



Kotlin Unit Test

Eko Kurniawan Khannedy



License

- Dokumen ini boleh Anda gunakan atau ubah untuk keperluan non komersial
- Tapi Anda wajib mencantumkan sumber dan pemilik dokumen ini
- Untuk keperluan komersial, silahkan hubungi pemilik dokumen ini

Eko Kurniawan Khannedy

- Technical architect at one of the biggest ecommerce company in Indonesia
- 10+ years experiences
- youtube.com/c/ProgrammerZamanNow



Pengenalan Software Testing



Sebelum Belajar Materi Ini

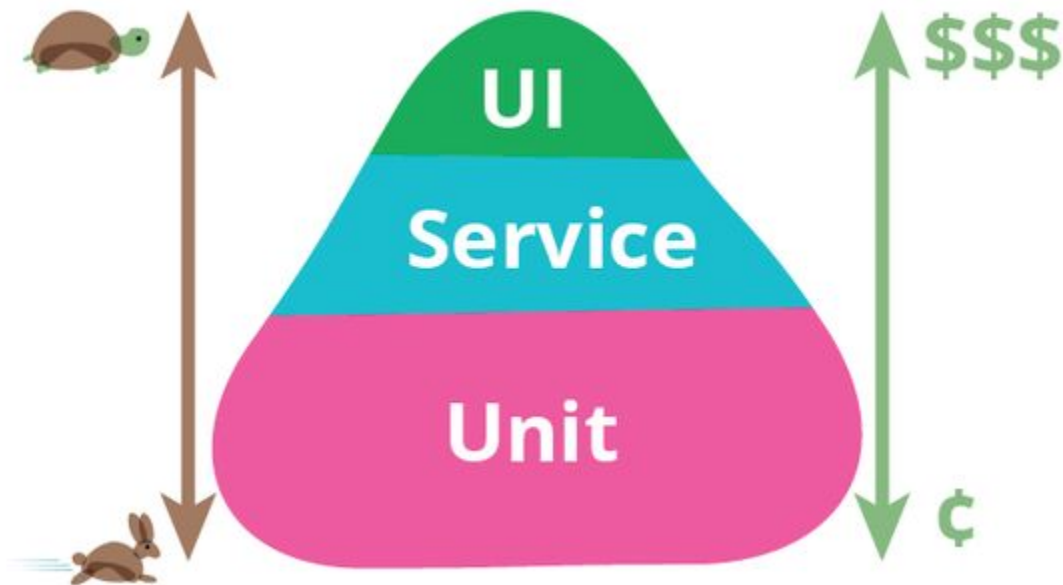
- Kotlin Dasar
- Kotlin Object Oriented Programming
- Kotlin Generic
- Kotlin Collection
- Gradle
- <https://www.udemy.com/course/pemrograman-kotlin-pemula-sampai-mahir/?referralCode=98BE2E779EB8A0BEC230>



