



성공은 열정을 잃지 않고 실패를 거듭하는
과정 속에 나온다.

- 원스틴 처칠

JUnit과 Hamcrest

앞서 진행한 예제에선 JUnit 테스트 프레임워크에 대한 자세한 설명 없이 곧바로 사용했었다. 이제부터는 JUnit에 대해 좀 더 자세히 살펴볼까 한다. JUnit은 Java 언어에서 TDD나 단위 테스트를 이야기할 때 빼놓고 이야기하기 어려울 정도로 절대적인 위치를 차지하고 있는 단위 테스트 프레임워크다. 비록 깊숙한 내용까지는 모르더라도 한 번은 꼭 배워둘 필요가 있다. 그리고 테스트 코드 내에서는 assertEquals 같은 비교 문장이 종종 사용되는데, 등호, 부등호 비교 수준의 간략한 형태의 비교에는 한계가 있다. 또한 부등호 비교나 객체 안의 값 찾기, 문자열 안의 특정 값 찾기 등의 좀 더 다양한 판단조건이 필요한 경우에는, 자칫 문장이 복잡해져서 판단조건이 무엇인지 혼란스러워지기 쉽다. 이럴 때 Hamcrest라는 라이브러리를 함께 사용하면, 좀 더 이해하기 쉽고 우아한 비교 문장을 만들 수 있다. JUnit을 사용해왔던 사람들조차 Hamcrest에 대해서는 잘 모르거나 낯설게 느껴질 수 있는데, 현재 Hamcrest 라이브러리는 JUnit 4 버전에 포함되어 함께 배포되고 있다. Hamcrest는 무엇이고, 어떤 식으로 비교 문장을 좀 더 자연스럽게 만들어주며, JUnit과는 어떤 식으로 함께 사용하게 되는지에 대해서도 이번 장에서 살펴볼 예정이다. 자, 그럼 JUnit부터 살펴보기로 하자.

2장 체크리스트

- ☐ 테스트 픽처의 개념
- ☐ JUnit 3
 - 단정문
 - 테스트 스위트

☐ JUnit 4

- @Test
- @BeforeClass, @AfterClass, @Before, @After
- 예외처리 테스트
- 시간 제한 테스트
- @RunWith
- @SuiteClasses

고급 기능 소개

- 파라미터화된 테스트
- Rule
- Theory

☐ Hamcrest

2.1 JUnit

에릭 감마와 켄트 벡이 탄생시킨 JUnit은 현재 전 세계적으로 가장 널리 사용되는 Java 단위 테스트 프레임워크다. TDD의 근간이 되는 프레임워크이며, 소위 xUnit 시리즈¹라고 불리는 다양한 단위 테스트 프레임워크들의 기원이 되는 프레임워크다. 전체 소스가 공개되어 있는 오픈소스 프로젝트² 방식으로 개발되며 지금도 꾸준히 버전업을 하고 있다. JUnit은 제공하는 기능도 기능이지만, 만들어진 방식이나 구조 자체에서도

1 JUnit은 다양한 언어로 포팅됐는데, 포팅된 단위 테스트 프레임워크는 실행구조가 유사하며, 일반적으로 Unit이라는 단어 앞에 프로그래밍 언어의 이름 일부를 붙이고 있다. Java는 JUnit, C는 CUnit, C++는 CppUnit, Python은 PyUnit, 닷넷은 NUnit 같은 식으로 말이다. 이런 프레임워크들을 통칭 xUnit 프레임워크라고 한다.

2 JUnit은 CPL(Commons Public License) 1.0을 따른다. CPL 라이선스 제품은, 해당 제품 혹은 제품의 일부를 이용해 개인 소프트웨어/상용 소프트웨어 구분 없이 만들 수 있다. 라이선스 관련해서는 <http://www.codeproject.com/info/Licenses.aspx>에 종류별로 자세히 설명되어 있으니 참고하자.

배울 점이 매우 많은 소프트웨어다. JUnit은 단위 테스트를 수행하는 데 있어 기본적으로 다음과 같은 기능을 제공한다.

- 테스트 결과가 예상과 같은지를 판별해주는 단정문(assertions)
- 여러 테스트에서 공용으로 사용할 수 있는 테스트 픽스처(test fixture)
- 테스트 작업을 수행할 수 있게 해주는 테스트 러너(test runner)

JUnit이 널리 퍼진 이유 중 하나는 위에서 보는 것과 같이 목표가 단순하고, 하고자 하는 일이 명료했기 때문이다. 거기에 쉬운 사용 방법도 크게 한몫 거들었다(물론 최신 버전은 머리에 스템이 살살 올라오는 다소 복잡한 기능들을 포함하고 있지만 말이다).

JUnit을 비롯하여 단위 테스트 케이스를 만드는 데 있어 꼭 알아둘 개념이 하나 있는데, 테스트 픽스처(테스트 기반환경 또는 테스트를 위한 구조물)라는 개념이다.

테스트 픽스처

일반적으로 소프트웨어 테스트에서 이야기하는 ‘테스트 픽스처’란 테스트를 반복적으로 수행할 수 있게 도와주고 매번 동일한 결과를 얻을 수 있게 도와주는 ‘기반이 되는 상태나 환경’을 의미한다. 일관된 테스트 실행환경이라고도 하며, 때로는 테스트 컨텍스트(test context)라 부르기도 한다. 테스트 케이스에서 사용할 객체의 인스턴스를 만든다든가, 데이터베이스와 연동할 수 있는 참조를 선언한다든가, 파일이나 네트워크 등의 자원을 만들어 지정한다든가 하는 등의 작업 혹은 그 작업의 결과로 만들어진 대상을 통칭한다. 1장 예제에서 사용한 setUp이나 tearDown 메소드는 테스트 픽스처를 만들고, 정리하는 작업을 수행하는 메소드인데, 이런 메소드를 테스트 픽스처 메소드(test fixture method)라고 한다.

테스트 케이스와 테스트 메소드

테스트 케이스(test case)와 테스트 메소드(test method)라는 용어는 흔히 혼용되어 사용된다. 정확히는 단위 테스트 케이스와 단위 테스트 메소드가 제대로 된 명칭이다.

테스트 케이스는 테스트 작업에 대한 시나리오적인 의미가 더 강하고 테스트 메소드는 JUnit의 메소드를 지칭한다. 그러나 테스트 케이스가 곧 테스트 메소드인 경우가 많기 때문에 두 단어는 종종 혼용되어 사용된다. 앞으로 동일한 뜻으로 함께 사용하더라도 크게 혼란스러워하지 않길 바란다.

저자
한·TDD

JUnit의 기원

켄트 벡과 에릭 감마의 이름을 어느 정도 들어봤는지 모르겠다. 두 사람 다 이 분야에선 매우 유명한 거물에 속한다. 한 사람은 IT 세계를 크게 움직인 XP와 TDD의 창시자이고, 다른 한 사람은 역사상 가장 크게 흥행한 IT 서적이라 불리는 『GoF의 디자인 패턴』의 저자 네 명 중 한 명이다. 이 두 사람에 의해 만들어진 JUnit의 기원은 이러하다.

시절은 바야흐로 1997년으로 돌아간다. 그 당시 켄트 벡은 CSLife라는 이름 모를 회사에서 Smalltalk 전문가로 일하고 있었고, 에릭은 OTI라는 IBM 산하 연구소에서 일하는 프로그래머였다. 둘은 강연 때문에 애틀랜타행 비행기를 함께 타게 되는데, Java를 써본 적이 없는 켄트 벡은 에릭에게 Java를 가르쳐달라고 했고, 마침 켄트 벡의 SUnit(스몰토크 테스트 프레임워크)에 평소 관심이 있었던 에릭은 SUnit의 Java 버전을 함께 만들어보기로 한다. 그래서 세 시간 남짓한 시간 동안 비행기 안에서 노트북으로 함께 코딩을 하게 됐고, 그때 나온 프로그램이 JUnit 테스트 프레임워크의 모태가 된다. 만일 JUnit의 기원을 좀 더 자세히 알아보고 싶다면, <http://members.pingnet.ch/gamma/>로 찾아가 보기 바란다. JUnit 1.0의 소스를 다운로드 받을 수 있고, 켄트 벡과 에릭이 쓴 'Test Infected'라는 TDD 전파의 신호를 알리는 기념비적인 TDD 소개 글도 볼 수 있다. 참고로 2010년 현재 켄트 벡은 Three Rivers Institute라는 개인 회사를 차려 부인과 함께 컨설팅 활동을 하고 있고, 에릭은 IBM에서 Distinguished Engineer라는 직함으로 여전히 엔지니어 생활을 하고 있다. 그런데 순수 엔지니어인 에릭과 달리 켄트 벡은, 오리전에 있는 자신의 농장에서 표범과 맹수인 쿠키(Cougar)와 싸우고 염소 젖을 짜서 치즈를 만드는 일을 컨설팅 업무와 겸업으로 하고 있다. 아.. 정말이다.

Test Infected: Programmers Love Writing Tests



Kent Beck, CSLife
Erich Gamma, OTI Zürich

Testing is not closely integrated with development. This prevents you from measuring the progress of development- you can't tell when something starts working or when something stops working. Using *JUnit* you can cheaply and incrementally build a test suite that will help you measure your progress, spot unintended side effects, and focus your development efforts.

Contents

- [The Problem](#)
- [Example](#)
- [Cookbook](#)
 - [Simple Test Case](#)
 - [Fixture](#)
 - [Test Case](#)
 - [Suite](#)
 - [TestRunner](#)
- [Testing Practices](#)
- [Conclusions](#)

The Problem

1997년에 두 사람이 JUnit 1.0을 만들면서 작성한 글, '테스트에 중독되다: 프로그래머들은 테스트 작성하는 걸 사랑한다.'³

2.1.1 JUnit 3

원로급 Java 개발자가 아니라면, 대부분 처음 접한 JUnit은 버전 3(이하 JUnit 3)일 것이다.⁴ 지금은 JUnit 4가 주로 사용되고 있지만, 아직 JUnit 3을 사용하고 있는 프로젝트들도 상당수 남아 있다. JUnit은 junit.org 사이트에서 파일을 내려받아서 클래스 경로 내에 junit.jar 파일을 포함시켜 놓으면 바로 사용할 수 있다. 설치가 쉽고 몇 분 정도만 시간을 들여 설명을 읽으면 대부분의 기능을 무리 없이 사용할 수 있을 정도로 사용법이 간단하다.

3 이 글은 다음과 같은 문장으로 시작한다. "테스트하는 일과 개발은 밀접하게 통합되어 있지가 않다. 그렇기 때문에 개발 진척을 측정하기가 어렵다. (누가 물어보더라도) 일이 언제 시작해서 언제 끝날지 말할 수 없을 것이다. JUnit을 사용하면 적은 비용으로, 그리고 점차 (케이스가) 증가되는 형태로 테스트 집합을 만들어낼 수 있다. 그리고 그 테스트 집합은 개발 진척이 어떻게 되는지를 알려줄 수 있다. 또한 JUnit을 사용하면 의도하지 않았던 부작용을 발견해낼 수 있고, 개발업무에 좀 더 집중할 수 있다."

4 JUnit 1은 97년에 만들어져서 98년에 발표됐고, JUnit 3 버전은 99년 11월에 공개된 이후로 지금까지 사용되고 있다.

JUnit 3 프레임워크는 테스트 케이스를 작성하는 데 있어, 크게 두 개의 규칙과 네 개의 구성요소를 갖는다.

JUnit 3 규칙

규칙 1 TestCase를 상속받는다.

예: AccountTest extends TestCase

규칙 2 테스트 메소드의 이름은 반드시 test로 시작해야 한다.

예: testGetAddress(), testCalculateValue() 등

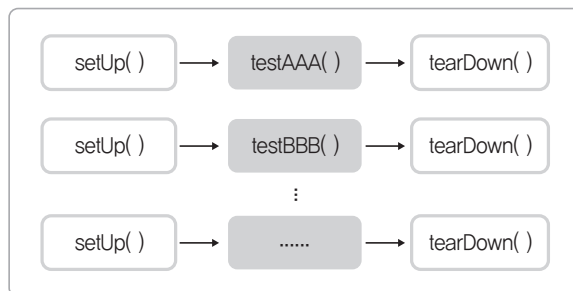
예

```
public class CalculatorTest extends TestCase {  
  
    public void testCalculateSum() {  
        ...  
    }  
}
```

JUnit 3 구성요소 1 **테스트 픽스처 메소드** Test Fixture Method

- setUp()
- tearDown()

TestClassA



테스트 클래스의 메소드 실행 순서

JUnit 3에서는 setUp과 tearDown이라는 두 개의 테스트 픽스처 메소드를 지원하고 있다. setUp은 각각의 테스트 메소드가 실행되기 전에 공통으로 호출되는 메소드다. 보통 테스트 환경 준비에 해당하는, 자원 할당, 객체 생성, DB 연결이나 네트워크 연결 등의 작업이 이뤄진다. tearDown은 테스트 메소드가 수행된 다음 수행되는 메소드다. setUp과는 반대로 자원 해체, 연결 해제, 객체 초기화 등의 뒷정리 작업을 하게 된다. setUp과 tearDown을 사용할 때는 대소문자에 유의하자. 오버라이드된 형태로 실행되기 때문에 이름이 다르면 실행이 안 된다. (너무 당연한 건가?) JDK 1.5 이상을 사용한다면, @Override 애노테이션을 붙여서 컴파일러가 강제로 체크할 수 있도록 유도할 수도 있다.

```
import junit.framework.TestCase;

public class DaoTest extends TestCase {

    Connection connection;

    protected void setUp() throws Exception {
        connection = Connection.getConnection();
    }
    public void testA() throws Exception { ... }
    public void testB() throws Exception { ... }
    ...
    protected void tearDown() throws Exception {
        account = Connection.releaseConnection();
    }
}
```

위 코드는 setUp → testA → tearDown, setUp → testB → tearDown 식으로 실행된다.

JUnit 3 구성요소 2 단정문 Assertions

대표적인 단정문

- assertEquals([message], expected, actual)
- assertTrue([message], expected) / assertFalse([message], expected)

- assertNull([message], expected) / assertNotNull([message], expected)
- fail([message])

* [message]는 선택사항임

JUnit은 테스트 케이스의 수행 결과를 판별해주는 다양한 단정문(무언가를 딱 잘라 한마디로 판단하는 짧은 문장)을 제공한다. 일반적으로 ‘단언하다’는 뜻의 assert라는 단어로 시작하며 ‘두 값 비교, 참/거짓, null 여부’ 등을 판별한다. 두 값의 일치 여부를 판단하는 assertEquals를 그중에서 제일 많이 사용한다.

assertEquals([message], expected, actual)

두 값이 같은지 비교하는 단정문으로, 예상에 해당하는 expected와 실제 테스트 수행 결과에 해당하는 actual이 서로 일치하는지 비교 판단한다. expect와 actual은 Java 언어의 기본 타입(primitive type) 전체에 대해 중첩구현(overloading)되어 있기 때문에 다양한 값을 서로 비교할 수 있다. 만일 비교 값이 기본 타입이 아닐 경우에는 equals 메소드 비교를 하게 되어 있다. 따라서 equals를 구현해놓은 클래스라면, 해당 equals 메소드를 호출해서 비교에 사용한다. 다음은 JUnit에서 assertEquals를 정의한 부분이다. 두 값을 비교하기 위해 어떤 방식을 취하고 있는지 살짝 살펴보자.

