SLF4J

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia https://twitter.com/tutorialspoint

About the Tutorial

SLF4J stands for Simple Logging Facade for Java. It provides a simple abstraction of all the logging frameworks. It enables a user to work with any of the logging frameworks such as Log4j, Logback, JUL (java.util.logging), etc. using single dependency.

Audience

This tutorial has been prepared for beginners to help them understand the basic functionality of SLF4J logging framework.

Prerequisites

As you are going to use SLG4J logging framework in various Java-based application development, it is imperative that you should have a good understanding of Java programming language.

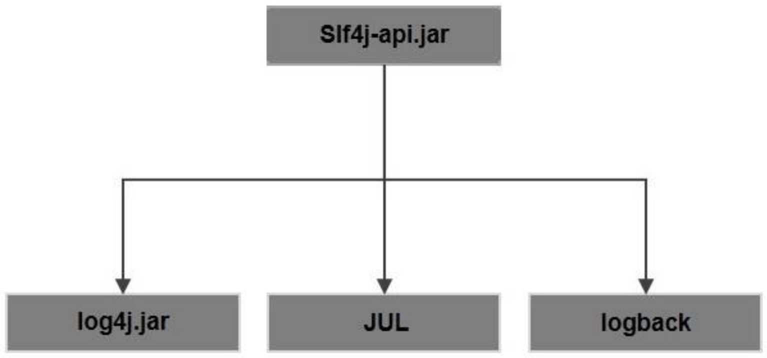
Copyright & Disclaimer

Table of Contents

1. SLF4J — Overview

SLF4J stands for **S**imple **L**ogging **F**acade for **J**ava. It provides a simple abstraction of all the logging frameworks in Java. Thus, it enables a user to work with any of the logging frameworks such as Log4j, Logback and **JUL** (java.util.logging) using single dependency. You can migrate to the required logging framework at run-time/deployment time.

Ceki Gülcü created SLF4J as an alternative to Jakarta commons-logging framework.



Advantages of SLF4J

Following are the advantages of SLF4J:

- Using SLF4J framework, you can migrate to the desired logging framework at the time of deployment.

- Slf4J provides bindings to all popular logging frameworks such as log4j, JUL, Simple logging and, NOP. Therefore, you can switch to any of these popular frameworks at the time of deployment.

- SLF4J provides support to parameterized logging messages irrespective of the binding you use.

- Since SLF4J decouples application and logging framework, you can easily write applications independent of logging frameworks. You need not bother about the logging framework being used to write an application.

- SLF4J provides a simple Java tool known as migrator. Using this tool, you can migrate existing projects, which use logging frame works like Jakarta Commons Logging (JCL) or, log4j or, Java.util.logging (JUL) to SLF4J.

2. SLF4J — Logging Frameworks

Logging in programming, refers to recording activities/events. Usually, the application developers should take care of logging.

To make the job of logging easier, Java provides various frameworks – log4J, java.util.logging (JUL), tiny log, logback, etc.

Logging Framework Overview

A logging framework usually contains three elements:

Logger

Captures the message along with the metadata.

Formatter

Formats the messages captured by the logger.

Handler

The Handler or appender finally dispatches the messages either by printing on the console or, by storing in the database or, by sending through an email.

Some frameworks combine the logger and appender elements to speed up the operations.

Logger Object

To log a message, the application sends a logger object (sometimes along with the exceptions if any) with name and security level.

Severity Level

The messages logged will be of various levels. The following table lists down the general levels of logging.

Severity level	Description
Fatal	Severe issue that causes the application to terminate.
ERROR	Runtime errors.
WARNING	In most cases, the errors are due to the usage of deprecated APIs.

INFO	Events that occur at runtime.
DEBUG	Information about the flow of the system.
TRACE	More detailed information about the flow of the system.

3. SLF4J — SLF4J Vs Log4j

What is log4j?

log4j is a reliable, fast and flexible **logging framework (APIs) written in Java**, which is distributed under the Apache Software License.

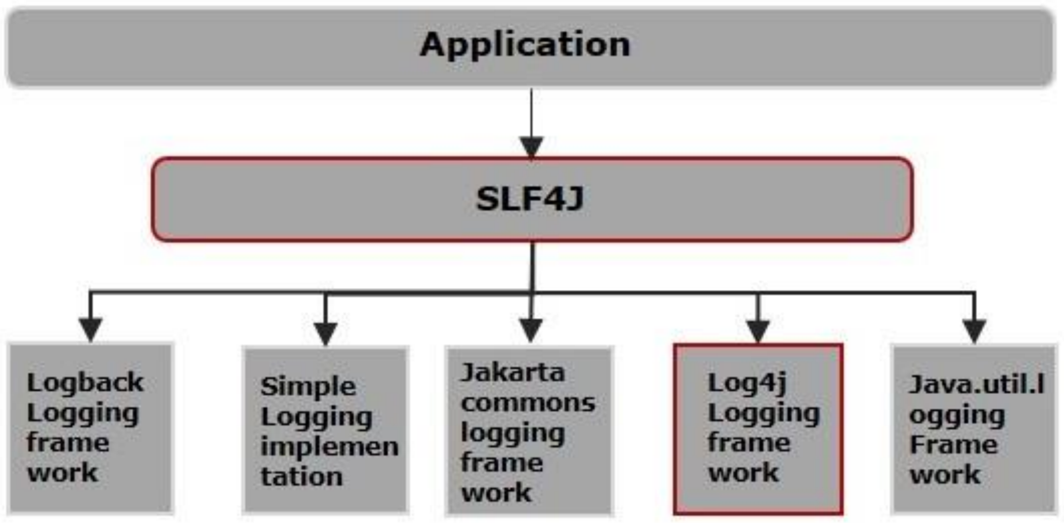
log4j is highly configurable through external configuration files at runtime. It views the logging process in terms of levels of priorities and offers mechanisms to direct logging information to a great variety of destinations, such as a database, file, console, UNIX Syslog, etc. (for more details on log4j refer our Tutorial).

Comparison SLF4J and Log4j

Unlike log4j, SLF4J (**S**imple **L**ogging **F**acade for **J**ava) is not an implementation of logging framework, it is an **abstraction for all those logging frameworks in Java similar to log4J**. Therefore, you cannot compare both. However, it is always difficult to prefer one between the two.

If you have a choice, logging abstraction is always preferable than logging framework. If you use a logging abstraction, SLF4J in particular, you can migrate to any logging framework you need at the time of deployment without opting for single dependency.

Observe the following diagram to have a better understanding.


Application
SLF4J
Logback Logging frame work
Simple Logging implemen tation
Jakarta commons logging frame work
Log4j Logging frame work
Java.util.l ogging Frame work

