Lab 04
Unit Testing & JUnit

NetDB
CS, NTHU,
Fall, 2013

Why Do We Need Testing ?

Actually,
you are doing that everyday…

```
System.out.println(“haha” + value);
```

But it is very inefficient

But it is very inefficient

Console JUnit Tasks

```
<terminated> App (4) [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/
abcabc
ha
1
1
ha
!!!
1
haha
abcabc
!!!
fuck
abcabc
main() executed
```

And there are always more
“important” things to do…

Use Program to test Program

Outline

- Unit Testing
- Using JUnit 4 in Eclipse
- Today's Mission

Outline

- Unit Testing
- Using JUnit 4 in Eclipse
- Today's Mission

Unit Testing

“Unit testing is a method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application.”

wikipedia.org

Why Unit Test

- Debugging is a time consuming process
- When new functionality is added, how do we make sure the old one doesn't break ?
- By looking at a Unit Test, you can see the class in action, which lets you easily understand its intent and proper use
- Unit tests are the only real measure of project health and code quality

Properties of a Unit Test

- Isolated
- Repeatable
- Fast
- Self-Documenting

Inside a Test

- Class Name - what do you want to test ?
  - Describe the component you want to test
- Before
  - The code that will be executed before every test
- After
  - The code that will be executed after every test
- Tests
  - Write your test cases here

Example

- There is a calculator class we want to test:

```
public class Calculator {
    public int plus(int op1, int op2) {
        return op1 + op2;
    }
    public int minus(int op1, int op2) {
        return op1 - op2;
    }
}
```

Example

- Create a new test file

```
CalculatorTest
```

Example

- Before every test, we want to recreate the calculator

```
calculator = new Calculator();
```

Example

- Write a test case to test if the plus method work correctly

```
int result = calculator.plus(10, 5);
Assert.assertEquals(15, result);
```

Example

- After each test, we want to destroy the calculator

```
calculator = null;
```

Outline

- Unit Testing
- Using JUnit 4 in Eclipse
- Today's Mission

JUnit

- JUnit is a simple framework to write repeatable tests

```
public class CalculatorTest {

    private Calculator calculator;
```

Before

```
@Before
public void setUp(){
    calculator = new Calculator();
}
```

After

```
@After
public void tearDown(){
    calculator = null;
}
```

Test

```
@Test
public void calculatorShouldPlusCorrectly(){
    int result = calculator.plus(10, 5);
    Assert.assertEquals(15, result);
}
```

Test

```
@Test
public void calculatorShouldMinusCorrectly(){
    int result = calculator.minus(10, 5);
    Assert.assertEquals(5, result);
}

}
```

Annotation

- @BeforeClass
- @Before
- @Test
- @After
- @AfterClass

What’s the result ?

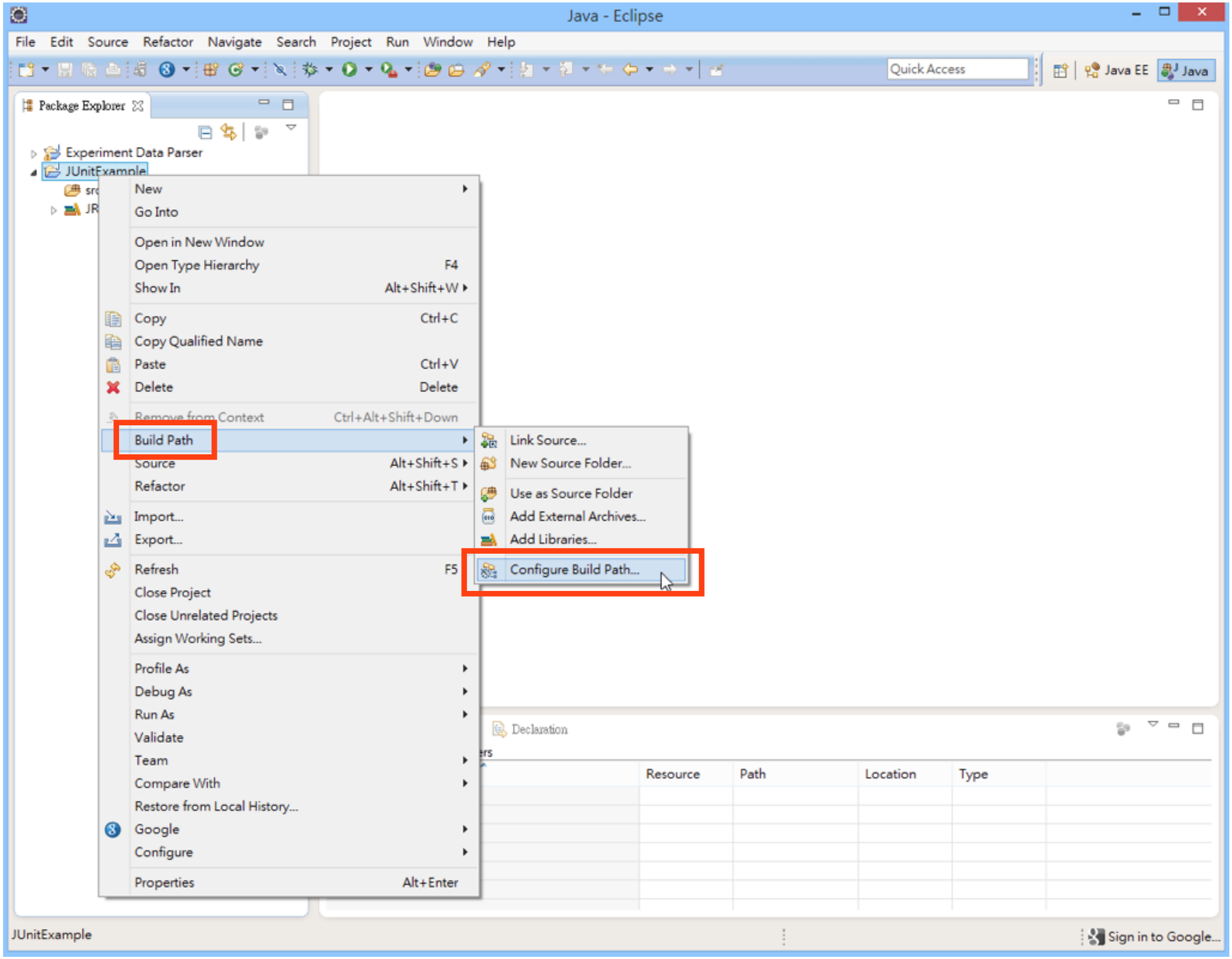
```
public class CalculatorTest {
    @Before
    public void setUp(){
        System.our.println(“before”);
    }

    @After
    public void tearDown(){
        System.our.println(“after”);
    }

    @Test
    public void test1(){
        System.our.println(“test1”);
    }

    @Test
    public void test2(){
        System.our.println(“test2”);
    }
}
```

