소프트웨어 검증 발표
JUnit

200511305 김성규
200511306 김성훈
200518036 곡진화
200611124 유성배
200614164 김효석

JUnit이 필요한 이유

- 기존의 테스트 방식
  - 클래스에서 테스트 최소 단위는 메소드이며
    어떤 것이 유효한지를 찾으려면 하나씩 테스트해야 함
  - 테스트 구현하는 과정에서 한 번의 단일 테스트가 실패할 경우,
    후속 테스트가 전혀 수행되지 않아 전체적인 테스트가 불가능
  - 테스트를 자동으로 시작해주는 프레임워크가 없어
    각 테스트를 시작하기 위해서는 코드를 작성해야 함
  - 테스팅 코드는 생성된 클래스에 존재하여 증가한 코드의 크기로
    인한 문제는 없겠지만 보안상 문제가 발생할 수 있음

JUnit이란?

- 가장 널리 사용되는 Java 단위 테스트 프레임워크
  - 단위 테스트
    - 프로그램의 기본 단위가 내부 설계 명세에 맞게 동작하는지 테스트
  - 단위 테스트 케이스
    - 특별한 목표 또는 테스트 상황을 테스팅하기 위해 개발된 입력 값, 실행 사전조건, 예상 결과, 실행 사후조건의 집합
  - 단위 테스트 메소드
    - 사용자가 임의로 테스팅을 위해 작성하는 메소드
- 에릭 감마와 켄트 벡이 탄생시킨 JUnit
- 현재 오픈소스 프로젝트 방식으로 개발(4.9Beta)
  - http://www.junit.org/
- Eclipse 3.2부터 JUnit 기본 내장
  - JUnit3 & JUnit4 지원

JUnit의 장점

- 단위모듈(Method)이 정확히 구현되었는 지를 확인
  - Testcase 생성 및 실행, 오류추적
- 단위모듈별 테스팅을 가능케 함으로써 코드품질을 보장
- 단위테스팅으로 통합 테스트 시의 회귀결함을 감소
- 다른 모듈에 의존하지 않고 원하는 모듈만
  임의의 순서대로 테스트 가능
- JFeature(요구사항 개발도구)와 통합되어
  요구사항의 정확한 구현 비율을 알 수 있음

JUnit의 주요기능

- 테스트 하고자 하는 메소드에 대해 TestCase 사용
  - 테스트 결과가 예상과 같은지 판별해주는 assert함수 이용
- 여러 테스트에서 공용으로 사용할 수 있는 Test Fixture
  - 일관된 테스트 실행환경 설정
- 테스트 작업을 수행할 수 있게 해주는 Test Runner
- 일부의 특정 테스트 메소드만을 실행해주거나
  혹은 테스트 클래스를 한데 묶어서 실행해주는 TestSuite

JUnit3의 특징

- TestCase 클래스를 상속
- 테스트 메소드의 이름은 반드시 test로 시작
- 구성 요소
  - Test Fixture Method : setUp(), tearDown()
  - assert 함수
    - assertEquals, assertTrue, assertFalse
    - assertNull, assetNotNull, fail
  - Test Ruuner
    - junit.swingui.TestRunner.run(테스트클래스.class)
    - junit.textui.TestRunner.run(테스트클래스.class)
    - junit.awtui.TestRunner.run(테스트클래스.class)
  - Test Suite
    - 테스트 케이스와 다른 Test Suite를 포함 가능
    - 반드시 public static Test suite() 형태
    - 테스트 추가 : suite.addTestSuite(테스트클래스.class)

