Szoftvertesztelés Java platformon

Üdvözöllek a *Szoftvertesztelés Java platformon* képzésen!

Az anyag elméleti és gyakorlati videókból áll, valamint laborfeladatokból.

Érdemes a gyakorlatokat a videó alapján egyedül is megcsinálni, majd önállóan megoldani a laborfeladatokat.

> A videókban előfordulhatnak hibák, a javítások és a kiegészítések ilyen kiemelt formában szerepelnek a videó alatt.

A képzéshez tartozó slide-ok, gyakorlati feladatok és laborfeladatok megtalálhatóak a https://github.com/Training360/szt-01-public címen.

A laborfeladatok az `.md` fájlokban vannak.

Telepítendő szoftverek

A videókban a fejlesztőeszköz IntelliJ IDEA Community, mely ingyenesen letölthető. A projekt során Maven build eszköz került felhasználásra.

JUnit bevezetés

Ebben a fejezetben megismerkedhetsz a unit tesztelés fogalmával, valamint hogy miért is érdemes és hasznos unit teszteket írni. Megtanulhatod a Java nyelven legelterjedtebb tesztelési keretrendszer, a JUnit egyszerűbb funkcióit és használati lehetőségeit. Megnézzük, hogyan írhatsz egyszerű teszteseteket a programod helyes működésének ellenőrzésére. Kitérünk az elnevezési konvenciókra és arra is, hogyan futtathatod le a tesztjeidet nemcsak fejlesztőeszközből, hanem Mavennel is.

Bevezetés a JUnit használatába

A **unit teszteléssel** az adott programozási nyelv legkisebb egységét tudjuk tesztelni. Az objektumorientált programozási nyelvek esetén ez az osztály, hiszen ahhoz, hogy vizsgálni tudjuk egy metódus működését, először példányosítani kell egy objektumot az adott osztály alapján. A unit tesztelés célja annak leellenőrzése, hogy egy alapegység helyesen oldja-e meg a rábízott feladatot, az előírt bemenetre az elvárt kimenetet adja-e.

A unit tesztek tipikusan automatizáltak, azaz emberi beavatkozás nélkül futtathatóak. Szoftverfejlesztők írják őket, hiszen magas szintű programozási ismeretek kellenek ezeknek a teszteknek a megalkotásához.

Mit nem nevezünk unit tesztnek?

- Ha nem egy osztályt, hanem osztályok együttműködését teszteljük, azt **integrációs tesztnek** hívjuk.
- Ha a teszt futtatásakor bevonunk valamilyen konténert, az nem unit teszt.
- Ha a tesztesetek futtatása közben adatbázis műveleteket is végzünk, az szintén nem az.

A Java programozási nyelven a legelterjedtebb tesztelési keretrendszer a **JUnit**. Kent Beck és Erich Gamma szoftverfejlesztők tették le az alapjait. Mindenféle *build*-eszközhöz integrált, tehát *Ant*-ból, *Maven*-ből és *Gradle*-ből is lehet teszteket futtatni, valamint a fejlesztőeszközök is támogatják, ezekből is lehet unit teszteket elindítani.

A JUnit 4-es verziója 2006-ban, az 5-ös pedig 2017-ben jelent meg.

A JUnit Java osztályokból épül fel, **tesztosztályoknak** hívjuk őket. Ezekben metódusokként jelennek meg az egyes **tesztesetek**, amelyeket `@Test` annotációval kell ellátni. Az elvárt és az aktuális eredmény összehasonlítására a JUnit statikus `assertXXX()` nevű metódusokat tartalmaz.

A tesztesetek felépítésénél érdemes valamilyen struktúrát követni. Az egyik ilyen a *Given-When-Then* struktúra, amely a következőket jelenti:

- A *Given*-fázis tartalmazza az előfeltételeket, itt előkészítjük az adott tesztesetet, példányosítunk egy objektumot, beállítjuk annak a megfelelő állapotát.
- A *When*-fázisban meghívjuk a tesztelendő metódust.
- A *Then*-fázisban leellenőrizzük, hogy a kapott eredmény megfelel-e az általunk elvárt eredménynek.

Tegyük fel, hogy a következő osztályt szeretnénk tesztelni.

```
public class Employee {

  private String name;

  private int yearOfBirth;

  // Konstruktorok

  public int getAge(int atYear) {
    return atYear - yearOfBirth;
  }

  // Getter és setter metódusok

}
```

Ehhez készítünk egy osztályt, mely tartalmazza a tesztesetet:

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
```

```
public class EmployeeTest {

  @Test
  void testGetAge() {
    // Given
    Employee employee = new Employee("John Doe", 1970);

    // When
    int age = employee.getAge(2019);

    // Then
    assertEquals(49, age);
  }
}
```

Ezt természetesen nem kell ennyire tagoltan megírni, hiszen könnyen olvasható mindez egyetlen utasításban összefoglalva is, de funkcionalitásban tényleg ebből a három részből áll össze a teszteset.

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.DynamicTest.dynamicTest;

public class EmployeeTest {

  @Test
  void testGetAge() {
    assertEquals(49,
      new Employee("John Doe", 1970).getAge(2019));
  }
}
```

A JUnit a legtöbb *build*-eszközhöz integrált, így Mavenhez is. Ahhoz, hogy használni tudjuk, először fel kell venni függőségként, `test` scope-pal. Így a JUnit könyvtár nem lesz az alkalmazás része, hanem csak a tesztesetek futtatásához szükséges. Maven projektben konvenció szerint az `src/test/java` könyvtárban helyezzük el a teszteseteket, a hozzájuk tartozó erőforrás állományokat pedig az `src/test/resources` könyvtárban. Így a Maven automatikusan megtalálja és lefordítja az osztályokat, futás közben a megfelelő könyvtárakat elhelyezi a classpath-on, valamint magát a futtatást is elvégzi. Ehhez a `pom.xml`-be a következőket kell felvenni függőségként:

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>${junit5.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.0</version>
    </plugin>
  </plugins>
</build>
```

A különböző fejlesztőeszközök is tartalmaznak integrációt a JUnithoz. Le lehet futtatni bennük egyszerre csak egy tesztesetet (metódust) vagy egy tesztosztályt a benne lévő összes metódussal, de lehet egyszerre futtatni egy csomag összes tesztosztályát, vagy akár a projekt összes tesztosztályát is. Van lehetőség arra, hogy csak az elbukott teszteseteket futtassuk újra, valamint lehet debugolni is a teszteseteket, vagyis lépésenként végrehajtani a bennük található utasításokat. A fejlesztőeszközök különböző színű jelölésekkel is segítik, hogy könnyen meg tudjuk különböztetni egymástól a sikeresen lefutott és az elbukott teszteseteket.

Az első teszteset létrehozása

Maven projektben egy tesztosztály létrehozásakor a következőkre kell/érdemes figyelni:

- Először fel kell venni a `pom.xml`-be függőségként a JUnit keretrendszer Jupiter modulját `test` scope-pal.
- A tesztosztály neve utaljon a tesztelendő osztály nevére, valamint az `src/test/java` könyvtáron belül ugyanolyan nevű csomagban helyezkedjen el, mint a tesztelendő osztály. A tesztosztály ugyanis csak így fogja "látni" a tesztelendő osztályt.
- A tesztmetódus (teszteset) neve utaljon a tesztelendő metódus nevére, és ne legyen visszatérési értéke (`void`).
- A tesztmetódust el kell látni a `@Test` annotációval, mert csak így fog lefutni.
- A tesztmetódusnak és a tesztben használt attribútumoknak nem kell `private` vagy `public` módosítószót adni, hiszen a tesztet kívülről senki nem fogja hívni.
- A teszteset automatikussá tételére a JUnit `Assertions` osztályában lévő `assertXXX()` nevű statikus metódusok szolgálnak. Ezeket ahhoz, hogy használni tudjuk őket, statikus importtal importálni kell.

Unit tesztelés ígéretei

A unit tesztelés hozadékai:

- Mivel ezeket a fejlesztő írja akkor, amikor az adott osztályt is, ezért a hibák nagyon gyorsan ki tudnak derülni, azonnal lehet őket javítani, nem megy el fölöslegesen sok idő és energia az utólagos hibakereséssel.

- A unit tesztelés támogatja a *refactoring*-ot, amikor az alkalmazásunkat csak strukturálisan módosítjuk valamilyen ok miatt (pl. később újabb kódrészletet szeretnénk beilleszteni), funkcionálisan viszont nem. A unit teszteknek ugyanis a refactorálás után is sikeresen le kell futniuk, tehát így ellenőrizhetjük, hogy a kód struktúrájának megváltoztatása közben a funkcionalitás tényleg nem módosult.
- Regresszió biztosítása, azaz ha valami már egyszer működött, akkor a későbbi módosítások során is működőképes marad majd.
- Segítségül lehet hívni a unit tesztelést hibajavítás közben is. Először unit teszttel reprodukáljuk a hibát, majd csak utána kezdünk hozzá a javításhoz. Így biztosíthatjuk, hogy ez a hiba többet nem fog jelentkezni.
- Ha egy komplex rendszerünk van sok unit teszttel, és egy későbbi módosítás során valamit elrontunk, akkor gyorsabban lehet rátalálni a hibás komponensre és javítani azt.
- A unit tesztelés támogatja azt, hogy az alapvető objektumorientált paradigmákat betartsuk. *High cohesion*: egy adott osztály mindig egy jól meghatározott feladattal rendelkezik, és ezt a feladatot hiba nélkül meg is valósítja. *Low coupling*: az osztályok közötti kapcsolat laza, egy osztály nem függ túlságosan egy másiktól. A unit teszteket úgy érdemes megírni, mintha mi független felek lennénk és ezt az osztályt használni szeretnénk. *Test Driven Development*: úgy fejlesztünk, hogy előbb írjuk meg a teszteseteket és csak utána magát az osztályt. Ez is azt támogatja, hogy először kívülről nézzünk rá a saját, fejlesztendő osztályunkra, és azt olyanra tudjuk megírni, hogy akár mások által is könnyen használható legyen.
- A unit tesztelés nagy előnye, hogy egyszerűsíti a fejlesztést. A hibák ezáltal hamarabb kiderülnek, vagyis a hibajavítás költsége jóval kevesebb. Egy bonyolult alkalmazás változtatása közben a tesztesetekkel lehet automatikusan ellenőrizni, hogy azok a funkciók, melyeket nem módosítunk, nem romlottak-e el a módosítás hatására.
- A unit teszt önmaga dokumentálja is a kódot, hiszen kiderül belőle az osztály működése, sőt, "élő dokumentációként" működik. Ugyanis amennyiben módosul az osztály, akkor nem kell külön észben tartani, hogy még a dokumentációt is módosítani kell. Erre ugyanis eleve rákényszerül a fejlesztő, hiszen a módosítás folytán a teszt eleve le sem fog fordulni vagy le sem fog futni.

Futtatás Mavennel

A Maven beépítetten támogatja a JUnitot, méghozzá a Surefire plugin segítségével. A Maven egy életciklust definiál az adott projektekhez, amely különböző fázisokból áll.

A *default életciklus* alatt azt értjük, hogy hogyan build-eljünk le egy projektet. Ez olyan lépésekből (fázisokból) áll, mint az erőforrások előkészítése, a Java forráskód lefordítása, az adott alkalmazás összecsomagolása, stb. Ennek az életciklusnak a szerves része a tesztesetek lefuttatása is. Ha csak az életciklus egyik fázisát akarjuk lefuttatni a Mavennel, akkor meg kell adnunk paraméterként ennek a fázisnak a nevét, és a Maven ekkor az eddig a fázisig tartó összes korábbi fázist lefuttatja. Pl. `mvn test` parancs kiadásakor lefut az összes fázis, amely a tesztesetek futtatása előtt le kell, hogy fusson (erőforrások

előkészítése, fordítás, tesztekhez szükséges erőforrások előkészítése), és végül a Maven futtatja a teszteseteket is.

A `package` fázis lefuttatása esetén automatikusan le fognak futni a tesztesetek is. Ha ezt nem szeretnénk, akkor lehetőség van ezt átugorni a `-DskipTests` kapcsolóval (`mvn package -DskipTests` parancs). Ez azonban nem javasolt, hiszen így nem biztosítható, hogy hibátlan lesz az összecsomagolt alkalmazásunk. Ha azért nem akarnánk lefuttatni őket, mert az túl sok időt venne igénybe, akkor valószínűleg maguk a tesztesetek hibásak, esetleg nem is unit tesztekről, hanem integrációs tesztekről van szó. Abban az esetben, ha a tesztek jól vannak megírva, akkor gyorsan kell lefutniuk.

Lehetőség van csak egy teszteset lefuttatására is a következő módon kiadott paranccsal: `mvn package -Dtest=EmployeeTest#testGetAge` Itt a `testGetAge` a konkrét teszteset neve, melyet futtatni kívánunk, a `EmployeeTest` pedig az osztály, amelyben ez a tesztmetódus található. Lehetőség van természetesen arra is, hogy ne csak parancssorból futtassuk a Mavent, hanem az IDEA fejlesztőeszközből.

Tesztesetek életciklusa

Teszteseteknél szükség lehet az adott objektum előkészítésére, például példányosítására, állapotának beállítására vagy módosítására. Az adott tesztesetre való előkészületet **test fixture**-nek nevezik, amikor az adott környezetet megfelelő állapotba hozzuk.

Ez kiemelhető külön metódusba is, ami több szempontból is hasznos. Egyrészt így elkülönül magától a tesztesettől, másrészt pedig ilyenkor újra fel lehet használni ugyanazt a test fixture-t akár több tesztesetnél is. Erre vannak a JUnitban különböző annotációk.

A `@BeforeEach` annotációval ellátott metódust a JUnit minden egyes teszteset előtt lefuttatja, az `@AfterEach` annotációval ellátott metódust pedig minden egyes teszteset után.

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class EmployeeTest {

    Employee employee;

    @BeforeEach
    void initEmployee() {
        employee = new Employee("John Doe", 1970);
    }

  @Test
  void testGetAge() {
      assertEquals(49, employee.getAge(2019));
  }

    @Test
```

```
    void testWithZeroAge() {
        assertEquals(0, employee.getAge(1970));
    }
}
```

