**JUNIT**

**Test nedir?**

Belirlenen girişlere göre beklenen bir sonuç vermesini denemek için kullanılan geliştirme-deneme yöntemidir.

**Unit Test nedir?**

Unit test veya birim testi programları oluşturulan alt parçaların-birimlerin test edilmesidir. Java tabanlı kodların test edilmesi için kullanılan bir Unit Test – Birim Testi kütüphanesidir, yazılımda en temel test çeşitidir. Yazılan fonksiyonların çalışıp çalışmadığının testlerini yapmak için kullanılır.

**NOT:** Java gibi OOP tabanlı programlama dillerinde bu parçalar metot olarak adlandırılır.

**Neden kullanılır?**

Programlama dilleri temel olarak giriş-input alarak çıktı-output üretir. Girişlerin ve çıkışların kaynağı farklı olsa da aynı girişlerin aynı sonuç vermesi beklenir. Bu beklentiyi doğrulamak için her bir giriş ve çıkış değerinin kontrolünün tek-tek yapılması gerekir. Her bir giriş ve çıkış değerinin tek-tek yapılması yazılım geliştirme sürecini uzatır. JUnit gibi araçlar test işlemlerini kolaylaştırır ve çeşitli ek özellikler sunarak bu süreyi kısaltır.

**NOT:** Unit test yazılan kodların hatalarını bulmak için kullanılmaz. JUnit ayrıca TDD veya test odaklı geliştirme yapmayı sağlar.

TDD yazılım geliştirme öncesi yapılan planın test edilerek kodun yazılmasıdır.

Aşağıda bazı test yöntemleri yer almaktadır.

* Unit Testing
* Integration Testing
* Smoke Testing
* Stress Testing
* Security Testing
* Smoke Testing
* Load Testing

JUnit 5 ile birlikte JUnit Platform, JUnit Jupiter, JUnit Vintage gibi alt parçalara ayrılarak platform haline gelmiştir.

Unit testlerde en büyük avantaj yazılımın geliştirilme hızıdır. Unit testlerin olmadığı bir projeyi düşünelim. Bu projede kullanıcı, bir ürün kaydı yapsın. Proje çalıştığında

1. Kullanıcı önce bilgileriyle giriş yapıyor,
2. İşlem yapacağı birimi seçiyor,
3. Kayıt yapacağı kategoriyi seçiyor,
4. Açılan formda gerekli bilgilerini yazıyor
5. Kayıt butonuna basıyor

Unit testlerin olmadığı bir projede yazılım geliştirici, yazılan kayıt fonksiyonu hemen arayüz katmanına bağlandığı taktirde runtime’da oluşacak hataların tespiti için bu işlemlerin hepsini tamamlayıp kayıt butonuna bastığında hatayla karşılaşacaktır. Hatayı düzeltip tekrar çalıştıracak, aynı işlemleri tekrar yapacak ve başka bir hatayla karşılaşabilecektir. Bu, yazılımcıya projenin ilerleyen aşamalarında büyük zaman kaybı yaşatmaktadır.

Unit testlerin en büyük avantajı da burada kendini göstermektedir. Bir üst satırda yazdığım örneği unit testlerle gerçekleştiren bir yazılımcı, bu işlemlerin hiçbirini yapmadan kendi oluşturduğu dummy datalarla bu hataları arayüz katmanına bağlama işlemini yapmadan alacak, testlerinin tümü başarıyla çalıştıktan sonra arayüz katmanına bağlama işlemini yapacak ve zaman kaybını önleyecektir.

Unit test oluştururken yapılması gereken, sadece kodun çalışacağı değerlerde değil çalışmayacağı değerlerin de parametrelerde gönderilip verilen hataların düzenlenmesidir. Örneğin bir fonksiyon içinde bir yerde bölme işlemi geçiyorsa parametreden bu bölme işleminde böleni 0 yapan bir değer gönderdiğimizde hata vereceğini biliyorsak bu hatanın sonucunda sistemin nasıl bir sonuç üreteceğini de görmemiz gerekmektedir. Bunun için de bir test yazmamız gerekecektir. Classlarımız tamamlandığında testlerinin de oluşturulup classlardaki tüm fonksiyonların testlerini başarıyla tamamlamalarını sağladıktan sonra bir sonraki class’ı kodlamaya başlamak gerekmektedir.

Unit Test’in avantajları ;

1. Yazılan kodun her satırının başka bir kod (test kodu) tarafından otomatik olarak test edilmesini sağlar.
2. Kodun anlaşılmasını kolaylaştırır.
3. Daha hızlı yazılım geliştirmeyi sağlar.(Proje başlangıcında projeyi yavaşlatır gibi görünmekte fakat projenin ileri dönemlerinde ciddi bir zaman kazanımı sağlamaktadır)
4. Koddaki hata oranını azaltır.
5. Kodların kalitesinin artmasını sağlar.
6. Hataların çabuk tespit edilip düzenlenmesini sağlar.

