**JAVA UNIT TEST**

## **Pendahuluan**

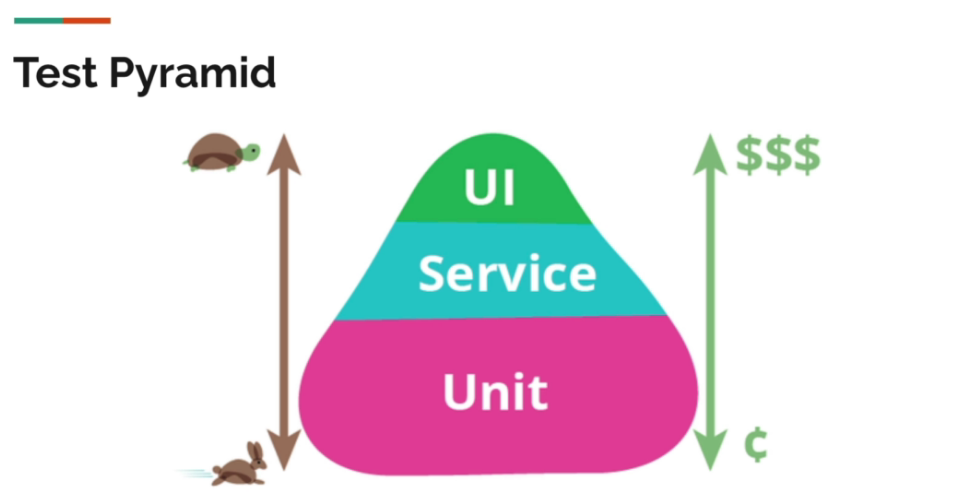
## ## Sebelum Belajar Materi ini ##

## sebaiknya mempelajari 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

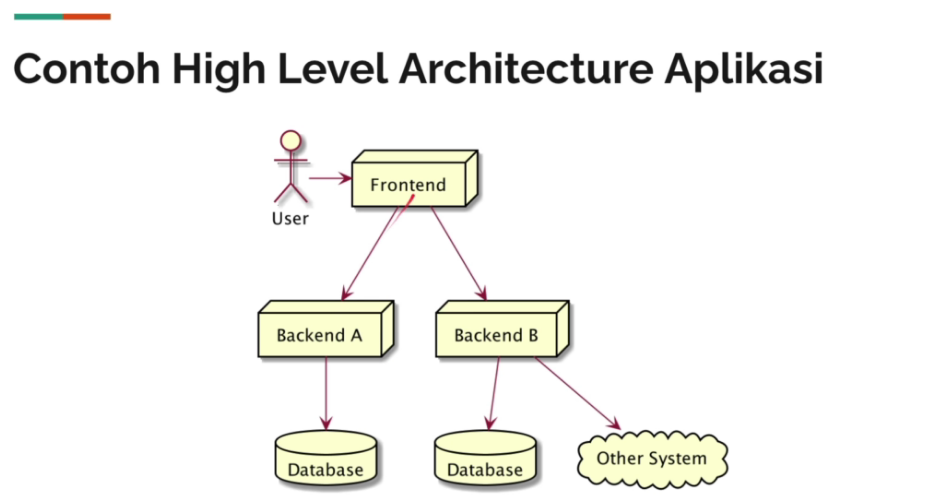
## **Pengenalan Software Testing**

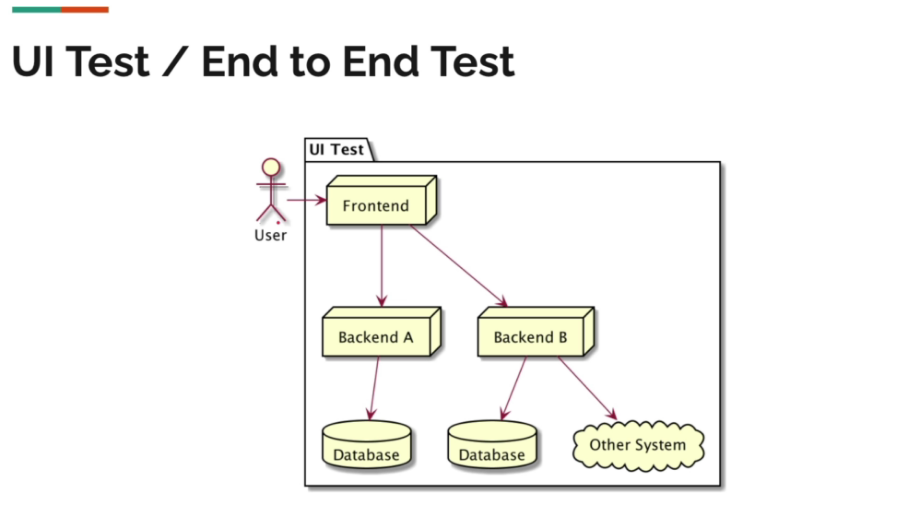