Vannak olyan metódusok is, melyeket ritkábban kell lefuttatni. A `@BeforeAll` annotációval ellátott metódus csak egyszer fut le, az adott osztályban szereplő összes teszteset lefutása előtt. Az `@AfterAll` annotációval ellátott metódus szintén csak egyszer fut le, az adott osztály összes tesztmetódusa után. Az ilyen annotációval ellátott metódusoknak statikusoknak kell lenniük.

Van lehetőség arra is, hogy ezeket a fixture-öket egy külön interfészbe helyezzük el, és az interfészben default interfész metódusként implementáljuk ezeket a teszteseteket előkészítő metódusokat. Ezután a tesztosztály implementálni tudja ezt az interfészt, és a benne lévő metódusokat a megfelelő időpontban le tudja futtatni.

```
public interface PrintNameCapable {

  @BeforeEach
  default void printName(TestInfo testInfo) {
    System.out.println("Test case display name: "
      + testInfo.getDisplayName());
  }
}

public class EmployeeTest
  implements PrintNameCapable {

  // ...

}
```

A tesztesetek írásakor törekedni kell az izoláltságra. Ez azt jelenti, hogy a tesztesetek legyenek egymástól függetlenek, ne legyen egymásra semmiféle hatásuk. Ez azért fontos, mert így a tesztesetben kapott eredmény függene az egyes tesztesetek lefuttatási sorrendjétől, illetve nagyon megnehezítené a hibakeresést is, hiszen nem lenne egyértelmű, hogy az adott hiba éppen melyik teszteset futása közben történt. Ezért figyelni kell arra, hogy az egyes tesztesetek között semmiféle állapotátmenet ne lehessen. (Például vigyázni kell statikus attribútumok értékeivel, aminek az értéke egy tesztosztály futása közben nem áll vissza metódusonként a kezdeti értékére.) A JUnit a tesztesetek egymástól való függetlenségét úgy támogatja, hogy minden egyes teszteset futtatása előtt újra példányosítja a tesztosztályt.

```
@BeforeEach
void initEmployee() {
    employee = new Employee("John Doe", 1970); // INIT
}

@Test
```

```
void testGetAge() {
    assertEquals(49, employee.getAge(2019)); // TC1
}

@Test
void testWithZeroAge() {
    assertEquals(0, employee.getAge(1970)); // TC2
}
```

A fenti példakód esetén tehát először megtörténik a példányosítás, majd lefut a `initEmployee()` és `testGetAge()` metódus, majd példányosítás, és lefut a `initEmployee()` és `testWithZeroAge()` metódus.

A JUnitnál soha nem szabad a tesztesetek lefutási sorrendjére építeni. Abban az esetben nem is lehet erre szükségünk, ha betartjuk azt, hogy az egyes tesztesetek egymástól függetlenek legyenek. Esetleg integrációs teszteknél elképzelhető az, hogy fontos lehet valamiféle lefutási sorrend, ezt a `@TestMethodOrder` annotációval tudjuk megadni, és egy `MethodOrderer` implementációval paraméterezni. Vannak előre elkészített implementációk:

- `MethodOrderer.Alphanumeric`: itt a tesztesetek ábécé-sorrendben kerülnek lefuttatásra.
- `MethodOrderer.OrderAnnotation`: ekkor tehetünk a tesztesetekre `@Order` annotációt, amelyet egy egész számmal paraméterezhetünk, és sorrendben, a számok alapján fogja a JUnit lefuttatni a tesztmetódusokat.
- `MethodOrderer.Random`: véletlenszerű lefuttatás. A JUnit egyébként alapértelmezetten így futtatja a teszteseteket. Ezzel nagyon jól lehet azt is tesztelni, hogy a teszteseteink valóban függetlenek-e egymástól.

Lehetőség van arra is, hogy bizonyos teszteseteket ne engedjünk a futtatókörnyezetnek lefuttatni. Ehhez rá kell tennünk a `@Disabled` annotációt vagy az adott tesztmetódusra, vagy akár az egész osztályra. Ez utóbbi esetben az adott osztályban szereplő egyetlenegy metódus sem fog lefutni. Opcionálisan megadható üzenet is, hogy miért kapcsoltuk ki ideiglenesen ezt a tesztesetet. Nagyon nem ajánlott ennek a használata, érdemes kizárólag ideiglenesen használni ezt az annotációt (például csak addig, amíg egy hibát gyorsan ki nem javítunk). Ha viszont ideiglenesen ki akarjuk kapcsolni az adott tesztesetet, még mindig jobb, ha ezzel az annotációval látjuk el, mintha úgy kapcsoljuk ki, hogy megjegyzésbe tesszük.

Lehetséges a teszteset kikapcsolása valamilyen feltétel alapján (*conditional test execution*). Ilyen feltétel lehet például az operációs rendszer típusa és verziószáma, a JVM verziószáma, illetve függhet a lefutás egy operációs rendszerben beállított környezeti változótól vagy akár egy Java környezeti változótól is. Ehhez a `@DisabledXXX` annotációkat kell használni, és paraméterül kell megadni azt, hogy milyen esetekben legyenek az adott tesztesetek kikapcsolva. Például: `@DisabledOnOs(OS.WINDOWS)`

Elnevezések

Tesztesetek futtatásakor a felületen és különböző reportokban megjelennek a tesztesetek nevei, a konkrét metódus- vagy osztálynevek. Lehetőség van ezek személyre szabására úgy, hogy az adott tesztesetre vagy tesztosztályra a `@DisplayName` annotációt tesszük, és ennek paraméterként átadjuk azt a nevet, amelyet szeretnénk, hogy megjelenjen a fejlesztőeszközben vagy a reportban.

```
@Test
@DisplayName("Test the calculation of the age with positive number")
void testGetAge() {
    // ...
}
```

Lehet az adott metódus- vagy osztálynévből generálni is a megjelenítendő nevet, például egyes karakterek automatikusan kicserélhetők más karakterekre, vagy az első szó nagybetűssé tehető, a többi pedig kisbetűssé. Erre használhatjuk a `@DisplayNameGeneration` annotációt, amelynek meg kell adni, hogy hogyan történjen maga a konvertálás. Erre vannak beépített implementációk, de akár magunk is írhatunk egyet.

```
@Test
@DisplayNameGeneration
  (DisplayNameGenerator.ReplaceUnderscores.class)
void get_age_with_positive_number() {
    // ...
}
```

Assert

Az **assert**-tel tudjuk megvizsgálni, hogy a megkapott, aktuális eredmény egyezik-e az elvárt eredménnyel. Erre többféle, `assertXXX()` nevű metódust tartalmaz a JUnit, amelyek mind statikus metódusok. Statikusan kell importálnunk őket, és ekkor elegendő csak a metódus nevét használni a tesztesetben. Ezek a következők:

- `assertNull()` és `assertNotNull()`: a paraméterként átadott referenciát vizsgálja, hogy az `null`-e.
- `assertEquals()` és `assertNotEquals()`: használható primitív és referencia típusú paraméterekkel is. Két paramétert kell neki átadni, és azok egyezőségét vizsgálja. Primitív típusú paraméterek esetén megnézi, hogy azok konkrét értéke egyenlő-e (`==`). Referencia típusú paraméterek esetén meghívja azok `equals()` metódusát és ez alapján nézi az egyezőségüket. Lebegőpontos számok esetén a bináris számábrázolásból adódó pontatlanságok figyelembevételére megadható egy harmadik paraméter is, ahol megadhatjuk a hibahatárt, amin belül egyenlőnek kell tekinteni az elvárt és az aktuális értéket.