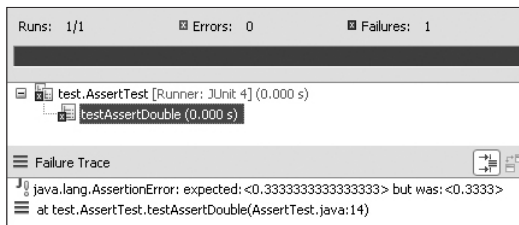
```
static public void assertEquals(String message, Object expected, Object
    actual) {
    if (expected == null && actual == null)
        return;
    if (expected != null && expected.equals(actual))
        return;
    failNotEquals(message, expected, actual);
}
```

둘 다 null이면 통과! expected가 null이 아니고 expected의 equals 메소드를 이용해 비교했을 때 같으면 또 통과! 그리고 위 두 경우가 아니면 실패(fail)로 처리된다.

assertEquals 시리즈 중 모양이 다소 다른 것이 있는데, 바로 `assertEquals([message], double expected, double actual, double delta)` 메소드다. 메소드 인자(argument) 마지막에 delta라는 항목이 하나 더 있다. 눈치 빠른 독자라면 바로 ‘아하!’ 할 텐데, 소수점을 갖는 float나 double 데이터형의 경우에는 정확하게 일치하는 값을 찾기 어려울 수 있다. 그럴 경우 delta라는 오차 보정 값을 이용해 적절한 오차 범위 내의 값은 동일한 값으로 판단할 수 있게 해준다.

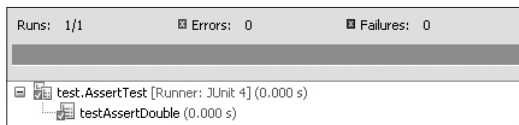
```
| assertEquals(0.3333, 1/3d, 0.00001);
```

위 경우는 1을 3으로 나눈 double 타입의 예상값으로 0.3333을 지정했다. 소수 다섯째 자리에서 1 이하의 오차를 갖는다면 일치하는 것으로 간주한다.



오차 delta 허용치 이상으로 값이 차이가 나면 테스트가 실패한다. 이럴 경우 delta 값의 자릿수를 아래와 같이 높이면, 테스트가 성공한다.

```
| assertEquals(0.3333, 1/3d, 0.00001);
```



JUnit 구 버전에는 double 타입을 비교할 때 오차 값을 지정하지 않는 `assertEquals(double expected, double actual)`이라는 메소드가 있었는데 지금은 권장하지 않음(deprecated)으로 처리되어 사용하지 않는다. 그리고 앞에서 assertEquals의 다양

한 타입을 중첩구현(overloading)해놓았다고 이야기했었는데, float 타입끼리의 비교는 제공하지 않는다. 소수를 표현하는 float와 double은 서로 호환되는 타입인데다, double 쪽이 훨씬 더 큰 숫자와 높은 정밀도를 사용할 수 있기 때문이다. 참고로 float는 4바이트의 저장영역에 2^{23} 의 정밀도를 갖고, double은 8바이트 저장소에 2^{52} 의 정밀도 영역을 갖는다.

assertSame([message], expected, actual)

assertNotSame([message], expected, actual)

assertSame은 두 객체가 정말 동일한 객체인지 주소값으로 비교하는 단정문이다. 따라서 객체를 비교할 때 equals 메소드를 사용하지 않고 바로 등가비교(==)를 한다. assertNotSame도 마찬가지로 두 객체를 주소로 비교한다. 다만 이 경우 주소 값이 다르면 무조건 true가 된다. 어! 그런데 assertSame 계열 단정문은 앞에서 박스로 된 소개에 없었는데? 이거 혹시 오타? 그건 아니고 같이 쓰면 복잡해 보일 것 같아서 빼왔다. ‘에엣? 이건 뭘 풀 뜯어 먹는 소리인 거냐?’라고 할 수도 있는데, 이쯤 되면 박스에 써놓을 때보다는 더 기억에 남지 않겠는가? :)

assertSame은 객체의 주소를 비교하는 것이다 보니, 주로 동일 객체임을 증명하는 데 쓰인다. 이를테면 캐시(cache) 기능을 만들었는데, 해당 캐시가 제대로 동작하는지 판단할 필요가 있다고 가정해보자. 이럴 때에 assertSame을 사용해서 특정 객체가 캐시에서 가져온 객체와 동일한지 여부를 판단할 수 있다.

```
cache.add(someObject, KEY);
assertSame("캐시처리 실패!", someObject, cache.lookup(KEY));
```

비슷한 방식으로, 싱글톤(Singleton)⁵으로 만들어진 객체를 비교할 때 쓰이기도 한다.

assertSame의 실제 구현은 다음과 같다.

5 싱글톤(Singleton): 디자인 패턴에서 나온 개념으로, 특정 클래스의 인스턴스가 오직 하나만 생성될 수 있게 만들어주는 패턴이다. 이때 static으로 지정된 getInstance() 같은 메소드를 통해서만 객체에 접근 가능하게 만든다. 따라서 몇 번을 호출해도 동일한 객체가 지속적으로 반환되거나 이용된다. 일반적으로 객체 생성과 소멸에 비용이 많은 드는 객체를 싱글톤으로 만들어놓아 효율을 높인다.

```
static public void assertSame(String message, Object expected, Object
actual) {
    if (expected == actual)
        return;
    failNotSame(message, expected, actual);
}
```

assertTrue([message], expected)

assertFalse([message], expected)

예상값의 참/거짓을 판별하는 단정문이다. boolean 타입 메소드를 이용할 경우나, 부등호 비교, 범위 비교 등을 판단할 때 사용한다. 가끔 assertTrue(account.getBalance() == 0) 같은 식으로 등가비교를 하는 경우가 있는데, 이때 단정문이 실패할 경우 account.getBalance()의 값을 바로 알 수 없어 불편하다. 등호 비교는 가급적 assertEquals(0, account.getBalance()) 형식을 사용하도록 권장한다.

assertNull([message], expected)

assertNotNull([message], expected)

대상 값의 null 여부를 판단하는 단정문이다. 자, 아래 테스트 메소드는 성공일까, 실패일까?

```
public void testAccount(){
    account = null;
    assertNull(account);
}
```

assertNull(account)가 성공해서 녹색 불이 들어오는 경우는, account가 null일 때이다. 따라서 답은 성공, 녹색 램프다! 별건 아니지만, 처음 사용할 때 자칫 헷갈릴 수 있으므로 유의하자. assert 시리즈는 예상값이 assert 문장 다음에 이어지는 글자와 상태가 일치한다는 걸 확인하는 문장이다. assert를 ‘~이어야 함!’으로 해석하면 헷갈리지 않을 수 있다.

fail([message])

이 메소드가 호출되면 해당 테스트 케이스는 그 즉시 실패한다. 앞에서 테스트 케이스의 성공/실패 조건에서도 설명했지만, 현재 작성 중인 메소드의 경우 단정문을 쓰지 않았으면 예외가 발생하지 않는 이상 무조건 성공하는 테스트 케이스가 된다.

```
public void testAccumulatedCharge()throws Exception {
    Charge charge = new Charge(this.year);
    chage.add(additionalFee);
    // 이하 계산 로직 미 작성 상태
}
```

위와 같이 단정문 작성을 누락한 경우 테스트를 수행하면 녹색 막대가 나온다. 이렇게 되면 자칫 수많은 테스트 케이스 속에 살짝 묻혀서 테스트가 성공적으로 수행됐다고 오해하고 넘어갈 수 있다. 만일 아직 테스트 케이스를 작성 중인데 완료하지 못한 구현을 중단해야 하는 경우라면 끝 부분에 fail()을 추가해놓으면 도움이 된다. 나중에 다시 돌아와야 할 때 어디부터 해야 할지를 알려주는 잣대가 되기 때문이다. 필자는 이 방식을 TDD 기본 가이드 중 하나로 삼고 있다.

JUnit 3에서는 위와 같은 경우 말고도, 예외처리를 테스트하기 위한 트릭으로 fail()을 사용하기도 한다.

```
public void testWithdraw_현재잔고이하_인출요구시() throws Exception {
    Account account = new Account(10000);
    account.withdraw(20000); // OverWithdrawRequestException이 발생해야 함!
}
```

이렇게면 위와 같은 경우처럼 특정 조건에서 예외가 발생해야 정상인 경우를 테스트 케이스로 작성하려면 어떻게 해야 할까? 해답은 다음과 같다.

```
public void testWithdraw_현재잔고이하_인출요구시() throws Exception {
    Account account = new Account(10000);
    try {
        account.withdraw(20000); // ❶
        fail(); // ❷
    }
```

```

    } catch (OverWithdrawRequestException e){
        assertTrue(true); // ❸
    }
}

```

- ❶ OverWithdrawRequestException이 발생해야 함!
- ❷ 만일 위에서 예외가 발생하지 않아서 이 부분까지 실행되면 실패함
- ❸ 빈 줄로 남겨둬도 무방하나 명시적으로 표시해놓음

결과적으로 try 구문에서 예외가 발생하지 않으면 실패로 간주하게 된다.

지금까지 JUnit의 기본적인 단정문들을 살펴보았다.

자, 단정문을 몇 개 정도 기억하는지 테스트해보고 넘어가자. 오른쪽 칸에 단정문 메소드 이름을 적어보자.

설명	이름
두 값이 같은지 비교하는 단정문은?	
두 객체가 동일한 객체인지 비교하는 단정문은?	
예상값의 참/거짓을 판별하는 단정문은?	
대상 값이 null이면 참이 되는 단정문은?	
호출 즉시 테스트 케이스를 실패로 판정하는 단정문은?	

JUnit 3 구성요소 3 테스트 러너 Test runner

- junit.swingui.TestRunner.run(테스트클래스.class);
- junit.textui.TestRunner.run(테스트클래스.class);
- junit.awtui.TestRunner.run(테스트클래스.class);

대부분의 Java 통합개발환경(IDE)은 JUnit 프레임워크를 내장 지원하고 있다. 그래서 종종 JUnit이 독립적인 프레임워크라기보다는 하나의 기능처럼 느껴질 수도 있다. 하

지만 JUnit 프레임워크는 엄연히 독립적인 소프트웨어이고, 애초부터 그렇게 만들어졌다. 때문에 명령행 프롬프트에서 실행하거나 셸 스크립트 등을 이용해 실행할 수도 있다. 이를 위해 JUnit은 테스트 러너라는 테스트 실행 클래스를 제공한다. 기본적으로 세 가지 실행 방법을 제공하는데, Swing UI, 텍스트 그리고 Java AWT UI이다. 다음은 DisplayTest라는 테스트 클래스를 세 가지 방식으로 모두 실행하는 예제다.

```
import junit.framework.TestCase;
public class DisplayTest extends TestCase {
    public void testGetString() {
        assertEquals("Happy", Display.getString());
    }
}

public static void main(String [] args){
    junit.swingui.TestRunner.run(DisplayTest.class);
    junit.textui.TestRunner.run(DisplayTest.class);
    junit.awtui.TestRunner.run(DisplayTest.class);
}
```

만일 특정 테스트 러너를 명령행⁶에서 직접 수행하려고 할 경우에는 다음과 같이 실행한다.

```
java - CP junit.jar;. junit.textui.TestRunner DisplayTest
```

JUnit 3 구성요소 4 **테스트 스위트** Test suite

- 여러 개의 테스트 케이스를 한꺼번에 수행하고자 할 때
- 테스트 스위트는 테스트 케이스와 다른 테스트 스위트를 포함시킬 수 있다.
- 메소드는 반드시 public static Test suite()여야 한다.
- 테스트 추가는 suite.addTestSuite(테스트 클래스.class) 형식을 갖는다.

⁶ cmd 창 혹은 커맨드 입력창이라고도 부른다.


```

Class DisplaySuiteTest {
    public static void main(String [] args){
        junit.swingui.TestRunner.run(DisplaySuiteTest.class);
    }

    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(DisplayTest.class); // 테스트 케이스 추가
        suite.addTestSuite(DisplayReceiverTest.class);
        suite.addTest(AnotherSuiteTest.suite()); // 다른 테스트 스위트를 포함할
                                                // 경우

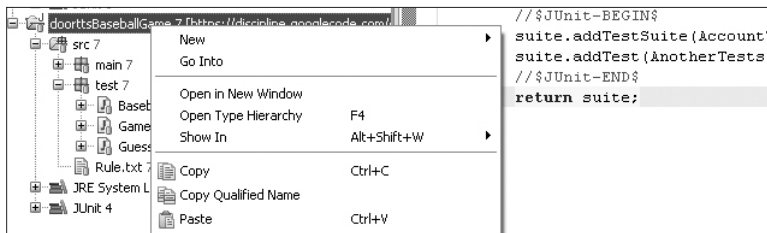
        return suite;
    }
}

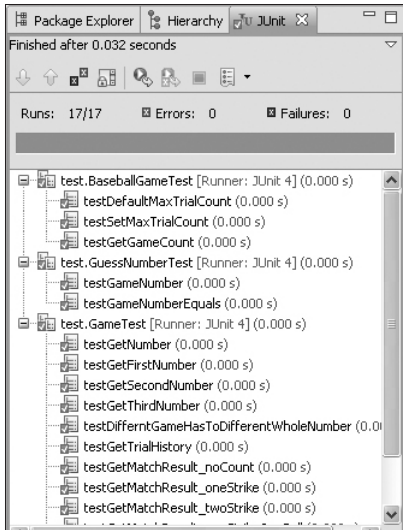
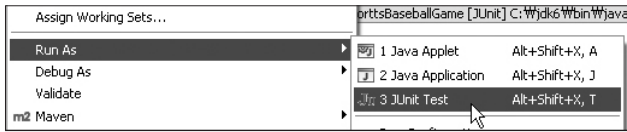
```

테스트 스위트는 여러 개의 테스트 케이스를 함께 수행할 때 사용한다. 예전에는 이런 식으로 테스트 스위트를 작성했으나 현재는 잘 사용하지 않는다. 테스트 클래스가 추가되거나 변경될 때마다 소스를 직접 수정해야 하기 때문이다. IDE에서 제공하는 기능을 이용하거나 Ant나 Maven 등의 빌드 도구를 사용해 다수의 테스트를 수행한다. Ant를 이용한 방법은 추후에 따로 다뤄보겠다. 다음은 이클립스 IDE의 기능을 이용해, 여러 개의 테스트를 한꺼번에 실행시키는 방법이다.

