소프트웨어 공학 개론

Tutorial #2: Junit

Eun Man Choi
emchoi@dgu.ac.kr

강의 목표

- JUnit 소개
- 테스트 케이스
- Assertion
- JUnit 테스트 실행

JUnit

- Java 언어를 위한 단위 테스팅 프레임워크
  - 저자: Erich Gamma, Kent Beck

- 목적:
  - “테스트를 생성하고 실행하기 쉽다면 프로그래머가 테스트를 생성하고 실행하도록 마음을 움직일 것이다.”

소개

- 테스트를 자동화 하기 위하여 무엇이 필요한가?

- 테스트 스크립트
  - 테스트 대상 시스템(SUT)에 보내는 액션
  - SUT 에 예상되는 반응
  - 테스트가 성공이냐 실패냐를 결정하는 방법

- 테스트 실행 시스템
  - 스크립트를 읽어 테스트 케이스를 SUT에 연결시키는 메카니즘
  - 테스트 결과를 추적

테스트 케이스 판결

- 판결(verdict)은 단일 테스팅을 실행한 결과의 선언
- 성공(Pass): 테스트 케이스가 의도한 목적을 이루고 테스트 대상이 예상되는 결과를 수행함
- 실패(Fail): 테스트 케이스가 의도한 목적을 성취하였으나 테스트 대상이 예상된 대로 수행하지 못함
- 오류(Error): 테스트 케이스가 의도한 목적을 성취하지 못함
  - 이유:
    - 테스트 케이스 수행도중 예상하지 못한 이벤트 발생
    - 테스트 케이스가 적절히 셋업되지 않음

JUnit 버전

- 2007년 3월 부터 현재의 버전 4.3.1
  - JUnit 4.x를 사용하려면 Java version 5나 6을 사용하여야
- 2006년 4월에 소개된 JUnit 4는 이전 버전에서 상당한 변화(호환성 없음)
- JUnit 4를 소개
- 대부분의 JUnit 문서와 사례는 현재 JUnit 3를 사용하여 상당히 다름
  - JUnit 3은 Java (1.4.2) 낮은 버전에서 사용가능.
  - junit.org 웹 사이트는 오래된 버전의 요구가 없는 한 JUnit version 4을 사용
  - Eclipse (3.2)는 플러그인으로 두 가지 버전 JUnit 3.8나 JUnit 4.1 중 고를 수 있게 함

JUnit 테스트란?

- 테스트 스크립트는 Java 메소드의 모임
  - 일반적인 아이디어는 Java 객체를 생성하고 이것으로 뭔가 관심 있는 일을 하고 객체가 올바른 특성을 가지고 있는지 결정하는 것

- 추가할 것은? Assertions.
  - 여러 가지 특성을 체크하기 위한 메소드의 패키지:
    - 객체의 동질성"equality"
    - 동일한 객체 참조
    - null / non-null 객체 참조
  - assertions은 테스트 케이스 판결을 결정하기 위하여 사용됨

Junit은 언제 사용하여야 하나?

- 이름에 있는 것처럼...
  - 적은 분량의 단위 테스팅을 위하여
- 즉 복잡한 테스팅이나 시스템 테스팅은 거리가 멈
- 테스트 중심 개발 방법에서 JUnit 테스트는 코드 개발 전에 테스트가 먼저 작성되어 실행됨
  - 구현 코드는 테스트를 통과하기 위한 최소의 코드로 작성되어야 함 – 즉 추가되는 기능이 없어야
  - 코드가 일단 작성되면 계속 테스트를 실행하여 통과되어야
  - 새로 코드가 추가될 때마다 모든 테스트를 재실행하여 이상을 일으키지 않음을 증명하여야

JUnit 4 테스트 케이스

```
/** Test of setName() method, of class Value  */

@Test
public void createAndSetName()
{
    Value v1 = new Value( );

    v1.setName( "Y" );

    String expected = "Y";
    String actual = v1.getName( );

    Assert.assertEquals( expected, actual );
}
```

JUnit 4 테스트 케이스

```
/** Test of setName() method, of class Value  */

@Test
public void
{
    Value v1 = new Value( );

    v1.setName( "Y" );

    String expected = "Y";
    String actual = v1.getName( );

    Assert.assertEquals( expected, actual );
}
```

