ke unit testing

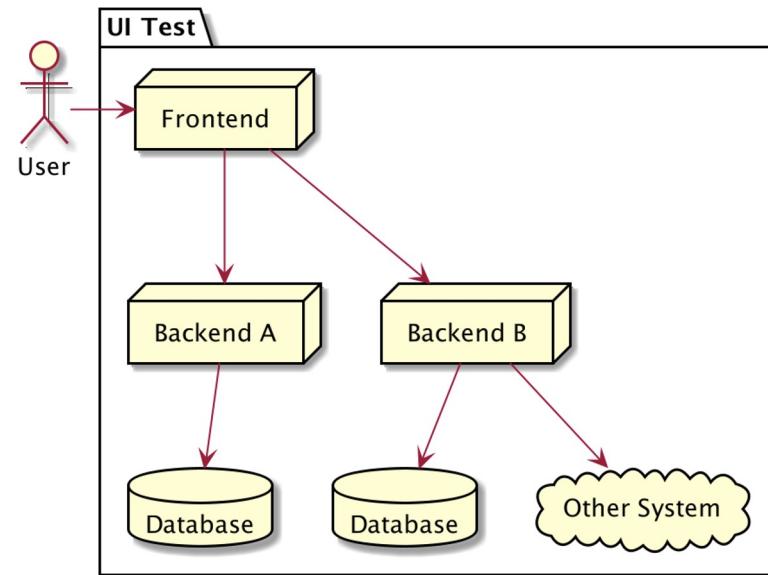
Test Pyramid



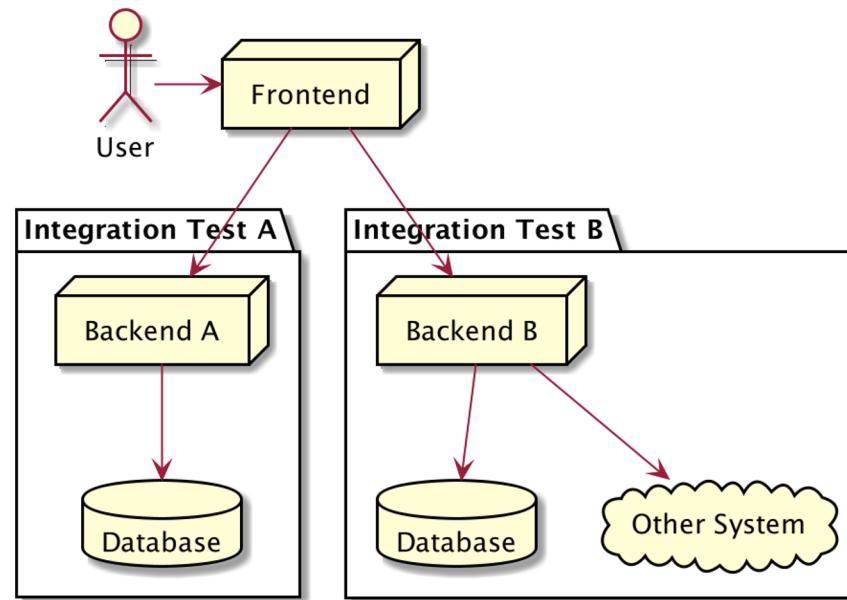
Contoh High Level Architecture Aplikasi



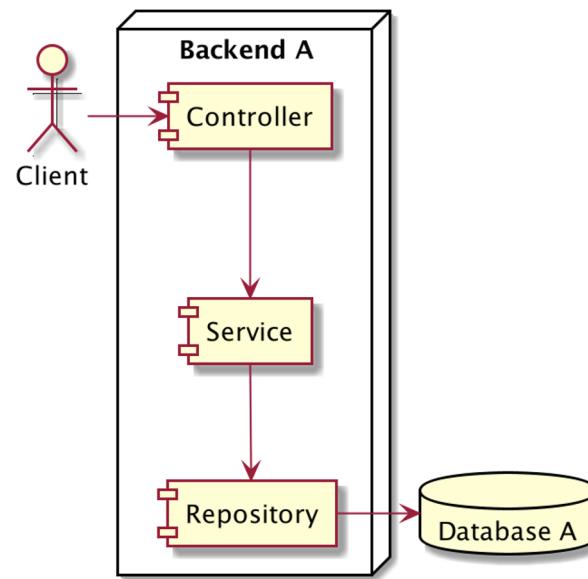
UI Test / End to End Test



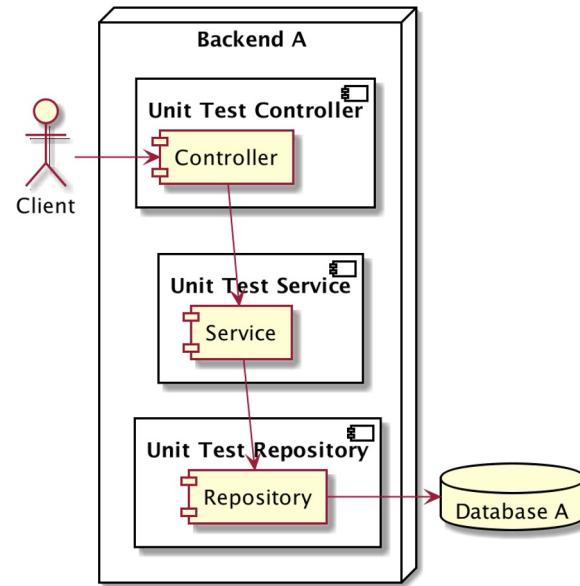
Service Test / Integration Test



Contoh Internal Architecture Aplikasi



Unit Test



Unit Test

- Unit test akan fokus menguji bagian kode program terkecil, biasanya menguji sebuah method
- Unit test biasanya dibuat kecil dan cepat, oleh karena itu biasanya kadang kode unit test lebih banyak dari kode program aslinya, karena semua skenario pengujian akan dicoba di unit test
- Unit test bisa digunakan sebagai cara untuk meningkatkan kualitas kode program kita

Pengenalan JUnit



JUnit

- JUnit adalah test framework yang paling populer di Java
- Saat ini versi terbaru JUnit adalah versi 5
- JUnit 5 membutuhkan Java minimal versi 8
- <https://junit.org/>