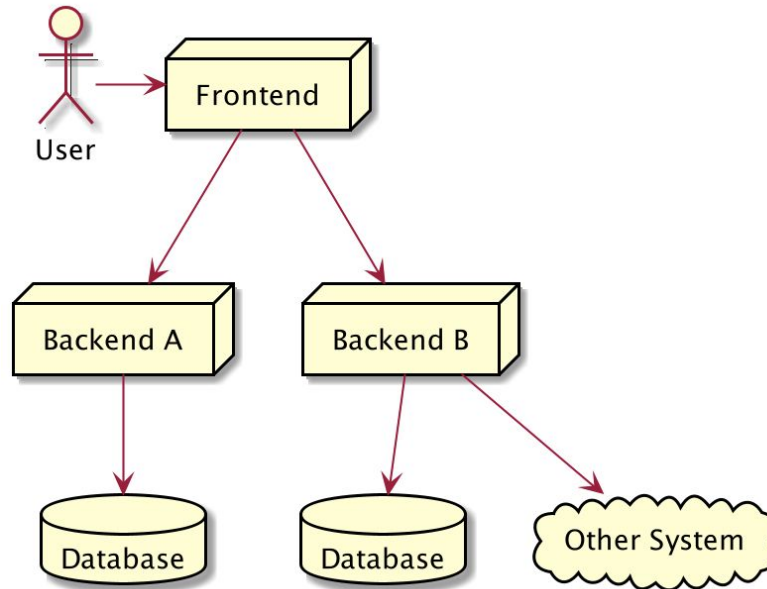
Pengenalan Software Testing

- 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

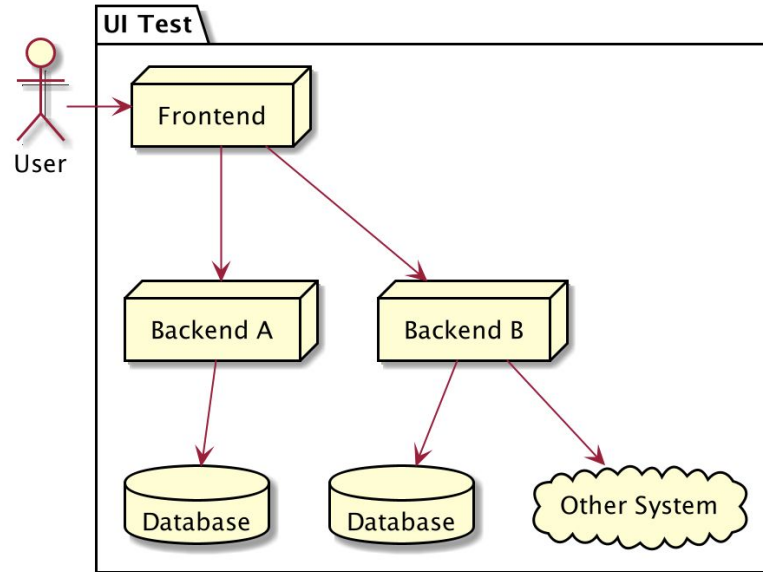
Test Pyramid



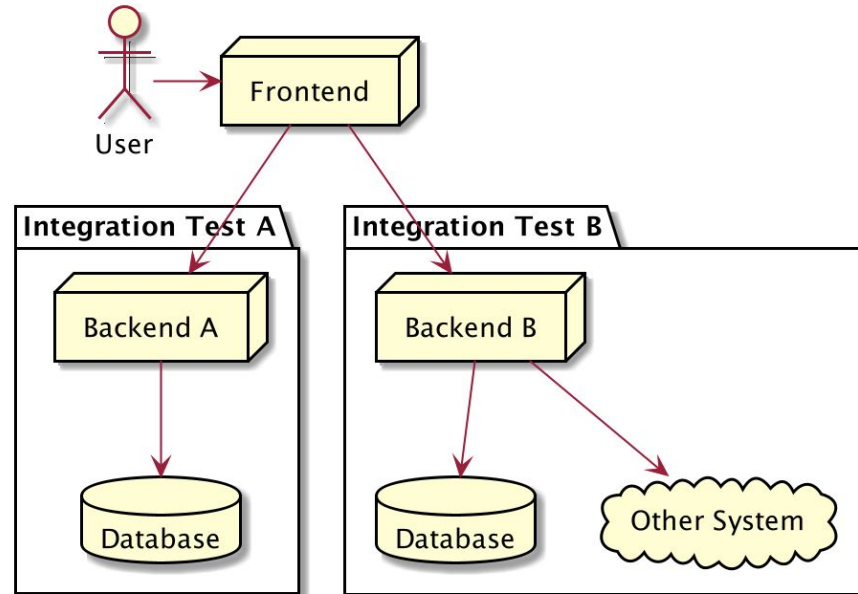
Contoh High Level Architecture Aplikasi



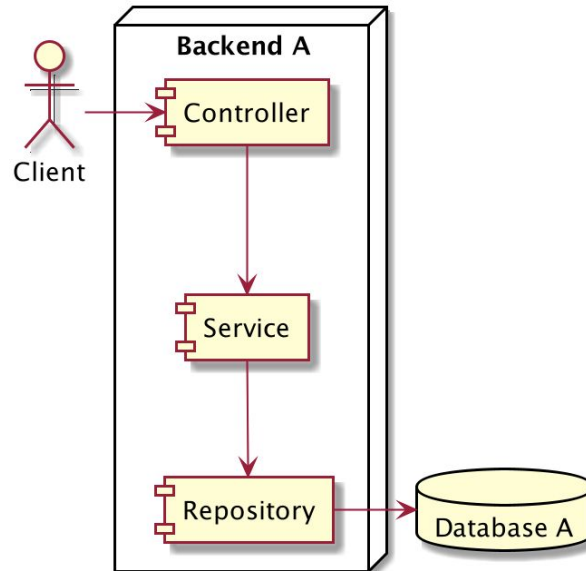
UI Test / End to End Test



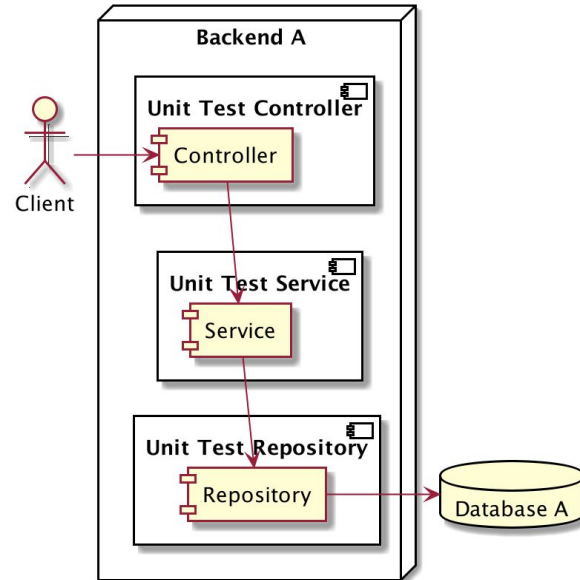
Service Test / Integration Test



Contoh Internal Architecture Aplikasi



Unit Test





Unit Test

- Unit test akan fokus menguji bagian kode program terkecil, biasanya menguji sebuah function
- 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



Kotlin Test

- Kotlin sendiri mendukung pembuatan menggunakan kotlin test
- Kotlin test sendiri sebenarnya di belakangnya adalah sebuah test framework yang bernama JUnit
- <https://kotlinlang.org/api/latest/kotlin.test/>



JUnit

- JUnit adalah test framework yang paling populer di Java
- Saat ini versi terbaru JUnit adalah versi 5, sedangkan sayangnya, yang digunakan oleh Kotlin Test adalah JUnit versi 4
- Di course Kotlin Unit Test ini, kita tidak akan menggunakan Kotlin Test (JUnit 4), namun kita akan menggunakan JUnit 5, karena fitur JUnit 5 sudah lebih lengkap dan canggih dibanding JUnit 4
- <https://junit.org/>



Membuat Project Menggunakan Gradle

```
→ belajar-kotlin-unit-test gradle init
Starting a Gradle Daemon (subsequent builds will be faster)

Select type of project to generate:
  1: basic
  2: application
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4] 2

Select implementation language:
  1: C++
  2: Groovy
  3: Java
  4: Kotlin
  5: Swift
Enter selection (default: Java) [1..5] 4

Select build script DSL:
  1: Groovy
  2: Kotlin
Enter selection (default: Kotlin) [1..2] 2

Project name (default: belajar-kotlin-unit-test):

Source package (default: belajar.kotlin.unit.test):

BUILD SUCCESSFUL in 17s
2 actionable tasks: 2 executed
→ belajar-kotlin-unit-test █
```