Örnek sınıf ve metot:

public class App {

public static void main(String[] args) {

System.out.println("Hello JUnit!");

}

public boolean isPositive(Integer number) {

return number > 0;

}

}

JUnit kütüphanesinin kullanımı annotations kullanımından ibarettir.

public class AppTest {

@Test

public void pozitif\_deger\_dogru() {

var number = 1453;

var app = new App();

boolean expResult = true;

boolean result = app.isPositive(number);

Assertions.assertEquals(expResult, result);

}

@Test

public void negatif\_deger\_yanlis() {

var number = -1453;

var app = new App();

boolean expResult = false;

boolean result = app.isPositive(number);

Assertions.assertEquals(expResult, result);

}

}

Testi çalıştırmak için aşağıdaki komutu komut yorumlayıcısına yazmak yeterli olacaktır.

Örnekte **@Test** ifadesi ile metodun test olduğu belirtilmiş **Assertions** sınıfında yer alan **assertEquals** metodu ile test işlemi yapılmıştır.

**NOT:** Netbeans, Eclipse ve Intellij gibi IDE özellikleri ile hızlıca oluşturma, yazma ve çalıştırma işlemi yapılabilir.

**Assertion method**

JUnit5 ile yapılan test işlemleri için Assertions sınıfında yer alan overload edilmiş statik **assert\*\*\*** metotları kullanılır.

Bazı assert metotları şunlardır;

* assertEquals
* assertNotEquals
* assertArrayEquals
* assertIterableEquals
* assertNull
* assertNotNull
* assertLinesMatch
* assertNotSame
* assertSame
* assertTimeout
* assertTimeoutPreemptively
* assertTrue
* assertFalse
* assertThrows
* \*fail

**NOT:** **fail** metodu test işleminin doğrudan başarısız olduğunu ifade etmek için kullanılır.

Örneğin; Sık kullanılan **assertEquals** metodu iki değerin eşit olup olmadığını test etmek için kullanılır.

**JUnit Test LifeCycle**

[Servlet](https://www.yusufsezer.com.tr/java-servlet/), [JSF](https://www.yusufsezer.com.tr/java-jsf/" \t "_blank), [JPA](https://www.yusufsezer.com.tr/java-jpa/" \t "_blank), [Spring](https://www.yusufsezer.com.tr/spring-framework/) vb birçok Java bileşeninde yer alan LifeCyle veya yaşam döngüsü JUnit’te de yer alır.

Test başlangıcında **@BeforeAll** ile belirtilen statik metotlar çalıştırılır.

Her bir test öncesi **@BeforeEach**, test sonrası **@AfterEach** ile belirtilen metotlar çalışır.

Test bitişinde **@AfterAll** ile belirtilen statik metotlar çalıştırılır.

public class AppTest {

@BeforeAll

public static void beforeAll() {

System.out.println("@BeforeAll");

}

@BeforeEach

public void beforeEach() {

System.out.println("@BeforeEach");

}

@Test

public void pozitif\_deger\_dogru() {

System.out.println("@Test");

var number = 1453;

var app = new App();

boolean expResult = true;

boolean result = app.isPositive(number);

Assertions.assertEquals(expResult, result);

}

@Test

public void negatif\_deger\_yanlis() {

System.out.println("@Test");

var number = -1453;

var app = new App();

boolean expResult = false;

boolean result = app.isPositive(number);

Assertions.assertEquals(expResult, result);

}

@AfterEach

public void afterEach() {

System.out.println("@AfterEach");

}

@AfterAll

public static void afterAll() {

System.out.println("@AfterAll");

}

}

Yaşam döngülerinin kullanım amacı test işlemi öncesi nesnelerin oluşturulması test işlemi sonrası kaynakların serbest bırakılmasıdır.

Önceki örnekte yer alan nesne oluşturma JUnit yaşam döngüsü ile aşağıdaki gibi yapılır.

public class AppTest {

App app;

@BeforeEach

public void beforeEach() {

app = new App();

}

@Test

public void pozitif\_deger\_dogru() {

System.out.println("@Test");

var number = 1453;

boolean expResult = true;

boolean result = app.isPositive(number);

Assertions.assertEquals(expResult, result);

}

@Test

public void negatif\_deger\_yanlis() {

System.out.println("@Test");

var number = -1453;

boolean expResult = false;

boolean result = app.isPositive(number);

Assertions.assertEquals(expResult, result);

}

}

Varsayılan olarak JUnit her bir metot öncesi yaşam döngüsünü **TestInstance.Lifecycle.PER\_METHOD** ile çalıştırır.