Membuat Project Menggunakan Maven

mvn archetype:generate

maven-archetype-quickstart

Menambah JUnit 5 di Apache Maven

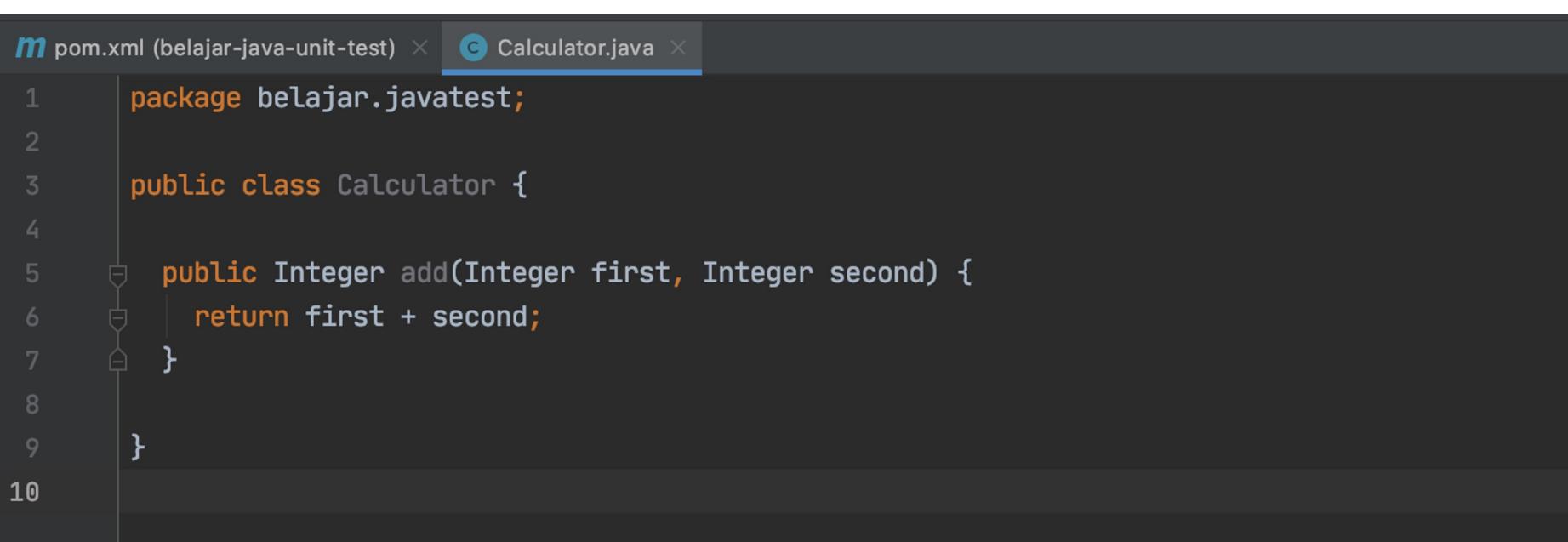
```
 20
 21 <dependencies>
 22   <dependency>
 23     <groupId>org.junit.jupiter</groupId>
 24     <artifactId>junit-jupiter</artifactId>
 25     <version>5.6.2</version>
 26     <scope>test</scope>
 27   </dependency>
 28 </dependencies>
 29
 30 <build>
 31   <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults (may be
```

Membuat Test

Membuat Test

- Untuk membuat test di JUnit itu sederhana, kita cukup membuat class, lalu menambahkan method-method test nya
- Method akan dianggap sebuah test jika ditambahkan annotation @Test
- Kode test disimpan dibagian test folder di maven, bukan di main folder
- Biasanya saat membuat class untuk test, rata-rata orang biasa membuat nama class nya sama dengan nama class yang akan di test, tapi diakhiri dengan kata Test, misal jika nama class nya adalah Calculator, maka nama class test nya adalah CalculatorTest

Kode : Class Calculator



The screenshot shows a Java code editor with two tabs: 'pom.xml (belajar-java-unit-test)' and 'Calculator.java'. The 'Calculator.java' tab is active, indicated by a blue bar and a blue icon. The code in the editor is:

```
1 package belajar.javatest;
2
3 public class Calculator {
4
5     public Integer add(Integer first, Integer second) {
6         return first + second;
7     }
8
9 }
10
```

The code defines a class named 'Calculator' with a single method 'add' that takes two integers as parameters and returns their sum.

Kode : Unit Test Class Calculator

```
3 import org.junit.jupiter.api.Test;  
4  
5 public class CalculatorTest {  
6  
7     private Calculator calculator = new Calculator();  
8  
9     @Test  
10    public void testAddSuccess() {  
11        var result = calculator.add(10, 10);  
12    }  
13}
```

Menggunakan Assertions

Assertions

- Saat membuat test, kita harus memastikan bahwa test tersebut sesuai dengan ekspektasi yang kita inginkan
- Jika manual, kita bisa melakukan pengecekan if else, namun itu tidak direkomendasikan
- JUnit memiliki fitur untuk melakukan assertions, yaitu memastikan bahwa unit test sesuai dengan kondisi yang kita inginkan
- Assertions di JUnit di representasikan dalam class Assertions, dan di dalamnya terdapat banyak sekali function static
- <https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html>

Meng-import Assertions

```
m pom.xml (belajar-java-unit-test) × C Calculator.java × C CalculatorTest.java ×
1 package belajar.javatest;
2
3 import org.junit.jupiter.api.Test;
4 import static org.junit.jupiter.api.Assertions.*;
5
6 public class CalculatorTest {
7
8     private Calculator calculator = new Calculator();
9
10    @Test
11    public void testAddSuccess() {
12
13        assertEquals(3, calculator.add(1, 2));
14    }
15}
```

Menggunakan Assertions

```
5
6  ↗ public class CalculatorTest {
7
8      private Calculator calculator = new Calculator();
9
10     @Test
11    ↗ public void testAddSuccess() {
12        var result = calculator.add(10, 10);
13        assertEquals(expected: 20, result);
14    }
15 }
```

Menggagalkan Test

- Kadang dalam membuat unit test, kita tidak hanya ingin mengetest kasus sukses atau gagal
- Ada kalanya kita ingin mengetes sebuah exception misalnya
- Assertions juga bisa digunakan untuk mengecek apakah sebuah exception terjadi

Kode : Calculator Divide

```
2
3  public class Calculator {
4
5      public Integer divide(Integer first, Integer second) {
6          if (second == 0) {
7              throw new IllegalArgumentException("Can not divide by zero");
8          } else {
9              return first / second;
10         }
11     }
12
13     public Integer add(Integer first, Integer second) {
```

Kode : Assertions Exception

```
17      @Test
18  ►   public void testDivideSuccess() {
19      |     var result = calculator.divide(100, 10);
20      |     assertEquals(expected: 10, result);
21  □   }
22
23      @Test
24  ►   public void testDivideError() {
25      |     assertThrows(IllegalArgumentException.class, () -> {
26      |         calculator.divide(100, 0);
27  □   });
28  □ }
```