Default Kotlin Test

```
25 // Use the Kotlin JDK 8 standard library.
26 implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")
27
28 // Use the Kotlin test library.
29 testImplementation("org.jetbrains.kotlin:kotlin-test")
30
31 // Use the Kotlin JUnit integration.
32 testImplementation("org.jetbrains.kotlin:kotlin-test-junit")
33 }
34
35 application {
36     // Define the main class for the application.
37     mainClassName = "hello.kotlincat.AppKt"
```



Menggunakan JUnit 5 di Project Gradle

```
28     testImplementation("org.junit.jupiter:junit-jupiter:5.6.2")
29 }
30
31 tasks.named<Test>("test"){
32     useJUnitPlatform()
33 }
34
35 tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompile::class).all {
36     kotlinOptions {
37         jvmTarget = "1.8"
38     }
39 }
```

Membuat Test



Membuat Test

- Untuk membuat test di JUnit itu sederhana, kita cukup membuat class, lalu menambahkan function-function test nya
- Function akan dianggap sebuah test jika ditambahkan annotation `@Test`
- Kode test disimpan dibagian test folder di gradle, 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

```
2
3 class Calculator {
4
5     fun add(first: Int, second: Int): Int {
6         return first + second
7     }
8
9 }
```

Kode : Unit Test Class Calculator

```
2
3  import org.junit.jupiter.api.Test
4
5  ▶ class CalculatorTest {
6
7      val calculator = Calculator()
8
9      @Test
10  ▶ fun testAddSuccess() {
11      |     val result = calculator.add(10, 10)
12      | }
13
14  }
```

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 (di kotlin adalah object function)
- Walaupun JUnit dibuat menggunakan Java, tapi JUnit menyediakan function khusus untuk Kotlin
- <https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org.junit.jupiter.api/Assertions.html>

Meng-import Assertions

```
3 import org.junit.jupiter.api.Test
4
5 import org.junit.jupiter.api.Assertions.*
6 import org.junit.jupiter.api.*
7
8 class CalculatorTest {
9
10     val calculator = Calculator()
11
12     @Test
13     fun testAddSuccess() {
14         val result = calculator.add(10, 10)
```

Menggunakan Assertions

```
7
8 class CalculatorTest {
9
10     val calculator = Calculator()
11
12     @Test
13     fun testAddSuccess() {
14         val result = calculator.add(10, 10)
15         assertEquals(20, result)
16     }
17
18
```



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

Code : Calculator Divide

```
3  class Calculator {  
4  
5  fun divide(first: Int, second: Int): Int {  
6      if (second == 0) {  
7          throw IllegalArgumentException("Can not divide by zero")  
8      } else {  
9          return first / second  
10     }  
11 }  
12  
13 fun add(first: Int, second: Int): Int {  
14     return first + second
```

Kode : Assertions Exception

```
25      @Test
26  ▶ fun testDivideSuccess() {
27      val result = calculator.divide(100, 10)
28      assertEquals(10, result)
29  }
30
31      @Test
32  ▶ fun testDivideError() {
33      assertThrows<IllegalArgumentException> {
34          calculator.divide(100, 0)
35      }
36  }
```

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 Test")
9  class CalculatorTest {
10
11      val calculator = Calculator()
12
13      @Test
14      @DisplayName("Test Function Calculator.add(Int, Int)")
15      fun testAddSuccess() {
16          val result = calculator.add(10, 10)
17          assertEquals(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

```
5
6  class SimpleDisplayNameGenerator : DisplayNameGenerator {
7
8  override fun generateDisplayNameForClass(testClass: Class<*>): String {
9      return "Test ${testClass.name}"
10 }
11
12 override fun generateDisplayNameForMethod(testClass: Class<*>?, testMethod: Method): S
13     val simpleName = testMethod.name.replace("_", "")
14     return "Test $simpleName"
15 }
16
17 override fun generateDisplayNameForNestedClass(nestedClass: Class<*>): String {
```

Kode : Display Name Generation

```
7
8  @DisplayNameGeneration(SimpleDisplayNameGenerator::class)
9  class CalculatorTest {
10
11      val calculator = Calculator()
12
13      @Test
14      fun testAddSuccess() {
15          val result = calculator.add(10, 10)
16          assertEquals(20, result)
17      }
18
19      @Test
```

Menonaktifkan Test



Menonaktifkan Test

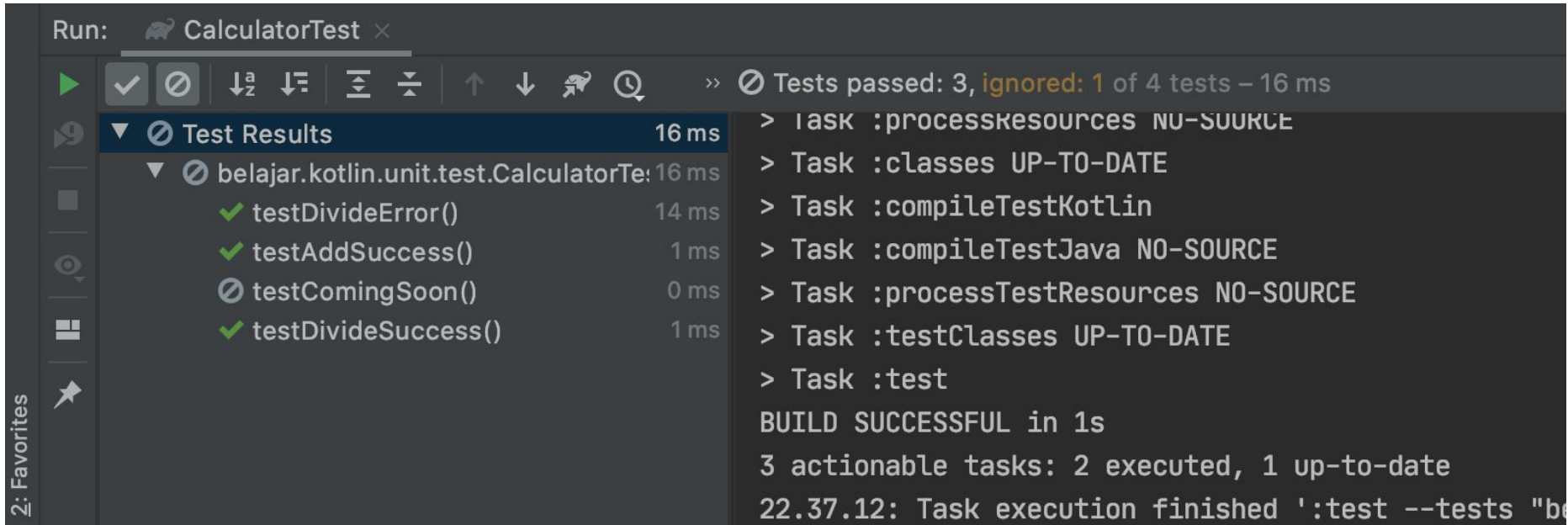
- Kadang ada kalanya kita ingin menonaktifkan unit test, misal karena terjadi error di unit test tersebut, dan belum bisa kita perbaiki
- Sebenarnya cara paling mudah untuk menonaktifkan unit test adalah dengan menghapus annotation `@Test`, namun jika kita lakukan itu, kita tidak bisa mendeteksi kalo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