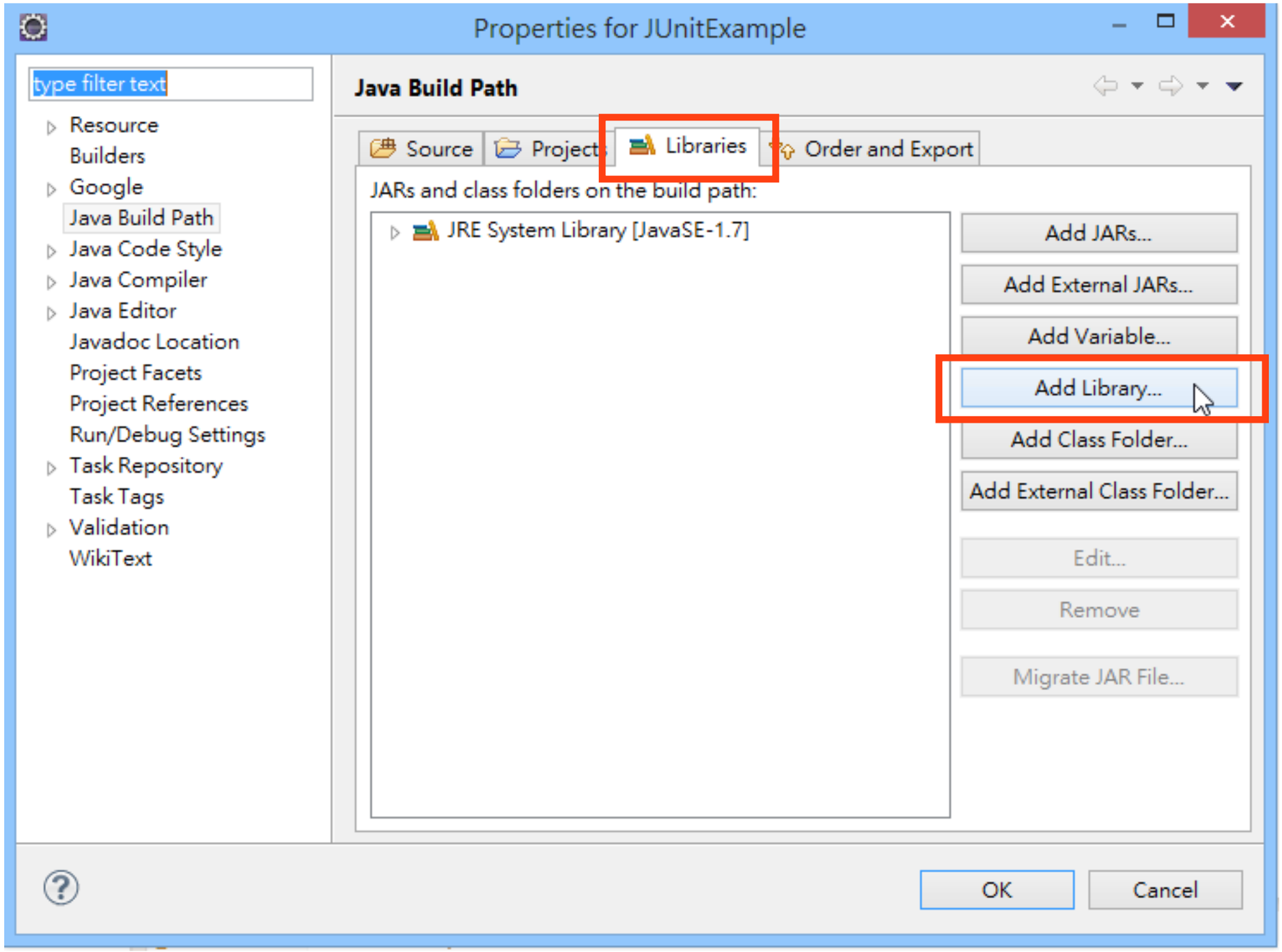
What's the result ?

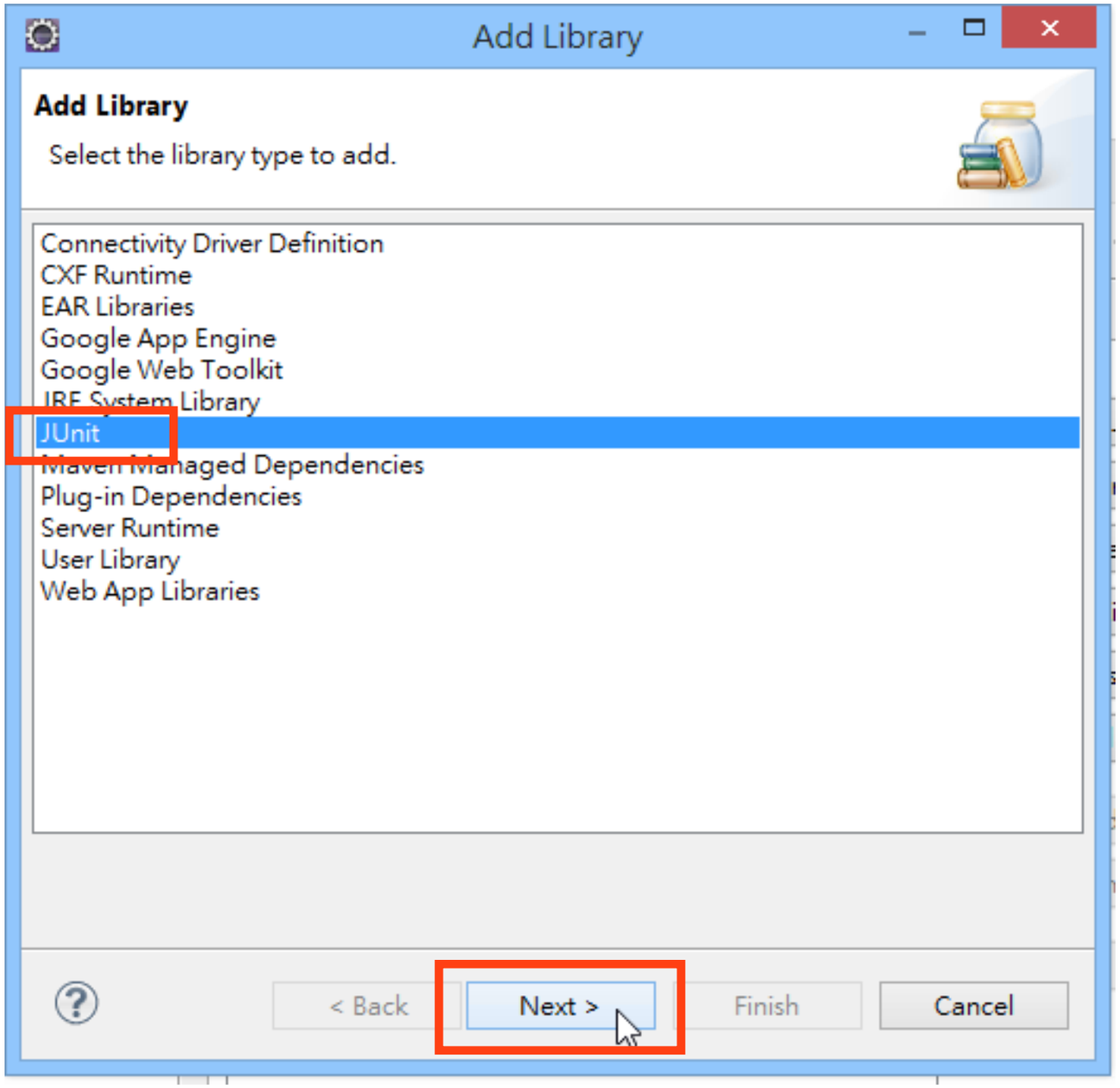
```
public class CalculatorTest {
    @Before
    public void setUp(){
        System.our.println("before");
    }

    @After
    public void tearDown(){
        System.our.println("after");
    }

    @Test
    public void test1(){
        System.our.println("test1");
    }

    @Test
    public void test2(){
        System.our.println("test2");
    }
}
```