JUnit4의 특징

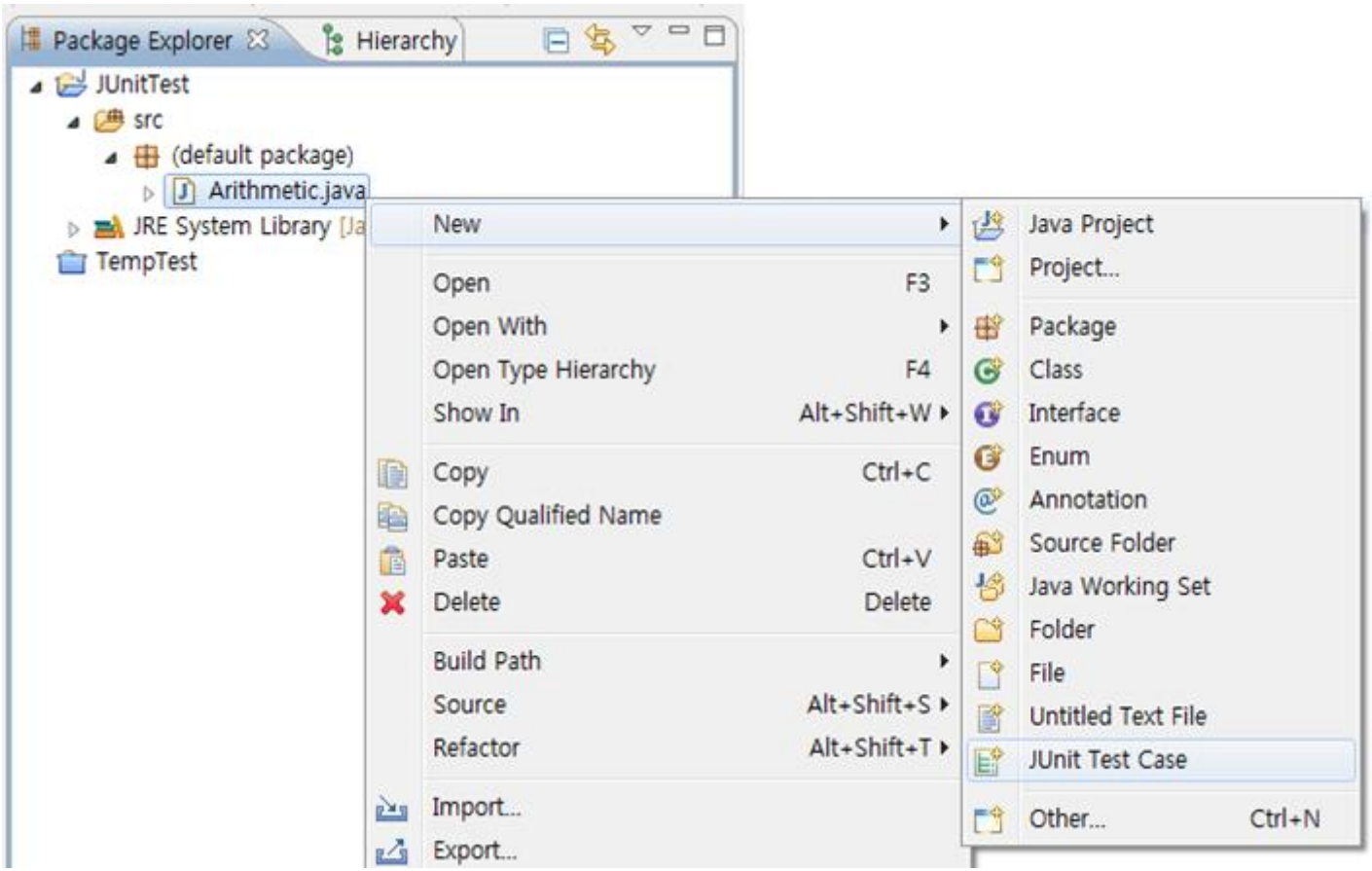
- Java 5 Annotation 사용
  - @Test
    - junit.framework.TestCase를 상속하지 않는 독립 클래스로 작성
    - test 로 시작해야만 하는 TestCase 메소드의 명명 규칙 제약 해소
    - @Test(timeout) : timeout 시간(ms)내에 완료되지 않으면 실패
    - @Test(expected=Exception 클래스.class) : 특정 예외가 발생 시 성공
  - @Before, @After, @BeforeClass, @AfterClass
    - 기존 Test Fixture인 setUp(), tearDown()의 확장
    - 언제 수행이 되고 이용 되는지 의도를 직관적으로 나타냄
  - @Ignore
    - 테스트에서 제외
    - 제외 된 것은 '/' 표시로 표현 됨

JUnit4의 특징

- Java 5 Annotation 사용
  - @RunWith(클래스명.class)
    - 지정된 클래스를 이용해 테스트를 진행 하도록 함
  - @SuiteClasses(Class[])
    - 여러 개의 테스트 클래스를 한꺼번에 수행
    - 테스트스위트 클래스 목록을 매개변수로 필요로 함
  - @Parameters
    - 파라미터를 이용한 테스트
- TestClass를 상속하지 않은 테스트 클래스
  - junit 함수 이용을 위해서는 import가 필요
  - import static org.junit.Assert.* 및 import org.junit.Test 등
- 배열 지원
  - assertArrayEquals 함수 추가

