OODP– Session 8

Session times

PT	– Thursday 18:00-21:00	room: Malet 407
FT	- Tuesday 13:30-17:00	room: Malet 404

Email: oded@dcs.bbk.ac.uk

Web Page: http://www.dcs.bbk.ac.uk/~oded

Visiting Hours: Tuesday 17:00 to 19:00

Mocking

Why do we need to mock?

Recall:

- Programming begins when most if not all of the code has yet to be written.

Problem:

- How can we do anything more than trivial testing?

Solution:

1. ~~Do something to the classes not yet implemented~~

Why not? Some of them may not be yours, so writing them is waste of time

2. Mock objects

So we need to mock?

What do we want?

- Simple
- Fast
- Easy to learn

How:

- Mockito - framework for creating mock objects

Remember there are other options such as: Jmock, EasyMock,...

Why choose Mockito?

1. Can easily tell mock methods what to return when invoked
2. Has easily accessible mechanisms for checking how the mocked object was treated
3. Mockito can be used for spying (we won't do that) on implemented objects

Next

1. We download Mockito and insert it into our infrastructure
2. We prepare an example for learning Mockito by implementing.
3. We pick some of Mockito’s features and try them out.
4. We start by creating a project and adding the Mockito JAR to it

Creating a Project And Adding Mockito JAR

Mockito Download

Mockito is library downloadable in Java Archive format (JAR)
(JAR is typically used in order to aggregate Java class files)

Downloads list

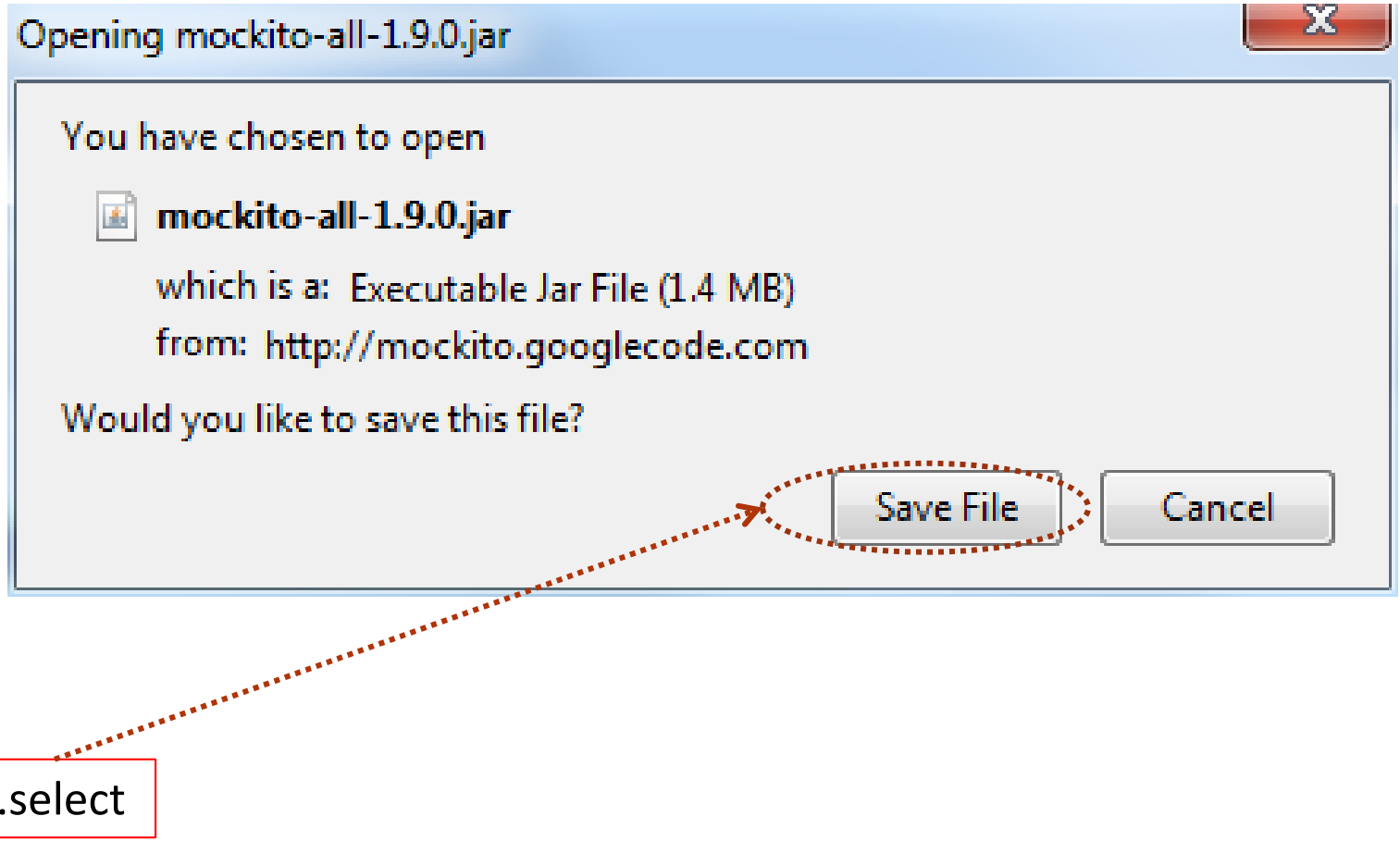
http://code.google.com/p/mockito/downloads/list