Példa a hibahatár megadására:

```
assertEquals(1.0, 1.0, 0.005);
```

- `assertSame()` és `assertNotSame()`: referenciákat hasonlít össze kifejezetten az `==` operátorral.
- `assertTrue()` és `assertFalse()`: ennek paraméterül egy `boolean` értékű kifejezést adhatunk át, és a teszteset akkor fut le sikeresen, ha ennek a kifejezésnek az értéke a metódus nevében szereplő értékkel egyezik meg.
- `fail()`: lehetséges az is, hogy explicit módon elbuktassuk a tesztesetet.
- `assertArrayEquals()`: két tömböt hasonlíthatunk össze.
- `assertIterableEquals()`: az `Iterable` interfész implementáló osztályok objektumainak összehasonlítására használható, vagyis tipikusan kollekcióknál használjuk. Végignézi, hogy ugyanazok az elemek szerepelnek-e a paraméterként átadott két kollekcióban.
- `assertLinesMatch()`: string listák összehasonlítására, de nem csak a pontos egyezőséget vizsgálja, hanem az elvárt értéknél megadhatunk például egy reguláris kifejezést is, és azt vizsgálja, hogy az aktuális érték ennek megfelel-e.
- `assertAll()`: egyszerre több `assertXXX()` kifejezést adhatunk neki át paraméterül (lambda kifejezésekben), és mindet kiértékeli attól függetlenül, hogy sikeres vagy sikertelen-e. Mindegyiknek az eredményét külön-külön megjeleníti.

Példa az `assertAll()` használatára:

```
assertAll(
                () -> assertEquals("John Doe", employee.getName()),
                () -> assertEquals(49, employee.getAge(2019))
        );
```

Akár tetszőlegesen egymásba is ágyazhatóak:

```
assertAll("employee",
  () -> {
      assertNotNull(employee.getName());
      assertAll("name",
              () -> assertTrue(employee.getName().startsWith("John")),
              () -> assertTrue(employee.getName().endsWith("Doe"))
      );
  },
  () -> assertAll("age",
          () -> assertTrue(age > 40),
          () -> assertTrue(age > 50))
);
```

A JUnit nem biztosít túl magas szintű lehetőséget arra, hogy bonyolultabb kollekciókat vizsgáljunk meg, hiszen referencia típusú elemeket tartalmazó kollekcióknál összesen annyit csinál, hogy a kollekció elemein sorban meghívja az `equals()` metódust. Ha az nincs (számunkra) megfelelő módon implementálva az adott osztályban, akkor a teszteset nem fog sikeresen lefutni. Az egyik lehetőség az, hogy egy objektum listát valamilyen módon String-listává konvertálunk és azt már össze tudjuk hasonlítani.

A következő példa mutatja meg, hogy készítünk az `List<Employee>` listából olyan `List<String>` listát, mely a neveket tartalmazza:

```
@Test
void testListEmployees() {
    List<Employee> employees = employeeRepository.listEmployees();

    assertEquals(2, employees.size());
    assertEquals("Jack Doe", employees.get(0).getName());

    assertEquals(List.of("Jack Doe", "John Doe"),
      employees.stream().map(Employee::getName)
        .collect(Collectors.toList()));
}
```

Vannak azonban olyan más eszközök, amelyek a kollekciók vizsgálatát jobban kiterjesztik. Ilyen például a **Hamcrest** és az **AssertJ**, amelyek általános célúak. Speciális összehasonlító eszköz például az **XMLUnit** (XML-ek összehasonlítására, strukturális egyezőség alapján), a **JSONassert** (JSON formátumú elemek egyezőségének vizsgálatára), **DbUnit** (adatbázis tartalmak összehasonlító vizsgálatára).

Lehetőség van arra, hogy ne a JUnit beépített hibaüzenetét írjuk ki abban az esetben, ha egy adott assert elbukik, hanem magunk adjuk meg azt az üzenetet, amelyet szeretnénk, hogy kiírjon. Ezt az `assertEquals()` metódusnak harmadik paraméterként adhatjuk meg.

```
assertEquals(0, employee.getAge(1970), "Age must be zero");
```

Ha itt egy string értéket visszaadó metódust hívunk meg, a paraméter kiértékelésre, a metódus meghívásra kerül. Ha viszont lambda-kifejezésként adjuk át, az csak akkor értékelődik ki, ha a teszteset elbukik (így gyorsítható a tesztek lefutása sikeresen lefutó tesztek esetén).

```
assertEquals(1970,
  getAgeFromXmlDocument(document), () -> documentToString(document));
```

Az *assert* mellett van egy másik fogalom is, az *assume*. Ez az előfeltételek ellenőrzésére használatos, és arra való, hogy ha ezek az assume-ok elbuknak, akkor a teszteset végrehajtása leáll, viszont nem fogja a JUnit hibásnak jelenteni az adott tesztesetet. Akkor használjuk, amikor egyes teszteseteket csak akkor érdemes lefuttatni, ha bizonyos feltételek fennállnak. Az assume hasonló metódusokkal rendelkezik, mint az assert metódusok, ezek az `Assumptions` osztályban találhatóak.

Haladó JUnit

Ebben a fejezetben mélyebben is tárgyaljuk a JUnit használatában rejlő lehetőségeket. Megnézzük, milyen módon lehet tesztet írni arra az esetre, amikor a programtól egy kivétel dobását várjuk el és azt is, milyen módokon lehet csoportosítani a teszteseteinket. Szó lesz a paraméterezett és dinamikus tesztekről, amelyekkel lehetőség nyílik ugyanazon tesztesetet

többször egymás után, különböző tesztelendő értékekkel lefuttatni. Megtanulhatod, hogyan lehet egyszerűen fájlműveleteket tesztelni. Kitérünk arra is, mik a legjobb gyakorlatok, amelyeket érdemes követned unit tesztek írása során, hogy a legegyszerűbben és a leghatékonyabban tudd használni a JUnitot a napi munkád során.

Kivételkezelés és timeout tesztelése

Amennyiben azt szeretnénk vizsgálni, hogy egy metódus a megfelelő kivételt dobja-e, akkor az `assertThrows()` metódust kell hívnunk. Ennek első paramétere maga a kivétel osztálya, amelynek az eldobását várjuk az adott metódustól, második paraméterként pedig egy lambda kifejezést kell átadnunk, amelyen belül meghívjuk ezt a metódust.

```
IllegalArgumentException iae = assertThrows(
  IllegalArgumentException.class,
  () -> new Employee("John Doe", 1800));
assertEquals("Invalid year: 1800", iae.getMessage());
```

Lehetőség van JUnitban a *timeout* vizsgálatára is, hogy egy adott időn belül egy adott metódus lefut-e. Erre az `assertTimeout()` metódus használható, amelynek első paramétere egy `Duration` példány, második paramétere pedig szintén egy lambda-kifejezés, amely meghívja azt a metódust, melynek lefutási idejét vizsgálni szeretnénk. Itt vigyázni kell egyrészt arra, hogy unit teszteknél ne futtassunk ilyeneket, mert ez meghosszabbítja a tesztek lefutási idejét. Másrészt arra is figyelni kell, hogy különböző számítógépek esetén ugyanannak a metódusnak a lefuttatási ideje nagyon különböző is lehet. Az `assertTimeout()` metódus úgy működik, hogy ugyanazon a szálon futtatja a vizsgálandó metódust, végigvárja annak lefutását (akkor sem szakítja meg azt, ha az már időközben túllépte az elvártként megadott időt), és csak ezután fogja kiértékelni, hogy belefért-e a metódus lefutása az adott időtartamba vagy sem.

```
assertTimeout(ofMinutes(2), () -> employeeService.calculateYearlyReport());
```

Visszatérési értékkel:

```
String result = assertTimeout(ofMinutes(2),
  () -> employeeService.calculateYearlyReport());
```

Az `assertTimeoutPreemptively()` metódus ugyanezt úgy valósítja meg, hogy külön szálon futtatja a metódust, és timeout esetén megszakítja annak futását.

```
String result =
  assertTimeoutPreemptively(ofMinutes(2), () ->
employeeService.calculateYearlyReport());
```

Egymásba ágyazás

A tesztesetek között hierarchiát is építhetünk föl. Ezt úgy tudjuk megoldani, hogy belső osztályokat használunk. Ez akkor hasznos, ha valamiféleképpen csoportosítani szeretnénk a

teszteseteinket, és a különböző csoportok inicializációjakor van egy közös rész is, illetve csoportonként pedig egy különböző rész is. A belső osztályokat a `@Nested` annotációval kell ellátni.

Létrehozhatok közös inicializációt a külső osztályban is és ezzel egyidőben a belső osztályok mindegyikében is.

```
public class EmployeeTest {

    Employee employee;

    @Nested
    class WithYearOfBirth1970 {

        @BeforeEach
        void init() {
            employee = new Employee("John Doe", 1970);
        }

        @Test
        void testAge() {
            // ...
        }

    }

    @Nested
    class WithYearOfBirth2000 {

        @BeforeEach
        void init() {
            employee = new Employee("John Doe", 2000);
        }

        @Test
        void testAge() {
            // ...
        }

    }
}
```

Tagek és metaannotációk használata

A JUnit unit tesztjeit el lehet látni *tag*-ekkel is, ezzel csoportosítva a teszteseteket. Ehhez a `@Tag` annotációt kell használnunk, amelyet rátehetünk az egész tesztosztályra vagy egyes tesztesetekre is, egy helyen egyszerre akár többet is.