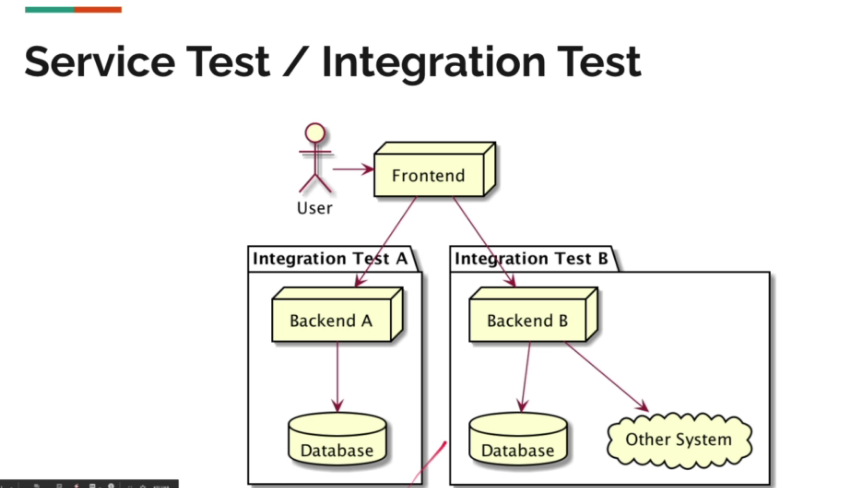
Unit test : testing yang harusnya dibuta sebanyak-banyaknya , yang cepat dan murah

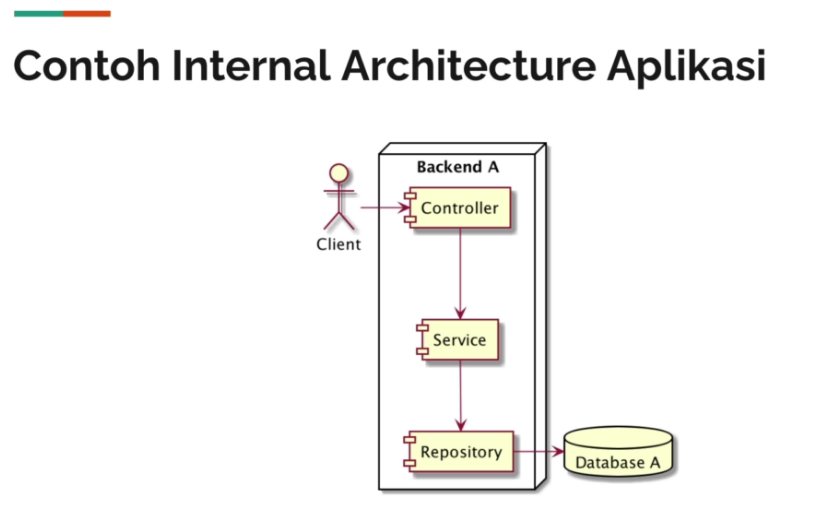
Service : testing pertengahan tidak murah banget, tidak lambat banget juga

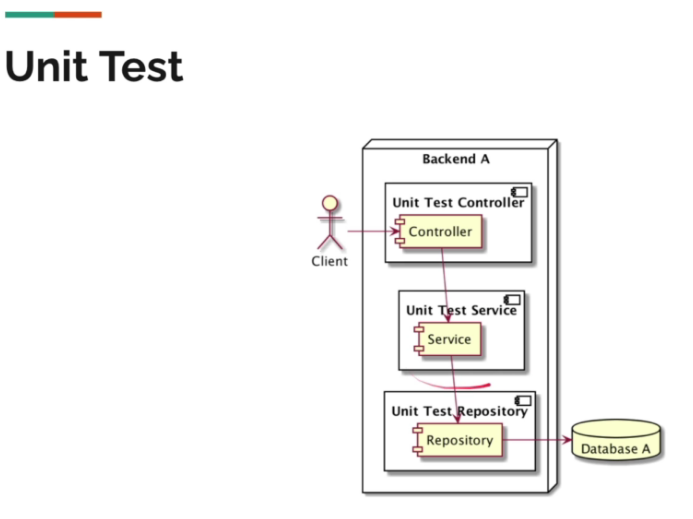
UI : paling lambat pengerjaan, paling mahal