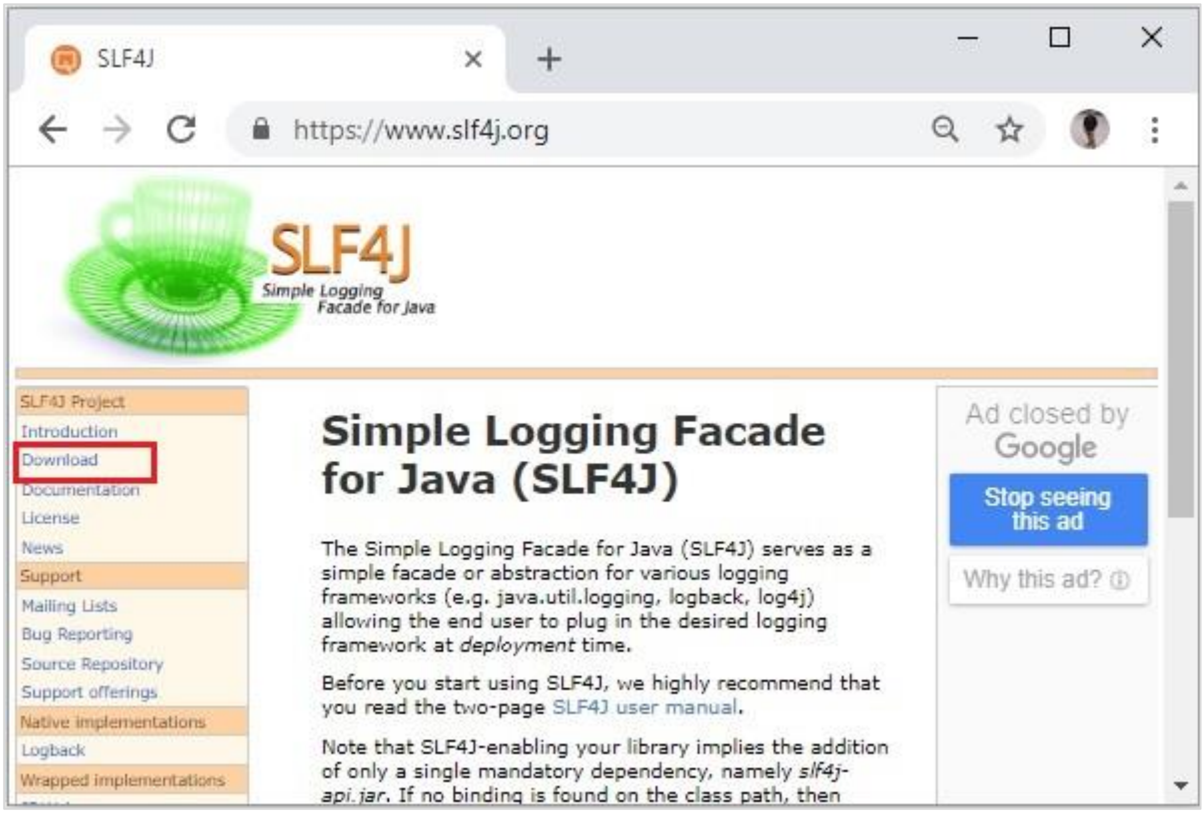
4. SLF4J — Environment Setup

In this chapter, we will explain how to set SLF4J environment in Eclipse IDE. Before proceeding with the installation, make sure that you already have Eclipse installed in your system. If not, download and install Eclipse.

For more information on Eclipse, please refer our **Eclipse Tutorial**.

Step 1: Download the dependency JAR file

Open the official homepage of the SLF4J website and go to the download page.



Simple Logging Facade for Java (SLF4J)

The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at *deployment* time.

Before you start using SLF4J, we highly recommend that you read the two-page SLF4J user manual.

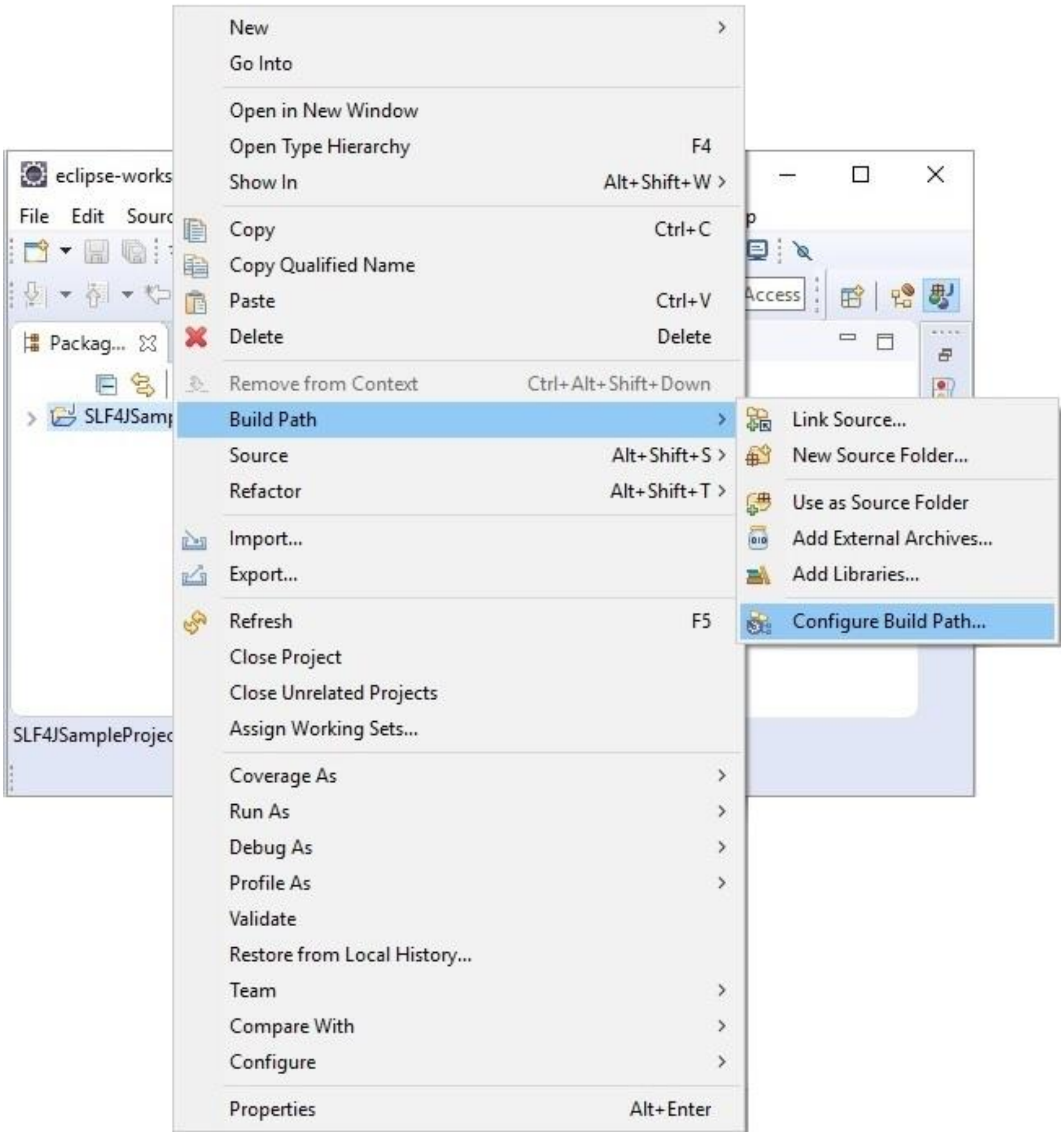
Note that SLF4J-enabling your library implies the addition of only a single mandatory dependency, namely *slf4j-api.jar*. If no binding is found on the class path, then

Now, download the latest stable version of ***slf4j-X.X.tar.gz*** or *slf4j-X.X.zip*, according to your operating system (if windows .zip file or if Linux tar.gz file).

