# JUnit

Unit Test & JUnit Execution Examples

200511349 장기웅
200511300 강정희
200511310 김진규
200711472 진교선

# Content

1. Unit Testing
    1. Concept of TDD
    2. Concept of Unit Testing
    3. Unit Test Benefit & Limitation
    4. Unit Test Diagram
2. JUnit
    1. JUnit Annotation
    2. JUnit Method
3. JUnit Examples
    1. JUnit Example 1
    2. JUnit Example 2
    3. JUnit Example 3
4. References

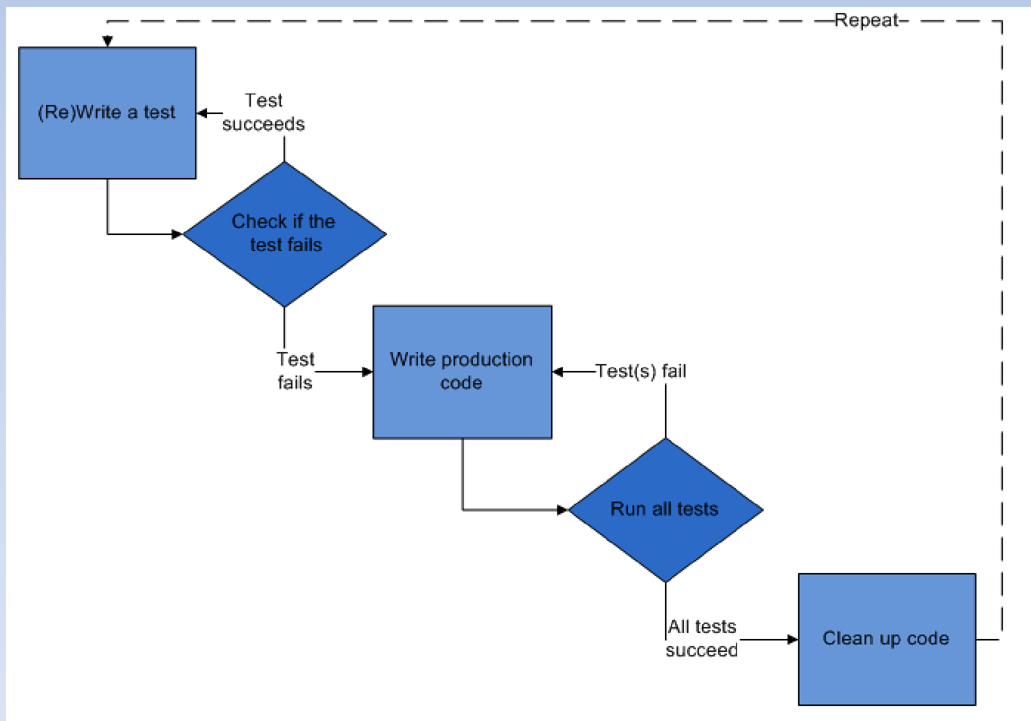
# Unit Testing

1. Concept of TDD
2. Concept of Unit Testing
3. Unit Test Benefit & Limitation
4. Unit Test Diagram

# Concept of TDD

## TDD – Test Driven Development

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle



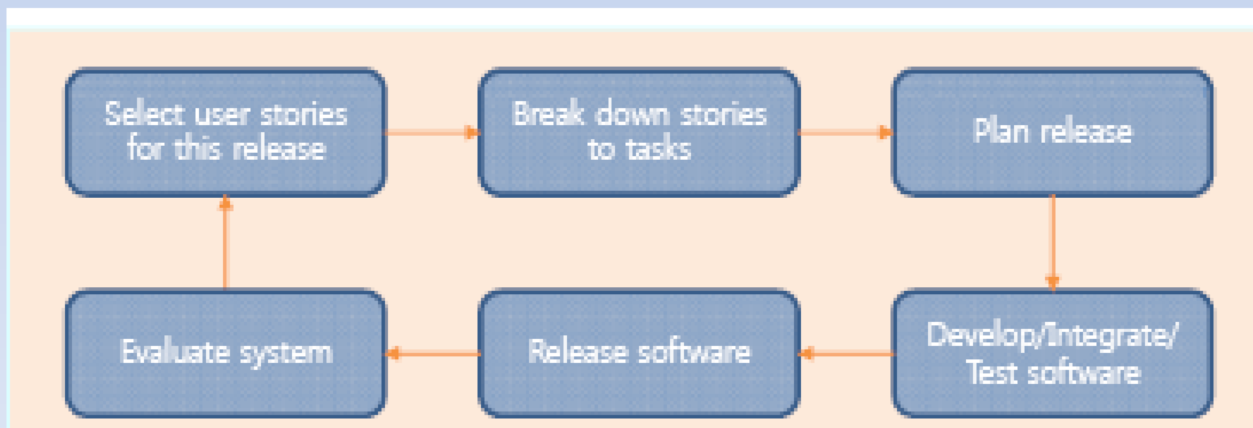

- 자동화된 Unit Test를 요구
- Code 작성 이전에 만들어진 Test Case를 요구
- Test의 통과 여부가 개발 진행과정을 통제한다.

