03. Java Unit Test

Java Unit Test 방법에 대해 기술합니다.
해당 문서는 세부 예제를 포함하여 작성될 예정 입니다.
작성 범위는 테스트 개요와 작성 방법 그리고 CI 툴과의 연동 부분입니다.

I. 테스트 개요

1. JUnit의 핵심

```
   .
  - public  .
  -      .


    4 .
  - @Test   .
  - public  .
  -    .
  -  void  .
```

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
        @Test
        public oid testAdd() {
                Calculator calculator = new Calculator();
                double result = calculator.add(1, 1);

                assertEquals(2, result, 0);
        }
}
```

- JUnit은 각 @Test메서드를 호출할 때마다 테스트 클래스의 인스턴스를 새로 생성한다.
- 테스트 메서드들을 독립된 메모리 공간에서 실행시킴으로써, 혹시 모를 의도치 않은 부작용을 방지하기 위함이다.
- 테스트 검증에는 JUnit의 Assert 클래스에 정의된 assert 메서드를 사용한다.

- assert 메서드

assert 메서드	설명
assertArrayEquals ("message", A, B)	배열 A와 B가 일치함을 확인한다.
assertEquals ("message", A, B)	객체 A와 B가 일치함을 확인한다. B를 파라미터로 A의 equals() 메서드를 호출한다.(A.equals(B))
assertSame ("message", A, B)	객체 A와 B가 같은 객체임을 확인한다. assertEquals 메서드는 두 객체의 값이 같은가를 검사하는데 반해(equals 메서드 사용), assertSame 메서드는 두 객체가 동일한, 즉 하나의 객체인가를 검사한다.
assertTrue ("message", A)	조건 A가 참(true)임을 확인한다.
assertNotNull ("message", A)	객체 A가 null이 아님을 확인한다.

- 여러개의 테스트 클래스를 동시에 실행해야 할 때는, 테스트 스위트라 불리는 또 다른 객체를 생성한다.
- 테스트 스위트는 특수한 형태의 테스트 러너로, 테스트 클래스와 똑같은 방식으로 실행할 수 있다.
- 테스트 클래스와 스위트, 러너의 동작 방식을 이해한다면 어떠한 클래스도 거뜬히 작성해내는 경지에 오를 것이다.

- 이 세 객체는 JUnit 프레임워크를 지탱하는 척추에 비유될 수 있다.

- JUnit 핵심 객체

JUnit 개념	역할
Assert	테스트하려는 조건을 명시한다. assert 메서드는 조건이 만족되면 아무 일도 없었다는 듯이 조용히 지나가며, 만족하지 못하면 얘외를 던진다.
Test	@Test 어노테이션이 부여된 메서드로, 하나의 테스트를 뜻한다. Junit은 먼저 메서드를 포함하는 클래스의 인스턴스를 만들고, 어너테이션된 메서드를 찾아 호출한다.
Test 클래스	@Test 메서드를 포함한 클래스이다.
Suite	스위트는 여러 테스트 클래스를 하나로 묶는 수단을 제공한다.
Runner	러너는 테스트를 실행시킨다.

파라메터화(parameterized) 테스트

```
// 1.
@RunWith(value=Parameterized.class)
public class ParameterizedTest {

        // 2.
        private double expected;
        private double valueOne;
        private double valueTwo;

        // 3.
        @Parameters
        public static Collection<Integer[]> getTestParameters() {
                return Arrays.asList(new Integer[][] {
                                {2, 1, 1},  //  ,  1,  2
                                {3, 2, 1},        //  ,  1,  2
                                {4, 3, 1}        //  ,  1,  2
                        }
                );
        }

        // 4.
        public ParameterizedTest(double expected, double valueOne, double valueTwo) {
                this.expected = expected;
                this.valueOne = valueOne;
                this.valueTwo = valueTwo;
        }

        // 5.
        @Test
        public void sum() {
                // 6.
                Calculator cal = new Calculator();