이클립스를 이용해 여러 개의 테스트를 한꺼번에 실행시키기

이클립스의 탐색창에서 프로젝트를 선택하거나 src를 선택한 다음 마우스 오른쪽 버튼을 이용해 JUnit Test를 실행하면, 그 하위에 위치하고 있는 테스트 클래스들이 모두 실행된다.





3개의 테스트 클래스, 총 17개의 테스트 케이스가 수행됐다.

JUnit 3으로 테스트 케이스 작성하기

1장에서 JUnit 4 버전으로 만들었던 Account 클래스와 Account에 해당하는 테스트 클래스를 JUnit 3 기준으로 다시 작성해보자. 동일하게 진행하면 다소 지겨울 수 있으니 이번엔 Account 클래스의 캡테기에 해당하는 클래스를 먼저 만들고, 테스트 케이스를 만들어보는 식으로 진행할 생각이다. 다음은 컴파일 에러가 나지 않는 수준으로 Account 클래스의 캡테기를 만들어본 모습이다.

```

package main;

public class Account {
    public Account(int i) {
    }

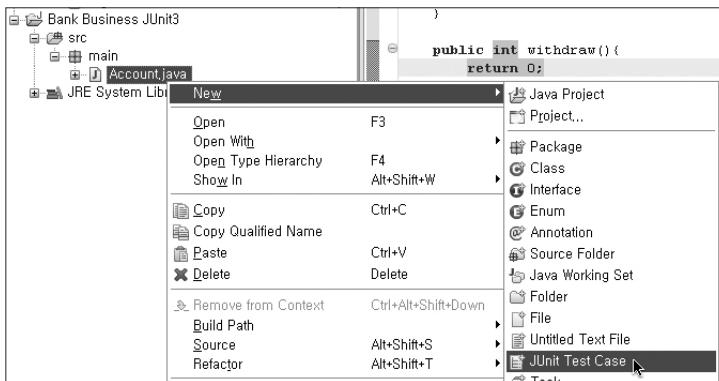
    public int getBalance() {
        return 0;
    }

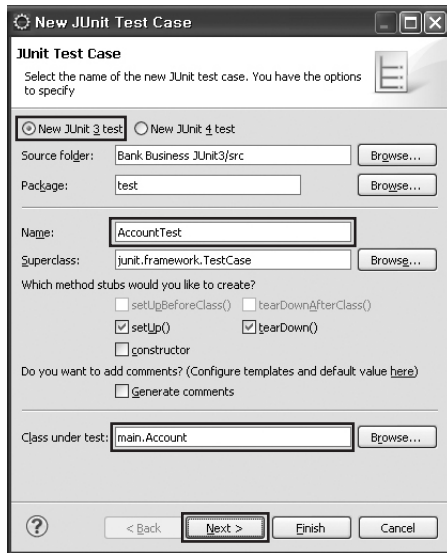
    public void withdraw(){
    }

    public void deposit(){
    }
}

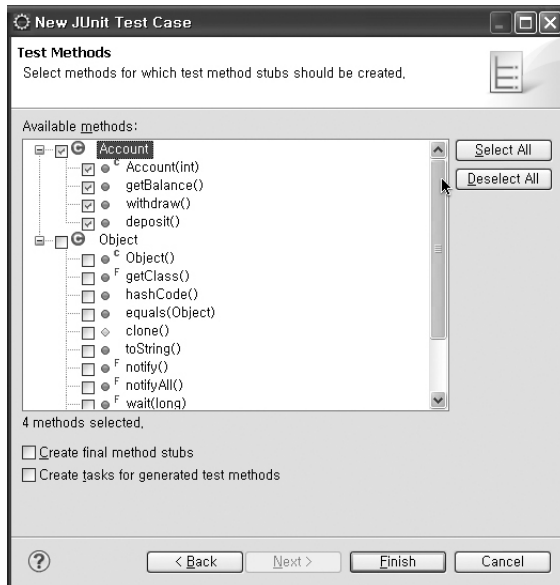
```

썬데기가 만들어졌으면, 그 다음엔 이클립스의 패키지 탐색창(Package Explorer)에서 Account.java를 선택하고 마우스 오른쪽 버튼을 눌러 JUnit Test Case를 선택한다.

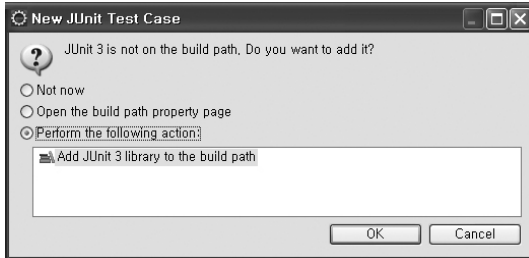




JUnit 3 test를 선택하고, 클래스의 이름과 테스트 대상이 되는 클래스를 적절하게 선택한 다음 Next 버튼을 누른다.

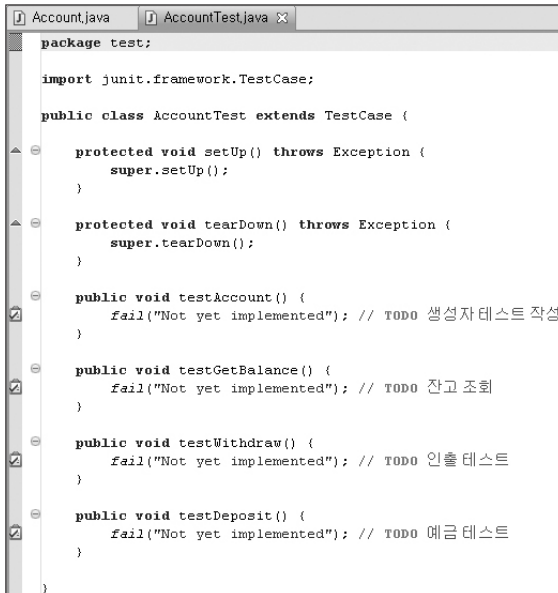


조금 전에 미리 만들어놓은 메소드들이 보인다. 대상 메소드 선택창 아래에 있는 Create tasks for generated test methods(자동 생성된 테스트 메소드들에 대해 작업 목록 만들기)를 선택하면 테스트 코드 껍데기가 만들어진 다음에 TODO 태그를 달아 준다. 이때 만든 TODO 태그는 이클립스의 Tasks 뷰에서 작업 목록처럼 확인할 수 있다. 테스트 케이스의 대상이 되는 메소드들을 선택한 다음 Finish 버튼을 누른다.



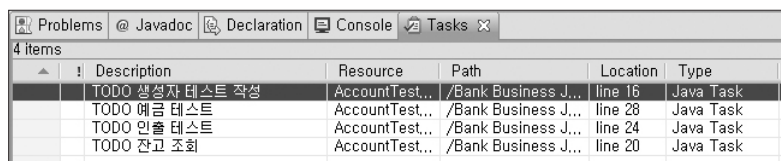
JUnit 3 라이브러리가 빌드 패스 안에 없네요. 추가하실래요?

만일 클래스패스 내에 JUnit 라이브러리가 없으면, 추가할 수 있도록 팝업창이 뜬다. OK를 누르자.



자동 생성된 테스트 클래스

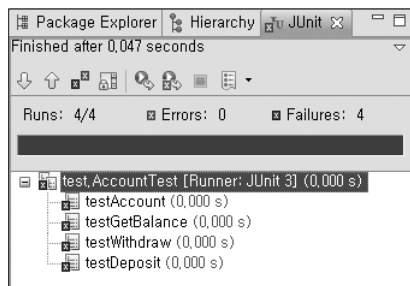
앞의 코드는 자동으로 생성된 테스트 케이스 껍데기에 TODO까지 적어본 모습이다. 지금은 로직이 간단해서 TODO를 적는 것이 크게 와 닿지 않을 수 있지만, 여기저기 왔다 갔다 하면서 소스를 작성해야 할 경우 TODO 표시를 해놓으면 나중에 이클립스의 Tasks 뷰를 통해 정리해서 볼 수 있어 편하다.



!	Description	Resource	Path	Location	Type
	TODO 생성자 테스트 작성	AccountTest...	/Bank Business J...	line 16	Java Task
	TODO 예금 테스트	AccountTest...	/Bank Business J...	line 28	Java Task
	TODO 인출 테스트	AccountTest...	/Bank Business J...	line 24	Java Task
	TODO 잔고 조회	AccountTest...	/Bank Business J...	line 20	Java Task

TODO로 지정해놓으면 Tasks 뷰에서 확인할 수 있다.

자동 생성된 테스트 케이스들을 수행시켜 보자.



테스트 케이스가 모두 실패하고 있다. 이제 테스트 메소드 각각에 들어 있는 fail() 메소드를 제거하며, 하나하나 테스트가 통과할 수 있도록 메소드를 작성해보자.

앞 장의 예제를 JUnit 3 버전으로 고쳐본 코드의 최종 모습은 아래와 같다.

AccountTest 클래스를 JUnit 3 버전에 맞추어 변경한 소스

```
import junit.framework.TestCase;

public class AccountTest extends TestCase {

    Account account;
```

```

protected void setUp() throws Exception {
    account = new Account(10000);
}

public void testGetBalance() {
    assertEquals(10000, account.getBalance());
}

public void testDeposit() {
    account.deposit(1000);
    assertEquals(11000, account.getBalance());
}

public void testWithdraw() {
    account.withdraw(1000);
    assertEquals(9000, account.getBalance());
}
}

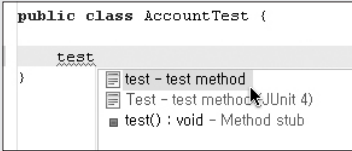
```

보면 알겠지만, **TestCase**를 상속했다는 점과 @로 시작하는 애노테이션이 없다는 점을 제외하면 작성 방식 자체는 유사하다(애노테이션에 대해서는 잠시 뒤에 나올 JUnit 4 부분에서 설명할 예정이니까, 잠시만 기다리자. 정 궁금하면 살짝 가서 보고 와도 무방하다). 앞서서도 이야기했지만, 이번처럼 작성하려는 클래스의 외형을 먼저 만들고 테스트 케이스를 작성하는 방식은 그다지 권장하는 방식은 아니다. 다만 기존에 작성되어 있는 코드⁷나, 일부 테스트 케이스 없이 작성 중인 코드에 대해 TDD를 진행할 때 IDE의 기능을 이용하면 좀 더 쉽고 빠르게 작성할 수 있다. 누누이 말할 테지만, 시작부터 습관을 잘 들이는 일이 중요하다. 가능한 한 하나씩 테스트 케이스를 작성해서 제품 코드를 만들어가는 방식으로 개발하자.

7 보통 이런 코드를 레거시 코드(legacy codes)라고 부른다. 이미 시스템이 구축되어 있고, 상당량의 코드가 작성되어 있는 상태에서 추가로 작업하게 될 때, 기존에 존재하는 코드를 레거시 코드로 통칭한다. 말 그대로 유산으로 물려받은 코드인 셈이다(비록 원했던 건 아니었을지라도 말이다).

이클립스 템플릿 기능을 이용한 테스트 메소드 만들기

이클립스 템플릿 기능을 사용하면 테스트 메소드를 좀 더 쉽게 만들 수 있다.



소문자로 test라고 친 다음 콘텐츠 어시스트(content assist)라 불리는 이클립스의 자동완성 기능(Ctrl+Space Bar)을 이용하면 된다. 위 그림에도 살짝 보이는데, 만일 JUnit 4 버전의 테스트 메소드를 만들고자 할 때는 소문자 대신, 대문자 T로 시작하는 Test라고 타이핑한 다음 자동완성 기능을 이용하면 된다.

JUnit 3으로 작성한 테스트 클래스의 구조

그리고 JUnit 3을 이용한 테스트는 일반적으로 다음과 같은 구조를 갖는다.

```
public class NetworkTest extends TestCase { // ❶
    private Connection connection; // ❷

    public void setUp() throws Exception {
        connection = new Connection("127.0.0.1"); // ❸
    }

    public void testSendMessage() throws Exception { // ❹
        ...
    }

    public void tearDown() throws Exception { // ❺
        connection.close();
    }
}
```

- ❶ TestCase를 상속(extends)받는 클래스를 만든다.
- ❷ 테스트에 사용할 테스트 픽스처를 정의한다.
- ❸ setUp 메소드를 사용해 픽스처의 상태를 초기화한다.

- ④ 이름이 test로 시작하는 테스트 케이스를 작성한다.
- ⑤ tearDown 테스트를 마친 다음 픽처를 정리할 코드를 작성한다.

저자가
한마디

JUnit 테스트 클래스에는 생성자가 없다?

느꼈을지 모르겠지만 JUnit 테스트 클래스는 자신의 생성자(constructor)를 만들지 않는다. 만들지 않은 게 아니라, Java 문법상 암묵적으로 기본 생성자는 생략 가능해서 표현하지 않은 것 아니냐고 물을 수도 있다. 그럼 명시적으로 만들어놓으면 어떻게 동작하는지 살펴보자. 이럴테면 앞선 AccountTest 예제에서 생성자를 만들었다고 가정해보자.

```
public class AccountTest extends TestCase {
    Account account;
    public AccountTest(){
        System.out.println("Constructor was called!");
    }
    ...
}
```

위와 같은 식으로 생성자를 만든 다음 JUnit 테스트를 한번 실행해보자. 테스트 케이스에 해당하는 메소드는 이전과 마찬가지로 정상 실행되고, 부가적으로 콘솔에 다음과 같이 출력이 된다.

```
Constructor was called!
Constructor was called!
Constructor was called!
Constructor was called!
```

생성자가 네 번 호출된 것처럼 보인다. 에엣? 뭐지? 생성자는 한 번만 호출되는 것 아닌가? 왜 이리 많이 호출되는 거지? 그리고 네 번 찍힌 이유는 또 뭘까?

AccountTest의 테스트 메소드를 세어보자. testAccount(), testDeposit(), testWithdraw(), testGetBalance() 네 개!!