Kode : Disabled Unit Test

```
30     }  
31  
32     @Test  
33     @Disabled("Coming Soon!")  
34     fun testComingSoon() {  
35         // TODO coming soon  
36     }  
37  
38 }
```

Tampilan di IntelliJ IDEA



The screenshot displays the IntelliJ IDEA Run window for a test named `CalculatorTest`. The window is divided into two main sections: Test Results on the left and the Build/Run output on the right.

Test Results:

- Test Results** (Total: 16 ms)
 - belajar.kotlin.unit.test.CalculatorTe** (Total: 16 ms)
 - ✓ `testDivideError()` (14 ms)
 - ✓ `testAddSuccess()` (1 ms)
 - ⊘ `testComingSoon()` (0 ms)
 - ✓ `testDivideSuccess()` (1 ms)

Build/Run Output:

```
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :compileTestKotlin
> Task :compileTestJava NO-SOURCE
> Task :processTestResources NO-SOURCE
> Task :testClasses UP-TO-DATE
> Task :test
BUILD SUCCESSFUL in 1s
3 actionable tasks: 2 executed, 1 up-to-date
22.37.12: Task execution finished ':test --tests "be
```

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 teset dijalankan
- Ingat, bahwa ini akan selalu dieksekusi setiap kali untuk function `@Test`, bukan sekali untuk class test saja



Kode : BeforeEach dan AfterEach

```
12      val calculator = Calculator()
13
14      @BeforeEach
15      fun setUp() {
16          println("Before unit test")
17      }
18
19      @AfterEach
20      fun tearDown() {
21          println("After unit test")
22      }
23
24      @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 (object function di kotlin) yang bisa menggunakan @BeforeAll dan @AfterAll



Kode : BeforeAll dan AfterAll

```
12 companion object {  
13  
14     @BeforeAll  
15     @JvmStatic  
16     fun setUpAll() {  
17         println("Before all unit test")  
18     }  
19  
20     @AfterAll  
21     @JvmStatic  
22     fun tearDownAll() {  
23         println("After all unit test")
```

Membatalkan Test



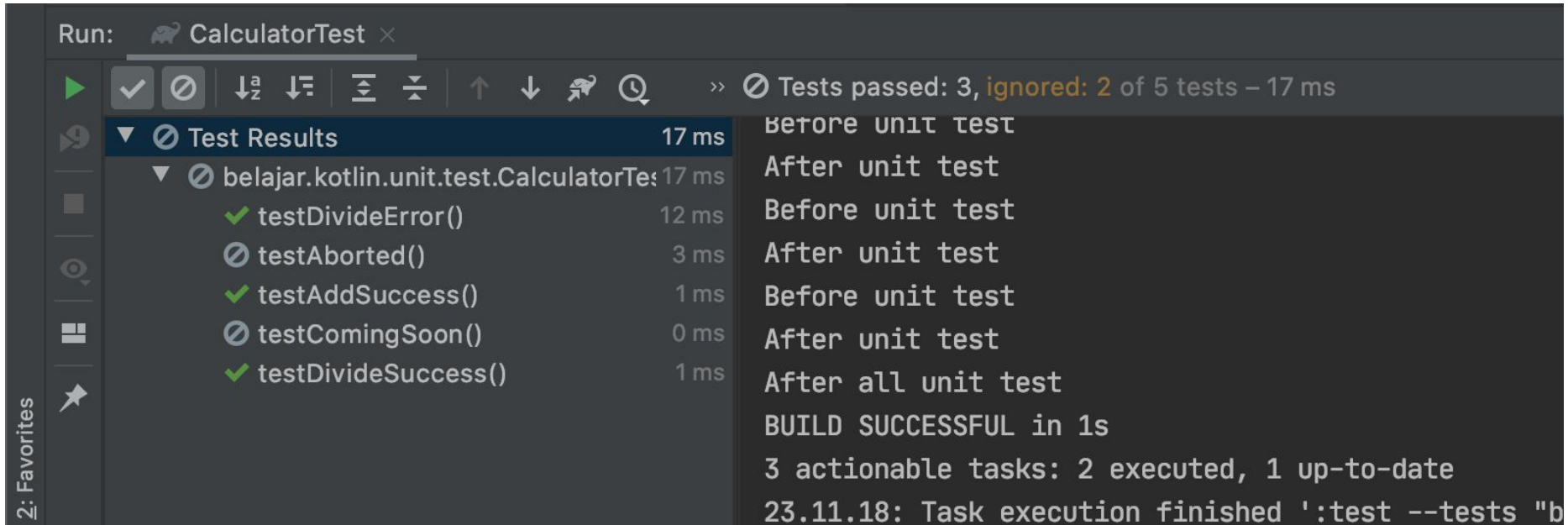
Membatalkan Test

- 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