# Concept of TDD

## Extreme Programming

Extreme Programming (XP) is the best-known agile method. XP is a test-first development

- Extreme Programming은 대표적은 TDD method이다.
- 새로운 Version이 계속 Release되는 방식이다.
- 모든 Version에 대해 Test가 이루어 진다.
- 이러한 테스트의 성공 여부가 Version의 수용 여부를 결정한다.

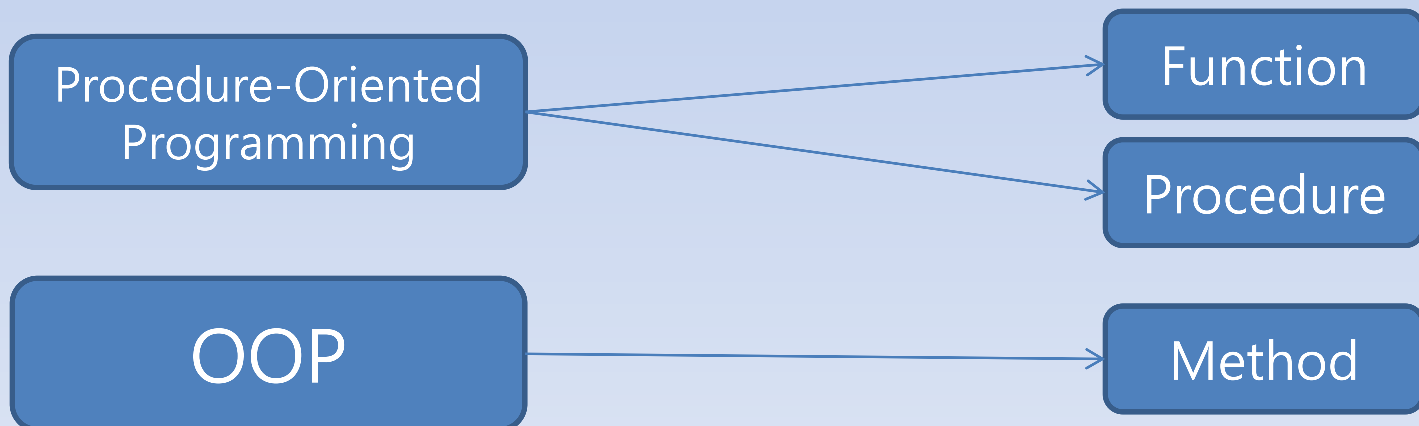


# Concept of Unit Testing

## What is Unit

the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. In object-oriented programming a unit is usually a method. created by programmers or occasionally by white box testers.

Application 에서 테스트 할 수 있는 최소 단위



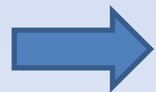

- Unit of Each Programming Style

# Concept of Unit Testing (Cont.)

## What is Unit Testing

In computer programming, unit testing is a method by which individual units of source code are tested to determine if they are fit for use.

- 단위 코드에서 문제 발생 소지가 있는 부분을 테스트
- 보통 클래스의 Public Method를 테스트
- 이상적인 테스트는 코딩 전에 Test Case 를 작성
- Test는 개발 후에 하는 것이 아니라 개발 단계에서 수행



Test를 통한 Code의 Robustness 를 보장!!!!

# Unit Test Benefits & Limitations

## Benefits of Unit Testing

- Facilitate Change

Unit testing provides a sort of living documentation of the system.

➡ 작은 단위의 테스트 이기 때문에 Code Refactoring의 부담이 줄어든다. 또한 다른 Unit이 올바르게 작동한다는 것을 보장하기도 한다.

- Simplifies Integration

Unit testing may reduce uncertainty in the units themselves and can be used in a bottom-up testing style approach

➡ Bottom Up 방식의 Unit 통합에서 Unit의 불확실함을 줄여준다. 올바르게 작동하는 것이 보장된 Unit의 통합을 통해 통합 시 발생하는 문제점들을 줄여준다.

# Unit Test Benefits & Limitations

## Benefits of Unit Testing

- Documentation

Unit testing provides a sort of living documentation of the system.

➡ Unit Test는 자동화된 테스트 과정이기 때문에 유닛의 적절/부적절한 작동에 대하여 정보를 문서화 시켜 제공해 준다.

- Design

When software is developed using a test-driven approach, the unit test may take the place of formal design.