자, 조금 느낌이 오는가? JUnit 프레임워크는 Java의 리플렉션(reflection)⁸을 사용해서 테스트 메소드를 실행할 때마다 테스트 클래스를 강제로 인스턴스화한다. 왜 그랬을까? 왜 테스트 메소

8 원 단어의 뜻은 '거울 등에 비친 영상', 또는 '반사'의 의미다. Java에서는 리플렉션이라는 기능을 통해 인스턴스화된 객체로부터 원래 클래스의 구조를 파악해내어 동적으로 조작하는 것이 가능하다. 마치 기계를 분해해서 마음대로 재구성하는 것처럼 말이다. 리플렉션을 이용하면 private 메소드나 필드까지도 마음대로 조작할 수 있다. 리플렉션을 사용하면, 이런 식으로 Java의 일반적인 규칙들을 무시할 수도 있기 때문에, 보통 한정적으로만 사용할 것을 권장한다. 그리고 시스템 비용이 매우 많이 드는 기능이다.

드를 실행할 때마다 객체를 새로 만들어내는 것일까? (계속 일어나가기 전에 잠깐 책을 덮고 생각해보자!) 좋은 테스트 케이스는 기본적으로 다른 테스트 케이스의 수행이나 수행 결과에 영향을 받지 않아야 한다. 이것이 테스트의 기본 원칙이다. 따라서 각각의 테스트 케이스를 독립적으로 수행하기 위해, 테스트 메소드 수행 전에 테스트 클래스 자체를 리셋한다. 그런 다음 각각의 테스트 메소드만 수행하는 식으로 다른 테스트 메소드들로 인한 영향을 최소화한다. 좀 더 자세한 동작 방식이 궁금하다면 JUnit 소스를 직접 살펴보면 아주 약하게 권한다. 왜냐하면 우리의 시간은 소중한니까. 이것 말고도 알아야 할 게 무궁무진하다. :)

2.1.2 JUnit 4

JUnit 4의 특징

1. Java 5 애노테이션 지원

2. test라는 글자로 method 이름을 시작해야 한다는 제약 해소

Test 메소드는 @Test를 붙인다.

3. 좀 더 유연한 픽스처

@BeforeClass, @AfterClass, @Before, @After

4. 예외 테스트

@Test(expected=NumberFormatException.class)

5. 시간 제한 테스트

@Test(timeout=1000)

6. 테스트 무시

@Ignore("this method isn't working yet")

7. 배열 지원

assertArrayEquals([message], expected, actual);

8. @RunWith(클래스이름.class)

JUnit Test 클래스를 실행하기 위한 러너(Runner)를 명시적으로 지정한다.

@RunWith는 junit.runner.Runner를 구현한 외부 클래스를 인자로 갖는다.

9. @SuiteClasses(Class[])

보통 여러 개의 테스트 클래스를 수행하기 위해 쓰인다. @RunWith를 이용해 Suite, class를 러너로 사용한다.

10. 파라미터를 이용한 테스트

```
@RunWith(Parameterized.class)
@Parameters
public static Collection data() {
    ...
}
```

2002년 JUnit 3.8 버전을 이후로 한동안 새로운 버전이 없던 JUnit이 2006년에 메이저 버전업을 하며 다시 돌아왔다. 기본적인 실행철학은 3 버전과 크게 다르지 않지만, 테스트 케이스 작성 및 실행을 편리하게 해주는 다양한 기능이 추가됐다. JUnit 4에서의 가장 큰 변경점을 꼽으라면 Java 5에서부터 지원되기 시작한 애노테이션(annotation)⁹ 지원 그리고 JUnit 4 버전대 중반부터 지원되기 시작한 비교 표현을 위한 테스트 매치 라이브러리(Test Matcher Library)인 Hamcrest 도입, 이렇게 크게 두 가지를 꼽을 수 있겠다. Hamcrest에 대해서는 조금 뒤에 보기로 하고 우선 애노테이션 도입에 따른 변경을 살펴보기로 한다.

애노테이션

JDK의 히스토리를 이야기할 수 있을 정도의 연륜 있는 개발자라면, JDK의 여러 버전 중에서 특별히 구분해 선을 그을 만한 부분을 기억할 것이다. 크게 두 버전 정도로 나누어 이야기를 하곤 하는데, 보통 첫 번째로 JDK 1.2를 꼽는다. 버전으로는 1.1에서 1.2로 0.1 올라섰지만, 개발자 입장에서는 변화의 폭이 상당히 커서 1.2 버전은 공공연히 Java 2라 불렸다. 그 다음으로 JDK 1.5를 많이 뽑는데, 이 버전은 공식적으로 Java 5¹⁰

9 마치 주석처럼 @ 기호와 함께 선언적인 형태로 코드에 달려 있는 문장.

10 이후 썬(Sun)은 Java 5, Java 6 같은 식으로 한 자리 숫자로 JDK 버전을 부르기로 결정한다. 하지만 다운로드 받는 파일은 여전히 1.5, 1.6, 1.7 형태로 되어 있는데, 이후 버전이 올라가서 2.0이 되면 Java 10이라고 부를 생각인 건지 궁금하다.

라 불린다. API 추가/변경 정도로 버전업을 하다가 1.5에 이르게 되면서 애노테이션, 제네릭스(generics, c++의 템플릿과 유사), 향상된 for 루프문, 타입세이프한 열거형 타입(type safe enumerations, Enum) 등 새로운 개념의 요소가 많이 추가됐다.¹¹ 특히 애노테이션과 제네릭스는 개발자들 사이에서도 찬반양론 말이 많았다. 그도 그럴 것이 그 이전 버전만 개발한 개발자들에게 1.5 버전에서 추가된 기능과 키워드, 문법 등으로 개발된 소스를 보여주면, 한참 멍하니 쳐다보게 된다. 그러나 그건 Java 5가 처음 나왔던 2004년 시절 이야기여야 한다. 최근에 새로 배우는 사람들은 Java 5의 문법이나 기능을 원래 Java가 그런 건가 보다 하고, 있는 그대로 받아들인다. 그렇기 때문에 버전업을 거쳐왔던 기존 개발자와는 달리 Java 5 기능들에 대한 거부감이 확실히 적다. 그리고 거부감 문제를 떠나서 새로 추가된 기능들은 Java 언어 자체가 좀 더 넓게 확장하고 다양하게 발전할 수 있도록 도와줬다. 특히 그중 하나가 바로 애노테이션이다.

애노테이션의 가장 큰 장점 하나는 프레임워크의 내부 모델에 대한 자세한 이해 없이도 각 메소드의 사용 의도를 명확하게 문서화한다는 점이다. 이를테면 JUnit 4의 경우처럼 @Test가 붙어 있으면, 척 보기도 ‘아! 테스트 대상 메소드이구나!’라고 직관적으로 파악이 가능하다. 물론, 이건 단순한 사례 중 하나일 뿐이지만 말이다.

근래에 만들어지는 대부분의 Java 애플리케이션은 애노테이션을 적극 활용하는 추세다. 한 가지 불편한 점은 JUnit 3에서 JUnit 4로 넘어가는 분기점이 애노테이션을 사용할 수 있느냐 없느냐로 구분된다는 점이다. 따라서 JUnit도 애노테이션 기능을 추가해서 확장과 발전을 도모하게 되는데, 그 시작이 JUnit 4.0이다. 물론 3.8 이후로 수년간 메이저 버전업이 없던 사이에 TestNG라는 애노테이션 기반의 xUnit 테

11 만일 아직 Java 5의 기능을 확인해보지 않았다면, 이번 기회에 꼭 한번 살펴볼 것을 권장한다. 근래에 만들어지는 대부분의 프레임워크는 Java 5 이후의 기능들을 적극 사용해 안정성과 가독성을 높이는 방향으로 작성되고 있다. 때문에 언제가 됐든 알아야 하는 날이 온다(그러니 이왕이면 미루지 말자). 참고로 Java 5도 이미 서비스 지원종료(end of service life, EOSL)로 들어섰다. 이전 Java 6, Java 7 등을 봐야 할 시기이지만, 우선 한 단계씩 차분히 나아가자. 웹 검색을 통해 여러 사람이 작성한 해설 내용을 보는 것도 좋지만, 우선은 썬의 공식문서인 ‘New Features and Enhancements J2SE 5.0’(http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html)을 볼 것을 추천한다. 검색으로 찾게 되는 단편적인 지식이 아닌, 좀 더 상세하고도 광범위한 정보를 얻을 수 있다. 정 시간이 없다면, 간략화 버전인 ‘J2SE 5.0 in a Nutshell’(http://java.sun.com/developer/technicalArticles/releases/j2se15/)이라도 봐두자.

스트 프레임워크가 주목을 받았었다. TestNG는 Retroweaver(<http://retroweaver.sourceforge.net/>)라는 라이브러리를 이용해 JDK 1.4 이하에서도 애노테이션을 쓸 수 있게 해줬다.¹²

애노테이션 적용에 있어 TestNG보다 후발 주자였던 JUnit 4는 의도했던 아니든 TestNG의 상당 기능을 그대로 차용해왔고, 그로 인해 일부 비난을 받기도 했다. 어쨌든 JUnit 4는 애노테이션 사용을 특정 라이브러리에 의존하지 않고 순수 JDK 기능 지원을 이용하기로 정했기 때문에 JUnit 4를 기동하기 위해 필요한 최소 JDK 버전은 Java 5(JDK 1.5)가 됐다. 어찌 보면 별일 아닐 수 있지만, 사용자 입장에서는 소위 말하는 ‘최소사양’이 올라가 버리게 된 상황이 불만족스러울 수도 있다. 그래서 예전에는 JDK 1.4 이하 버전에서도 다양한 픽스처나 부가기능을 누리기 위해서 JUnit 대신 TestNG를 테스트 프레임워크로 사용하는 경우가 종종 있었다. 하지만 흐르는 강물은 되돌릴 수 없다고 했던가? 인지도 측면에서 JUnit이 월등히 높았고, 거의 비슷한 표현식에 동일한 기능까지 제공하는 JUnit 4가 나오면서, 애노테이션을 이용한 테스트 케이스 작성에도 JUnit이 주로 사용되게 됐다. 사실 JUnit 3과 그 이전 버전에서 공격받았던 점 중 하나는, JUnit이 테스트 클래스 내에 존재하는 메소드의 이름 중에서 test라는 글자로 시작하는 메소드만을 테스트 메소드로 인식해 실행한다는 점이였다. 애노테이션은 오랫동안 끌어왔던 이런 문제점을 해결해줬다. 그뿐 아니라, 애노테이션을 사용해서 더 강력한 테스트 방식들을 더 간결히 지원해줄 수 있게 됐다. 예를 들어 확장 기능을 사용하기 위해서 라이브러리를 호출하거나 부모 클래스로부터 물려받지 않고, 미리 정의된 애노테이션만 호출하면 끝나는 식으로 말이다. 그럼 이제부터는 JUnit 4에서 제공하는 애노테이션들과 이를 통한 추가 기능에는 어떤 것들이 있는지 차근차근 살펴보자.

12 Retroweaver 라이브러리를 사용하면 애노테이션뿐 아니라 Java 5의 새로운 기능 대부분을 1.4에서 사용할 수 있다. 심지어 기본 타입(primitive type)과 Wrapper 클래스를 직접 비교/할당 가능하게 만들어주는 오토박싱(autoboxing/unboxing)까지 말이다.



애노테이션의 기원, XDoclet

Java 언어는 @ 기호와 Javadoc 유틸리티를 이용해 한정적으로 문서를 자동으로 생성해주는 기능을 초창기부터 제공했었다. 프로그래밍 코드 중간에 @ 기호를 이용해 일종의 짤막한 명세(spec)를 작성하면, 그걸 툴이 알아서 문서로 만들어주는 기능인데, 많은 사람이 편리해했다. 이런 식의 Javadoc 기능을 이용한 API 문서 생성처럼, 프로그램 코드 내부에 적혀 있는 선언적인 주석기호 @을 이용한 작업 방식에 관심을 기울였던 사람들이 있었다. 이들이 만들어낸 것이 바로 XDoclet이라는 소스코드 생성 엔진이다. XDoclet은 Javadoc의 Doclet 엔진을 확장해서 만든 오픈소스 프로젝트로, 개발자의 작업량을 줄여준다는 점과 속성 기반 프로그래밍(attribute oriented programming)이라는 이름의 개발 방식으로 한동안 유명했었다. 간단히 말하면 ‘주석 문장으로 문서만 만드냐? 소스코드도 만들어내자!’ 이런 취지였는데, 현재는 초반 열기와 다르게 이런저런 이유로 한정적으로만 사용되는 분위기다. 어쨌든 XDoclet은 속성 기반 프로그래밍이라는 개발 방식에 대한 인지를 넓히는 계기가 됐고, 현재의 애노테이션을 이용한 개발에 많은 영향을 줬다.

@Test

기존에는 테스트 케이스에 해당하는 메소드로 지정하기 위해서는 메소드 이름을 소문자 test로 시작해야 한다는 규칙이 있었다. 이런 방식을 흔히 작명 패턴(naming pattern)이라고 하는데, JUnit 4에 와서는 메소드 이름과 관계없이 @Test 애노테이션만 붙이면 테스트 메소드로 인식한다. 즉, 더 이상 소문자 test로 메소드의 이름을 시작하지 않아도 된다는 뜻이다. 하지만 많은 사람이 여전히 메소드 이름을 소문자 test로 시작하는 방식을 따르고 있다.

@Test

```
public void testDeposit() throws Exception {
    account.deposit(1000);
    assertEquals(11000, account.getBalance());
}
```

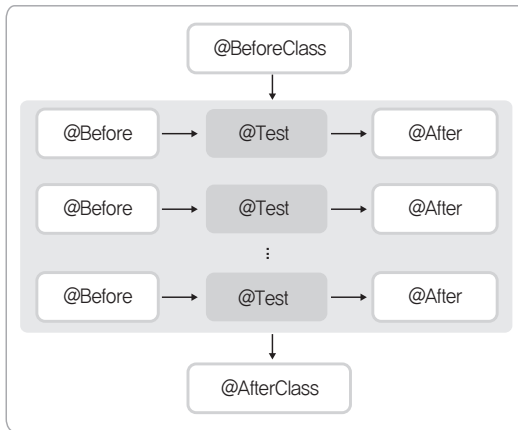
개발자에 따라서는 “test 소문자로 시작하던 걸 애노테이션 붙여서 @Test로 사용하는 게 무슨 장점이 된거냐?”고 물을 수도 있는데, 생각보다 많은 사람이 testDeposit 대신

에 `tsetDepoist`으로 타이핑하고는 ‘왜 실행이 안 될까?’ 고민한다. `@Test`로 지정해놓으면 그럴 일이 없다.¹³

테스트 픽스처 메소드 추가 지원

JUnit 3에서는 `setUp`과 `tearDown`이라는 두 개의 테스트 픽스처 메소드를 제공했는데, JUnit 4에서는 각각을 `@Before`와 `@After`라는 이름의 애노테이션으로 지원해준다. 그리고 JUnit 3에서는 테스트 클래스에서 단 한 번만 실행할 수 있게 해주는 방법을 공식적으로 제공하지 않아 다소 불편했었는데, JUnit 4에서는 `@BeforeClass`, `@AfterClass` 두 개의 애노테이션을 이용해 하나의 테스트 클래스 내에서 한 번만 실행하는 메소드를 만들 수 있게 됐다.