Mengubah Nama Test

Mengubah Nama Test

- Kadang agak sulit membuat nama function yang merepresentasikan kasus test nya
- Jika kita ingin menambahkan deskripsi untuk tiap test, kita bisa menggunakan annotation @DisplayName
- Dengan menggunakan annotation @DisplayName, kita bisa menambahkan deskripsi unit testnya

Kode : Menggunakan DisplayName

```
7  
8     @DisplayName("Test Calculator")  
9     public class CalculatorTest {  
10  
11         private Calculator calculator = new Calculator();  
12  
13         @Test  
14         @DisplayName("Test Function Calculator.add(Integer, Integer)")  
15         public void testAddSuccess() {  
16             var result = calculator.add(10, 10);  
17             assertEquals(expected: 20, result);  
18         }  
19     }
```

Menggunakan Display Name Generator

- JUnit mendukung pembuatan DisplayName secara otomatis menggunakan generator
- Yang perlu kita lakukan adalah membuat class turunan dari interface DisplayNameGenerator, lalu menambahkan annotation @DisplayNameGeneration di test class nya

Kode : Display Name Generator

```
7  public class SimpleDisplayNameGenerator implements DisplayNameGenerator {  
8  
9      @Override  
10     @I@ public String generateDisplayNameForClass(Class<?> testClass) {  
11         return "Test " + testClass.getSimpleName();  
12     }  
13  
14     @Override  
15     @I@ public String generateDisplayNameForMethod(Class<?> testClass, Method testMethod) {  
16         return "Test " + testMethod.getName();  
17     }  
18 }
```

Kode : Display Name Generation

```
6
7     import static org.junit.jupiter.api.Assertions.*;
8
9     @DisplayNameGeneration(value = SimpleDisplayNameGenerator.class)
10    public class CalculatorTest {
11
12        private Calculator calculator = new Calculator();
13
14        @Test
15        public void testAddSuccess() {
16            var result = calculator.add(10, 10);
17            assertEquals(expected: 20, result);
18        }
}
```

Menonaktifkan Test

Menonaktifkan Test

- Kadang ada kalanya kita ingin menonaktifkan unit test, misal karena terjadi error di unit test tersebut, dan belum bisa kita perbaiki
- Sebenarnya cara paling mudah untuk menonaktifkan unit test adalah dengan menghapus annotation `@Test`, namun jika kita lakukan itu, kita tidak bisa mendeteksi kalo ada unit test yang di disabled
- Untuk menonaktifkan unit test secara benar, kita bisa menggunakan annotation `@Disabled`

Kode : Disabled Unit Test

```
30     |    calculator.divide(100, 0),  
31     |    });  
32     }  
33  
34     @Test  
35     @Disabled  
36 ►  public void testComingSoon() {  
37         // TODO coming soon!  
38     }  
39 }
```

Tampilan di IntelliJ IDEA

The screenshot shows the IntelliJ IDEA interface with the 'Run' tool window open. The title bar says 'Run: CalculatorTest'. The tool window displays the following test results:

Test	Time
Test testDivideError	26 ms
Test testAddSuccess	6 ms
Test testComingSoon	0 ms
Test testDivideSuccess	1 ms

Below the table, a code snippet is shown:

```
public void belajar.javatest.CalculatorTest.testComi
```

Sebelum & Setelah Test

Sebelum & Setelah Unit Test

- Kadang kita ingin menjalankan kode yang sama sebelum dan setelah eksekusi unit test
- Hal ini sebenarnya bisa dilakukan secara manual di function @Test nya, namun hal ini akan membuat kode duplikat banyak sekali
- JUnit memiliki annotation @BeforeEach dan @AfterEach
- @BeforeEach digunakan untuk menandai function yang akan dieksekusi sebelum unit test dijalankan
- @AfterEach digunakan untuk menandai function yang akan dieksekusi setelah unit test dijalankan
- Ingat, bahwa ini akan selalu dieksekusi setiap kali untuk function @Test, bukan sekali untuk class test saja

Kode : BeforeEach dan AfterEach

```
10     private Calculator calculator = new Calculator();
11
12     @BeforeEach
13     public void setUp() {
14         System.out.println("Before unit test");
15     }
16
17     @AfterEach
18     public void tearDown() {
19         System.out.println("After unit test");
20     }
21
22     @Test
```



Sebelum & Setelah Semua Unit Test

- @BeforeEach & @AfterEach akan dieksekusi setiap kali function @Test jalan
- Namun kadang kita ingin melakukan sesuatu sebelum semua unit test berjalan, atau setelah semua unit test berjalan
- Ini bisa dilakukan menggunakan annotation @BeforeAll dan @AfterAll
- Namun hanya static function yang bisa menggunakan @BeforeAll dan @AfterAll

Kode : BeforeAll dan AfterAll

```
11  
12     @BeforeAll  
13     public static void beforeAll() {  
14         System.out.println("Before all");  
15     }  
16  
17     @AfterAll  
18     public static void afterAll() {  
19         System.out.println("After all");  
20     }  
21  
22     @BeforeEach  
23     public void setup() {
```

Membatalkan Test

Membatalkan Test

- Kadang kita ingin membatalkan unit test ketika kondisi tertentu terjadi
- Untuk membatalkan, kita bisa menggunakan exception TestAbortedException
- Jika JUnit mendapatkan exception TestAbortedException, secara otomatis test tersebut akan dibatalkan

Kode : Membatalkan Test

```
57
58     @Test
59 ►   public void testAborted() {
60         var profile = System.getenv( name: "PROFILE");
61         if (!"DEV".equals(profile)) {
62             throw new TestAbortedException();
63         }
64
65         // dev unit test
66     }
67 }
```

Ketika Test Dibatalkan

Run: CalculatorTest

| » Tests passed: 3, ignored: 2 of 5 tests – 58 ms

▾ **Test Results** 58 ms /Users/khannedy/Tools/jdk-14.jdk/Contents/Home/bin/ja

▾ **Test CalculatorTest** 58 ms Before all

 Test testDivideError 22 ms

 Test testAborted 21 ms Before unit test

 Test testAddSuccess 6 ms After unit test

 Test testComingSoon 9 ms

 Test testDivideSuccess 9 ms Before unit test

 Test testDivideSuccess 9 ms After unit test

Menggunakan Assumptions

Menggunakan Assumptions

- Sebelumnya kita sudah tahu jika ingin membatalkan test, kita bisa menggunakan exception TestAbortException
- Namun sebenarnya ada cara yang lebih mudah, yaitu dengan menggunakan Assumptions
- Penggunaan Assumptions mirip seperti Assertions, jika nilainya tidak sama, maka function Assumptions akan thrown TestAbortException, sehingga secara otomatis akan membatalkan unit test yang sedang berjalan
- <https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assumptions.html>