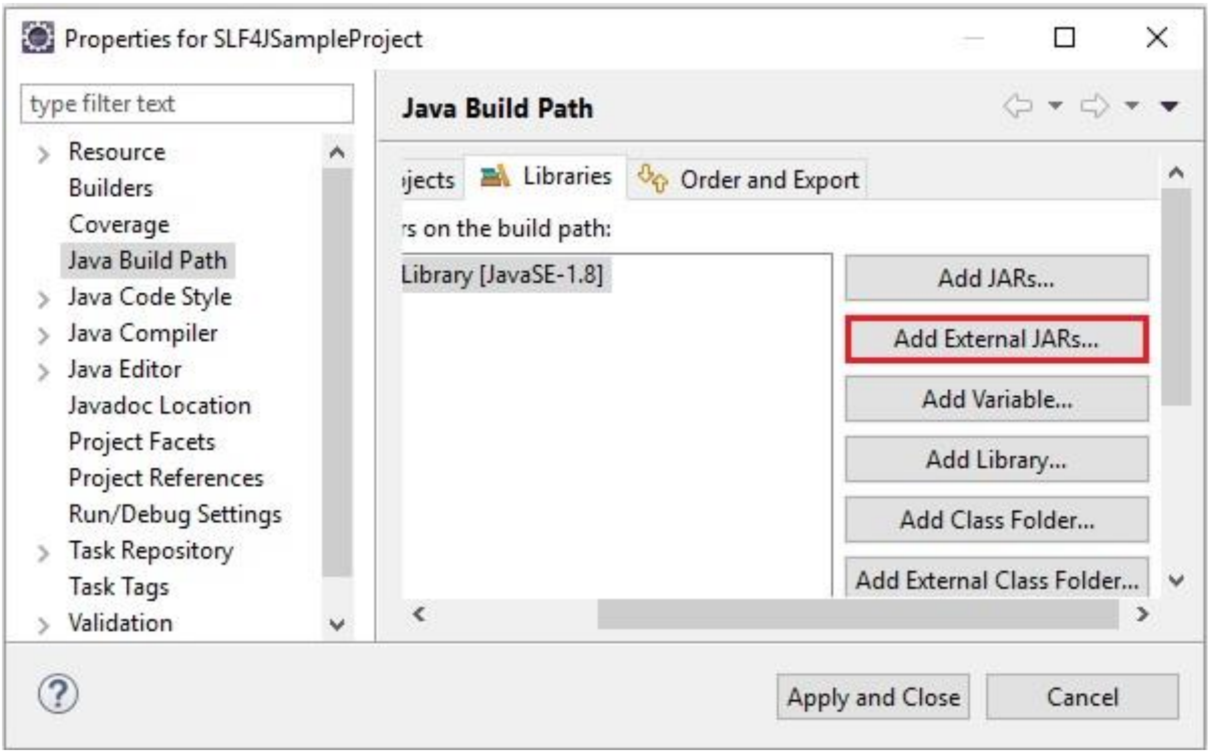
Within the downloaded folder, you will find *slf4j-api-X.X.jar*. This is the required Jar file.

Step 2: Create a project and set build path

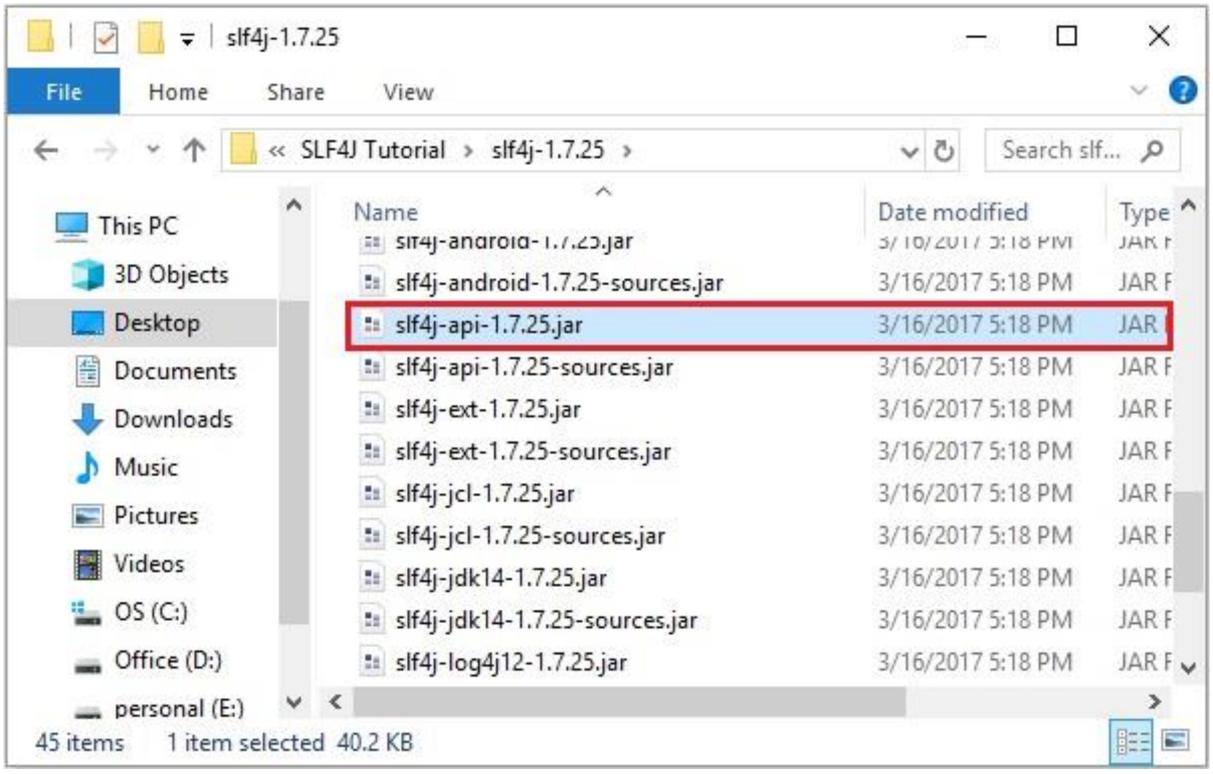
Open eclipse and create a sample project. Right-click on the project, select the option **Build Path -> Configure Build Path…** as shown below.



In the **Java Build Path** frame in the **Libraries** tab, click **Add External JARs…**



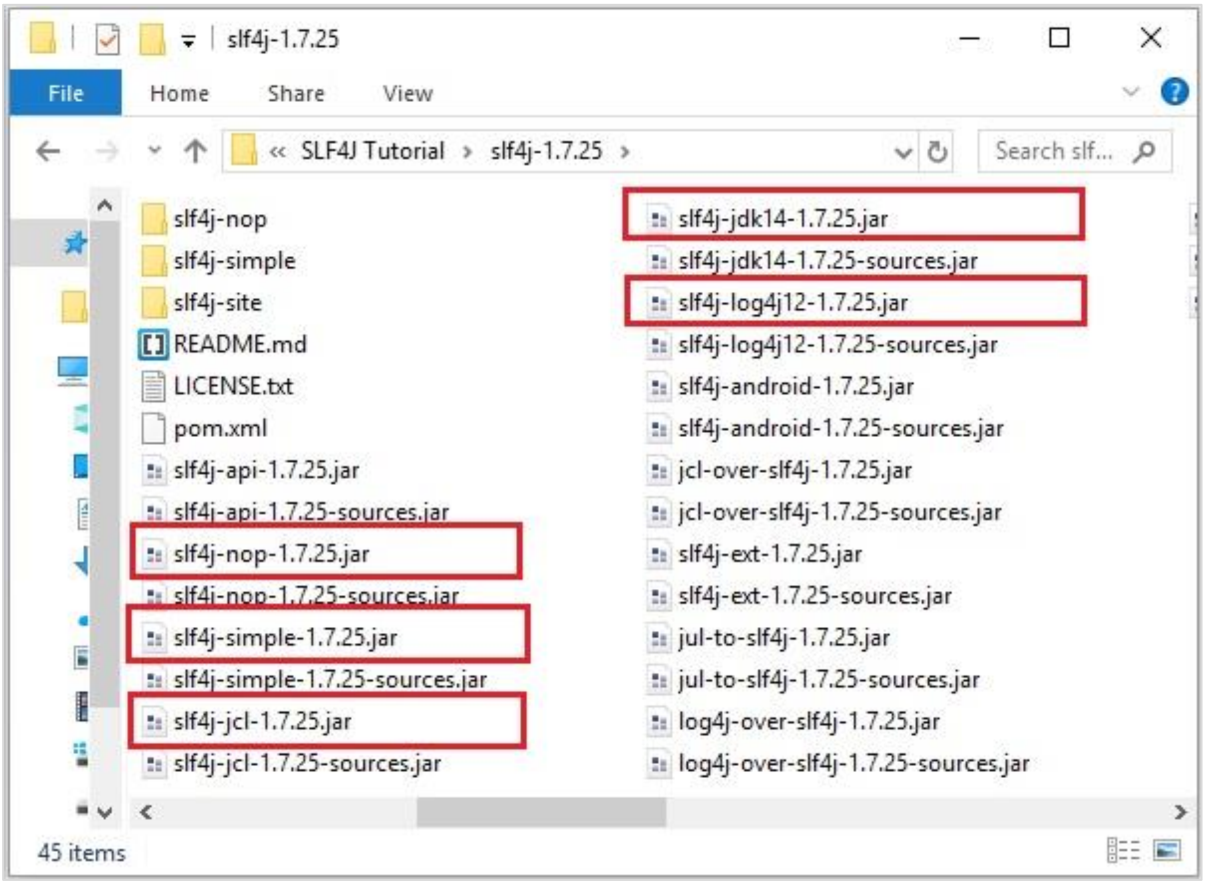
Select the **slf4j-api.x.x.jar** file downloaded and click **Apply and Close.**