이 Java 메소드는 테스트 케이스임을
테스트 실행자에게 표시

JUnit 4 테스트 케이스

```
/** Test of setName() method, of class Value  */

@Test
public void createAndSetName()
{
    Value v1 = new Value( );

    v1.setName( "Y" );

    String expected = "Y";
    String actual = v1.getName( );

    Assert.assertEquals( expected, actual );
}
```

목적:
setName이 Value 객체에 특정 이름을 저장하였는지 확인

JUnit 4 테스트 케이스

```
/** Test of setName() method, of class Value  */

@Test
public void createAndSetName()
{
    Value v1 = new Value( );

    v1.setName( "Y" );

    String expected = "Y"
    String actual = v1.getName( );

    Assert.assertEquals( expected, actual );
}
```

Value 객체가 정말 이름을 저장하였는지 확인하기 위하여 호출

JUnit 4 테스트 케이스

```
/** Test of setName() method, of class Value  */

@Test
public void createAndSetName()
{
    Value v1 = new Value( );

    v1.setName( "Y" );

    String expected = "Y";
    String actual = v1.getName( );

    Assert.assertEquals( expected, actual );
}
```

expected와 actual 는 같기를 희망

같지 않다면 테스트 케이스는 fail

Assertions

- 가설(assertion) 은 Junit에서 Assert라는 클래스로 정의
  - assertion가 참이면 메소드의 실행은 계속됨
  - 어떤 assertion이 거짓이면 메소드의 실행은 그 자리에서 중지되고 테스트 케이스의 결과는 실패(fail).
  - 메소드 수행 중 예외가 발생하면 테스트 케이스의 결과는 error.
  - 모든 메소드에서 가설(assertion)이 위배되지 않았다면 테스트 케이스는 통과 (pass)

- 모든 assertion 메소드는 정적(static)

Assertion 메소드(1)

- 진위 조건이 true나 false
  - assertTrue(condition)
  - assertFalse(condition)

- 객체가 null 이냐 null이 아닌가?
  - assertNull(object)
  - assertNotNull(object)

- 객체가 동일한가(즉 같은 객체를 참조하고 있나) 아닌가?
  - assertSame(expected, actual)
    - true if: expected == actual
  - assertNotSame(expected, actual)

Assertion 메소드(2)

- 객체의 동일성("Equality"):
  assertEquals(expected, actual)
    - expected.equals(actual) 이면 참

- 배열의 동일성("Equality"):
  assertArrayEquals(expected, actual)
    - 배열은 같은 길이를 가져야 함
    - 모든 i의 정상적인 값에 대하여 다음을 체크:
      assertEquals(expected[i],actual[i])
      또는
      assertArrayEquals(expected[i],actual[i])

- 항상 fail로 판결되는 무조건적 실패 assertion fail()이 있음

Assertion 메소드 매개변수

- 모든 assertion 메소드는 2개의 매개 변수를 가짐. 첫 매개 변수는 예상되는 값(expected value) 두 번째 매개 변수는 실제 값(actual value)이 됨
  - 순서가 비교에는 영향이 없으나 사용자에게 실패 메시지를 생성할 때 이런 순서로 간주

- 모든 assertion 메소드는 첫 매개 변수로 추가 String 파라미터를 가질 수 있음. 스트링은 assertion 이 실패로 끝나면 failure 메시지에 포함됨
  - 예:

```
fail( message )
assertEquals( message, expected, actual)
```

동일성(Equality) assertions

- assertEquals(a,b) 는 테스트 대상 클래스의 equals() 메소드에 영향을 받음.
  - a.equals(b) 의 결과에 좌우됨
  - 동일성 관계를 결정하는 것은 테스트 대상 클래스에 달려 있음 JUnit은 적용가능한 것을 사용
  - 테스트 대상 클래스가 equals() 메소드를 재정의하지 않으면 Object 클래스로부터 받은 default equals() 메소드를 적용하여 객체 동일성을 체크.