Később, amikor futtatjuk a teszteseteket, akkor szűrni lehet, hogy milyen tagekkel ellátott tesztesetek fussanak csak le. Ezt Mavenben a Surefire pluginon belül úgy tehetjük meg, hogy megadjuk a `<configuration>` tagen belüli `<groups>` tagen belül, hogy pontosan milyen taggel ellátott tesztesetek lefuttatását szeretnénk.

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.0</version>
    <configuration>
        <groups>unit</groups>
    </configuration>
</plugin>
```

Itt meg lehet adni egyszerre csak egy szót (taget) is, de akár egy több szót tartalmazó, bonyolultabb kifejezést is. Használhatóak a `|` (vagy), az `&` (és), illetve a `!` (negáció) operátorok, valamint zárójelek is. Például az itt következő példa jelentése: fussanak le azok a tesztesetek, amelyeken rajta van a `unit` és a `feature-329` tag, de nincs rajta a `long-running` tag:

```
unit & feature-329 & !long-running
```

Lehetőség van JUnitban úgynevezett **metaannotációk** implementálására is, ami azt jelenti, hogy mi magunk hozhatunk létre annotációkat, ezekre az annotációkra további annotációkat tehetünk, és az általunk létrehozott annotációt használhatjuk, mintegy rövidítésként, a tesztmetódusokon. Egy ilyen metaannotáció létrehozására látható példa az alábbiakban.

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Test
@Tag("unit")
public @interface UnitTest {
}
```

Itt meg vannak adva a következőek:

- az új annotáció neve (`public @interface UnitTest`)
- mire tehető rá a létrehozott új annotáció (`@Target(ElementType.TYPE, ElementType.METHOD)`, vagyis osztályra és metódusra),
- mikor kerüljön feldolgozásra ez az új annotáció (`@Retention(RetentionPolicy.RUNTIME)`, vagyis futásidőben),
- az, hogy tesztesetnél akarjuk használni ezt az annotációt (`@Test`)
- és kerül rá egy tag is (`@Tag("unit")`).

És ezután a tesztesetekre vagy tesztosztályokra elég csak azt az annotációt rátenni, amelyet saját magunk hoztunk létre, vagyis azt, hogy `@UnitTest`.

```
@UnitTest
void testGetAge() {
}
```

Tesztesetek ismétlése

A JUnit keretrendszernek megmondhatjuk azt is, hogy egy tesztesetet többször futtasson le. Erre akkor lehet szükség, ha ugyanazt a tesztesetet különböző bemeneti paraméterekkel szeretnénk lefuttatni (*data driven* tesztelés). Ehhez a `@RepeatedTest` annotációt kell használnunk, és első paraméterként egy egész számot kell megadnunk, amely azt jelenti, hogy az adott teszteset hányszor fusson le. Minden lefuttatás nevet kap, és ebben szerepel az, hogy ez hányadik lefuttatás és az is, hogy összesen hány lefutása van ennek a tesztesetnek.

```
@RepeatedTest(5)
void testGetAge() {
  Employee employee = new Employee("John Doe", 2000);
  assertEquals(19, employee.getAge(2019));
}
```

Lehetőség van arra, is, hogy ezt a nevet, amit minden lefutás kap és megjelenik a felhasználói felületen vagy a reportban, személyre szabjuk. Ekkor az annotáció `name` paraméterét kell felülbírálni. Megadhatunk benne állandó szövegeket vagy placeholdereket is. Ilyen utóbbi a `{currentRepetition}`, amely azt mondja meg, hogy hányadik lefutás a mostani, vagy a `{totalRepetitions}`, amely pedig azt, hogy hány lefutás lesz összesen.

```
@RepeatedTest(value = 5,
  name = "Get age {currentRepetition}/{totalRepetitions}")
void testGetAge() {
  Employee employee = new Employee("John Doe", 2000);
  assertEquals(19, employee.getAge(2019));
}
```

Lehetőség van arra is, hogy kódból hozzáférjünk ahhoz, hogy éppen hányadik lefutásnál vagyunk, ekkor az adott tesztesetnek deklarálnunk kell egy `RepetitionInfo` paramétert, és amikor a JUnit meghívja ezt a tesztesetet, akkor a tesztesetnek átad egy `RepetitionInfo` objektumot, amelyből kiolvasható, hogy hányadik lefutásnál tart éppen és hány lesz összesen.

```
private int[][] values = {{2000, 0}, {2010, 10}, {2015, 15}, {2050, 50},
{1990, -10}};

@RepeatedTest(value = 5, name = "Get age
{currentRepetition}/{totalRepetitions}")
void testGetAge(RepetitionInfo repetitionInfo) {
  Employee employee = new Employee("John Doe", 2000);
  assertEquals(values[repetitionInfo.getCurrentRepetition() - 1][1],
    employee.getAge(values[repetitionInfo.getCurrentRepetition() - 1][0]));
}
```

Paraméterezett tesztek

Abban az esetben, ha ugyanazt a tesztesetet szeretnénk lefuttatni különböző adatokkal, **paraméterezett tesztet** használhatunk. Ekkor gyakorlatilag megírjuk egyszer a tesztmetódust, és ezt különböző paraméterekkel futtatjuk le. Ezeket a paramétereket megadhatjuk közvetlenül a forráskódban is, de akár külső állományból is beolvashatjuk, például CSV-állományból (ez egy szöveges állomány, amely vesszővel elválasztva tartalmazza az értékeket). Ez egy kísérleti stádiumban lévő funkció a JUnitban, és ahhoz, hogy ezt használhassuk, egy további függőséget kell felvennünk a `pom.xml`-ben.

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.5.1</version>
    <scope>test</scope>
</dependency>
```

Ha egyszerű értékekkel paraméterezve szeretnénk meghívni az adott metódust, azt úgy tehetjük meg, hogy rátesszük a `@ParameterizedTest` annotációt, valamint egy `@ValueSource` annotációt, amelynek paraméterként megadjuk a konkrét értékeket. A teszteset maga pedig vár egy paramétert. A JUnit ezután a `@ValueSource`-on belül értékek mindegyikével egyenként meghívja a tesztmetódust.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(isPalindrome(candidate));
}
```

Nem csak szövegeket lehet megadni paraméterként, hanem például primitív típusokat is. A `@ValueSource` annotációnak tehát többféle paraméterezése létezik: `strings`, `ints`, `doubles`, stb., és a konkrét értékeket tömbliterálként kell megadni, tehát kapcsos zárójelek között. Nem csak a beviteli értékeket lehet megadni, hanem az elvárt értékeket is, második paraméterként.

Lehetőség van még további speciális annotációk használatára is. A `@NullSource` azt jelenti, hogy hívja meg ezt a metódust `null` értékkel is, az `@EmptySource`-ot akkor használhatjuk, ha String vagy kollekció a paraméter típusa, ekkor üres stringgel vagy kollekcióval fogja meghívni a metódust. A `@NullAndEmptySource` mindkét módon, tehát `null` értékű és üres stringgel vagy kollekcióval is meghívja az adott metódust.

Itt is minden egyes lefutás nevet kap, és ebben a névben szerepel a paraméter értéke is.

`@EnumSource` annotáció használatával megadhatjuk egy enum különböző értékeit is paraméterként. Sőt, szűrni is lehet a különböző enum értékekre. Megadhatjuk explicit módon, hogy melyik különböző enum értékekkel hívja meg az adott metódust, vagy pedig akár reguláris kifejezéseket is használhatunk a szűrésre.

```
@EnumSource(value = TimeUnit.class, names = { "DAYS", "HOURS" })
```

Arra is lehetőségünk van, hogy a paramétereket ne explicit módon soroljuk föl egy tömbben, hanem ezeket egy metódus állítsa elő. Ekkor a `@MethodSource` annotációt használjuk, amelynek paraméterül meg kell adni a metódus nevét, amely metódus visszaadhat egy `Stream`, egy `Collection`, egy `Iterator` vagy egy `Iterable` példányt is. Ekkor a JUnit végig fog menni a metódus által visszaadott értékeken, és ezekkel az értékekkel paraméterezve fogja meghívni az adott tesztesetet. A `@MethodSource`-nak egy statikus metódus nevét adhatjuk meg paraméterül.

`@ArgumentsSource` annotációval egy `ArgumentsProvider` implementációt adhatunk át paraméterül. Ennek az interfésznek egyetlen metódusa van, amellyel egy `Stream` formájában adhatjuk meg a paramétereket.

A `@ParameterizedTest` annotációban `name` paraméterként megadhatjuk a teszteset nevét is, hogy hogyan kerüljön megjelenítésre a felhasználói felületen vagy a reportban.

```
@ParameterizedTest(name = "year {0} - age {1}")
@MethodSource("createAges")
void testGetAge(int year, int age) {
    Employee employee = new Employee("John Doe", 2000);
    assertEquals(age, employee.getAge(year));
}

static Stream<Arguments> createAges() {
    return Stream.of(
            arguments(2000, 0),
            arguments(2010, 10),
            arguments(2050, 50),
            arguments(1990, -10)
    );
}
```

CSV-állományok megadására több lehetőség is van. A `@CsvSource` annotációval van lehetőség az állomány tartalmát `String[]` formájában közvetlenül a forráskódban megadni. A `@CsvFileSource` annotációnak pedig egy, a classpath-on lévő CSV-fájlt adhatunk meg. Ezenkívül több más paraméter is adható ennek az annotációnak, például a karakterkódolás, vagy hogy milyen típusú sortörés karakterrel szeretnénk dolgozni. Továbbá megadhatjuk az elválasztó karaktert, valamint azt is, hogy hány sort ugorjon át a JUnit a fájl elején, amikor majd beolvassa az állományt (ha például van címsor is a fájl elején). Ilyen esetben a fájl elérési útvonalának megadási módja:

```
@CsvFileSource(resources = "/get-age.csv")
```

Dinamikus tesztek

JUnitban használhatunk úgynevezett **dinamikus teszteket** is. Ez azt jelenti, hogy a fejlesztő maga hozhat létre, dinamikusan teszteseteket, kódból. Ekkor egy olyan metódust kell írnom, amely a `@TestFactory` annotációval van ellátva, és `DynamicTest` típusú teszteset példányokat ad vissza valamilyen kollekcióban vagy stream-ben.