➡ Unit Test는 TDD에서 디자인과 같은 위치를 가진다. Unit Test의 Pass/Non-Pass 여부가 개발의 진행 과정을 결정하게 된다. 또한 Test Case의 특성이 Unit의 특성을 결정하고, 이러한 Unit들이 전체 소프트웨어의 특성이 된다.

# Limitation of Unit Testing

- Disable to catch all errors

it is impossible to evaluate every execution path in all but the most trivial programs. The same is true for unit testing.

➡ Test Case가 모든 Error를 잡아 낼 수 없다. 또한 Unit에 대한 테스트 만을 하므로 통합 과정에서의 에러를 잡아 낼 수 없다.

- Size of Unit

For every line of code written, programmers often need 3 to 5 lines of test code.

➡ 각 테스트 과정에는 Unit에 대한 테스트 코드가 삽입되는 데, Unit의 분리가 확실하지 않거나 적절하지 못한 경우 중복된 테스트가 발생하거나 테스트 코드를 작성하는 데 드는 비용(시간 + 돈)이 증가할 가능성이 있다.

# JUnit

1. JUnit Annotation
2. JUnit Method

# J Unit Annotation

| Annotation | 설명 |
| --- | --- |
| @Test | • 실행할 method 앞에 붙임. |
| @Test(expected) | • 발생할 것으로 예상되는 예외를 지정한다. 예외가 생기지 않으면 실패. |
| @Test(timedout) | • 테스트가 끝나는 시간을 예측하여 지정된 시간보다 길게 테스트가 진행되면 실패. |

# J Unit Annotation

| Annotation | 설명 |
| --- | --- |
| @Ignore | • 다음에 오는 테스트를 무시함. 테스트를 하지 않을 method 앞에 붙임.(일종의 주석처리와 같은 역할을 함) |
| @Before, @After | • 각 단위 테스트 method의 실행 앞뒤에서 초기화와 자원 정리 작업을 수행 |
| @BeforeClass, @AfterClass | • 각 단위 테스트 클래스 수행 전후에 초기화와 자원정리 작업을 수행 |

# J Unit Annotation

**@RunWith**

- 지정된 러너가 아닌 사용자가 지정된 러너를 통해 특정 클래스를 실행하게 해준다.

**@SuiteClasses**

- 테스트 하려고 하는 여러 클래스를 지정한다.

**@Parameters**

- 여러 개의 파라미터 값을 테스트 하려고 할 때 자동적으로 테스트를 실행한다.

# J Unit Method

- **assertEquals**
  - 변수 나 객체의 값이 같은지 비교한다.
- **assertNull**
  - Null 값을 리턴하는지 비교
- **assertNotNull**
  - 인자로 넘겨받은 객체가 null인지 판정하고 반대인 경우 실패로 처리한다.

# J Unit Method

| Method | Description |
| --- | --- |
| assertSame | •assertSame은 expected 와 actual이 같은 객체를 참조하는지 판정하고 그렇지 않다면 실패로 처리한다. |
| assertNotSame | •Expected 와 actual이 서로 '다른' 객체를 참조하는지 판정하고, 만약 같은 객체를 참조한다면 실패로 처리한다. |
| assertTrue | •Boolean 조건이 참인지 판정한다. 만약 조건이 거짓이라면 실패로 처리한다. |
| assertFalse | •Boolean 조건이 거짓인지 판정한다. 만약 조건이 참이라면 실패로 처리한다. |