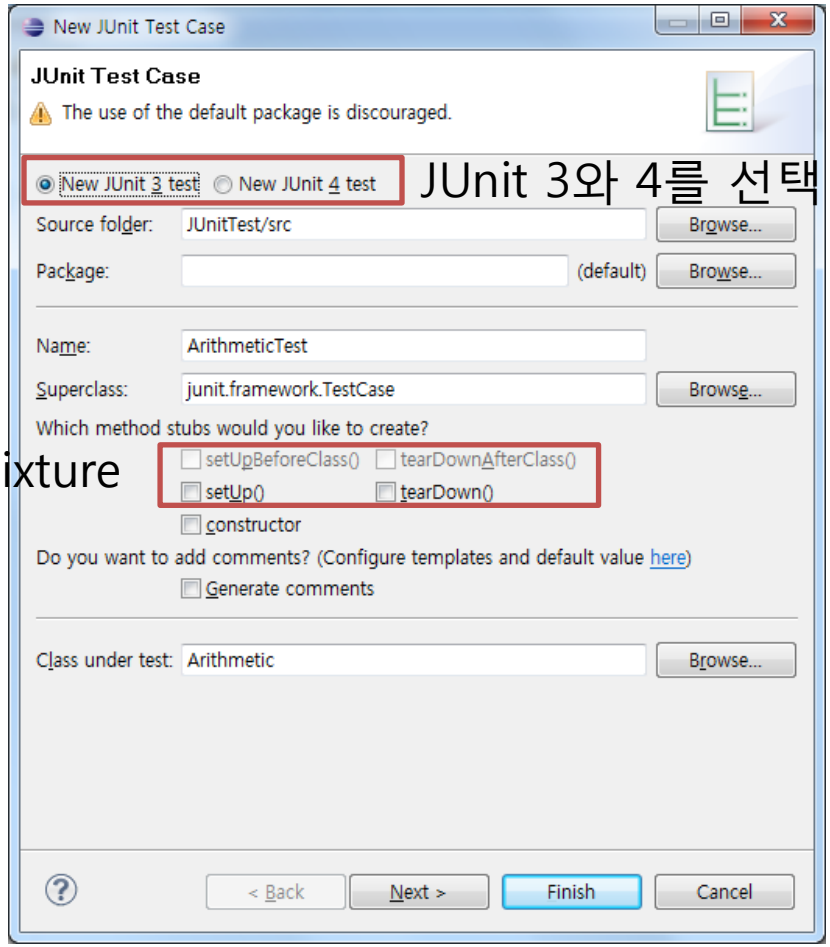
JUnit3의 Example

1. 이클립스의 패키지 탐색창에서 파일을 선택하고
   마우스 오른쪽 버튼을 눌러 JUnit Test Case 선택

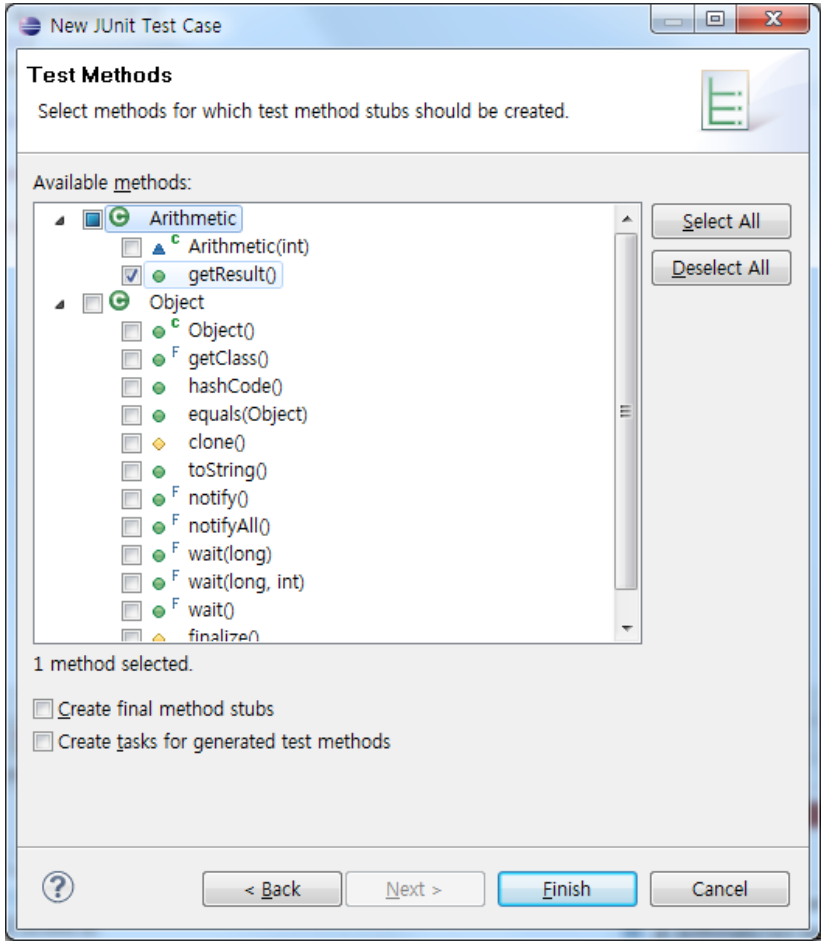


JUnit3의 Example

2. JUnit 3 선택 한 후 테스트할 메소드를 선택

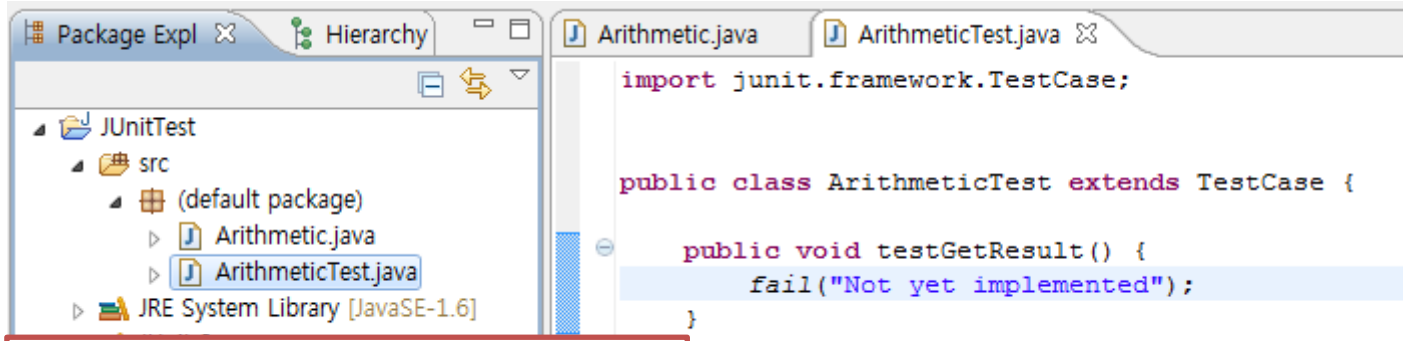


Test Fixture



JUnit3의 Example

3. 생성된 TestCase를 편집

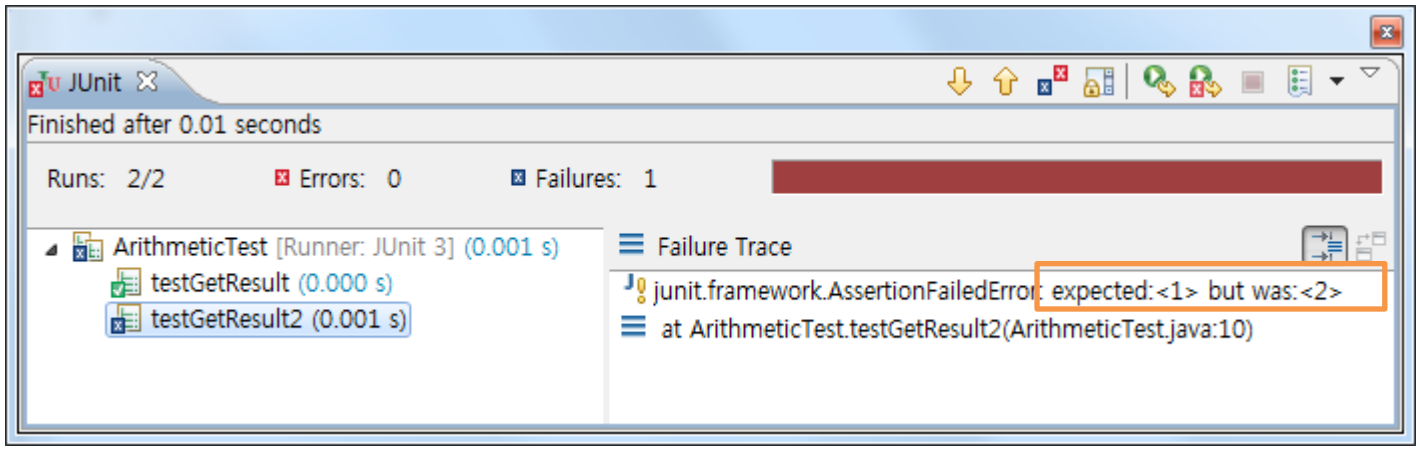



```
import junit.framework.TestCase;
public class ArithmeticTest extends TestCase{
    Arithmetic am;              //am.getResult = 2;
    public void testGetResult() {
        am = new Arithmetic();
        assertEquals(2,am.getResult());
    }
    public void testGetResult2() {
        am = new Arithmetic();
        assertEquals(1,am.getResult());
    }
}
```

- TestCase를 상속
- test로 시작하는 테스트 메소드

JUnit3의 Example

4. TestCase 수행



에러 메시지
expected : <1> but was : <2>
(expected) (actual)

JUnit4의 Example

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

public class ArithmeticTest {
        Arithmetic am;
        //am.getResult = 2;