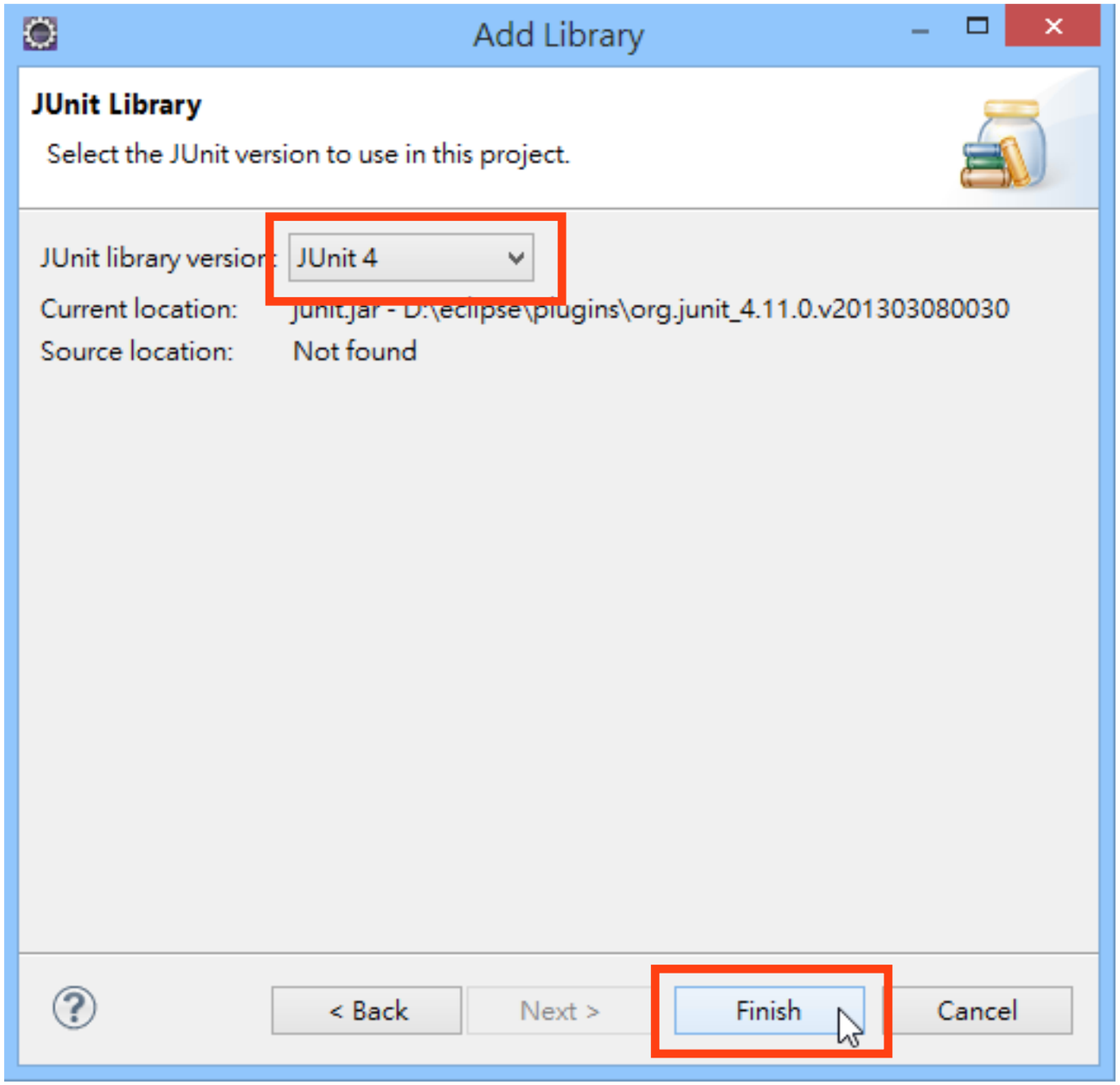
```
before
test1
after
before
test2
after
```