```
66  @Test
67  ▶ fun testAborted() {
68      val profile = System.getenv()["PROFILE"];
69      if ("DEV" != profile) {
70          throw TestAbortedException()
71      }
72
73      // Batalkan Test
74  }
75
76 }
```


Ketika Test Dibatalkan



Run: CalculatorTest x

Tests passed: 3, ignored: 2 of 5 tests – 17 ms

Test Results 17 ms

- belajar.kotlin.unit.test.CalculatorTest 17 ms
 - ✓ testDivideError() 12 ms
 - ⊘ testAborted() 3 ms
 - ✓ testAddSuccess() 1 ms
 - ⊘ testComingSoon() 0 ms
 - ✓ testDivideSuccess() 1 ms

Before unit test
After unit test
Before unit test
After unit test
Before unit test
After unit test
After all unit test
BUILD SUCCESSFUL in 1s
3 actionable tasks: 2 executed, 1 up-to-date
23.11.18: Task execution finished ':test --tests "b

Menggunakan Assumptions



Menggunakan Assumptions

- Sebelumnya kita sudah tahu jika ingin membatalkan test, kita bisa menggunakan exception `TestAbortedException`
- Namun sebenarnya ada cara yang lebih mudah, yaitu dengan menggunakan `Assumptions`
- Penggunaan `Assumptions` mirip seperti `Assertions`, jika nilainya tidak sama, maka function `Assumptions` akan throw `TestAbortedException`, 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

```
2
3 import org.junit.jupiter.api.Test
4
5 import org.junit.jupiter.api.Assertions.*
6 import org.junit.jupiter.api.*
7 import org.junit.jupiter.api.Assumptions.*
8
9 @DisplayNameGeneration(SimpleDisplayNameGenerator::class)
10 class CalculatorTest {
11
12     val calculator = Calculator()
13
14     @Test
```

Kode : Menggunakan Assumptions

```
64
65
66 ▶ @Test
67   fun testAssumption() {
68       assertTrue("DEV" == System.getenv().get("PROFILE"))
69
70       // Executed if assumptions true
71   }
72
```

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

```
83
84     @Test
85     @EnabledOnOs(value = [OS.WINDOWS])
86     fun onlyRunOnWindows() {
87         // put your unit test
88     }
89
90     @Test
91     @DisabledOnOs(value = [OS.WINDOWS])
92     fun disabledOnWindows() {
93         // put your unit test
94     }
95
```



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

```
93
94     @Test
95     @EnabledOnJre(value = [JRE.JAVA_14])
96     ▶ fun onlyRunOnJava14() {
97         // put your unit test
98     }
99
100    @Test
101    @DisabledOnJre(value = [JRE.JAVA_14])
102    ▶ fun disabledRunOnJava14() {
103        // put your unit test
104    }
```



Kode : Kondisi Range Versi Java

```
05
06     @Test
07     @EnabledForJreRange(min = JRE.JAVA_11, max = JRE.JAVA_14)
08     ▶ fun onlyRunOnJava11ToJava14() {
09         // put your unit test
10     }
11
12     @Test
13     @DisabledForJreRange(min = JRE.JAVA_11, max = JRE.JAVA_14)
14     ▶ fun disabledRunOnJava11ToJava14() {
15         // put your unit test
16     }
```



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

```
17
18     @Test
19     @EnabledIfSystemProperty(named = "java.vendor", matches = "Oracle Corporation")
20     fun enabledOnOracle() {
21         // put your test here
22     }
23
24     @Test
25     @DisabledIfSystemProperty(named = "java.vendor", matches = "Oracle Corporation")
26     fun disabledOnOracle() {
27         // put your test here
28     }
```



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

```
30     @Test
31     @EnabledIfEnvironmentVariable(named = "PROFILE", matches = "DEV")
32     ▶ fun enabledOnProfileDev() {
33         // put your test here
34     }
35
36     @Test
37     @DisabledIfEnvironmentVariable(named = "PROFILE", matches = "DEV")
38     ▶ fun disabledOnProfileDev() {
39         // put your test here
40     }
41
```

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

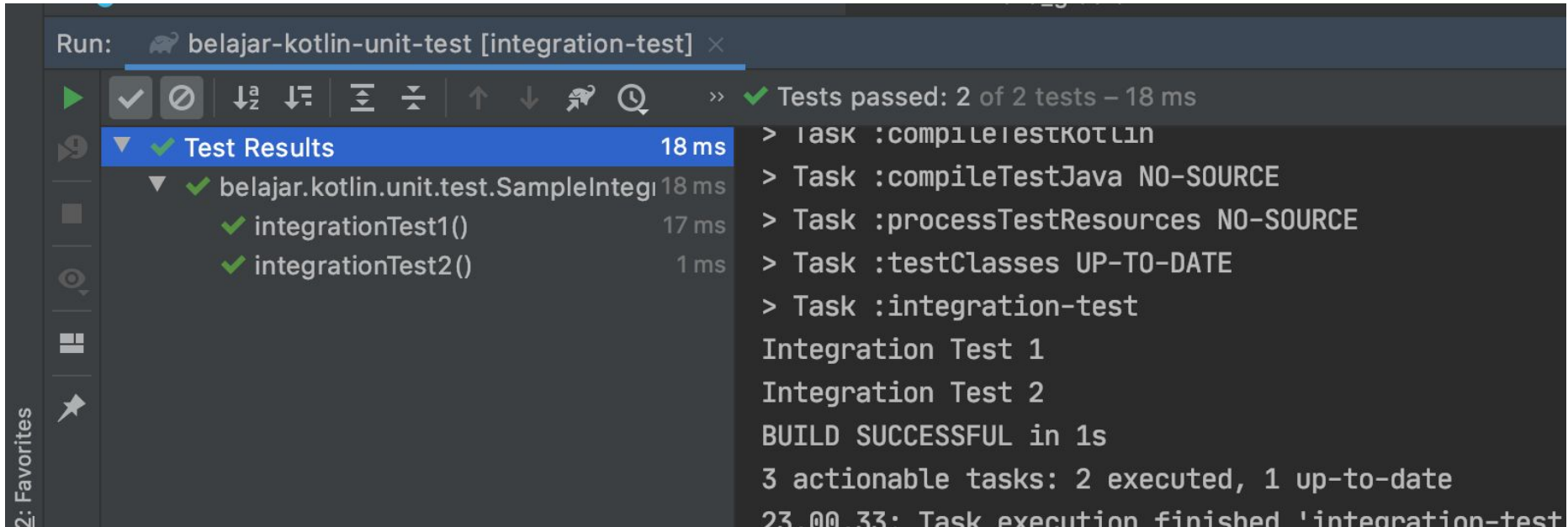
```
5
6  @Tag("integration-test")
7  ▶ class SampleIntegrationTest {
8
9      @Test
10     ▶ fun integrationTest1() {
11         println("Integration Test 1")
12     }
13
14     @Test
15     ▶ fun integrationTest2() {
16         println("Integration Test 2")
17     }
18 }
```



Kode : Membuat Task Test di Gradle