- a 와 b 가 int, boolean, 등 기본 타입이라면 assertEquals(a,b)이 다음과 같이 동작:
  - a 와 b 가 동일 객체(Integer, Boolean, 등)로 변환된 후 a.equals(b) 가 계산됨

변동 소수점 assertions

- 변동 소수점 타입(double 이나 float)가 비교될 때는 매개변수 delta 가 추가로 필요.

- assertion 은 다음을 계산

  Math.abs(expected – actual) <= delta

  변동 소수점 비교에서 round-off 오류를 피하기 위하여

- 예:

  assertEquals(aDouble, anotherDouble, 0.0001)

JUnit 테스트의 구성

- 각 메소드는 독립적으로 판결(pass, error, fail)될 수 있는 테스트 케이스를 나타냄.
- 일반적으로 하나의 Java 클래스를 위한 모든 테스트는 분리된 클래스 안에 모여 그루핑됨.
- 코딩 스타일(예):
  - 테스트 대상 클래스: Counter
  - 테스트를 포함한 클래스: CounterTest

Junit 테스트 사례

```
public class Counter {
        int count = 0;

        public int increment() {
            return count += 1;
        }

        public int decrement() {
            return count -= 1;
        }

      public int getCount() {
          return count;
      }
}
```

- 메소드 내부가 구현되지 않았더라도 Junit 테스트 코딩 가능함
- 명세만 정해져 있어도 테스트 코딩 가능
- 메소드 스텁 이용

Counter를 위한 JUnit 테스트

```
public class CounterTest {
    Counter counter1; // declare a Counter here

    @Before
    void setUp() {
        counter1 = new Counter(); // initialize the Counter here
    }

    @Test
    public void testIncrement() {
        assertTrue(counter1.increment() == 1);
        assertTrue(counter1.increment() == 2);
    }

    @Test
    public void testDecrement() {
        assertTrue(counter1.decrement() == -1);
    }
}
```

- 각 테스트 케이스는 새로운 cointer로 시작
- 테스트 케이스 실행 순서 의미 없음

테스트 기반(fixture)

- 테스트 기반은 테스트 케이스가 실행되는 배경

- 테스트 기반은 다음을 포함:
  - 테스트 케이스에 의하여 사용될 수 있는 객체나 자원.
  - 객체를 사용할 수 있게 만들고 자원을 할당 또는 해지하는 데 필요한 작업: “setup” 과 “teardown”.

Setup과 Teardown

- 특정 클래스를 위한 테스트 셋에 대하여 각 테스트 케이스 수행에 앞서 이루어져야 하는 반복적인 작업이 있음.
  - 예: 작업하려는 "관심 대상" 객체의 생성, 예를 들면 네트워크 연결.

- 테스트 케이스 끝에는 수행 후 청소하기(객체 삭제 등) 위한 반복되는 작업이 있음.
  - 자원 할당이 해지되고 다음 테스트 케이스를 위한 상태로 바뀌었는지 확인
  - 테스트 케이스가 실패하면 테스트 메소드의 수행이 거기서 끝나게 되어 청소하는 코드는 실행될 수 없음.

Setup과 Teardown

- Setup:
  - 각 테스트 케이스 전에 수행될 메소드에는 @Before라는 주석을 사용
- Teardown (판결에 상관 없이):
  - 각 테스트 케이스 뒤에 수행될 코드를 가진 메소드에는 @After 주석을 사용
  - 이런 메소드는 테스트 케이스에서 예외가 발생하거나 assertion 이 실패되더라도 실행될 것임
- 주석의 수는 제한 없음
  - @Before 주석을 가진 모든 메소드는 각 테스트 케이스 실행 전에 수행되나 순서는 임의로.