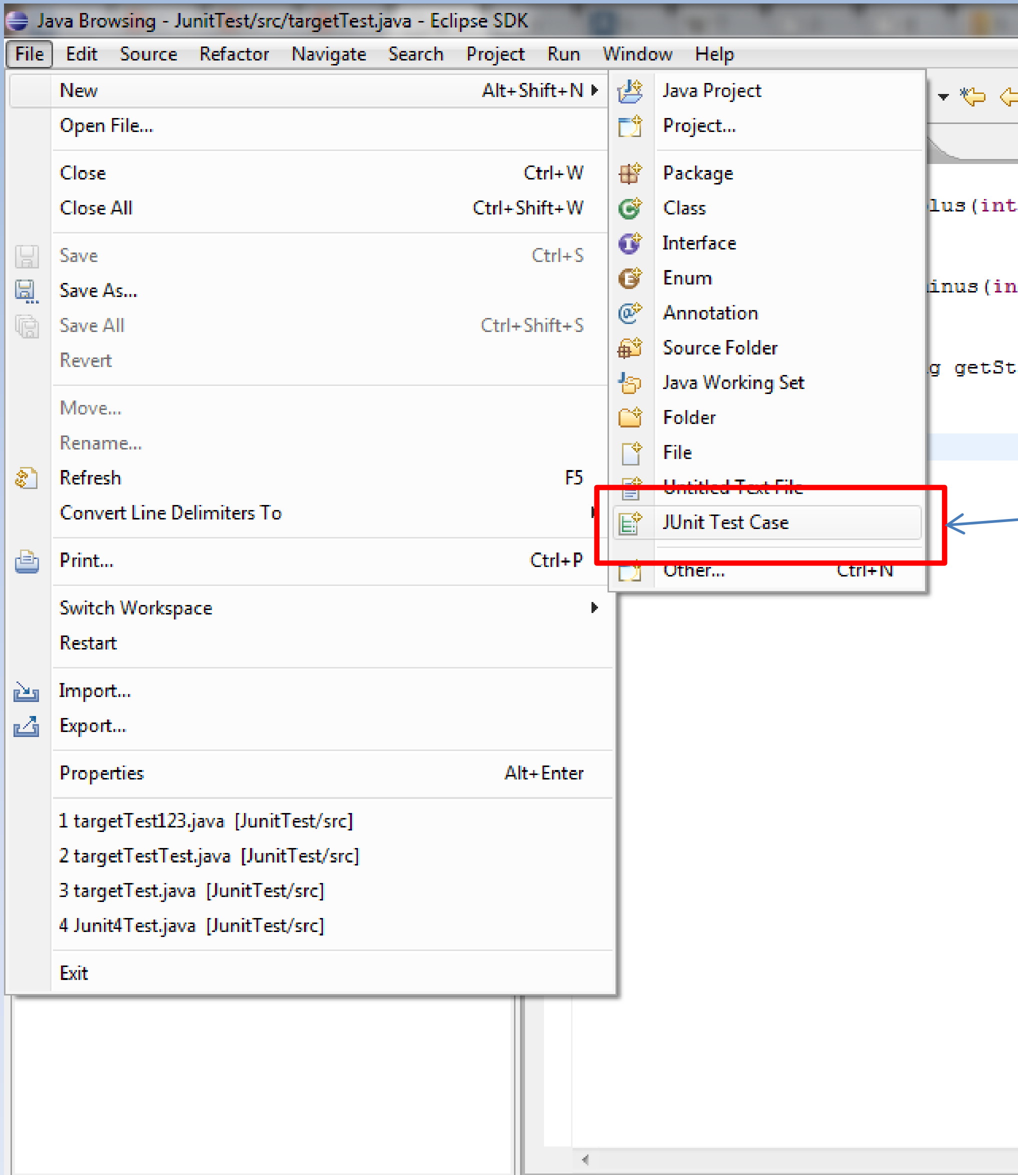
# JUnit Examples

1. JUnit Example1
2. JUnit Example2
3. JUnit Example3

# JUnit Example 1 – Target class

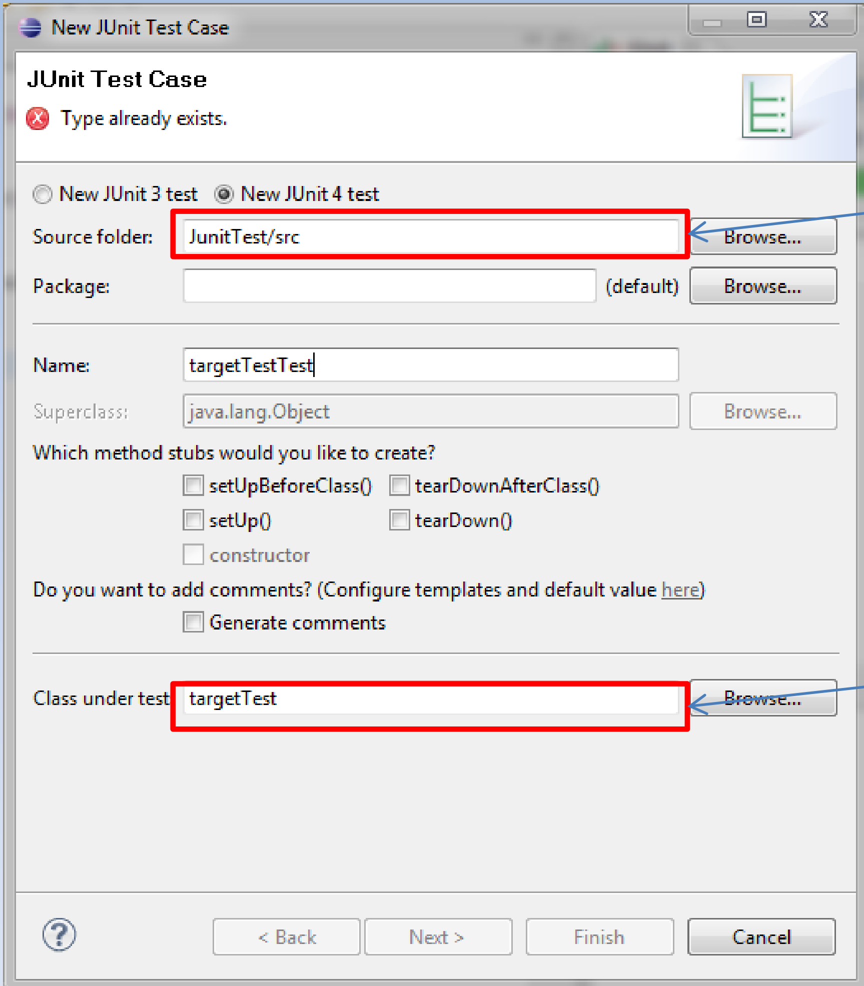
```
public class targetTest {
 public static int plus(int x, int y){
  return x + y;
 }
 public static int minus(int x, int y){
  return x - y;
 }
 public static String getString(){
  return "TEST";
 }
}
```

