MOCKITO PROGRAMMING COOKBOOK

Hot Recipes for Mockito Development

Java Code Geeks
Java 2 Java Developers Resource Center

Mockito Programming Cookbook

Contents

Preface

Mockito is an open source testing framework for Java released under the MIT License. The framework allows the creation of test double objects (mock objects) in automated unit tests for the purpose of Test-driven Development (TDD) or Behavior Driven Development (BDD).

In software development there is an opportunity of ensuring that objects perform the behaviors that are expected of them. One approach is to create a test automation framework that actually exercises each of those behaviors and verifies that it performs as expected, even after it is changed. Developers have created mock testing frameworks. These effectively fake some external dependencies so that the object being tested has a consistent interaction with its outside dependencies. Mockito intends to streamline the delivery of these external dependencies that are not subjects of the test. A study performed in 2013 on 10,000 GitHub projects found that Mockito is the 9th most popular Java library. (https://en.wikipedia.org/wiki/Mockito)

In this ebook, we provide a compilation of Mockito programming examples that will help you kick-start your own web projects. We cover a wide range of topics, from initialization and simple test cases, to integration with JUnit, Maven and other frameworks. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

JCGs (Java Code Geeks) is an independent online community focused on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike.

JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

You can find them online at https://www.javacodegeeks.com/

Chapter 1

Mockito Tutorial for Beginners

Mocking is a testing technique widely used not only in Java, but in any other object oriented programming language, that consists in exchanging . There are several mocking testing frameworks for Java, but this tutorial will explain how to use Mockito, probably the most popular for Java language.

For this tutorial, we will use:

- Java 1.7.0
- Eclipse Mars 2, release 4.5.2.
- JUnit 4.
- Mockito 1.10.19.

1.1 What is mocking?

Mocking is a testing technique where real components are replaced with objects that have a predefined behavior (mock objects) only for the test/tests that have been created for. In other words, a mock object is an object that is configured to return a specific output for a specific input, without performing any real action.

1.1.1 Why should we mock?

If we start mocking wildly, without understanding why mocking is important and how can it help us, we will probably put on doubt the usefulness of mocking.

There are several scenarios where we should use mocks:

- When we want to **test a component that depends on other component, but which is not yet developed**. This happens often when working in team, and component development is divided between several developers. If mocking wouldn't exist, we would have to wait until the other developer/developers end the required component/component for testing ours.
- When the **real component performs slow operations**, usual with dealing with database connections or other intense disk read/write operations. It is not weird to face database queries that can take 10, 20 or more seconds in production environments. Forcing our tests to wait that time would be a considerable waste of useful time that can be spent in other important parts of the development.
- When there are **infrastructure concerns that would make impossible the testing**. This is similar in same way to the first scenario described when, for example, our development connects to a database, but the server where is hosted is not configured or accessible for some reason.

1.2 Project creation

Go to "File/New/Java Project". You will be asked to enter a name for the project. Then, **press "Next", not "Finish"**.

In the new window that has appeared, go to "Libraries" tab, select "Add library" button, and then select "JUnit", as shown in the following images below:

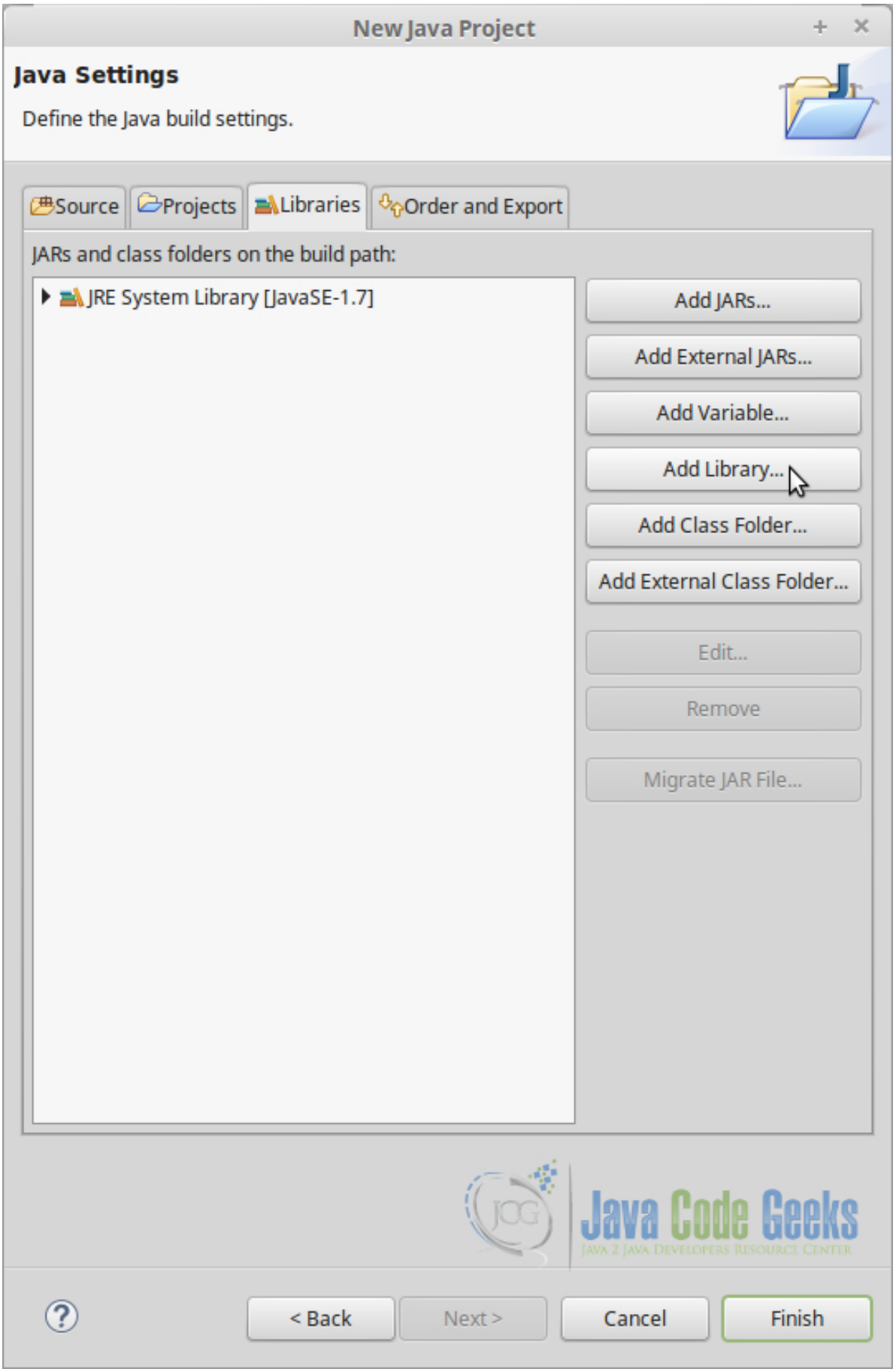

Figure 1.1: Java Settings

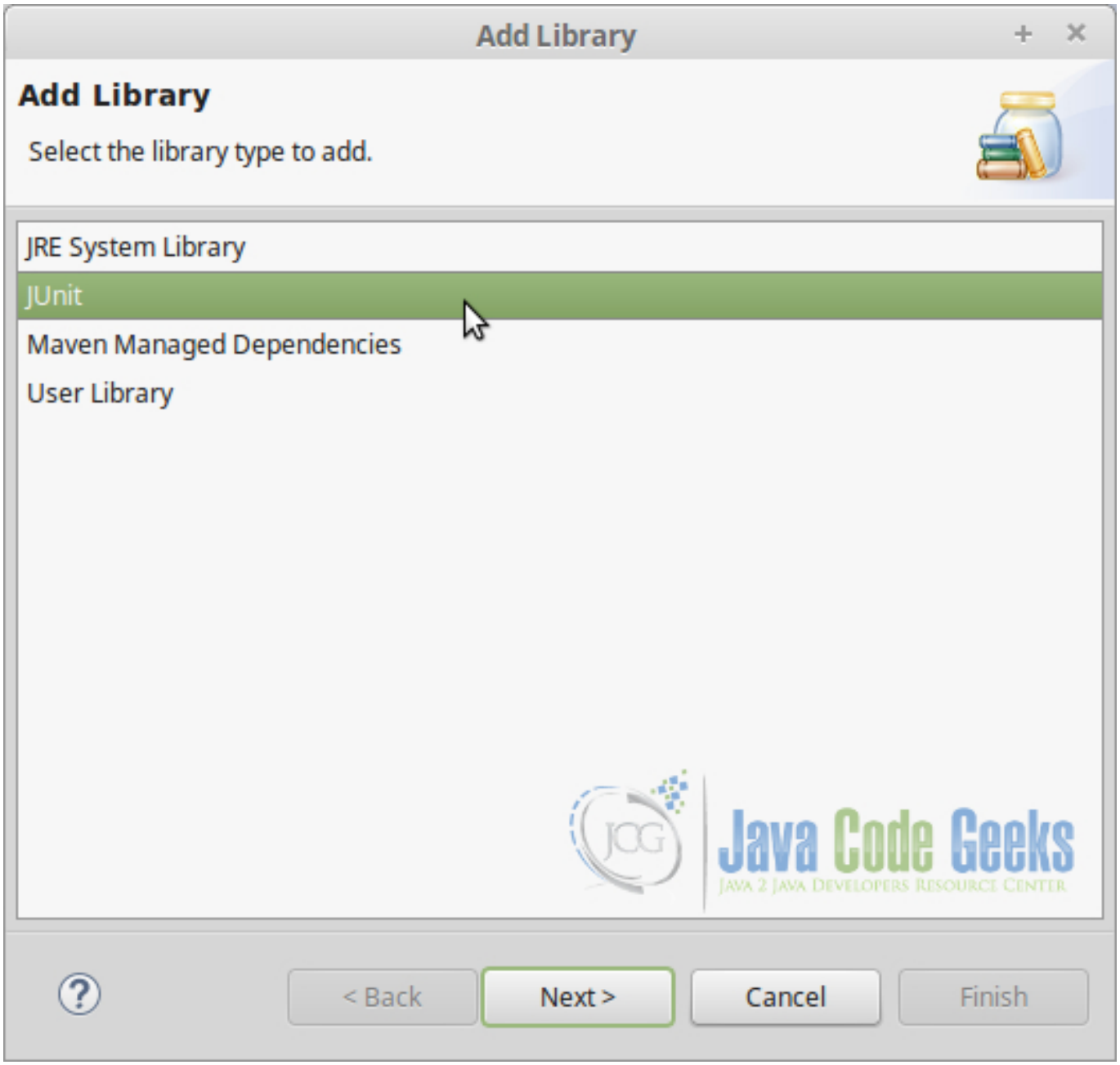

Figure 1.2: Add Library

You can now finish the project creation.

1.3 Mockito installation

1.3.1 Download the JAR

- Download Mockito JAR file from Maven Repository.
- Place it inside your working directory, for example, in a lib directory in the directory root.
- Refresh the Package Explorer in Eclipse (F5).

- Now, a new lib directory should be displayed, with the Mockito JAR file inside it. Right click on it an select "Build Path/Add to Build Path" (shown in image below).

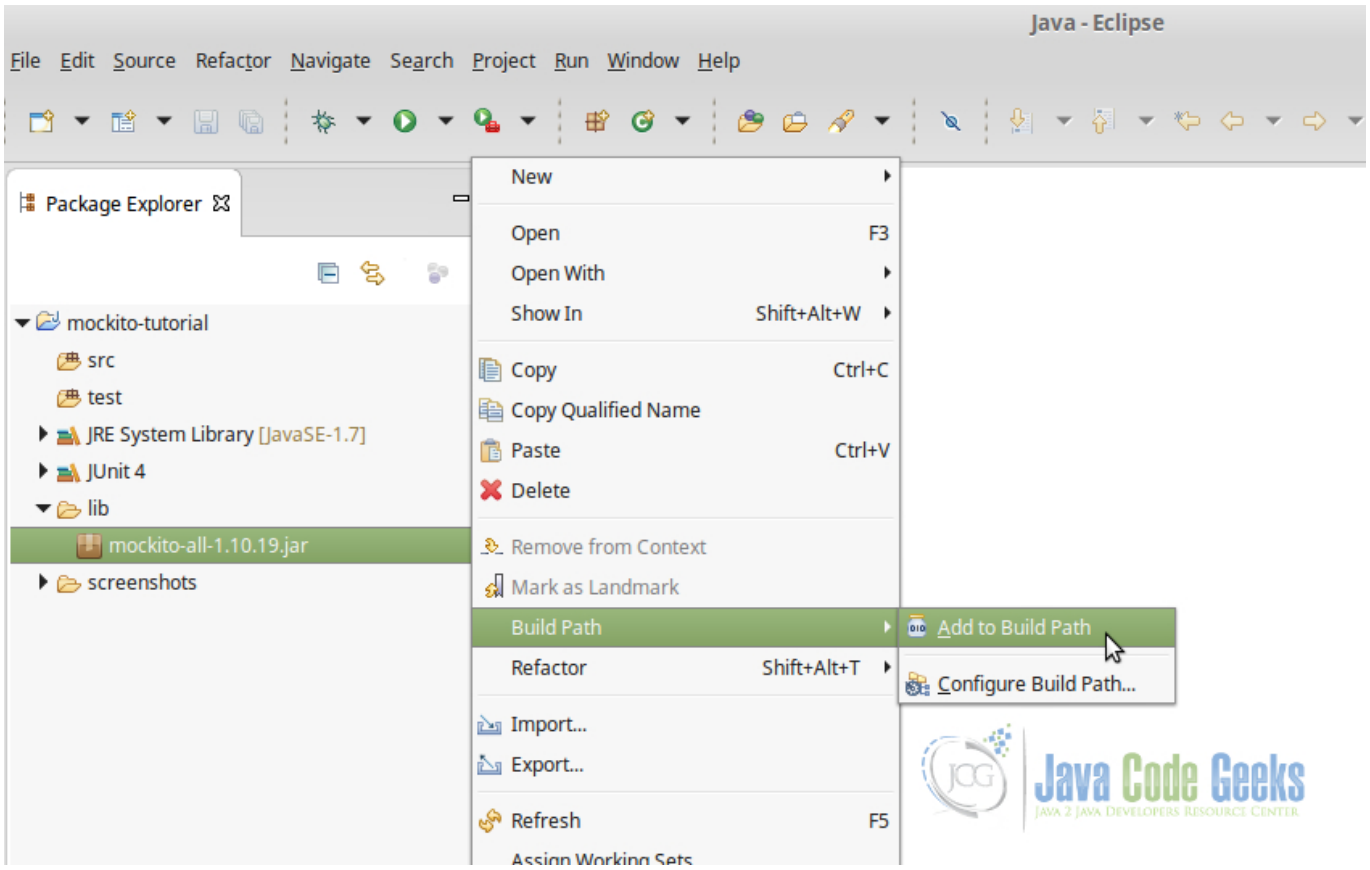

Figure 1.3: Add to Classpath

1.3.2 With build tools

1.3.2.1 Maven

Just declare the dependency as it follows:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.10.19</version>
</dependency>
```

1.3.2.2 Gradle

Declare the dependency as it is shown below:

```
repositories {
    jcenter()
}

dependencies {
```

```
    testCompile "org.mockito:mockito-core:1.+"
}
```

1.4 Base code to test

Let's suppose that our application is for authenticating users, and that our job is to develop the interface that the final user will use, and that developing the logic is someone else's job. For mocking, is indispensable to agree the interfaces to mock, that is, the method definitions: name, parameters, and return type. For this case, the agreed interface will be a public method `authenticateUser`, that receives two strings, the user name and the password; returning a boolean indicating if the authentication succeeded or not. So, the interface would be the following:

AuthenticatorInterface.java

```
package com.javacodegeeks.mockitotutorial.basecode;

public interface AuthenticatorInterface {

    /**
     * User authentication method definition.
     *
     * @param username The user name to authenticate.
     * @param password The password to authenticate the user.
     * @return True if the user has been authenticated; false if it has not.
     * @throws EmptyCredentialsException If the received credentials (user name, password) ↩
         are
     * empty.
     */
    public boolean authenticateUser(String username, String password);

}
```

And the source that uses this interface:

AuthenticatorApplication.java

```
package com.javacodegeeks.mockitotutorial.basecode;

public class AuthenticatorApplication {

    private AuthenticatorInterface authenticator;

    /**
     * AuthenticatorApplication constructor.
     *
     * @param authenticator Authenticator interface implementation.
     */
    public AuthenticatorApplication(AuthenticatorInterface authenticator) {
        this.authenticator = authenticator;
    }

    /**
     * Tries to authenticate an user with the received user name and password, with the ↩
         received
     * AuthenticatorInterface interface implementation in the constructor.
     *
     * @param username The user name to authenticate.
     * @param password The password to authenticate the user.
     * @return True if the user has been authenticated; false if it has not.
     */
    public boolean authenticate(String username, String password) {
```

```
        boolean authenticated;

        authenticated = this.authenticator.authenticateUser(username, password);

        return authenticated;
    }
}
```

We will suppose that this piece of code also implements the `main` method, but is not important for this example.

Now, we are going to code the tests for `AuthenticatorApplication`. The testing method returns a boolean, so we will code tests for covering both possible cases: failed login, and succeeded one.

As the code that handles the authentication is not developed, we have to make some suppositions. We are not doing any real authentication. We have to define for which values the function will succeed, and for which not.

1.5 Adding behavior

Let's see how we can mock the Authenticator:

AuthenticatorApplicationTest.java