We use: *mockito-all-1.9.0.jar*

http://code.google.com/p/mockito/downloads/detail?name=mockito-all-1.9.0.jar&can=2&q=

Downloading Mockito

Mockito Download

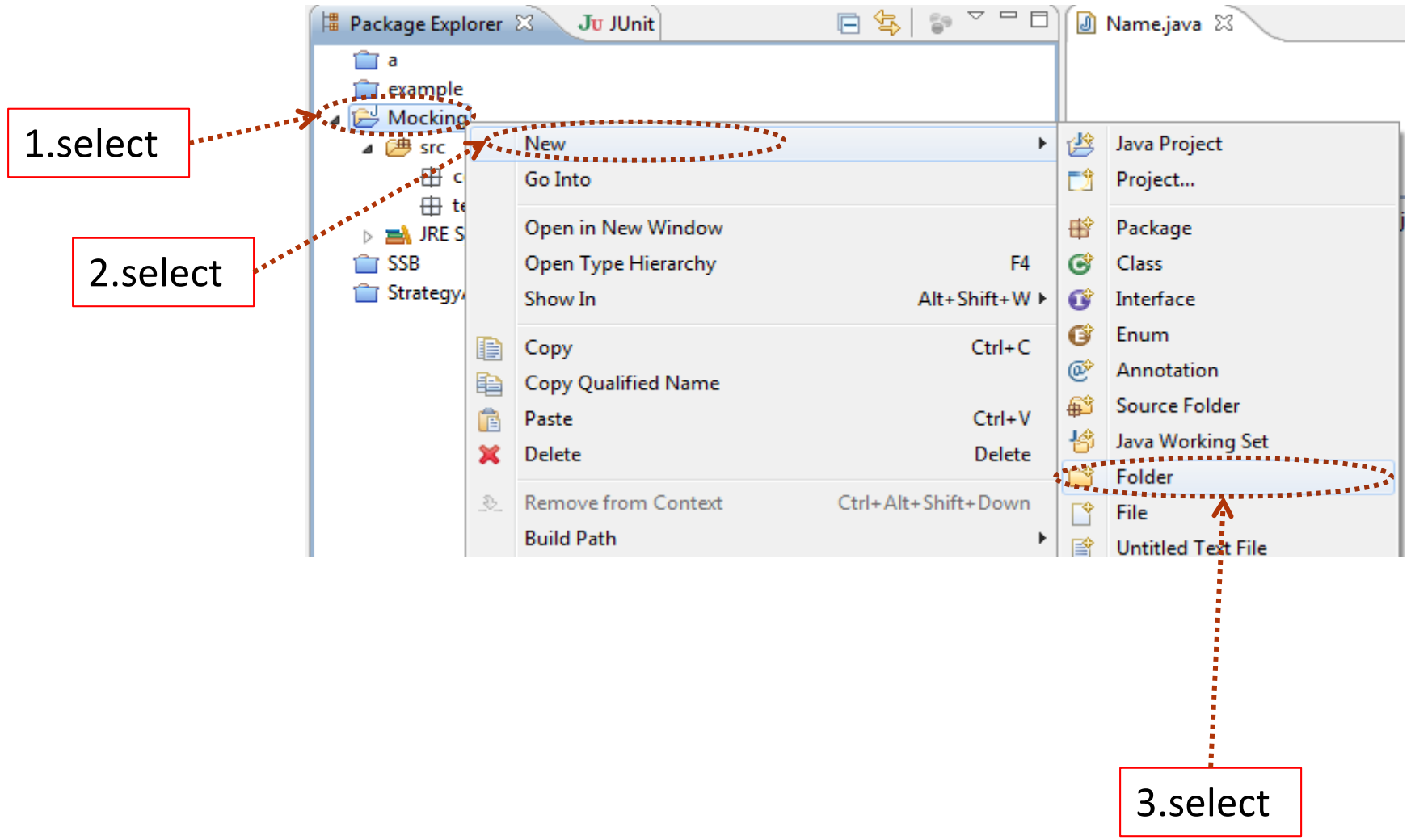


New Project

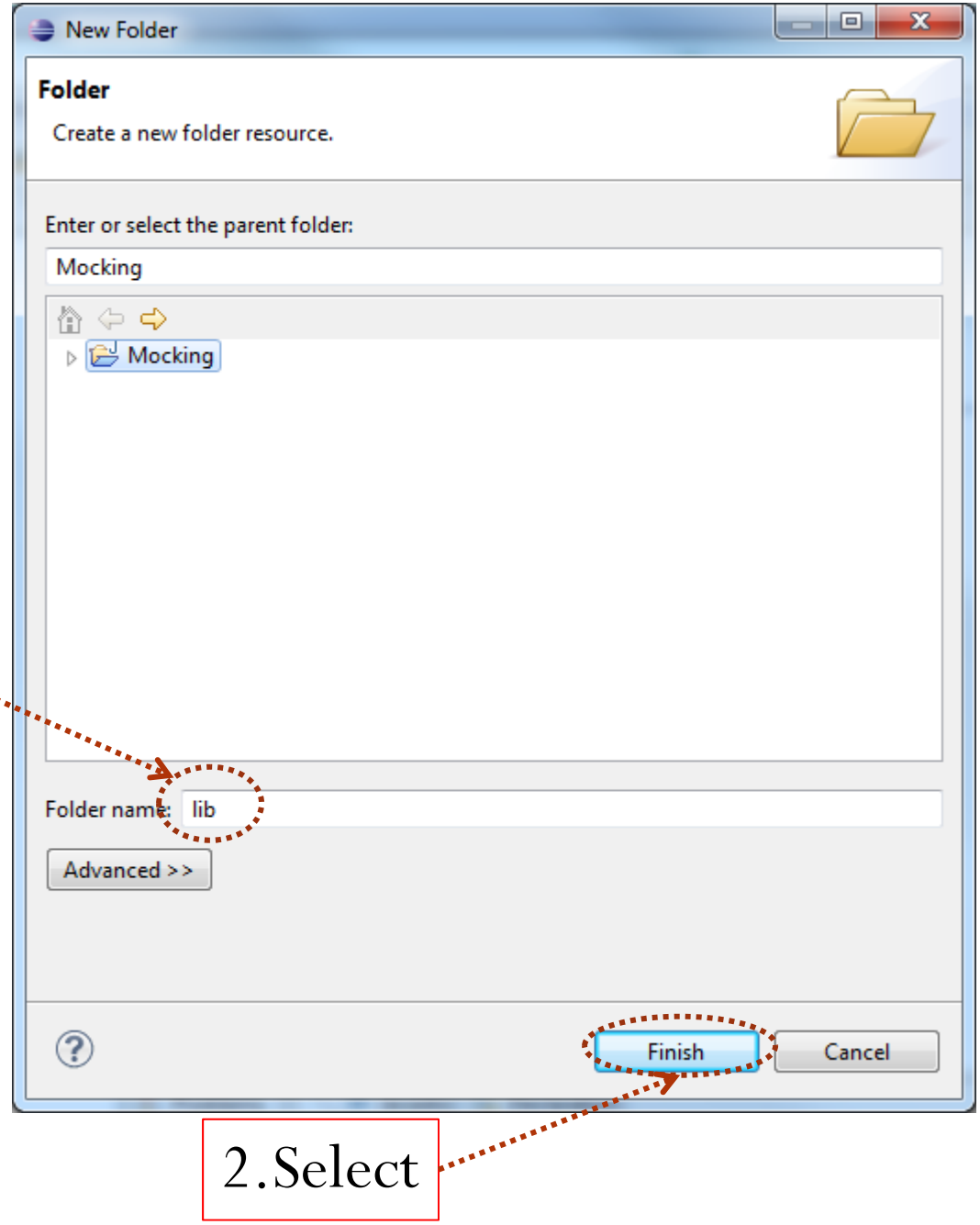
1. Close all previous projects.
2. Open a new project. Call it ‘Mocking’.