                // 7.
                assertEquals(expected, cal.add(valueOne, valueTwo), 0);
        }
}
```

- 파라메터화 테스트 러너를 사용하려면 다음 조건을 만족시켜야 한다.
    1. 테스트 클래스에는 반드시 @RunWith 어노테이션을 부여해야 하며, 그 파라미터로는 Parameterized 클래스를 사용한다.
    2. 테스트에 사용될 값을 인스턴스 변수로 선언하고
    3. @Parameters라 표시된 메서드가 하나 필요하다.
        - @Paramters public static java.util.Collection 이어야 한다.
        - 어떠한 파라미터도 입력 받아서는 안된다.
        - Collection의 원소는 배열(array)이고, 길이는 모두 같아야 한다.
    4. 그 길이는 유일한 public 생성자가 받는 파라미터의 수와도 일치해야 한다.

5. 테스트 메서드 sum()을 구현했다.
6. sum 메서드는 Calculator 프로그램의 인스턴스를 생성하고,
7. 앞서 제공된 파라미터들을 사용해 결과를 단언한다.

- JUnit 런타임 작동 방식
    1. 정적 메서드인 getTestParameters를 호출해 컬렉션 객체를 얻는다.
        - 컬렉션에 저장된 배열의 수만큼 순환한다.
    2. JUnit은 유일한 public 생성자를 찾는다.
        - 이때, public 생성자가 두 개 이상이면 AssertionError를 던진다.
        - 이제 찾은 생성자에 배열의 원소를 파라미터로 넣어 호출한다.
        - 첫번째 배열 {2, 1, 1}을 파라미터로 입력 파라미터 세 개짜리 생성자를 호출할 것이다.
    3. @Test 메서드를 호출한다.

"테스트는 두려움을 지루함으로 변화시켜주는 프로그래머의 돌이다." 켄트 백 : 테스트 주도 개발

부트스트랩 단계

```
import static org.junit.Assert.*;
import org.junit.Test;

// 1.
public class TestDefaultController {
        private DefaultController controller;

        // 2.
        @Before
        public void instantiate() throws Exception {
                controller = new DefaultController();
        }

        // 3.
        @Test
        public void testMethod() {
                // 4.
                throw new RuntimeException("implement me");
        }
}
```

1. 테스트 클래스의 이름에는 Test라는 접두어를 붙인다.
2. DefaultController 인스턴스를 만들기 위해 @Before 어노테이션 메서드를 이용한다.
    - @Before 메서드는 각 테스트 메서드 사이에서 호출되는 JUnit의 기본 확장 포인트이다.
3. 실행할 테스트 케이스가 하나도 없으면 안되므로 더미 테스트 메서드를 추가하였다.
    - 이후에 테스트를 수행할 기반이 다 갖춰졌다는 확신이 서면, 그 때 진짜 테스트 메서드를 추가하면 된다.
    - 이 테스트는 실행은 되지만, 실패하기 때문에 다음 단계는 테스트가 성공하도록 수정할 것이다.
4. 구현이 덜 끝난 테스트 코드는 예외를 던져야 한다는 모범 사례를 따르도록 하자.
    - 이를 통해 테스트가 통과하는 것을 막고, 구현할 게 아직 남아있음을 잊지 않을 수 있다.

@Before와 @After

```
- @Before @After     @Test      .
- @Test     .
-      ,            .
-    @Before @After    ,         .

- @BeforeClass @AfterClass  .
-     @Test         .
- @BeforeClass @AfterClass       ,     .

- @Before/@After, @BeforeClass/@AfterClass    public ,
- @BeforeClass @AfterClass  public  static .
```

@Test

```
 -      .
 - assert     .
 -  @Test    .
 -       .
 -    .
 -     .
 -       .

 - @Test(expected=SomeExcception.class) :     ,    ,      .
 - @Test(timeout=130) :        ,       .
 - @Ignore(value="   ") :        .
```

햄크레스트 (Hamcrest)

- 수 많은 유용한 매처(matcher) 객체(제약이나 술어라 불린다)를 제공하며, 다양한 언어로 포팅되어 있다.
- 테스트 프레임워크는 아니고, 그 보다는 간단한 매칭 규칙을 선언적으로 정의할 때 큰 도움이 된다.

```
public class HamcrestTest {
        private List<String> values;

        @Before
        public void setUpList() {
                values = new ArrayList<String>();
                values.add("x");
                values.add("y");
                values.add("z");
        }

        @Test
        public void testWithoutHamcrest() {
                assertTrue(values.contains("one") || values.contains("two") || values.contains("three"));
        }
}
```

```
import static org.junit.Assert.assertThat;
import static org.hamcrest.CoreMatchers.anyOf;
import static org.hamcrest.Corematchers.equalTo;
import static org.junit.JUnitMatchers.hasItem;

public class HamcrestTest {
        private List<String> values;

        @Before
        public void setUpList() {
                values = new ArrayList<String>();
                values.add("x");
                values.add("y");
                values.add("z");
        }