```
package com.javacodegeeks.mockitotutorial.basecode;

import org.junit.Test;
import org.mockito.Mockito;

import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

public class AuthenticatorApplicationTest {

    @Test
    public void testAuthenticate() {
        AuthenticatorInterface authenticatorMock;
        AuthenticatorApplication authenticator;
        String username = "JavaCodeGeeks";
        String password = "unsafePassword";

        authenticatorMock = Mockito.mock(AuthenticatorInterface.class);
        authenticator = new AuthenticatorApplication(authenticatorMock);

        when(authenticatorMock.authenticateUser(username, password))
            .thenReturn(false);

        boolean actual = authenticator.authenticate(username, password);

        assertFalse(actual);
    }
}
```

Let's see carefully what we are doing:

- We import the required stuff, as in lines 4 and 7. The IDE will help us to do it.
- We define the mock object, in line 18. This is how the mock "learns" the method definitions to mock.
- The key part is when we **add the behavior**, as in lines 21 and 22, with the `when()` and `thenReturn()` functions. Is quite expressive: "**When** the mock object is called for this method with this parameters, **then** it returns this value". Note that we are **defining the behavior in the mock object, not to the class calling the mock object**.

As we are adding the behavior to the reference that has been passed to `AuthenticatorApplication` instance, it doesn't matter if we first add the behavior and then we pass the reference, or reverse.

When the `AuthenticatorApplication` calls to its `AuthenticatorInterface`, it won't know what is actually happening, the only thing it knows is just how to deal with the defined interface, which for this case has been designed to return `false` when it receives `"JavaCodeGeeks"` and `"unsafePassword"` as inputs.

1.6 Verifying behavior

Mockito allows to make several verifications about our mock objects. Let's see which are they.

1.6.1 Verify that method has been called

We can check if a method has been called with certain parameters. For that, we would do something similar to the following:

AuthenticatorApplicationTest.java

```
// ...

verify(authenticatorMock).authenticateUser(username, password);

// ...
```

To verify that `authenticatorMock` mock's `authenticateUser` method, with `username` and `password` parameters.

Of course, this verification only makes sense if we make it after the call is supposed to be done.

Apart from checking that the method is actually being called, this verifications are **useful to check that the parameters arrive to the method call as they are supposed to arrive**. So, for example, if you run the test with the following verification:

AuthenticatorApplicationTest.java

```
// ...

verify(authenticatorMock).authenticateUser(username, "not the original password");

// ...
```

The test will fail.

1.6.2 Verify that method has been called n times

Apart from checking that the method has been called or not, we have many possibilities regarding to number of method calls. Let's see how we can do it:

AuthenticatorApplicationTest.java

```
// ...

verify(authenticatorMock, times(1)).authenticateUser(username, password);
verify(authenticatorMock, atLeastOnce()).authenticateUser(username, password);
verify(authenticatorMock, atLeast(1)).authenticateUser(username, password);
verify(authenticatorMock, atMost(1)).authenticateUser(username, password);

// ...
```

As you can see, we have different notations available to make the verifications: specifying the number of times that the mocking method should be called, how much times should be called at least, and how much at most.

As in the previous example, **the verifications are made for the exact parameters that the mocking method uses**.

We can also verify that the method has never been called:

AuthenticatorApplicationTest.java

```
// ...

verify(authenticatorMock, never()).authenticateUser(username, password); // This will make ↩
    the test fail!

// ...
```

Which, actually, is equivalent to `times(0)`, but would be more expressive when we really want to verify that a method has never been called.

1.6.3 Verify method call order

We can also verify in which order have been executed the mock methods.

To see how it works, let's add a dummy method in the interface:

AuthenticatorInterface.java

```
// ...

public void foo();

// ...
```

And also call it from the original `AuthenticatorApplication.authenticate()` method:

AuthenticatorApplication.java

```
// ...

public boolean authenticate(String username, String password) throws ↩
    EmptyCredentialsException{
    boolean authenticated;

    this.authenticator.foo();
    authenticated = this.authenticator.authenticateUser(username, password);

    return authenticated;
}

// ...
```

Now, let's see how we would verify that the `foo()` method is called before `authenticateUser()` method:

AuthenticatorApplicationTest.java

```
// ...

InOrder inOrder = inOrder(authenticatorMock);
inOrder.verify(authenticatorMock).foo();
inOrder.verify(authenticatorMock).authenticateUser(username, password);

// ...
```

We just have to create an `InOrder` instance for the mock object to make the verification, and then call its `verify()` method in the same order we want to make the verification. So, the following snippet, for the current `AuthenticatorApplication.authenticate()` method, will make the test fail:

AuthenticatorApplicationTest.java

```
// ...

InOrder inOrder = inOrder(authenticatorMock);
inOrder.verify(authenticatorMock).authenticateUser(username, password); // This will make ↩
    the test fail!
inOrder.verify(authenticatorMock).foo();

// ...
```

Because in the method the mocking object is used, `authenticateUser()` is called after `foo()`.

1.6.4 Verification with timeout

Mockito verification also allows to specify a timeout for the mock methods execution. So, if we want to ensure that our `authenticateUser()` method runs in, for example, 100 milliseconds or less, we would do the following:

AuthenticatorApplicationTest.java

```
// ...

verify(authenticatorMock, timeout(100)).authenticateUser(username, password);

// ...
```

The timeout verification can be combined with the method call, so, we could verify the timeout for n method calls:

AuthenticatorApplicationTest.java

```
// ...

verify(authenticatorMock, timeout(100).times(1)).authenticateUser(username, password);

// ...
```

And any other method call verifier.

1.7 Throwing exceptions

Mockito allows its mocks to throw exceptions. Is possible to make a mock method throw an exception that is not defined in the method signature, but is better to agree in a common method definition from the beginning, including exception throwing.

We could create an exception class to be thrown when, for example, empty credentials are provided:

EmptyCredentialsException.java

```
package com.javacodegeeks.mockitotutorial.basecode;

public class EmptyCredentialsException extends Exception {

    public EmptyCredentialsException() {
        super("Empty credentials!");
    }
}
```

We add it to the method signature of our `AuthenticatorInterface`, and also to its call in `AuthenticatorApplication`:

AuthenticatorInterface.java

```
package com.javacodegeeks.mockitotutorial.basecode;

public interface AuthenticatorInterface {

    /**
     * User authentication method definition.
     *
     * @param username The user name to authenticate.
     * @param password The password to authenticate the user.
     * @return True if the user has been authenticated; false if it has not.
     * @throws EmptyCredentialsException If the received credentials (user name, password) ↩
         are
     * empty.
     */
    public boolean authenticateUser(String username, String password) throws ↩
        EmptyCredentialsException;

}
```

For the test, we will create another test case for expecting the exception:

AuthenticatorApplicationTest.java

```
// ...

@Test (expected = EmptyCredentialsException.class)
public void testAuthenticateEmptyCredentialsException() throws EmptyCredentialsException {
    AuthenticatorInterface authenticatorMock;
    AuthenticatorApplication authenticator;

    authenticatorMock = Mockito.mock(AuthenticatorInterface.class);
    authenticator = new AuthenticatorApplication(authenticatorMock);

    when(authenticatorMock.authenticateUser("", ""))
        .thenThrow(new EmptyCredentialsException());

    authenticator.authenticate("", "");
}
```

As you can see, is almost identical to adding return values to the mock. The only difference is that we have to call `thenThrow()`, passing the exception instance we want to be thrown. And, of course, we have to handle the exception; in this case, we have used the `expected` rule to "assert" the exception.

1.8 Shorthand mock creation

For a few mocks, creating every mock object is not a problem. But, when there is a considerable number of them, it can be quite tedious to create every mock.

Mockito provides a shorthand notation, which is really expressive, to **inject the mock dependencies**.

If we want to inject dependencies with Mockito, we have to take the two things into account:

- Only works for class scope, not for function scope.
- We must run the test class with `MockitoJUnitRunner.class`.

So, we would have to do the following:

AuthenticatorApplicationTest.java

```
// ...

@RunWith(MockitoJUnitRunner.class)
public class AuthenticatorApplicationTest {

    @Mock
    private AuthenticatorInterface authenticatorMock;

    @InjectMocks
    private AuthenticatorApplication authenticator;

    // ...
}
```

With the `@Mock` annotation, we define the dependencies to inject. And then, with `@InjectMocks`, we specify where to inject the defined dependencies. With only those annotations, we have an instance of `AuthenticatorApplication` with the `AuthenticatorInterface` injected.

To perform the injection, Mockito tries the following ways, in order:

- By constructor (as we have).
- By setter.
- By class field.

If Mockito is unable to do the injection, the result will be a null reference to the object to be injected, which in this case, would be `AuthenticatorApplication`.

But, as we have a constructor where the interface is passed, Mockito is supposed to do the injection properly. So now, we could make another test case to test it:

AuthenticatorApplicationTest.java

```
@Test
public void testAuthenticateMockInjection() throws EmptyCredentialsException {
    String username = "javacodegeeks";
    String password = "s4f3 p4ssw0rd";

    when(this.authenticatorMock.authenticateUser(username, password))
        .thenReturn(true);

    boolean actual = this.authenticator.authenticate("javacodegeeks", "s4f3 p4ssw0rd");

    assertTrue(actual);
}
```

We don't have to do anything more than the test itself, Mockito has created an instance for the `AuthenticatorApplication` with the injected mock.

1.9 Mocking void returning methods

In the previous examples, we have used `when()` for adding behavior to the mocks. But this way won't work for methods that return `void`. If we try to use `when()` with a void method, the IDE will mark an error, and it won't let us compile the code.

First, we are going to change the previous example to make `AuthenticatorInterface` method return `void`, and make it throw an exception if the user has not been successfully authenticated, to give sense to the `void` return. We are going to create another package `com.javacodegeeks.mockitotutorial.voidmethod`, not to modify the previous working code.

AuthenticatorInterface.java

```
package com.javacodegeeks.mockitotutorial.voidmethod;

public interface AuthenticatorInterface {

    /**
     * User authentication method definition.
     *
     * @param username The user name to authenticate.
     * @param password The password to authenticate the user.
     * @throws NotAuthenticatedException If the user can't be authenticated.
     */
    public void authenticateUser(String username, String password) throws  ↩
        NotAuthenticatedException;

}
```

And also, its call:

AuthenticatorApplication.java

```
package com.javacodegeeks.mockitotutorial.voidmethod;

public class AuthenticatorApplication {

    private AuthenticatorInterface authenticator;

    /**
     * AuthenticatorApplication constructor.
     *
     * @param authenticator Authenticator interface implementation.
     */
    public AuthenticatorApplication(AuthenticatorInterface authenticator) {
        this.authenticator = authenticator;
    }

    /**
     * Tries to authenticate an user with the received user name and password, with the  ↩
         received
     * AuthenticatorInterface interface implementation in the constructor.
     *
     * @param username The user name to authenticate.
     * @param password The password to authenticate the user.
     * @throws NotAuthenticatedException If the user can't be authenticated.
     */
    public void authenticate(String username, String password) throws  ↩
        NotAuthenticatedException {
        this.authenticator.authenticateUser(username, password);
    }
}
```

The required exception class also:

NotAuthenticatedException.java

```
package com.javacodegeeks.mockitotutorial.voidmethod;

public class NotAuthenticatedException extends Exception {

    public NotAuthenticatedException() {
        super("Could not authenticate!");
    }
}
```

Now, to mock `AuthenticatorInterface.authenticateUser`, we have to use the `do` family methods:

AuthenticatorApplicationTest.java

```
package com.javacodegeeks.mockitotutorial.voidmethod;

import static org.mockito.Mockito.doThrow;

import org.junit.Test;
import org.mockito.Mockito;

public class AuthenticatorApplicationTest {

    @Test(expected = NotAuthenticatedException.class)
    public void testAuthenticate() throws NotAuthenticatedException {
        AuthenticatorInterface authenticatorMock;
        AuthenticatorApplication authenticator;
        String username = "JavaCodeGeeks";
        String password = "wrong password";

        authenticatorMock = Mockito.mock(AuthenticatorInterface.class);
        authenticator = new AuthenticatorApplication(authenticatorMock);

        doThrow(new NotAuthenticatedException())
            .when(authenticatorMock)
            .authenticateUser(username, password);

        authenticator.authenticate(username, password);
    }
}
```

We are doing the same thing as in the previous example, but using a different notation (lines 20, 21, 22). We could say that it's almost the same syntax, but inverted: first, we add the behavior (a `throw` behavior); and then, we specify the method we are adding the behavior to.

1.10 Mocking real objects: @Spy

Exists the possibility of creating mocks that wrap objects, i.e., instances of implemented classes. This is called "spying" by Mockito.

When you call the method of a spied object, the real method will be called, unless a predefined behavior was defined.

Let's create a new test case in a new package to see how it works:

SpyExampleTest.java

```
package com.javacodegeeks.mockitotutorial.spy;

import static org.mockito.Mockito.*;

import java.util.HashMap;
import java.util.Map;

import org.junit.Test;

public class SpyExampleTest {

    @Test
    public void spyExampleTest() {
        Map<String, String> hashMap = new HashMap<String, String>();
        Map<String, String> hashMapSpy = spy(hashMap);
```

```
        System.out.println(hashMapSpy.get("key")); // Will print null.

        hashMapSpy.put("key", "A value");
        System.out.println(hashMapSpy.get("key")); // Will print "A value".