        @Test
        public void testGetResult(){
                am = new Arithmetic();
                assertEquals(2,am.getResult());
        }
        @Test
        public void tGetResult2(){
                am = new Arithmetic();
                assertEquals(1,am.getResult());
        }
}
```

- TestClass를 상속하지 않음
- @Test라는 Annotation을 사용하여 테스트 클래스로 인식
- test로 시작하지 않아도 됨

JUnit - assert 메소드

❖ assert 메소드 – [message]는 선택적 사용

assertEquals([message], expected, actual)

- 기대값(expected) 과 실제값(actual) 이 같은지 비교

ex) assertEquals("EnoYa Hello~", hello.sayHello());

assertTrue([message], expected)
assertFalse([message], expected)

- 기대값의 참(true)/거짓(false) 판단

ex)assertTrue("EnoYa".startsWith("e"));

JUnit - assert 메소드

❖ assert 메소드 – [message]는 선택적 사용

assertNull([message], expected)
assertNotNull([message], expected)

- 기대값(객체)의 null 여부 판단

ex) assertNull("Is Not Null", hello);

assertSame([message], expected, actual)
assertNotSame([message], expected, actual)

- 기대값과 실제값이 동일한지 판단

ex)assertSame(someObject, cache.lookup(KEY));

JUnit - assert 메소드

❖ assert 메소드 – [message]는 선택적 사용

fail([message])

- 해당 메소드 호출 즉시 해당 테스트 케이스는 실패
- 아직 테스트 케이스가 미완료 상태 이거나,
  예외처리 테스트 등에 이용 가능

assertArrayEquals([message], expected, actual)

- 배열인 기대값과 실제값이 같은지 비교
- 배열 원소의 자리 순서로 비교하므로
  값 집합이 동일하더라도 순서가 다르면 테스트가 실패

JUnit - Test Fixture

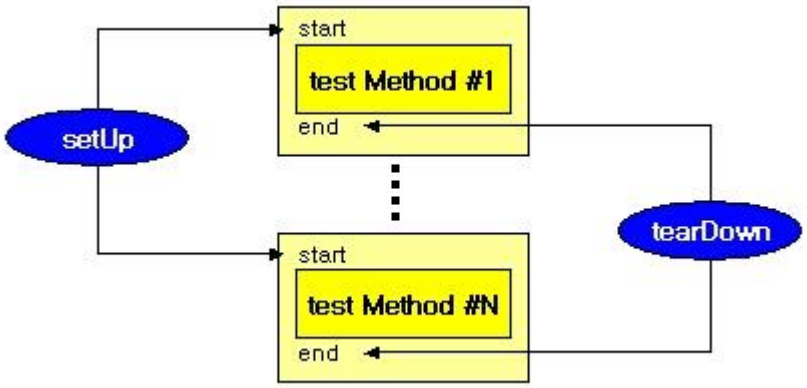
- Test Fixture
  - 테스트를 반복적으로 수행할 수 있게 도와주고

    동일한 결과를 얻을 수 있게 도와주는 기반이 되는 상태나 환경
  - 일관된 테스트 실행환경
- Text Fixture 메소드
  - JUnit 3 : setUp & tearDown 함수
  - JUnit4 : @Before, @Atfer, @BeforeClass, @AfterClass 이용
- 테스트 메소드들의 중복을 제거할 수 있을 뿐만 아니라

  각각의 테스트의 독립성을 보장 할 수 있게 됨

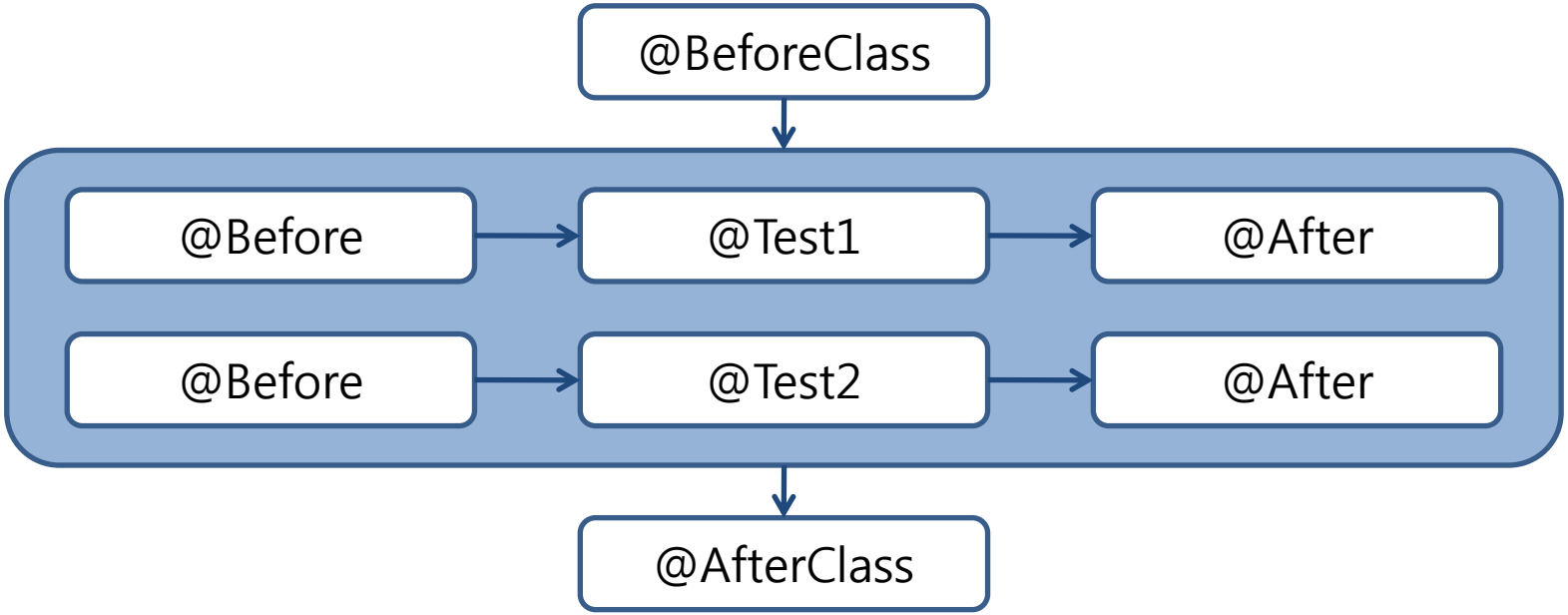
JUnit - Test Fixture

- JUnit 3의 Test Fixture
  - protected void setUp()
    - 각 테스트 메소드 실행전 환경 준비에 해당하는 공통 호출 메소드
    - DB Connection, 객체 생성 등의 작업
  - protected void tearDown()
    - 테스트 메소드 종료 된 직후 공통 호출 메소드
    - Connection close, 객체 초기화 등의 자원 해제 작업



JUnit - Test Fixture

❖ JUnit 4의 Test Fixture

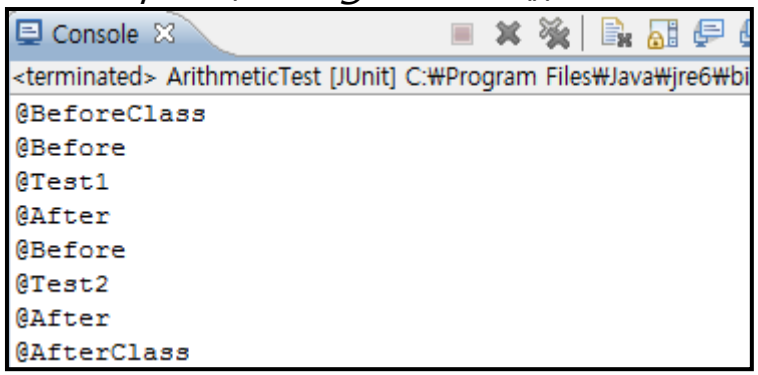


순서 : BeforeClass -> Before -> Test1 -> After
Before -> Test2 -> After -> AfterClass

JUnit - Test Fixture

❖ Example