3. Open new packages, ‘code’ and ‘tests’

Next we are going to add ‘Mockito JAR’ top the project

Parameter Passing

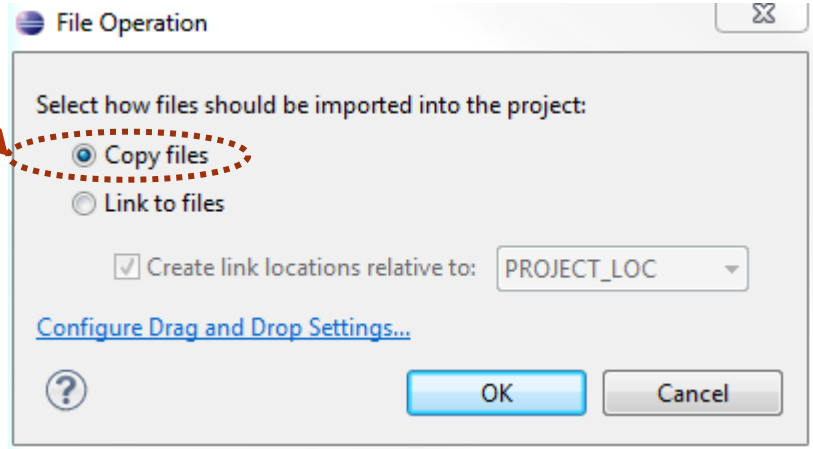


Folder menu


New Folder
Folder
Create a new folder resource.
Enter or select the parent folder:
Mocking
Mocking
Folder name:
lib
Advanced >>
Finish
Cancel
1.Fill In
2.Select

