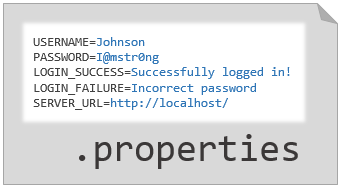
|  |
| --- |
| PROPERTIES AND LOGGING |

In a typical enterprise application, we will have many configuration information that needs to be maintained. This configuration information can be related to database configuration, logging configuration, exception messages, etc. Instead of hardcoding this configuration its always better to keep them separate so that they can be modified easily without recompiling the Java classes. One way to do this is to use a properties file.

**What is a properties file?**

**Properties**file is a text file used to store any kind of textual information in the form of key-value pairs. A typical properties file looks like this-



**Properties files are usually used for**

* storing configuration related data
* storing information which can be changed frequently as unlike a Java class file, recompilation is not needed if the information is changed from a properties file
* containing locale-specific data. It internationalizes Java code by making the code locale-dependent.

There are different ways in which you can read a Properties file, like by using the Properties class in java.util package, Java ResourseBundle, or we can use any other third-party library like Apache Commons.

Next, let us see how we can use Apache Commons to read a Properties file.

Let us now see how we can read a .properties file. One way to read it is by using a third-party library called Apache Commons.

The Apache Commons Configuration is a third-party library that provides classes that can be used to perform certain operations on a ".properties" file. It contains a PropertiesConfiguration class which is used to read a properties file.

**org.apache.commons.configuration.PropertiesConfiguration Class:**

The org.apache.commons.configuration.PropertiesConfiguration class is a class that loads the values from a properties file.

**Configuring and using PropertiesConfiguration class:**

Discussed below is the configuration and the usage of the PropertiesConfiguration class.

**Configuration:**

To configure the PropertiesConfiguration class to load your properties file, we have to use **the org.apache.commons.configuration2.builder.fluent.Configurations.Configurations()**class to build it as shown below,

1. Configurations configurations = new Configurations();
2. PropertiesConfiguration config = configurations.properties("configuration.properties");

As you can see, we build an object of PropertiesConfiguration class for the configuration.properties. This object can be used for accessing data from the specified properties file.

**Note:** If you want to pass the properties name directly as shown above, keep your properties file in the src folder or any of its subfolders. If kept anywhere else, you will have to provide the exact path to the properties file.

**Methods:**

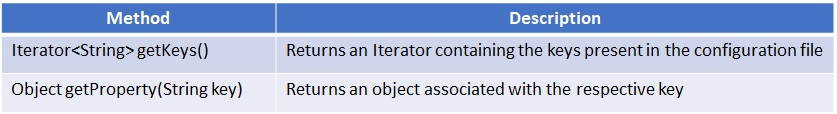
The PropertiesConfiguration class defines a few methods like:

**Iterator<String> getKeys():**This will provide all the keys present in the properties file.

1. Iterator<String> keys = config.getKeys();

**Object getProperty(String key):**This provides the corresponding value present in the properties file pertaining to the key passed. If the key value passed is not present in the properties file, it return **null**.

1. System.out.println("Value for server.port=" + config.getProperty("server.port"));



Here, we will mostly deal with reading the properties from a .properties file.

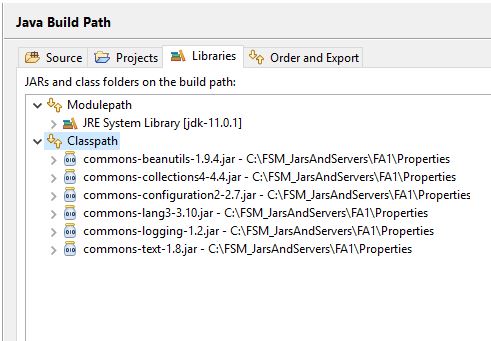
Note: Java also provides a class called the Properties class in the java.util package which also provides methods for operations on a .properties file. You can read about it [here](https://docs.oracle.com/javase/7/docs/api/java/util/Properties.html).