        when(hashMapSpy.get("key")).thenReturn("Another value");
        System.out.println(hashMapSpy.get("key")); // Will print "Another value".
    }
}
```

As you can see, we can both delegate the method call to the real implementation, or define a behavior.

You might think that this is a quite odd feature. And you'll probably right. In fact, **Mockito documentation recommends to use this only occasionally**.

1.11 Summary

This tutorial has explained what mocking is, and how to put in practice this technique in Java with Mockito framework. We have seen how to add predefined behaviors to our mock objects, and several ways of verifying that those mock objects behave as they are supposed to do. We also have seen the possibility of mocking real objects, a feature that should be used carefully.

1.12 Download the Eclipse Project

This was a tutorial of Mockito.

Download

You can download the full source code of this example here: **MockitoTutorialForBeginners**

Chapter 2

Test-Driven Development With Mockito

In this example we will learn how to do Test Driven Development (TDD) using Mockito. A unit test should test a class in isolation. Side effects from other classes or the system should be eliminated if possible. Mockito lets you write beautiful tests with a clean & simple API. Tools and technologies used in this example are Java 1.8, Eclipse Luna 4.4.2

2.1 Introduction

Mockito is a popular mocking framework which can be used in conjunction with JUnit. Mockito allows us to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly. We can create the mock objects manually or we can use the mocking framewors like Mockito, EasyMock. jMock etc. Mock frameworks allow us to create mock objects at runtime and define their behavior. The classical example for a mock object is a data provider. In production a real database is used, but for testing a mock object simulates the database and ensures that the test conditions are always the same.

2.2 Test Driven Development

Test-Driven Development (TDD) is an evolutionary approach to development. It offers test-first development where the production code is written only to satisfy a test. TDD is the new way of programming. Here the rule is very simple; it is as follows:

- Write a test to add a new capability (automate tests).
- Write code only to satisfy tests.
- Re-run the tests-if any test is broken, revert the change.
- Refactor and make sure all tests are green.
- Continue with step 1.

2.3 Creating a project

Below are the steps required to create the project.

- Open Eclipse. Go to File⇒New⇒Java Project. In the ‘Project name’ enter ‘TDDMockito’.

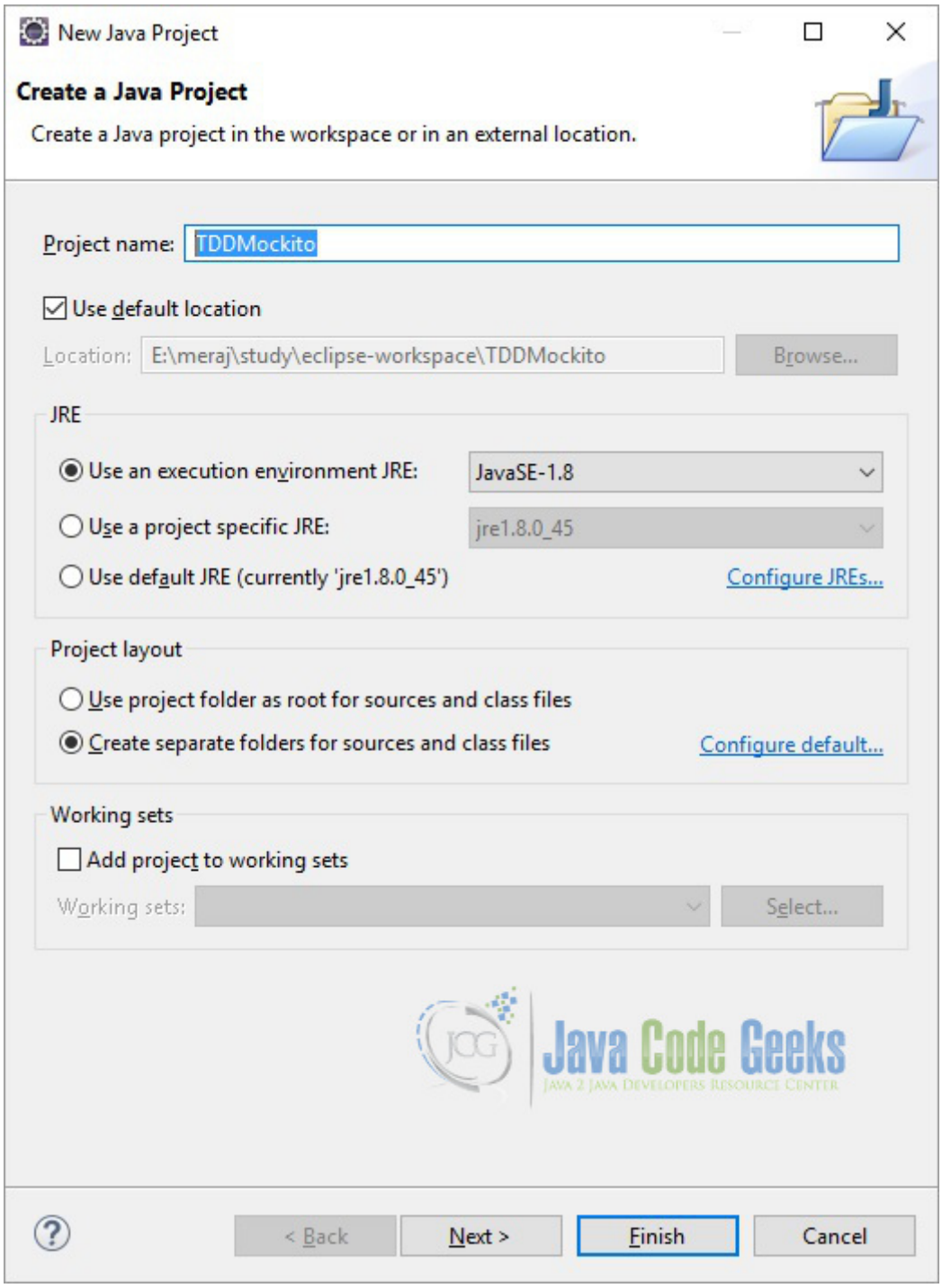

Figure 2.1: Create Java Project

- Eclipse will create a 'src' folder. Right click on the 'src' folder and choose New⇒Package. In the 'Name' text-box enter 'com.javacodegeeks'. Click 'Finish'.

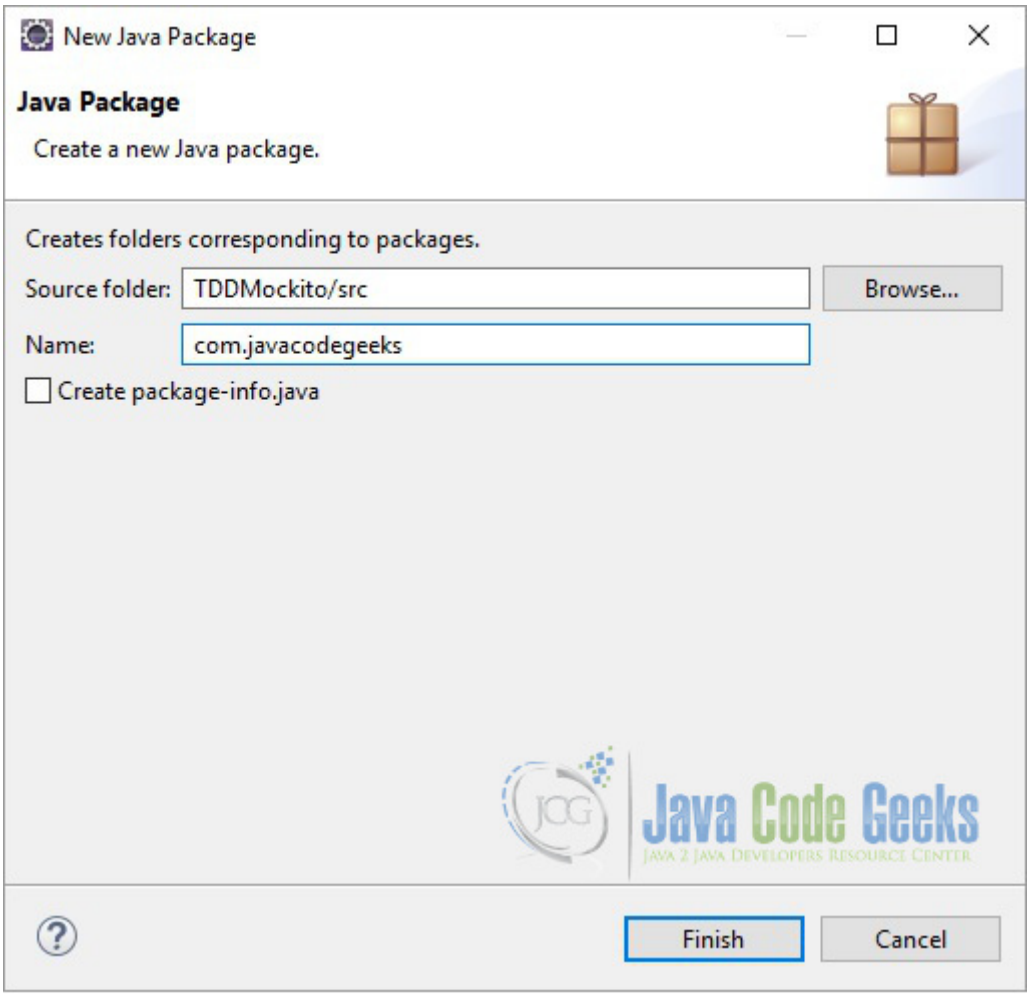

Figure 2.2: New Java Package

- Right click on the package and choose New⇒Class. Give the class name and click 'Finish'. Eclipse will create a default class with the given name.

2.3.1 Dependencies

For this example we need the junit and mockito jars. These jars can be downloaded from Maven repository. We are using 'junit-4.12.jar' and 'mockito-all-1.10.19.jar'. There are the latests (non-beta) versions available as per now. To add these jars in the classpath right click on the project and choose Build Path⇒Configure Build Path. The click on the 'Add External JARs' button on the right hand side. Then go to the location where you have downloaded these jars. Then click ok.

2.4 Test first

Let's say we want to build a tool for Report generation. Please note that this is a very simple example of showing how to use mockito for TDD. It does not focus on developing a full report generation tool.

For this we will need three classes. The first one is the interface which will define the API to generate the report. The second one is the report entity itself and the third one is the service class. First we will start with writing the test.

We will inject the service class by using @InjectMocks.

```
@InjectMocks private ReportGeneratorService reportGeneratorService;
```

@InjectMocks mark a field on which injection should be performed. It allows shorthand mock and spy injection. Mockito will try to inject mocks only either by constructor injection, setter injection, or property injection in order and as described below. If any of the following strategy fail, then Mockito won't report failure i.e. you will have to provide dependencies yourself.

Constructor injection: the biggest constructor is chosen, then arguments are resolved with mocks declared in the test only. If the object is successfully created with the constructor, then Mockito won't try the other strategies. Mockito has decided not to corrupt an object if it has a parametered constructor. If arguments can not be found, then null is passed. If non-mockable types are wanted, then constructor injection won't happen. In these cases, you will have to satisfy dependencies yourself.

Property setter injection: mocks will first be resolved by type (if a single type matches injection will happen regardless of the name), then, if there are several property of the same type, by the match of the property name and the mock name. If you have properties with the same type (or same erasure), it's better to name all @Mock annotated fields with the matching properties, otherwise Mockito might get confused and injection won't happen. If @InjectMocks instance wasn't initialized before and have a no-arg constructor, then it will be initialized with this constructor.

Field injection: mocks will first be resolved by type (if a single type matches injection will happen regardless of the name), then, if there is several property of the same type, by the match of the field name and the mock name. If you have fields with the same type (or same erasure), it's better to name all @Mock annotated fields with the matching fields, otherwise Mockito might get confused and injection won't happen. If @InjectMocks instance wasn't initialized before and have a no-arg constructor, then it will be initialized with this constructor.

Now we will mock the interface using @Mock annotation:

```
@Mock private IReportGenerator reportGenerator;
```

Now we will define the argument captor on report entity:

```
@Captor private ArgumentCaptor<ReportEntity> reportCaptor;
```

The ArgumentCaptor class is used to capture argument values for further assertions. Mockito verifies argument values in natural java style: by using an equals() method. This is also the recommended way of matching arguments because it makes tests clean & simple. In some situations though, it is helpful to assert on certain arguments after the actual verification.

Now we will define a setup method which we will annotate with @Before. This we will use to initialize the mocks.

```
MockitoAnnotations.initMocks(this);
```

initMocks() initializes objects annotated with Mockito annotations for given test class.

In the test method we will call the generateReport() method of the ReportGeneratorService class passing the required parameters:

```
reportGeneratorService.generateReport(startDate.getTime(), endDate.getTime(), reportContent ↩
    .getBytes());
```

Below is the snippet of the whole test class:

ReportGeneratorServiceTest.java

```
package com.javacodegeeks;

import static org.junit.Assert.assertEquals;

import java.util.Calendar;

import org.junit.Before;
import org.junit.Test;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.MockitoAnnotations;

public class ReportGeneratorServiceTest {
```

```
  @InjectMocks private ReportGeneratorService reportGeneratorService;
  @Mock private IReportGenerator reportGenerator;
  @Captor private ArgumentCaptor<ReportEntity> reportCaptor;

  @Before
  public void setUp() {
    MockitoAnnotations.initMocks(this);
  }

  @SuppressWarnings("deprecation")
  @Test
  public void test() {
    Calendar startDate = Calendar.getInstance();
    startDate.set(2016, 11, 25);
    Calendar endDate = Calendar.getInstance();
    endDate.set(9999, 12, 31);
    String reportContent = "Report Content";
    reportGeneratorService.generateReport(startDate.getTime(), endDate.getTime(),  ↩
        reportContent.getBytes());

    Mockito.verify(reportGenerator).generateReport(reportCaptor.capture());

    ReportEntity report = reportCaptor.getValue();

    assertEquals(116, report.getStartDate().getYear());
    assertEquals(11, report.getStartDate().getMonth());
    assertEquals(25, report.getStartDate().getDate());

    assertEquals(8100, report.getEndDate().getYear());
    assertEquals(0, report.getEndDate().getMonth());
    assertEquals(31, report.getEndDate().getDate());

    assertEquals("Report Content", new String(report.getContent()));
  }

}
```

The test class will not compile as the required classes are missing here. Don't worry as this is how TDD works. First we write the test then we build our classes to satisfy the test requirements.

Now lets start adding the classes. First we will add the interface. This is the same interface which we mocked in our test class. The service class will have reference to this interface.

IReportGenerator.java

```
package com.javacodegeeks;

/**
* Interface for generating reports.
* @author Meraj
*/
public interface IReportGenerator {

  /**
  * Generate report.
  * @param report Report entity.
  */
  void generateReport(ReportEntity report);

}
```

Please note that this interface will also not compile as the ReportEntity class is still missing. Now lets add the entity class. This class represents the domain object in our design.

ReportEntity.java

```
package com.javacodegeeks;

import java.util.Date;

/**
* Report entity.
* @author Meraj
*/
public class ReportEntity {

  private Long reportId;
  private Date startDate;
  private Date endDate;
  private byte[] content;

  public Long getReportId() {
    return reportId;
  }

  public void setReportId(Long reportId) {
    this.reportId = reportId;
  }

  public Date getStartDate() {
    return startDate;
  }

  public void setStartDate(Date startDate) {
    this.startDate = startDate;
  }

  public Date getEndDate() {
    return endDate;
  }

  public void setEndDate(Date endDate) {
    this.endDate = endDate;
  }

  public byte[] getContent() {
    return content;
  }

  public void setContent(byte[] content) {
    this.content = content;
  }
}
```

Now lets add the service class:

ReportGeneratorService.java

```
package com.javacodegeeks;

import java.util.Date;

/**
* Service class for generating report.
```

```
* @author Meraj
*/
public class ReportGeneratorService {

  private IReportGenerator reportGenerator;

  /**
  * Generate report.
  * @param startDate start date
  * @param endDate end date
  * @param content report content
  */
  public void generateReport(Date startDate, Date endDate, byte[] content) {
    ReportEntity report = new ReportEntity();
    report.setContent(content);
    report.setStartDate(startDate);
    report.setEndDate(endDate);
    reportGenerator.generateReport(report);
  }

}
```

Now all the classes will compile and we can run our test class.

2.5 Download the source file

This was an example of using Mockito to do Test Driven Development.

Download

You can download the full source code of this example here: **TDD Mockito**

Chapter 3

Mockito Initmocks Example

In this example we will learn how to initialize mocks in Mockito. A unit test should test a class in isolation. Side effects from other classes or the system should be eliminated if possible. Mockito lets you write beautiful tests with a clean & simple API. Tools and technologies used in this example are Java 1.8, Eclipse Luna 4.4.2

3.1 Introduction

Mockito is a popular mocking framework which can be used in conjunction with JUnit. Mockito allows us to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly. We can create the mock objects manually or can use the mocking frameworks like Mockito, EasyMock, jMock etc. Mock frameworks allow us to create mock objects at runtime and define their behavior. The classical example for a mock object is a data provider. In production a real database is used, but for testing a mock object simulates the database and ensures that the test conditions are always the same.

3.2 Creating a project

Below are the steps we need to take to create the project.

- Open Eclipse. Go to File⇒New⇒Java Project. In the ‘Project name’ enter ‘MockitoInitmocks’.

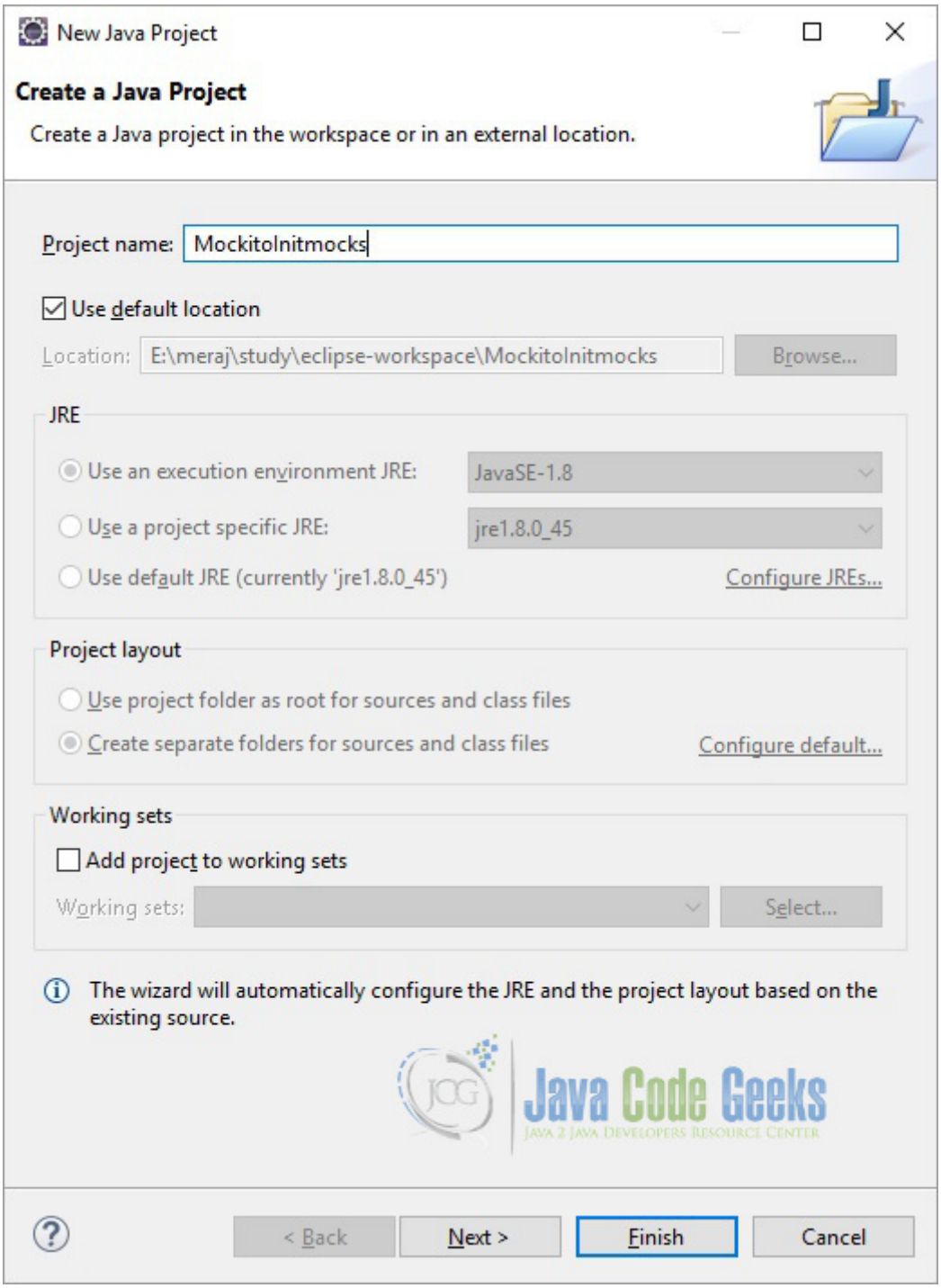

Figure 3.1: Create a Java Project

- Eclipse will create a 'src' folder. Right click on the 'src' folder and choose New⇒Package. In the 'Name' text-box enter 'com.javacodegeeks'. Click 'Finish'.

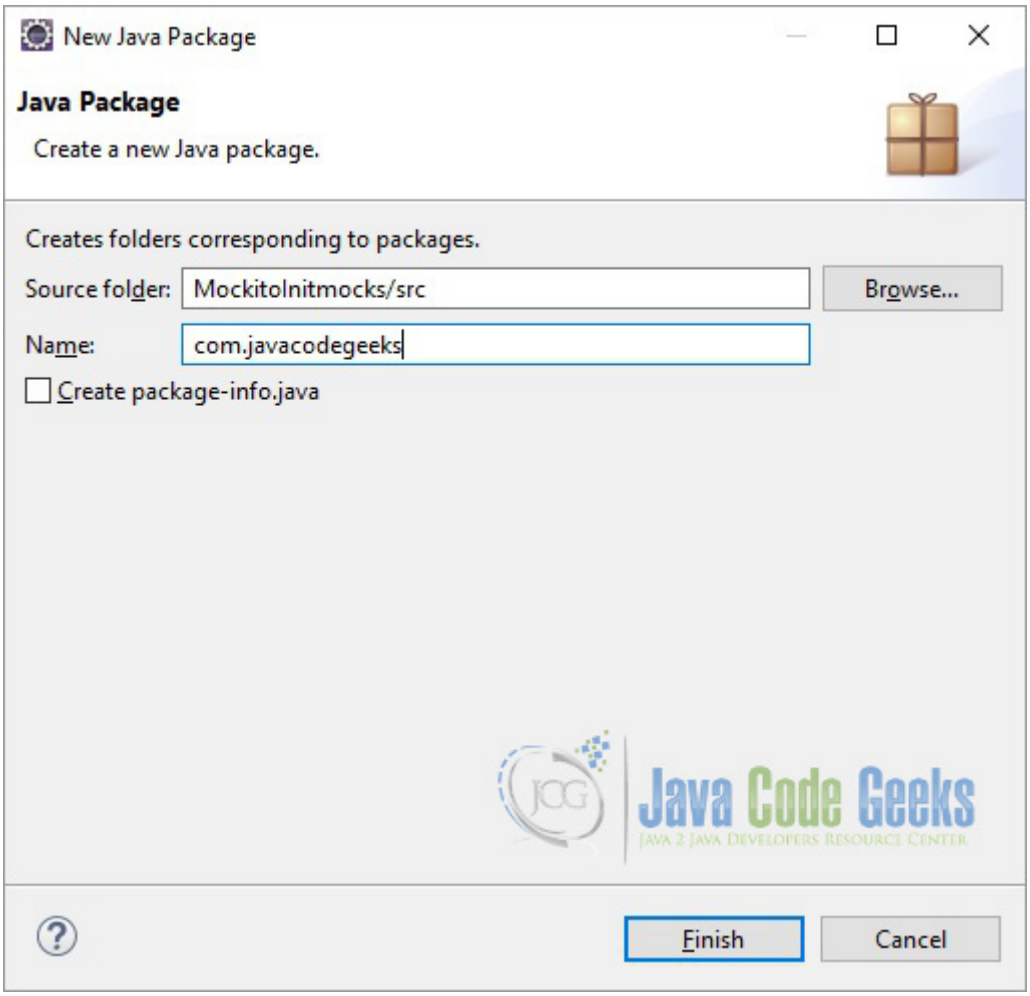

Figure 3.2: New Java Package

3.2.1 Dependencies

For this example we need the junit and mockito jars. These jars can be downloaded from Maven repository. We are using 'junit-4.12.jar' and 'mockito-all-1.10.19.jar'. There are the latests (non-beta) versions available as per now. To add these jars in the classpath right click on the project and choose Build Path⇒Configure Build Path. The click on the 'Add External JARs' button on the right hand side. Then go to the location where you have downloaded these jars. Then click ok.

3.3 Init Mocks

There are various ways how we can initialize the mocks.

3.3.1 Using Mockito.mock()

The first option is to use `mock()` method of `org.mockito.Mockito` class. For this example we will mock the `java.util.LinkedList` class.