## ## Unit Test ##

## Unit test akan focus menguji bagian kode program terkecil, biasanya menguji sebuah method

* Unit test biasanya dibuat dan cepat, oleh karena itu biasanya kadang kode unit test lebih bnyak dari kode program aslinya, karena semua scenario 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 popular di Java
* Saat ini versi terbaru Junit adalah versi 5
* Junit 5 membutuhkan Java minimal versi 8
* https://junit.org/

## Creatre Maven ##

* mvn archetype:generate
* maven-archetype-quickstart
* Pilih *org.maven.archetypes-maven-archetypes-quickstart*
* Pilih version
* Groupid, biasanya nama atau url organisasi atau perusahaan
* artifacId, nama project bisanya hurus kecil semua dan dipisah dengan tanda “-”
* version yaitu terkait versi project
* package yaitu nama package project , biasanya dipihasahkan dengan titik penamaannya

## **Membuat Test**

* Untuk membuat test di Junit itu sederhana, kita cukup membuat class, lalu menambahkan method-method test nya
* Method akan sianggap 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, missal jika nama class nya adalah Calculator, maka nama class test nya adalah Calculator Test

## Menggagalkan Test ##

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

## **Menggunakan Assertions**

## Assertions ##

Saat membuat test, kita harus memastikan bahwa test tersebut sesuai dengan ekspetasi yang kita inginkan

Jika manual, kitab isa 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

## Mengagalkan 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

## **Mengubah Nama Test**

Kadang agak sulit membuat nama function yang mempresentasikan kasus test nya

Jika kita ingin menambahkan deskripsi untuk tiap test, kitab isa menggunakan annotation @DisplayName

Dengan menggunakan annotation@DisplayName, kitab isa menambahkan deskripsi unit testnya

## 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 @DisplayNameGneneration di test class nya

## **Menonaktifkan Test**

* Kadang adakalanya kita ingin menonaktifkan unit test, missal karena terjadi eror 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 kalua ada unit testnya yang di disabled
* Untuk menonaktifkan unit test secara benar, kita bisa menggunakan annotation @Disabled

## **Sebelum & Setelah Test**

## Sebelum & Setelah Unit Test ##

* Kadang kita ingin menjalankan kode yang sama sebelum dan setelah ekseskusi 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 sebelum unit test dijalankan
* Ingat, bahwa ini akan selalu dieksekusi setiap kali untuk function @Test, bukan sekali untuk class test saja

## Sebelum & Setelah Semua Unit Test ##

* @BeforeEach & @AfterEach akan diekseskusi 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

## **Membatalkan Test**

* Kadang kita ingin membatalkan unit test Ketika kondisi tertentu terjadi
* Untuk membatalkan, kitab isa menggunkan exception TestAbortedException
* Jika JUnit mendapatkan exception TestAbortedException, secara otomatis test tersebut akan dibatalkan

## **Menggunakan Assumptions**

* Sebelumnya kita sudah tahu jika ingin membatalkan test, kitab isa menggunakan exception TestAbortException
* Namun sebenarnya ad acara yang paling 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/5.0.0/api/org/junit/jupiter/api/Assumptions.html>

## **Test Berdasarkan Kondisi**

* Sebenarnya kita bisa menggunakan Assumptions untuk menjalankan unit test berdasarkan kondisi tertentu
* Namun JUnit menyediakan fitur yang lebih mudah untuk menjalankan unit test berdasarkan kondisi tertentu
* Namun JUnit menyediakan fitur yang lebih mudah untuk pengecekan beberapa kondisi, seperti kondisi system 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 sisttem operasi yang ditentukan
* @DisabledOnOs digunakan untuk penanda bahwa unit test tidak boleh berjalan di sistem operasi yang ditentukan

## 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

## Kondisi System Property ##

* Untuk kondisi nilai dari system property, kitab isa 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 menggunkan @EnabledIfSystemProperties dan @DisabledIfSystemIfProperties

## Kondisi Environment Variable ##

* Untuk kondisi nilai dari environment variable, kitab isa 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 varible sesuai dengan yang ditentukan
* Jika kondisinya lebih dari satu, kita bisa menggunakan @EnabledIfEnvironmentVariable dan @DisabledIfEnvironmentVariables