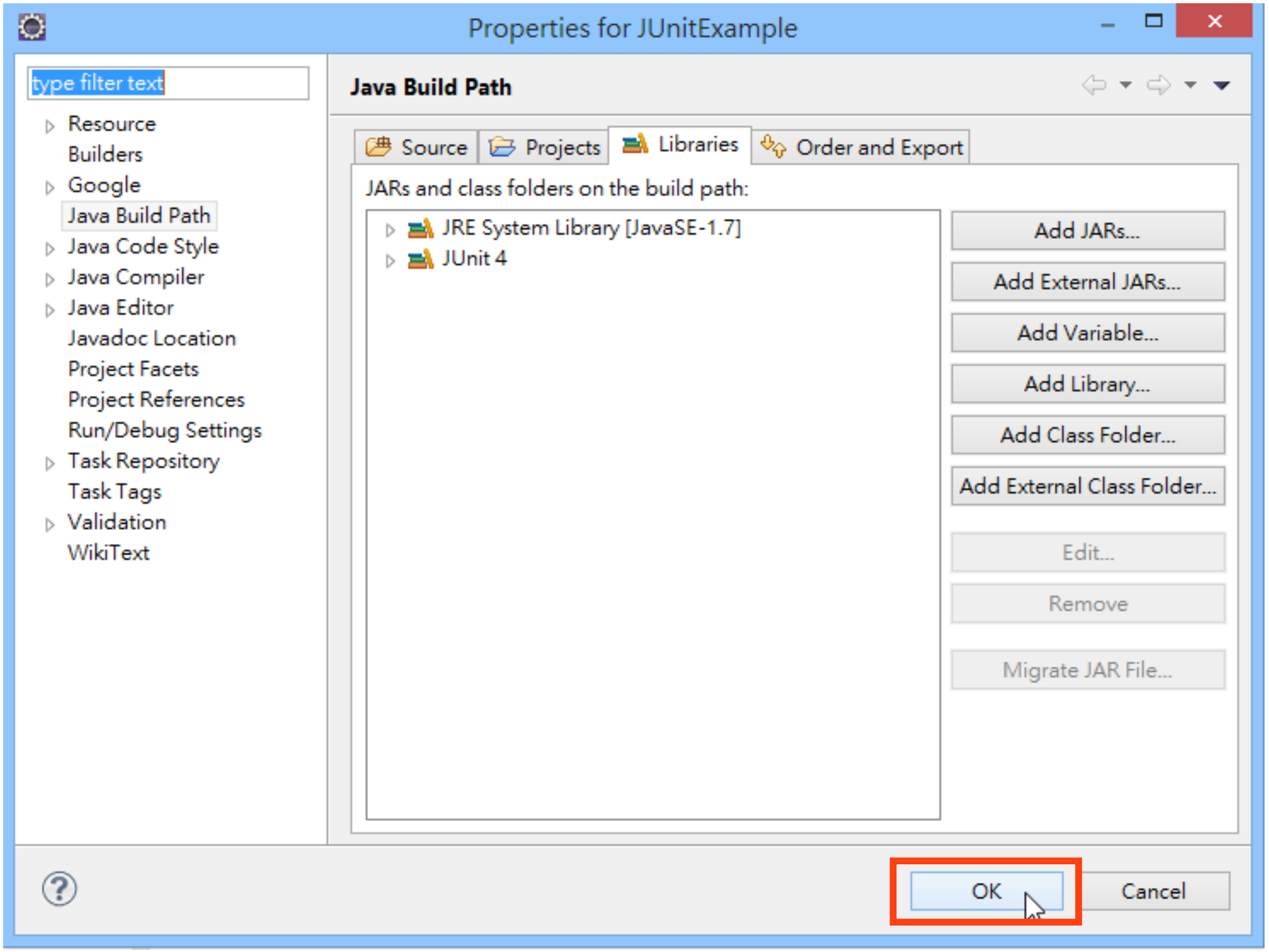
Adding the JUnit4 Library


Java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java EE
Java
Package Explorer
Experiment Data Parser
JUnitExample
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Profile As
Debug As
Run As
Validate
Team
Compare With
Restore from Local History...
Google
Configure
Properties Alt+Enter
Link Source...
New Source Folder...
Use as Source Folder
Add External Archives...
Add Libraries...
Configure Build Path...
Declaration
Resource
Path
Location
Type
JUnitExample
Sign in to Google...


Properties for JUnitExample
type filter text
Resource
Builders
Google
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
Project Facets
Project References
Run/Debug Settings
Task Repository
Task Tags
Validation
WikiText
Java Build Path
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
JRE System Library [JavaSE-1.7]
Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Add External Class Folder...
Edit...
Remove
Migrate JAR File...
OK
Cancel

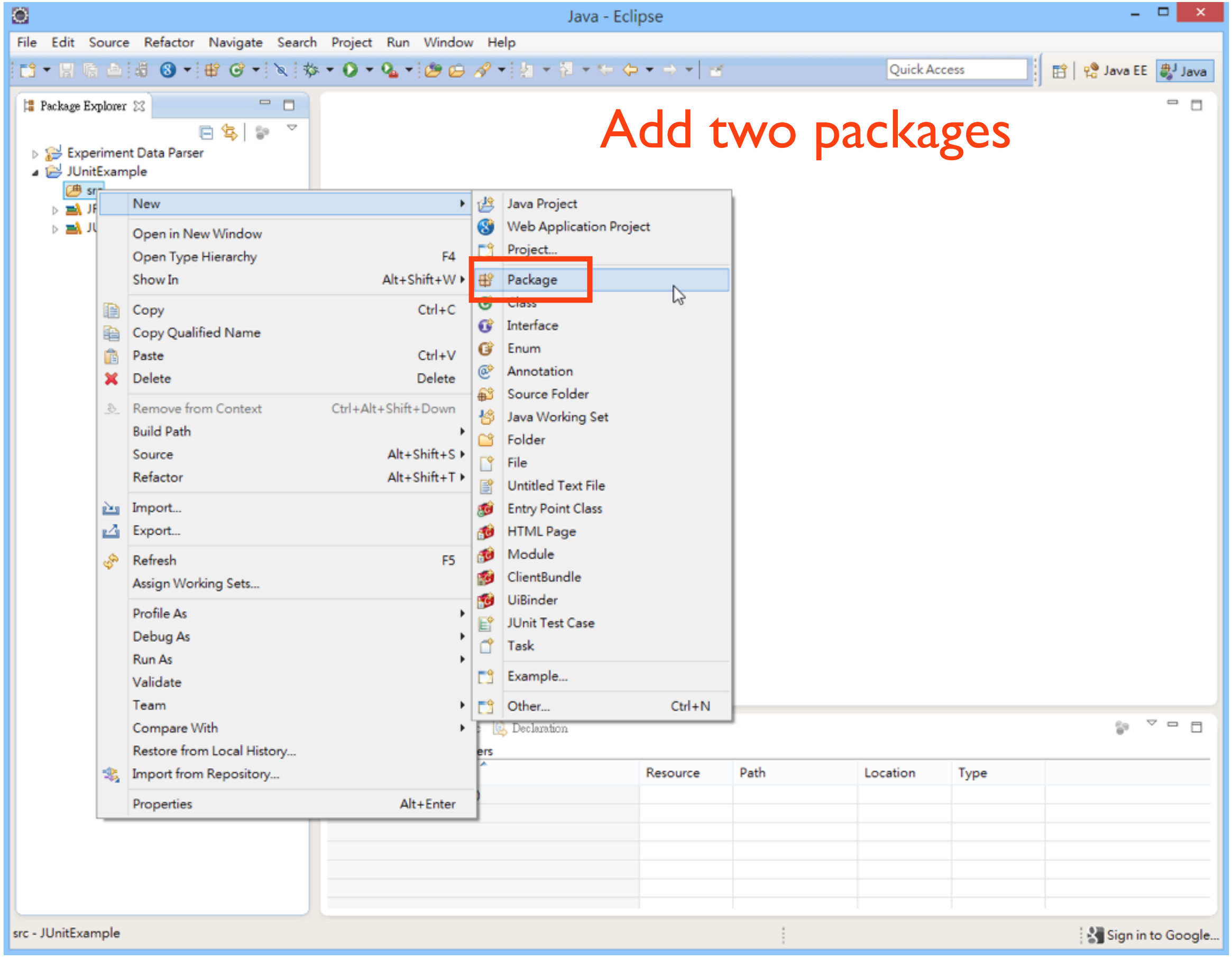

Add Library
Add Library
Select the library type to add.
Connectivity Driver Definition
CXF Runtime
EAR Libraries
Google App Engine
Google Web Toolkit
JRE System Library
JUnit
Maven Managed Dependencies
Plug-in Dependencies
Server Runtime
User Library
Web App Libraries
< Back
Next >
Finish
Cancel

