

JUnit

Author : Baek Myoung Sook

목차

- ◆ JUnit?
- ◆ JUnit의 필요성
 - JUnit을 도입하지 않은 테스트 환경
 - JUnit을 도입한 테스트 환경
- ◆ JUnit을 위한 테스트 코드 작성
- ◆ Annotation & Assertion

JUnit

◆ JUnit란?

- xUnit이라는 단위 테스트 프레임워크의 자바 구현물
- **Eric Gamma와 Kent Beck**
- 코드를 릴리즈 하기 전에 단위 테스트에 사용하는 툴
 - 외부 테스트 프로그램을 작성하여 **System.out**으로 일일이 디버깅하지 않아도 됨
 - 테스트 결과를 확인 하는 것 이외에도 최적화된 코드를 유추해내는 기능도 하므로 성능 향상도 기대
 - 테스트에 걸린 시간 또한 별도로 출력해 볼 필요가 없다
 - 테스트 결과를 단순한 텍스트로 남기는 것이 아니라 **Test**클래스로 남김으로 다음에 인계할 개발자에게 테스트 방법 및 클래스의 **history**를 넘겨줄 수 있다

JUnit

◆ 단위 테스트

- 프로그램의 기본 단위가 내부 설계 명세에 맞게 제대로 동작하는지를 테스트 하는 것
 - **Java**의 경우 기본단위가 클래스이므로 각 클래스에 포함된 메서드가 제대로 동작하는지를 테스트 하는 것
 - 범위가 한정적이다.

◆ 기능 테스트

- 소프트웨어 전체가 제대로 동작하는지를 확인하는 테스트
 - 기능 테스트는 보통 별도의 테스트 팀이 수행하며, 개발할 때와는 다른 별도의 도구와 기술을 사용

JUnit의 필요성

◆ JUnit를 도입하지 않은 테스트 환경

```
import java.util.Calendar;
```

```
public class DayCounter{
    public int MILLIS_PER_DAY = 1000*60*60*24;
    private Calendar day1 = null;
    private Calendar day2 = null;

    public void setDay1(Calendar day){
        this.day1 = day;
    }

    public Calendar getDay1(){
        return day1;
    }

    public void setDay2(Calendar day){
        this.day2 = day;
    }

    public Calendar getDay2(){
        return day2;
    }

    public long getDays(){
        long l_remain_date = day1.getTime().getTime() - day2.getTime().getTime();
        long remain_date = l_remain_date/MILLIS_PER_DAY;
        return remain_date;
    }
}
```

```
public static void main(String[] args){
    DayCounter counter = new DayCounter();
    Calendar day = Calendar.getInstance();

    day.set(2003, 10, 5);
    counter.setDay1(day);

    day = Calendar.getInstance();
    day.set(2002, 6, 2);
    counter.setDay2(day);

    System.out.println("day1 - day2 = " + counter.getDays());
}
```

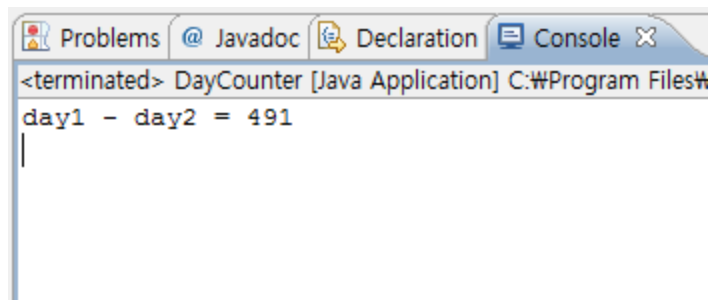
DayCounter클래스를 테스트 하기 위한 코드

두 날짜의 차이를 계산하는 DayCounter클래스

JUnit의 필요성(cont.)