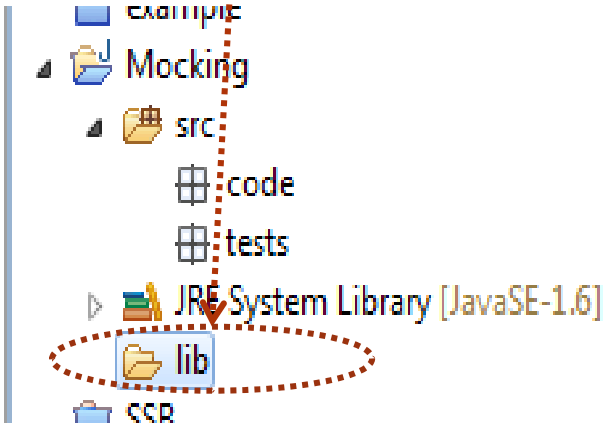
Adding Mockito JAR step 1

2.Select

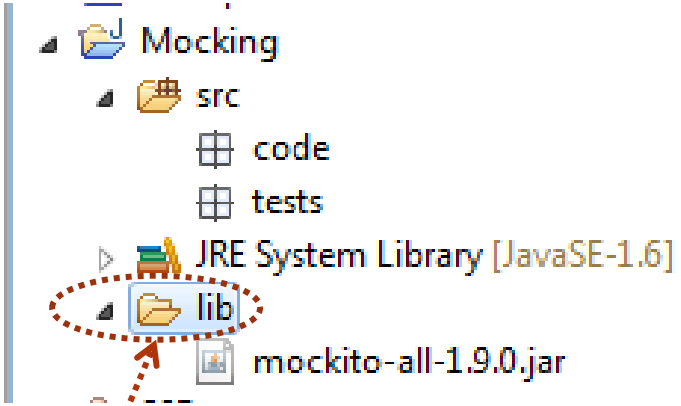


Drag JAR Drop

Before



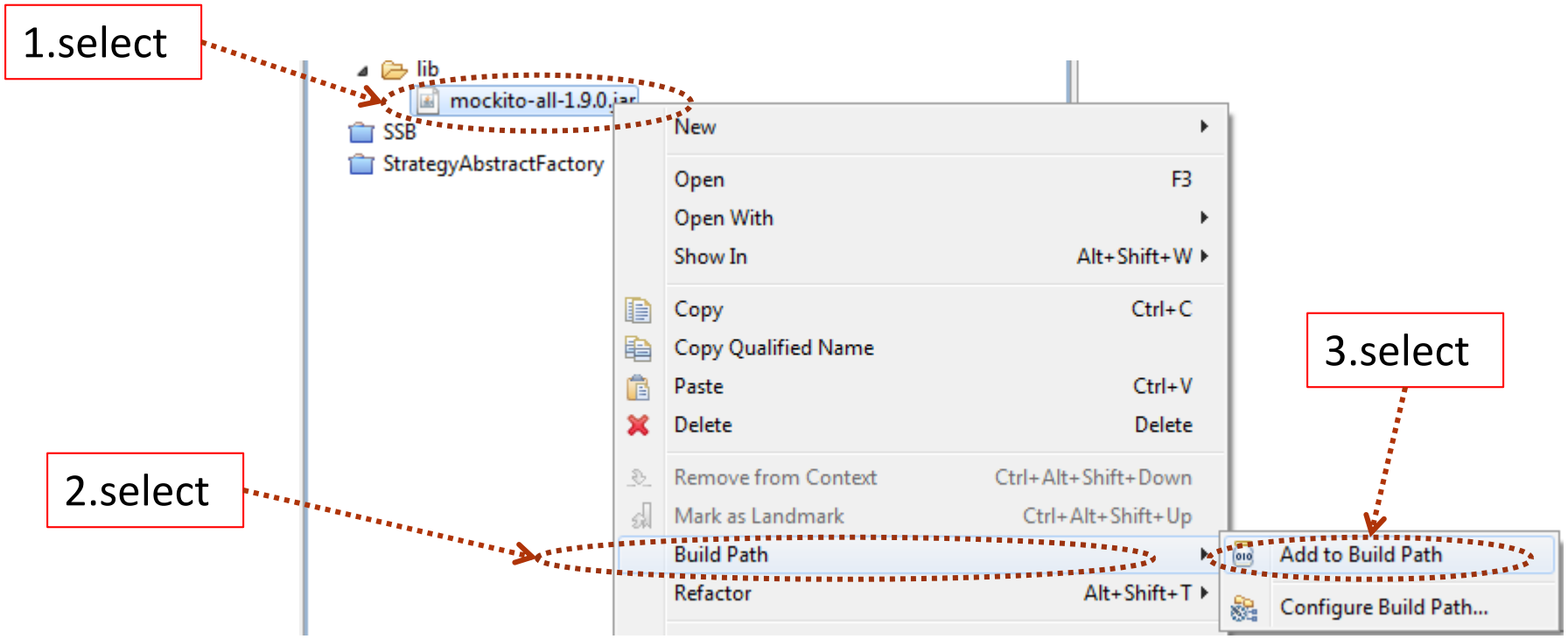
After



(3. Select if did not appear)

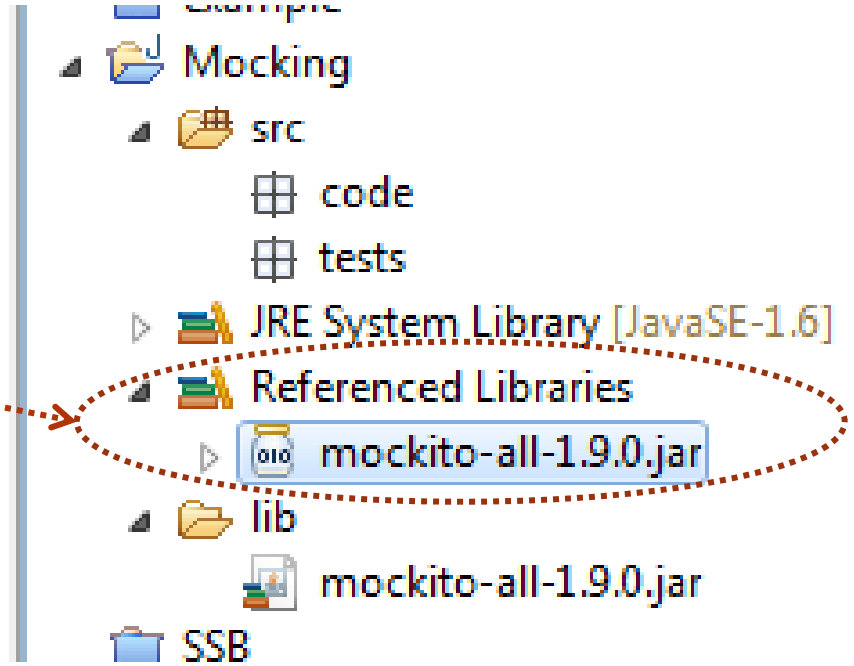
Adding to Build Path

Not enough to add to Mockito JAR to lib directory. Eclipse needs to know where it is located.



After adding to Build Path

Now it also appears in referenced libraries.



Simple Pre-prepared Code

New Project

1. Create new Junit test. Call it ‘CounterTest’.
2. Code for test

```
@Test
public void test() {

        Integer first;
        Integer expectedFirst =1;

        Counter count = new Counter();

        first = count.getValue();

        assertEquals("Wronge Answer !",expectedFirst, first);
}
```

3. Use test code to generate Counter Stub

New Project

1. To ensure that we can use Mockito library add the following line to the test file (should be added at the top with the other imports)

```
import static org.mockito.Mockito.*;
```

”import static” – required since the mockito code is not in the package and we also want to use static members, the easy way.

New Project

1. Code for ‘counter’ class

```
public class Counter {

	private Integer count;

	public Counter() {
		super();
		this.count = 1;
	}

	public Integer getValue() {

		return count++;
	}

}
```

3. Try to generate code by yourself (or copy from slide)
4. Run test if passes commit

Creating a Mock Counter

What next

1. We create a Mock ‘counter’ object using the Mockito library

(It should require minimal codeing)

2. To understand how it works we shall compare it to the real ‘counter’ object that we have prepared in advance.
3. Next slide the actual code is given

Next we are going to add ‘Mockito JAR’ top the project

Test with Mock 'Counter' object

```
@Test
public void test() {

        Integer first,mockedFirst;
        Integer expectedFirst =1;

        Counter count = new Counter();

        Counter mockedCounter = mock(Counter.class);
        when(mockedCounter.getValue()).thenReturn(1);

        first = count.getValue();
        mockedFirst = mockedCounter.getValue();

        assertEquals("Wronge Answer !",first , mockedFirst);
}
```

Next slide we explain new code added
(try it out first)

New Project

This tells Mockito to create a class of type mock

```
Counter mockedCounter = mock(Counter.class);
```