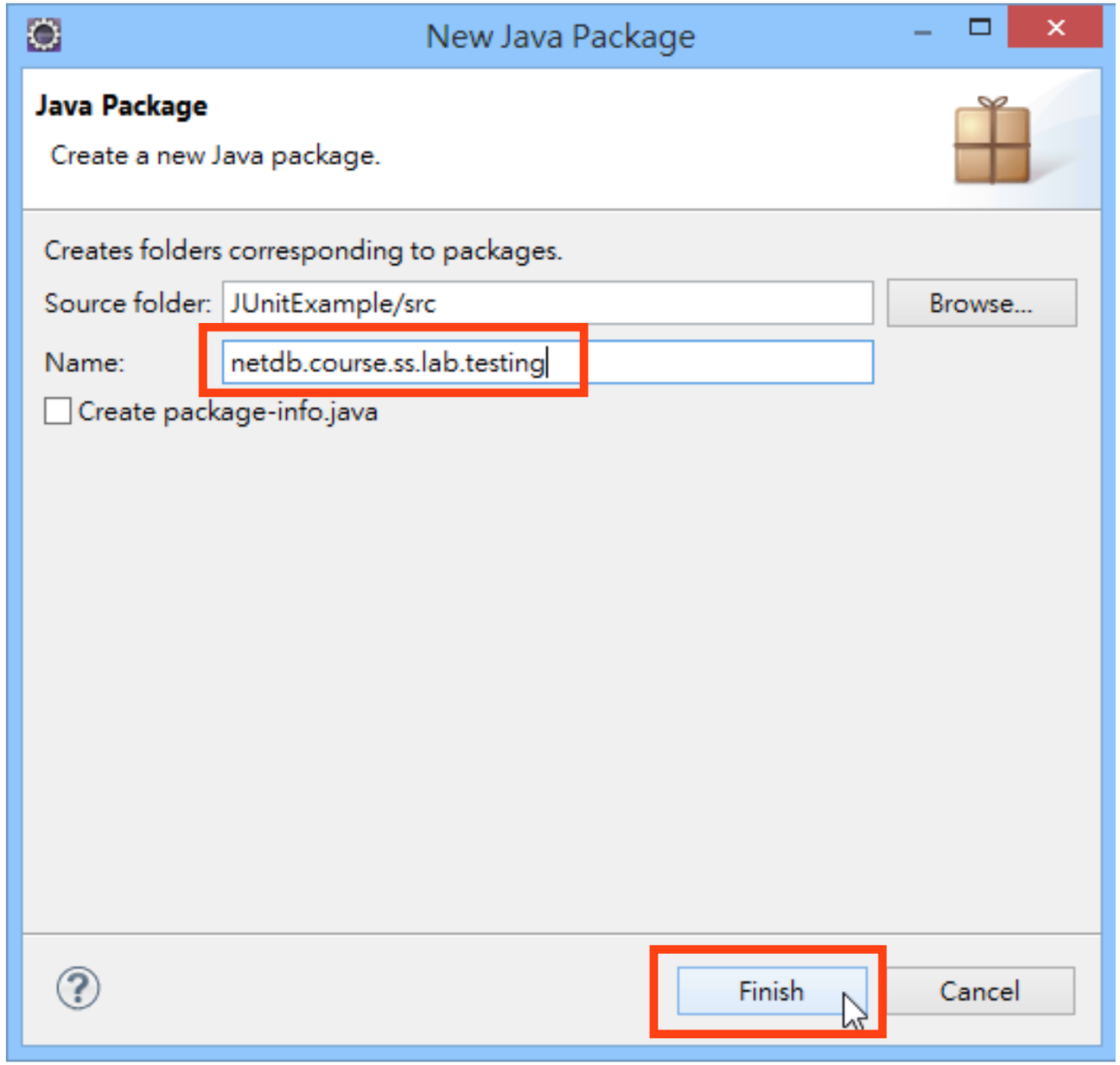

Add Library
JUnit Library
Select the JUnit version to use in this project.
JUnit library version:
JUnit 4
Current location:
junit.jar - D:\eclipse\plugins\org.junit_4.11.0.v201303080030
Source location:
Not found
< Back
Next >
Finish
Cancel


Properties for JUnitExample
type filter text
Resource
Builders
Google
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
Project Facets
Project References
Run/Debug Settings
Task Repository
Task Tags
Validation
WikiText
Java Build Path
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
JRE System Library [JavaSE-1.7]
JUnit 4
Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Add External Class Folder...
Edit...
Remove
Migrate JAR File...
OK
Cancel