```
LinkedList mocklinkedList = Mockito.mock(LinkedList.class);
```

The `mock()` method is used to creates mock object of given class or interface. By default, for all methods that return a value, a mock will return either null, a primitive/primitive wrapper value, or an empty collection, as appropriate. For example 0 for an int/Integer and false for a boolean/Boolean. Now we will define the expectation of the `get()` method as below:

```
Mockito.when(mocklinkedList.get(0)).thenReturn("First Value");
```

`when()` enables stubbing methods. Use it when you want the mock to return particular value when particular method is called. `when()` is a successor of deprecated `Mockito.stub(Object)`. Stubbing can be overridden: for example common stubbing can go to fixture setup but the test methods can override it. Please note that overridding stubbing is a potential code smell that points out too much stubbing.

Once stubbed, the method will always return stubbed value regardless of how many times it is called. Last stubbing is more important - when you stubbed the same method with the same arguments many times. Although it is possible to verify a stubbed invocation, usually it's just redundant. Now we will do the verification as below:

```
Assert.assertEquals("First Value", mocklinkedList.get(0));
Mockito.verify(mocklinkedList).get(0);
```

Below is the snippet of whole test method

```
@Test
public void testMock() {
  // Mock
  LinkedList mocklinkedList = Mockito.mock(LinkedList.class);
  // Stub
  Mockito.when(mocklinkedList.get(0)).thenReturn("First Value");
  // Verify
  Assert.assertEquals("First Value", mocklinkedList.get(0));
  Mockito.verify(mocklinkedList).get(0);
}
```

3.3.2 MockitoAnnotations initMocks()

We can initialize mock by calling `initMocks()` method of `org.mockito.MockitoAnnotations`

```
MockitoAnnotations.initMocks(this);
```

This initializes objects annotated with Mockito annotations for given testClass. This method is useful when you have a lot of mocks to inject. It minimizes repetitive mock creation code, makes the test class more readable and makes the verification error easier to read because the field name is used to identify the mock.

```
@Test
public void testFindById() {
  MockitoAnnotations.initMocks(this);
  MyService myService = new MyService(myDao);
  myService.findById(1L);
  Mockito.verify(myDao);
}
```

`initMocks()` is generally called in `@Before` (JUnit4) method of test's base class. For JUnit3 `initMocks()` can go to `setup()` method of a base class. You can also put `initMocks()` in your JUnit runner `(@RunWith)` or use built-in runner:

3.3.2.1 Inject Mocks

Mark a field on which injection should be performed. It allows shorthand mock and spy injection and minimizes repetitive mock and spy injection. Mockito will try to inject mocks only either by constructor injection, setter injection, or property injection in order and as described below. If any of the following strategy fail, then Mockito won't report failure; i.e. you will have to provide dependencies yourself.

- **Constructor injection:** The biggest constructor is chosen, then arguments are resolved with mocks declared in the test only. If the object is successfully created with the constructor, then Mockito won't try the other strategies. Mockito has decided to no corrupt an object if it has a parametered constructor. Note: If arguments can not be found, then null is passed. If non-mockable types are wanted, then constructor injection won't happen. In these cases, you will have to satisfy dependencies yourself.
- **Property setter injection:** Mocks will first be resolved by type (if a single type match injection will happen regardless of the name), then, if there is several property of the same type, by the match of the property name and the mock name. Note: If you have properties with the same type (or same erasure), it's better to name all `@Mock` annotated fields with the matching properties, otherwise Mockito might get confused and injection won't happen. If `@InjectMocks` instance wasn't initialized before and have a no-arg constructor, then it will be initialized with this constructor.
- **Field injection:** Mocks will first be resolved by type (if a single type match injection will happen regardless of the name), then, if there is several property of the same type, by the match of the field name and the mock name. Note: If you have fields with the same type (or same erasure), it's better to name all `@Mock` annotated fields with the matching fields, otherwise Mockito might get confused and injection won't happen. If `@InjectMocks` instance wasn't initialized before and have a no-arg constructor, then it will be initialized with this constructor.

3.3.3 MockitoJUnitRunner

Another way to initialize mocks is to use `@RunWith(org.mockito.runners.MockitoJUnitRunner.class)` annotation at the test class level. This is compatible with JUNit 4.4 and higher. It initializes mocks annotated with `@Mock`. `MockitoJUnitRunner` so that explicit usage of `MockitoAnnotations.initMocks(Object)` is not necessary. Mocks are initialized before each test method.

It validates framework usage after each test method. Runner is completely optional - there are other ways you can get Mock working, for example by writing a base class. Explicitly validating framework usage is also optional because it is triggered automatically by Mockito every time you use the framework.

MyServiceJUnitRunnerTest.java

```
package com.javacodegeeks;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class MyServiceJUnitRunnerTest {

  private MyService myService;
  @Mock private MyDao myDao;

  @Test
  public void testFindById() {
    myService = new MyService(myDao);
    myService.findById(1L);
    Mockito.verify(myDao).findById(1L);
  }
}
```

3.3.4 MockitoRule

Another way of initializing the mocks is to use the `org.mockito.junit.MockitoRule` class. You first annotate the class reference which needs to be mocked with `@Mock` annotation:

```
@Mock private MyDao myDao;
```

Then you define the rule as below:

```
@Rule public MockitoRule rule = MockitoJUnit.rule();
```

It initializes mocks annotates with `@Mock` so that explicit usage of `org.mockito.MockitoAnnotations#initMocks(Object)` is not necessary. Mocks are initialized before each test method. It validates framework usage after each test method.

MyServiceRuleTest.java

```
package com.javacodegeeks;

import org.junit.Assert;
import org.junit.Rule;
import org.junit.Test;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.MockitoJUnit;
import org.mockito.junit.MockitoRule;

public class MyServiceRuleTest {

  @Mock private MyDao myDao;

  @Rule public MockitoRule rule = MockitoJUnit.rule();

  @Test
  public void test() {
    MyService myService = new MyService(myDao);
    Mockito.when(myDao.findById(1L)).thenReturn(createTestEntity());
    MyEntity actual = myService.findById(1L);
    Assert.assertEquals("My first name", actual.getFirstName());
    Assert.assertEquals("My surname", actual.getSurname());
    Mockito.verify(myDao).findById(1L);
  }

  private MyEntity createTestEntity() {
    MyEntity myEntity = new MyEntity();
    myEntity.setFirstName("My first name");
    myEntity.setSurname("My surname");
    return myEntity;
  }