Ehhez a `dynamicTest()` metódust kell meghívni, amelynek első paramétere maga a teszteset neve (ez megadható explicit módon és dinamikusan is), második paramétere pedig egy lambda kifejezés, ami valójában a teszteset implementációja.

```
@TestFactory
Stream<DynamicTest> arePalindromes() {
  return Stream.of("racecar", "radar", "mom", "dad")
    .map(text ->
       dynamicTest(
         "Is palindrome? " + text,
         () -> assertTrue(isPalindrome(text))));
}
```

A példában a streamként átadott paraméterek mindegyikével meghívódik ez a metódus, ami először lepéldányosítja a teszteseteket, majd a JUnit a teszteset assertjén belül minden paraméterrel meghívja a tesztelendő metódust.

A következő példában a paramétereket egy olyan tömbként adjuk át, ahol a tömb tartalmazza a bemeneti és az elvárt értékeket is.

```
@TestFactory
Stream<DynamicTest> testGetAge() {
  return Stream
   .of(new Integer[][]{{2000, 0}, {2010, 10}, {2015, 15}, {2050, 50}, {1990,
-10}})
   .map(item ->
      dynamicTest(
        "In year " + item[0] + " employee must be " + item[1] + "years old",
        () -> assertEquals((int) item[1],
        new Employee("John Doe", 2000).getAge(item[0]))));
}
```

Tempdirectory extension

Tesztelésnél problémát okozhat a fájlműveletek tesztelése. Erre akkor lehet szükségünk, amikor az adott metódus valamilyen fájlt ír ki vagy olvas be, dolgoz fel.

Ez valójában már nem unit tesztelés, hanem ezek már integrációs tesztek, hiszen itt már a fájlrendszert is részt vesz a tesztelésben, nem csak egy osztály.

Az első kérdés, ami felvetődik, hogy a tesztelendő fájlokat hol helyezzük el. A JUnitnak erre van egy `@TempDir` annotációja, amely a JUnitnak egy kiterjesztése. Ez úgy működik, hogy a teszteset lefutása előtt létrehoz egy könyvtárat az adott operációs rendszer `temp` könytárán belül, és a teszteset végén pedig letörli ezt a könyvtárat.

Egy fájlt kiíró művelet teszteléséhez először tehát meg kell adni egy `Path` típusú attribútumot, amelyet el kell látni a `@TempDir` annotációval. Ezáltal létre fog jönni az az ideiglenes könyvtár, amelynek az elérési útvonalát ez a `Path` típusú attribútum fogja reprezentálni. Ebbe a könyvtárba a teszteset által meghívott tesztelendő metódus bele tudja

írni a fájlt. Majd a teszteset innen vissza is tudja olvasni a fájl tartalmát, hogy végül az assertben megvizsgálhassa, hogy a fájl tartalma megfelelő-e, azaz a kiírás helyesen lett-e implementálva.

```
public class EmployeeWriterTest {

  @TempDir
  Path tempDir;

  @Test
  void testWriteEmployee() throws IOException {
    Path file = tempDir.resolve("john-doe.txt");
    new EmployeeWriter()
      .write(file, new Employee("John Doe", 1970));
    assertEquals("John Doe, 1970", Files.readString(file));
  }
}
```

JUnit legjobb gyakorlatok

Amikor unit teszteket írunk, akkor érdemes néhány szabályt figyelembe venni és észben tartani. A könnyebb megjegyezhetőség érdekében adtak ennek egy rövidítést: ezek a **F.I.R.S.T. elvek**. Ez a betűszó minden szabálynak az első betűjét tartalmazza, ezek a következők:

- **F – Fast**: A unit teszteknek gyors lefutásúaknak kell lenniük. Ha ugyanis lassúak, sokat kell rájuk várni, akkor a fejlesztő nem fogja őket lefuttatni, és pont ezért az értelmüket vesztik. Fontos, hogy a fejlesztés során minél többször le legyenek futtatva a tesztek, hogy nagyon hamar kiderüljenek az esetleges problémák. A legjobb az, ha a verziókövető rendszerbe be sem kerül olyan hibás kód, amelynél elbuknak a tesztesetek.
- **I – Isolated/Independent**: A tesztesetek legyenek egymástól függetlenek, ne legyenek egymásra hatással. Fontos, hogy bármilyen sorrendben is futtassuk le őket, mindig ugyanazt az eredményt kapjuk. Ezt úgy érhetjük el, ha a tesztesetek között nem tárolunk állapotot. Ezért a statikus változók használatára vigyázzunk, sőt, lehetőleg kerüljük ezek használatát!
- **R – Repeatable**: megismételhetőség. A tesztesetek akkor jók, ha bármennyiszer is futtatjuk őket, mindig ugyanazt az eredményt kapjuk.
- **S – Self-Validating**: a tesztek automatizáltak legyenek, ne kelljen a fejlesztőnek manuálisan/szemmel ellenőriznie semmit, hanem ezt a JUnit keretrendszer automatikusan végezze el. Ez azt is jelenti, hogy a tesztesetben ne írjunk ki a konzolra semmit, ne használjuk a `System.out.println()` metódust, ne használjunk naplózást, hanem ezek helyett a JUnit `assertXXX()` metódusaival ellenőriztessük le a kapott eredményeket.
- **T – Thorough**: alaposság. Lehetőleg az adott, tesztelendő funkció összes elágazását teszteljük le, teszteljük például határértékekkel, extrém nagy, extrém alacsony

értékekkel, üres stringekkel, `null` értékekkel, ékezetes karakterekkel, nagy számokkal, kis számokkal, nullával, stb. A ciklusoknál érdemes arra tesztet írni, amikor egyszer sem fut le a ciklusmag, illetve amikor csak egyszer vagy amikor többször is lefut. Mivel a unit teszteket a fejlesztő írja, ezért bele tud nézni a forráskódba, és így ő maga építheti föl úgy a teszteseteket, hogy mindent alaposan átfogjanak a tesztek.

További tanácsok:

- A lehető legkisebb scope-ot tartalmazza az adott teszteset, tehát egy unit teszt ne több osztály együttműködési rendszerét vizsgálja, hanem egy osztály egyetlen metódusának működését.
- Legyenek egyszerűek a tesztek.
- A tesztek dokumentációs célzattal is használhatóak legyenek, tehát ha meg akarjuk nézni, hogy mire használható az adott osztály, akkor erre már a tesztesetekben is találjunk példákat.
- A teszteset kódja tartsa be ugyanazokat a programozási konvenciókat, mint a forráskód, legyen a kód tiszta, áttekinthető, jól olvasható.
- Ha tudunk, akkor `assertTrue()` és `assertFalse()` helyett `assertEquals()` metódust használjunk, mert ez ki is írja a konzolra egy elbukott teszt esetén, hogy mi az aktuális és az elvárt érték, ezáltal a különbség jobban láthatóvá válik.
- Egy teszteset egyszerre egy logikai állítást ellenőrizzen. Nem kell, hogy egy tesztesetben tényleg csak egy assert legyen, de kevés legyen, és azok is tartozzanak logikailag össze, mind ugyanazt az egy fogalomkört ellenőrizzék.

Gyakori hibák, amelyeket ne kövessünk el:

- Nem kell mindent leellenőrizni az assertben, elég a szükséges dolgokat szúrópróbaszerűen ellenőrizni.
- Használjuk az assert metódusok megfelelő formáját!
- Legyen a tesztlefedettség minél nagyobb! Vannak eszközök arra, amelyek ellenőrzik számunkra, hogy mely kódsorok futottak le a tesztesetek futtatása közben, és ebből kiszámolják számunkra a tesztlefedettség értékét. A legjobb az, ha ez a szám 80% fölötti, mert az azt jelenti, hogy a forráskód sorainak 80%-a lefutott a tesztek futtatása során.
- Ne használjunk külső függőségeket, mert az már nem unit teszt, hanem integrációs teszt.
- Hiba az a két szélsőség is, ha vagy nincsenek unit tesztjeink, vagy pedig mindent le akarunk tesztelni, akár a getter és setter metódusokat is, a generált kódokat vagy third party library-ket, amelyek már eleve le vannak tesztelve. Ez szükségtelen, mert fölöslegesen túl sok energiát vesz el. Emiatt nem cél a 100%-os tesztlefedettség.

JUnit 4 és 5 együttes használata bevezetés

Ebben a fejezetben röviden megnézzük azt, hogyan lehet egyszerre használni a JUnit két különböző verzióját, mivel egy régebb óta létező, de manapság is folyamatosan fejlesztés alatt álló projekt esetén erre szükség lehet.

JUnit 4 és 5 használata

Amennyiben egy projektet már régebb óta fejlesztenek, valószínűleg a unit tesztek (egy része) a JUnit 4-es verziójával készültek el. Ha szeretnénk átállni az 5-ös verzióra, azt megtehetjük, sőt, a JUnit még ahhoz is ad támogatást, hogy a 4-es verzióban írt teszteket 5-ös verziójú tesztekre tudjuk konvertálni. De arra is van lehetőség, hogy a régebben írt teszteket 4-es verzióban hagyjuk, az újakat pedig már 5-ösben írjuk meg, ugyanis a JUnit két verziója képes az együttműködésre.

A JUnit a következő modulokból épül fel:

- *Platform*: ez futtatja a teszteseteket.
- *Jupiter*: JUnit 5-ös tesztesetek implementálására való.
- *Vintage*: ez futtatja a JUnit 4-es teszteseteket.

Maven projekt esetén nekünk csak annyi a teendőnk, hogy a használni kívánt modul(ok)-at függőségként fel kell vennünk a `pom.xml` állományba.