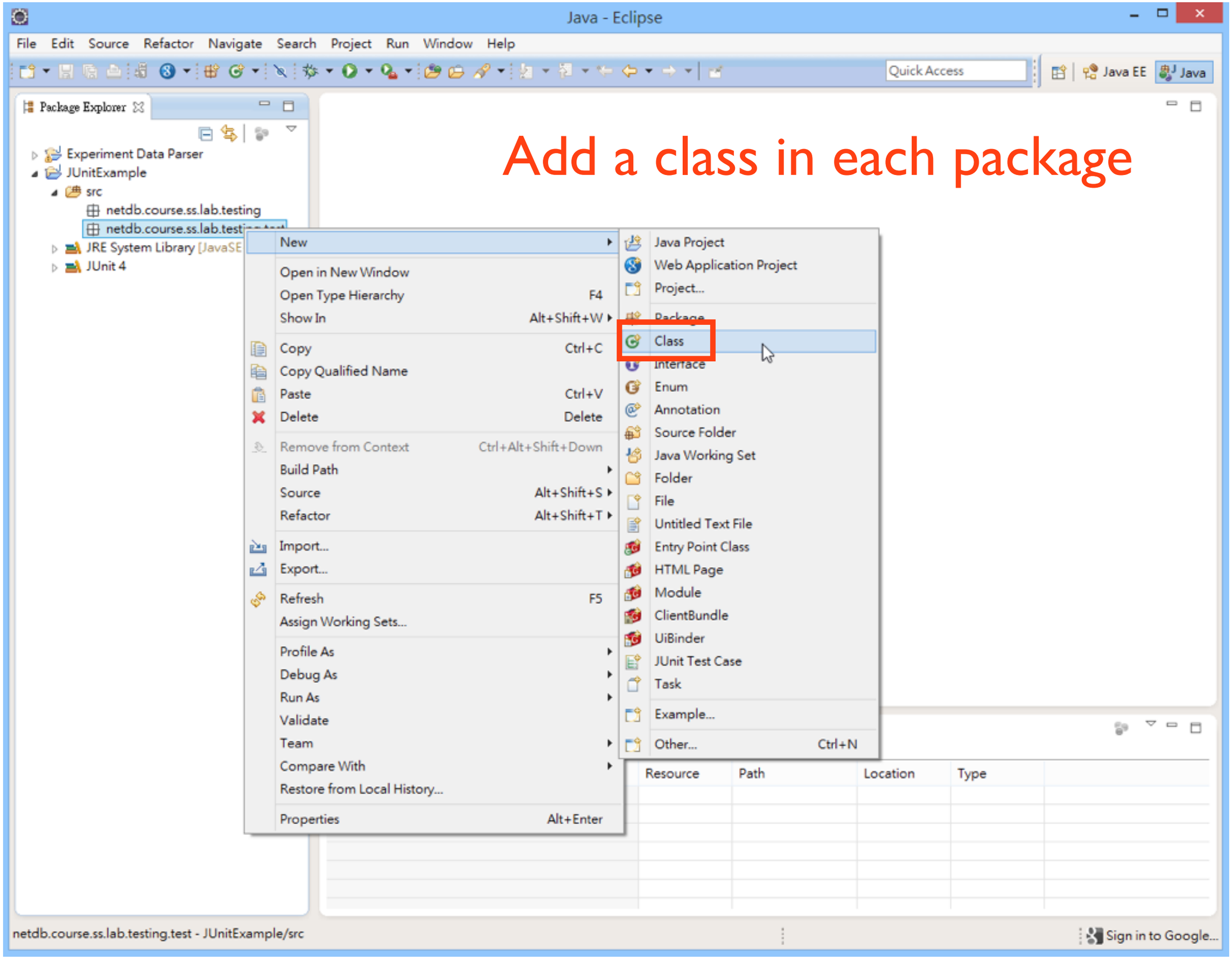
Create a Calculator

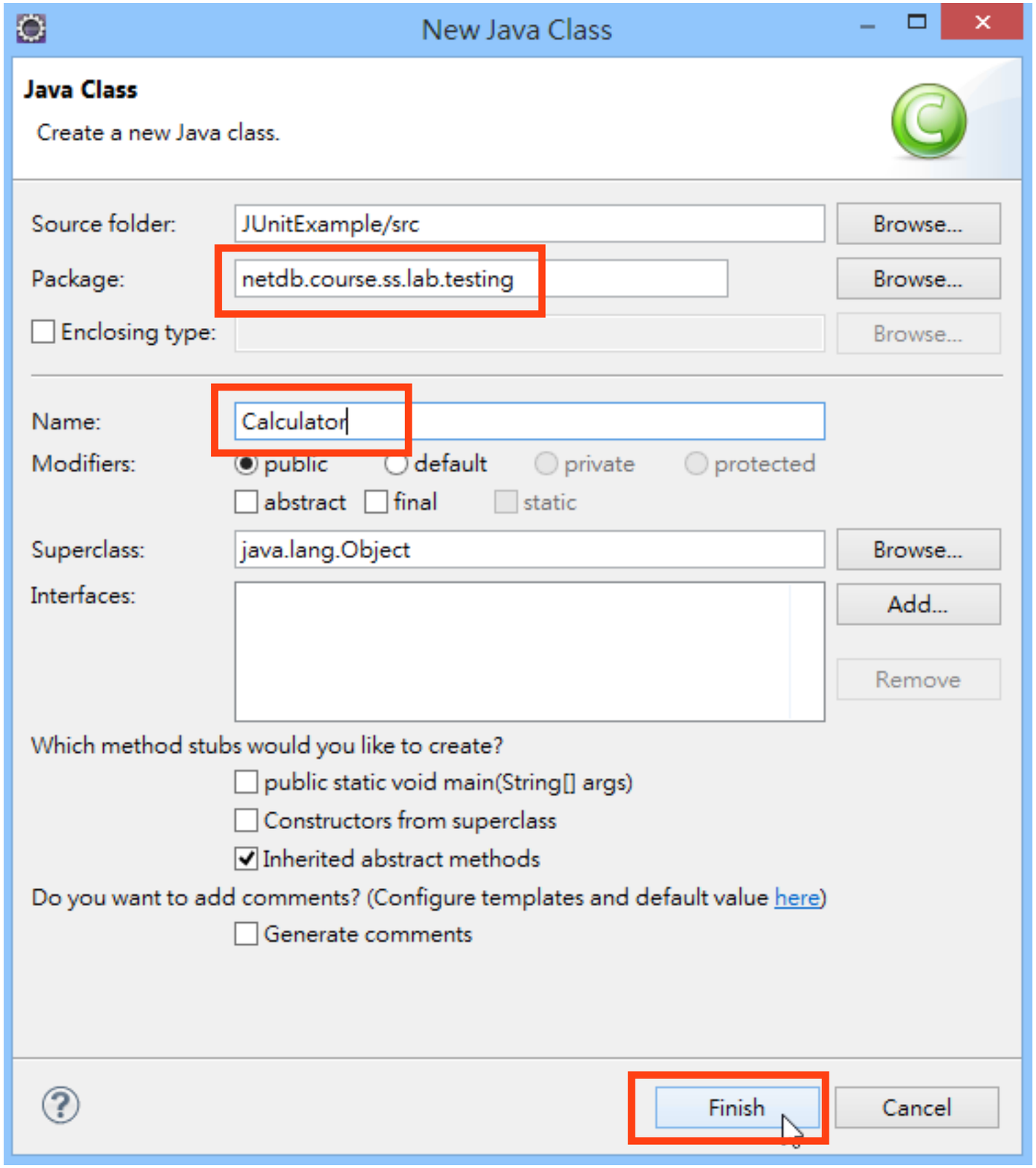
Add two packages




New Java Package
Java Package
Create a new Java package.
Creates folders corresponding to packages.
Source folder:
JUnitExample/src
Browse...
Name:
netdb.course.ss.lab.testing
Create package-info.java
Finish
Cancel

Add a class in each package



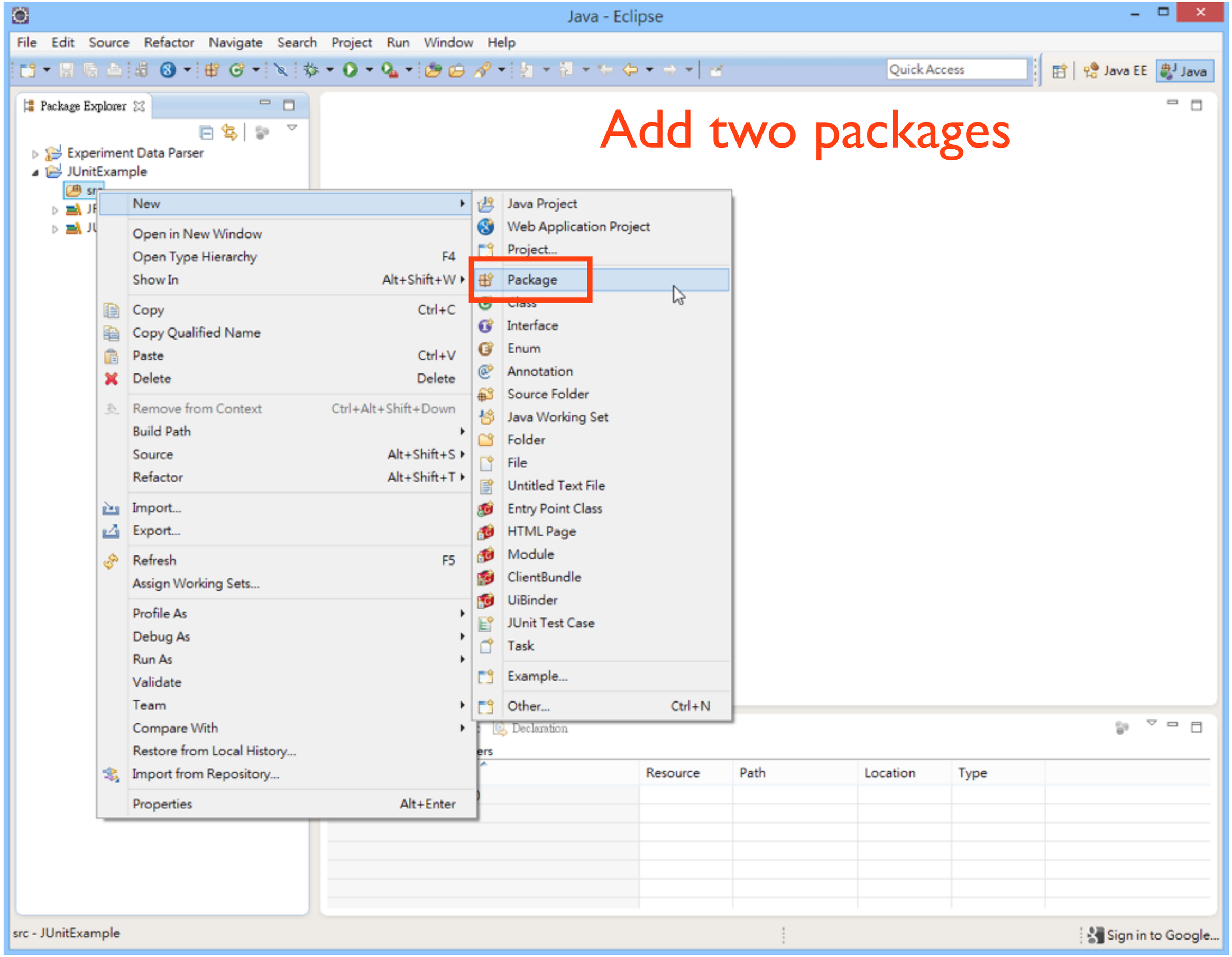

New Java Class
Java Class
Create a new Java class.
Source folder:
JUnitExample/src
Browse...
Package:
netdb.course.ss.lab.testing
Browse...
Enclosing type:
Browse...
Name:
Calculator
Modifiers:
public
default
private
protected
abstract
final
static
Superclass:
java.lang.Object
Browse...
Interfaces:
Add...
Remove
Which method stubs would you like to create?
public static void main(String[] args)
Constructors from superclass
Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
Generate comments
Finish
Cancel