}
```

3.4 Download the source file

In this example we saw the various methods of initializing mock objects.

Download

You can download the full source code of this example here: **MockitoInitmocks**

Chapter 4

Mockito Maven Dependency Example

A unit test should test a class in isolation. Side effects from other classes or the system should be eliminated if possible. Mockito lets you write beautiful tests with a clean & simple API. In this example we will learn how to define Mockito dependency in maven and how to use it. Tools and technologies used in this example are Java 1.8, Eclipse Luna 4.4.2

4.1 Introduction

Mockito is a popular mocking framework which can be used in conjunction with JUnit. Mockito allows us to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly. We can create the mock objects manually or can use the mocking framewors like Mockito, EasyMock. jMock etc. Mock frameworks allow us to create mock objects at runtime and define their behavior. The classical example for a mock object is a data provider. In production a real database is used, but for testing a mock object simulates the database and ensures that the test conditions are always the same.

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. When creating a project in Eclipse, one may use Maven to manage dependencies more easily and to resolve transitive dependencies automatically

4.2 Creating a project

In this section we will see how Eclipse can help us create a simple maven project. Below are the steps we need to take to create the project.

- Open Eclipse. Go to File⇒New⇒Other. Type *Maven* in the search wizard and choose *Maven Project* under *Maven* folder.

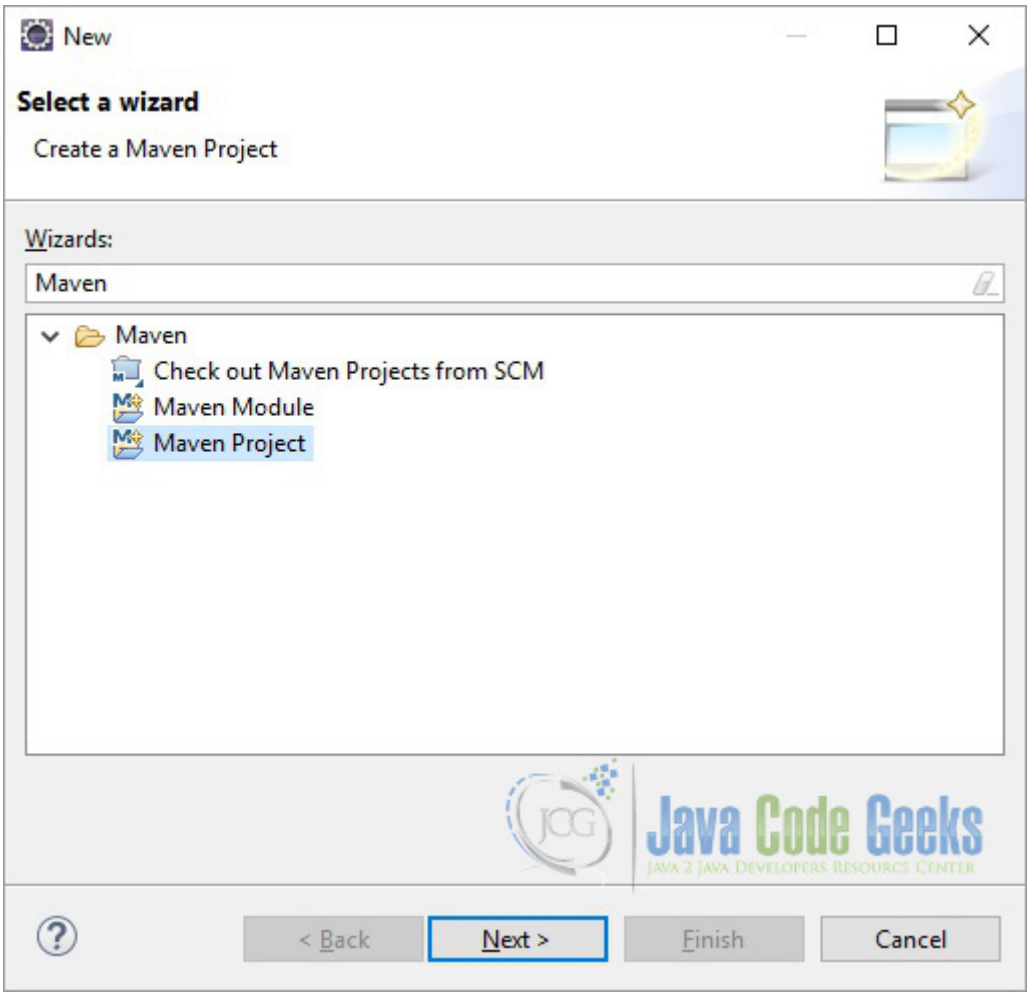

Figure 4.1: Create Maven Project

- Click *Next*. In the next section you need to select the project name and location. Tick the checkbox *Create a simple project (skip archetype selection)*. For the purposes of this tutorial, we will choose the simple project. This will create a basic, Maven-enabled Java project. If you require a more advanced setup, leave this setting unchecked, and you will be able to use more advanced Maven project setup features. Leave other options as is, and click *Next*.

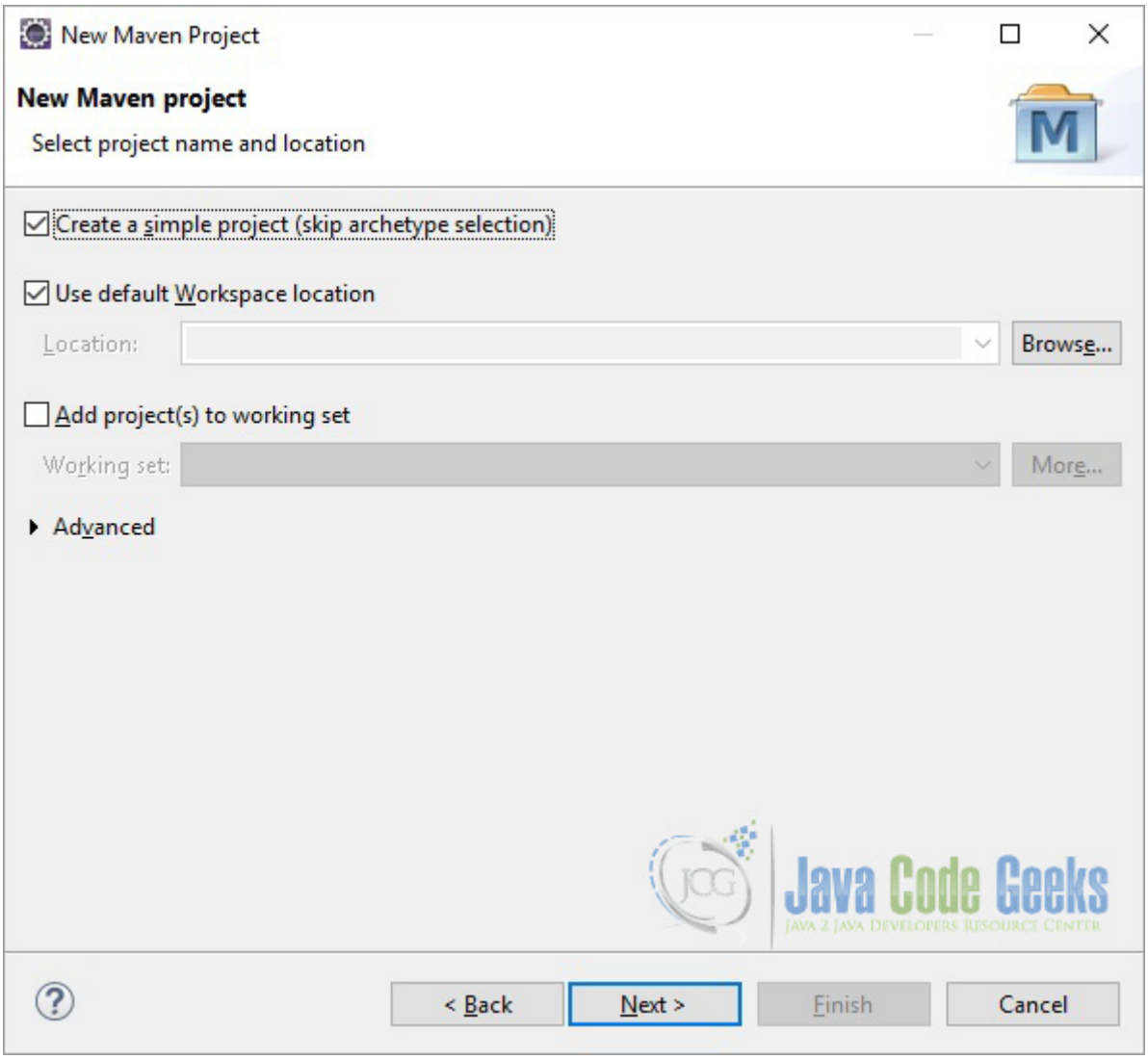

Figure 4.2: New Maven Project

- Now, you will need to enter information regarding the Maven Project you are creating. You may visit the Maven documentation for a more in-depth look at the Maven Coordinates (Maven Coordinates). In general, the **Group Id** should correspond to your organization name, and the **Artifact Id** should correspond to the project's name. The version is up to your discretion as is the packing and other fields. If this is a stand-alone project that does not have parent dependencies, you may leave the **Parent Project** section as is. Fill out the appropriate information, and click **Finish**.

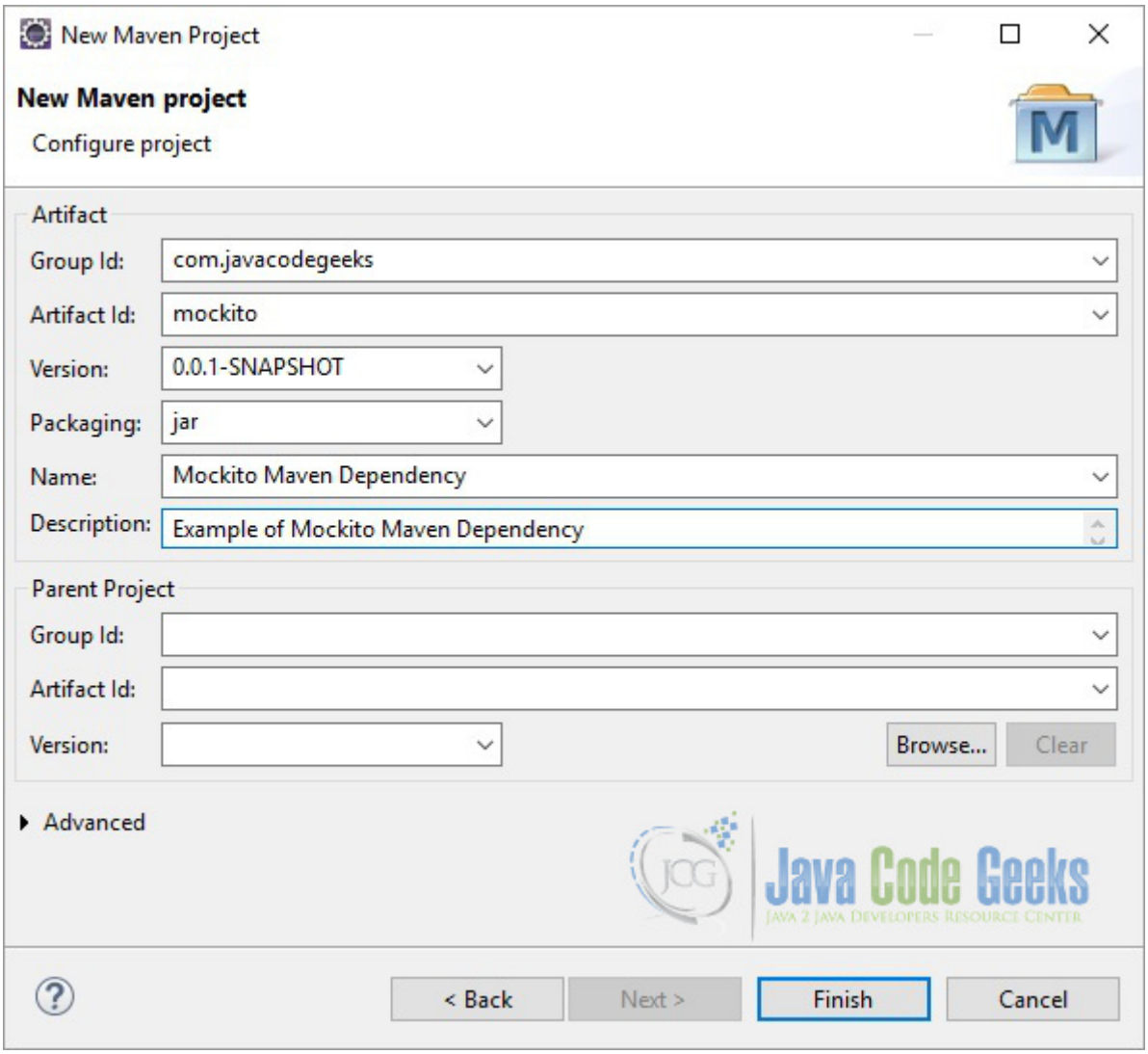

Figure 4.3: Configure Project

- You will now notice that your project has been created. You will place your Java code in **/src/main/java**, resources in **/src/-main/resources**, and your testing code and resources in **/src/test/java** and **/src/test/resources** respectively.

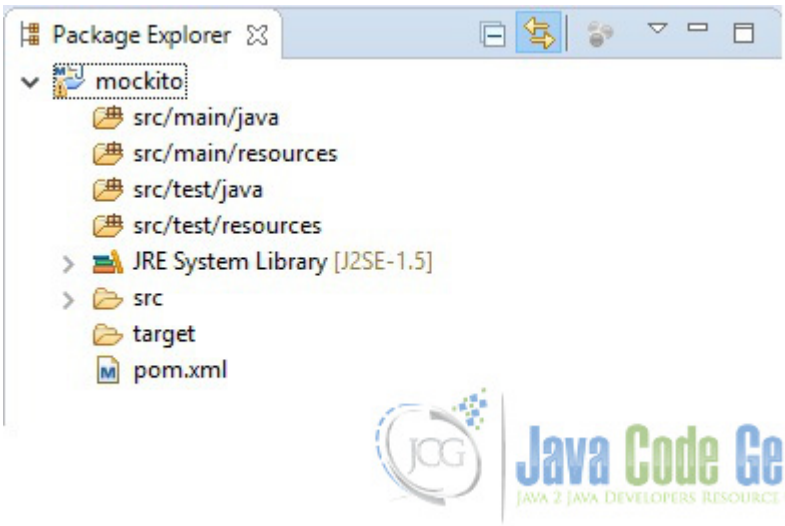

Figure 4.4: Maven Project Structure

Open the **pom.xml** file to view the structure Maven has set up. In this file, you can see the information entered in the steps above. You may also use the tabs at the bottom of the window to change to view **Dependencies**, the **Dependency Hierarchy**, the **Effective POM**, and the raw xml code for the pom file in the **pom.xml** tab.

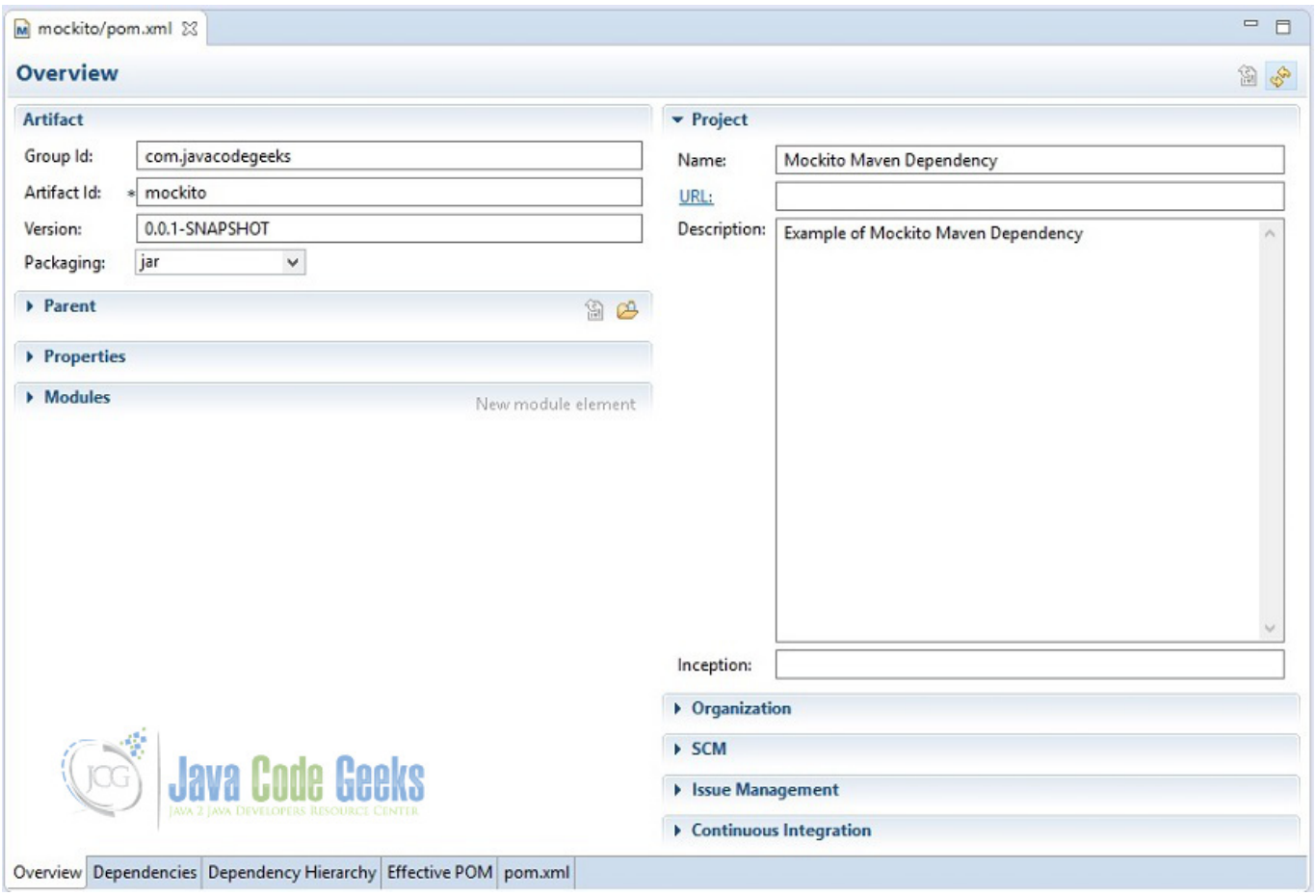

Figure 4.5: POM

4.3 Adding dependencies

Dependencies can be added in two ways. Either directly specifying the dependencies in the pom.xml tab or using *Dependencies* tab to add dependencies. We will use the later.

Open the pom.xml file and click on the *Dependencies* tab. Click on the *Add...* button. Eclipse will open a popup where you can define dependencies. Enter the details as below:

Group Id: org.mockito

Artifact Id: mockito-all

Version: 1.9.5

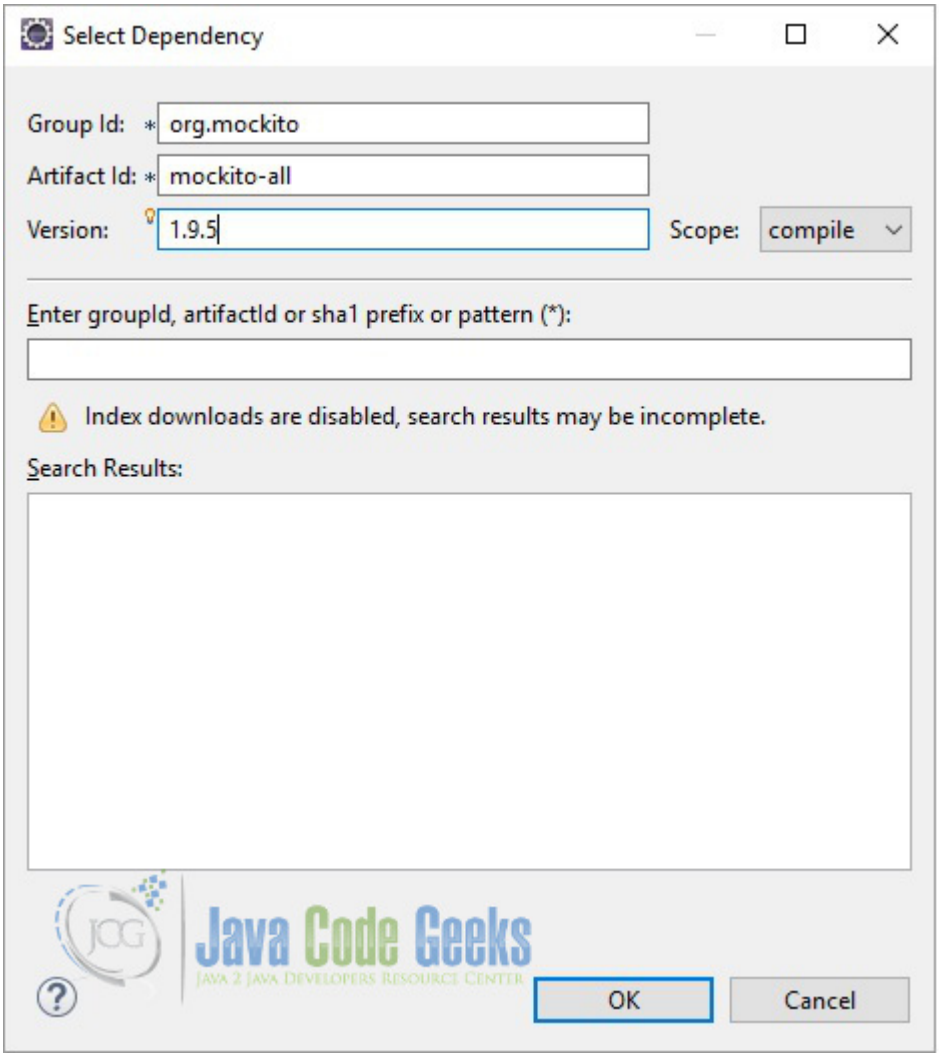

Figure 4.6: Select dependency

Click OK. Check the pom.xml file. Eclipse will add the below section:

```
<dependencies>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.9.5</version>
  </dependency>
</dependencies>
```

Repeat the same steps to add the JUnit dependency

Group Id: junit

Artifact Id: junit

Version: 4.12

Now our final pom will look like below:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance" xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven ↩
    .apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks</groupId>
  <artifactId>mockito</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Mockito Maven Dependency</name>
  <description>Example of Mockito Maven Dependency</description>
  <dependencies>
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-all</artifactId>
      <version>1.9.5</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency>
  </dependencies>
</project>
```

4.4 Testing

Now we will test if our maven project has been set up correctly or not. We will create a simple test class to test this.

MockitoExample.java

```
package mockito;

import java.util.List;

import org.junit.Test;

import static org.mockito.Mockito.*;
import static org.junit.Assert.*;

public class MockitoExample {

  @Test
  public void test() {
    List<String> mockList = mock(List.class);
    mockList.add("First");
    when(mockList.get(0)).thenReturn("Mockito");
    when(mockList.get(1)).thenReturn("JCG");
    assertEquals("Mockito", mockList.get(0));
    assertEquals("JCG", mockList.get(1));
  }
}
```

Run this class as JUnit test and it should run successfully. This will prove that your dependencies are setup correctly.

4.5 Download the source file

In this example we saw how to setup a maven dependency for Mockito using Eclipse

Download

You can download the full source code of this example here: **Mockito Maven Dependency**

Chapter 5

Writing JUnit Test Cases Using Mockito

In this example we will learn how to write JUnit tests using Mockito. A unit test should test a class in isolation. Side effects from other classes or the system should be eliminated if possible. Mockito lets you write beautiful tests with a clean & simple API. Tools and technologies used in this example are Java 1.8, Eclipse Luna 4.4.2

5.1 Introduction

Mockito is a popular mocking framework which can be used in conjunction with JUnit. Mockito allows us to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly. We can create the mock objects manually or can use the mocking framewors like Mockito, EasyMock. jMock etc. Mock frameworks allow us to create mock objects at runtime and define their behavior. The classical example for a mock object is a data provider. In production a real database is used, but for testing a mock object simulates the database and ensures that the test conditions are always the same.

5.2 Creating a project

Below are the steps we need to take to create the project.

- Open Eclipse. Go to File⇒New⇒Java Project. In the ‘Project name’ enter ‘MockitoJUnitExample’.