Kode : Import Assumptions

```
3 import org.junit.jupiter.api.*;
4 import org.opentest4j.TestAbortedException;
5
6 import static org.junit.jupiter.api.Assertions.*;
7 import static org.junit.jupiter.api.Assumptions.*;
8
9 @DisplayNameGeneration(value = SimpleDisplayNameGenerator.class)
10 public class CalculatorTest {
11
12     private Calculator calculator = new Calculator();
13
14     @BeforeEach
```

Kode : Menggunakan Assumptions

```
66 // dev unit test
67 }
68
69 @Test
70 public void testAssumption() {
71     assumeTrue("DEV".equals(System.getenv("PROFILE")));
72
73 // dev unit test
74 }
75 }
```

Test Berdasarkan Kondisi

Test Berdasarkan Kondisi

- Sebenarnya kita bisa menggunakan Assumptions untuk menjalankan unit test berdasarkan kondisi tertentu
- Namun JUnit menyediakan fitur yang lebih mudah untuk pengecekan beberapa kondisi, seperti kondisi sistem operasi, versi java, system property atau environment variable
- Ini lebih mudah dibandingkan menggunakan Assumptions

Kondisi Sistem Operasi

- Untuk kondisi sistem operasi, kita bisa menggunakan beberapa annotation
- @EnabledOnOs digunakan untuk penanda bahwa unit test boleh berjalan di sistem operasi yang ditentukan
- @DisabledOnOs digunakan untuk penanda bahwa unit test tidak boleh berjalan di sistem operasi yang ditentukan

Kode : Kondisi Sistem Operasi

```
8
9      @Test
10     @EnabledOnOs(value = {OS.WINDOWS})
11     public void onlyRunOnWindows() {
12         // put your unit test
13     }
14
15     @Test
16     @DisabledOnOs(value = {OS.WINDOWS})
17     public void disabledRunOnWindows() {
18         // put your unit test
19     }
```

Kondisi Versi Java

- Untuk kondisi versi Java yang kita gunakan, kita bisa menggunakan beberapa annotation
- `@EnabledOnJre` digunakan untuk penanda bahwa unit test boleh berjalan di Java versi tertentu
- `@DisabledOnJre` digunakan untuk penanda bahwa unit test tidak boleh berjalan di Java versi tertentu
- `@EnabledForJreRange` digunakan untuk penanda bahwa unit test boleh berjalan di range Java versi tertentu
- `@DisabledForJreRange` digunakan untuk penanda bahwa unit test tidak boleh berjalan di range Java versi tertentu

Kode : Kondisi Versi Java

```
88
89     @Test
90     @EnabledOnJre(value = {JRE.JAVA_14})
91     public void onlyRunOnJava14() {
92         // put your unit test
93     }
94
95     @Test
96     @DisabledOnJre(value = {JRE.JAVA_14})
97     public void disabledRunOnJava14() {
98         // put your unit test
99     }
00
```

Kode : Kondisi Range Versi Java

```
01  @Test
02  @EnabledForJreRange(min = JRE.JAVA_11, max = JRE.JAVA_14)
03  public void onlyRunOnJava11ToJava14() {
04      // put your unit test
05  }
06
07  @Test
08  @DisabledForJreRange(min = JRE.JAVA_11, max = JRE.JAVA_14)
09  public void disabledRunOnJava11ToJava14() {
10      // put your unit test
11  }
```

Kondisi System Property

- Untuk kondisi nilai dari system property, kita bisa menggunakan beberapa annotation
- `@EnabledIfSystemProperty` untuk penanda bahwa unit test boleh berjalan jika system property sesuai dengan yang ditentukan
- `@DisabledIfSystemProperty` untuk penanda bahwa unit test tidak boleh berjalan jika system property sesuai dengan yang ditentukan
- Jika kondisinya lebih dari satu, kita bisa menggunakan `@EnabledIfSystemProperties` dan `@DisabledIfSystemProperties`

Kode : Kondisi System Property

```
12
13     @Test
14     @EnabledIfSystemProperty(named = "java.vendor", matches = "Oracle Corporation")
15     public void enableOnOracle() {
16         // put your unit test
17     }
18
19     @Test
20     @DisabledIfSystemProperty(named = "java.vendor", matches = "Oracle Corporation")
21     public void disabledOnOracle() {
22         // put your unit test
23     }
24
```

Kondisi Environment Variable

- Untuk kondisi nilai dari environment variable, kita bisa menggunakan beberapa annotation
- `@EnabledIfEnvironmentVariable` untuk penanda bahwa unit test boleh berjalan jika environment variable sesuai dengan yang ditentukan
- `@DisabledIfEnvironmentVariable` untuk penanda bahwa unit test tidak boleh berjalan jika environment variable sesuai dengan yang ditentukan
- Jika kondisinya lebih dari satu, kita bisa menggunakan `@EnabledIfEnvironmentVariables` dan `@DisabledIfEnvironmentVariables`

Kode : Kondisi Environment Variable

```
24  
25     @Test  
26     @EnabledIfEnvironmentVariable(named = "PROFILE", matches = "DEV")  
27 ►   public void enabledOnProfileDev() {  
28         // put your unit test  
29     }  
30  
31     @Test  
32     @DisabledIfEnvironmentVariable(named = "PROFILE", matches = "DEV")  
33 ✓   public void disabledOnProfileDev() {  
34         // put your unit test  
35     }  
36
```

Menggunakan Tag

Menggunakan Tag

- Class atau function unit test bisa kita tambahkan tag (tanda) dengan menggunakan annotation @Tag
- Dengan menambahkan tag ke dalam unit test, kita bisa fleksibel ketika menjalankan unit test, bisa memilih tag mana yang mau di include atau di exclude
- Jika kita menambahkan tag di class unit test, secara otomatis semua function unit test di dalam class tersebut akan memiliki tag tersebut
- Jika kita ingin menambahkan beberapa tag di satu class atau function unit test, kita bisa menggunakan annotation @Tags