# JUnit Example 1 – Making Test Case




Junit Test Case 생성

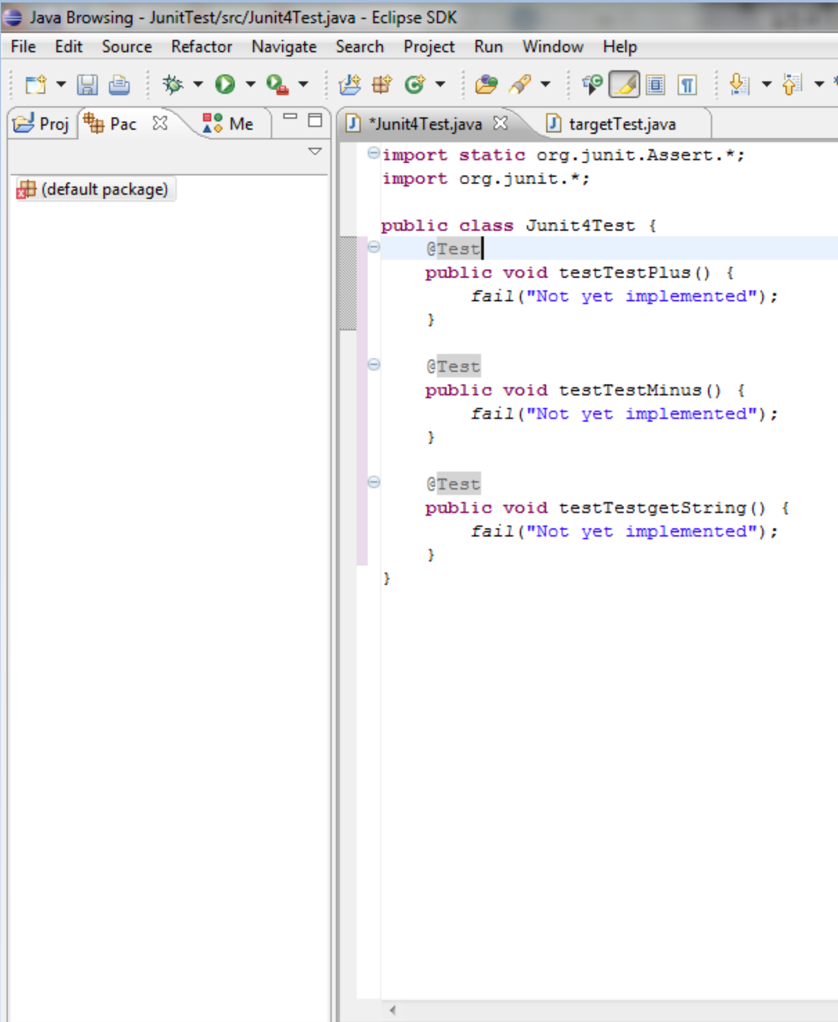
# JUnit Example 1 – Making Test Case (Cont.)