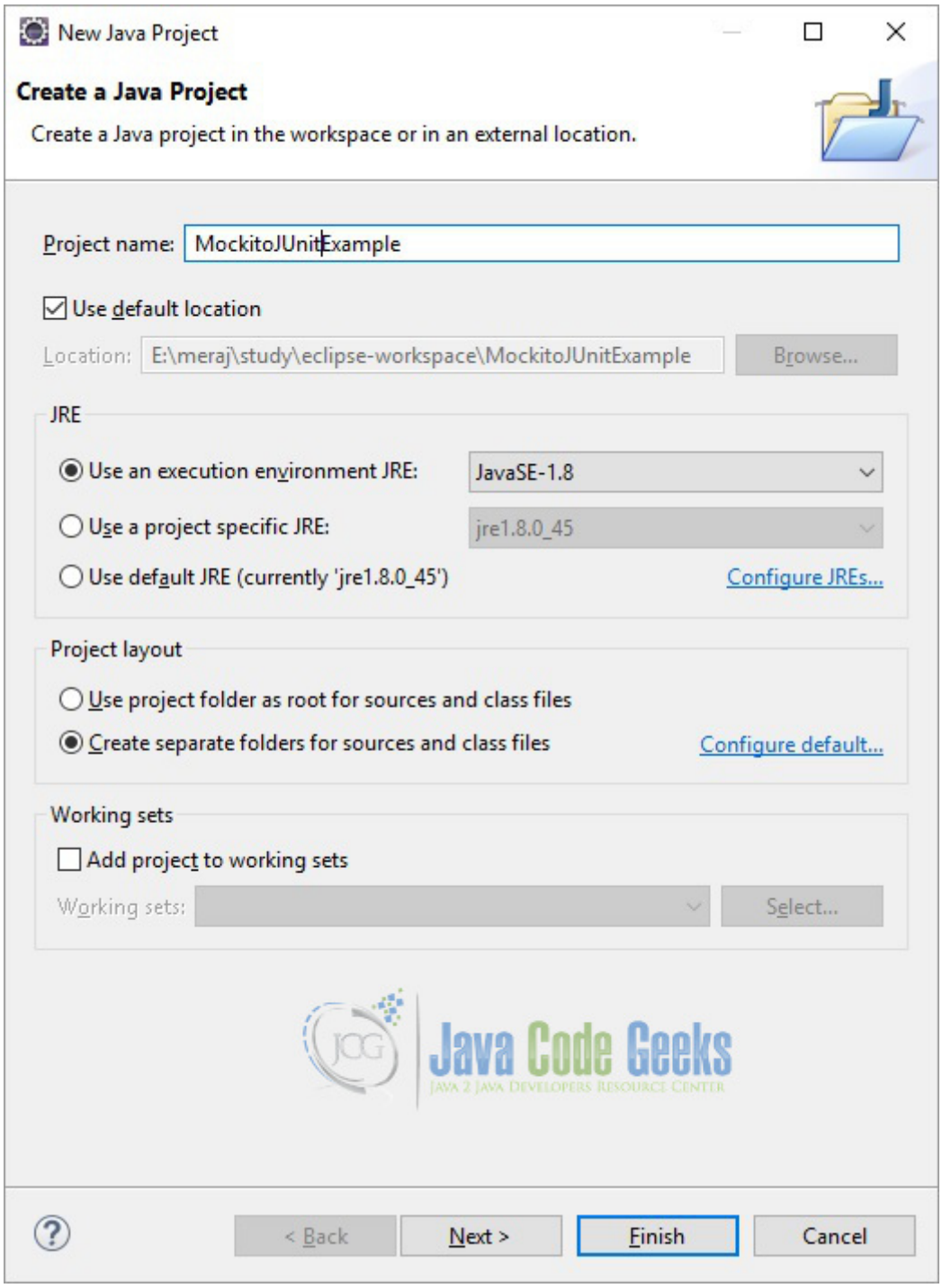

Figure 5.1: New Java Project

- Eclipse will create a 'src' folder. Right click on the 'src' folder and choose New⇒Package. In the 'Name' text-box enter 'com.javacodegeeks'. Click 'Finish'.

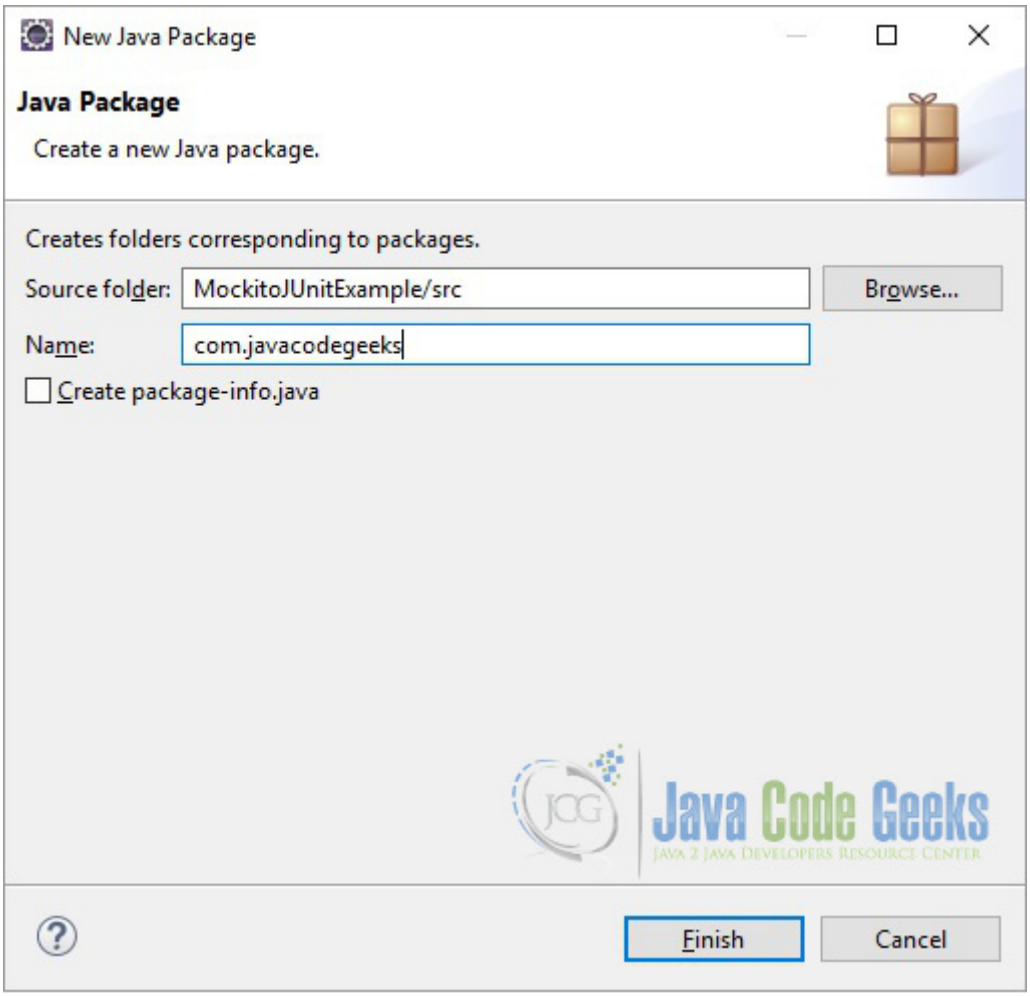

Figure 5.2: New Java Package

- Right click on the package and choose New⇒Class. Give the class name as JUnitMockitoExample. Click 'Finish'. Eclipse will create a default class with the given name.

Figure 5.3: New Java Class

5.2.1 Dependencies

For this example we need the junit and mockito jars. These jars can be downloaded from Maven repository. We are using 'junit-4.12.jar' and 'mockito-all-1.10.19.jar'. There are the latests (non-beta) versions available as per now. To add these jars in the classpath right click on the project and choose Build Path⇒Configure Build Path. The click on the 'Add External JARs' button on the right hand side. Then go to the location where you have downloaded these jars. Then click ok.

5.3 Verify interactions

In this section we will see how we can verify the mock object interactions. We will make use of the `java.util.Set` interface for this. First we will create the mock `Set` by calling the `org.mockito.Mockito.mock()` method and passing the `Set`

class to it as a parameter.

```
Set mockSet = mock(Set.class);
```

The `mock()` method creates mock object of given class or interface.

Now we will call two methods (`addAll() and clear()`) of the `Set` class on this mock object as shown below:

```
mockSet.addAll(toAdd);
mockSet.clear();
```

Now we will verify that these methods have been called

```
verify(mockSet).addAll(toAdd);
verify(mockSet).clear();
```

This verifies certain behavior happened once. Argument passed are compared using `equals()` method. Below is the snippet of the full method:

```
@Test
public void verifyInteractions() {
  Set mockSet = mock(Set.class);
  Set<String> toAdd = new HashSet<String>();

  mockSet.addAll(toAdd);
  mockSet.clear();

  verify(mockSet).addAll(toAdd);
  verify(mockSet).clear();
}
```

5.4 Stub method calls

In this section we will see how to stub method calls. We will again make use of the `Set` class for to demonstrate this. First we will create a mock of the `Set` class by calling the `mock()` method:

```
Set mockSet = mock(Set.class);
```

Now we will use the `when()` and `thenReturn()` method to define the behavior of `size()` method as below:

```
when(mockSet.size()).thenReturn(10);
```

To check that the stubbing is done correctly we will call the `size()` method to see what it returns.

```
Assert.assertEquals(10, mockSet.size());
```

Below is the snippet of the whole test method:

```
@Test
public void stubMethodCalls() {
  Set mockSet = mock(Set.class);
  when(mockSet.size()).thenReturn(10);
  Assert.assertEquals(10, mockSet.size());
}
```

5.5 Spy

Spy is used for partial mocking. It creates a spy of the real object. The spy calls real methods unless they are stubbed. Real spies should be used carefully and occasionally, for example when dealing with legacy code. Sometimes it's impossible or impractical to use `when(Object)` for stubbing spies. Therefore for spies it is recommended to always use `doReturn|Answer|Throw()|CallRealMethod` family of methods for stubbing.

```
@Test
public void testSpy() {
  List list = new LinkedList();
  List spy = spy(list);

  try {
    when(spy.get(0)).thenReturn("foo");
    } catch(IndexOutOfBoundsException e) {
      // Expected
    }

    doReturn("foo").when(spy).get(0);
}
```

Mockito does not delegate calls to the passed real instance, instead it actually creates a copy of it. So if you keep the real instance and interact with it, don't expect the spied to be aware of those interaction and their effect on real instance state. The corollary is that when an **unstubbed** method is called **on the spy** but **not on the real instance**, you won't see any effects on the real instance. Note that the spy won't have any annotations of the spied type, because CGLIB won't rewrite them. It may troublesome for code that rely on the spy to have these annotations.

5.6 InjectMocks

`@InjectMock` allows shorthand mock and spy injection. Mockito will try to inject mocks only either by constructor injection, setter injection, or property injection in order and as described below. If any of the following strategy fail, then Mockito won't report failure; i.e. you will have to provide dependencies yourself.

Constructor injection: the biggest constructor is chosen, then arguments are resolved with mocks declared in the test only. If the object is successfully created with the constructor, then Mockito won't try the other strategies. Mockito has decided to no corrupt an object if it has a parametered constructor. If arguments can not be found, then null is passed. If non-mockable types are wanted, then constructor injection won't happen. In these cases, you will have to satisfy dependencies yourself.

Property setter injection: mocks will first be resolved by type (if a single type match injection will happen regardless of the name), then, if there is several property of the same type, by the match of the property name and the mock name. If you have properties with the same type (or same erasure), it's better to name all @Mock annotated fields with the matching properties, otherwise Mockito might get confused and injection won't happen. If @InjectMocks instance wasn't initialized before and have a no-arg constructor, then it will be initialized with this constructor.

Field injection: mocks will first be resolved by type (if a single type match injection will happen regardless of the name), then, if there is several property of the same type, by the match of the field name and the mock name. If you have fields with the same type (or same erasure), it's better to name all @Mock annotated fields with the matching fields, otherwise Mockito might get confused and injection won't happen. If @InjectMocks instance wasn't initialized before and have a no-arg constructor, then it will be initialized with this constructor.

Now we will see an example of this. First we will create a domain class. This class represents the Report entity.

ReportEntity.java

```
package com.javacodegeeks.initmocks;

import java.util.Date;

/**
```

```
* Report entity.
* @author Meraj
*/
public class ReportEntity {

  private Long reportId;
  private Date startDate;
  private Date endDate;
  private byte[] content;

  public Long getReportId() {
    return reportId;
    }

  public void setReportId(Long reportId) {
    this.reportId = reportId;
  }

  public Date getStartDate() {
    return startDate;
  }

  public void setStartDate(Date startDate) {
    this.startDate = startDate;
  }

  public Date getEndDate() {
    return endDate;
  }

  public void setEndDate(Date endDate) {
    this.endDate = endDate;
  }

  public byte[] getContent() {
    return content;
  }

  public void setContent(byte[] content) {
    this.content = content;
  }
}
```

Now we will create create an interface which will refer to the above defined entity class.

IReportGenerator.java

```
package com.javacodegeeks.initmocks;

/**
* Interface for generating reports.
* @author Meraj
*/
public interface IReportGenerator {

  /**
  * Generate report.
  * @param report Report entity.
  */
  void generateReport(ReportEntity report);

}
```

Now we will define a service which will have reference to this interface.

ReportGeneratorService.java

```
package com.javacodegeeks.initmocks;

import java.util.Date;

/**
* Service class for generating report.
* @author Meraj
*/
public class ReportGeneratorService {

private IReportGenerator reportGenerator;

  /**
  * Generate report.
  * @param startDate start date
  * @param endDate end date
  * @param content report content
  */
  public void generateReport(Date startDate, Date endDate, byte[] content) {
    ReportEntity report = new ReportEntity();
    report.setContent(content);
    report.setStartDate(startDate);
    report.setEndDate(endDate);
    reportGenerator.generateReport(report);
  }
}
```

Now we will define out test class. In the test class we will will annotate the `ReportGeneratorService` class with `@InjectMocks`.

```
@InjectMocks private ReportGeneratorService reportGeneratorService;
```

The `IReportGenerator` class will be annotated with the `@Mock` annotation.

```
@Mock private IReportGenerator reportGenerator;
```

In the setup method we will initialize the mocks.

```
@Before
public void setUp() {
  MockitoAnnotations.initMocks(this);
}
```

5.7 Argument Matchers

Mockito verifies argument values in natural java style: by using an `equals()` method. Sometimes, when extra flexibility is required then you might use argument matchers. Argument matchers allow flexible verification or stubbing. If you are using argument matchers, all arguments have to be provided by matchers. Matcher methods like anyObject(), eq() do not return matchers. Internally, they record a matcher on a stack and return a dummy value (usually null). This implementation is due to static type safety imposed by the java compiler. The consequence is that you cannot use anyObject(), eq() methods outside of verified/stubbed method.

ArgumentCaptor is a special implementation of an argument matcher that captures argument values for further assertions:

```
ArgumentCaptor<Report> argument = ArgumentCaptor.forClass(Report.class);
verify(mock).doSomething(argument.capture());
assertEquals(ReportType.PAYMENT_REPORT, argument.getValue().getType());
```

5.8 Download the source file

In this example we saw how we can use Mockito to write JUnit tests.

Download

You can download the full source code of this example here: **Mockito JUnit Example**

Chapter 6

Mockito: How to mock void method call

A unit test should test a class in isolation. Side effects from other classes or the system should be eliminated if possible. Mockito lets you write beautiful tests with a clean & simple API. In this example we will learn how to mock a void method call using Mockito. Tools and technologies used in this example are Java 1.8, Eclipse Luna 4.4.2

6.1 Introduction

Mockito is a popular mocking framework which can be used in conjunction with JUnit. Mockito allows us to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly. We can create the mock objects manually or we can use the mocking framewors like Mockito, EasyMock. jMock etc. Mock frameworks allow us to create mock objects at runtime and define their behavior. The classical example for a mock object is a data provider. In production a real database is used, but for testing a mock object simulates the database and ensures that the test conditions are always the same.

6.2 Creating a project

Below are the steps required to create the project.

- Open Eclipse. Go to File⇒New⇒Java Project. In the ‘Project name’ enter ‘MockitoMockVoidMethod’.

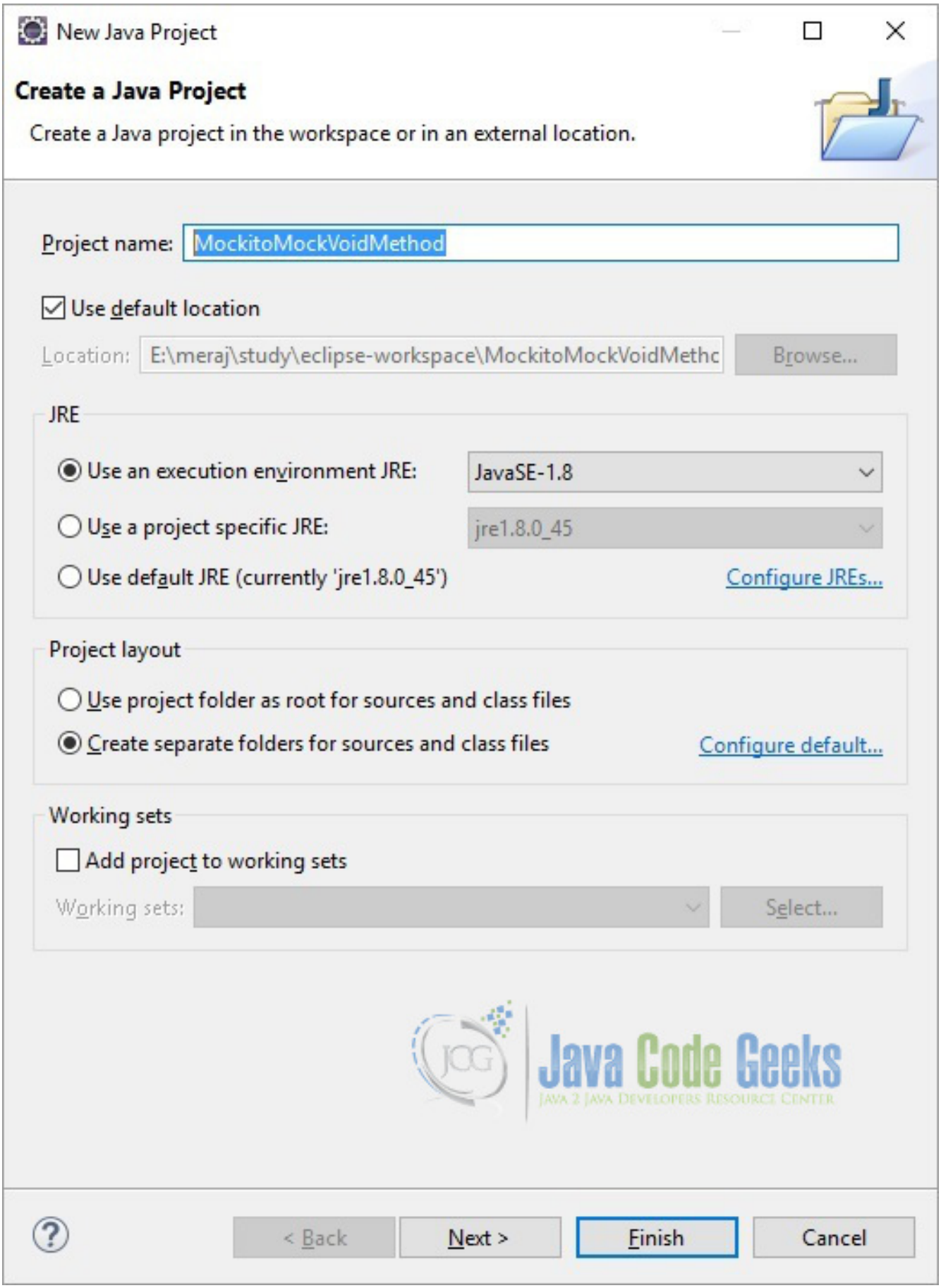

Figure 6.1: Create Java Project

- Eclipse will create a ‘src’ folder. Right click on the ‘src’ folder and choose New⇒Package. In the ‘Name’ text-box enter ‘com.javacodegeeks’. Click ‘Finish’.