Let us see a demo on how to read the properties from a configuration.properties file.

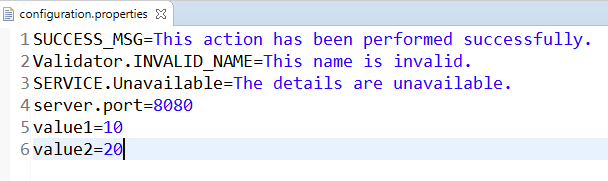
**Step 1:** Download the demo from [here](https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_0130010099188613121924_shared/web-hosted/assets/PropertiesDemo1641375999670.zip). Extract the file and import the project in your workspace.

**Step 2:** Download the jars required for the apache-commons configuration from [here](https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_0130010099188613121924_shared/web-hosted/assets/PropertiesJars.zip). Extract the file and add all the jars to the Project Build Path.

After you have added the jars, the build path should look something like this:



**Step 3:** Observe the **configuration.properties** file under src/resources :



**Step 4:** Open the AppEnvironment.java file and observe the code:

1. package application;
2. import java.util.Iterator;
3. import org.apache.commons.configuration.PropertiesConfiguration;
4. import org.apache.commons.configuration2.builder.fluent.Configurations;
5. public class AppEnvironment {
6. public static void main(String[] args) throws Exception {
7. PropertiesConfiguration config = null;
8. Configurations configurations = new Configurations();
9. config = configurations.properties("configuration.properties");
10. *// Fetching all the key-value pairs from the properties file*
11. System.out.println("---- Fetching all the key value pairs ----");
12. Iterator<String> keys = config.getKeys();
13. while (keys.hasNext()) {
14. String key = keys.next();
15. System.out.println(key + " = " + config.getProperty(key));
16. }
17. System.out.println("---- Fetching a single value by passing the key ----");
18. System.out.println("Value for server.port=" + config.getProperty("server.port"));
19. *// Calculating sum from two number values stored in .properties file*
20. *// Since the return type of getProperty() is an object, you have to type-cast it*
21. *// accordingly*
22. Integer value1 = Integer.parseInt((String) config.getProperty("value1"));
23. Integer value2 = Integer.parseInt((String) config.getProperty("value2"));
24. Integer sum = value1 + value2;
25. System.out.println("Sum=" + sum);
26. }
27. }

**Step 5:** Specifying the properties file name:

1. PropertiesConfiguration config = null;
2. Configurations configurations = new Configurations();
3. config = configurations.properties("configuration.properties");

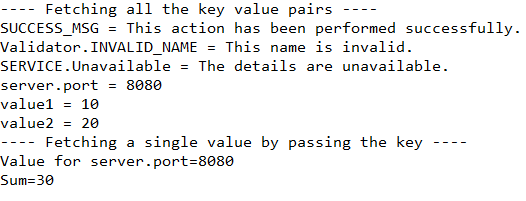
**Step 6:** Fetching all the key-value pairs from the properties file:

1. Iterator<String> keys = config.getKeys();
2. while (keys.hasNext()) {
3. String key = keys.next();
4. System.out.println(key + " = " + config.getProperty(key));
5. }

**Step 7:** Fetching a single value by passing the key:

1. System.out.println("Value for server.port=" + config.getProperty("server.port"));

**Step 8:** After you have executed the program, you should get an output like this in your console:



During the execution of an Enterprise application, several events are generated. These events could be triggered at different points while the user interacts with the application. They could be simple informational events or events that need to be managed.

Such events can help solve issues, understand performance and statistics and improve the system. In large applications, it becomes essential to keep track of all such events. And keeping a log of all the associated information helps in quick problem diagnosis, debugging and maintenance.

|  |  |
| --- | --- |
| Logging is the process of tracking the execution of the program where, |  |
| * Any event that is to the interest of the programmer or administrator can be logged. * Relevant messages can be recorded in the event of exceptions and errors.   The logs can be analysed by the administrators later. |

For the ease of the developer, there are many logging APIs available. Based on the requirement, the developer can choose any of them. Few popular APIs for logging are,

* JDK Logging API
* Apache Log4J
* Apache Commons Logging API

In this course, we will be using the combination of Apache Commons Logging API and Apache Log4J.