예: 파일을 테스트 기반으로 사용

```
public class OutputTest
{
    private File output;

    @Before public void createOutputFile()
    {
       output = new File(...);
    }

    @After public void deleteOutputFile()
    {
        output.delete();
    }

    @Test public void test1WithFile()
    {
       // code for test case objective
    }

   @Test public void test2WithFile()
    {
       // code for test case objective
    }
}
```

메소드 수행 순서

1. createOutputFile()
2. test1WithFile()
3. deleteOutputFile()
4. createOutputFile()
5. test2WithFile()
6. deleteOutputFile()

- assertion: test1WithFile 이 test2WithFile보다 먼저 실행된다는 보장은 없음

단 한 번의 setup

- 전체 테스트 클래스를 위하여 다른 테스트 전에, 다른 어떤 @Before 메소드보다 앞서 메소드를 한 번만 수행 시킬 수 있는 방법이 있음

- 서버를 기동시키거나 통신을 개시하는데 유용함. 테스트할 때마다 닫고 다시 여는 것은 시간이 많이 걸리기 때문

- @BeforeClass 주석으로 표시(메소드 하나에서만 사용 가능이며 static 이 되어야 함):

```
@BeforeClass public static void anyNameHere()
{
   // class setup code here
}
```

단 하나의 tear down

- 단 하나의 청소 메소드도 가능. 클래스 안에 있는 모든 테스트 케이스가 수행된 후, 어떤 @After 메소드 후에 수행됨

- 서버를 종료시키거나 통신 연결을 끊을 때 사용.

- @AfterClass 주석으로 표시(하나의 메소드에만 사용 가능하며 static이 되어야):

```
@AfterClass public static void anyNameHere()
{
   // class cleanup code here
}
```

예외 테스팅(1)

- @Test 주석에 파라미터를 추가하면 특정 예외가 테스트 중에 발생할 수 있음을 예측.

```
@Test(expected=ExceptedTypeOfException.class)
public void testException()
{
   exceptionCausingMethod();
}
```

- 예외가 발생하지 않거나 예측하지 못한 예외가 발생한다면 테스트는 실패.
  - 즉 메소드의 종료 시점에 예외가 발생하지 않으면 테스트 케이스는 실패임.

- 예외 메시지 내용을 테스트하거나 예외가 발생되는 범위를 제한하는 것은 다음 슬라이드에 있는 방법을 사용

예외 테스팅(2)

- 예외 Catch, 예외 발생이 없으면 fail( ) 사용

```
public void testException()
{
    try
    {
        exceptionCausingMethod();

        // If this point is reached, the expected
        // exception was not thrown.

        fail("Exception should have occurred");
    }
    catch ( ExceptedTypeOfException exc )
    {
        String expected = "A suitable error message";
        String actual = exc.getMessage();
        Assert.assertEquals( expected, actual );
    }
}
```

JUnit 3

- 현재 JUnit 3 에서 JUnit 4로 이전이 이루어지고 있음
  - Eclipse 3.2는
    - Eclipse 테스트 및 성능 도구 플랫폼은 JUnit 4와 동작하지 않음.
  - Netbeans 5.5 는 JUnit 3에만 동작.