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>${junit5.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit4.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <version>${junit5.version}</version>
    <scope>test</scope>
</dependency>
```

Hamcrest bevezetés

Ebben a fejezetben a Hamcrestről third party library-ről lesz szó, amely a JUnitot kiegészítve szélesebb lehetőségeket nyújt assertek írására.

Hamcrest

A JUnit képességei az assertekkel kapcsolatban eléggé korlátozottak, különösen akkor, ha JavaBeans osztályok, vagy ezek kollekcióinak példányait szeretnénk vizsgálni. Ha széleskörűbb funkcionalitást szeretnénk, akkor érdemes valamilyen third party library-t használni.

Az egyik ilyen a **Hamcrest**. Ez ún. *harmadik generációs assertek* írására való. A neve a *Matchers* szóból jön, annak az anagrammája. A Hamcrest a JUnit assertjeihez képest a következőkben nyújt többet:

- Informatívabb hibaüzeneteket ad.
- Több lehetőséget kínál az összehasonlításra.
- Könnyen használható, jól olvasható API-ja van.
- Egyszerűbb összehasonlításokból bonyolultabb egyezőségvizsgálatokat lehet összerakni.
- Saját asserteket is lehet implementálni.

Egyszerű ellenőrzésekhez a Hamcrestben az `assertThat()` metódust kell használnunk, amelynek az első paramétere az aktuális érték, a második paramétere pedig egy matcher.

A leggyakrabban használt matcher az `equalTo()`, mely egyenlőséget vizsgál.

```
assertThat(employee.getAge(2019), equalTo(40));
// instanceOf, nullValue, sameInstance, stb.
```

Ilyen matcherekből sokféle van, például a következők:

- `instanceOf()`: azt lehet vizsgálni, hogy egy adott osztálynak példánya-e az adott objektum.
- `nullValue()`: megadja, hogy egy adott objektum értéke `null`-e.
- `sameInstance()`: visszaadja, hogy ugyanaz-e a referencia.
- `closeTo()`: ha lebegőpontos számoknál egyenlőséget akarunk vizsgálni, akkor ebben a metódusban megadhatunk egy tűrési határt, egy küszöbértéket.
- Lehet különböző kisebb-nagyobb relációkat is vizsgálni (pl. `greaterThan()` és `greaterThanOrEqualTo()` metódusok).
- `is()`: az olvashatóságot könnyítő metódus.
- Stringek összehasonlítására való metódusok is vannak, melyek mind elég beszédes nevekkel rendelkeznek, pl. `startsWith()`, `endsWith()`, `startsWithIgnoringCase()`, `equalToIgnoringWhiteSpace()`, `containsString()`, stb.

Pl. így használhatóak:

```
assertThat(pi, closeTo(3.14, 0.005));

// Olvashatóságot könnyítve
assertThat(employee.getAge(2019), is(equalTo(40)));
```

```
assertThat(employee.getName(),
  startsWithIgnoringCase("john"));
```

Ha JavaBean-ekkel akarunk dolgozni, akkor le lehet kérdezni például azt, hogy az adott objektumnak van-e egy megadott nevű attribútuma, és annak az-e az értéke, amelyet paraméterül átadtunk. Sőt, mivel *property*-re vizsgál, ezért az is kiderül, hogy az adott attribútumhoz tartozik-e az osztályban getter metódus. Ezzel kapcsolatban arra kell csak figyelni, hogy ha változtatjuk az attribútum nevét, akkor a tesztesetben is át kell írni, mert az új értékkel nem fog lefutni a teszteset.

```
assertThat(employee, hasProperty("name", equalTo("John Doe")));
```

A Hamcrestben lehetőség van arra is, hogy a matchereket összeépítsük egy komplexebb rendszerré. Az `allOf()`, `anyOf()` és a `not()` metódusoknak további matchereket lehet megadni paraméterül, és ezek együtt futnak le a tesztesetben:

```
assertThat(employee, allOf(hasProperty("name", equalTo("John Doe")),
                hasProperty("yearOfBirth", equalTo(1970))));
```

A Hamcrest a kollekciók kezelésére is tartalmaz különböző metódusokat:

- `contains()`: Két kollekciót hasonlít össze, és megtévesztő a neve, ugyanis nem csak tartalmazást, hanem pontos egyezőséget vizsgál (elemek száma és sorrendje).
- `containsInAnyOrder()`: ugyanaz, mint az előző, csak az elemek sorrendje lehet tetszőleges.
- `hasItem()`: részegyezőséget vizsgál.

Ezeknek a metódusoknak is lehet másik matchert megadni paraméterül, és így már nem csak számokat és szövegeket lehet összehasonlítani, hanem bonyolultabb, objektumokat tartalmazó kollekciókat is.

```
// Pontos egyezőséget néz
assertThat(List.of("John", "Jane", "Jack"),
  contains("John", "Jane", "Jack"));
// containsInAnyOrder

// Rész egyezőség
assertThat(List.of("John", "Jane", "Jack"), hasItem("Jane"));

assertThat(List.of(employees),
  hasItem(hasProperty("name", startsWithIgnoringCase("john"))));
// hasEntry, hasKey, hasValue Map esetén
```

A Hamcrest használatához a következő függőséget kell felvenni a Maven `pom.xml` állományába:

```
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest</artifactId>
    <version>${hamcrest.version}</version>
```

```
    <scope>test</scope>
</dependency>
```

Saját Hamcrest matcher implementálása

A Hamcrestben bonyolultabb objektumok vizsgálata a `hasProperty()` metódussal történhet. Azonban ez nem egy refactor-tűrő megoldás, mert ha az adott attribútumot átnevezzük, akkor a tesztesetben nem változik ezzel egyidőben a string, amiben az attribútum nevét megadtunk. Vagyis ha nem írjuk át kézzel, akkor a teszteset onnantól kezdve hibára fog futni. Ez az egyik legtipikusabb esete annak, hogy mikor lehet szükség saját matcher implementálására.

Saját matcher osztály implementálásakor a mi osztályunknak a `TypeSafeMatcher` ősosztályból kell öröklődnie, és annak generikus típusaként meg kell adni azt a típust, amelyre majd a vizsgálatainkat el akarjuk végezni. Ha esetleg másik matchert is szeretnénk majd később felhasználni ebben az osztályban, azt konstruktorban kell átadnunk az osztály privát attribútumának. Ebben az osztályban valójában a `matchesSafely()` metódus vizsgálja az egyezőséget.

```
public class EmployeeWithNameMatcher extends TypeSafeMatcher<Employee> {

  private Matcher<String> matcher;

  public EmployeeWithNameMatcher(Matcher<String> matcher) {
      this.matcher = matcher;
  }

  @Override
  protected boolean matchesSafely(Employee item) {
      return matcher.matches(item.getName());
  }

}
```

Van két metódus, amelyeknek meg lehet adni, hogy milyen hibaüzenet kerüljön kiírásra, ha a teszteset elbukna. A `describeMismatchSafely()` azt mondja meg, hogy hogyan írja ki a program az aktuális értéket, ha nem történik egyezőség, a `describeTo()` pedig azt, hogy hogyan kerüljön az elvárt érték kiírásra a konzolon. Az alábbi kódból látható, hogy nem csak az általunk implementált matchernek az üzenetét kell megadni, hanem hozzá kell kombinálni a beágyazott matchernek az üzenetét is.

```
@Override
protected void describeMismatchSafely(Employee item, Description
mismatchDescription) {
  mismatchDescription
    .appendText(" employee with name ")
    .appendValue(item.getName());
}
```

```
@Override
public void describeTo(Description description) {
  description
    .appendText(" employee with name ")
    .appendDescriptionOf(matcher);
}
```

Még egy további metódust érdemes létrehozni, amely egy statikus factory metódus, és az a feladata, hogy létrehozzon számunkra egy példányt a saját magunk által írt matcherből. Így implementálva a saját matcherünk használata kifejezetten egyszerűvé válik, pontosan olyanná, mint a Hamcrest beépített matcherei.

```
public static Matcher employeeWithName(Matcher<String> matcher) {
    return new EmployeeWithNameMatcher(matcher);
}
```

És a következőképp használható:

```
assertThat(employee, employeeWithName(startsWithIgnoringCase("jack")));
```

AssertJ bevezetés

Ebben a fejezetben az AssertJ third party library használati lehetőségeiről lesz szó, amellyel a JUnitot kiegészítve főleg JavaBeanekre és kollekciókra tudunk egyszerűen asserteket írni.

AssertJ

Ha JUnitban JavaBean-ekre vagy kollekciókra akarunk ellenőrzéseket végezni, akkor korlátokba ütközünk. Az **AssertJ** egy olyan library, amivel egyszerűbben tudunk ilyen típusú asserteket írni.

Az AssertJ assertjei *fluent*-ek, vagyis a benne megfogalmazott állítások hasonlítanak a természetes nyelvben megfogalmazott mondatokra. A metódusokat nem kell statikusan importálni, ezért a fejlesztőeszközök segíteni tudnak a különböző metódusok használatában. Lehetőség van saját assertek írására vagy generálására is.

Az egyszerű egyezőségvizsgálatok az `assertThat()` metódussal végezhetők, amelyben a paraméter a tesztelendő metódus által visszaadott aktuális eredmény, az elvárt eredményt pedig nem második paraméterként adhatjuk meg, hanem további metódushívásokkal. Funkciójában megegyezik a JUnit `assertEquals()` metódusával. Az alábbi példán jól látható mindez, valamint az, hogy mit is jelent az, hogy *fluent API*: egy asserten belül több elvárás is megfogalmazható és láncoltan hívhatóak a metódusok.

```
assertThat(employee.getName())
    .startsWith("John")
    .endsWith("Doe");
```

Kollekciókkal kapcsolatban is több feltételt adhatunk meg egy utasításban. Itt a következő metódusok szerepelnek: `hasSize()` (az `assertThat()`-nek paraméterként megadott kollekció mérete), `contains()` (tartalmazza-e az adott elemeket, az esetleges többi egyéb elemtől függetlenül), `doesNotContain()` (ne tartalmazza az adott elemeket). Van még a `containsOnly()` metódus, amely viszont azt vizsgálja, hogy csak a megadott elemek legyenek a kollekcióban, és más ne. Az `extracting()` metódus a megadott név alapján a listát egy más típusú elemek listájává alakítja, és onnantól azt a listát vizsgálja. Lehetőség van az attribútum nevét explicit módon és refactor-tűrő módon is megadni.