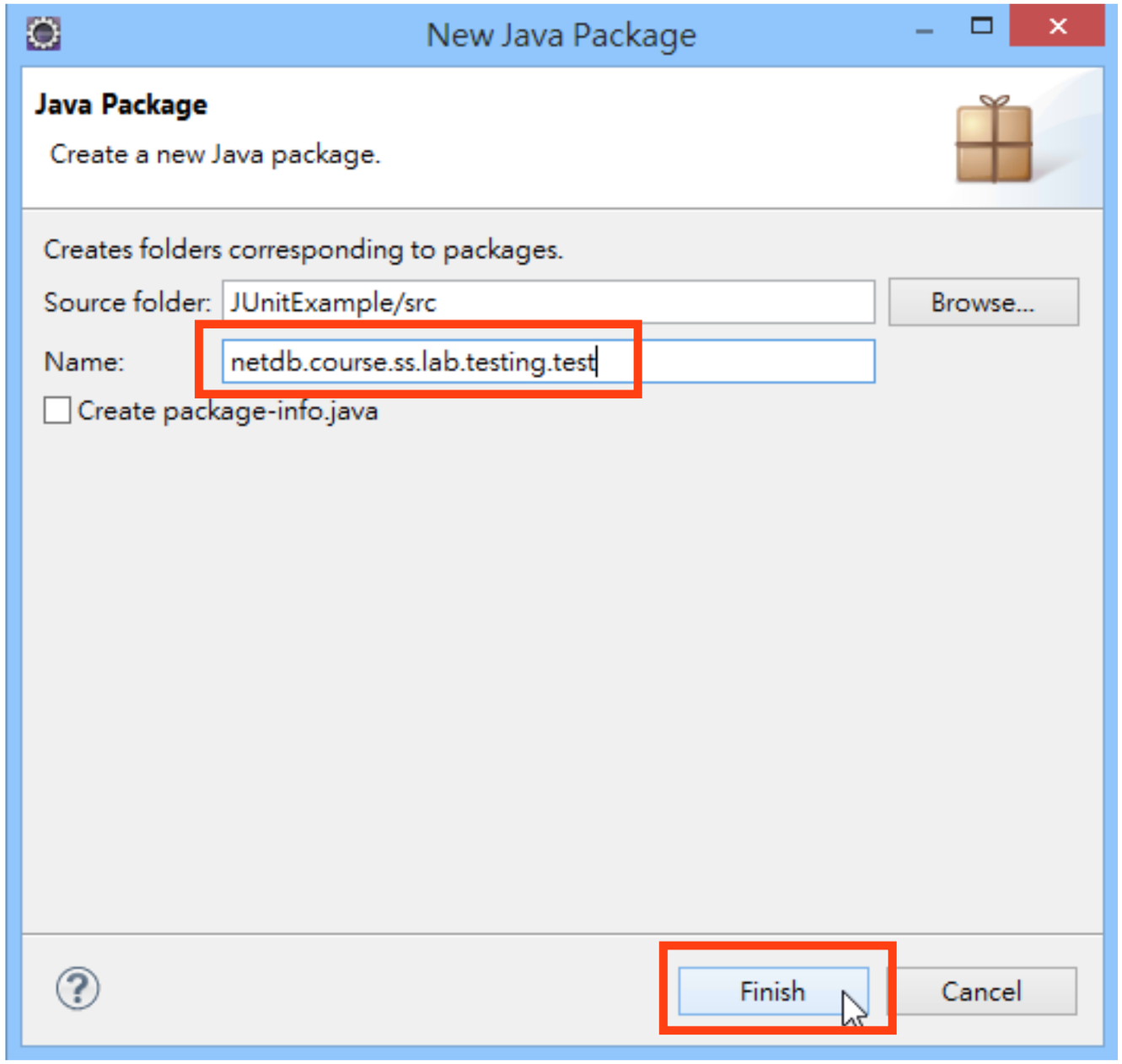
Calculator.java

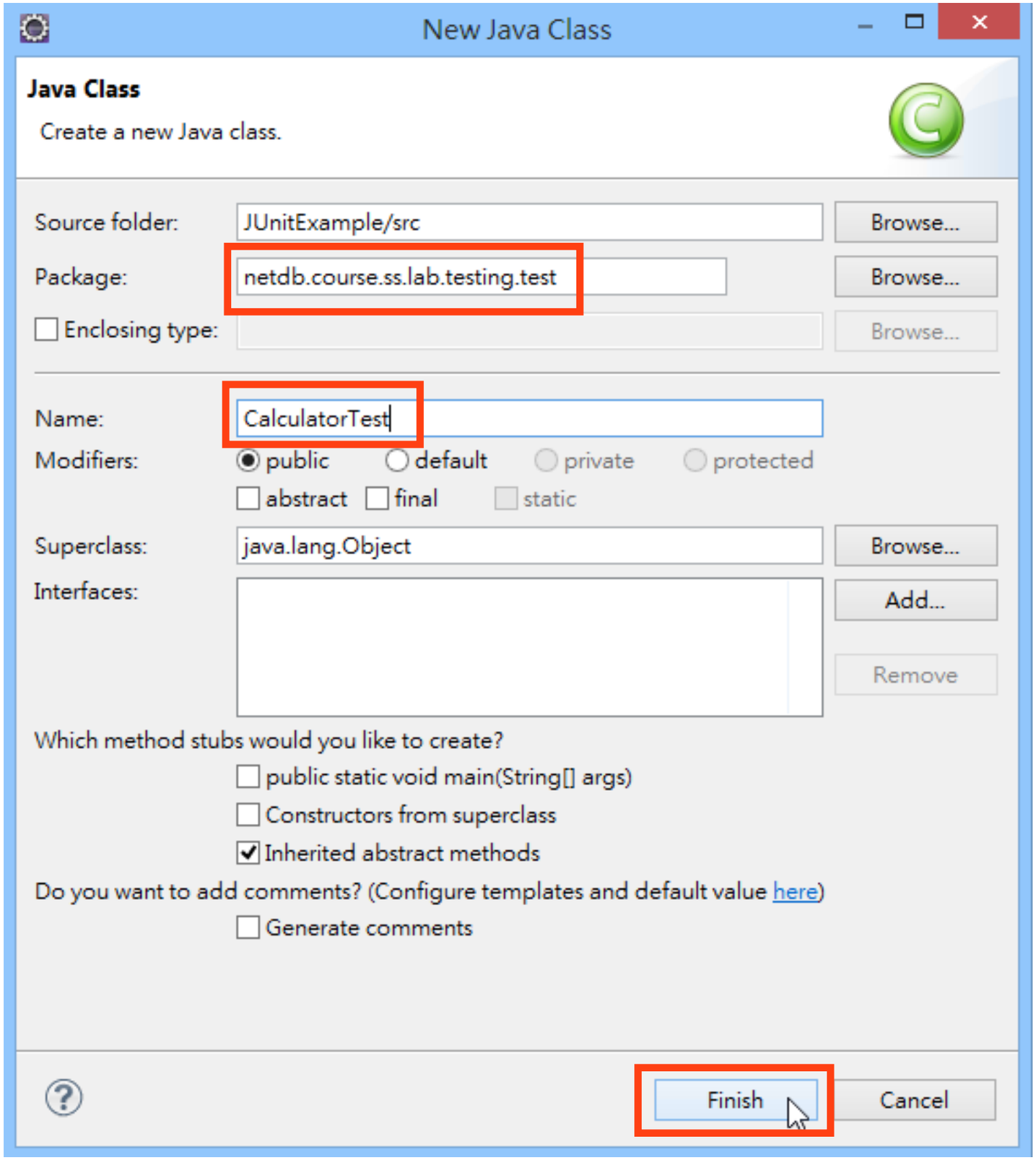
```
public class Calculator {
    public int plus(int op1, int op2) {
        return op1 + op2;
    }
    public int minus(int op1, int op2) {
        return op1 - op2;
    }
}
```