```
30
31 tasks.named<Test>("test") {
32     useJUnitPlatform {
33         excludeTags("integration-test")
34     }
35 }
36
37 tasks.register("integration-test", Test::class) {
38     useJUnitPlatform {
39         includeTags("integration-test")
40     }
41 }
```

Contoh Running Test dengan Tag



The screenshot displays the 'Run' window of an IDE, showing the execution of tests for the project 'belajar-kotlin-unit-test' with the configuration '[integration-test]'. The 'Test Results' tab is active, showing a summary of 2 passed tests in 18 ms. Below the summary, a tree view lists the tests: 'belajar.kotlin.unit.test.SampleIntegri' (18 ms), 'integrationTest1()' (17 ms), and 'integrationTest2()' (1 ms). The right pane shows the build output, including task execution details and a successful build message.

Run: belajar-kotlin-unit-test [integration-test] ×

✓ Tests passed: 2 of 2 tests – 18 ms

▼ ✓ Test Results 18 ms

- ▼ ✓ belajar.kotlin.unit.test.SampleIntegri 18 ms
 - ✓ integrationTest1() 17 ms
 - ✓ integrationTest2() 1 ms

> task :compileTestKotlin
> Task :compileTestJava NO-SOURCE
> Task :processTestResources NO-SOURCE
> Task :testClasses UP-TO-DATE
> Task :integration-test
Integration Test 1
Integration Test 2
BUILD SUCCESSFUL in 1s
3 actionable tasks: 2 executed, 1 up-to-date
23.00.33: Task execution finished 'integration-test'

2: Favorites

Urutan Eksekusi Test



Urutan Eksekusi Test

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

Kode : Menggunakan Order Annotation

```
8      @TestMethodOrder(value = MethodOrderer.OrderAnnotation::class)
9  ▶   class OrderedTest {
10
11       @Test
12       @Order(3)
13  ▶   fun test3() {
14
15       }
16
17       @Test
18       @Order(2)
19  ▶   fun 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 function test, artinya object unit test akan selalu dibuat baru per function unit test, oleh karena itu kita tidak bisa bergantung dengan function test lain
- Cara pembuatan object test di JUnit ditentukan oleh annotation `@TestInstance`, dimana defaultnya adalah `Lifecycle.PER_METHOD`, artinya tiap function 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 function 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

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@TestMethodOrder(value = MethodOrderer.OrderAnnotation::class)
class OrderedTest {

    var count: Int = 0

    @Test
    @Order(3)
    fun test3() {
        count++
        println(count)
    }
}
```



Keuntungan Instance Per Class

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



Kode : BeforeAll dan AfterAll

```
var count: Int = 0
```

```
@BeforeAll
```

```
fun beforeAll(){
```

```
}
```

```
@AfterAll
```

```
fun afterAll(){
```

```
}
```

Test di dalam Test



Test di dalam Test

- Saat membuat unit test, ada baiknya ukuran test class nya tidak terlalu besar, karena akan sulit di baca dan dimengerti.
- Jika test class sudah semakin besar, ada baiknya kita pecah menjadi beberapa test class, lalu kita grouping sesuai dengan jenis function 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`

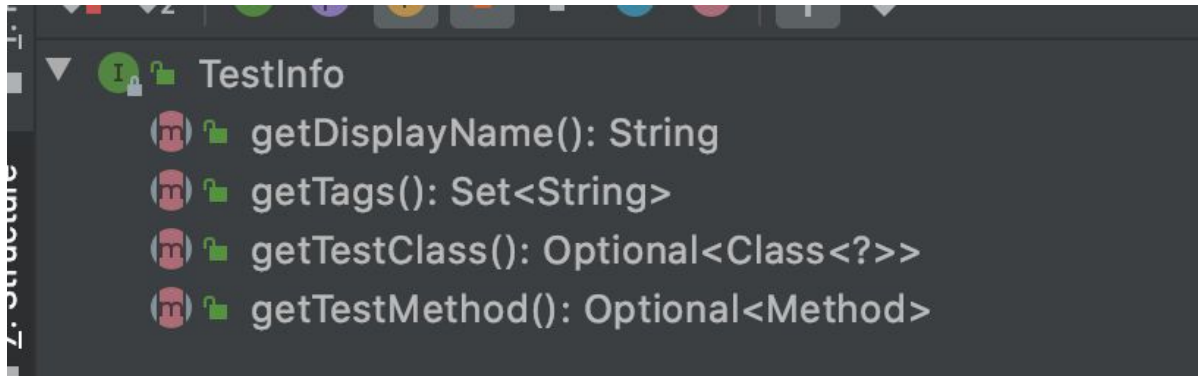
Kode : Test di dalam Test