```
@Test
public void testGetResult(){
        System.out.println("@Test1");
        am = new Arithmetic();
        assertEquals(2,am.getResult());
}
@Test
public void testGetResult2(){
        System.out.println("@Test2");
        am = new Arithmetic();
        assertEquals(1,am.getResult());
}
```

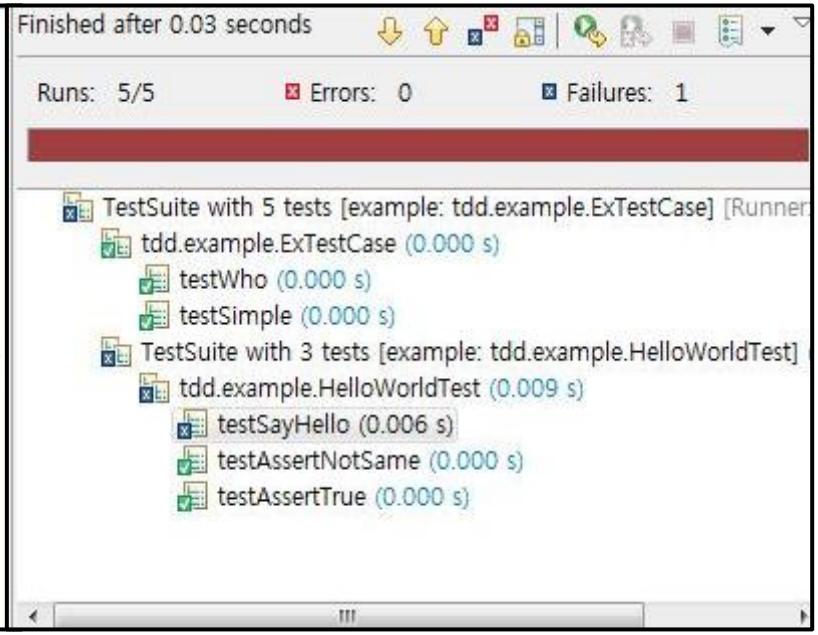



```
@BeforeClass
public static void setUpBeforeClass()
throws Exception {
        System.out.println("@BeforeClass"); }
@AfterClass
public static void tearDownAfterClass()
throws Exception {
        System.out.println("@AfterClass"); }
@Before
public void setUp() throws Exception {
        System.out.println("@Before"); }
@After
public void tearDown() throws
Exception {
        System.out.println("@After"); }
```

JUnit - Test Suite

- JUnit3 에서 여러 개의 테스트 클래스를 일괄적으로 수행
  - TestSuite 클래스를 이용
  - addTestSuite(클래스명.class) : 테스트 클래스 추가
  - addTest (테스트 스위트.suite()) : 테스트 스위트 추가

```
public class SimpleSuiteTest {

    public static void main(String args[]){
        junit.textui.TestRunner.run(ExTestCase.class);
    }


    public static Test suite(){  // 메소드는 꼭 이렇게 적어야 함.
        TestSuite suite = new TestSuite();