Name	Date modified	Type
slf4j-android-1.7.25.jar	3/16/2017 5:18 PM	JAR F
slf4j-android-1.7.25-sources.jar	3/16/2017 5:18 PM	JAR F
slf4j-api-1.7.25.jar	3/16/2017 5:18 PM	JAR
slf4j-api-1.7.25-sources.jar	3/16/2017 5:18 PM	JAR F
slf4j-ext-1.7.25.jar	3/16/2017 5:18 PM	JAR F
slf4j-ext-1.7.25-sources.jar	3/16/2017 5:18 PM	JAR F
slf4j-jcl-1.7.25.jar	3/16/2017 5:18 PM	JAR F
slf4j-jcl-1.7.25-sources.jar	3/16/2017 5:18 PM	JAR F
slf4j-jdk14-1.7.25.jar	3/16/2017 5:18 PM	JAR F
slf4j-jdk14-1.7.25-sources.jar	3/16/2017 5:18 PM	JAR F
slf4j-log4j12-1.7.25.jar	3/16/2017 5:18 PM	JAR F

SLF4J Bindings

In addition to **slf4j-api.x.x.jar** file, **SLF4J** provides several other Jar files as shown below. These are called **SLF4J bindings**.



slf4j-nop
slf4j-simple
slf4j-site
README.md
LICENSE.txt
pom.xml
slf4j-api-1.7.25.jar
slf4j-api-1.7.25-sources.jar
slf4j-nop-1.7.25.jar
slf4j-nop-1.7.25-sources.jar
slf4j-simple-1.7.25.jar
slf4j-simple-1.7.25-sources.jar
slf4j-jcl-1.7.25.jar
slf4j-jcl-1.7.25-sources.jar
slf4j-jdk14-1.7.25.jar
slf4j-jdk14-1.7.25-sources.jar
slf4j-log4j12-1.7.25.jar
slf4j-log4j12-1.7.25-sources.jar
slf4j-android-1.7.25.jar
slf4j-android-1.7.25-sources.jar
jcl-over-slf4j-1.7.25.jar
jcl-over-slf4j-1.7.25-sources.jar
slf4j-ext-1.7.25.jar
slf4j-ext-1.7.25-sources.jar
jul-to-slf4j-1.7.25.jar
jul-to-slf4j-1.7.25-sources.jar
log4j-over-slf4j-1.7.25.jar
log4j-over-slf4j-1.7.25-sources.jar

Where each binding is for its respective logging framework.

The following table lists the SLF4J bindings and their corresponding frameworks.

Jar file	Logging Framework
slf4j-nop-x.x.jar	No operation, discards all loggings.
slf4j-simple-x.x.jar	Simple implementation where messages for info and higher are printed and, remaining all outputs to System.err.
slf4j-jcl-x.x.jar	Jakarta Commons Logging framework.
slf4j-jdk14-x.x.jar	Java.util.logging framework (JUL).
slf4j-log4j12-x.x.jar	Log4J frame work. In addition, you need to have **log4j.jar.**

To make SLF4J work along with slf4l-api-x.x.jar, you need to add the respective Jar file (binding) of the desired logger framework in the classpath of the project (set build path).

To switch from one framework to other, you need to replace the respective binding. If no bounding is found, it defaults to no-operation mode.

Pom.xml for SLF4J

If you are creating the maven project, open the **pom.xml** and paste the following content in it and refresh the project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>Sample</groupId>
  <artifactId>Sample</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.7.0</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.25</version>
    </dependency>
  </dependencies>
</project>
```

5. SLF4J — Referenced API

In this chapter, we will discuss the classes and methods that we will be using in the subsequent chapters of this tutorial.

Logger Interface

The logger interface of the **org.slf4j** package is the entry point of the SLF4J API. The following lists down the important methods of this interface.

S.No.	Methods and Description
1	**void debug(String msg)** This method logs a message at the DEBUG level.
2	**void error(String msg)** This method logs a message at the ERROR level.
3	**void info(String msg)** This method logs a message at the INFO level.
4	**void trace(String msg)** This method logs a message at the TRACE level.
5	**void warn(String msg)** This method logs a message at the WARN level.

LoggerFactory Class

The LoggerFactory class of the **org.slf4j** package is a utility class, which is used to generate loggers for various logging APIs such as log4j, JUL, NOP and simple logger.

S.No.	Method and Description
1	**Logger getLogger(String name)** This method accepts a string value representing a name and returns a **Logger** object with the specified name.

Profiler Class

This class belongs to the package **org.slf4j** this is used for profiling purpose and it is known as poor man's profiler. Using this, the programmer can find out the time taken to carry out prolonged tasks.

Following are the important methods of this class.

S.No.	Methods and Description
1	**void start(String name)** This method will start a new child stop watch (named) and, stops the earlier child stopwatches (or, time instruments).
2	**TimeInstrument stop()** This method will stop the recent child stopwatch and the global stopwatch and return the current Time Instrument.
3	**void setLogger(Logger logger)** This method accepts a Logger object and associates the specified logger to the current Profiler.
4	**void log()** Logs the contents of the current time instrument that is associated with a logger.
5	**void print()** Prints the contents of the current time instrument.

6. SLF4J — Hello world

In this chapter, we will see a simple basic logger program using SLF4J. Follow the steps described below to write a simple logger.

Step 1: Create an object of the slf4j.Logger interface

Since the **slf4j.Logger** is the entry point of the SLF4J API, first, you need to get/create its object.

The **getLogger()** method of the **LoggerFactory** class accepts a string value representing a name and returns a **Logger** object with the specified name.

```
Logger logger = LoggerFactory.getLogger("SampleLogger");
```

Step 2: Log the required message

The **info()** method of the **slf4j.Logger** interface accepts a string value representing the required message and logs it at the info level.

```
logger.info("Hi This is my first SLF4J program");
```

Example

Following is the program that demonstrates how to write a sample logger in Java using SLF4J.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SLF4JExample {
  public static void main(String[] args) {
    //Creating the Logger object
    Logger logger = LoggerFactory.getLogger("SampleLogger");

    //Logging the information
    logger.info("Hi This is my first SLF4J program");
  }
}
```

Output

On running the following program initially, you will get the following output instead of the desired message.

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".