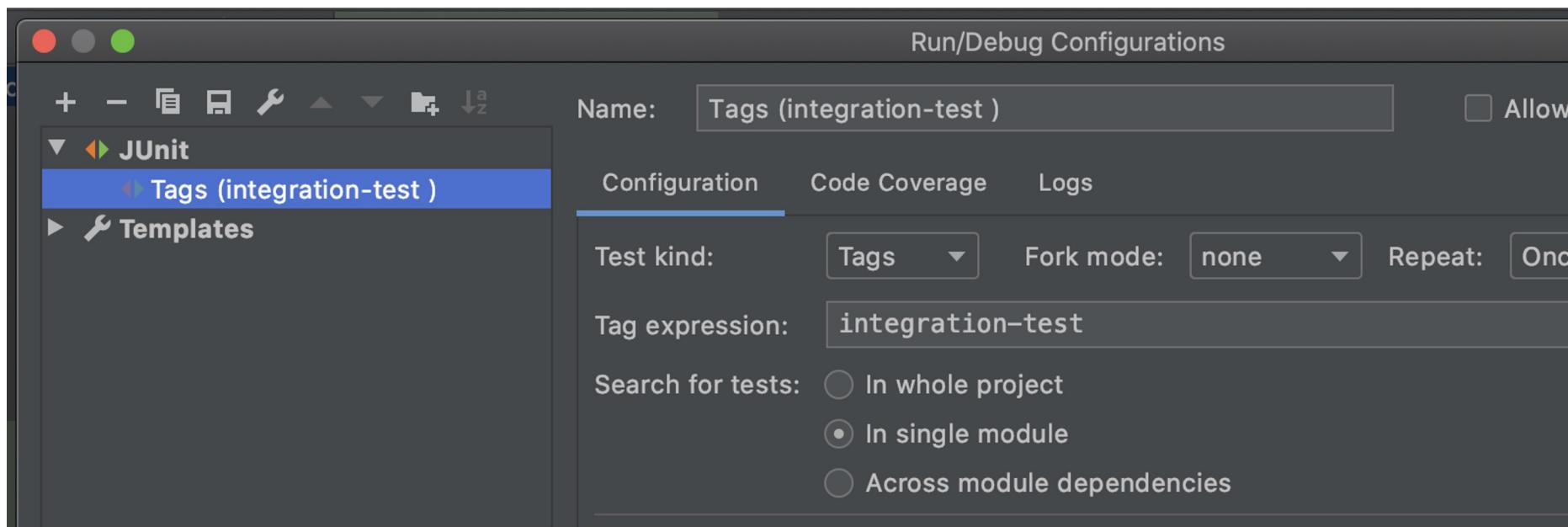
Kode : Menambahkan Tag

```
6  @Tag("integration-test")
7 ➤ public class SampleIntegrationTest {
8
9     @Test
0 ➤     public void integrationTest1() {
1         System.out.println("Integration Test 1");
2     }
3
4     @Test
5 ➤     public void integrationTest2() {
6         System.out.println("Integration Test 2");
7     }
}
```

Memilih Tag dengan Maven

```
mvn test -Dgroups=tag1,tag2
```

Memilih Tag dengan IntelliJ IDEA



Urutan Eksekusi Test

Urutan Eksekusi Test

- Secara default, urutan eksekusi unit test tidak ditentukan, jadi jangan berharap jika sebuah method berada diatas method lainnya, maka akan dijalankan lebih dulu
- Hal ini karena memang sebaiknya method unit test itu harus independen, tidak saling ketergantungan
- Secara default pun, object class unit test akan selalu dibuat ulang tiap method, jadi jangan berharap kita bisa menyimpan data di variable untuk digunakan di unit test method selanjutnya

Mengubah Urutan Eksekusi Test

- JUnit mendukung perubahan urutan eksekusi test jika kita mau menggunakan annotation @TestMethodOrder, ada beberapa cara yang bisa kita lakukan
- Alphanumeric, artinya urutan eksekusi unit test akan diurutkan berdasarkan alphanumeric
- Random, artinya urutan urutan eksekusi unit test akan dieksekusi secara random
- OrderAnnotation, artinya urutan eksekusi unit test akan mengikuti annotation @Order yang ada di tiap method unit test

Kode : Menggunakan Order Annotation

```
8  @TestMethodOrder(value = MethodOrderer.OrderAnnotation.class)
9  ► public class OrderedTest {
10
11    ▷ @Test
12    ▷ @Order(3)
13    ► public void test3() {
14
15      ▷ }
16
17    ▷ @Test
18    ▷ @Order(1)
19    ► public void test1() {
```

Membuat Urutan Sendiri

- Jika kita ingin membuat cara mengurutkan urutan unit test function sendiri, kita bisa dengan mudah tinggal membuat class baru turunan dari MethodOrderer interface

Siklus Hidup Test

Siklus Hidup Test

- Secara default, lifecycle (siklus hidup) object test adalah independent per method test, artinya object unit test akan selalu dibuat baru per method unit test, oleh karena itu kita tidak bisa bergantung dengan method test lain
- Cara pembuatan object test di JUnit ditentukan oleh annotation @TestInstance, dimana defaultnya adalah Lifecycle.PER_METHOD, artinya tiap method akan dibuat sebuah instance / object baru
- Kita bisa merubahnya menjadi Lifecycle.PER_CLASS jika mau, dengan demikian instance / object test hanya dibuat sekali per class, dan method test akan menggunakan object test yang sama
- Hal ini bisa kita manfaatkan ketika membuat test yang tergantung dengan test lain

Kode : Menggunakan Instance Per Class

```
5  @TestInstance(value = TestInstanceLifecycle.PER_CLASS)
6  @TestMethodOrder(value = MethodOrderer.OrderAnnotation.class)
7  public class OrderedTest {
8
9      private int count = 0;
10
11     @Test
12     @Order(3)
13     public void test3() {
14         count++;
15         System.out.println(count);
16     }
}
```

Keuntungan Instance Per Class

- Salah satu keuntungan saat menggunakan Lifecycle.PER_CLASS adalah, kita bisa menggunakan @BeforeAll dan @AfterAll di method biasa, tidak harus menggunakan function object / static

Kode : BeforeAll dan AfterAll

```
9  private int count = 0;  
0  
1  @BeforeAll  
2  public void beforeAll() {  
3  
4  }  
5  
6  @AfterAll  
7  public void afterAll() {  
8  
9  }
```