        // 테스트케이스 추가. - 테스트 클래스.class 로 확장자 까지 다 적어준다.
        suite.addTestSuite(ExTestCase.class);

        // 다른 테스트 스위트 추가.
        suite.addTest(ExSuiteTest.suite());

        return suite;
    }
}
```



JUnit - @SuiteClasses

- @SuiteClasses
  - JUnit3의 Test Suite 메소드와 동일한 일을 수행
  - @RunWith 와 함께 사용
    - import org.junit.runner.RunWith;
    - import org.junit.runners.Suite;
  - 별도의 메소드 없이 사용 가능

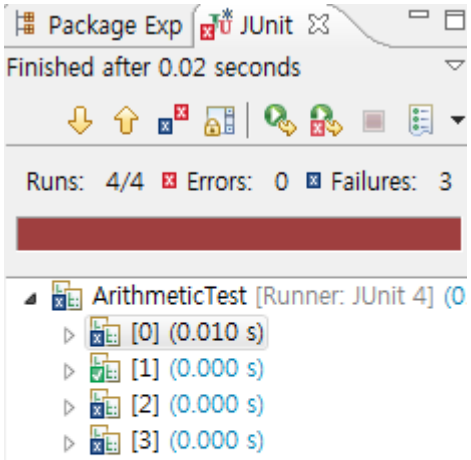
```
@RunWith(Suite.class)
@SuiteClasses( {클래스명1.class, 클래스명2.class, ... })
public class SuiteTest {
}
```

JUnit - @Parameters

- 하나의 메소드에 대해 다양한 값을 한꺼번에 실행
- @RunWith와 함께 사용
  - import org.junit.runner.RunWith;
  - import org.junit.runners.Parameterized;
  - import org.junit.runners.Parameterized.Parameters;
- @RunWith(Parameterized.class)로 테스트 클래스 선언
- @Parameters로 선언된 파라미터 제공 메소드 필요
  - 반드시 static 메소드며 Collection을 리턴
- 입력된 파라미터를 멤버 변수로 값을 할당하는 생성자
- @Test 메소드가 Collection 배열의 size 만큼 반복 수행

JUnit - @Parameters

```
@RunWith(Parameterized.class)
public class ArithmeticTest {
    Arithmetic am;
     int expected, actual;
    public ArithmeticTest(int expected, int actual) {
      this.expected = expected;
      this.actual = actual;
    }
    @Parameters
    public static Collection data() {
      return Arrays.asList(new Object[][]
      {{1,1+1}, //실패      {2,1+1},  //성공
      {3,1+1},  //실패      {4,1+1}});//실패
    }
    @Test
    public void testGetResult(){
      am = new Arithmetic(actual);
      assertEquals(expected, am.getResult());
    }
}
```



여러 입력값으로
한 번에 테스트

JUnit - assume 메소드

- JUnit 4.4 부터 지원
- import static org.junit.Assume.*; 추가
  - assumeThat, assumeTrue, assumeNotException, assumeNotNull
- 특정 조건을 만족시키면 계속해서 다음 코드 수행
- 조건을 만족시키지 않으면 @Test를 중단
  - assume이 fail이 되더라도 Error를 발생시키지는 않음

JUnit - @Rule

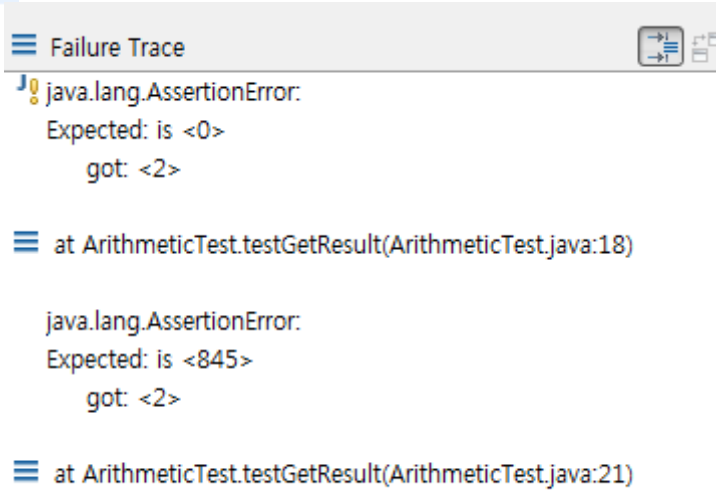
- JUnit 4.7 부터 지원
- 하나의 테스트 클래스 내에서 각 테스트 메소드의 동작 방식을 재정의하거나 추가하기 위해 사용

Rule	설 명
TemporaryFolder	테스트 메소드 내에서만 사용 가능한 임시 폴더나 파일을 생성
ExternalResource	외부 자원을 명시적으로 초기화
ErrorCollertor	테스트 실패에도 중단하지 않고 진행하게 도와줌
Verifier	테스트 케이스와 별개의 조건을 만들어서 확인할 때 사용
TestWatchman	테스트 실행 중간에 사용자가 끼어들 수 있게 함
TestName	테스트 메소드의 이름을 알려줌
Timeout	일괄적인 타임아웃을 설정
ExpectedException	테스트 케이스 내에서 예외와 예외 메시지를 직접 확인

JUnit - @Rule

- Rule의 ErrorCollector
  - @Rule을 이용하여 ErrorCollector 클래스 생성
  - ErrorCollector의 checkThat 함수를 이용하여 수행
  - Assertion Fail이 발생하더라도 테스트 계속 수행