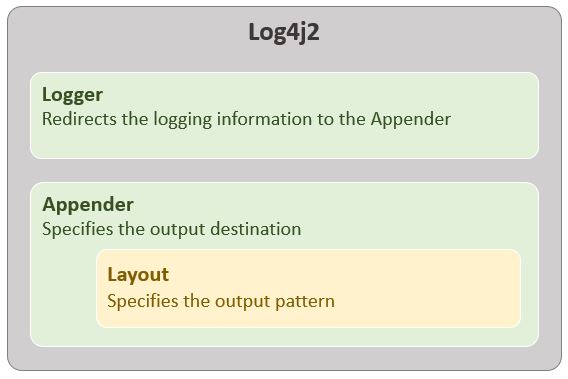
Apache Commons Logging is a logging utility that is Java-based and provides the implementation of loggers and also supports the different Logging APIs.

Log4j is written in Java and is open source. It provides mechanisms to directly log information to a variety of destinations such as database, file, console, etc. It also logs information in various formatting styles such as HTML, XML, Log, etc.

Logging using Log4J can be configured in many ways. In this course, we will be using the properties file.

**Note:** We are using an updated version of Log4J, i.e., Log4j2.

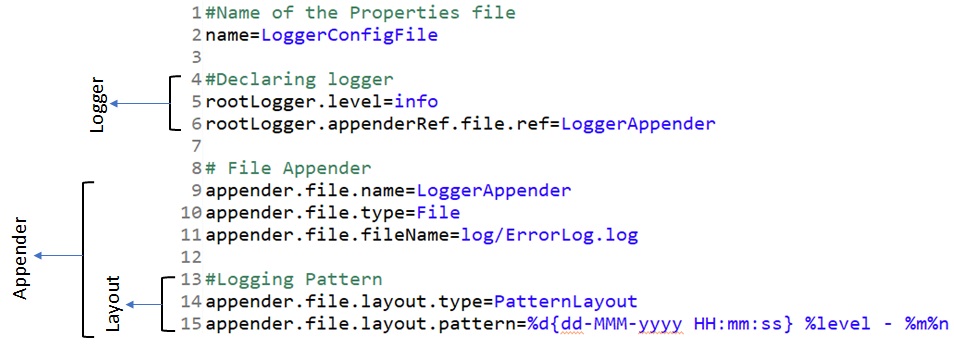
For configuring Log4j2, we must first know what all it comprises of. Log4j2 has 3 components,



* **Appender**– Specifies where the logging output should be stored. It can be the console, an external file, any database, etc.
* **Layout**– Specifies in what format should the information be logged. It can be a simple layout which logs the level followed by the message, or we can specify a pattern for it to follow, etc.
* **Logger** – Accepts all the information to be logged and redirects it to an appropriate Appender.

Configuring each of these components is important to use the Log4J2 API. Now let us look into how to configure these using the Properties File.

The properties file is a normal file in which the data is stored in a key-value pair as you saw in the previous module. When we use it to configure Log4J2, it looks as shown below,



As you can see, the properties file has three parts in total. Lines 5-6 are configuring the Logger component, lines 9-15 are configuring the Appender component and lines 14-15 are configuring the Layout component. But what does each line mean?

In the *Logger* component, we are first declaring the Logging Level we want to use, i.e., in this case, *INFO*. Logging Levels are like the indicators of the severity of each log (We will discuss the Logging Levels on the next page). In the next line, we are configuring the Logger to use the *Appender*component having the name *LoggerAppender*.