Test di dalam Test

Test di dalam Test

- Saat membuat unit test, ada baiknya ukuran test class nya tidak terlalu besar, karena akan sulit dibaca dan dimengerti.
- Jika test class sudah semakin besar, ada baiknya kita pecah menjadi beberapa test class, lalu kita grouping sesuai dengan jenis method test nya.
- JUnit mendukung pembuatan class test di dalam class test, jadi kita bisa memecah sebuah class test, tanpa harus membuat class di file berbeda, kita bisa cukup menggunakan inner class
- Untuk memberi tahu bahwa inner class tersebut adalah test class, kita bisa menggunakan annotation @Nested

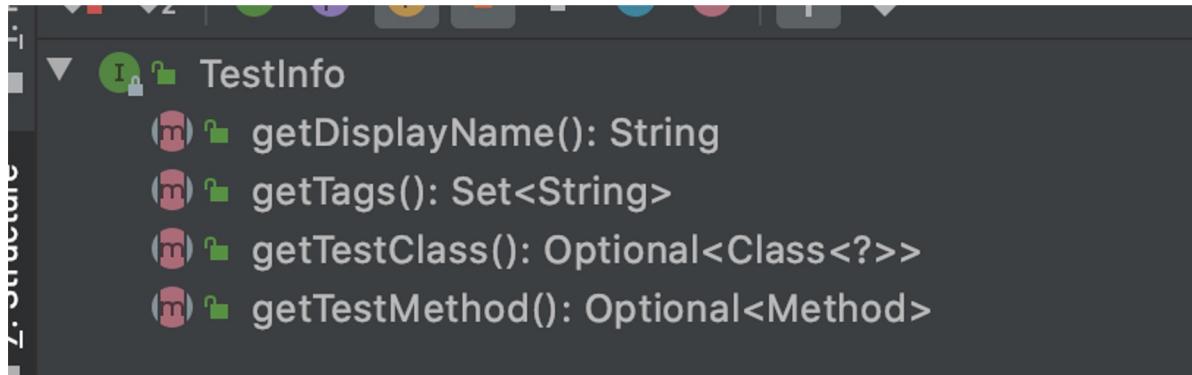
Kode : Test di dalam Test

```
13  @DisplayName("A Queue")
14  public class QueueTest {
15
16      private Queue<String> queue;
17
18      @Nested
19      @DisplayName("when new")
20      public class WhenNew {
21
22          @BeforeEach
23          public void setUp() {
24              queue = new LinkedList<>();
```

Informasi Test

Informasi Test

- Walaupun mungkin jarang kita gunakan, tapi kita juga bisa mendapatkan informasi test yang sedang berjalan menggunakan interface TestInfo
- Kita bisa menambahkan TestInfo sebagai parameter di function unit test



Kode : Menggunakan Test Info

```
8      @DisplayName("Test with TestInfo")
9  ➤  public class InformationTest {
10
11    @Test
12    @Tag("cool")
13    @DisplayName("This is test one")
14  ➤ @  public void testOne(TestInfo testInfo){
15      System.out.println(testInfo.getDisplayName());
16      System.out.println(testInfo.getTags());
17      System.out.println(testInfo.getTestMethod());
18      System.out.println(testInfo.getTestClass());
19
```

Dependency Injection di Test

Dependency Injection di Test

- Tidak ada magic di JUnit, sebenarnya fitur TestInfo yang sebelumnya kita bahas adalah bagian dari dependency injection di JUnit
- Dependency Injection sederhananya adalah bagaimana kita bisa memasukkan dependency (object-instance) ke dalam unit test secara otomatis tanpa proses manual
- Saat kita menambah parameter di function unit test, sebenarnya kita bisa secara otomatis memasukkan parameter tersebut dengan bantuan ParameterResolver
- Contohnya TestInfo yang kita bahas sebelumnya, sebenarnya objectnya dibuat oleh TestInfoParameterResolver

Kode : Membuat Random Parameter Resolver

```
10 | public class RandomParameterResolver implements ParameterResolver {  
11 |  
12 |     private Random random = new Random();  
13 |  
14 |     @Override  
15 |     public boolean supportsParameter(ParameterContext parameterContext, ExtensionContext exten  
16 |         return parameterContext.getParameter().getType() == Random.class;  
17 |     }  
18 |  
19 |     @Override  
20 |     public Object resolveParameter(ParameterContext parameterContext, ExtensionContext exten  
21 |         return random;
```

Menggunakan Parameter Resolver

- Untuk menggunakan parameter resolver yang sudah kita buat, kita bisa menggunakan annotation `@ExtendWith` di test class
- Jika lebih dari 1 parameter resolver, kita bisa menggunakan `@Extentions`

Kode : Menggunakan Random Resolver

```
10  
11 @Extensions(value = {@ExtendWith(RandomParameterResolver.class)})  
12 ► public class RandomCalculatorTest {  
13  
14     private Calculator calculator = new Calculator();  
15  
16     @Test  
17     @ public void testRandom(Random random) {  
18         var a = random.nextInt();  
19         var b = random.nextInt();  
20         assertEquals(expected: a + b, calculator.add(a, b));  
21     }  
22 }
```

Pewarisan di Test

Pewarisan di Test

- JUnit mendukung pewarisan di test, artinya jika kita membuat class atau interface dan menambahkan informasi test disitu, maka ketika kita membuat turunannya, secara otomatis semua fitur test nya dimiliki oleh turunannya
- Ini sangat cocok ketika kita misal contohnya sering membuat code sebelum dan setelah test yang berulang-ulang, sehingga dibanding dibuat di semua test class, lebih baik dibuat sekali di parent test class, dan test class tinggal menjadi child class dari parent test class

Kode : Membuat Parent Test Class

```
7  @Extensions(value = {@ExtendWith(RandomParameterResolver.class)})  
8  public class ParentCalculatorTest {  
9  
10     protected Calculator calculator = new Calculator();  
11  
12     @BeforeEach  
13     public void setUp() {  
14         System.out.println("Before each");  
15     }  
16 }  
17 |
```

Kode : Membuat Child Test Class

```
10  
11 ►  public class RandomCalculatorTest extends ParentCalculatorTest {  
12  
13     @Test  
14 ► @    public void testRandom(Random random) {  
15         var a = random.nextInt();  
16         var b = random.nextInt();  
17         assertEquals( expected: a + b, calculator.add(a, b));  
18     }  
19 }  
20 |
```

Test Berulang

Test Berulang

- JUnit mendukung eksekusi unit test berulang kali sesuai dengan jumlah yang kita tentukan
- Untuk mengulang eksekusi unit test, kita bisa menggunakan annotation @RepeatedTest di method unit test nya
- @RepeatedTest juga bisa digunakan untuk mengubah detail nama test nya, dan kita bisa menggunakan value {displayName} untuk mendapatkan display name, {currentRepetition} untuk mendapatkan perulangan ke berapa saat ini, dan {totalRepetitions} untuk mendapatkan total perulangan nya