◆ JUnit를 도입하지 않은 테스트 환경

□ 실행결과



The screenshot shows a console window with tabs for Problems, Javadoc, Declaration, and Console. The console output is as follows:

```
<terminated> DayCounter [Java Application] C:\Program Files\#  
day1 - day2 = 491  
|
```

□ 문제점

- 단위 테스트를 필요로 하는 클래스의 개수가 늘어나게 되면 이 예제의 경우처럼 각 클래스마다 **main()**을 다시 만들어야 함
- **main()**은 테스트용으로 작성된 것이기 때문에 실제 릴리즈를 하고 제품을 납품하는 단계에서 **main()**을 모두 삭제
- 별도로 기록을 보관할 수 없고 체계적으로 테스트를 관리할 수 없음

JUnit의 필요성(cont.)

◆ JUnit를 도입한 테스트 환경

```
import java.util.Calendar;

public class DayCounter{
    public int MILLIS_PER_DAY = 1000*60*60*24;
    private Calendar day1 = null;
    private Calendar day2 = null;

    public void setDay1(Calendar day){
        this.day1 = day;
    }

    public Calendar getDay1(){
        return day1;
    }

    public void setDay2(Calendar day){
        this.day2 = day;
    }

    public Calendar getDay2(){
        return day2;
    }

    public long getDays(){
        long l_remain_date = day1.getTime().getTime() - day2.getTime().getTime();
        long remain_date = l_remain_date/MILLIS_PER_DAY;
        return remain_date;
    }
}
```

두 날짜의 차이를 계산하는 DayCounter클래스

```
import java.util.Calendar;
import junit.framework.TestCase;

public class DayCounterTest1 extends TestCase{
    public void testGetDays(){

        DayCounter counter = new DayCounter();
        Calendar day = Calendar.getInstance();

        day.set(2003, 10, 5);
        counter.setDay1(day);

        day = Calendar.getInstance();

        day.set(2002, 6, 2);
        counter.setDay2(day);

        assertTrue(counter.getDays() == 491);
    }
}
```

DayCounter클래스를 테스트 하기 위한 코드 (JUnit3)

JUnit의 필요성(cont.)

◆ JUnit4에서 테스트 하기 위한 코드

```
import static org.junit.Assert.assertTrue;  
import java.util.Calendar;  
import org.junit.Test;
```

```
public class TestDayCounter {  
  
    @Test  
    public void testGetDays(){  
        DayCounter counter = new DayCounter();  
        Calendar day = Calendar.getInstance();  
  
        day.set(2003, 10, 5);  
        counter.setDay1(day);  
  
        day = Calendar.getInstance();  
  
        day.set(2002, 6, 2);  
        counter.setDay2(day);  
  
        assertTrue(counter.getDays() == 491);  
    }  
}
```


실습1

◆ largest 메소드를 테스트 하는 코드를 작성하고 다음을 만족하도록 largest 메소드를 수정

- 주어진 정수의 목록 중 가장 큰 원소를 반환
- assertTrue 사용

```
public int largest(int[] list){  
    int index, max = Integer.MAX_VALUE;  
    for(index = 0; index<list.length-1; index++){  
        if(list[index] > max){  
            max = list[index];  
        }  
    }  
    return max;  
}
```

실습 1

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

```
import org.junit.Test;
```

```
public class TestLargest {
```

```
    @Test
```

```
    public void Testlargest(){
```

```
        Largest large = new Largest();
```

```
        int[] list;
```

```
        list = new int[3];
```

```
        list[0] = 7;
```

```
        list[1] = 8;
```

```
        list[2] = 9;
```

```
        assertTrue(large.largest(list) == 9);
```

```
        assertEquals(large.largest(list), 9);
```