Create a Test Class

Add two packages




New Java Package
Java Package
Create a new Java package.
Creates folders corresponding to packages.
Source folder:
JUnitExample/src
Browse...
Name:
netdb.course.ss.lab.testing.test
Create package-info.java
Finish
Cancel


New Java Class
Java Class
Create a new Java class.
Source folder:
JUnitExample/src
Browse...
Package:
netdb.course.ss.lab.testing.test
Browse...
Enclosing type:
Browse...
Name:
CalculatorTest
Modifiers:
public
default
private
protected
abstract
final
static
Superclass:
java.lang.Object
Browse...
Interfaces:
Add...
Remove
Which method stubs would you like to create?
public static void main(String[] args)
Constructors from superclass
Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
Generate comments
Finish
Cancel

CalculatorTest.java

```
import org.junit.Assert;
import netdb.course.ss.lab.testing.Calculator;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
public class CalculatorTest {
     private Calculator calculator;

     @Before
     public void setUp(){
          calculator = new Calculator();
     }

     @After
     public void tearDown(){
          calculator = null;
     }

     @Test
     public void calculatorShouldPlusCorrectly(){
          int result = calculator.plus(10, 5);
          Assert.assertEquals(15, result);
     }

     @Test
     public void calculatorShouldMinusCorrectly(){
          int result = calculator.minus(10, 5);
          Assert.assertEquals(5, result);
     }

}
```

Outline

- Unit Testing
- Using JUnit 4 in Eclipse
- Today's Mission

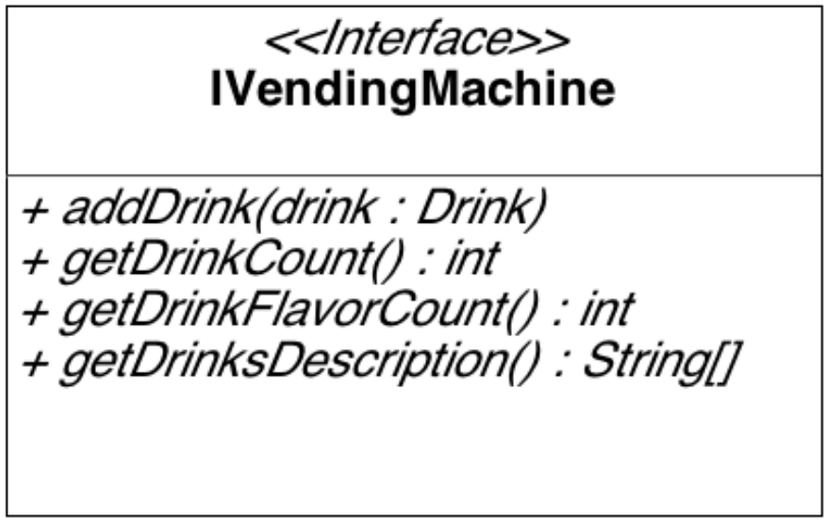
Today's Mission

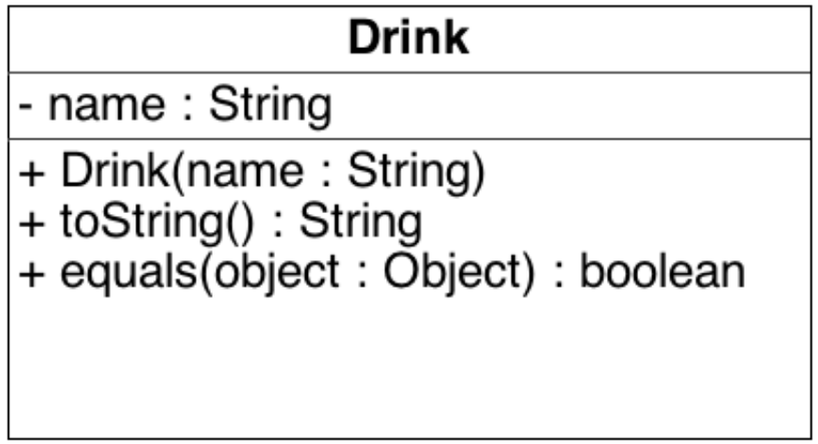
- Checkout the code from SVN http://netdb.cs.nthu.edu.tw/svn/courses/software_studio/2013_fall/samples/
- Today's lab is separated into 2 parts
  - The first part is mentioned in the following slides
  - The second part is mentioned in another files on the course page named "OnePiece"
    - The OnePiece is locked by a password
    - Finish the first part, and then the TAs will reveal the OnePiece to you

Today’s Mission

- In the first part:
  - You are going to implement a Vending Machine which have 4 functions
    - Add Drink
    - Get Drink Count
    - Get Drink Flavor Count
    - List All Drinks

UML


<<Interface>>
IVendingMachine
+ addDrink(drink : Drink)
+ getDrinkCount() : int
+ getDrinkFlavorCount() : int
+ getDrinksDescription() : String[]


Drink
- name : String
+ Drink(name : String)
+ toString() : String
+ equals(object : Object) : boolean

Writing the Test First

- You need to write the test cases before writing the code
- 3 test cases should be written
  - itShouldReturnCorrectDrinkCount
  - itShouldReturnCorrectDrinkFlavorCount
  - itShouldReturnCorrectDrinkDescription
- Remember to write the @Before and @After method

itShouldReturnCorrectDrinkDescription

- (This test case is already written for you)
- Add one “Cola” into the machine
- Assert that the machine.getDrinkDescription[0] = “Cola”

itShouldReturnCorrectDrinkCount

- New and add one “Cola” into the machine
- New and add one “Juice” into the machine
- New and add one “Cola” into the machine
- Assert that the machine.getDrinkCount = 3

itShouldReturnCorrectDrinkFlavorCount

- New and add one “Cola” into the machine
- New and add one “Juice” into the machine
- New and add one “Cola” into the machine
- Assert that the machine.getDrinkFlavorCount = 2

Hint

- You can use Assert.assertxxxx to do the test
  - ex. Assert.assertEquals(<expected>,<actual>)
- You can use ArrayList to store the added drinks
  - Reference
- You can use equals method to check if two drinks are the same flavor
- To get the drink's description, you can use drink.toString()