SLF4J: Defaulting to no-operation (NOP) logger implementation

SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
```

Since we have not set the classpath to any binding representing a logging framework, as mentioned earlier in this tutorial, SLF4J defaulted to a no-operation implementation. So, to see the message you need to add the desired binding in the project classpath. Since we are using eclipse**,** set **build path** for respective JAR file or**,** add its dependency in the pom.xml file.

For example, if we need to use JUL (Java.util.logging framework), we need to set build path for the jar file **slf4j-jdk14-x.x.jar.** And if we want to use log4J logging framework, we need to set build path or, add dependencies for the jar files **slf4j-log4j12-x.x.jar** and **log4j.jar**.

After adding the binding representing any of the logging frameworks except **slf4j-nop-x.x.jar** to the project (classpath), you will get the following output.

```
Dec 06, 2018 5:29:44 PM SLF4JExample main

INFO: Hi Welcome to Tutorilspoint
```

7. SLF4J — Error Messages

In this chapter, we will discuss the various error messages or warning we get while working with SLF4J and the causes/ meanings of those messages.

Failed to load class "org.slf4j.impl.StaticLoggerBinder".

This is a warning which is caused when there are no SLF4J bindings provided in the classpath.

Following is the complete warning:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
```

To resolve this, you need to add either of the logging framework bindings. This is explained in the Hello world chapter of this tutorial.

Note: This occurs in versions of SLF4J which are between 1.6.0 and 1.8.0-beta2.

No SLF4J providers were found

In slf4j-1.8.0-beta2, the above warning is more clear saying "**No SLF4J providers were found**".

Following is the complete warning:

```
SLF4J: No SLF4J providers were found.
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#noProviders for further details.
```

Classpath contains SLF4J bindings targeting slf4j-api versions prior to 1.8

If you are using SLF4J 1.8 version and you have the bindings of previous versions in the classpath but not the bindings of 1.8 you will see a warning as shown below.

```
SLF4J: No SLF4J providers were found.
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#noProviders for further details.
```

```
SLF4J: Class path contains SLF4J bindings targeting slf4j-api versions prior to
1.8.

SLF4J: Ignoring binding found at
[jar:file:/C:/Users/Tutorialspoint/Desktop/Latest%20Tutorials/SLF4J%20Tutorial/
slf4j-1.7.25/slf4j-jdk14-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: See http://www.slf4j.org/codes.html#ignoredBindings for an explanation.
```

NoClassDefFoundError: org/apache/commons/logging/LogFactory

If you are working with **slf4j-jcl** and if you have only **slf4j-jcl.jar** in your classpath, you will get an exception such as the one given below.

```
Exception in thread "main" java.lang.NoClassDefFoundError:
org/apache/commons/logging/LogFactory

     at org.slf4j.impl.JCLLoggerFactory.getLogger(JCLLoggerFactory.java:77)

     at org.slf4j.LoggerFactory.getLogger(LoggerFactory.java:358)

     at SLF4JExample.main(SLF4JExample.java:8)

Caused by: java.lang.ClassNotFoundException:
org.apache.commons.logging.LogFactory

     at java.net.URLClassLoader.findClass(Unknown Source)

     at java.lang.ClassLoader.loadClass(Unknown Source)

     at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)

     at java.lang.ClassLoader.loadClass(Unknown Source)

     ... 3 more
```

To resolve this, you need to add **commons-logging.jar** to your classpath.

Detected both jcl-over-slf4j.jar AND bound slf4j-jcl.jar on the classpath..

The binding **slf4j-jcl.jar** redirects calls of the slf4j logger to JCL and the **jcl-over-slf4j.jar** redirects calls of JCL logger to slf4j. Therefore, you cannot have both in the classpath of your project. If you do so, you will get an exception such as the one given below.

```
SLF4J: Detected both jcl-over-slf4j.jar AND bound slf4j-jcl.jar on the class
path, preempting StackOverflowError.

SLF4J: See also http://www.slf4j.org/codes.html#jclDelegationLoop for more
details.

Exception in thread "main" java.lang.ExceptionInInitializerError

     at org.slf4j.impl.StaticLoggerBinder.<init>(StaticLoggerBinder.java:71)

     at org.slf4j.impl.StaticLoggerBinder.<clinit>(StaticLoggerBinder.java:42)

     at org.slf4j.LoggerFactory.bind(LoggerFactory.java:150)

     at org.slf4j.LoggerFactory.performInitialization(LoggerFactory.java:124)
```

```
        at org.slf4j.LoggerFactory.getILoggerFactory(LoggerFactory.java:412)

        at org.slf4j.LoggerFactory.getLogger(LoggerFactory.java:357)

        at SLF4JExample.main(SLF4JExample.java:8)

Caused by: java.lang.IllegalStateException: Detected both jcl-over-slf4j.jar
AND bound slf4j-jcl.jar on the class path, preempting StackOverflowError. See
also http://www.slf4j.org/codes.html#jclDelegationLoop for more details.

        at org.slf4j.impl.JCLLoggerFactory.<clinit>(JCLLoggerFactory.java:54)

        ... 7 more
```

To resolve this, delete either of the jar files.

Detected logger name mismatch

You can create a Logger object by -

- Passing the name of the logger to be created as an argument to the **getLogger()** method.
- Passing a class as an argument to this method.

If you are trying to create the logger factory object by passing a class as an argument, and if you have set the system property **slf4j.detectLoggerNameMismatch** to true, then the name of the class you pass as an argument to the **getLogger**() method and the class you use should be the same otherwise you will receive the following warning –

"Detected logger name mismatch.

Consider the following example.

```
import org.slf4j.Logger;

import org.slf4j.LoggerFactory;


public class SLF4JExample {

  public static void main(String[] args) {


    System.setProperty("slf4j.detectLoggerNameMismatch", "true");

    //Creating the Logger object

    Logger logger = LoggerFactory.getLogger(Sample.class);


    //Logging the information

    logger.info("Hi Welcome to Tutorilspoint");

  }

}
```

Here, we have set the *slf4j.detectLoggerNameMismatch* property to true. The name of the class we used is **SLF4JExample** and the class name we have passed to the *getLogger()* method is **Sample** since they both are not equal we will get the following warning.

```
SLF4J: Detected logger name mismatch. Given name: "Sample"; computed name:
"SLF4JExample".

SLF4J: See http://www.slf4j.org/codes.html#loggerNameMismatch for an
explanation

Dec 10, 2018 12:43:00 PM SLF4JExample main

INFO: Hi Welcome to Tutorilspoint
```

Note: This occurs after slf4j 1.7.9

Classpath contains multiple SLF4J bindings.

You should have only one binding in the classpath. If you have more than one binding, you will get a warning listing the bindings and the locations of them.

For suppose, if we have the bindings **slf4j-jdk14.jar** and **slf4j-nop.jar** in the classpath we will get the following warning.

```
SLF4J: Class path contains multiple SLF4J bindings.

SLF4J: Found binding in
[jar:file:/C:/Users/Tutorialspoint/Desktop/Latest%20Tutorials/SLF4J%20Tutorial/
slf4j-1.7.25/slf4j-nop-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: Found binding in
[jar:file:/C:/Users/Tutorialspoint/Desktop/Latest%20Tutorials/SLF4J%20Tutorial/
slf4j-1.7.25/slf4j-jdk14-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an
explanation.

SLF4J: Actual binding is of type [org.slf4j.helpers.NOPLoggerFactory]
```

Detected both log4j-over-slf4j.jar AND bound slf4j-log4j12.jar on the class path

To redirect the log4j logger calls to slf4j, you need to use **log4j-over-slf4j.jar** binding and if you want to redirect slf4j calls to log4j, you need to use **slf4j-log4j12.jar** binding.