TestClassB



추가된 테스트 픽스처 메소드와 실행 순서

13 “`@Test` 애노테이션 스펠링도 틀릴 순 있진 않느냐? `@Tset`으로 타이핑하는 건 왜 배제하느냐?”고 까탈스럽게 물을 수도 있는데, 애노테이션을 사용하면 컴파일러가 개발자의 눈 대신에 체크를 해주기 때문에 그런 일은 생기지 않는다. 이와 비슷한 경우로 수많은 개발자가 클래스를 상속받아서 오버라이드할 때 오타를 내서 의도하지 않은 결과를 만들곤 했다. Java 5부터 지원되는 `@Override` 애노테이션이 바로 그런 상속 시 발생하는 작명 패턴(부모와 메소드 이름과 시그니처가 같아야 한다)의 오류를 막아준다. 편리하고 유용하니 여러분도 적극적으로 사용해보면 좋을 것이다.

다음은 터미널 클라이언트 프로그램에 대한 테스트 클래스다. 네트워크 연결은 한 번만 하고, 각각의 테스트 케이스를 **로그인** → **테스트 수행** → **로그오프** 순으로 실행한다. 그리고 테스트 케이스를 모두 수행한 다음엔 네트워크 접속을 종료한다.

```
public class TerminalTest {

    private static Terminal term;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        term = new Terminal();
        term.netConnect(); // 터미널에 접속한다.
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
        term.netDisconnect(); // 터미널과의 연결을 해제한다.
    }

    @Before
    public void setUp() throws Exception {
        term.logon("guest", "guest"); // 시스템에 로그인
    }

    @After
    public void tearDown() throws Exception {
        term.logoff(); // 시스템으로부터 로그오프
    }

    @Test // 정상적으로 로그인됐는지 테스트
    public void testTerminalConnected() throws Exception {
        assertTrue( term.isLogon() );
        System.out.println("== logon test");
    }

    @Test // 터미널 메시지 테스트
```



```

        public void testGetReturnMessage() throws Exception {
            term.sendMessage("hello");
            assertEquals("world!", term.getReturnMessage());
            System.out.println("== message test");
        }
    }

```

실행 결과는 다음과 같다.

```

Network is established. .... setUpBeforeClass()
>>logon guest:guest ..... setUp()
== logon test ..... testTerminalConnected()
<<logoff ..... tearDown()
>>logon guest:guest ..... setUp()
== message test ..... testGetReturnMessage()
<<logoff ..... tearDown()
Network is disconnected. .... tearDownAfterClass()

```

예외 테스트

JUnit 3에서는 예외를 테스트하는 공식적인 방법을 제공하지 않았다. 대신에 try/catch 문과 assert 단정문을 일종의 트릭처럼 사용해서 테스트를 진행하곤 했다. 다음은 JUnit 3에서 예외를 테스트하는 코드다.

```

public void testException(){
    String value = "a108";
    try {
        System.out.println(Integer.parseInt(value));
        assertTrue(false);
    } catch (NumberFormatException nfe){
        assertTrue(true);
    }
}

```

JUnit 3에서는 예외처리를 테스트하려면 위와 같이 try/catch 문을 사용해야만 했다. 다음은 동일한 코드를 JUnit 4의 애노테이션을 이용해 작성한 코드다.

```
@Test (expected=NumberFormatException.class)
public void testException(){
    String value = "a108";
    System.out.println(Integer.parseInt(value));
}
```

JUnit 4에서는 @Test 애노테이션의 속성 중 expected를 이용해 예외처리를 테스트한다. expected의 값으로 예외 클래스를 지정했을 때, 만일 테스트 메소드 내에서 해당 예외가 발생하지 않는다면 테스트 메소드를 실패로 간주한다.

테스트 시간 제한

```
@Test(timeout=5000)
public void testPingback() throws Exception {
    ciServer.sendNotimail();
    assertEquals(NOTIFICATED, ciServer.getState());
}
```

@Test의 속성으로 timeout을 지정하고, 그 값으로 밀리초 단위의 시간을 정해준 후 해당 시간 내에 테스트 메소드가 수행완료되지 않으면 실패한 테스트 케이스로 간주한다.

테스트 무시

```
@Ignore @Test
public void testTerminal() throws Exception {
    assertTrue(term.isLogon());
    System.out.println("== logon test");
}
```

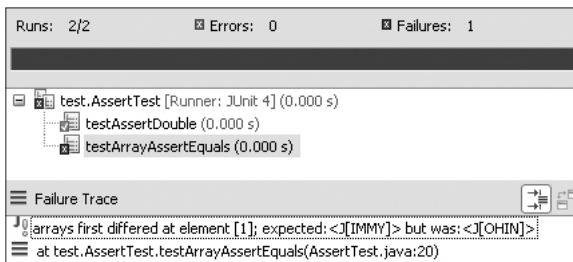
@Ignore가 붙은 테스트 메소드는 해당 애노테이션을 지우기 전까진 수행하지 않는다.

배열 지원

```
@Test
public void testArrayAssertEquals() throws Exception {
    String [] names = {"Tom", "JIMMY", "JOHIN"};
    String [] anotherNames = {"Tom", "JIMMY", "JOHIN"};
    assertArrayEquals(names, anotherNames);
}
```

JUnit 3에서는 지원되지 않았던 기능으로, 배열을 비교할 수 있게 해준다. 아쉽게도 배열 원소의 자리 순서 기준으로 equals 비교가 이뤄지기 때문에 비록 배열 안의 값 집합은 동일하더라도 순서가 다르면 테스트가 실패한다는 점을 유의하자. 다음은 해당 예를 보여준다.

```
@Test
public void testArrayAssertEquals() throws Exception {
    String [] names = {"Tom", "JIMMY", "JOHIN"};
    String [] anotherNames = {"Tom", "JOHIN", "JIMMY"};
    assertArrayEquals(names, anotherNames);
}
```

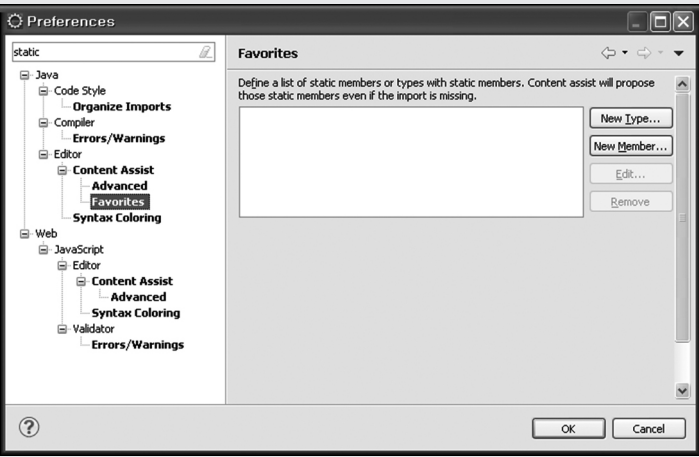


배열 안의 순서가 다르면 실패한다.

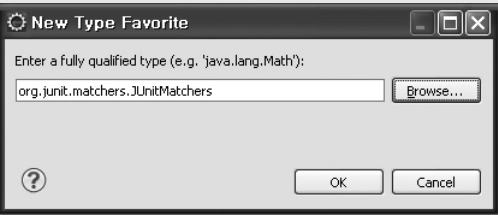


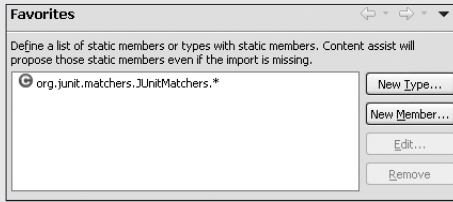
자동완성 기능을 이용해 static 클래스의 메소드 사용하기

JUnit 4는 TestCase 상속 대신에 애노테이션을 사용하기 때문에 assertEquals 같은 기본 메소드조차 static import로 지정하지 않으면 쓸 수가 없다. 자동완성 기능도 제대로 동작하지 않을 때가 많은데, 이클립스가 모든 static 클래스에 대한 메소드를 미리 인지하고 있을 수 없기 때문이다. 결국 개발자가 직접 static import 구문을 타이핑하거나, 전체 클래스 이름(full qualified name)을 적어줘야 하는 불편함이 발생한다. 이를테면 JUnit 4의 matcher 메소드 중 하나인 containsString이라는 메소드를 호출하기 위해서는 static import org.junit.matchers.JUnitMatchers.*;와 같은 import 구문을 직접 타이핑하거나 org.junit.matchers.JUnitMatchers.containsString(~~); 식으로 호출해야 한다. 이런 불편함을 해결하는 방법은 이클립스 자동완성(Content Assist)의 Favorites 기능을 이용하는 것이다. 이클립스 설정(Preferences)의 필터창에 static이라고 타이핑한 다음 Editor → Content Assist → Favorites를 선택한다.

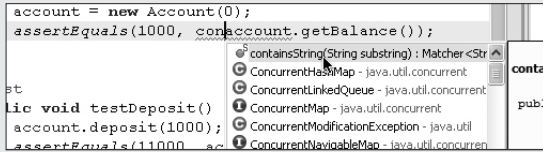


그 다음 New Type을 선택해서 자주 사용하는 static 클래스를 등록한다.





등록된 static 클래스가 표시된다.



이제 에디터 창에서 자동완성 기능을 이용해 쉽게 static 클래스의 메소드들을 사용할 수 있다.

@RunWith

JUnit 프레임워크에서 테스트 클래스 내에 존재하는 각각의 테스트 메소드 실행을 담당하고 있는 클래스를 테스트 러너(Test Runner)라고 한다. 이 테스트 러너는 테스트 클래스의 구조에 맞게 테스트 메소드들을 실행하고 결과를 표시하는 역할을 수행한다. 우리 눈에는 보이지 않지만, 테스트 케이스를 이클립스에서 실행하면 내부적으로는 JUnit의 BlockJUnit4ClassRunner라는 테스트 러너 클래스가 실행되고, 이클립스는 그 결과를 해석해서 우리에게 보기 편한 화면으로 보여준다. @RunWith 애노테이션은 JUnit에 내장된 기본 테스트 러너인 BlockJUnit4ClassRunner 대신에 @RunWith로 지정된 클래스를 이용해 클래스 내의 테스트 메소드들을 수행하도록 지정해주는 애노테이션이다. 일종의 JUnit 프레임워크의 확장지점인 셈이다. 사실, 사용하는 입장에서는 이렇게까지 자세히 알 필요는 없다. 다만, 이런 구조를 이용해서 많은 애플리케이션이나 프레임워크가 자신에게 필요한 테스트 러너를 직접 만들어 자신만의 고유한 기능을 추가해 테스트를 수행하고 있다는 것 정도만 알아두자. 이를테면 스프링 프레임워크

(Spring Framework)¹⁴에서 제공하는 SpringJUnit4ClassRunner 같은 클래스는 이 확장 기능을 이용한 대표적인 사례 중 하나다. @RunWith로 SpringJUnit4ClassRunner.class를 지정하면 @Repeat, @ProfileValueSourceConfiguration, @IfProfileValue 등 스프링에서 자체적으로 만들어놓은 추가적인 테스트 기능을 이용할 수 있게 된다. 그리고 굳이 외부가 아니더라도 JUnit 내부에도 기본 러너를 확장한 클래스가 몇 개 있는데, 그중 하나가 SuiteClasses이다.

@SuiteClasses

Suite 클래스는 JUnit 3에서의 static Test Suite() 메소드와 동일한 일을 수행한다. 즉, @SuiteClasses 애노테이션을 이용하면 여러 개의 테스트 클래스를 일괄적으로 수행할 수 있다. 다음 예제는 ATest, BTest, CTest 세 개의 테스트 클래스를 순차적으로 실행하고 싶을 때 JUnit 3과 JUnit 4에서는 각각 어떻게 작성해야 하는지를 보여준다.

JUnit 4의 경우

```
@RunWith(Suite.class)
@SuiteClasses(ATest.class, BTest.class, CTest.class)
public class ABCSuite {
}
```

JUnit 3의 경우

```
public class ABCSuite extends TestCase {
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(ATest.class);
        suite.addTestSuite(BTest.class);
        suite.addTestSuite(CTest.class);
    }
}
```

14 객체 간에서 발생하는 의존성 처리를 컨테이너가 처리하도록 만든 의존관계 주입 컨테이너(Dependency Injection Container)와 관점지향 프로그래밍(Aспект-Ориентированное Программирование) 기법 활용 등으로 유명한 자바 프레임워크. 각각에 대해 좀 더 알고 싶으면 위키피디어를 참고하면 많은 도움이 된다.

```

        return suite;
    }
    ...

```

JUnit 4에는 이 외에도 흥미로운 기능들이 몇 개 더 추가되어 있다. 해당 내용은 이 책의 부록 A.1절 ‘놓치기엔 아까운 JUnit 4의 고급 기능들’에서 설명하고 있다. 중요하지 않아서라기보다는 중급 이상의 개발자들에게조차 다소 어렵고, 도전적인 내용들이기에 위치를 옮겼다. 힘은 들겠지만, 익혀두면 확실히 도움이 될 내용들이 포함되어 있으니 언젠가 되었던 꼭 살펴보도록 하자. 대신 여기서는 어떤 내용이 있는지 정도만 간략히 소개할까 한다. 2장 시작 부분의 체크 리스트에 표시해놓고 기회가 될 때 부록에 수록된 내용을 살펴보자.

파라미터화된 테스트(Parameterized Test)

하나의 메소드에 대해 다양한 테스트 값을 한꺼번에 실행시키고자 할 때 사용한다. 이를테면 다음과 같은 과세표준 전 구간을 테스트하려 한다면 어떻게 테스트 시나리오를 작성하고 실행할 것인가가 이슈가 된다.

2008년 과세표준	세율	누진공제액
1200만 원 이하	8%	0원
1천2백만 원 초과~4천6백만 원 이하	17%	108만 원
4천6백만 원 초과~8천8백만 원 이하	26%	522만 원
8천8백만 원 초과	35%	1,314만 원

테스트 메소드가 많아지거나 구문이 장황해질 수 있는데, 파라미터화된 테스트를 사용하면 테스트 케이스를 효율적으로 작성할 수 있다.

룰(Rule)

JUnit 4.7 버전부터 추가된 기능으로 하나의 테스트 클래스 내에서 각 테스트 메소드의 동작 방식을 재정의하거나 추가하기 위해 사용하는 기능이다. 테스트 케이스 수행을 좀 더 세밀하게 조작할 수 있게 된다. 룰의 종류는 다음과 같다.