In the *Appender*component, we are first declaring the name of the Appender, in this case, LoggerAppender. This links the Logger output to the exact Appender component (This connects line 6 to the appender we want to use). Next, we declare the destination where the logged data should be placed. It can be the console, an external file or an HTML layout also. In this case, we are using an external file to log all the data. The next line declares the name and location the external file is located. It can be an absolute path, or as shown above a relative path. Here we are declaring that the external file with the name *ErrorLog.log* will be present in the *log*folder in the project. (Note: If the external file is not found in the mentioned location, then the logger will create a new file in the mentioned location)

Last, but not the least, in the *Layout*component, we first declare the type of layout we want to use (whether it is SimpleLayout, HTMLLayout, PatternLayout), in this case, a PatternLayout. A PatternLayout is a layout where we configure the pattern of the logging information on our own. Next line we are declaring the pattern to be used. The pattern we want the information to be logged is,   
**<<$timestamp>> <<$LoggingLevel>> <<$Message>>**  
where,

* timestamp is given by, *%d{dd-MMM-yyyy HH:mm:ss}*
* Logging level is given by, *%level*
* Message is given by, *%m*

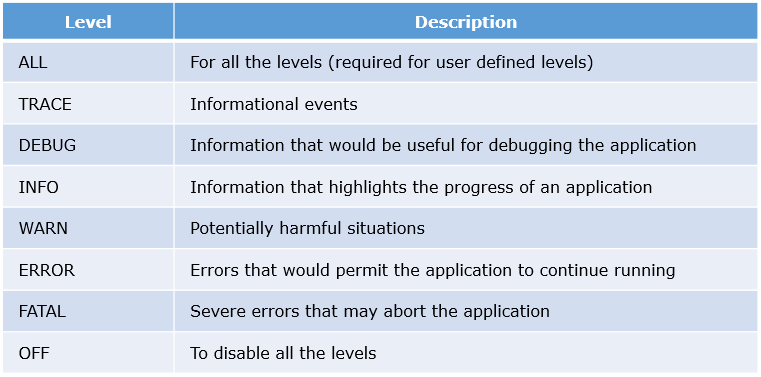
The configurations used in the above properties file is one of the basic configurations. You can change it to suit your needs and project requirements.

**Note:**

* The name of the properties file must be log4j2.properties for the logger to detect it.
* The log4j2.properties file should be placed directly in the src folder or any of its subfolders. If placed anywhere else, it will not be detected automatically by the Logger.

Logging Levels as told in the previous page are used to differentiate between the severity between different logs. Consider two logged information, one which provides the details of some user using your application and other which provides details of a component of your application failing. Which among these two will require your immediate attention? Which of these can be left for a later stage? All this can be determined by using Logging levels while logging the information.

Logging Levels are provided by the class, **org.apache.log4j.Level**. Each level has its unique usage.



Logging Levels also have a special functionality, i.e., when we choose any level, all the levels below that chosen level can be used to log the information. For example, if we choose the logging level as *WARN*, then during implementation, along with *WARN*, we can also make use of the levels *ERROR*and *FATAL*. The complete coverage of logging levels is given below,



The theory is all well and good. But how to log in the Java application?

Till now we have seen the configuration of the log4j2 in our application. But how do we use that configuration? For this, we will use the elements of the Apache Commons Logging API. Apache Commons Logging and Apache Log4J2 APIs are both external to the Java library, hence we have to add few additional JAR files to our Java Project. Those JARs can be downloaded from [here](https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_0130086339299164161314_shared/web-hosted/assets/ApacheLogging1641376389165.zip). It contains,

* commons-logging: This has all the required files of Apache Commons Logging
* log4j-api: This has all the required files of the Log4J2 API
* log4j-core: This has the core files and logger implementation of Log4J2
* log4j-jcl: This acts as a connector bridge between Apache Commons Logging and Log4J2.

**Note:** Please add these JAR files to the Java Build Path in your project.

To log information, we first need to get a logger object. This logger object is obtained by invoking *getLog()* method of the *org.apache.commons.logging.LogFactory* class. This method takes the details of the class in which we want to log as a parameter and gives us an object of *org.apache.commons.logging.Log* interface.