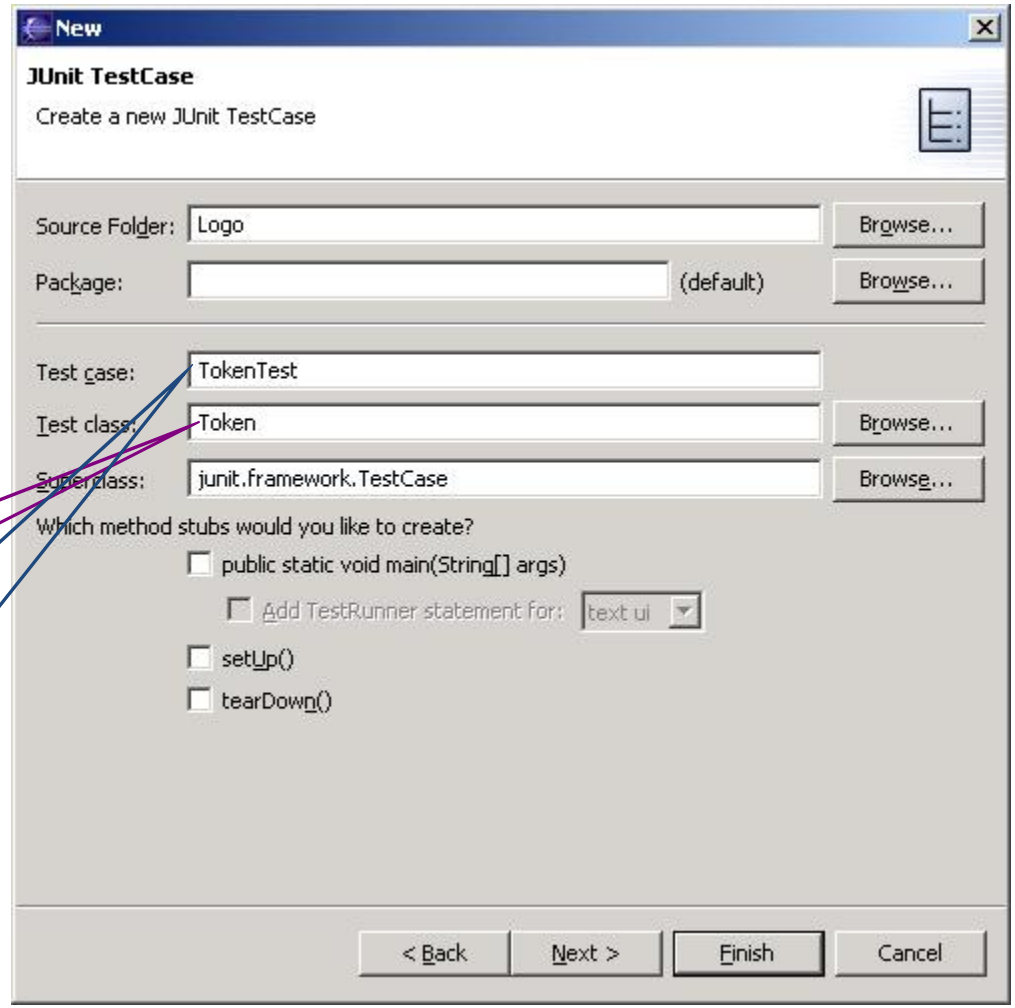
- JUnit archive에는 다음 두 페키지가 공존
  - JUnit 3: junit.framework.*
  - JUnit 4: org.junit.*