규칙 이름	설명
TemporaryFolder 임시폴더	테스트 메소드 내에서만 사용 가능한 임시 폴더나 임시 파일을 만들어준다.
ExternalResource 외부 자원	외부 자원을 명시적으로 초기화한다.
ErrorCollector 에러 수집기	테스트 실패에도 테스트를 중단하지 않고 진행할 수 있게 도와준다.
Verifier 검증자	테스트 케이스와는 별개의 조건을 만들어서 확인할 때 사용한다.
TestWatchman 테스트 감시자	테스트 실행 중간에 사용자가 끼어들 수 있게 도와준다.
TestName 테스트 이름	테스트 메소드의 이름을 알려준다.
Timeout 타임아웃	일괄적인 타임아웃을 설정한다.
ExpectedException 예상된 예외	테스트 케이스 내에서 예외와 예외 메시지를 직접 확인할 때 사용한다.

이론(Theory)

테스트 데이터와 상관없이 작성 대상 메소드를 항상 유지해야 하는 논리적인 규칙을 표현할 때 사용한다. 아직은 실험적인 성격이 강한 기능이다. 작성된 코드의 모습이 어떠한지 정도만 살펴보고, 흥미가 간다면 마찬가지로 부록을 참조하기 바란다. 외국 개발자들조차도 많이 어려워하는 내용이다.

```
@RunWith(Theories.class)
public class SquareRootTheoryTest {
    @DataPoint public static double FOUR = 4.0;
    @DataPoint public static double ZERO = 3.0;

    @Theory
    public void defineOfSquareRoot(double n){
        assertEquals(n, sqrt(n)*sqrt(n), 0.01);
    }
}
```



```

        assertTrue( sqrt(n) >= 0 );
    }
    ...

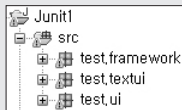
```

정수 n 은 $\sqrt{n} * \sqrt{n}$ 과 같다는 이론을 정의해놓고 있다. 테스트 케이스의 데이터 값뿐만 아니라, 이 공식도 만족시켜야 테스트가 정상 통과한다.

스어자
한 = tcl

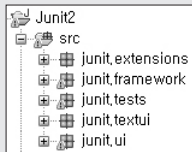
JUnit 버전대별 패키지 크기와 파일 개수(예제 샘플패키지 제외)

• JUnit 1



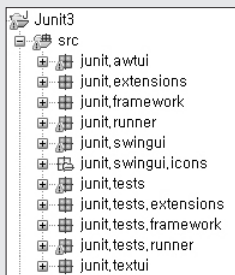
패키지 수 3개, Java 파일 수 14개, 테스트 파일 포함되어 있지 않음

• JUnit 2(2.1 기준)



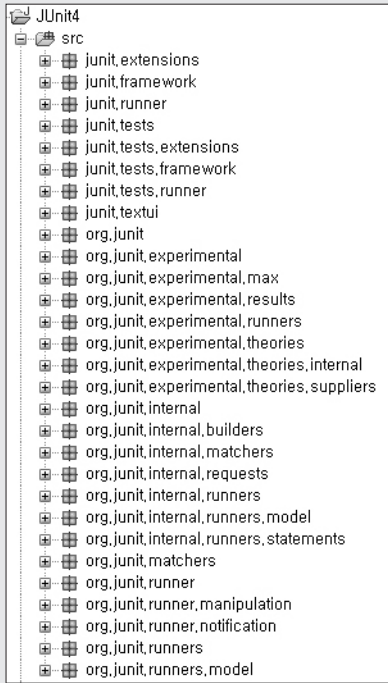
패키지 수 5개, Java 파일 수 21개, 테스트 파일 11개

• JUnit 3(3.8 기준)



패키지 수 10개, Java 파일 수 49개, 테스트 파일 37개

• JUnit 4(4.6 기준)



패키지 29개, Java 파일 137개, 테스트 36개

그리고 JUnit 4.4 이상 버전부터는 Hamcrest 라이브러리를 의존패키지로 갖는다. 보면 알겠지만 JUnit 4가 한때 비난을 받았던 이유 중 하나가 바로 클래스의 증가로 인한 복잡도 증가였다. 다양한 요구사항과 변화하는 개발환경에 대응하기 위해서는 어쩔 수 없는 선택이었겠지만, 원 저작자였던 켄트 벡도 언급했듯이 프레임워크 자체의 복잡도 증가는 다소 아쉬운 점이다.

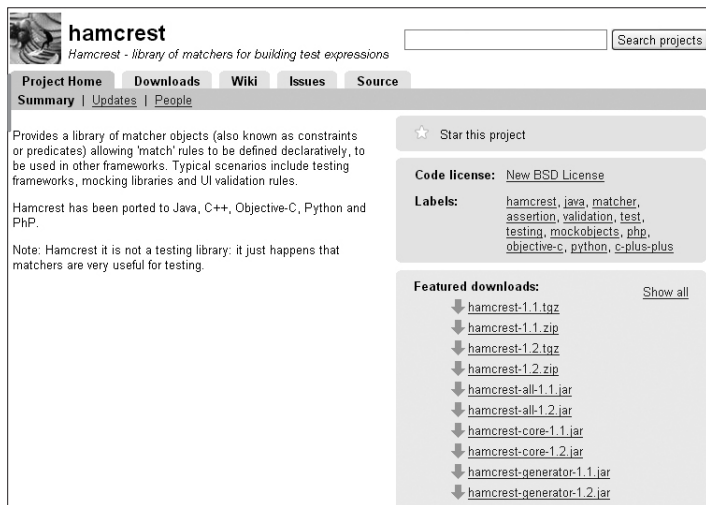
2.2 비교표현의 확장: Hamcrest

Hamcrest(햄크레스트)는 jMock이라는 Mock 라이브러리 저자들이 참여해 만들고 있는 Matcher 라이브러리¹⁵로, 테스트 표현식을 작성할 때 좀 더 문맥적으로 자연스럽게

15 Matcher 라이브러리: 필터나 검색 등을 위해 값을 비교할 때 좀 더 편리하게 사용할 수 있게 도와주는 라이브러리. 앞에서 JUnit 라이브러리를 이클립스의 빌드 경로에 추가할 때 함께 뒤따라 붙었다고 이야기했던 org.hamcrest.core_1.1.0의 정체가 바로 이것이다.

고, 우아한 문장을 만들 수 있게 도와준다. Matcher는 이름 그대로 어떤 값들의 상호 일치 여부나 특정한 규칙 준수 여부 등을 판별하기 위해 만들어진 메소드나 객체를 지칭한다. 지칭하는 사람에 따라서는 언어 자체에서 지원하는 기본적인 비교표현식인 `==`, `>`, `>=`, `<=`에서부터 `equals`나 `contains`처럼 대조를 위해 만드는 메소드까지, 조건 일치 여부를 알려줄 수 있다면 어떠한 것이든 Matcher의 범주에 포함시키기도 한다. 흔히 데이터 필터링(data filtering)이나 UI 유효성 검증(validation) 등의 기능을 구현할 때 Matcher 관련 기능이 많이 사용된다.

Hamcrest는 다양한 Matcher들이 모인, Macher 집합체다. 초기에 jMock 라이브러리의 일부로 포함되어 들어 있던 '비교표현 API'들에 불과했으나, 그 유용성을 인정받아 리팩토링을 통해 jMock으로부터 독립했다. 물론 현재는 엄연한 하나의 Matcher 라이브러리 프로젝트로서 독자적으로 발전해나가고 있다. Hamcrest 라이브러리를 단위 테스트 프레임워크와 함께 사용하면 테스트 케이스 작성 시에 문맥적으로 좀 더 자연스러운 문장을 만들어준다. 좀 더 생활언어에 가까운 테스트 케이스 구문이 만들어지고, 소스를 보는 사람이 편하게 읽을 수 있게 되며, 테스트 케이스 작성자의 의도가 더욱더 명확하게 드러나게 된다. 이런 이유로 시간이 지나면서 점차 많은 사람이 테스트 케이스 작성 시에 Hamcrest Matcher 라이브러리를 사용하는 추세다. 예전에는 JUnit과 같은 테스트 프레임워크를 사용할 때, 곁들여 함께 사용하는 라이브러리였는데, 인기 덕분인지 JUnit 버전 4.4부터는 아예 JUnit 배포 라이브러리 안에 포함되어 함께 배포된다. 앞으로 더 많은 프레임워크나 라이브러리, 혹은 프로그램 언어들이 Hamcrest 라이브러리가 추구하는 방식을 따르게 되리라 예상된다. 개발자만이 읽을 수 있는 프로그래밍 언어라는 느낌보다는 좀 더 문장체에 가까운 느낌으로 넓은 범주의 사람들이 함께 이해할 수 있는 형태로 만들어주기 때문이다. 현재 Hamcrest는 Java 외에도 C++, Objective-C, Python 그리고 PHP 버전으로도 포팅되어 있다.



Hamcrest Project 사이트, <http://code.google.com/p/hamcrest/>

Hamcrest 라이브러리는 기본적으로 assertEquals 대신에 assertEquals이라는 구문 사용을 권장한다. 공학적인 느낌을 주는 딱딱한 assertEquals보다는 assertEquals이 좀 더 문맥적인 흐름을 만들어준다고 여기기 때문이다.

```
| assertEquals( "YoungJin", customer.getName() );
```

```
| assertEquals( customer.getName(), is("YoungJin") );
```

두 문장의 차이점이 느껴지는가? 어쩌면 그다지 별반 다르지 않다고 느낄 수도 있다. 하지만 조금만 더 관심을 갖고 살펴보면 문장의 흐름이라는 측면에서는 아래쪽 구분이 더 자연스럽다는 걸 알 수 있다.¹⁶ 이렇듯 자연스런 문장을 통해서 테스트 비교구문을 만드는 것이 Hamcrest의 사용 목적이다.

¹⁶ 모국어가 영어가 아닌지라 크게 와 닿진 않을 수 있다.

자, 그럼 `assertThat`의 일반적인 사용 방법은 어떤지 한번 살펴보자.

```
assertThat(테스트대상, Matcher구문);
assertThat("메시지", 테스트대상, Matcher구문);
```

`assertEquals`와 마찬가지로 메시지는 선택사항이다. 이제 추가적으로 간단한 몇 가지 예제 문장을 더 살펴보자.

Hamcrest 적용

적용 전	<code>assertEquals(100, account.getBalance());</code>
적용 후	<code>assertThat(account.getBalance(), is(equalTo(10000)));</code>
적용 전	<code>assertNotNull(resource.newConnection());</code>
적용 후	<code>assertThat(resource.newConnection(), is(notNullValue()));</code>
적용 전	<code>assertTrue(account.getBalance() > 0);</code>
적용 후	<code>assertThat(account.getBalance(), is(greaterThan(0)));</code>
적용 전	<code>assertTrue(user.getLoginName().indexOf("Guest") > -1);</code>
적용 후	<code>assertThat(user.getLoginName(), containsString("Guest"));</code>

위 예제는 일반적인 JUnit의 `assertEquals` 계열을 Hamcrest의 `assertThat` 계열로 바꾸면 어떻게 되는지를 보여주고 있다. 잠시 뒤에 좀 더 자세히 설명하겠지만 `is`, `equalTo`, `greaterThan` 등의 메소드가 `Matcher` 구문에 해당한다. `Matcher` 구문(메소드)은 기본적으로 `static`으로 선언되어 있고, 리턴값은 `Matcher` 클래스로 되어 있다. 프로그래밍 문장 자체만 보면, Hamcrest 사용 유무 여부가 큰 차이가 없어 보이지만, 전반적으로 좀 더 읽기 편한 자연어(natural language)¹⁷에 가깝게 바뀐다.

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
```

¹⁷ 프로그래밍 언어가 아닌 일상언어나 문장체를 지칭한다.

```
import org.junit.Test;