        @Test
        public void testWithHamcrest() {
                assertTrue(values, hasItem(anyOf(equalTo("one"), equalTo("two"), equalTo("three"))));
        }
}
```

- 매처의 강력한 기능 중 하나가 서로 다른 매처를 조합해 사용하는 능력이다.
- assert 코드에 Hamcrest 매처를 사용할지 말지는 순전히 개인 취향이다.
- 표준 assert 매커니즘 대비 Hamcrest가 제공하는 독특한 장점은, assert가 실패했을 때 사람이 읽을 수 있는 형태의 설명을 제공한다는 것이다.

- 가장 널리 쓰이는 Hamcrest 매처

코어(core)	의미
anything	무엇이든 상관없이 모든 것을 가리킴. assert 문의 가독성을 높이고 싶을 때 유용하다.
is	문장 가독성 향상 목적으로만 사용된다.
allOf	포함한 모든 매처가 매칭되는지 검사한다(&& 연산자와 동일).
anyOf	포함한 매처 중 어느 하나라도 매치되는 것이 있는지 검사한다(
not	포함한 매처들의 의미를 부정한다(! 연산자와 동일)
instanceOf, isCompatibleType	객체들이 호환 가능한 타입인지 확인한다.
sameInstance	객체 신원을 확인한다.
notNullValue, nullValue	값이 null인지(혹은 null 아닌지) 검사한다.
hasProperty	자바빈이 특정 속성을 갖는지 검사한다.
hasEntry, hasKey, hasValue	주어진 Map이 명시된 entry, key, value를 포함하는지 검사한다.
hasItem, hasItems	주어진 컬렉션이 명시한 아이템, 혹은 아이템들을 포함하는지 검사한다.
closeTo, greaterThan, greaterThanOrEqual, lessThan, lessThanOrEquals	주어진 숫자가 또 다른 숫자에 근접한지, 더 큰지, 더 크거나 같은지, 더 작은지, 더 작거나 같은지 검사한다.
equalToIgnoringCase	주어진 문자열이 다른 문자열과 일치하는지 검사한다(대소문자 무시).
equalToIgnoringWhiteSpace	주어진 문자열이 다른 문자열과 일치하는지 검사한다(공백문자 무시).
containsString, endsWith, startWith	주어진 문자열이 다른 문자열을 포함하는지, 그 문자열로 시작하거나 끝나는지 검사한다.

단위 테스트가 필요한 이유

1. 기능 테스트보다 훨씬 높은 테스트 커버리지 달성이 가능하다.
2. 팀 생산성을 향상시킨다.
3. 회귀 테스트(regression test)를 수행하고 디버깅의 필요성을 줄여준다.
4. 리팩터링과 코드 수정 시 올바로 하고 있다는 확신을 준다.
5. 구현 품질을 향상시킨다.
6. 기대하는 행위를 문서화한다.
7. 코드 커버리지 등 각종 측정을 가능하게 한다.

테스트의 종류

1. 통합 테스트(integration test)
2. 기능 테스트(functional test)
3. 스트레스 테스트(stress test)와 부하 테스트(load test)
4. 인수 테스트(acceptance test)

단위 테스트의 세 가지 맛

1. 논리 단위 테스트 : 한 메서드에 집중한 테스트, 목 객체(Mock Object)나 스텁(Stub)을 이용해 테스트 메서드의 경계를 제어할 수 있다.
2. 통합 단위 테스트 : 실제 운영 환경(혹은 그 일부)에서 컴포넌트 간 연동에 치중한 테스트, 예를 들어 데이터베이스를 사용하는 코드라면 데이터 베이스를 효과적으로 호출하는가를 테스트할 수 있다.
3. 기능 단위 테스트 : 자극 반응(Stimulus Response)을 확인하기 위해 통합 단위 테스트의 경계를 확장한 테스트. 예를 들어 인증된 클라이언트만 접근 가능한 보안 웹 페이지를 가진 웹 애플리케이션을 가정해보자. 만약 로그인을 하지 않은 채 보안 페이지에 접근하면, 클라이언트를 로그인 페이지로 돌려보내야 한다. 이 상황을 검사하려면, 기능 단위 테스트(Functional Unit Test)는 이 페이지에 접속하려는 HTTP 요청을 보내고, 응답으로 재전송 코드(HTTP 상태 코드 302)가 오는지 확인하는 방법이 있다.

테스트 주도 개발 (Test-driven development, TDD)

- 자동화 테스트가 실패했을 때와 코드 중복을 제거하려 할 때에만 새로운 코드를 작성하도록 권하는 프로그래밍 실천법이다.
- TDD의 목표는 '작동하는 깨끗한 코드(Clean code that works)'이다.

1. 개발 주기 조정하기

    - 전통적인 개발 주기는 코딩, 테스트, (반복), 커밋 순
    - TDD 실천자들은 테스트, 코드, (반복), 커밋 순
    - 테스트가 설계를 이끌고 메서드의 첫 대상이 된다.

- 코드를 설계 > 동작 방식을 문서화 > 단위 테스트를 수행한다.
- 테스트 > 코드, 리팩터링, (반복), 커밋
- 지속적으로 회귀 테스트를 수행하라.

2. TDD 실천으로 가는 두 단계

   a. 새 코드를 작성하기 앞서 실패하는 자동화 테스트를 작성하라.
   b. 중복을 제거하라.

2. Unit Test 원칙