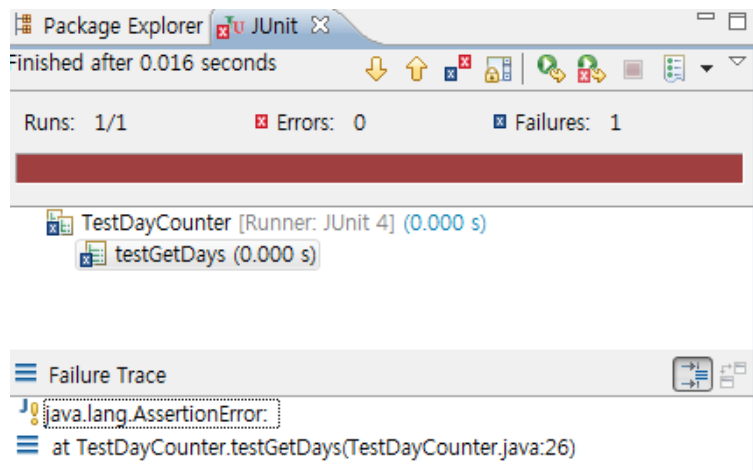
```
    }
```

```
}
```

JUnit의 필요성(cont.)

◆ JUnit를 도입한 테스트 환경

□ 실행결과(실패시)



□ 해결점

- 테스트의 검증을 별도의 클래스에서 작성하고, 테스트 클래스를 실제 소스와 함께 보관함으로 테스트를 체계적으로 관리
- 테스트에서의 걸린 시간, 테스트 실패여부, **Exception** 발생여부 등을 한번의 실행으로 알 수 있다.
- Ant의 **junit** 태스크까지 활용하면 테스트 결과 화면을 **html**로 볼 수 있다.

JUnit을 위한 테스트 코드 작성

◆ TestCase 클래스(JUnit3)

□ 가장 간단하게 JUnit를 사용하는 방법

○ TestCase 클래스를 상속

○ test로 시작하는 메소드 구현

```
import java.util.Calendar;  
import junit.framework.TestCase;
```

```
public class DayCounterTest1 extends TestCase{  
    public void testGetDays(){
```

```
        DayCounter counter = new DayCounter();  
        Calendar day = Calendar.getInstance();
```

```
        day.set(2003, 10, 5);  
        counter.setDay1(day);
```

```
        day = Calendar.getInstance();
```

```
        day.set(2002, 6, 2);  
        counter.setDay2(day);
```

```
        assertTrue(counter.getDays() == 491);
```

```
    }  
}
```

JUnit을 위한 테스트 코드 작성(cont.)

◆ TestSuite 클래스(JUnit3)

- 특정 메소드만 실행하거나 Test클래스를 한꺼번에 실행할 경우
 - suite()라는 메소드에서 TestSuite클래스를 만들어 리턴하는 방식

```
import junit.framework.*;

public class FirstTestCase extends TestCase{
    public void testOne(){
        System.out.println("[FirstTestCase] test one");
    }
}

import junit.framework.*;

public class SecondTestCase extends TestCase{
    public void testTwo(){
        System.out.println("[SecondTestCase] test two");
    }
}
```

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTest{
    public static Test suite(){
        TestSuite test = new TestSuite("All Test");
        test.addTestSuite(FirstTestCase.class); //방법1
        test.addTest(new TestSuite(SecondTestCase.class)); //방법2

        return test;
    }
}

.[FirstTestCase] test one
.[SecondTestCase] test two
```

JUnit을 위한 테스트 코드 작성(cont.)

◆ JUnit4 에서의 코드

```
import org.junit.Test;
public class FirstTestCase{
    @Test
    public void testOne(){
        System.out.println("[FirstTestCase] test one");
    }
}
```

```
import org.junit.Test;
public class SecondTestCase{
    @Test
    public void testTwo(){
        System.out.println("[SecondTestCase] test two");
    }
}
```

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
```

```
@RunWith(Suite.class)
@Suite.SuiteClasses({ FirstTestCase.class, SecondTestCase.class })
public class AllTest{
}
```

실습2

- ◆ Largest클래스의 largest 메소드를 참고하여 최소값을 구하는 smallest메소드와 테스트 코드를 작성하고 Junit TestSuite를 사용하여 largest와 smallest의 테스트 코드를 묶어서 실행