## **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 menjalan 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 annotations @Tags

## Memilih Tag dengan Maven ##  
mvn test -Dgroups=tag1.tag2

## **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 independent, tidak saling ketergantungan
* Secara default pun, object class unit test akan selalu dibuat ulang tiap method, jadi jangan berharap kitab isa menyimpan data di variable untuk digunakan di unit test method selanjutnya

## Mengubah Urutan Ekseskusi 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 tert akan mengikuti annotation @Order yang di tiap method unit test

## Mengubah Urutan Ekseskusi Test ##

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

## **Siklus Hidup Test**

* Secara default, lifecycle (siklus hidup) object test adalah independent per method test, artinya object unit test akan selalu dibuat baru per unit method unit test, oleh karena itu kita tidak bisa bergantung dengan method test lain
* Cara pembuatan object test di JUnit ditentukan oleh annotation @TestInterface, dimana defaultnya adalah Lifecycle. PER METHOD, artinya tiap method akan dibuat sebuah instance/ object baru
* Kita bisa mengubahnya 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 bis akita manfaatkan Ketika membuat test yang tergantung dengan test lain

## Keuntungan Instance Per Class ##

Salah satu keuntungan saat menggunakan Lifecycle.PER\_CLASS adalah, kitab isa menggunakan @BeforeAll dan @AfterAll di method biasa, tidak harus menggunakan function object / static

## **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 kitab isa memecah sebuah class test, tanpa harus membuat class di file berbeda, kitab isa cukup menggunakan inner class
* Untuk memberi tahu bahwa inner class tersebut adalah test class, kitab isa menggunakan annotation @Nested

## **Informasi Test**

* Walaupun mungkin jarang kita gunakan, tapi kita juga bisa mendapatkan informasi test yang sedang berjalan menggunakan interface Test Info
* Kita bisa menambahkan TestInfo sebagai parameter di function unit 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 kitab isa secara otomatis memasukkan parameter di function unit test, sebenarnya kitab isa secara otomatis memasukkan parameter tersebut dengan bantuan ParameterResolver
* Contohnya TestInfo yang kita bahas sebelumnya, sebenarnya objectnya dibuat oleh TestInfoParameterRevolver

## Menggunakan Parameter Resolver ##

* Untuk menggunakan parameter resolver yang sudah kita buat, kitab isa menggunakan annotation @ExtendWith di test class
* Jika lebih dari 1 parameter resolver, kitab isa menggunkan @Extentions

## **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 missal contohnya sering membuat code sebelum dan setelah test yang berulang-ulang, sehingga disbanding dibuat di semua test class, lebih baik dibuat sekali di parent test class, dan test class tinggal menjadi child class dari parent test class

## **Test Berulang**

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

## Informasi Perulangan ##

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

## **Test dengan Parameter**

* Sebelum 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
* @CsvFileSouce, untuk sumber berupa file CSV
* @ArgumentSource, untuk data dari class ArgumentProvider

## **Timeout di Test**

* Kadang kita ingin memastikan bahwa sebuah unit test berjalan 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 tersebut

## **Eksekusi Test Secara Paralel**

* Secara default, JUnit tidak mendukung eksekusi unit tesrt secara parallel, artinya unit test akan dijalankan secara sequential satu per satu
* Namun kadang ada kasus kita ingin mempercepat proses unit test sehingga dijalankan secara parallel, 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 parallel, tapi bukan berarti secara otomatis semua unit test berjalan parallel, agar unit test berjalan parallel kita perlu menggunakan annotation @Execution
* Lalu memilih jenis execution nya, missal untuk parallel bisa menggunakan ExecutionMode.CONCURRENT

## **Pengenalan Mocking**

## Ketergantungan Antar Class ##

* Saat membuat aplikasi yang besat, source code pun akan semakin besar, struktur class pun akan semakin kompleks
* Kita tidak bisa memungkiri lagi bahwa aka nada 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

## Apa itu Mocking ##

* Mocking sederhananya adalah membuat object tiruan
* Hal ini dilakukan agar behavior object tersebut bisa kita tentukan sesuai dengan keinginan kita
* Dengan mocking, kitab isa 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 popular di Java, dan bisa digunakan juga untuk Kotlin
* Dan Mockito bisa diintegrasikan baik dengan JUnit
* https://site.mockito.org/

## **Mocking di Test**

* Mockito memiliki MockitoExtention yang bis akita 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

## **Verifikasi di Mocking**

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

## 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

## 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 database