```
Unit Test  F.I.R.S.T
  -      5  .

  F - Fast :       .
  I - Independent/Isolated :    .  .
  R - Repeatable :   .
  S - Self Validating :    .
  T - Timely :    .
```

3. 무엇을 테스트 할 것인가?

```
Right-BICEP
  - [Right]-BICEP :  ?
  - Right-[B]ICEP :  (boundary conditions) ?
    - CORRRECT
      - [C]onformance :     ?
      - [O]rdering :      ?
      - [R]Range :     ?
      - [R]eference :          ?
      - [E]xistence :  ? ( (non-null), 0 (nonzero),   )?
      - [C]ardinality :    ?
      - [T]ime :    ?  ? ?
  - Right-B[I]CEP :  (inverse relationship)   ?
  - Right-BI[C]EP :     (cross-check)  ?
  - Right-BIC[E]P :  (error conditions)     ?
  - Right-BICE[P] :  (performance characteristics)  ?
```

II. 테스트 라이브러리

1. Mock Object 란?

다른 누군가로부터 휴대 전화 서비스(CellPhoneService) 기능을 제공 받아 이를 사용한 휴대 전화 문자 발신기(CellPhoneMmsSender)를 프로그래밍 한다고 생각해 보자.

이를 코드로 나타내면 아래와 같다.

```
public class CellPhoneMmsSender {
        private CellPhoneService cellPhoneService;


        public CellPhoneMmsSender(CellPhoneService cellPhoneService) {
                this.cellPhoneService = cellPhoneService;
        }


        public void send(String msg) {
                cellPhoneService.sendMMS(msg);
        }
}
```

CellphoneMmsSender의 send() 메소드에 대한 테스트 코드를 작성 하려면 어떻게 해야 할까?
먼저 반환값을 검증하는 것을 고려할 수 있을 것이다. 하지만 테스트 대상인 send() 메소드의 반환 타입은 void 이다. 반환 값이 아니라면 무엇을 검증해야 할까?
CellphoneMmsSender 테스트 관점에서 중요한 것은 실제 문자 메시지를 보내는 것이 아니다. 실제 문자 메시지를 보내는 것은 CellphoneService의 책임이다.
그렇다면 CellphoneMmsSender send() 메소드에서 검증해야 하는 것은 전달 받은 메시지(msg)를 CellphoneService sendMMS()의 파라메터로 호출 했는지 여부이다.
CellphoneService의 sendMMS 호출 여부를 테스트 하기 위해서는 CellphoneMmsSender가 참조하고 있는 CellphoneService 객체를 ***가짜(대역) 객체***로 대체하고 이를 검증하는 방법이 있다.
여기서의 사용하는 가짜 객체를 Mock Object 라고 한다.

Mock Object는 테스트 더블(Test Double) 중 하나이며, 위키피디아에서는 아래와 같이 정의한다.

```
In object-oriented programming, mock objects are simulated objects that mimic the behavior of real objects in
controlled ways.
A programmer typically creates a mock object to test the behavior of some other object,
in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior of a human in
vehicle impacts.
– https://en.wikipedia.org/wiki/Mock_object


'  (test the behavior of some other object)'   ?
      '  (msg) CellphoneService sendMMS()    '     .
  (Behavior Verification) .
```

Stub과 Mock Object는 무엇이 다른가?