JUnit in Eclipse

- 테스트케이스 생성하려면

File→ New→ Other... → Java, JUnit, TestCase

- 테스트할 클래스의 이름 입력



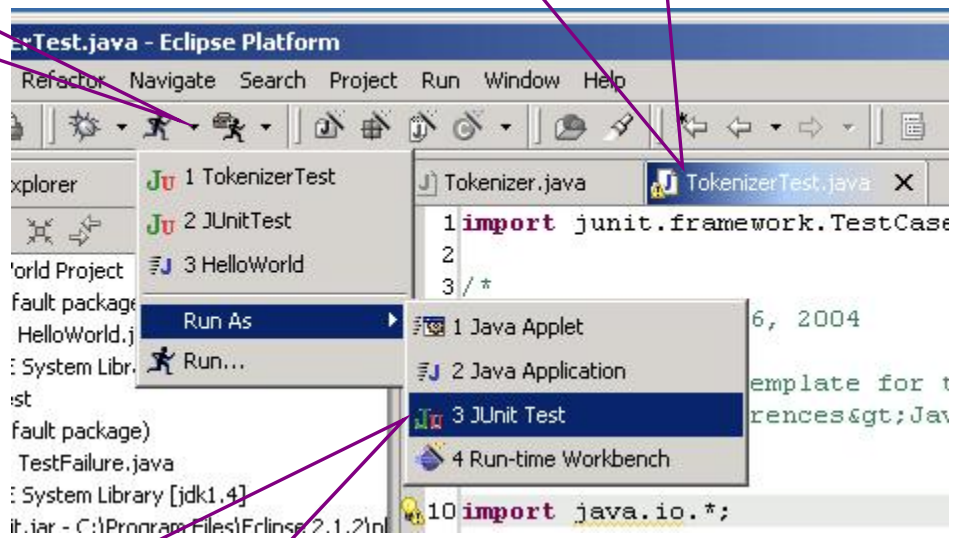
테스트할 클래스 입력

자동으로 채워짐

JUnit 실행

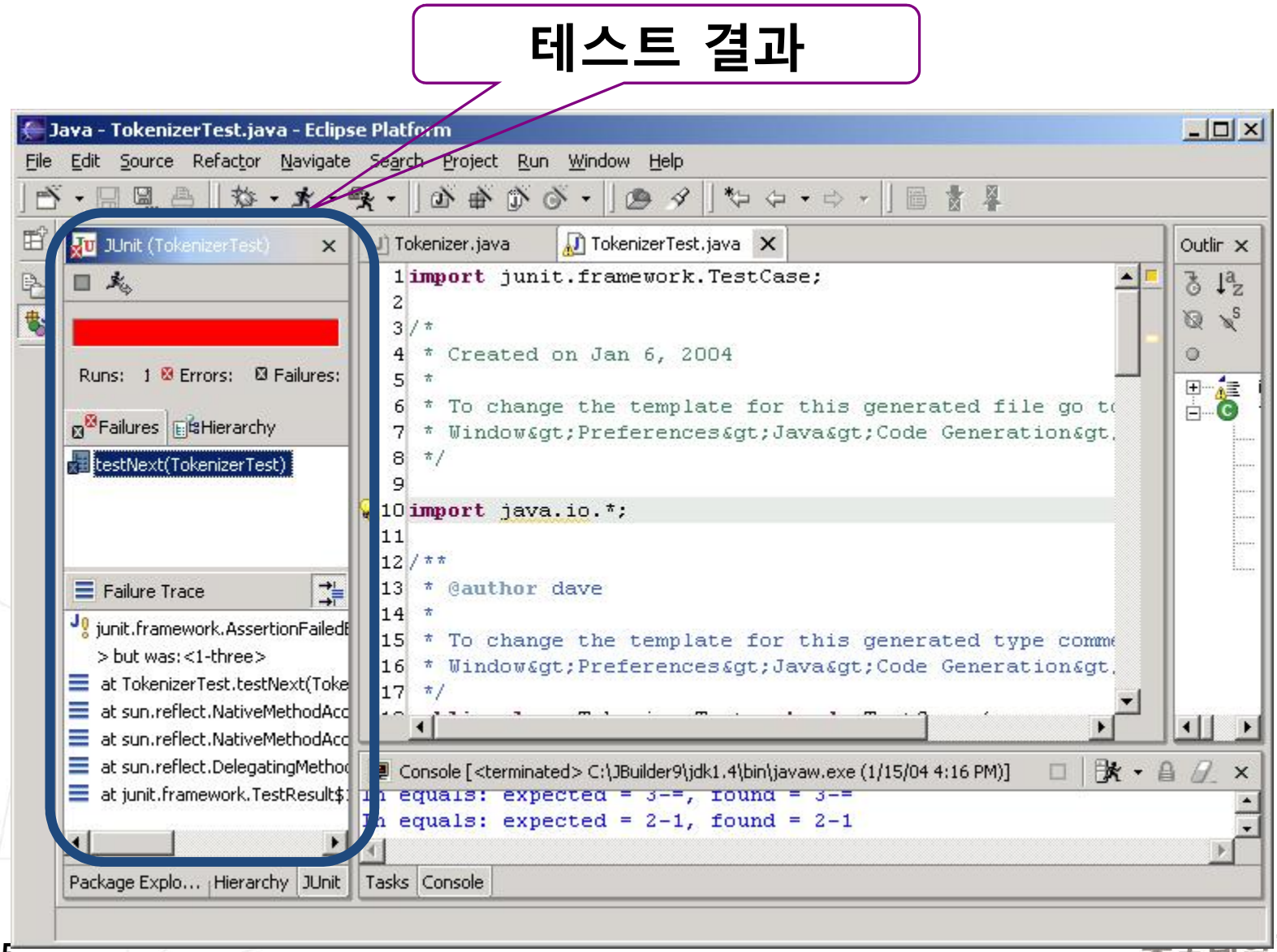
2. pulldown 메뉴 선택

1. 테스트 케이스 선택



3. Run As → JUnit Test

JUnit 실행 결과



참고 사이트

- http://www.junit.org
  - Junit 다운
  - Junit 이용을 위한 많은 정보
- http://sourceforge.net/projects/cppunit
  - Junit의 C++ 버전
- http://www.thecoadletter.com
  - 테스트 중심 개발(Test-Driven Development) 정보