Yaşam döngüsünün çalışmasını değiştirmek için test sınıfına **@TestInstance(TestInstance.Lifecycle.PER\_CLASS)** eklemek yeterli olacaktır.

@TestInstance(TestInstance.Lifecycle.PER\_CLASS)

public class AppTest {

Integer myNumber = 0;

@Test

public void test1() {

myNumber += 1;

System.out.println(myNumber);

Assertions.assertEquals(1, myNumber);

}

@Test

public void test2() {

myNumber += 1;

System.out.println(myNumber);

Assertions.assertEquals(2, myNumber);

}

}

Test çalıştırıldığında her metot için yeniden yaşam döngüsü çalışmayacaktır.

**@DisplayName** – Test metodunun IDE ortamında görünen ismini değiştirir.

@Test

@DisplayName("Pozitif değer")

public void test1() {...}

NOT: Sonucun görünmesi için IDE ayarının yapılması gerekebilir.

**@Disabled** – Testi devre dışı bırakmak için kullanılır.

@Test

@Disabled

public void bir\_test() {

// ...

}

Koşula bağlı test

JUnit içerisindeki **@EnabledOnOs, @EnabledOnJre, @EnabledIf, @EnabledIfSystemProperty** ve **@EnabledIfEnvironmentVariable** sayesinde bir koşula bağlı metodun çalışması sağlanır.

@Test

@EnabledOnOs(OS.MAC)

public void bir\_test() {

// ...

}

Test Mac tabanlı işletim sisteminde çalışacaktır.

**Assumtations**

Bazı testler için bazı kaynakların hazır olması gerekebilir.

Assumtations ile kaynağın test edilerek çalıştığının belirlenmesi aksi halde testin çalışmaması sağlanabilir.

@Test

public void bir\_test() {

boolean siteye\_erisildi = false; // erişim kontrolü yapılıyor

Assumptions.assumeTrue(siteye\_erisildi);

// test işlemi

}

Test işlemini başlaması için **siteye\_erisildi** değerinin **true** olması gerekir.

**AssertAll** – Testleri gruplamak ve çalıştırmak için kullanılır.

@Test

public void bir\_test() {

Assertions.assertAll(

() -> Assertions.assertEquals(0, 0),

() -> Assertions.assertEquals(1, 2),

() -> Assertions.assertEquals(3, 3)

);

}

**@Nested** – Testleri bir alt sınıf içerisinde gruplamak için kullanılır.

public class AppTest {

@Nested

@DisplayName("Alt test")

class AltTest {

@Test

public void pozitif\_deger\_dogru() {

var number = 1453;

var app = new App();

boolean expResult = true;

boolean result = app.isPositive(number);

Assertions.assertEquals(expResult, result);

}

@Test

public void negatif\_deger\_yanlis() {

var number = -1453;

var app = new App();

boolean expResult = false;

boolean result = app.isPositive(number);

Assertions.assertEquals(expResult, result);

}

}

// Diğer testler

}

**@RepeatedTest** – Parametre kadar test çalıştırmak için kullanılır.

public class AppTest {

@RepeatedTest(5)

public void bir\_test() {

System.out.println("Yusuf SEZER");

}

}

Toplam test sayısı, hangi test adımında olduğu bilgisine **RepetitionInfo** ile erişilir.

public class AppTest {

@RepeatedTest(5)

public void bir\_test(RepetitionInfo repetitionInfo) {

int current = repetitionInfo.getCurrentRepetition();

int total = repetitionInfo.getTotalRepetitions();

System.out.printf("Toplam: %d Tekrar: %d %n", total, current);

}

}

**@Tag** – Testleri gruplamak, etiketlemek için kullanılır.

public class AppTest {

@Test

@Tag("dev")

public void dev\_test() {

System.out.println("dev");

}

@Test

@Tag("prod")

public void prod\_test() {

System.out.println("prod");

}

}

Maven ile sadece DEV etiketine sahip testleri çalıştırmak için aşağıdaki komut kullanılır.

mvn test -Dgroups=DEV

**TestInfo** – Test hakkında bilgi almak için kullanılır.

public class AppTest {

@Test

@Tag("dev")

@DisplayName("Örnek bir test")

public void bir\_test(TestInfo testInfo) {

System.out.println(testInfo.getDisplayName());

testInfo.getTags().forEach(System.out::println);

testInfo.getTestClass().ifPresent(System.out::println);

testInfo.getTestMethod().ifPresent(System.out::println);

}

}