```
There is a difference in that the stub uses state verification while the mock uses behavior verification.
 : http://martinfowler.com/articles/mocksArentStubs.html

    Mock Object  (behavior verification) , Stub  (state verification)    .
     ,           .

public interface MailService {
    public void send (Message msg);
}

public class MailServiceStub implements MailService {
    private List<Message> messages = new ArrayList<Message>();    public void send (Message msg) {
         messages.add(msg);
    }    public int numberSent() {
         return messages.size();
    }
}

class OrderStateTester...public void testOrderSendsMailIfUnfilled() {
   Order order = new Order(TALISKER, 51);
   MailServiceStub mailer = new MailServiceStub();
   order.setMailer(mailer);
   order.fill(warehouse);
   assertEquals(1, mailer.numberSent());
}

  (State) MailServiceStub messages .
 OrderStateTester MailServiceStub  (mailer.numberSent() –   ) (assertEquals) .
              .
```

2. Mockito

단위 테스트를 위한 Java Mocking Framework 이다.

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <scope>test</scope>
</dependency>
```

Mockito를 이용한 Mock 객체 생성

WriteArticleServiceImpl 클래스를 테스트하는 코드는 다음과 같을 것이다.

```
@Test
public void writeArticle() {
        WriteArticleServiceImpl writeArticleService = new WriteArticleServiceImpl();
        Article article = new Article();
        Article writtenArticle = writeArticleService.writeArticle(article);

        assertNotNull(writtenArticle);
        assertNotNull(writtenArticle.getId());
}
```

하지만 위 테스트를 실행하면 NullPointerException이 발생하는 데,

그 이유는 WriteArticleServiceImpl.writeArticle() 메서드에서 IdGenerator와 ArticleDao을 구현한 객체를 사용하기 때문이다.

따라서, WriteArticleServiceImpl 클래스를 테스트 하려면 IdGenerator와 ArticleDao를 가짜로 구현한 Mock 객체를 전달해 주어야 한다.

Mockito를 이용할 경우 이는 다음과 같이 Mockito.mock() 이라는 메서드를 이용해서 생성할 수 있다. 아래는 Mock 객체 생성 예이다.

```
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

import org.junit.Test;

public class WriteArticleServiceImplTest {

    @Test
    public void writeArticle() {
        // mock
        ArticleDao mockedDao = mock(ArticleDao.class);
        IdGenerator mockedGenerator = mock(IdGenerator.class);

        WriteArticleServiceImpl writeArticleService = new WriteArticleServiceImpl();
        writeArticleService.setArticleDao(mockedDao);
        writeArticleService.setIdGenerator(mockedGenerator);

        Article article = new Article();
        Article writtenArticle = writeArticleService.writeArticle(article);

        assertNotNull(writtenArticle);
    }
}
```

Mock 객체의 메서드 호출 검증하기

Mock 객체를 사용하는 이유는 테스트 하려는 클래스가 연관된 객체와 올바르게 협업하는 지를 테스트 하기 위함도 있다.

따라서, Mock 객체의 메서드가 올바르게 실행되는 지 확인해볼 필요가 있다.

Mock 객체의 특정 메서드가 호출되었는 지 확인하려면 Mockito.verify() 메서드와 Mock 객체의 메서드를 함께 사용하면 된다. 아래는 사용 예이다.

```
@Test
public void writeArticle() {
    // mock
    ArticleDao mockedDao = mock(ArticleDao.class);
    IdGenerator mockedGenerator = mock(IdGenerator.class);

    WriteArticleServiceImpl writeArticleService = new WriteArticleServiceImpl();
    writeArticleService.setArticleDao(mockedDao);
    writeArticleService.setIdGenerator(mockedGenerator);

    Article article = new Article();
    Article writtenArticle = writeArticleService.writeArticle(article);

    assertNotNull(writtenArticle);
    verify(mockedGenerator).getNextId();
    verify(mockedDao).insert(article);
}
```

위 코드에서 verify(mockedGenerator).getNextId() 메서드는 mockedGenerator 객체의 getNextId() 메서드가 호출되었는 지의 여부를 확인한다.

verify(mockedDao).insert(article) 메서드의 경우 mockedDao 객체의 insert() 메서드 호출 중에서 article 객체를 인자로 전달받는 호출이 있었는 지 여부를 확인한다.

일단 Mock 객체가 만들어지면 해당 Mock 객체는 메서드 호출을 모두 기억하기 때문에, 어떤 메서드 호출이든 검증할 수 있다.