This tells Mockito that when this method is called

It should return this

```
when(mockedCounter.getValue()).thenReturn(1);
```

Mocked class interfaced like the real thing

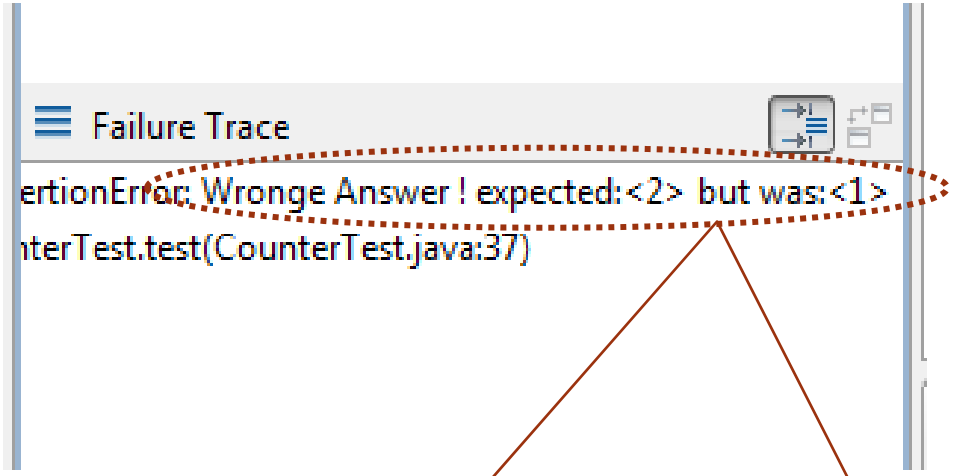
```
mockedFirst = mockedCounter.getValue();
```

Pushing Mockito over the edge

1. New test

```
@Test
public void test() {

        Integer first,mockedFirst;
        Integer expectedFirst =1;

        Counter count = new Counter();

        Counter mockedCounter = mock(Counter.class);
        when(mockedCounter.getValue()).thenReturn(1);


        first = count.getValue();
        mockedFirst = mockedCounter.getValue();

        assertEquals("Wronge Answer !",first , mockedFirst);

        first = count.getValue();
        mockedFirst = mockedCounter.getValue();

        assertEquals("Wronge Answer !",first , mockedFirst);
}
```

After Running test



Mockito new the first answer (“1”) but kept answering the same answer in the second time.

Regretfully this was the wrong answer – what can you expect from a Mock

- Next we change the code so that the test passes.
- We do so by telling Mockito what the second answer should be.

Giving Mockito the second answer

1. New test

```
@Test
public void test() {

        Integer first,mockedFirst;
        Integer expectedFirst =1;

        Counter count = new Counter();

        Counter mockedCounter = mock(Counter.class);
        when(mockedCounter.getValue()).thenReturn(1).thenReturn(2);


        first = count.getValue();
        mockedFirst = mockedCounter.getValue();

        assertEquals("Wronge Answer !",first , mockedFirst);

        first = count.getValue();
        mockedFirst = mockedCounter.getValue();

        assertEquals("Wronge Answer !",first , mockedFirst);
}
```

- Next –
  - Now you change the code to check what the third answer is wrong

Verifying that

The Mock Object

Got Proper Treatment

Do either (1) or (2)