Kode : Test Berulang

```
12
13  public class RandomCalculatorTest extends ParentCalculatorTest {
14
15      @DisplayName("Test Calculator Random")
16      @RepeatedTest(
17          value = 10,
18          name = "{displayName} ke {currentRepetition} dari {totalRepetitions}"
19      )
20      @
21      public void testRandom(Random random) {
22          var a = random.nextInt();
23          var b = random.nextInt();
24          assertEquals( expected: a + b, calculator.add(a, b));
25      }
26  }
```

Informasi Perulangan

- @RepeatedTest juga memiliki object RepetitionInfo yang di inject oleh class RepetitionInfoParameterResolver, sehingga kita bisa mendapatkan informasi RepetitionInfo melalui parameter function unit test



Kode : Informasi Perulangan

```
@DisplayName("Test Calculator Random")
@RepeatedTest(
    value = 10,
    name = "{displayName} ke {currentRepetition} dari {totalRepetitions}"
)
public void testRandom(Random random, TestInfo testInfo, RepetitionInfo repetitionInfo) {
    System.out.println(testInfo.getDisplayName() + " ke " + repetitionInfo.getCurrentRepetition());
    var a = random.nextInt();
    var b = random.nextInt();
    assertEquals( expected: a + b, calculator.add(a, b));
}
```

Test dengan Parameter

Test dengan Parameter

- Sebelumnya kita sudah tau jika ingin menambahkan parameter di function unit test, maka kita perlu membuat ParameterResolver
- Namun jika terlalu banyak membuat ParameterResolver juga agak menyulitkan kita
- JUnit memiliki fitur yang bernama @ParameterizedTest, dimana jenis unit test ini memang khusus dibuat agar dapat menerima parameter
- Yang perlu kita lakukan adalah dengan mengganti @Test menjadi @ParameterizedTest

Sumber Parameter

@ParameterizedTest mendukung beberapa sumber parameter, yaitu

- @ValueSource, untuk sumber Number, Char, Boolean dan String
- @EnumSource, untuk sumber berupa enum
- @MethodSource, untuk sumber dari static method
- @CsvSource, untuk sumber berupa data CSV
- @CsvFileSource, untuk sumber berupa file CSV
- @ArgumentSource, untuk data dari class ArgumentProvider

Kode : Parameter dengan @ValueSource

```
public class RandomCalculatorTest extends ParentCalculatorTest {  
  
    @DisplayName("Test Calculator with Parameter")  
    @ParameterizedTest(name = "{displayName} with data {0}")  
    @ValueSource(ints = {1, 2, 3, 4, 5})  
    public void testWithParameter(int value) {  
        var result = value + value;  
        assertEquals(result, calculator.add(value, value));  
    }  
  
    @DisplayName("Test Calculator Random")
```

Kode : Parameter dengan @MethodSource

```
public class RandomCalculatorTest extends ParentCalculatorTest {  
  
    @  
    public static List<Integer> parameterSource() {  
        return List.of(1, 2, 3, 4, 5);  
    }  
  
    @ParameterizedTest  
    @MethodSource(value = {"parameterSource"})  
    public void testWithMethodSource(Integer value) {  
        var result = value + value;  
        assertEquals(result, calculator.add(value, value));  
    }  
}
```

Timeout di Test

Timeout di Test

- Kadang kita ingin memastikan bahwa sebuah unit test berjalan tidak lebih dari sekian detik
- Misal ketika kasus kita ingin memastikan kode program kita mempunyai performa bagus dan cepat
- JUnit memiliki fitur timeout, yaitu memastikan bahwa unit test berjalan tidak lebih dari waktu yang ditentukan, jika melebihi waktu yang ditentukan, secara otomatis unit test tersebut akan gagal
- Kita bisa menggunakan annotation @Timeout untuk melakukan hal tersebut



Kode : Timeout di Test

```
import java.util.concurrent.TimeUnit;  
  
▶ public class SlowTest {  
  
    ▷ @Test  
    ▷     @Timeout(value = 5, unit = TimeUnit.SECONDS)  
    ▷     public void slow() throws InterruptedException {  
        Thread.sleep( millis: 10_000);  
    }  
}  
|
```

Eksekusi Test Secara Paralel

Eksekusi Test Secara Paralel

- Secara default, JUnit tidak mendukung eksekusi unit test secara paralel, artinya unit test akan dijalankan secara sequential satu per satu
- Namun kadang ada kasus kita ingin mempercepat proses unit test sehingga dijalankan secara paralel, hal ini bisa kita lakukan di JUnit, namun perlu beberapa langkah
- Tapi ingat, pastikan unit test kita aman ketika dijalankan secara paralel

Menambah Konfigurasi Paralel

- Hal pertama yang perlu kita lakukan adalah membuat file junit-platform.properties di resource
- Lalu menambah value :
 - junit.jupiter.execution.parallel.enabled = true

Menggunakan @Execution

- Walaupun sudah mengaktifkan fitur paralel, tapi bukan berarti secara otomatis semua unit test berjalan paralel, agar unit test berjalan paralel, kita perlu menggunakan annotation @Execution
- Lalu memilih jenis execution nya, misal untuk paralel bisa menggunakan ExecutionMode.CONCURRENT



Kode : Test Secara Paralel

```
@Execution(value = ExecutionMode.CONCURRENT)
public class SlowTest {

    @Test
    @Timeout(value = 5, unit = TimeUnit.SECONDS)
    public void slow1() throws InterruptedException {
        Thread.sleep( millis: 4_000);
    }

    @Test
    @Timeout(value = 5, unit = TimeUnit.SECONDS)
    public void slow2() throws InterruptedException {
```

Pengenalan Mocking

Ketergantungan Antar Class

- Saat membuat aplikasi yang besar, source code pun akan semakin besar, struktur class pun akan semakin kompleks
- Kita tidak bisa memungkiri lagi bahwa akan ada ketergantungan antar class
- Unit test yang bagus itu bisa terprediksi dan cukup nge test ke satu function, jika harus mengetes interaksi antar class, lebih disarankan integration test
- Lantas bagaimana jika kita harus mengetest class yang ketergantungan dengan class lain?
- Solusinya adalah melakukan mocking terhadap dependency yang dibutuhkan class yang akan kita test