```
import static org.hamcrest.CoreMatchers.*;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ErrorCollector;

public class ArithmeticTest {
    Arithmetic am;
    @Rule
    public ErrorCollector collector = new ErrorCollector();
    @Test
    public void testGetResult() {
        am = new Arithmetic();       // 성공
        collector.checkThat(2, is(am.getResult()));
        am = new Arithmetic(0);      // 실패
        collector.checkThat(2, is(am.getResult()));
        am = new Arithmetic(845);    // 실패
        collector.checkThat(2, is(am.getResult()));
    }
}
```



JUnit - Hamcrest

- JUnit 4.4 부터 Matcher 라이브러리 Hamcrest를 포함
- 문맥적인 흐름을 위해 assertThat 함수 추가
  - assertEquals([message], expected , actual)
  - assertThat([message], actual , matcher statement);
- 장점
  - 영어 문법적 표현으로 쉬운 작성과 좋은 가독성
    - assertThat(name.length(), is(6));
  - 가독성 높은 에러 메시지
  - Macher의 조합으로 기대하는 결과 표현이 쉬움
    - is(not(3))
  - 사용자 정의 Matcher 선언이 가능하여 표현 확장 가능
    - NotANumber(), isCapital()

JUnit - Hamcrest

- Hamcrest 1.1 Core 라이브러리
  - import static org.hamcrest.CoreMatchers.*; 추가

Matcher	기 능
is, equalTo	두 오브젝트가 동일한지 판별
not	두 오브젝트가 서로 같지 않은지 판별
nullValue notNullValue	Null 인지 아닌지 판별
allof	여러 개의 다른 오브젝트를 포함하고 있을 경우 서로 동일한지 판별
anyof	여러 개의 다른 오브젝트를 포함하고 있을 경우 하나라도 일치한다면 true
anything	어떤 오브젝트가 사용되든 일치한다고 판별
instanceOf	동일 인스턴스인지 타입 비교
sameInstance	Object가 완전히 동일한지 비교, equals 비교가 아닌 ==로(주소 비교) 비교

  - 상위 버전의 Hamcrest를 탑재하여 콜렉션, 숫자, 텍스트 등을 위한 다양한 Matcher 이용 가능 (hasKey, greaterThan 등)

JUnit - junit.matchers

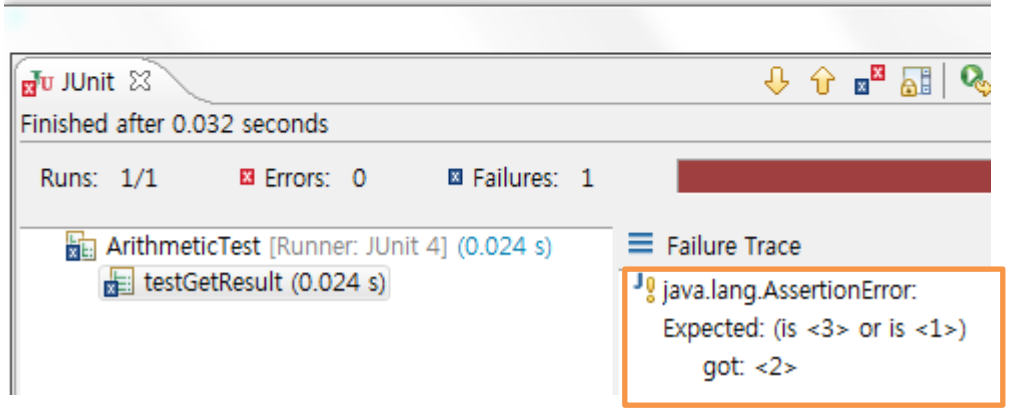
- junit.matchers
  - 유용하지만 Hamcrest에 포함되지 않은 matcher 라이브러리 제공
  - import static org.junit.matchers.JUnitMatchers.*; 추가

Matcher	기 능
both / either	matcher들을 결합시킬 때 사용
contrainString	문자열 포함 여부 판별
everyItem	콜렉션 내에 모든 elements 포함 여부 판별
hasItem / hasItems	콜렉션 내 element / elements 포함 여부 판별

JUnit - Matcher Example

❖ Hamcrest Example

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import static org.junit.matchers.JUnitMatchers.*;
import org.junit.Test;

public class ArithmeticTest {
    Arithmetic am;      //am.getResult = 2;
    @Test
    public void testGetResult(){
        am = new Arithmetic();
        assertThat(am.getResult(), either(is(1)).or(is(3)));
    }
```



Expected : (is <3> or is<1>)
got : <2>

- assertEquals 보다 가독성 좋은 에러 메시지