```
6 @DisplayName("A Queue")
7 class QueueTest {
8
9     lateinit var queue: Queue<String>
10
11     @Nested
12     @DisplayName("when new")
13     inner class WhenNew {
14
15         @BeforeEach
16         fun createNew() {
17             queue = LinkedList<String>()
```

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

```
7
8  @DisplayName("Test with TestInfo")
9  class InformationTest {
10
11      @Test
12      @Tag("cool")
13      @DisplayName("This is test one")
14      fun testOne(testInfo: TestInfo) {
15          println(testInfo.displayName)
16          println(testInfo.tags)
17          println(testInfo.testMethod)
18          println(testInfo.testClass)
19      }
20  }
```

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

```
7
8 class RandomParameterResolver : ParameterResolver {
9
10     private val random: Random = Random()
11
12     override fun supportsParameter(parameterContext: ParameterContext, extensionContext: ExtensionContext): Boolean {
13         return parameterContext.parameter.type == Random::class.java
14     }
15
16     override fun resolveParameter(parameterContext: ParameterContext, extensionContext: ExtensionContext): Any? {
17         return random
18     }
19 }
```




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

```
9  @Extensions(value = [  
10      ExtendWith(RandomParameterResolver::class)  
11  ])  
12  ▶ class RandomCalculatorTest {  
13  
14      private val calculator = Calculator()  
15  
16      @Test  
17  ▶ fun testRandom(random: Random) {  
18      val a = random.nextInt()  
19      val b = random.nextInt()  
20      assertEquals(a + b, calculator.add(a, b))  
}
```

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

```
@Extensions(value = [
    ExtendWith(RandomParameterResolver::class)
])
abstract class ParentCalculatorTest {

    val calculator = Calculator()

    @BeforeEach
    fun beforeEach() {
        println("Before each")
    }
}
```

Kode : Membuat Child Test Class

```
class RandomCalculatorTest : ParentCalculatorTest() {  
  
    @Test  
    fun testRandom(random: Random) {  
        val a = random.nextInt()  
        val b = random.nextInt()  
        assertEquals(a + b, calculator.add(a, b))  
    }  
}
```

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 function 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

```
class RandomCalculatorTest : ParentCalculatorTest() {  
  
    @DisplayName("Test Calculator Random")  
    @RepeatedTest(  
        value = 10,  
        name = "{displayName} ke {currentRepetition} dari {totalRepetitions}"  
    )  
  
    fun testRandom(random: Random) {  
        val a = random.nextInt()  
        val b = random.nextInt()  
        assertEquals(a + b, calculator.add(a, b))  
    }  
}
```



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}"
)
fun testRandom(random: Random, testInfo: TestInfo, repetitionInfo: RepetitionInfo) {
    println("${testInfo.displayName} ke ${repetitionInfo.currentRepetition} dari ${repetitionInfo.totalRepetitions}")
    val a = random.nextInt()
    val b = random.nextInt()
    assertEquals(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 function object (static)
- @CsvSource, untuk sumber berupa data CSV
- @CsvFileSource, untuk sumber berupa file CSV
- @ArgumentSource, untuk data dari class ArgumentProvider

Kode : Parameter dengan @ValueSource

```
class RandomCalculatorTest : ParentCalculatorTest() {  
  
    @DisplayName("Test Calculator with Parameter")  
    @ParameterizedTest(  
        name = "{displayName} with data {0}"  
    )  
    @ValueSource(ints = [1, 2, 3, 4, 5])  
    fun testWithParameter(value: Int) {  
        val result = value + value  
        assertEquals(result, calculator.add(value, value))  
    }  
}
```

Kode : Parameter dengan @MethodSource

```
@ParameterizedTest
@MethodSource(value = ["parameterSource"])
fun testWithMethodSource(value: Int) {
    assertEquals(value + value, calculator.add(value, value))
}

companion object {

    @JvmStatic
    fun parameterSource(): List<Int> {
        return listOf(1, 2, 3, 4, 5)
    }
}
```

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

```
class SlowTest {  
  
    @Test  
    @Timeout(value = 5, unit = TimeUnit.SECONDS)  
    fun slow() {  
        Thread.sleep(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)
class SlowTest {

    @Test
    @Timeout(value = 5, unit = TimeUnit.SECONDS)
    fun slow1() {
        Thread.sleep(4_000)
    }

    @Test
    @Timeout(value = 5, unit = TimeUnit.SECONDS)
    fun slow2() {
```

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 sederhana 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

```
7
8  testImplementation("org.junit.jupiter:junit-jupiter:5.6.2")
9
10 testImplementation("org.mockito:mockito-junit-jupiter:3.4.4")
11 }
12
13 tasks.named<Test>("test") {
14     useJUnitPlatform {
15         excludeTags("integration-test")
16     }
17 }
18 }
```

Kode : Contoh Mocking dengan Mockito