1. Have at least 3 method calls to ‘getvalue()’ of ‘MockedCounter’
2. Clone code from

   git://github.com/odedlac/Mocking.git

Next we are going to add ‘Mockito JAR’ top the project

Test code

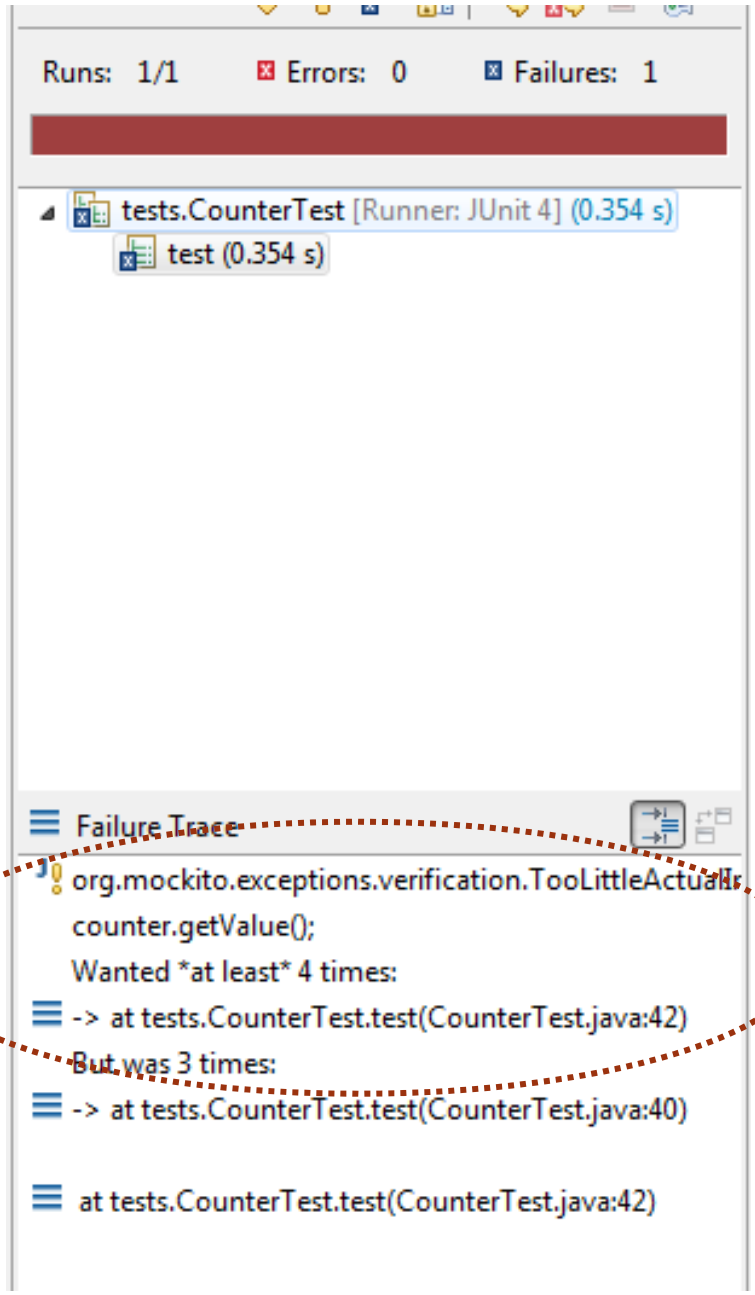
```
@Test
public void test() {

            Integer first,mockedFirst;

            Counter count = new Counter();

            Counter mockedCounter = mock(Counter.class);
            when(mockedCounter.getValue()).thenReturn(1).thenReturn(2);

            first = count.getValue();
            mockedFirst = mockedCounter.getValue();

            assertEquals("Wronge Answer !",first , mockedFirst);

            first = count.getValue();
            mockedFirst = mockedCounter.getValue();

            assertEquals("Wronge Answer !",first , mockedFirst);
            first = count.getValue();
            mockedFirst = mockedCounter.getValue();

            verify(mockedCounter,atLeast(4)).getValue();

            assertEquals("Wronge Answer !",first , mockedFirst);
}
```

If you don't have this code add it.
Then, run the test

This tells Mockito to check that 'getValue' has been called at least 4 times

Verify test failed

Test failed as expected – fix it



Verifying Number of Invocations

*verify(mockedCounter,**times(3)**).getValue();*

Verify that *getValue* method of *mockedCounter* was invoked 3 times.

Other options:

- *verify(mockedCounter,**atLeast(2)**). getValue();* - *at least 2 times*
- *verify(mockedCounter,**atMost(4)**). getValue();* - *at most 4 times*
- *verify(mockedCounter,**atLeastOnce()**). getValue();* - *at least once*
- *verify(mockedCounter,**never()**). getValue();* - *never*

Can you think of a simple way to do this without ***Mockito***

Next

- Write tests that demonstrate each of the ‘verify’ option

“demonstrate” means that you should show examples in which the test fails and examples in which the test passes)

If you get stuck or do not understand something please ask me.

Returning a value that Depends on Parameters

Value at Invocation

What have we done until now

- Created a mock objects with mockito
- Told the mock object what to return on first and second invocation of method “getValue”

What next

- We can set the value returned to depend on the parameters of the method called.

Next we shall see how

Import:

git://github.com/odedlac/Mocking3.git

New Project

This tells Mockito what to answer on invocation with ‘1’

```
when(mockedCounter.getValue(1)).thenReturn(1).thenReturn(2);
```

This tells Mockito what to answer on invocation with ‘2’

```
when(mockedCounter.getValue(2)).thenReturn(2).thenReturn(4);
```

Next

- Fix the test so that it works
- Extend the test to check the third invocation
- What happens if the mock method is called with a value not defined, say ‘3’? (why?)

If you get stuck or do not understand something please ask me.

Returning a value that Does not depend on Parameters value at Invocation (argument matchers)

What if we do not want the values returned by the method to depend on the parameters values at invocation?

Solution:

Argument matchers

'anyInt()', anyString(), anyObject()

and many more see:

http://docs.mockito.googlecode.com/hg/latest/org/mockito/Matchers.html

Example import from:

git://github.com/odedlac/Mocking5.git

In order with one mocked object

```
@Test
public void test1() {

Integer i = 1;
String one = "1";

Example example = new Example();
Example mockedExample = mock(Example.class);

when(mockedExample.isSame(anyInt(),anyString())).thenReturn(true).thenReturn(false);

Boolean same = example.isSame(i, one);
Boolean mockedSame = mockedExample.isSame(i, one);

assertEquals("Wrong Answer !", same, mockedSame);

}
```

Instead of specifying the exact values,
we used argument matchers

Next

- Write a similar test, check if argument matchers can be mixed with actual values (this shouldn’t work)

- Try to write test that demonstrate other argument matchers from the list at the given URL.