Pengenalan Mocking

- Mocking sederhananya adalah membuat object tiruan
- Hal ini dilakukan agar behavior object tersebut bisa kita tentukan sesuai dengan keinginan kita
- Dengan mocking, kita bisa membuat request response seolah-olah object tersebut benar dibuat

Pengenalan Mockito

- Ada banyak framework untuk melakukan mocking, namun di materi ini kita akan menggunakan Mockito
- Mockito adalah salah satu mocking framework paling populer di Java, dan bisa digunakan juga untuk Kotlin
- Dan Mockito bisa diintegrasikan baik dengan JUnit
- <https://site.mockito.org/>

Kode : Menambah Dependency Mockito

```
24      <artifactId>junit-jupiter</artifactId>
25      <version>5.6.2</version>
26      <scope>test</scope>
27    </dependency>
28    <dependency>
29      <groupId>org.mockito</groupId>
30      <artifactId>mockito-junit-jupiter</artifactId>
31      <version>3.4.4</version>
32      <scope>test</scope>
33    </dependency>
34  </dependencies>
35
36  <build>
```

Kode : Contoh Mocking dengan Mockito

```
15  
16     // Membuat Mock Object  
17     List<String> list = Mockito.mock(List.class);  
18  
19     // Menambah Behaviour ke Mock Object  
20     Mockito.when(list.get(0)).thenReturn("Eko");  
21  
22     // Test Mock  
23     assertEquals( expected: "Eko", list.get(0));  
24  
25     // Verify Mock  
26     Mockito.verify(list, Mockito.times( wantedNumberOfInvocations: 1 )).get(0);  
27
```

Mocking di Test

Mocking di Test

- Mockito memiliki MockitoExtention yang bisa kita gunakan untuk integrasi dengan JUnit
- Hal ini memudahkan kita ketika ingin membuat mock object, kita cukup gunakan @Mock
- Agar terbayang bagaimana proses mock, kita akan coba kasus yang lumayan panjang

Contoh Kasus

- Kita punya sebuah class model dengan nama class Person(id: String, name: String)
- Selanjutnya kita punya interface PersonRepository sebagai interaksi ke database, dan memiliki function selectById(id: String) untuk melakukan mendapatkan data Person di database
- Dan terakhir kita memiliki class PersonService yang digunakan sebagai class bisnis logic, dimana di class tersebut kita akan coba gunakan PersonRepository untuk mendapatkan data dari database, jika gagal, kita akan throw Exception

Kode : Class Person

```
4
5  public class Person {
6
7      private String id;
8
9      private String name;
10
11     public Person(String id, String name) {
12         this.id = id;
13         this.name = name;
14     }
15
```



Kode : Interface PersonRepository

PersonRepository.java 

```
1 package belajar.javatest.repository;  
2  
3 import belajar.javatest.model.Person;  
4  
5 public interface PersonRepository {  
6  
7     Person selectById(String id);  
8  
9 }  
10 |
```

Kode : Class PersonService

```
9
10    public PersonService(PersonRepository personRepository) {
11        this.personRepository = personRepository;
12    }
13
14    public Person get(String id) {
15        var person = personRepository.selectById(id);
16        if (person != null) {
17            return person;
18        } else {
19            throw new IllegalArgumentException("Person not found");
20        }
21    }
```

Kode : Integrasi Mockito dan JUnit

```
12  @ExtendWith(MockitoExtension.class)
13  public class PersonServiceTest {
14
15      @Mock
16      private PersonRepository personRepository;
17
18      private PersonService personService;
19
20      @BeforeEach
21      public void setUp() {
22          personService = new PersonService(personRepository);
23      }
```

Kode : Test Get Person

```
@Test
public void testGetNotFound() {
    assertThrows(IllegalArgumentException.class, () -> personService.get("not found"));
}

@Test
public void testGetSuccess() {
    when(personRepository.selectById(id: "eko")).thenReturn(new Person(id: "eko", name: "Eko"));
    var person = personService.get("eko");
    assertEquals(expected: "eko", person.getId());
    assertEquals(expected: "Eko", person.getName());
}
```

Verifikasi di Mocking

Verifikasi di Mocking

- Pada materi sebelumnya, kita tidak melakukan verifikasi terhadap object mocking, apakah dipanggil atau tidak
- Pada kasus sebelumnya mungkin tidak terlalu berguna karena kebetulan function nya mengembalikan value, sehingga kalo kita lupa memanggil method nya, sudah pasti unit test nya gagal
- Lantas bagaimana jika function nya tidak mengembalikan value? Alias function unit

Contoh Kasus

- Kita akan melanjutkan kasus sebelumnya
- Di interface PersonRepository kita akan membuat method insert(person: Person) yang digunakan untuk menyimpan data ke database, namun tidak mengembalikan value, alias void
- Di class PersonService kita akan membuat method register(name: String) dimana akan membuat object Person dengan id random, lalu menyimpan ke database via PersonRepository, lalu mengembalikan object person tersebut



Kode : Interface PersonRepository

PersonRepository.java 

```
1 package belajar.javatest.repository;  
2  
3 import belajar.javatest.model.Person;  
4  
5 public interface PersonRepository {  
6  
    Person selectById(String id);  
8  
    void insert(Person person);  
10  
11 }
```

Kode : Class PersonService

```
21     }  
22 }  
23 }  
24  
25 public Person register(String name) {  
26     var person = new Person(UUID.randomUUID().toString(), name);  
27     personRepository.insert(person);  
28     return person;  
29 }  
30 }  
31 }
```

Kode : Unit Test (Sebenarnya Salah)

```
38     }
39
40     @Test
41     public void testCreateSuccess() {
42         var person = personService.register( name: "Eko");
43         assertEquals( expected: "Eko", person.getName());
44         assertNotNull(person.getId());
45     }
46 }
```

Kenapa Salah?

- Coba hapus kode personRepository.insert(person)
- Maka unit test nya pun tetap sukses
- Hal ini terjadi karena, kita tidak melakukan verifikasi bahwa mocking object dipanggil
- Hal ini sangat berbahaya, karena jika code sampai naik ke production, bisa jadi orang yang registrasi datanya tidak masuk ke database

Kode : Unit Test dengan Verifikasi

```
38
39
40 @Test
41 public void testCreateSuccess() {
42     var person = personService.register( name: "Eko");
43     assertEquals( expected: "Eko", person.getName());
44     assertNotNull(person.getId());
45
46     verify(personRepository, times( wantedNumberOfInvocations: 1))
47         .insert(new Person(person.getId(), person.getName()));
48 }
49 }
```

Materi Selanjutnya

Materi Selanjutnya

- Java Stream
- Java Database