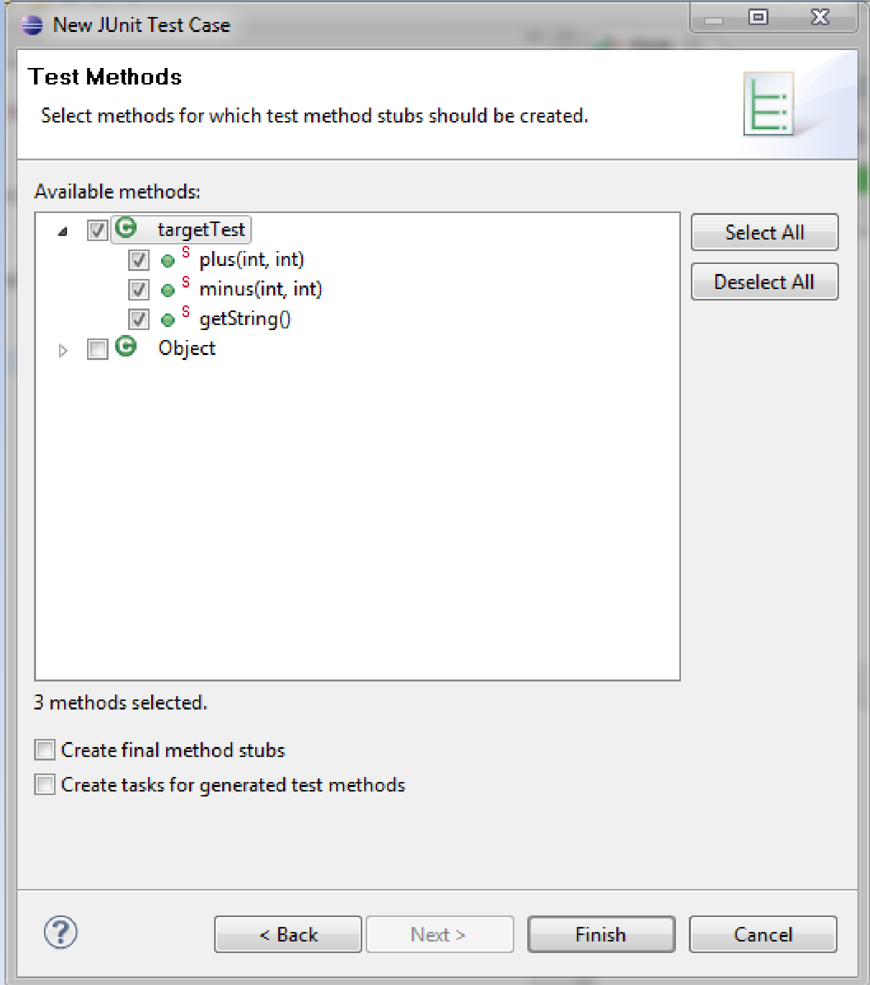



Test 할 Source

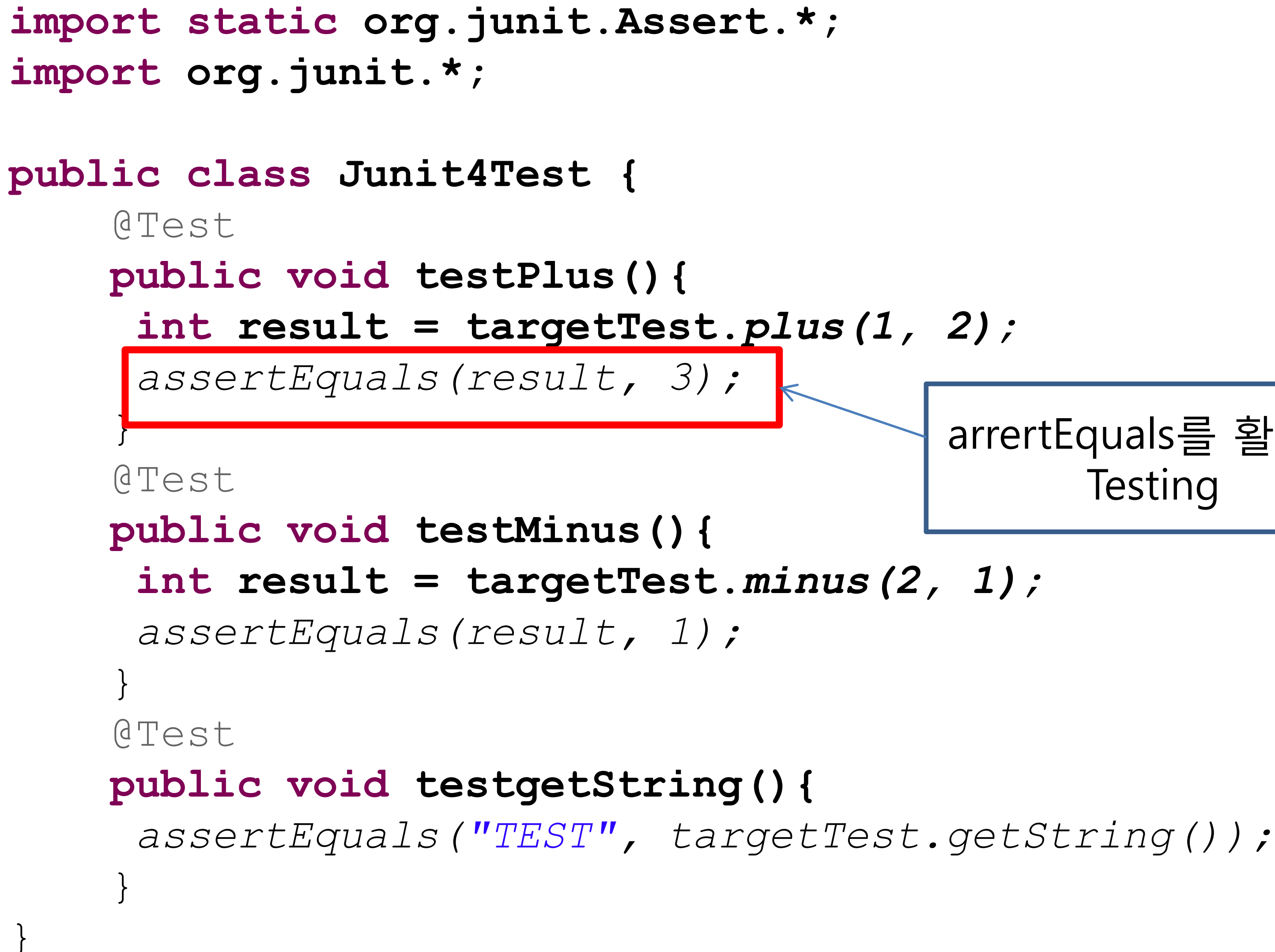
Test 할 Case

# JUnit Example 1 – Making Test Case (Cont.)

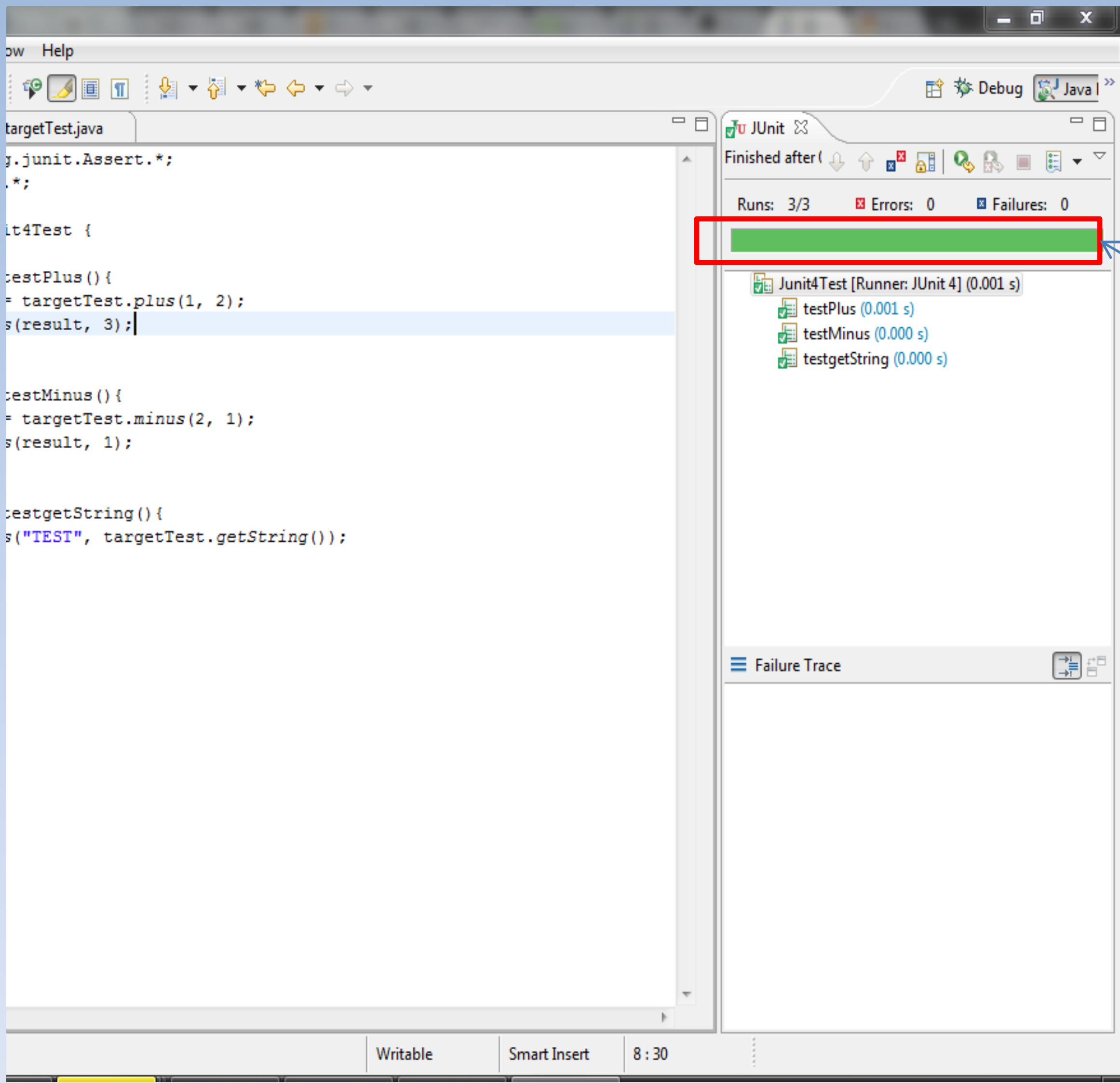


# JUnit Example 1 – Implemented Test Case Code



# JUnit Example 1 – Test Case Runing Result




Test 성공시
초록색으로 표시

# JUnit Example 1 – Test Case Runing Result (Fail)

- assertEquals()의 변수 값을 조정하여 고의적으로 Fail이 나게 조정하였다.

```
ic org.junit.Assert.*;
junit.*;

s Junit4Test {

void testPlus(){
sult = targetTest.plus(1, 2);
Equals(result, 4);


void testMinus(){
sult = targetTest.minus(2, 1);
Equals(result, 1);


void testgetString(){
Equals("TEST", targetTest.getString());
```

Runs: 3/3 Errors: 0 Failures: 1

Junit4Test [Runner: JUnit 4] (0.005 s)
- testPlus (0.002 s)
- testMinus (0.000 s)
- testgetString (0.000 s)

Failure Trace

java.lang.AssertionError: expected:<3> but was:<4>
at Junit4Test.testPlus(Junit4Test.java:8)

Test 실패시 붉은색으로 표시

Error 메시지 확인

# JUnit Example 2
# – Example of Executing Annotations

```
import static org.junit.Assert.*;
import org.junit.*;
public class Junit4Test {