If you get stuck or do not understand something please ask me.

Verifying Order of Invocations

- We would like to do something of the sort

1. Check that the mocked ‘getValue’ method was invoked first with ‘1’ and then with ‘2’
2. We would also like to do similar things with more than one mocked object
3. We start by getting the example

Import:

git://github.com/odedlac/Mocking4.git

In order with one mocked object

```
@Test
public void test1() {
        // Testing in order with only one object

        // Mocked object
        Counter mockedCounter = mock(Counter.class);

        // Four mock object invocations
        mockedCounter.getValue(1);
        mockedCounter.getValue(2);

        // created an in order verifier for mockedCounter
        InOrder inOrder = inOrder(mockedCounter);

        inOrder.verify(mockedCounter).getValue(1);
        inOrder.verify(mockedCounter).getValue(2);

}
```

This is an object required by the Mockito api in order to check in order

This tells mockito to check whether 'getvalue' was invoked first with '1' and then with '2'

In order with two mocked objects

```
@Test
public void test2() {
// Testing in order with only one object

// Two mocked objects
Counter mockedCounter1 = mock(Counter.class);
Counter mockedCounter2 = mock(Counter.class);

// Four mock object invocations
mockedCounter1.getValue(1);
mockedCounter2.getValue(2);

// created an in order verifier for mockedCounter1 and mockedCounter2
InOrder inOrder = inOrder(mockedCounter1,mockedCounter2);

inOrder.verify(mockedCounter1).getValue(1);
inOrder.verify(mockedCounter2).getValue(2);

}
```

This time we give to mock objects to inOrder

This tells mockito to check whether 'getvalue' of 'mockedCounter1' was invoked first with '1' and and then 'getvalue' of 'mockedCounter2' was invoked with '2'

Next

- Extend the tests one and two to more invocation
- Write test demonstrating how ‘inOrder’ fails

If you get stuck or do not understand something please ask me.

Some Java Exceptions And Mocking Them

Java Exceptions

What do you do when things don't work?

Obviously ,

handle them discretely.

To do that you need

'**exceptions'**.

Java Exceptions

We start with an example

Import:

git://github.com/odedlac/MockingExceptions.git

Extra reading about exceptions
http://tutorials.jenkov.com/java-exception-handling/basic-try-catch-finally.html

Divider class with *try catch* block

```
public class Divider {

        public Integer divide(Integer i, Integer j) throws IllegalArgumentException{
                Integer k = 0;
                try {
                        k = i / j;
                } catch (Exception IllegalArgumentException) {
                        System.out.println("Division by zero");
                        throw new IllegalArgumentException();
                }
                return k;
        }
}
```

Type of Exception caught

Try catch block
(can be added automatically with right click mouse and surround with option)

Let Java continue dealing with this by artificially throwing an exception

Declaration that method throws exception

Exception handling

Important: Exception handling is not instead of testing

It is used when thing are out of your control. For example:

1. Can't open a file
2. Wrong input

Etc.

Note: there is more to exception handling

Like in other cases you can make Eclipse do a lot of the work for you.

Divider class with *try catch* block

```
@Test(expected = IllegalArgumentException.class)
public void test2() {
        // Test with exception mocking
        Integer firstInput = 6;
        Integer secondInput = 0;

        Divider mockedDivider = mock(Divider.class);

        doThrow(newIllegalArgumentException()).when(mockedDivider).divide(anyInt(),eq(0));

        mockedDivider.divide(firstInput,secondInput);

}
```

Test that exception happened (part of Junit)

Standard use of argument matchers

Tell Mockito to throw exception

Exception handling

See what happens if we don’t throw an exception in the divide(method)

and instead just return -1.

Rapping up

Will be added if done in class

Other things Mockito can do for us

- **Spying** – real methods are called (unless stubbed), not recommended.
- **Real partial mocks** – like spying but slightly different
- **Reset mocks** - *reset(mockObject)*
- **Stubbing with generic interface** – *not recommended*

Bottom line

- The goal of using **Mockito** is to replace code with much simpler code, if you start doing complicated things you miss the point.

More information at:
http://mockito.googlecode.com/svn/branches/1.6/javadoc/org/mockito/Mockito.html

Other Mock Libraries

- **EasyMock**
- **PowerMock**
- **JMock**
- **JMockit**

More mockito information

http://docs.mockito.googlecode.com/hg/latest/org/mockito/Mockito.html#3