원하는 값을 리턴하는 스텁 만들기

앞서 테스트에서 다음과 같이 writeArticleService.writeArticle(aritlce)이 리턴한 객체가 알맞은 ID 값을 갖는 지 확인하는 검증 코드를 넣었다고 하자.

```
Article writtenArticle = writeArticleService.writeArticle(article);
assertNotNull(writtenArticle);
assertNotNull(writtenArticle.getId()); //
```

위 코드에서 assertNotNull(writtenArticle.getId()) 메서드는 검증에 실패한다.

그 이유는 WriteArticleServiceImpl.writeArticle() 메서드가 IdGenerator.nextId() 메서드를 이용해서 ID 값을 가져온 뒤 aritIcle 객체에 저장하기 때문이다. (아래 코드 참조)

```
public Article writeArticle(Article article) {
    Integer id = idGenerator.getNextId();
    article.setId(id);
    articleDao.insert(article);
    return article;
}
```

테스트 코드에서는 IdGenerator로 Mock 객체를 전달했는데, Mockito.mock()을 이용해서 생성한 객체의 메서드는 리턴 타입이 객체인 경우 null을 리턴하고 기본 데이터 타입인 경우 기본 값을 리턴한다.

따라서, 리턴 타입이 Integer인 (즉, 객체인) IdGenerator.getNextId()에 대해서는 null을 리턴하고 따라서 assertNotNull(writtenArticle.getId()) 코드에서 writtenArticle.getId() 메서드가 null을 리턴하게 되어 검증에 실패하는 것이다.

Mockito는 Mock 객체의 메서드가 알맞은 값을 리턴하는 스텁을 만들 수 있는 기능을 제공하고 있다. 이 메서드는 when - then의 형식을 띄고 있는데, 아래 코드는 실제 사용 예를 보여주고 있다.

```
...
IdGenerator mockedGenerator = mock(IdGenerator.class);
when(mockedGenerator.getNextId()).thenReturn(new Integer(1));

WriteArticleServiceImpl writeArticleService = new WriteArticleServiceImpl();
writeArticleService.setIdGenerator(mockedGenerator);

Article article = new Article();
Article writtenArticle = writeArticleService.writeArticle(article);

assertNotNull(writtenArticle);
assertNotNull(writtenArticle.getId());
verify(mockedGenerator).getNextId();
```

Mockito.when() 메서드는 메서드 호출 조건을 그리고 thenReturn()은 그 조건을 충족할 때 리턴할 값을 지정한다. 위 코드의 경우 mockedGenerator.getNextId() 메서드가 호출되면 Integer(1)을 리턴하라는 의미를 갖는다.

Mock 객체의 메서드 호출 시 전달되는 인자 값에 따라서 리턴 값을 다르게 지정할 수도 있다. 아래 코드는 예를 보여주고 있다.

```
Article article = new Article();
when(mockedArticleDao.insert(article)).thenReturn(article));
```

Argument Matcher를 이용한 인자 매칭

보통, when()으로 스텁을 생성하거나 verify()로 메서드 호출 여부를 확인할 때는 특정한 값을 지정한다.

```
//   1
when(mockedListService.getArticles(1)).thenReturn(someList);
//   1  getArticles()
verify(mockedListService).getArticles(1);
```

하지만, 특정한 값이 아닌 임의의 값에 대해서 when() 메서드와 verify() 메서드를 실행하고 싶을 때가 있다.

이런 경우에는 Argument Matcher를 이용해서 인자 값을 지정하면 된다.

예를 들어, 임의의 정수 값을 인자로 전달받은 메서드 호출을 when()과 verify()에서 표현하고 싶다면 다음과 같이 Matchers.anyInt() 메서드를 사용하면 된다.

```
when(mockedListService.getArticles(anyInt())).thenReturn(someList);
...
verify(mockedListService).getArticles(anyInt());
```

Matchers 클래스는 anyInt() 뿐만 아니라 anyString(), anyDouble(), anyLong(), anyList(), anyMap() 등의 메서드를 제공하는데,

이들 메서드에 대한 자세한 내용은 http://mockito.googlecode.com/svn/branches/1.6/javadoc/org/mockito/Matchers.html 사이트를 참고하기 바란다.