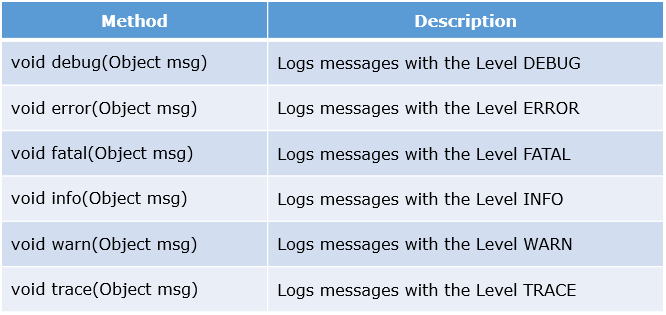
1. Log logger = LogFactory.getLog(Clazz clazz);

For example, if we want to log in a class, *UserAuthentication*, the code will be,

1. import org.apache.commons.logging.Log;
2. import org.apache.commons.logging.LogFactory;
3. public class UserAuthentication {
4. Log logger = LogFactory.getLog(UserAuthentication.class);
5. }

In the above example, we are creating a log object as an attribute of the class. You can also create the log object as a local variable in a method.

Once done, we can use this log object to log our data. For logging, the log object provides us with many methods to use. These methods directly correspond with the Logging Level we want to use,



So, if we want to log anything with logging level *INFO*, the statement will be,

1. logger.info("Information to be logged");

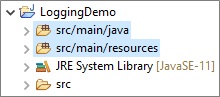
**Note:** When logging exceptions, if you want to log the exception message and the exception stack trace, each logging method will change slightly. They will become,

1. void <logginglevel>(Object exceptionMessage, Throwable exceptionObject);

**Problem Statement:**

LoggersCorp wants to record the details of every user who will log in to their systems. If the user enters the wrong credentials, an appropriate error message should be shown as well as the event should be logged to an external file. And if the user logs in successfully, then the timestamp and the user name should be logged.

**Step 1:** Create a new Java project with the name *LoggingDemo*. While creating, delete the src folder that gets created automatically, and create two new source folders with the names, *src/main/java* and *src/main/resources* as shown below,



**Step 2:** Add the required JAR files to the Classpath in the Java Build Path of the project. Click [here](https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_0130086339299164161314_shared/web-hosted/assets/ApacheLogging1641376524579.zip) to download to get the JAR files.

**Step 3:** Create a new file with the name*log4j2.properties* in *src/main/resources* source folder and type the below code in it,

1. *#Name of the Properties file*
2. name=LoggerConfigFile
3. *#Declaring logger*
4. rootLogger.level=info
5. rootLogger.appenderRef.file.ref=LoggerAppender
6. *# File Appender*
7. appender.file.name=LoggerAppender
8. appender.file.type=File
9. appender.file.fileName=log/ErrorLog.log
10. *#Logging Pattern*
11. appender.file.layout.type=PatternLayout
12. appender.file.layout.pattern=%d{dd-MMM-yyyy HH:mm:ss} %level - %m%n

**Step 4:** Create a new package, *com.infy*in the *src/main/java* source folder, and create a new Java class *UserAuthentication.java* in it. Type the below code in the new java class,

1. package com.infy;
2. import org.apache.commons.logging.Log;
3. import org.apache.commons.logging.LogFactory;
4. public class UserAuthentication {
6. Log logger = LogFactory.getLog(UserAuthentication.class);
8. public void loginUser(String username, String password) throws Exception{
9. try {
10. if(username.isBlank() || password.isBlank()) {
11. throw new Exception("Service.INVALID\_CREDENTIALS");
12. }
13. logger.info(username+" logged in successfully");
14. } catch(Exception exception) {
15. logger.error(exception.getMessage(), exception);
16. throw exception;
17. }
18. }
19. }

**Step 5:** Now to test the functionality of the logger, we will create a new package, com.infy.*userinterface*, and in it, we will create a *Tester.java* class file.