static private int i, j;

@BeforeClass
public static void ExeBeforeTest() {
i = 3;
j = 2;
System.out.println("Execute
@BeforeClass");
}
@Before
public void TestAdd() {
int k = i + j;
assertEquals(5, k);
System.out.println("Execute @Before");
}
```

BeforeClass Annotation 은 전체 Test에서 1회 실행된다

Before Annotation 은 Unit Test마다 1회 실행된다

# JUnit Example 2
# – Example of Executing Annotations

```
@Test
public void TestSub() {
int k = i - j;
assertEquals(1, k);
System.out.println("Execute @Test_1st");
}
@Test
public void Test2nd()
{System.out.println("Execute @Test_2nd");}
@After
public void TestMul() {
int k = i * j;
assertEquals(6, k);
System.out.println("Execute @After");
}
@AfterClass
public static void ExeAfterTest() {
System.out.println("Execute @AfterClass");
}
}
```

After Annotation 은 Unit Test마다 실행된다

AfterClass Annotation 은 전체 Test에서 1회 실행된다

# JUnit Example 2
# – Example of Executing Annotations

```
;
Test() {
ute @BeforeClass");
ute @Before");
ute @Test_1st");
m.out.println("Execute @Test_2nd");}
ute @After");
est() {
ute @AfterClass");
```

JUnit
Finished after