인자 중 한가지라도 Argument Matcher를 사용하면 나머지 인자에 대해서도 Matcher를 사용해야 한다. 예를 들어, 아래 코드는 예외를 발생한다.

```
Authenticator authenticator = mock(Authenticator.class);
when(authenticator.authenticate(anyString(), "password")).thenReturn(authObj);
```

만약 여러 인자 중 특정 값을 명시해야 하는 경우가 필요하다면 eq() Matcher를 사용하면 된다. 아래는 위 코드를 eq()를 이용해서 수정한 코드를 보여주고 있다.

```
Authenticator authenticator = mock(Authenticator.class);
when(authenticator.authenticate(anyString(), eq("password"))).thenReturn(authObj);
```

Mockito 클래스는 Matchers 클래스를 상속받고 있기 때문에 Mockito 클래스의 static 메서드를 static import 하면 Matchers 클래스에 정의된 메서드를 사용할 수 있다.

thenThrow()를 이용한 예외 발생

Mock 객체의 메서드 호출시 예외를 발생시키고 싶을 때가 있는데, 이런 경우에는 thenThrow() 메서드를 사용하면 된다. 아래는 사용 예를 보여주고 있다.

```
when(mockedDao.insert(article)).thenThrow(new RuntimeException("invalid title"));
```

thenThrow() 메서드에서 발생시킬 예외 객체를 전달해주면, when()에서 지정한 조건의 메서드가 호출될 때 예외를 발생시킨다.

메서드 호출 회수 검사

메서드가 지정한 회수 만큼 호출되었는 지의 여부를 확인하려면 times() 메서드를 사용하면 된다. 예를 들어, Mock 객체의 특정 메서드가 3번 호출되었는 지 확인하려면 다음과 같이 verify() 메서드의 두 번째 인자에 times() 메서드를 (정확히는 times() 메서드의 리턴 값을) 전달해주면 된다.

```
verify(mockedAuthenticator, times(3)).authenticate(anyString(), anyString());
```

호출 회수를 따로 지정하지 않을 경우 times(1)이 기본 값이 된다. times() 외에 다음과 같은 메서드를 사용할 수 있다.

- times(int) - 지정한 회수 만큼 호출되었는 지 검증
- never() - 호출되지 않았는지 여부 검증
- atLeastOnce() - 최소한 한번은 호출되었는 지 검증
- atLeast(int) - 최소한 지정한 회수 만큼 호출되었는 지 검증
- atMost(int) - 최대 지정한 회수 만큼 호출되었는 지 검증

다수의 Mock 객체들이 사용되지 않은 것을 검증하고 싶은 경우에는 verifyZeroInteractions(Object ... mocks) 메서드를 사용하면 된다. 아래는 사용 예이다.

```
verifyZeroInteractions(mockedOne, mockedTwo, mockedThree);
```

Answer를 이용한 메서드 구현

Mock 객체를 사용하다보면 직접 Mock의 동작 방식을 구현해 주고 싶을 때가 있다. (사실, 필자가 개인적으로 굳이 간단한 Mock 라이브러릴 만든 이유도 이것 때문이었다.) 이런 경우 thenAnswer() 메서드와 Answer 인터페이스를 사용하면 된다. 아래 코드는 사용 예이다.

```
when(mockedGenerator.getNextId()).thenAnswer(new Answer<Integer>() {
    private int nextId = 0;
    public Integer answer(InvocationOnMock invocation) throws Throwable {
        return new Integer(++nextId);
    }
});
```

위와 같이 Answer를 사용하면, mockedGenerator의 getNextId() 메서드를 호출할 때 마다 answer() 메서드가 호출된다. 위 코드의 경우 getnextId() 메서드가 호출될 때 마다 1씩 증가된 값을 리턴하는 Anwser 구현 클래스를 리턴하였다.

만약 파라미터로 전달되는 값을 사용하고 싶다면 answer() 메서드에 전달된 InvocationOnMock을 이용하면 된다. 아래 코드는 사용 예이다.

```
when(authenticator.authenticate(anyString(), anyString())).thenAnswer(new Answer<Object> (){
    public Object answer(InvocationOnMock invocation) throws Throwable {
        Object[] arguments = invocation.getArguments();
        String userId = (String) arguments[0];
        String password = (String) arguments[1];
        Object authObject = null;
        // ...
        return authObject;
    }
});
```

@Mock 어노테이션을 이용한 코드 단순화

Mockito.mock() 메서드를 이용해서 Mock 객체를 생성하는 코드가 다소 성가시게 느껴진다면, @Mock 어노테이션을 이용해서 Mock 객체를 생성할 수 있다. 예를 들어, 아래 코드와 같이 테스트 클래스의 멤버 필드에 @Mock 어노테이션을 적용하면 해당 타입에 대한 Mock 객체가 할당된다.