Therefore, you cannot have both in the classpath. If you do, you will get the following exception.

```
SLF4J: Detected both log4j-over-slf4j.jar AND bound slf4j-log4j12.jar on the
class path, preempting StackOverflowError.

SLF4J: See also http://www.slf4j.org/codes.html#log4jDelegationLoop for more
details.
```

```
Exception in thread "main" java.lang.ExceptionInInitializerError
	at org.slf4j.impl.StaticLoggerBinder.<init>(StaticLoggerBinder.java:72)
	at org.slf4j.impl.StaticLoggerBinder.<clinit>(StaticLoggerBinder.java:45)
	at org.slf4j.LoggerFactory.bind(LoggerFactory.java:150)
	at org.slf4j.LoggerFactory.performInitialization(LoggerFactory.java:124)
	at org.slf4j.LoggerFactory.getILoggerFactory(LoggerFactory.java:412)
	at org.slf4j.LoggerFactory.getLogger(LoggerFactory.java:357)
	at org.slf4j.LoggerFactory.getLogger(LoggerFactory.java:383)
	at SLF4JExample.main(SLF4JExample.java:8)
Caused by: java.lang.IllegalStateException: Detected both log4j-over-slf4j.jar
AND bound slf4j-log4j12.jar on the class path, preempting StackOverflowError.
See also http://www.slf4j.org/codes.html#log4jDelegationLoop for more details.
```

8. SLF4J — Parameterized logging

As discussed earlier in this tutorial SLF4J provides support for parameterized log messages.

You can use parameters in the messages and pass values to them later in the same statement.

Syntax

As shown below, you need to use placeholders ({}) in the message (String) wherever you need and later you can pass value for place holder in **object** form, separating the message and value with comma.

```
Integer age;
Logger.info("At the age of {} ramu got his first job", age);
```

Example

The following example demonstrates parameterized logging (with single parameter) using SLF4J.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class PlaceHolders {
  public static void main(String[] args) {
    //Creating the Logger object
    Logger logger = LoggerFactory.getLogger(PlaceHolders.class);
    Integer age = 23;
    //Logging the information
    logger.info("At the age of {} ramu got his first job", age);

  }
}
```

Output

Upon execution, the above program generates the following output:

```
Dec 10, 2018 3:25:45 PM PlaceHolders main
```

```
INFO: At the age of 23 Ramu got his first job
```

Advantage of Parameterized Logging

In Java, if we need to print values in a statement, we will use concatenation operator as -

```
System.out.println("At the age of "+23+" ramu got his first job");
```

This involves the conversion of the integer value 23 to string and concatenation of this value to the strings surrounding it.

And if it is a logging statement, and if that particular log level of your statement is disabled then, all this calculation will be of no use.

In such circumstances, you can use parameterized logging. In this format, initially SLF4J confirms whether the logging for particular level is enabled. If so then, it replaces the placeholders in the messages with the respective values.

For example, if we have a statement as

```
Integer age;
Logger.debug("At the age of {} ramu got his first job", age);
```

Only if debugging is enabled then, SLF4J converts the age into integer and concatenates it with the strings otherwise, it does nothing. Thus incurring the cost of parameter constructions when logging level is disabled.

Two Argument Variant

You can also use two parameters in a message as -

```
logger.info("Old weight is {}. new weight is {}.", oldWeight, newWeight);
```

Example

The following example demonstrates the usage of two placeholders in parametrized logging.

```
import java.util.Scanner;


import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


public class PlaceHolders {
  public static void main(String[] args) {
    Integer oldWeight;
    Integer newWeight;

    Scanner sc = new Scanner(System.in);
    System.out.println("Enter old weight:");
    oldWeight = sc.nextInt();

    System.out.println("Enter new weight:");
    newWeight = sc.nextInt();

    //Creating the Logger object
    Logger logger = LoggerFactory.getLogger(Sample.class);

    //Logging the information
    logger.info("Old weight is {}. new weight is {}.", oldWeight, newWeight);



    //Logging the information
    logger.info("After the program weight reduced is: "+(oldWeight-newWeight));
  }
}
```

Output

Upon execution, the above program generates the following output.

```
Enter old weight:
85
Enter new weight:
74
Dec 10, 2018 4:12:31 PM PlaceHolders main
INFO: Old weight is 85. new weight is 74.
Dec 10, 2018 4:12:31 PM PlaceHolders main
INFO: After the program weight reduced is: 11
```

Multiple Argument Variant

You can also use more than two placeholders as shown in the following example:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class PlaceHolders {
  public static void main(String[] args) {

    Integer age = 24;
    String designation  = "Software Engineer";
    String company = "Infosys";

    //Creating the Logger object
    Logger logger = LoggerFactory.getLogger(Sample.class);

    //Logging the information
    logger.info("At the age of {} ramu got his first job as a {} at {}", age, 
designation, company);

  }
}
```

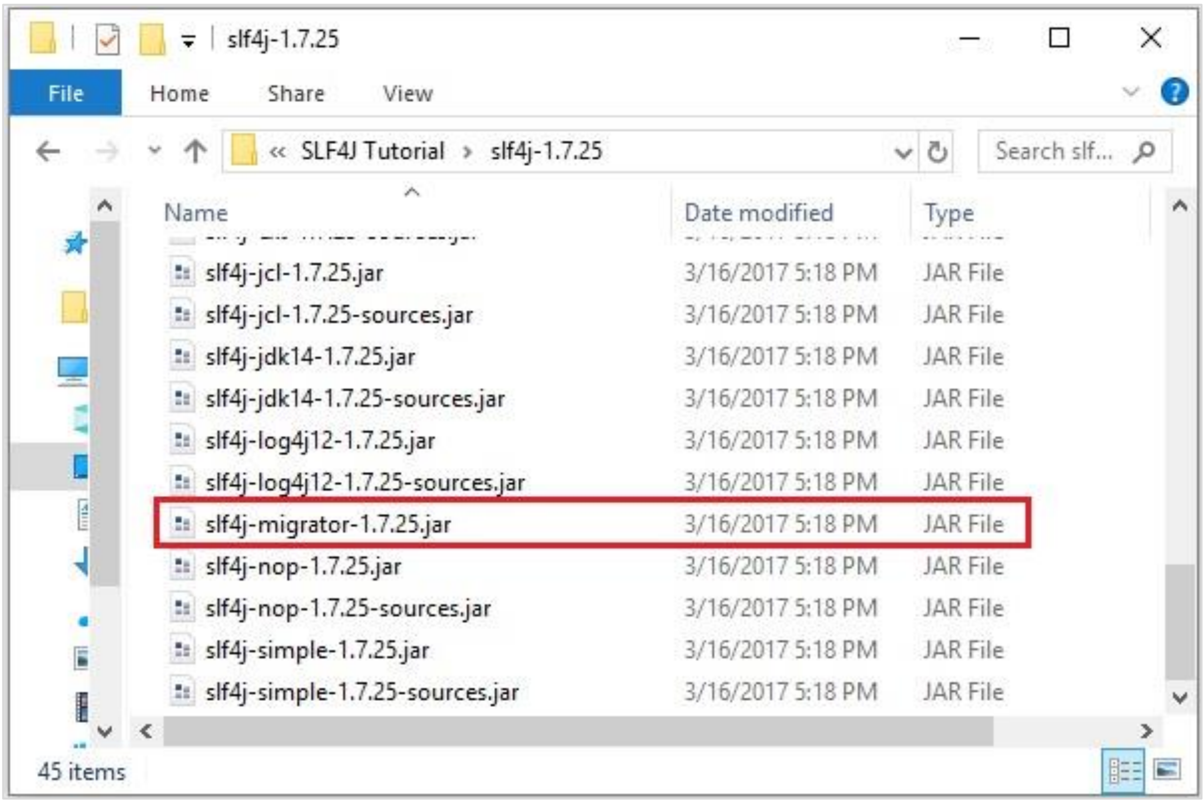
Output

Upon execution, the above program generates the following output:

```
Dec 10, 2018 4:23:52 PM PlaceHolders main
INFO: At the age of 24 ramu got his first job as a Software Engineer at Infosys
```

9. SLF4J — Migrator

If you have a project in Jakarta Commons Logging (JCL) or, log4j or, java.util.logging (JUL) and you want to convert these projects to SLF4J, you can do so using the migrator tool provided in the SLF4J distribution.