Runs: 2/2 Errors: 0 Failures: 0
Junit4Test [Runner: JUnit 4] (0.001 s)
Failure Trace

Console
<terminated> Junit4Test [JUnit] C:\Program Files\Java\jr

```
Execute @BeforeClass
Execute @Before
Execute @Test_1st
Execute @After
Execute @Before
Execute @Test_2nd
Execute @After
Execute @AfterClass
```

각 Annotation 에 대한 실행 수와 실행 우선순위를 알 수 있다.

# JUnit Example 3 – Timeout Examples

```
import org.junit.*;
public class Junit4Test {
@Test(timeout = 3000)
public void testTimeout1() throws
Exception {
Thread.sleep(2000); // success
}
@Test(timeout = 3000)
public void testTimeout2() throws
Exception {
Thread.sleep(4000); // failure
}
}
```

Timeout 값보다 Sleep 값이 적다.

Timeout 값보다 Sleep 값이 크다.

# References


http://pentium3538.springnote.com/pages/174356

http://asisis.tistory.com/316

http://www.javajigi.net/pages/viewpage.action?pageId=278

http://devtainer.blogspot.com/2010/11/junit-4x.html

http://www.java2go.net/blog/archive/20080106

http://scroogy.tistory.com/30

http://en.wikipedia.org/wiki/Unit_test

http://en.wikipedia.org/wiki/JUnit

http://www.javajigi.net/pages/viewpage.action?pageId=278