@Mock 어노테이션이 동작하려면 테스트가 실행되기 전에 @Mock 어노테이션이 적용된 필드에 Mock 객체를 할당하도록 해 주어야 한다. JUnit 4 버전의 경우 @RunWith 어노테이션에서 MockitoJUnit44Runner.class를 값으로 지정해주면 된다.

```
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;

@RunWith(MockitoJUnit44Runner.class)
public class WriteArticleServiceImplTest {

    @Mock Authenticator authenticator;
    @Mock ArticleDao mockedDao;
    @Mock IdGenerator mockedGenerator;

    @Test
    public void setup() {
        when(authenticator.authenticate(anyString(), eq("password"))).thenReturn(null);
        ...
    }

}
```

또는 테스트가 실행되기 전에 명시적으로 MockitoAnnotations.initMocks(this) 메서드를 호출해주면 된다.

```
public class WriteArticleServiceImplTest {

    @Mock Authenticator authenticator;
    @Mock ArticleDao mockedDao;
    @Mock IdGenerator mockedGenerator;

    @Before
    public void test() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void writeArticle() {
        when(authenticator.authenticate(anyString(), eq("password"))).thenReturn(null);
    }
}
```

III. 테스트 작성

1. Spring 설정 참조가 필요 없는 클래스의 단위 테스트

```
# MockitoJUnitRunner  : Spring
#    spring annotation     Mock annotation
#    InjectMocks annotation
# setUp  MockitoAnnotations.initMocks(this);
```

```
@RunWith(MockitoJUnitRunner.class)
public class TestTargetServiceTest {
    @Mock
    private MockService mockService;

    @InjectMocks
    private TestTargetService testTargetService = new TestTargetServiceImpl();

    @Before
    public void setUp() throws Exception {
        MockitoAnnotations.initMocks(this);
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void test() {
    }
}
```

2. Spring 설정 참조가 필요한 클래스의 단위 테스트

```
# SpringJUnit4ClassRunner  : Spring
#     .           .
#   Spring context   (test-resource-rootpath/package/TestTargetServiceTest-context.xml)
#
#    spring annotation     Mock annotation
#    Autowired InjectMocks annotation
#  InjectMocks
# setUp  MockitoAnnotations.initMocks(this);
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class TestTargetServiceTest {
    @Mock
    private MockService mockService;

    @Autowired
    @InjectMocks
    private TestTargetService testTargetService;

    @Before
    public void setUp() throws Exception {
        MockitoAnnotations.initMocks(this);
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void test() {
    }
}
```

3. 통합 테스트

```
# SpringJUnit4ClassRunner  : Spring
# 2  Mock
#   Spring context   (test-resource-rootpath/package/TestTargetServiceTest-context.xml)
#
#    Autowired annotation
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class TestTargetServiceTest {
    @Autowired
    private TestTargetService testTargetService;

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void test() {
    }
}
```

4. Static 메소드 mocking 테스트

```
# PowerMockRunner
# Static    @PrepareForTest annotation
#    spring annotation     Mock annotation
#    InjectMocks annotation
# setUp  MockitoAnnotations.initMocks(this); PowerMockito.mockStatic(TestTargetStaticMethodClass.class);
```

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(TestTargetStaticMethodClass.class)
public class TestTargetServiceTest {
    @Mock
    private MockService mockService;

    @InjectMocks
    private TestTargetService testTargetService = new TestTargetServiceImpl();

    @Before
    public void setUp() throws Exception {
        MockitoAnnotations.initMocks(this);
        PowerMockito.mockStatic(TestTargetStaticMethodClass.class);
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void test() {
    }
}
```

5. Controller 단위 테스트

```
@RunWith(MockitoJUnitRunner.class)
public class TestTargetControllerTest {
    @Mock
    private MockService mockService;

    @InjectMocks
    private TestTargetController testTargetController;
    private MockMvc mockMvc;

    @Before
    public void setUp() throws Exception {
        MockitoAnnotations.initMocks(this);
        mockMvc = MockMvcBuilders.standaloneSetup(testTargetController).build();
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void test() {
    }
}
```

6. Controller 통합테스트

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@TransactionConfiguration(transactionManager = "transactionManager", defaultRollback = false)
@Transactional
@WebAppConfiguration
public class TestTargetControllerTest {
    @Autowired
    private WebApplicationContext webApplicationContext;

    private MockMvc mockMvc;

    @Before
    public void setUp() throws Exception {
        mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void test() {
    }
}
```

IV. Jenkins 연동