Name	Date modified	Type
slf4j-jcl-1.7.25.jar	3/16/2017 5:18 PM	JAR File
slf4j-jcl-1.7.25-sources.jar	3/16/2017 5:18 PM	JAR File
slf4j-jdk14-1.7.25.jar	3/16/2017 5:18 PM	JAR File
slf4j-jdk14-1.7.25-sources.jar	3/16/2017 5:18 PM	JAR File
slf4j-log4j12-1.7.25.jar	3/16/2017 5:18 PM	JAR File
slf4j-log4j12-1.7.25-sources.jar	3/16/2017 5:18 PM	JAR File
slf4j-migrator-1.7.25.jar	3/16/2017 5:18 PM	JAR File
slf4j-nop-1.7.25.jar	3/16/2017 5:18 PM	JAR File
slf4j-nop-1.7.25-sources.jar	3/16/2017 5:18 PM	JAR File
slf4j-simple-1.7.25.jar	3/16/2017 5:18 PM	JAR File
slf4j-simple-1.7.25-sources.jar	3/16/2017 5:18 PM	JAR File

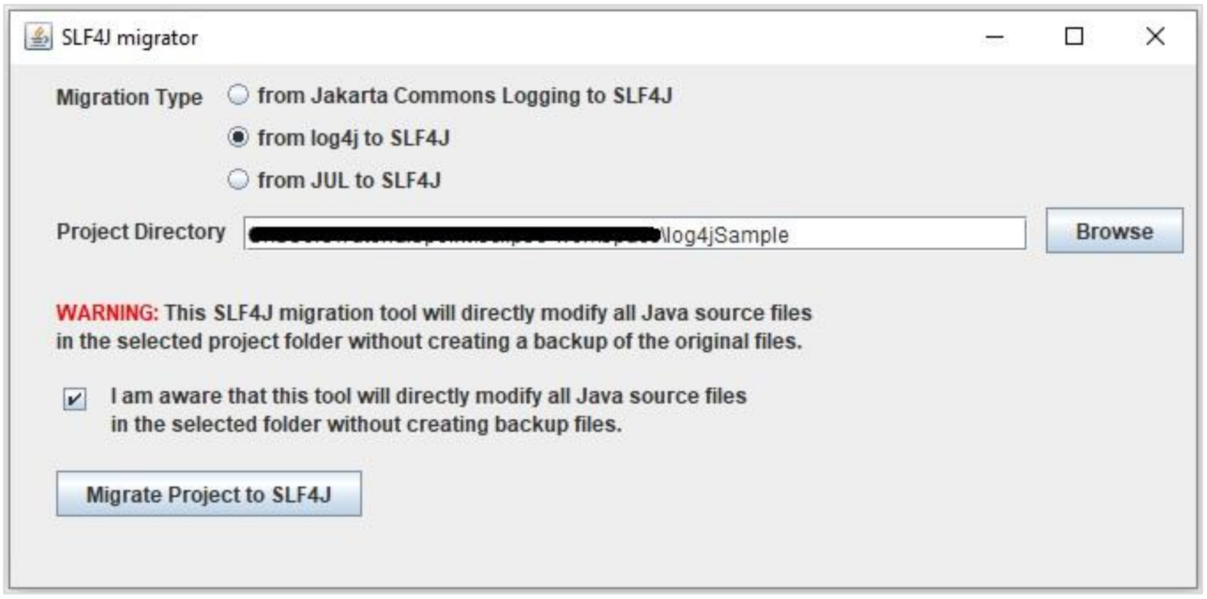
Running SLF4J Migrator

SLF4J is a simple single jar file (slf4j-migrator.jar) and you can run it using the java –jar command.

To run it, in command prompt, browse through the directory where you have this jar file and execute the following command.

```
java -jar slf4j-migrator-1.8.0-beta2.jar

Starting SLF4J Migrator
```

This starts the migrator and you can see a standalone java application as:


SLF4J migrator

Migration Type
( ) from Jakarta Commons Logging to SLF4J
(●) from log4j to SLF4J
( ) from JUL to SLF4J

Project Directory [redacted]\log4jSample

WARNING: This SLF4J migration tool will directly modify all Java source files in the selected project folder without creating a backup of the original files.

☑ I am aware that this tool will directly modify all Java source files in the selected folder without creating backup files.

As specified in the window, you need to check the type of migration you want to do and select the project directory and click on the button Migrate Project to SLF4J.

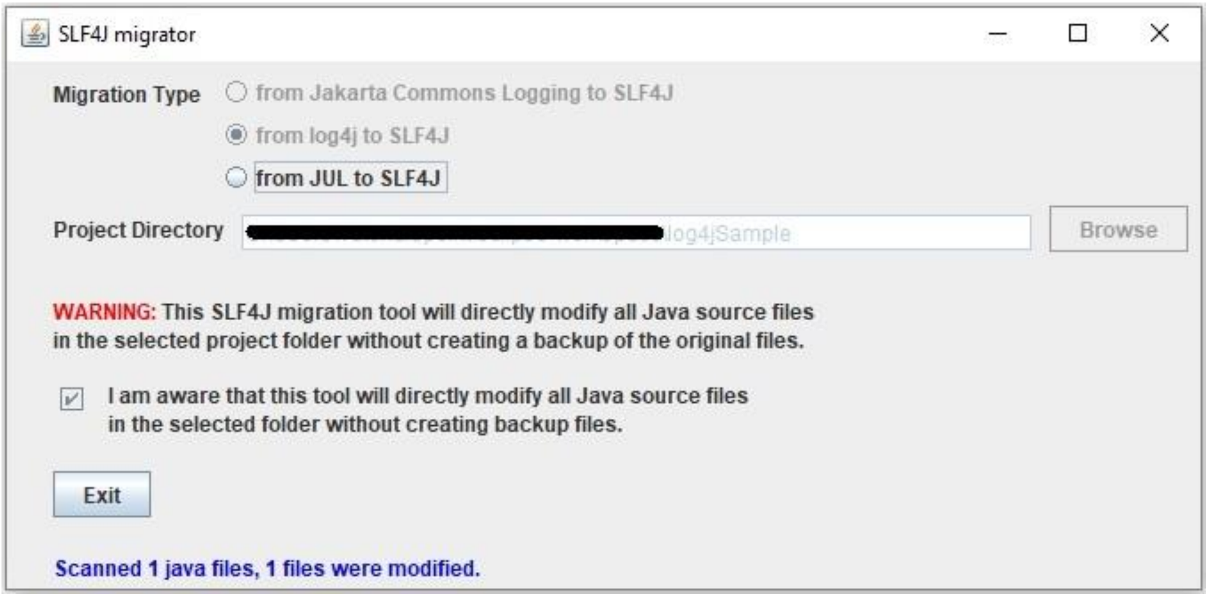
This tool goes to the source files you provide and performs simple modifications like changing the import lines and logger declarations from the current logging framework to SLF4j.