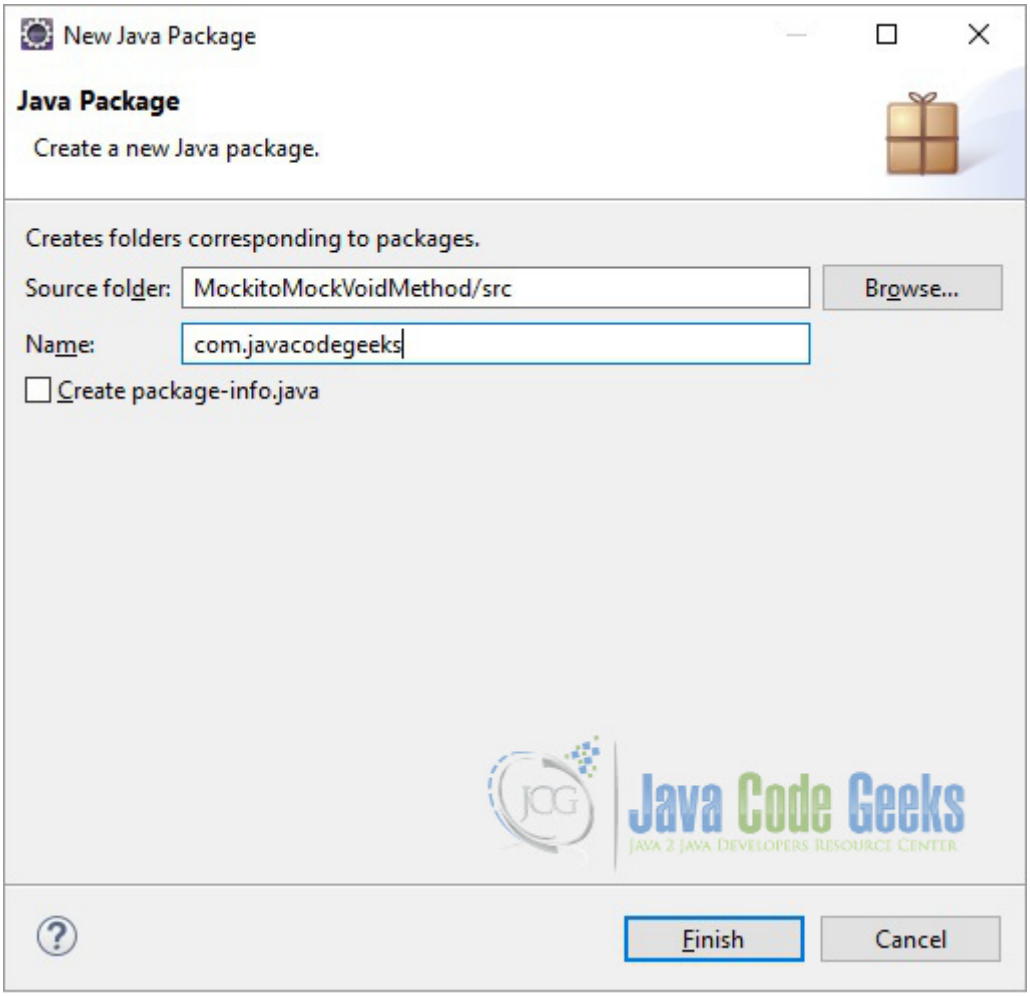

Figure 6.2: Java Package

- Right click on the package and choose New⇒Class. Give the class name and click ‘Finish’. Eclipse will create a default class with the given name.

6.2.1 Dependencies

For this example we need the junit and mockito jars. These jars can be downloaded from Maven repository. We are using ‘junit-4.12.jar’ and ‘mockito-all-1.10.19.jar’. There are the latests (non-beta) versions available as per now. To add these jars in the classpath right click on the project and choose Build Path⇒Configure Build Path. The click on the ‘Add External JARs’ button on the right hand side. Then go to the location where you have downloaded these jars. Then click ok.

6.3 Stub

The role of the test stub is to return controlled values to the object being tested. These are described as indirect inputs to the test. We replace a real object with a test-specific object that feeds the desired indirect inputs into the system under test.

6.3.1 doThrow()

In this section we will see how we can mock void methods which throw exceptions. To do this we make use of `doThrow()` method of Mockito class. Stubbing void methods requires a different approach from `when(Object)` because the compiler does not like void methods inside brackets.

```
doThrow(new Exception()).when(mockObject).methodWhichThrowException();
mockedObject.methodWhichThrowException();
```

6.3.2 doAnswer()

Use `doAnswer()` when you want to stub a void method with generic `org.mockito.stubbing.Answer`. `Answer` specifies an action that is executed and a return value that is returned when you interact with the mock.

```
doAnswer(new Answer() {
   public Object answer(InvocationOnMock invocation){
        Object[] args = invocation.getArguments();
        Mock mock = invocation.getMock();
        return null;
    }
}).when(mock).someMethod();
```

6.3.3 doNothing()

Use `doNothing()` for setting void methods to do nothing. Beware that void methods on mocks do nothing by default! However, there are rare situations when `doNothing()` comes handy:

6.3.3.1 Stubbing consecutive calls on a void method:

```
doNothing().doThrow(new IllegalArgumentException()).when(mockObject).someVoidMethod();

//does nothing the first time:
mockObject.someVoidMethod();

//throws IllegalArgumentException the next time:
mockObject.someVoidMethod();
```

6.3.3.2 When you spy real objects and you want the void method to do nothing:

```
Map map = new HashMap();
Map spy = spy(map);

//let's make clear() do nothing
doNothing().when(spy).clear();

spy.put("one", "1");

//clear() does nothing, so the map still contains "one", "1"
spy.clear();
```

6.4 Example

In this section we will see the working example of mocking a void method. First we will create a simple class with one void method.

VoidMethodClass.java

```
package com.javacodegeeks;

public class VoidMethodClass {

  public void voidMethodThrowingExcetion(boolean check) {
    if (check) {
      throw new IllegalArgumentException();
    }
  }
}
```

Now we will create a test class for this where we will mock this method using Mockito.

VoidMethodClassTest.java

```
package com.javacodegeeks;

import org.junit.Assert;
import org.junit.Test;
import org.mockito.Mockito;

public class VoidMethodClassTest {

  private VoidMethodClass mock;

  @Test
  public void testVoidMethodThrowingExcetion() {
    mock = Mockito.mock(VoidMethodClass.class);
    Mockito.doThrow(new IllegalArgumentException()).when(mock).voidMethodThrowingExcetion( ↩
        false);
    mock.voidMethodThrowingExcetion(true);
    Mockito.doThrow(new IllegalArgumentException()).when(mock).voidMethodThrowingExcetion( ↩
        true);
    try {
      mock.voidMethodThrowingExcetion(true);
      Assert.fail();
    } catch (IllegalArgumentException e) {
      // Expected
    }
  }
}
```

6.5 Download the source file

In this example we saw how we can mock void classes using Mockito

Download

You can download the full source code of this example here: **MockitoMockVoidMethod**

Chapter 7

Spring Test Mock Example

A unit test should test a class in isolation. Side effects from other classes or the system should be eliminated if possible. Mockito lets you write beautiful tests with a clean & simple API. In this example we will learn how to mock spring components using Mockito. Tools and technologies used in this example are Java 1.8, Eclipse Luna 4.4.2

7.1 Introduction

Mockito is a popular mocking framework which can be used in conjunction with JUnit. Mockito allows us to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly. We can create the mock objects manually or can use the mocking framewors like Mockito, EasyMock. jMock etc. Mock frameworks allow us to create mock objects at runtime and define their behavior. The classical example for a mock object is a data provider. In production a real database is used, but for testing a mock object simulates the database and ensures that the test conditions are always the same.

The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform.

Be able to unit test spring components without the need of loading the full spring-context is a very useful behavior provided by Mockito.

7.2 Creating a project

Below are the steps we need to take to create the project.

- Open Eclipse. Go to File⇒New⇒Java Project. In the ‘Project name’ enter ‘SpringTestMock’.
- Eclipse will create a ‘src’ folder. Right click on the ‘src’ folder and choose New⇒Package. In the ‘Name’ text-box enter ‘com.javacodegeeks’. Click ‘Finish’.

7.2.1 Dependencies

For this example we need the below mentioned jars:

- junit-4.1.2
- mockito-all-1.10.19
- spring-beans-4.2.5.RELEASE
- spring-context-4.2.5.RELEASE

These jars can be downloaded from Maven repository. These are the latest (non-beta) versions available as per now. To add these jars in the classpath right click on the project and choose Build Path⇒Configure Build Path. The click on the 'Add External JARs' button on the right hand side. Then go to the location where you have downloaded these jars. Then click ok.

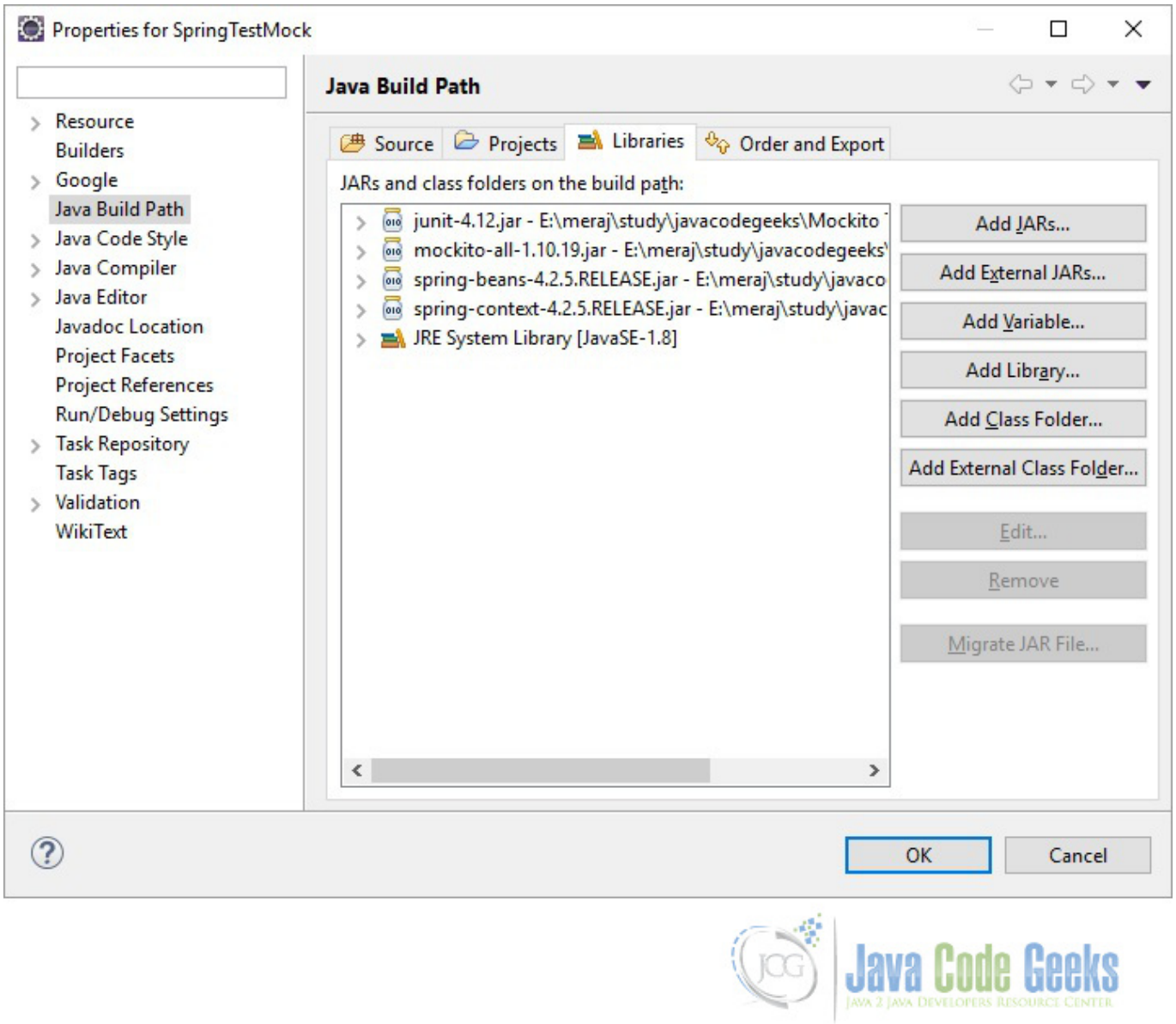

Figure 7.1: Dependencies

7.3 Code

To show how to use Mockito for mocking the Spring components we will use the User maintenance example. We will create a service class (UserMaintenanceService) with one method. This class will call the corresponding Data Access Object (DAO) to serve the request. First we will create a simple POJO class which represents the User domain entity.

User.java

```
package com.javacodegeeks;
```

```
import java.util.Date;

/**
* Class representing the user domain.
* @author Meraj
*/
public class User {

  private Long userId;
  private String firstName;
  private String surname;
  private Date dateOfBirth;

  public Long getUserId() {
    return userId;
  }

  public void setUserId(Long userId) {
    this.userId = userId;
  }

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getSurname() {
    return surname;
  }

  public void setSurname(String surname) {
    this.surname = surname;
  }

  public Date getDateOfBirth() {
    return dateOfBirth;
  }

  public void setDateOfBirth(Date dateOfBirth) {
    this.dateOfBirth = dateOfBirth;
  }
}
```

Now we will see how the DAO class looks like. The DAO class will be responsible for talking to the database. We will skip that part for this example. This class will be annotated as `@Component`. Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning

UserDao.java

```
package com.javacodegeeks;

import org.springframework.stereotype.Component;

/**
* DAO class for User related actions.
* @author Meraj
*/
@Component
public class UserDao {
```

```
  /**
  * Search for user using the id.
  * @param id user id
  * @return Retrieved user
  */
  public User findUserById(Long id) {
    // Find user details from database
    return new User();
  }
}
```

Now we will see how the service class looks like. This class will also be annotated with `@Component`. It has the reference to the UserDao class which it injects using the `@Autowired` annotation.

Autowire marks a constructor, field, setter method or config method as to be autowired by Spring's dependency injection facilities. Only one constructor (at max) of any given bean class may carry this annotation, indicating the constructor to autowire when used as a Spring bean. Such a constructor does not have to be public. Fields are injected right after construction of a bean, before any config methods are invoked. Such a config field does not have to be public. Config methods may have an arbitrary name and any number of arguments; each of those arguments will be autowired with a matching bean in the Spring container. Bean property setter methods are effectively just a special case of such a general config method. Such config methods do not have to be public.

UserMaintenanceService.java

```
package com.javacodegeeks;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

/**
* Service class for User related actions.
* @author Meraj
*/
@Component
public class UserMaintenanceService {

  @Autowired private UserDao userDao;

  /**
  * Find user.
  * @param userId user id
  * @return Retrieved user
  */
  public User findUserById(Long userId) {
  // Do business validations.
    return userDao.findUserById(userId);
  }
}
```

7.4 Test

Below is the test class which we will use to test in this example.