JUnit - 초기 값 설정 및 해제

◆ setUp() -> @Before

- 초기 값을 설정할 필요가 있을 경우
- test 메소드가 수행되기 직전에 매번 실행

◆ tearDown() -> @After

- 필요 없는 값을 해제
- test 메소드가 종료될 때마다 매번 실행

=> junit3

```
public void setUp(){  
    x = 20;  
}
```

```
public void tearDown(){  
    x = 0;  
}
```

=> junit4

```
@Before  
public void setUp(){  
}
```

```
@After  
public void tearDown(){  
}
```


JUnit – 초기 값 설정 및 해제

◆ @BeforeClass

- 테스트 클래스에서 테스트 메소드의 개수와 상관없이 모든 테스트 시작 전에 한번만 수행
- Ex) connect database

◆ @AfterClass

- 테스트 클래스에서 테스트 메소드의 개수와 상관없이 모든 테스트 완료된 후에 한번만 수행
- Ex) disconnect database

◆ @Ignore

- 테스트 되지 않아야 할 테스트 메소드를 지정

JUnit - Assertion

◆ Assertions: 비교확인, 조건확인, Null확인

- ❑ test메소드 중간중간에 어떤 조건이나 객체의 비교를 통해서 문제점을 꼬집어 낼 수 있도록 도와준다.
- ❑ test메소드 안에서 사용
 - 비교한 결과나 조건이 **false**면, **assertionFailure**라는 **Failure**를 내고 해당 **test**메서드를 종료
- ❑ 종류
 - **assertEquals(primitive expected, primitive actual)**
 - Expected는 기대하는 값, actual은 테스트 대상이 된 코드에서 실제로 나오는 값이다.
 - 두 개의 기본형 변수의 값이 같은지 검사
 - **assertEquals(Object expected, Object actual)**
 - 두 개의 객체 값이 같은지 검사(내부적으로 **equals()**메소드 사용)

JUnit - Assertion(cont.)

- `assertSame(Object expected, Object actual)`
 - `expected`와 `actual`이 같은 객체를 참조하는지 판정하고 그렇지 않으면 실패로 처리한다. (내부적으로 두 객체의 메모리 주소가 같은지 검사)
- `assertNotSame(Object expected, Object actual)`
 - 두 개의 객체가 다른지 검사(내부적으로 두 객체의 메모리 주소가 다른지 검사)
- `assertNull(Object object)`
 - 인자로 넘겨받은 객체가 `Null`인지 검사
- `assertNotNull(Object object)`
 - 인자로 받은 객체가 `Null`이 아닌지 검사
- `assertTrue(boolean condition)`
 - 조건문이 `true`인지 검사
- `assertFalse(boolean condition)`
 - 조건문이 `false`인지 검사

JUnit – 예외처리

◆ fail(message)

- ❑ 예외처리를 해주기 위해 사용
- ❑ 절대실행 되지 말아야 할 부분을 표시
- ❑ 해당 메소드 호출 즉시 해당 테스트 케이스는 실패

```
import junit.framework.*;
```

```
public class ExceptionTest extends TestCase{  
  
    public void testMyException(){  
        try{  
            throw new Exception("my exception");  
        }  
        catch(Exception e){  
            fail("my exception");  
        }  
    }  
}
```

JUnit - 예외처리

◆ 예외인식

- 작성한 테스트케이스가 **Exception**을 던지고, 그 **Exception**이 던져진 것이 맞는 상황임을 검증

```
@Test(expected=ArithmeticException.class)
public void testDivideByZeroV3(){
    int a = 3/0;
}
```

```
@Test(expected = NumberFormatException.class)
public void parseInteger(){
    int n = Integer.parseInt("two");
    System.out.println(n);
}
```

실습3

- ◆ 사칙연산을 수행하는 메소드를 작성하고 Junit을 이용하여 테스트
 - plus, minus, mult, div 메소드
 - Assertion을 이용하여 검증
 - 0으로 나눌 때 fail(message) 호출
 - 초기값 설정 이나 해제 사용

The End