```
assertThat(employeeNames)
  .hasSize(2)
  .contains("John Doe", "Jack Doe")
  .doesNotContain("Jane Doe");
// containsOnly

assertThat(employees)
  .extracting("name")
  .contains("John Doe", "Jack Doe");
assertThat(employees)
  .extracting(Employee::getName)
  .contains("John Doe");
```

Lehetőség van arra is, hogy egyszerre ne csak egy, hanem több attribútum értékét is "kinyerjük" az objektumból. Az `extracting()` metódusnak több method reference-t is át lehet adni, és ekkor az objektumból több attribútum értékét is lekérdezi. Az eredményt az AssertJ egy ún. *tuple*-ben tárolja. Ez nem Java fogalom, Javaban nincs ilyen. A tuple nem más, mint több érték tárolására szolgáló adatszerkezet. AssertJ-ben ilyet a `tuple()` metódus hívásával lehet létrehozni. Az `assertThat()` metódus tehát összehasonlítja a visszakapott és a paraméterként átadott tuple-t.

```
assertThat(employees)
  .extracting(Employee::getName, Employee::getYearOfBirth)
  .contains(tuple("John Doe", 1970),
    tuple("Jack Doe", 1980));
```

Szűrni is lehet az adott kollekcióban a `filteredOn()` metódus használatával. Paraméterként ez is egy lambda-kifejezést vár, és kiszűri a kollekcióból azokat az elemeket, amelyek az adott feltételnek megfelelnek.

```
assertThat(employees)
  .filteredOn(e -> e.getName().contains("a"))
  .extracting(Employee::getName)
  .containsOnly("Jack Doe");
```

Testre is lehet szabni a hibaüzeneteket. Az `as()` metódus használatával meg lehet adni, hogyha elbukik az összehasonlítás, akkor mi kerüljön kiírásra. Ez egy *description*, az AssertJ által megjelenített hibaüzenet előtt fog megjelenni, szögletes zárójelek között.

```
assertThat(employee.getAge())
  .as("check %s's age", employee.getName())
  .isEqualTo(40);
```

Ha viszont a teljes üzenetet át szeretnénk írni, akkor a `withFailMessage()` metódussal tudjuk ezt megadni. Az is látható az alább található példakódból, hogy az üzenetek paraméterezett stringként is megadhatóak.

```
assertThat(employee.getName())
  .withFailMessage("should be %s", "John Doe")
  .isEqualTo("John Doe");
```

Lehet megadni úgynevezett *soft assert*-et is. Ez azt jelenti, hogy nem bukik el azonnal a teszteset, ha a feltétel nem teljesül, hanem ezeket összegyűjti, és csak a végén, egyben írja ki a hibára futott asserteket. Ehhez egy `SoftAssertions` objektumot kell példányosítani, amelynek ugyanolyan metódusai vannak, mint a sima `Assertions`-nek. Ezek a metódusok sorban hívhatóak egymás után, és csak a végén, az `assertAll()` metódusnál fog leállni a végrehajtás és fognak kiíródni a hibára futott assertek. Ha nem akarjuk mi magunk ezt példányosítani, akkor azt is megtehetjük, hogy a tesztosztályunkra rátesszük az `@ExtendWith(SoftAssertionsExtension.class)` annotációt, és innentől kezdve a tesztmetódusunk *parameter injection*-nel fogja kapni ezt a `SoftAssertions` példányt. Ez esetben nem kell a végén `assertAll()` hívás.

```
@ExtendWith(SoftAssertionsExtension.class)
public class EmployeeTest {

  @Test
  public void testEmployee(SoftAssertions softly) {
    Employee employee = ...

    softly.assertThat(employee.getName())
      .isEqualTo("John Doe");
    softly.assertThat(employee.getAge())
      .isEqualTo(40);
    // Nem kell assertAll() hívás!
  }
}
```

Lehetőség van `Assumption` megadására is. Ez esetben megáll ugyan a teszteset futása egy nem teljesült feltétel esetén, de nem fogja az AssertJ hibásnak jelenteni a tesztesetet. Ez akkor jó, mikor csak egy bizonyos feltétel teljesülése esetén van értelme a teszteset lefuttatásának.

```
assumeThat(employee.getName()).isEqualTo("John Doe");
```

Az AssertJ használatához a Maven `pom.xml` állományába a következő külső függőséget kell felvenni:

```
<dependency>
  <groupId>org.assertj</groupId>
```

```
  <artifactId>assertj-core</artifactId>
  <version>${assertj.version}</version>
  <scope>test</scope>
</dependency>
```

AssertJ kiterjeszthetőség

Az AssertJ esetén is lehet további kiterjesztéseket implementálni.

Újrafelhasználható asserteket `Condition`-nel lehet előállítani. Példányosítani kell egy `Condition`-t, meg kell neki adni típusparaméterként azt az osztályt, amelyre az adott feltétel vonatkozik, majd át kell neki adni egy lambda kifejezést, amelyben megadunk egy feltételt és egy üzenetet, hogy mit írjon ki akkor, ha ez a feltétel nem teljesül. Egy `assertThat()` implementálásakor meghívható a `has()` metódus, aminek egy ilyen `Condition`-t adhatunk át paraméterül.

```
Condition<Employee> familyNameDoe =
 new Condition<>(e-> e.getName().endsWith("Doe"), "family name is Doe");
assertThat(employee).has(familyNameDoe);
```

Fontos, hogy a `Condition`-nek kifejező nevet adjunk, mert így angol mondathoz hasonló kifejezést kaphatunk, ami nagyon könnyen olvasható, értelmezhető. Az `is()` metódus ugyanerre használható, csak annyi a haszna, hogy ha angol mondatokként akarjuk megírni a kódsorokat, akkor néha a `has()`, néha az `is()` illik jobban a mondatba. A `doesNotHave()` és a `not()` metódus negálja a paraméterként átadott `Condition`-t.

Ha ugyanarra az aktuális értékre több `Condition`-t is meg akarunk adni, akkor arra az `allOf()` és az `anyOf()` metódusok használhatóak. (Az első azt nézi, hogy a megadott `Condition`-ök közül mindenek teljesülnie kell, a második pedig azt, hogy a megadottak közül legalább egynek teljesülnie kell.)

Nem csak egy értékre, hanem kollekciókra is megadhatunk feltételeket. Az `are()` és a `have()` azt nézi, hogy a megadott `Condition` teljesül-e az adott kollekcióban lévő összes elemre (itt is ugyanazt csinálja mindkét metódus). Az `areAtLeast()` metódus azt nézi, hogy a kollekcióból legalább mennyinek kell megfelelnie a paraméterként átadott `Condition`-nek, az `areAtMost()` azt, hogy legfeljebb mennyi feleljen meg, az `areExactly()` pedig azt, hogy pontosan mennyi feleljen meg.

Lehet implementálni saját assertet is. Ekkor egy saját osztályt kell írnunk, amelynek az `AbstractAssert`-ből kell leszármaznia, paraméterként meg kell adni neki ezt a most létrehozott osztályt és azt az osztályt is, amelyre vonatkozik. A könnyebb kezelhetőség végett írhatunk ebbe az osztályba egy statikus metódust, amely létrehozza számunkra a saját assertünk egy példányát (ez egy statikus factory metódus, amely meghívja az assert osztályunk konstruktorát). Majd definiálhatjuk az osztályban azokat a metódusokat, amiket majd szeretnénk meghívni, amikkel szeretnénk az asserteket kiegészíteni. Ahhoz, hogy *fluent interface*-t tudjunk használni, és láncoltan tudjuk hívni a metódusainkat, ezeknek a metódusoknak azt a konkrét példányt kell visszaadniuk, amelyen meghívták őket. (`return this;`)

```
public class EmployeeAssert extends AbstractAssert<EmployeeAssert, Employee>
{

  public static EmployeeAssert assertThat(Employee employee) {
    return new EmployeeAssert(employee);
  }

  public EmployeeAssert(Employee employee) {
    super(employee, EmployeeAssert.class);
  }

  public EmployeeAssert hasName(String name) {
    if (!Objects.equals(actual.getName(), name)) {
      failWithMessage(
        "Expected employees name " +
          "to be <%s> but was <%s>",
        name, actual.getName());
    }

    return this;
  }
}
```

Mockito bevezetés

Ebben a fejezetben megvizsgáljuk a unit tesztelés egyszerű lehetőségeit egy olyan osztály esetében, melynek külső függőségei vannak. Segítségünkre lesz ebben a Mockito keretrendszer.

Mockito

Amikor unit teszteket írunk, gyakran abba a problémába ütközhetünk, hogy amely osztályt tesztelni akarunk (*SUT – system under test*), az az osztály rendelkezik külső függőségekkel, együttműködik más osztályokkal. Egy ilyen osztály tesztelése viszont már nem unit teszt, hanem integrációs teszt.

Hogyan lehet egy ilyen osztályt a külső függőségei nélkül, önmagában tesztelni? Úgy, hogy lecseréljük a külső függőségét egy tesztelésre előkészített példánnyal, ami nem az eredeti példány, hanem annak egy cseréje. Ez ugyanolyan interfésszel rendelkezik, mint az eredeti osztályunk, de úgy vannak benne implementálva a metódusok, hogy az a teszteléshez megfelelő legyen. Ezeknek a csereosztályoknak a neve **test double**.

A test double-k típusai:

- *Dummy*: ez gyakorlatilag nem csinál semmit, a metódusai üresek, csak arra van, hogy leforduljon a tesztesetben az adott osztály.

- *Fake*: ez már működő implementáció, hasonlóan viselkedik, mint az éles, de sok korlátozással rendelkezik. Ilyen például egy valódi adatbázis helyett egy *in-memory*, tehát egy memóriában működő, az adatokat a memóriába elmentő kollekció.
- *Stub*: olyan test double, amely előkészített válaszokkal rendelkezik. Tehát megadjuk neki, hogy ha meghívják valamelyik metódusát, akkor konkrétan mit adjon vissza. Emiatt nem is használható általánosan, csak az adott, konkrét tesztesetnél tud működni.
- *Mock*: ez nemcsak hogy meg lehet mondani neki, hogy milyen válaszokat adjon vissza, de még arra is képes, hogy rögzíti a tesztelendő osztály rajta futtatott hívásait, és ezekre tudunk asserteket írni, hogy a megfelelő metódusai lettek-e meghívva a megfelelő paraméterekkel.

A unit teszteket két nagyobb csoportba lehet osztani. Az egyik az *állapot alapú* unit tesztek csoportja, amely a tesztelendő osztály egy metódusát meghívja, és megnézi, hogy a megadott bemenetre a megfelelő kimenetet kapjuk-e vissza. A másik a *viselkedés alapú* tesztesetek, ami pedig azt vizsgálja, hogy egy külső kapcsolattal rendelkező osztály a külső kapcsolatát a megfelelő módon veszi-e igénybe, vagyis a kapcsolódó osztályban a megfelelő metódusokat hívja-e meg a megfelelő paraméterekkel.

A **Mockito** egy olyan keretrendszer, amely mock objektumok programozott előállítására szolgál. Ezek az objektumok úgy néznek ki, mint az eredeti példányok, ugyanolyan interfésszel rendelkeznek, azonban programozottan definiálható rajtuk az, hogyha meghívunk rajtuk valamilyen metódust, akkor milyen értékeket adjanak vissza, valamint ellenőrizhető az, hogy a megfelelő metódus meg lett-e hívva, és az is ellenőrizhető, hogy milyen paraméterrel.

Alapból ha generálunk egy mock objektumot, akkor ez olyan metódusokat tartalmaz, ami objektum esetén `null`-t ad vissza, boolean esetén `false` értéket, primitív típusok esetén pedig `0` értéket ad vissza.

A leggyakoribb helyzet, amikor mock objektumokra szükségünk lehet, az az, amikor az üzleti logikát szeretnénk tesztelni, az pedig hívja a perzisztens réteget. Ekkor az üzleti logika rétegben lévő osztálynak (Service) a függősége az adatbázis műveletekért felelős osztály (Repository). A Service osztály a függőséget a konstruktorában paraméterként kapja meg, így az könnyen kicserélhető egy mock objektumra.

Teszteléskor ez úgy néz ki, hogy meghívjuk a Mockito `mock()` metódusát, amely létrehoz számunkra egy mock objektumot. Ennek a metódusnak paraméterül a létrehozandó objektum osztályát kell átadni, és ebből fogja tudni, hogy milyen típusú mock objektumot kell létrehoznia. Ennek a metódusai üresek lesznek, nem fognak csinálni semmit.

Adott a következő repository:

```
public class EmployeeRepository {

  private List<Employee> employees = new ArrayList<>();

  public void saveEmployee(Employee employee) {
```

```
      employees.add(employee);
  }

  public Optional<Employee>
      findEmployeeByName(String name) {
    return employees
      .stream()
      .filter(e -> e.getName().equals(name))
      .findFirst();
  }
}
```

Valamint az ezt használó service:

```
public class EmployeeService {

  private EmployeeRepository employeeRepository;

  public EmployeeService(EmployeeRepository employeeRepository) {
    this.employeeRepository = employeeRepository;
  }

  public boolean createEmployee(String name, int yearOfBirth) {
    name = name.trim();
    Optional<Employee> employee = employeeRepository
      .findEmployeeByName(name);
    if (employee.isPresent()) {
      return false;
    }
    else {
      employeeRepository.saveEmployee(
        new Employee(name, yearOfBirth));
      return true;
    }
  }
}
```

Ekkor a teszteset:

```
@Test
void testSaveEmployee() {
    EmployeeRepository repository = mock(EmployeeRepository.class);
    EmployeeService service = new EmployeeService(repository);
    boolean created = service.createEmployee("John Doe", 1970);
    assertTrue(created);
}
```

Megírható máshogy is ugyanez a teszteset, ahol nem mi hozzuk létre a mock objektumot, hanem a Mockito. Ehhez a tesztosztályra rá kell rakni az `@ExtendWith(MockitoExtension.class)` annotációt. Amely osztály mockolni szeretnénk (a

Repository-t), azt felvesszük attribútumként és a `@Mock` annotációt rakjuk rá, amelyiknek pedig át akarjuk adni (a Service-nek) **dependency injection**-nel, arra pedig az `@InjectMocks` annotációt tesszük rá.

```
@ExtendWith(MockitoExtension.class)
public class EmployeeServiceTest {

  @Mock
  EmployeeRepository repository;

  @InjectMocks
  EmployeeService service;

  @Test
  void testSaveEmployee() {
    boolean created =
      service.createEmployee("John Doe", 1970);
    assertTrue(created);
  }
}
```

A példaként hozott két kód működése teljesen megegyezik.

Nézzük meg, hogyan lehet megmondani a mockolt objektumnak, hogy bizonyos metódusának meghívása esetén mivel térjen vissza: egymás után meg kell hívni a `when()` és a `thenReturn()` metódusokat, a megfelelő paraméterekkel.

```
when(repo.findEmployeeByName(anyString()))
                .thenReturn(Optional.of(new Employee("John Doe", 1970)));

boolean created = service.createEmployee("John Doe", 1970);
assertFalse(created);
```

Ha azt akarjuk, hogy kivételt dobjon a metódus, akkor pedig a `when()` és a `thenThrow()` metódusokat kell hívni.

```
when(repo.findEmployeeByName(anyString()))
  .thenThrow(
    new IllegalStateException("Constraint violation"));
```

Még azt is ellenőrizni lehet, hogy a mock objektumon egy adott metódus meghívásra került-e, ezt a `verify()` metódussal tehetjük meg.

```
verify(repo).saveEmployee(any());
```

Olyan feltételt is szabhatunk, hogy a teszteset lefutása közben nem lehet meghívni egy konkrét metódust. Ezt a `never()` metódussal ellenőrizhetjük. Ha ez mégis meghívásra kerül, akkor a teszteset elbukik.

```
verify(repo, never()).saveEmployee(any());
```

Az `any()`-t és az `anyString()`-et paraméterül lehet átadni, és azt jelentik, hogy az adott metódus bármilyen vagy bármilyen szöveg típusú paraméterrel meghívásra került-e.

`ArgumentCaptor`-ral azt lehet ellenőrizni, hogy az adott metódus milyen paraméterrel került meghívásra.

Ehhez létre kell hozni egy ArgumentCaptor példányt, és ezt kell paraméterül adni az ellenőrzésnél. Ekkor a Mockito az ArgumentCaptor értékét beállítja arra az értékre, amely paraméter értékkel a metódust meghívták. Utána ezt az értéket az ArgumentCaptor `getValue()` metódusával le lehet kérni.

```
ArgumentCaptor<Employee> employeeCaptor =
  ArgumentCaptor.forClass(Employee.class);
verify(repo).saveEmployee(employeeCaptor.capture());

assertEquals("John Doe", employeeCaptor.getValue().getName());
assertEquals(1970, employeeCaptor.getValue().getYearOfBirth());
```

Lambda kifejezések használatával ezt egyszerűbben megfogalmazhatjuk:

```
verify(repo).saveEmployee(argThat(e -> e.getName().equals("John Doe")));
```

Ez azt ellenőrzi, hogy a `saveEmployee()` metódust olyan paraméterrel hívták-e meg, melyre igaz a paraméterként átadott lambda kifejezés.

Ahhoz, hogy a Mockito-t használni tudjuk, a Maven projekt `pom.xml` állományába a következő függőséget kell felvenni:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
```

Tesztlefedettség

Ebben a fejezetben megismerkedhetsz a tesztlefedettség fogalmával, valamint egy, ennek a mérésére használható egyszerű eszközzel, a Jacocoval.

Tesztlefedettség

A tesztlefedettség egy mérőszám, és azt mondja meg, hogy az adott alkalmazás tesztelésekor az alkalmazás kódsorainak hány százaléka futott le. Ezt a mérőszámot meg lehet adni az utasítások szintjén is (tehát hogy az alkalmazás utasításai közül mennyi futott le a tesztesetek futtatása közben), illetve meg lehet adni aszerint is, hogy a végrehajtható ágak közül mennyi futott le (ez különösen elágazásoknál és ciklusoknál hasznos).

A 70-80%-os lefedettség már jónak mondható. Ez azt jelenti, hogy az alkalmazás kódsorainak ekkora része futott le a tesztesetek futtatása során, tehát az alkalmazásnak ekkora része van tesztekkel lefedve. Ennél nagyobb lefedettséget unit tesztekkel elérni már elég nehéz és nincs is értelme, ugyanis akkor már a getter és setter metódusokat vagy más generált metódusokat is tesztelnünk kellene. Ez nem biztos, hogy megéri a befektetett munkát.

Mikor a tesztlefedettséget nézzük, akkor **white box testing** módszert használunk, vagyis először megnézzük az alkalmazás forráskódját és aztán az alapján írunk tesztet, azért, hogy biztosan minden lehetséges ágra ráfusson a vezérlés.

A tesztlefedettség akár manuális tesztelésnél is használható, tehát lehetséges az, hogy beállítunk egy tesztlefedettség eszközt, amely méri, hogy a kódsorok mely hányada fut le, majd manuális tesztelést végzünk (tehát például kattintgatunk a felületen), és végül meg tudjuk nézni, hogy mely kódsorok futottak le és melyek nem. Ezzel tudunk segítséget nyújtani a tesztelőknek, hogy milyen ágakat nem teszteltek még végig.

A különböző fejlesztőeszközök támogatást adhatnak a tesztlefedettség méréséhez, tehát tudják úgy futtatni a teszteseteket, hogy közben rögzítik és vizualizálják a fejlesztő számára, hogy mely kódsorok futottak le a tesztesetek futtatása közben.

Az egyik legelterjedtebb eszköz a tesztlefedettség mérésére a **Jacoco**. Ennek használatához Maven projekt esetén a jacoco-maven-plugint kell felvenni a `pom.xml`állományba.

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>${jacoco.version}</version>
    <executions>
        <execution>
            <id>default-prepare-agent</id>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

A Jacoco bájtkód manipulációt végez olyan módon, hogy az adott osztályok saját maguk rögzítsék le egy fájlba, hogy mely utasítások futottak le. A futtatás során létre fog jönni tehát egy `jacoco.exec` bináris állomány, amelybe bele van kódolva, hogy mely utasítás hányszor futott le a tesztesetek futtatása során. Ahhoz, hogy ezt a fájlt olvasható formában is meg tudjuk jeleníteni, egy reportot kell belőle készíteni az `mvn jacoco:report` paranccsal. Ekkor HTML formátumban, jól láthatóan jelenik meg számunkra az adott alkalmazás tesztlefedettsége, mind a statisztikai adatok, mind pedig a konkrét, lefutott kódsorok.