UserMaintenanceServiceTest.java

```
package com.javacodegeeks;

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

```
import static org.junit.Assert.fail;
import static org.mockito.Mockito.when;
import static org.mockito.MockitoAnnotations.initMocks;

import java.util.Date;

import org.junit.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;

public class UserMaintenanceServiceTest {

  @InjectMocks private UserMaintenanceService userMaintenanceService;
  @Mock private UserDao userDao;

  @Test
  public void testFindUserByIdPositive() {
    initMocks(this);
    when(userDao.findUserById(1000L)).thenReturn(getMeTestUser());
    User user = userMaintenanceService.findUserById(1000L);
    assertNotNull(user);
    assertEquals("Test first name", user.getFirstName());
    assertEquals("Test surname", user.getSurname());
  }

  @Test (expected = NullPointerException.class)
  public void testFindUserByIdNegetive() {
    userMaintenanceService = new UserMaintenanceService();
    userMaintenanceService.findUserById(1000L);
    fail();
}

  private User getMeTestUser() {
    User user = new User();
    user.setUserId(1000L);
    user.setFirstName("Test first name");
    user.setSurname("Test surname");
    user.setDateOfBirth(new Date());
    return user;
  }
}
```

Now we will discuss few things in this class. If you would have notice you will see that the UserMaintenanceService class is annotated with `@InjectMocks`. This marks a field on which injection should be performed. It minimizes repetitive mock and spy injection. Mockito will try to inject mocks only either by constructor injection, setter injection, or property injection in order and as described below. If any of the following strategy fail, then Mockito won't report failure; i.e. you will have to provide dependencies yourself.

- **Constructor injection:** the biggest constructor is chosen, then arguments are resolved with mocks declared in the test only. **Note:** If arguments can not be found, then null is passed. If non-mockable types are wanted, then constructor injection won't happen. In these cases, you will have to satisfy dependencies yourself.
- **Property setter injection:** mocks will first be resolved by type, then, if there is several property of the same type, by the match of the property name and the mock name. **Note:** If you have properties with the same type (or same erasure), it's better to name all `@Mock` annotated fields with the matching properties, otherwise Mockito might get confused and injection won't happen. If `@InjectMocks` instance wasn't initialized before and have a no-arg constructor, then it will be initialized with this constructor.
- **Field injection** mocks will first be resolved by type, then, if there is several property of the same type, by the match of the field name and the mock name. **Note:** If you have fields with the same type (or same erasure), it's better to name all

`@Mock` annotated fields with the matching fields, otherwise Mockito might get confused and injection won't happen. If `@InjectMocks` instance wasn't initialized before and have a no-arg constructor, then it will be initialized with this constructor.

The UserDao class is annotated with `@Mock`. This is the class which we want to mock.

In the first test method the first thing we do is call the `MockitoAnnotations.initMocks()` method. It initializes objects annotated with `@Mock` for given test class. Then we define the behaviour of the DAO class method by using the `org.mockito.Mockito.when()`. We return our own test User object here.

In the second test we are not calling the `MockitoAnnotations.initMocks()` so the DAO class will not be injected in this case hence it will throw `NullPointerException`.

7.5 Download the source file

This was an example of mocking spring components.

Download

You can download the full source code of this example here: **SpringTestMock**

Chapter 8

Mockito Captor Example

A unit test should test a class in isolation. Side effects from other classes or the system should be eliminated if possible. Mockito lets you write beautiful tests with a clean & simple API. In this example we will learn how to use ArgumentCaptor class/ Captor annotation of Mockito. Tools and technologies used in this example are Java 1.8, Eclipse Luna 4.4.2

8.1 Introduction

Mockito is a popular mocking framework which can be used in conjunction with JUnit. Mockito allows us to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly. We can create the mock objects manually or can use the mocking framewors like Mockito, EasyMock. jMock etc. Mock frameworks allow us to create mock objects at runtime and define their behavior. The classical example for a mock object is a data provider. In production a real database is used, but for testing a mock object simulates the database and ensures that the test conditions are always the same.

8.2 Creating a project

Below are the steps we need to take to create the project.

- Open Eclipse. Go to File⇒New⇒Java Project. In the 'Project name' enter 'MockitoCaptorExample'.
- Eclipse will create a 'src' folder. Right click on the 'src' folder and choose New⇒Package. In the 'Name' text-box enter 'com.javacodegeeks'. Click 'Finish'.
- Right click on the package and choose New⇒Class. Give the class name as MockitoCaptorExample. Click 'Finish'. Eclipse will create a default class with the given name.

8.2.1 Dependencies

For this example we need the junit and mockito jars. These jars can be downloaded from Maven repository. We are using 'junit-4.12.jar' and 'mockito-all-1.10.19.jar'. There are the latests (non-beta) versions available as per now. To add these jars in the classpath right click on the project and choose Build Path⇒Configure Build Path. The click on the 'Add External JARs' button on the right hand side. Then go to the location where you have downloaded these jars. Then click ok.

8.3 ArgumentCaptor class

`ArgumentCaptor` class is used to capture argument values for further assertions. Mockito verifies argument values in natural java style: by using an `equals()` method. This is also the recommended way of matching arguments because it makes tests clean & simple. In some situations though, it is helpful to assert on certain arguments after the actual verification. For example:

```
ArgumentCaptor<Contact> argument = ArgumentCaptor.forClass(Contact.class);
verify(mockClass).doSomething(argument.capture());
assertEquals("Meraj", argument.getValue().getName());
```

It is recommended to use `ArgumentCaptor` with verification but not with stubbing. Using `ArgumentCaptor` with stubbing may decrease test readability because captor is created outside of assert (aka verify or *then*) block. Also it may reduce defect localization because if stubbed method was not called then no argument is captured.

In a way `ArgumentCaptor` is related to custom argument matchers. Both techniques can be used for making sure certain arguments where passed to mocks. However, `ArgumentCaptor` may be a better fit if:

- custom argument matcher is not likely to be reused
- you just need it to assert on argument values to complete verification

Custom argument matchers via `ArgumentMatcher` are usually better for stubbing.

8.3.1 Methods

In this section we will describe the methods defined in the `ArgumentCaptor` class.

8.3.1.1 public T capture()

Use it to capture the argument. This method must be used inside of verification. Internally, this method registers a special implementation of an `ArgumentMatcher`. This argument matcher stores the argument value so that you can use it later to perform assertions.

8.3.1.2 public T getValue()

Returns the captured value of the argument. If the method was called multiple times then it returns the latest captured value.

8.3.1.3 public java.util.List<T> getAllValues()

Returns all captured values. Use it in case the verified method was called multiple times.

8.4 Captor annotation

Captor annotation allows shorthand `ArgumentCaptor` creation on fields. One of the advantages of using `@Captor` annotation is that you can avoid warnings related capturing complex generic types. The Captor annotation is defined as below:

```
@Retention(value=RUNTIME)
@Target(value=FIELD)
@Documented
public @interface Captor
```

8.5 Code

In this section first we wee see a simple example of using the @Captor annotation. Then we will discuss a more complex one.

8.5.1 Simple Code

For this simple example we will make use of the java.util.Stack class. We will create a stack of strings then add one value to it. Then we will capture he argument and verify it. Below is the code snippet for this:

```
stack.add("Java Code Geeks");
Mockito.verify(stack).add(argumentCaptor.capture());
assertEquals("Java Code Geeks", argumentCaptor.getValue());
```

In the second example we will add two values in the Stack and will extract all the values using the getAllValues() method. Below is the code snippet for this:

```
stack.add("Java Code Geeks");
stack.add("Mockito");
Mockito.verify(stack, Mockito.times(2)).add(argumentCaptor.capture());
List<String> values = argumentCaptor.getAllValues();
assertEquals("Java Code Geeks", values.get(0));
assertEquals("Mockito", values.get(1));
```

Below is the code which shows the usage of `@Captor` annotation

MockitoCaptorExample.java

```
package com.javacodegeeks;

import static org.junit.Assert.assertEquals;

import java.util.Stack;

import org.junit.Before;
import org.junit.Test;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.MockitoAnnotations;

public class MockitoCaptorExample {

  @Mock Stack<String> stack;
  @Captor ArgumentCaptor<String> argumentCaptor;

  @Before
  public void setUp() {
    MockitoAnnotations.initMocks(this);
  }

  @Test
  public void test() throws Exception {
    stack.add("Java Code Geeks");
    Mockito.verify(stack).add(argumentCaptor.capture());
    assertEquals("Java Code Geeks", argumentCaptor.getValue());
  }
}
```

8.5.2 Stub example

In this section we will see how we can use @Captor for stubbing. We will use the report generation example.

Create an interface with one method.

IReportGenerator.java

```
package com.javacodegeeks;

/**
* Interface for generating reports.
* @author Meraj
*/
public interface IReportGenerator {

  /**
  * Generate report.
  * @param report Report entity.
  */
  void generateReport(ReportEntity report);
}
```

Now we will create the report entity class which is a simple POJO class.

ReportEntity.java

```
package com.javacodegeeks;

import java.util.Date;

/**
* Report entity.
* @author Meraj
*/
public class ReportEntity {

  private Long reportId;
  private Date startDate;
  private Date endDate;
  private byte[] content;

  public Long getReportId() {
    return reportId;
  }

  public void setReportId(Long reportId) {
    this.reportId = reportId;
  }

  public Date getStartDate() {
    return startDate;
  }

  public void setStartDate(Date startDate) {
    this.startDate = startDate;
  }

  public Date getEndDate() {
    return endDate;
  }

  public void setEndDate(Date endDate) {
    this.endDate = endDate;
  }

  public byte[] getContent() {
    return content;
  }
```

```
  public void setContent(byte[] content) {
    this.content = content;
  }
}
```

Now we will have a look at the service class which we will use to generate the report.

ReportGeneratorService.java

```
package com.javacodegeeks;

import java.util.Date;

/**
* Service class for generating report.
* @author Meraj
*/
public class ReportGeneratorService {

  private IReportGenerator reportGenerator;

  /**
  * Generate report.
  * @param startDate start date
  * @param endDate end date
  * @param content report content
  */
  public void generateReport(Date startDate, Date endDate, byte[] content) {
    ReportEntity report = new ReportEntity();
    report.setContent(content);
    report.setStartDate(startDate);
    report.setEndDate(endDate);
    reportGenerator.generateReport(report);
  }
}
```

Now we will look at the test.

ReportGeneratorServiceTest.java

```
package com.javacodegeeks;

import static org.junit.Assert.assertEquals;

import java.util.Calendar;

import org.junit.Before;
import org.junit.Test;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.MockitoAnnotations;

public class ReportGeneratorServiceTest {

  @InjectMocks private ReportGeneratorService reportGeneratorService;
  @Mock private IReportGenerator reportGenerator;
  @Captor private ArgumentCaptor<ReportEntity> reportCaptor;

  @Before
  public void setUp() {
```

```
    MockitoAnnotations.initMocks(this);
  }

  @SuppressWarnings("deprecation")
  @Test
  public void test() {
    Calendar startDate = Calendar.getInstance();
    startDate.set(2016, 11, 25);
    Calendar endDate = Calendar.getInstance();
    endDate.set(9999, 12, 31);
    String reportContent = "Report Content";
    reportGeneratorService.generateReport(startDate.getTime(), endDate.getTime(),  ↩
        reportContent.getBytes());

    Mockito.verify(reportGenerator).generateReport(reportCaptor.capture());

    ReportEntity report = reportCaptor.getValue();

    assertEquals(116, report.getStartDate().getYear());
    assertEquals(11, report.getStartDate().getMonth());
    assertEquals(25, report.getStartDate().getDate());

    assertEquals(8100, report.getEndDate().getYear());
    assertEquals(0, report.getEndDate().getMonth());
    assertEquals(31, report.getEndDate().getDate());

    assertEquals("Report Content", new String(report.getContent()));
  }
}
```

8.6 Download the source file

This was an example of `Mockito Captor annotation`.

Download

You can download the full source code of this example here: **Mockito Captor Example**

Chapter 9

Mockito ThenReturn Example

In this example we will learn how to use *thenReturn* method of Mockito. A unit test should test a class in isolation. Side effects from other classes or the system should be eliminated if possible. Mockito lets you write beautiful tests with a clean & simple API. Tools and technologies used in this example are Java 1.8, Eclipse Luna 4.4.2

9.1 Introduction

Mockito is a popular mocking framework which can be used in conjunction with JUnit. Mockito allows us to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly. We can create the mock objects manually or can use the mocking framewors like Mockito, EasyMock. jMock etc. Mock frameworks allow us to create mock objects at runtime and define their behavior. The classical example for a mock object is a data provider. In production a real database is used, but for testing a mock object simulates the database and ensures that the test conditions are always the same.

9.2 Creating a project

Below are the steps we need to take to create the project.

- Open Eclipse. Go to File⇒New⇒Java Project. In the *Project name* enter *MockitoThenReturnExample*.
- Eclipse will create a *src* folder. Right click on the *src* folder and choose New⇒Package. In the *Name* text-box enter *com.javacodegeeks*. Click *Finish*.
- Right click on the package and choose New⇒Class. Give the class name as ThenReturnExampleTest. Click *Finish*. Eclipse will create a default class with the given name.

9.2.1 Dependencies

For this example we need the junit and mockito jars. These jars can be downloaded from Maven repository. We are using *junit-4.12.jar* and *mockito-all-1.10.19.jar*. There are the latests (non-beta) versions available as per now. To add these jars in the classpath right click on the project and choose Build Path⇒Configure Build Path. The click on the *Add External JARs* button on the right hand side. Then go to the location where you have downloaded these jars. Then click ok.

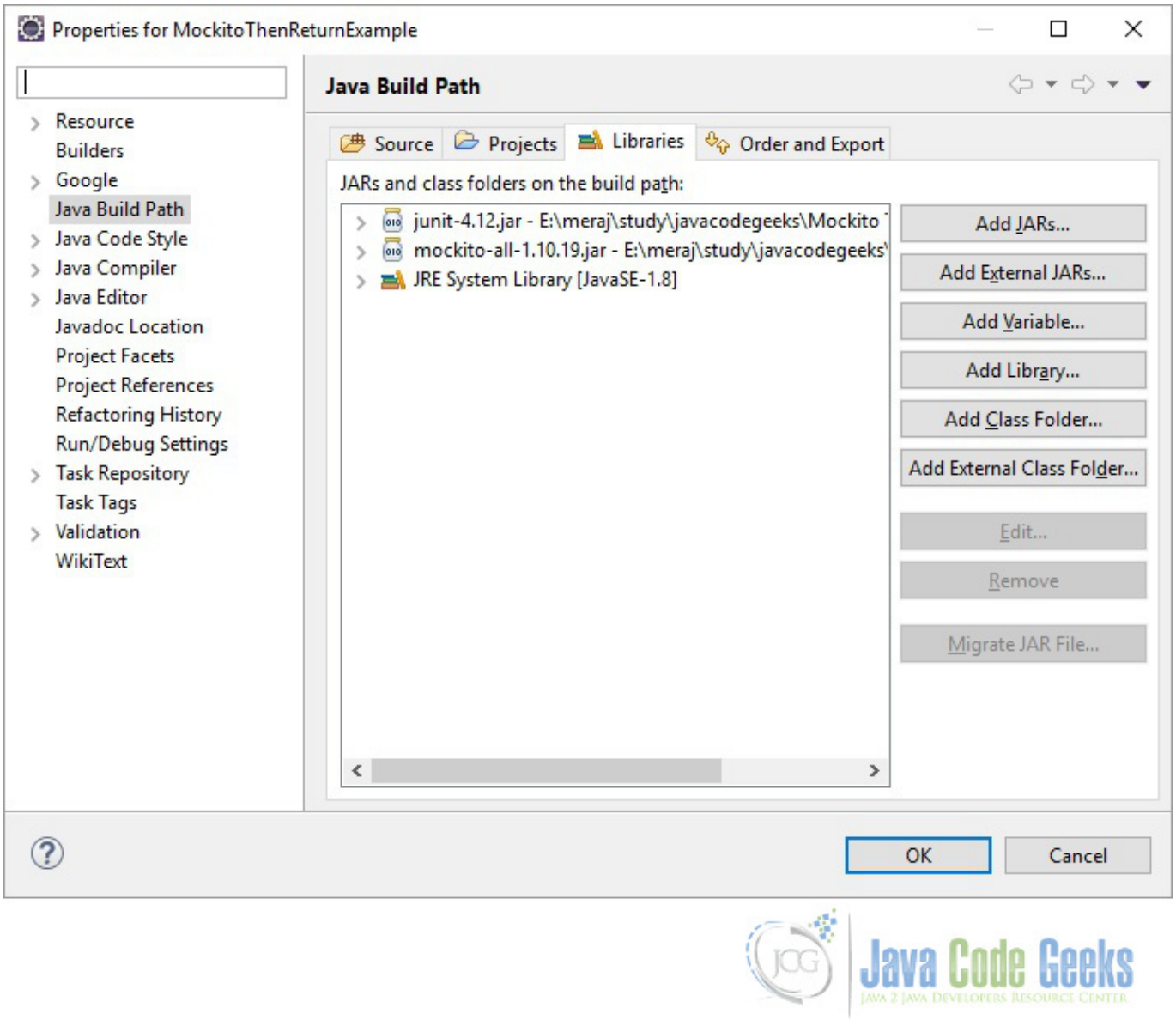

Figure 9.1: Add External JAR

9.3 thenReturn

The `thenReturn()` methods lets you define the return value when a particular method of the mocked object is been called. The below snippet shows how we use `thenReturn` to check for multiple values.

```
Iterator i = mock(Iterator.class);
when(i.next()).thenReturn("Java Code Geeks").thenReturn("Mockito");
String result = i.next() + " " + i.next();
System.out.println(result);
```

The first time `next()` method is called *Java Code Geeks* is returned and when it's called the second time *Mockito* is returned. So the result is Java Code Geeks Mockito.

The below code snippet shows how to return values based on input parameter.

```
Comparable c= mock(Comparable.class);
when(c.compareTo("Java Code Geeks")).thenReturn(100);
when(c.compareTo("Mockito")).thenReturn(200);
assertEquals(200,c.compareTo("Mockito"));
```

The code snippet below shows how you can return the same value independent of the value of the parameter passed.

```
Comparable c = mock(Comparable.class);
when(c.compareTo(anyInt())).thenReturn(0);
assertEquals(0 ,c.compareTo(9));
```

9.4 Code

Below is the test class we will use to show the usage of *thenReturn()*. This class can be run as a JUnit test from eclipse.

ThenReturnExampleTest.java

```
package com.javacodegeeks;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
import static org.mockito.Matchers.anyInt;

import java.util.Iterator;

import org.junit.Test;

@SuppressWarnings({"rawtypes", "unchecked"})
public class ThenReturnExampleTest {

        /**
        * This will test multiple return values.
        * @throws Exception
        */
        @Test
        public void test1() throws Exception {
                Iterator i = mock(Iterator.class);
                when(i.next()).thenReturn("Java Code Geeks").thenReturn("Mockito");
                String result = i.next() + " " + i.next();
                assertEquals("Java Code Geeks Mockito", result);
        }

        /**
        * This test demonstrates how to return values based on the input
        */
        @Test
        public void test2() {
                Comparable c= mock(Comparable.class);
                when(c.compareTo("Java Code Geeks")).thenReturn(100);
                when(c.compareTo("Mockito")).thenReturn(200);
                assertEquals(200,c.compareTo("Mockito"));
        }

        /**
        * This test demonstrates how to return values independent of the input value
        */
        @Test
        public void test3() {
```

```
                Comparable c = mock(Comparable.class);
                when(c.compareTo(anyInt())).thenReturn(0);
                assertEquals(0 ,c.compareTo(9));
                }
        }
}
```

9.5 Download the source file

This was an example of Mockito `thenReturn()`.

Download

You can download the full source code of this example here **Mockito thenReturn Example**