public class HamcrestTest {
    @Test
    public void testArray() throws Exception {
        assertThat("Start Date 비교", "2010/02/03", is("2010/02/04"));
    }
}
```

위 소스는 JUnit 4에 포함된 Hamcrest 라이브러리를 사용한 소스다. `assertThat`은 `org.junit.Assert` 클래스 내에 존재하고, `is`를 비롯한 `Matcher`들은 `org.hamcrest.CoreMatchers` 클래스를 통해 호출한다. 앞에서 JUnit 4에 대해 설명할 때도 이야기했지만, 이클립스 자동완성 기능의 도움을 바로 받을 수 없기 때문에, 위와 같이 `static import`로 사용할 클래스를 직접 적어주거나 사전에 Favorite 기능으로 추가해놓아야 한다. 정리해서 말하면, JUnit 4에서 `assertThat`을 사용하려면 위 소스의 첫 두 줄에 해당하는 부분이 반드시 `static import`되어 있어야 한다는 이야기다.

실행 결과

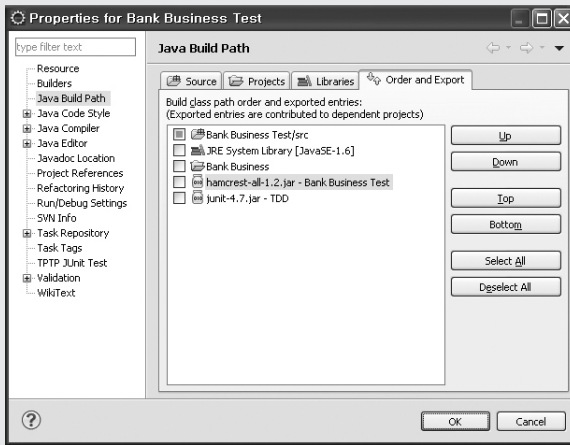
```
Java.lang.AssertionError:
Expected: is "2010/02/04"
got: "2010/02/03"
```

Hamcrest 라이브러리를 사용했을 때의 기본 실패 메시지는 위와 같다. “예상은 ‘무엇’인데, 실제로는 ‘무엇’이 나왔음”의 형태다. 좀 더 문맥적 흐름이 자연스러워서 받아들이기 더 편하다(물론 영어실력이 뒷받침되어 있어야 한다. 에휴…).

JUnit 4 내부적인 이야기를 조금 더 하자면, JUnit 4는 Hamcrest 라이브러리를 자연스럽게 사용하기 위해 Hamcrest의 `assertThat`을 `org.junit.Assert` 클래스에서 일부 다시 구현하고 있다. 그래서 JUnit 사용자 입장에서는 Hamcrest 라이브러리의 존재유무를 알지 못해도 `assertThat` 메소드를 자연스럽게 사용할 수 있게 했다. 물론 `is` 같은 Hamcrest 고유 `Matcher` 메소드들은 Hamcrest 쪽을 라이브러리를 사용하는 걸로 놔두고 있다.

클래스패스 내에 존재하지만 인식 안 되는 Hamcrest 라이브러리?

경우에 따라서는 최신 Hamcrest 라이브러리를 다운로드 받아서 클래스패스 내에 갖다 놓더라도 정상적으로 동작하지 않을 때가 있다. 왜냐하면 JUnit 라이브러리 내에도 Hamcrest 라이브러리가 이미 들어 있기 때문이다. 개뿔참외도 먼저 맑은 놈이 임자라고 했던가? JUnit 라이브러리 쪽에 포함된 Hamcrest의 클래스들이 먼저 인식되어 있기 때문에, 나중에 추가된 라이브러리를 인식하지 못하는 것이다. 이런 경우가 가끔씩 발생하기 때문에, 이클립스에서 외부 모듈을 갖다 쓸 경우에는 해당 문제가 발생하지 않도록 다소 주의를 기울일 필요가 있다. 만일 클래스패스 내에 Hamcrest 라이브러리가 존재함에도 특정 클래스나 메소드를 인식하지 못한다면, 클래스패스 우선순위를 따져보자. 의심스러운 경우 원하는 클래스의 로딩이 먼저 일어날 수 있도록 우선순위를 강제로 변경시켜 준다. 이클립스에서는 프로젝트 메뉴의 Properties 탭의 Java Build Path에서 우선순위를 임의로 조정해준다. 아래 그림은 Hamcrest 라이브러리를 JUnit 라이브러리보다 상위로 올려놓은 모습이다.



사실 이런 식의 PATH 우선순위 관련 문제들이 단지 Java 클래스패스 경우뿐 아니라, 일반적인 PATH 관련 부분에서 종종 발생한다. 이를테면, 윈도우용 오라클을 설치하면 실행 패스 내에서 오라클 설치 시에 사용되는 Java 런타임 1.4 버전이 떡! 하니 시스템패스 맨 앞으로 잡히게 된다. 누가 이런 걸 예상했겠는가? 그냥, 어느 날 어느 순간인가부터는 자신의 시스템 기본 Java 런타임환경이 1.4로 바뀌는 원인불규명 상황에 빠지게 돼버릴 뿐이다. 그러니, 컴퓨터의 세계에는 심령술과 동급 레벨인 미지의 영역이 한자리 차지하고 앉아 있는 것이다. 지금도 그러는지 모르겠지만, 예전엔 전산실 앞에서 돼지머리 놓고 시스템 장애가 발생하지 않게 해달라고 고사를

지내는 모습도 몇 번 봤다. 그 당시엔 그게 참 어이없고 아이러니했는데, 지나고 보니 꼭 그렇지 않은 않더라. 이야기가 너무 나갔다고 생각하지만, 몇 마디 덧붙이자면 필자는 컴퓨터가 거짓말을 안 한다고 생각한다. 모든 일에는 이유가 있지만, 밝혀내지 못하는 것뿐이라고 말이다. 하지만 때때로 이런 생각이 들곤 한다. 컴퓨터는 거짓말은 안 하지만, 종종 시침 뚝 떼고 눈 멀뚱멀뚱한 모습으로 사람을 속이긴 한다고 말이다.

Hamcrest 라이브러리를 한번 자세히 살펴보자. JUnit 4.7이나 4.8에 기본으로 탑재되어 있는 Hamcrest 버전은 1.1 core 버전이다. Hamcrest가 제공하는 클래스와 메소드의 수가 적지 않다 보니, 버전별로 그리고 패키지별로 따로 제공하고 있다. 여기서는 여러 패키지를 포함하고 있는 1.2 all 버전을 기준으로, 많이 사용하는 Matcher 위주로 살펴볼 예정이다. 따라서 현재 이클립스에 기본으로 탑재되어 있는 JUnit 4.5 버전 안에 들어 있는 Hamcrest와는 조금 차이가 날 수 있다. 하지만 차기 이클립스에는 JUnit 상위 버전과 함께 Hamcrest도 상위 버전이 탑재될 예정이라고 하니, 미리 살펴본다고 생각하자. 그리고 Hamcrest 1.2 버전에 좀 더 다양한 기능이 추가되어 있기 때문에, 이왕 사용할 거면 1.2를 내려받아서 사용할 것을 권장한다. 필요에 따라 Hamcrest 사이트에서 해당 버전을 내려받아서 클래스패스 내에 추가해서 사용하면 되겠다.

Hamcrest 라이브러리는 기본적으로 core 패키지와 그 외의 확장 패키지로 구성되어 있다.

패키지	설명
org.hamcrest. core	오브젝트나 값들에 대한 기본적인 Matcher들
org.hamcrest. beans	Java 빈(Been)과 그 값 비교에 사용되는 Matcher들
org.hamcrest. collection	배열과 컬렉션 Matcher들
org.hamcrest. number	수 비교를 하기 위한 Matcher들
org.hamcrest. object	오브젝트와 클래스들 비교하는 Matcher들
org.hamcrest. text	문자열 비교
org.hamcrest. xml	XML 문서 비교

다음은 분류에 따른 Matcher의 종류를 좀 더 자세히 표시해봤다. 단, 패키지 기준으로 분류한 게 아니라 Matcher의 종류에 따라 분류해놓았기 때문에, 분류가 곧 패키지와 동일하진 않다는 점에 유의하자. 이를테면 allOf 같은 경우엔 논리적 비교에 가깝지만, 실제 들어 있는 것은 core 패키지 내에 포함되어 있다. 패키지 위치보다는 어떤 종류의 Matcher를 사용할 수 있는지 중심으로 살펴보자. 대부분은 클래스를 직접 import 하지 않고 org.hamcrest.CoreMatchers나 org.hamcrest.Matchers 클래스만 static import 처리하면 호출할 수 있도록 되어 있다. 또한 여기 소개된 것 외에도 추가로 제공되는 Matcher도 있지만, 원 저작자가 밝힌 자주 사용되는 대표적인 Matcher 위주로 소개한다.

코어(Core)

메소드	설명	클래스명
anything	어떤 오브젝트가 사용되든 일치한다고 판별한다.	IsAnything
describedAs	테스트 실패 시에 보여줄 추가적인 메시지를 만들어주는 메시지 데코레이터	DescribedAs
equalTo	두 오브젝트가 동일한지 판별한다.	IsEqual
is	내부적으로 equalTo와 동일하다. 가독성 증진용. 즉, 아래 세 문장은 의미가 동일하다. assertThat(entity, equalTo(expectedEntity)); assertThat(entity, is(equalTo(expectedEntity))); assertThat(entity, is(expectedEntity));	Is

오브젝트(Object)

메소드	설명	클래스명
hasToString	toString 메소드의 값과 일치 여부를 판별한다. assertThat(account, hasToString("Account"));	HasToString
instanceOf typeCompatibleWith	동일 인스턴스인지 타입 비교(instance of). 동일하거나 상위 클래스, 인터페이스인지 판별	InstanceOf IsCompatibleType
notNullValue nullValue	Null인지, 아닌지를 판별	IsNotNull
sameInstance	Object가 완전히 동일한지 비교. equals 비교가 아닌 ==(주소 비교)로 비교하는 것과 동일	IsSame

논리(Logical)

메소드	설명	클래스명
allOf	비교하는 두 오브젝트가 각각 여러 개의 다른 오브젝트를 포함하고 있을 경우에, 이를테면 collection 같은 오브젝트일 경우 서로 동일한지 판별한다. Java의 숏서킷(&& 비교)과 마찬가지로 한 부분이라도 다른 부분이 나오면 그 순간 false를 돌려준다.	AllOf
anyOf	allOf와 비슷하나 anyOf는 하나라도 일치하는 것이 나오면 true로 판단한다. Java의 숏서킷()과 마찬가지로 한 번이라도 일치하면 true를 돌려준다.	AnyOf
not	서로 같지 않아야 한다.	IsNot

빈즈(Beans)

메소드	설명	클래스명
hasProperty	Java 빈즈 프로퍼티 테스트	HasProperty

컬렉션(Collection)

메소드	설명	클래스명
array	두 배열 내의 요소가 모두 일치하는지 판별	IsArray
hasEntry, hasKey, hasValue	맵(Map)요소에 대한 포함 여부 판단	IsMapContaining
hasItem, hasItems	특정 요소들을 포함하고 있는지 여부 판단	IsCollectionContaining
hasItemInArray	배열 내에 찾는 대상이 들어 있는지 여부를 판별	IsArrayContaining

숫자(Number)

메소드	설명	클래스명
closeTo	부동소수점(floating point) 값에 대한 근사값 내 일치 여부, 값(value)과 오차(delta)를 인자로 갖는다.	IsCloseTo
greaterThan greaterThanOrEqualTo	값 비교. >, >=	OrderingComparison
lessThan lessThanOrEqualTo	값 비교. <, <=	OrderingComparison

텍스트(Text)

메소드	설명	클래스명
containsString	문자열이 포함되어 있는지 여부	StringContains
startsWith	특정 문자열로 시작	StringStartsWith
endsWith	특정 문자열로 종료	StringEndsWith
equalTolgnoringCase	대소문자 구분하지 않고 문자열 비교	IsEqualIgnoringCase
equalTolgnoringWhiteSpace	문자열 사이의 공백 여부를 구분하지 않고 비교	IsEqualIgnoringWhiteSpace

보통 JUnit 같은 xUnit 계열 프레임워크에서는 테스트의 성공/실패를 판단하는 데 있어 기본적으로 1:1 값 비교가 기반이 된다. 예상값과 실제 수행값, 두 값을 놓고, 동등(equal), 등가(same value), 비교우위(larger, smaller) 등의 단순한 연산을 이용해 테스트 결과를 판별한다. 경우에 따라서는 이런 기본적인 연산만으로는 해당 테스트 자체의 참/거짓을 판별하기가 다소 불편할 때가 있는데, 위에서 소개한 Matcher를 사용하면 그런 경우에도 좀 더 편리하게 값을 비교할 수 있다.

이 외에도 많은 Matcher가 있으며, 계속 추가되는 추세다. 한 가지 아쉬운 점은 정적(static)으로 되어 있는 matcher 메소드의 이름과 소속 클래스의 이름을 직관적으로 생각해내기 어려운 경우가 종종 있다는 점이다. 메소드 자체와 클래스 이름에 집중하기보다는 해당 라이브러리가 지원해주는 메소드의 종류를 살펴보는 데 좀 더 관심을 기울이길 권한다. 굳이 외울 필요까지는 없고, 일반적인 수준에서 이러저러한 기능을 제공하는구나 하는 정도에서 이해하면 될 듯싶다. 그래야 하는 이유 중 하나로, Hamcrest에는 묘한 클래스가 하나 있는데, Matchers라는 클래스다. 다음은 Matchers라는 클래스의 소스 일부다.

```

package org.hamcrest;

public class Matchers {

    public static <T> org.hamcrest.Matcher<T> allOf(...) {
        return org.hamcrest.core.AllOf.<T>allOf(first, second, third,
            fourth);
    }

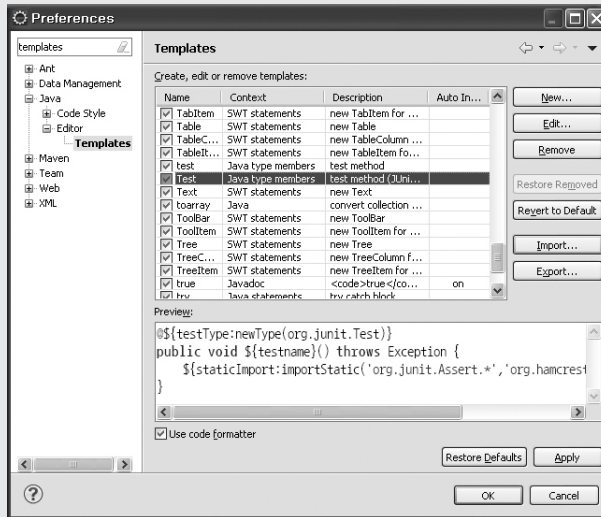
    public static <T> org.hamcrest.core.AnyOf<T> anyOf(...) {
        return org.hamcrest.core.AnyOf.<T>anyOf(matchers);
    }

    public static <T> org.hamcrest.Matcher<T> is(org.hamcrest.Matcher<T>
        matcher) {
        return org.hamcrest.core.Is.<T>is(matcher);
    }
    ...
    ...
}

```

Hamcrest 라이브러리는 사용자의 편의를 위해 일종의 프론트 매처 클래스(Front Matcher Class)에 해당하는 Matchers라는 클래스를 만들어놓았다. 이 클래스의 메소드들은 앞에서 설명한 Matcher 메소드들과 이름이 중복되는데, 내부를 살펴보면 원래 클래스들의 메소드를 호출하도록 위임형태로 작성되어 있다. 따라서 Hamcrest를 사용할 때는 org.hamcrest 패키지의 Matchers 클래스만 static import 처리해놓으면 대부분의 Matcher 메소드를 편하게 사용할 수 있다. 그마저도 불편하다면 JUnit 테스트 템플릿을 고쳐서 자동으로 Matchers 클래스가 import 될 수 있게 만들어놓으면 된다.

Hamcrest 라이브러리를 JUnit 테스트 케이스 작성 시 템플릿으로 지정해놓기



이클립스 Preferences 페이지에서 Java → Editor → Templates를 선택한다. Templates 종류 중에서 Test라고 되어 있는 항목을 선택하고 Edit를 선택한다. 다음과 같이 importStatic에 org.junit.matchers.JUnitMatchers.*를 추가한다.

```

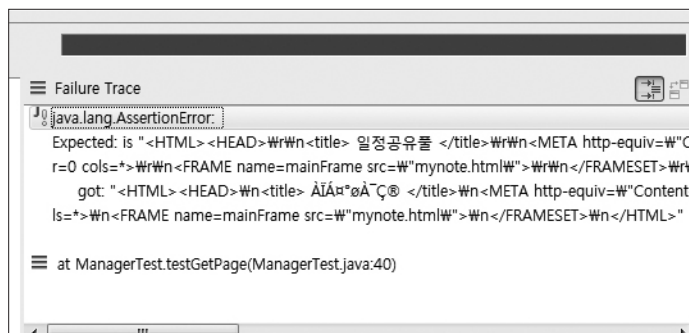
@${testType:newType(org.junit.Test)}
public void ${testname}() throws Exception {
    ${staticImport:importStatic('org.junit.Assert.*',
    'org.hamcrest.Matchers.*')}${cursor}
}
    
```

주의! 이클립스 갈릴레오 버전에는 JUnit 4.5가 탑재되어 있고, Hamcrest는 1.1 버전이 들어 있는데, 해당 버전에서는 org.hamcrest.Matchers가 존재하지 않는다. Hamcrest 최신 버전을 내려받지 않고 현재 포함된 그대로 사용하려면 org.hamcrest.Matchers.* 대신에 org.hamcrest.CoreMatchers.*를 템플릿에 넣어야 한다.

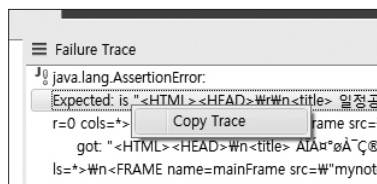
현재 Hamcrest 라이브러리는 원 저자의 의도와는 별개로, 비교가 핵심이 되는 테스트 프레임워크에서 많이 쓰이고 있으며, 향후에는 일종의 텍스트용 정규식(regular

expression)처럼 일반적으로 쓰일 가능성이 높다. 앞으로는 예제에서 Hamcrest를 이용한 표현을 때때로 쓸 예정이니 그때그때 형태를 살펴보는 걸로 하자. 그런데 Hamcrest 라이브러리가 항상 편하기만 한 것은 아니다. 때에 따라서는 Hamcrest를 사용하는 것이 좀 더 불편하게 느껴질 때도 있다. 아래는 Hamcrest 비교를 사용한 경우다.