```
11 // membuat mock
12 val listMock = Mockito.mock(List::class.java) as List<String>
13
14 // menambah behaviour di mock object
15 Mockito.`when`(listMock.get(0)).thenReturn("Eko")
16
17 // test mock
18 assertEquals("Eko", listMock.get(0))
19
20 // verify mock
21 Mockito.verify(listMock, Mockito.times(1)).get(0)
22 }
```

Mocking di Test



Mocking di Test

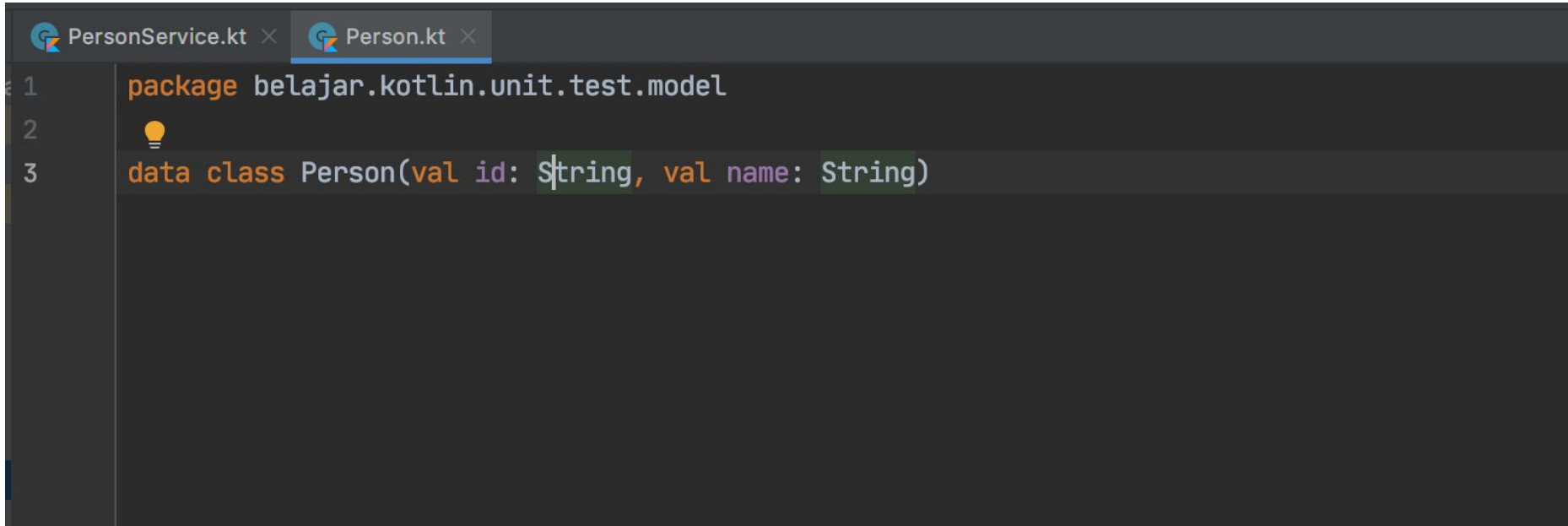
- Mockito memiliki MockitoExtention yang bisa kita gunakan untuk integrasi dengan JUnit
- Hal ini memudahkan kita ketika ingin membuat mock object, kita cukup gunakan @Mock
- Agar terbayang bagaimana proses mock, kita akan coba kasus yang lumayan panjang



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



The screenshot shows an IDE with two tabs: 'PersonService.kt' and 'Person.kt'. The 'Person.kt' tab is active and displays the following Kotlin code:

```
1 package belajar.kotlin.unit.test.model
2
3 data class Person(val id: String, val name: String)
```

Line 2 is empty and features a lightbulb icon, indicating a suggestion or tip. The code defines a data class named 'Person' with two properties: 'id' and 'name', both of type 'String'.

Kode : Interface PersonRepository

```
PersonService.kt × PersonRepository.kt ×
1 package belajar.kotlin.unit.test.repository
2
3 import belajar.kotlin.unit.test.model.Person
4
5 interface PersonRepository {
6
7     fun selectById(id: String): Person?
8
9 }
```

Kode : Class PersonService

```
5
6 class PersonService(private val personRepository: PersonRepository) {
7
8     fun get(id: String): Person {
9         val person = personRepository.selectById(id)
10        if (person != null) {
11            return person
12        } else {
13            throw Exception("Person not found")
14        }
15    }
16
17 }
```

Kode : Integrasi Mockito dan JUnit

```
12  @ExtendWith(value = [MockitoExtension::class])
13  class PersonServiceTest {
14
15      @Mock
16      lateinit var personRepository: PersonRepository
17
18      lateinit var personService: PersonService
19
20      @BeforeEach
21      fun beforeEach() {
22          personService = PersonService(personRepository)
23      }
```

Kode : Test Get Person

@Test

```
fun testGetNotFound() {  
    assertThrows<Exception> { personService.get("not found") }  
}
```

@Test

```
fun testGetSuccess() {  
    Mockito.`when`(personRepository.selectById("eko")).thenReturn(Person("eko", "Eko"))  
    val person = personService.get("eko")  
    assertEquals("eko", person.id)  
    assertEquals("Eko", person.name)  
}
```

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 function 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 function `insert(person: Person)` yang digunakan untuk menyimpan data ke database, namun tidak mengembalikan value, alias unit
- Di class `PersonService` kita akan membuat function `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.kt x
1  package belajar.kotlin.unit.test.repository
2
3  import belajar.kotlin.unit.test.model.Person
4
5  interface PersonRepository {
6
7      fun insert(person: Person)
8
9      fun selectById(id: String): Person?
10
11 }
```

Kode : Class PersonService

```
PersonRepository.kt x PersonService.kt x
3 import belajar.kotlin.unit.test.model.Person
4 import belajar.kotlin.unit.test.repository.PersonRepository
5 import java.util.*
6
7 class PersonService(private val personRepository: PersonRepository) {
8
9     fun register(name: String): Person {
10         val person = Person(UUID.randomUUID().toString(), name)
11         personRepository.insert(person)
12         return person
13     }
```

Kode : Unit Test (Sebenarnya Salah)

```
38     }
39
40     @Test
41     fun testCreateSuccess() {
42         val person = personService.register("Eko")
43         assertEquals("Eko", person.name)
44         assertNotNull(person.id)
45     }
46
47 }
```



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

```
@Test
fun testCreateSuccess() {
    val person = personService.register("Eko")
    assertEquals("Eko", person.name)
    assertNotNull(person.id)

    Mockito.verify(personRepository, Mockito.times(1)).insert(Person(person.id, person.name))
}
```

```
}
```

Materi Selanjutnya



Materi Selanjutnya

- Kotlin Coroutine



Eko Kurniawan Khannedy

- Telegram : @khannedy
- Facebook : fb.com/khannedy
- Twitter : twitter.com/khannedy
- Instagram : instagram.com/programmerzamannow
- Youtube : youtube.com/c/ProgrammerZamanNow
- Email : echo.khannedy@gmail.com