Example

For example, let us suppose we have a sample **log4j(2)** project in eclipse with a single file as follows:

```
import org.apache.log4j.Logger;


import java.io.*;

import java.sql.SQLException;

import java.util.*;


public class Sample{


   /* Get actual class name to be printed on */

   static Logger log = Logger.getLogger(Sample.class.getName());


   public static void main(String[] args)throws IOException,SQLException{

      log.debug("Hello this is a debug message");

      log.info("Hello this is an info message");

   }
}
```

To migrate the sample **log4j(2)** project to slf4j, we need to check the radio button **from log4j to slf4j** and select the directory of the project and click **Exit** to migrate.



Migration Type ( ) from Jakarta Commons Logging to SLF4J [✓ from log4j to SLF4J] ( ) from JUL to SLF4J

Project Directory [redacted]log4jSample

WARNING: This SLF4J migration tool will directly modify all Java source files in the selected project folder without creating a backup of the original files.

✓ I am aware that this tool will directly modify all Java source files in the selected folder without creating backup files.

Scanned 1 java files, 1 files were modified.

The migrator changed the above code as follows. Here if you observe the import and logger statements have been modified.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class Sample{

   static Logger log = LoggerFactory.getLogger(Sample.class.getName());

   public static void main(String[] args)throws IOException,SQLException{
      log.debug("Hello this is a debug message");
      log.info("Hello this is an info message");
   }
}
```

Since you already have **log4j.jar** in your project, you need to add **slf4j-api.jar** and **slf4j-log12.jar** files to the project to execute it.

Limitations of SLF4J Migrator

Following are the limitations of the SLF4J migrator.

- Migrator will not modify build scripts like ant, maven and, ivy you need to do it yourself.
- Migrator does not support messages other than the String type.
- Migrator does not support the FATAL level.
- While working with log4j, migrator will not migrate calls to PropertyConfigurator or DomConfigurator.

10. SLF4J — Profiling

SLF4J Distribution provides **slf4j-ext.jar** this contains APIs for the functionalities such as profiling, Extended logging, Event logging and, logging with java agent.

Profiling

Sometimes the programmer wants to measure some attributes like the use of memory, time complexity or usage of particular instructions about the programs to measure the real capability of that program. Such kind of measuring about the program is called profiling. Profiling uses dynamic program analysis to do such measuring.

SLF4J provides a class named **Profiler** in the **org.slf4j.profiler** package for profiling purpose. This is known as the poor man's profiler. Using this, the programmer can find out the time taken to carry out prolonged tasks.

Profiling Using the Profiler class

The profiler contains stopwatches and child stopwatches and we can start and stop these using the methods provided by the profiler class.

To carry on with profiling using the profiler class, follow the steps given below.

Step 1: Instantiate the profiler class

Instantiate the Profiler class by passing a String value representing the name of the profiler. When we instantiate a Profiler class, a global stopwatch will be started.

```
//Creating a profiler
Profiler profiler = new Profiler("Sample");
```

Step 2: Start a child stopwatch

When we invoke the **start()** method it will start a new child stopwatch (named) and, stops the earlier child stopwatches (or, time instruments).

Invoke the **start()** method of the **Profiler** class by passing a String value representing the name of the child stopwatch to be created.

```
//Starting a child stopwatch and stopping the previous one.
profiler.start("Task 1");
obj.demoMethod1();
```

After creating these stopwatches, you can perform your tasks or, invoke those methods, which run your tasks.

Step 3: Start another child stopwatch (if you wish to)

If you need, create another stopwatch using the **start()** method and perform the required tasks. If you do so, it will start a new stop watch and stops the previous one (i.e. task 1).

```
//Starting another child stopwatch and stopping the previous one.
profiler.start("Task 2");
obj.demoMethod2();
```

Step 4: Stop the watches

When we invoke the **stop()** method, it will stop the recent child stopwatch and the global stopwatch and returns the current Time Instrument.

```
//Stopping the current child stopwatch and the global stopwatch.
TimeInstrument tm = profiler.stop();
```

Step 5: Print the contents of the time instrument.

Print the contents of the current time instrument using the **print()** method.

```
 //printing the contents of the time instrument
 tm.print();
```

Example

The following example demonstrates the profiling using Profiler class of SLF4J. Here we have taken two sample tasks, printing the sum of squares of the numbers from 1 to 10000, printing the sum of the numbers from 1 to 10000. We are trying to get the time taken for these two tasks.

```
import org.slf4j.profiler.Profiler;
import org.slf4j.profiler.TimeInstrument;

public class ProfilerExample {

     public void demoMethod1(){
       double sum = 0;
       for(int i=0; i< 1000; i++){
          sum = sum+(Math.pow(i, 2));
        }
       System.out.println("Sum of squares of the numbers from 1 to 10000: "+sum);
    }
```

```
      public void demoMethod2(){
             int sum = 0;
        for(int i=0; i< 10000; i++){
            sum = sum+i;
         }
        System.out.println("Sum of the numbers from 1 to 10000: "+sum);
     }

     public static void main(String[] args) {
       ProfilerExample obj = new ProfilerExample();

       //Creating a profiler
       Profiler profiler = new Profiler("Sample");

       //Starting a child stop watch and stopping the previous one.
       profiler.start("Task 1");
       obj.demoMethod1();

       //Starting another child stop watch and stopping the previous one.
       profiler.start("Task 2");
       obj.demoMethod2();

       //Stopping the current child watch and the global watch.
       TimeInstrument tm = profiler.stop();

       //printing the contents of the time instrument
       tm.print();
     }
}
```

Output:

Upon execution, the above program generates the following output:

```
Sum of squares of the numbers from 1 to 10000: 3.328335E8
Sum of the numbers from 1 to 10000: 49995000
+ Profiler [BASIC]
```

```
|-- elapsed time                        [Task 1]  2291.827 microseconds.
|-- elapsed time                        [Task 2]   225.802 microseconds.
|-- Total                                [BASIC]  3221.598 microseconds.
```

Logging the Profiler Information

Instead of printing the result of a profiler to log this information, you need to:

- Create a logger using the **LoggerFactory** class.
- Create a profiler by instantiating the Profiler class.
- Associate the logger to profiler by passing the logger object created to the **setLogger()** method of the **Profiler** class.
- Finally, instead of printing log the information of the profiler using the **log()** method.

Example

In the following example, unlike the previous one (instead of printing), we are trying to log the contents of the time instrument.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.profiler.Profiler;
import org.slf4j.profiler.TimeInstrument;


public class ProfilerExample_logger {


     public void demoMethod1(){
            double sum = 0;
       for(int i=0; i< 1000; i++){
           sum = sum+(Math.pow(i, 2));
        }
       System.out.println("Sum of squares of the numbers from 1 to 10000: "+sum);
    }


     public void demoMethod2(){
            int sum = 0;
       for(int i=0; i< 10000; i++){
            sum = sum+i;
          }
        System.out.println("Sum of the numbers from 1 to 10000: "+sum);
    }

    public static void main(String[] args) {

      ProfilerExample_logger obj = new ProfilerExample_logger();

      //Creating a logger
      Logger logger = LoggerFactory.getLogger(ProfilerExample_logger.class);



      //Creating a profiler
      Profiler profiler = new Profiler("Sample");

      //Adding logger to the profiler
      profiler.setLogger(logger);

      //Starting a child stop watch and stopping the previous one.
      profiler.start("Task 1");
      obj.demoMethod1();

      //Starting another child stop watch and stopping the previous one.
      profiler.start("Task 2");
      obj.demoMethod2();

      //Stopping the current child watch and the global watch.
      TimeInstrument tm = profiler.stop();

      //Logging the contents of the time instrument
      tm.log();
    }
}
```

Output

Upon execution, the above program generates the following output.

```
Sum of squares of the numbers from 1 to 10000: 3.328335E8

Sum of the numbers from 1 to 10000: 49995000
```