```
@Test
public void testGetPage() throws Exception {
    manager.connect("www.doortts.com/csw /");
    assertThat(manager.getPage(), is(expectedString));
}
```



Hamcrest의 기본 실패 추적 메시지(Failure Trace) 형식에 따라 Expected와 got이 위와 같이 표현되고 있다. 하지만 위 실패 메시지만으로는 차이점이나 원인 등을 제대로 파악하기 어렵다. 결국 해당 트레이스(Failure Trace)를 복사한 다음 다른 텍스트 비교 툴(diff tool)을 써서 비교할 수밖에 없다. 여간 불편한 게 아니다.

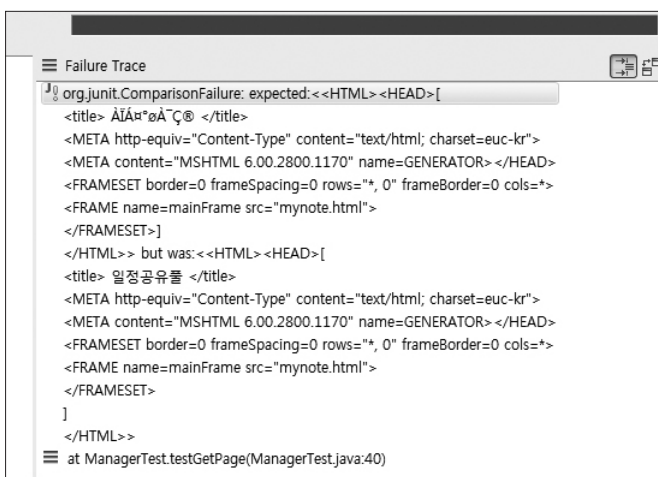


그나마 마우스 오른쪽 버튼을 눌러서 텍스트 메시지를 복사(Copy Trace)하는 건 가능하다.

다음은 `assertThat ~ is` 표현을 제거하고 `assertEquals`만을 사용한 경우다.

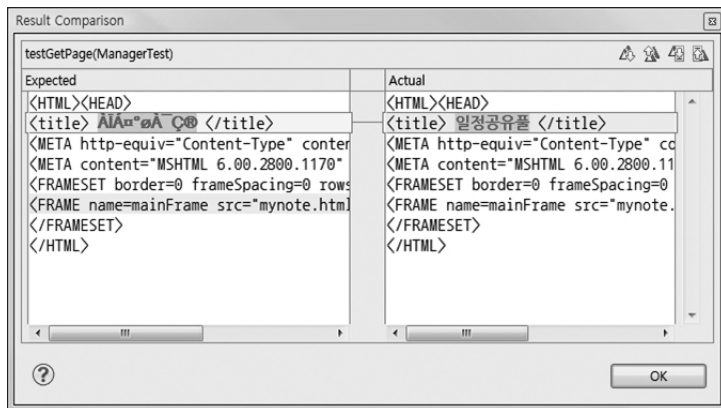
```
@Test
public void testGetPage() throws Exception {
    manager.connect("www.doortts.com/csw/");
    assertEquals(manager.getPage(), expectedString );
}
```

테스트 내용 자체는 동일하기 때문에 동일한 실패가 발생한다.



`assertEquals`를 사용했을 때의 JUnit의 실패 추적 메시지

위 그림은 `assertEquals`를 사용해 비교했을 때의 실패 메시지다. 예상값(expected)과 차이가 나는 부분을 []로 감싸서 보여주고 있다. 거기에는 이번엔 이클립스에서 자체 비교 툴을 사용해서 좀 더 가독성 높은 형태로 볼 수도 있다. 이클립스에서 실패 추적 메시지(Failure Trace) 부분을 마우스로 더블클릭해보자. 다음과 같은 결과비교 창이 바로 뜬다.



아! 인코딩 문제였구나! IDE 기능으로 인해 상쇄돼버리는 hamcrest 실패 구문 설명의 장점

좀 아쉽긴 하지만, 이건 이클립스라는 강력한 통합개발환경으로 인해 Hamcrest 실패 구문 설명(failure description)의 장점이 다소 퇴색된 부분이다. 향후에는 이클립스가 Hamcrest의 메시지도 비교하는 기능을 지원해주길 기대한다.

중급 이상 도전!

사용자 정의 Matcher 만들기

만일 Hamcrest 라이브러리에서 제공하는 Matcher 외의 자신만의 비교 구문 Matcher 를 만들고 싶다면, `TypeSafeMatcher`를 상속받아서 `matchesSafely`, `describeTo`를 재 정의하면 된다.

`TypeSafeMatcher`를 상속받은 `IsNotANumber` 클래스

```
import org.hamcrest.Description;
import org.hamcrest.Factory;
import org.hamcrest.Matcher;
import org.hamcrest.TypeSafeMatcher;

public class IsNotANumber extends TypeSafeMatcher<Double> {

    @Override
    public boolean matchesSafely(Double number) {
```



```

        return number.isNaN();
    }

    public void describeTo(Description description) {
        description.appendText("not a number");
    }

    @Factory
    public static <T> Matcher<Double> notANumber() {
        return new IsNotANumber();
    }
}

```

사용자가 직접 정의해서 만든 새로운 Matcher인 notANumber의 실제 사용 방식

```

public void testSquareRootOfMinusOneIsNotANumber() {
    assertThat(Math.sqrt(-1), is(notANumber()));
}

```

2.3 정리

이번 장에서는 단위 테스트 프레임워크의 기본이 되는 JUnit과 이와 함께 사용하는 Matcher 라이브러리인 Hamcrest에 대해 살펴봤다. 만일 JUnit 3과 JUnit 4 중에서 하나만 알아야 한다면 당연히 JUnit 4를 추천한다. 그리고 Hamcrest 라이브러리는 쓸 수 있다면 최대한 쓸 것을 권장한다. 비교표현 구문도 간결해지고, 테스트 케이스의 의미도 좀 더 명확하게 만들어준다.

만일 2장의 주제에 대해 추가적인 학습을 원한다면 junit.org 사이트의 뉴스와 기사들을 살펴보길 권한다. 또한 JUnit 라이브러리 구조 자체에 관심이 있다면, <http://junit.sourceforge.net/doc/cookstour/cookstour.htm>에서 찾아볼 수 있다. 어떤 디자인 패턴이 쓰였고, 어떤 원리로 동작하는지 자세히 설명하고 있다.



좋은 코드? 나쁜 코드?

좋은 소스와 그렇지 않은 소스를 구분하는 방법은 무엇일까? 여러 가지 기준이 있을 수 있겠지만, 가장 쉬운 권장 방법은 어떤 코드가 자연스럽게 읽힐 수 있는지 살펴보는 것이다. 무슨 말인가 하면, 소스코드를 볼 때 한 줄 한 줄의 의미를 생각하며 따라가는 식으로 '분석한다'는 느낌이 들면 좋지 않은 소스, 소설책 읽듯이 문맥을 갖고 '쉽게 읽힌다'면 좋은 소스로 구분하는 것이다. 우리가 '소스코드'라는 걸 작성하는 목적이자, 프로그래밍 언어가 계속 발전하는 중요한 이유 중 하나는, 컴퓨터와 좀 더 잘 대화하기 위해서가 아니라 다른 사람들과 좀 더 잘 대화하기 위해서라는 사실을 잊지 말자.

김창준 애자일 컨설팅 대표

실제 TDD를 적용하고 전파하고 계신 전문가 분들께 인터뷰 요청을 드렸습니다. '책 안의 지식'이 아닌 '살아있는 기술로써의 TDD'에 대한 의견을 묻고 듣고 나누고 싶었기 때문입니다. 인터뷰 내용은 가감 없이 원문 그대로 실었습니다. 흠, 어색한 문장이 나 통일되지 않은 단어가 보이더라도 너그러운 마음으로 양해 부탁드립니다. 이 인터뷰는 김창준, 변신철, 황상철, 안영희, 박재성, 이일민님 순으로 진행됩니다.

Q. 안녕하십니까? 대표님. 인터뷰에 응해주셔서 감사합니다. 대표님을 아직 잘 모르는 분들을 위해 간략한 소개 부탁드립니다. 그와 함께 현재 하고 계신 일이나 최근 관심사에 대해 이야기해주실 수 있으신지요?

A. 네. 저는 애자일 컨설팅의 대표 김창준입니다. 저희 회사는 애자일 방법론을 조직과 개인에게 컨설팅 및 코칭하는 회사입니다. 최근에는 애자일 코치를 키우는 과정(AC2, <http://ac2.kr>)에 모든 노력을 기울이고 있습니다. 우리나라에 애자일이 퍼지고 성숙하려면 수준 높은 애자일 코치들이 생기지 않으면 불가능하다는 생각에 일종의 사명감을 갖고 노력하고 있습니다.

Q. 대표님께서 TDD 책을 번역해서 국내에 본격적으로 소개한 지도 적지 않은 시간이 지났습니다. 하지만 TDD는 여타 기술이나 기법에 비해 확산이 느린 것처럼 보입니다. 이유가 무엇이라고 생각하시는지요? 혹시 TDD의 가치가 크지 않아서 확산이 더딘 건 아닌지요?

A. 몸에 좋은 약은 쓴 법입니다. TDD는 개발자에게 체질적 변화를 요구합니다. 그래서 생각만큼 확산이 빠르지 않습니다. 하지만 점진적인 변화는 계속 벌어지고 있으며, 100도를 넘으면 산업 전체의 체질적 변화가 생길 것이라 예측합니다. 실제로 국내 성공 사례를 보면 2차원에서 3차원으로의 도약 같은 엄청난 변화가 있었습니다. 그리고 TDD 도입에 어려움을 겪는 개인과 조직은 TDD를 좀 더 넓은 의미로 그리고 원칙(결정과 피드백 사이의 갭을 조정) 중심으로 생각해볼 필요가 있습니다.

Q. 혹 기억에 남는 TDD에 대한 경험담이나 이야기를 소개해주실 수 있으신지요? 좋은 기억이 아니어도 무방합니다.

A. 제가 코칭한 프로젝트에서 TDD의 효과에 대한 실험을 한 적이 있습니다. C 언어로 개발했던 금융거래 프로젝트였습니다. TDD로 개발한 경우 그렇지 않은 코드에 비해 모듈별 개발 시간이 15% 증가했습니다. 반면 TDD로 개발한 코드에서는 결함이 60% 줄었습니다. 이 프로젝트에서 TDD를 하는가 안 하는가의 선택은 결국, 조금 빨리 개발하면서 버그는 2.5배 늘어나는 방식(TDD로 개발하지 않는 방식)을 택하겠는가 말겠는가의 문제였습니다.

Q. 초급 개발자가 TDD를 적용할 때 집중해야 할 부분이나 유의해야 할 부분은 무엇이라고 생각하십니까?

A. 모든 단위 테스트가 통과하는 시점에서 다시 모든 단위 테스트가 통과하는 시점까지 걸리는 시간을 GBC(Green Bar Cycle)라고 저는 부릅니다. 이 GBC를 짧게 유지하는 것이 첫 번째 혈자리이고, 두 번째 혈자리는 시스템의 핵심을 끝에서 끝까지 간단하게 관통하는 테스트가 가급적 일찍 나오게 하는 것입니다. 이 두 가지 혈자리를 잘 보존하면 TDD를 성공적으로 할 수 있습니다.

Q. 강규영님과 함께 TDD를 통해 Line Reader를 만들어가는 동영상*은, 지금까지도 많은 이들이 참고하는 소중한 자료 중 하나입니다. 혹시 그 동영상에 대해 아쉬운 점이나, 생각이 변해서 지금이라면 좀 다르게 접근할 것 같은 부분은 없으신지요?

A. 촬영한 것이 2003년입니다. 웹 채팅 서비스를 TDD로 구현했는데, 그중에서 일부분을 보여드렸죠. 재미있게 잘했다고 생각합니다. 아쉬운 점도 몇 가지가 있지만 하나만 꼽는다면 위에서 언급한 두 번째 혈자리를 잘 지키지 못했다고 생각합니다. 예를 들자면 아무것도 없을 때(Empty)를 첫 번째 테스트 케이스로 잡았는데, 지금이라면 그것보다 뭔가 있는 케이스를 먼저 잡고 전체를 가볍게 관통한 다음, 부분들을 키워나가는 식을 시도하겠습니다.

Q. 만일 개발을 잘 모르는 관리자나 경영층 인사가 “TDD가 뭔가요?”라고 묻는다면 어떻게 대답하십니까?

A. 일단 그런 경우가 많지가 않았습시다만, 요즘은 이렇게 시작합니다. “개발 기간은 조금 늘어나지만 결함을 획기적으로 줄일 수 있는 개발 기법인데...”

Q. 굳이 TDD 관련 내용이 아니더라도, 좋은 소프트웨어를 만들고 싶어하는 IT 개발자들에게 해주고 싶으신 이야기나 조언은 없으신지요?

A. 다음 세 가지 질문을 자신에게 묻고 솔직하게 대답해볼 것을 권합니다.

- 내가 어떻게 해서 이 자리에 있게 되었는가?
- 내가 이 자리에 있는 것에 대해 어떻게 느끼는가?
- 나는 어떤 일이 일어나기를 원하는가?

Q. 귀한 시간을 내어 인터뷰에 응해주셔서 감사합니다.

A. TDD를 소개하는 좋은 책을 써주셔서 감사합니다. 이 책을 통해 우리나라에 TDD가 더 많이 퍼지고, 개발자들이 스트레스를 덜 받고 자신의 작업에 대해 뿌듯함을 느끼며 일하게 되기를 기원합니다.

* 현재 해당 동영상은 XPer 커뮤니티의 위키 사이트(<http://xper.org/LineReaderTdd/>)에서 찾아볼 수 있습니다.