1. package com.infy.userinterface;
2. import com.infy.UserAuthentication;
3. public class Tester {
4. public static void main(String[] args) {
5. try {
6. UserAuthentication authentication = new UserAuthentication();
7. authentication.loginUser("Jon", "Asdf@123");
8. System.out.println("User has logged in successfully");
9. } catch(Exception exception) {
10. System.out.println(exception.getMessage());
11. }
12. }
13. }

**Step 6:** Execute the *Tester*class and observe the output in the console and the logger file.

Output in console,

1. User has logged in successfully

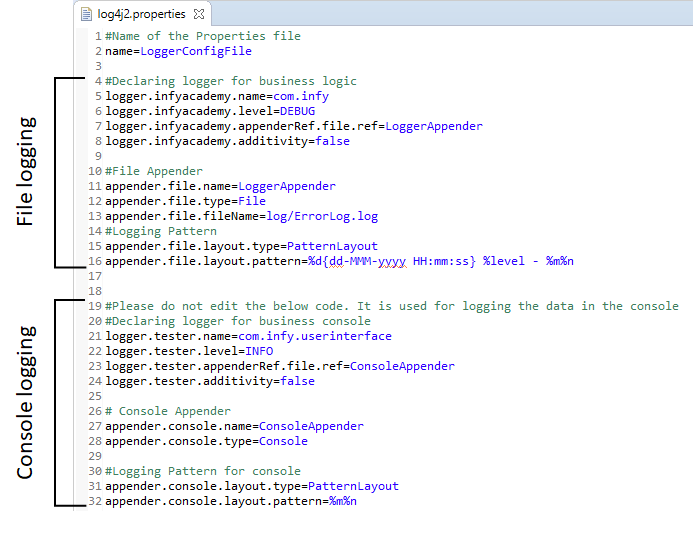
Output in the Log file,

1. 10-Apr-2020 10:28:48 INFO - Jon logged in successfully

**Note:** To get the Log file, please refresh your project once.

Moving forward, all information should be logged only using the logger object be it in a log file or on the console since the usage of **System.out** is a big violation of efficient coding practices. We will discuss how to use the logger object to log information on the console along with the log file.

The **log4j2.properties** file should be modified as shown below by adding Console logging



From the above image, we can notice that there are two configurations one for logging in a file and another for console logging. For simplification, our exercises and assignments are divided into two parts, **business logic**, and the **business console**. All information from the business logic part of the application will be logged on a log file whereas any information from the business console will be logged on the console. The name of the loggers, **com.infy,**and **com.infy.userinterface**will be used by the logger object to decide where to log.

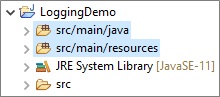
Since logging to a file has been discussed already, we will focus on the Console logging part. On line 23 in the above image, the console logger is using the **ConsoleAppender**component. In this appender component, we are declaring a name for this appender as **ConsoleAppender** on line 27 and declaring the destination as **Console** on line 28.

In the **layout component** of the ConsoleAppender, first, the layout is selected as PatternLayout and the pattern is set to ***%m%n*.**This pattern logs the same message without any additional information

**Problem Statement:**

LoggersCorp wants to record the details of every user who will log in to their systems. If the user enters the wrong credentials, an appropriate error message should be shown as well as the event should be logged to an external file. And if the user logs in successfully, then the timestamp and the username should be logged.

**Step 1:**Create a new Java project with the name LoggingDemo. While creating, delete the src folder that gets created automatically, and create two new source folders with the names, *src/main/java* and *src/main/resources*as shown below,



**Step 2:** Add the required *JAR* files to the classpath in the*Java Build Path* of the project. Click [here](https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_0130086339299164161314_shared/web-hosted/assets/ApacheLogging1641377445805.zip) to get the JAR files.

**Step 3:** Create a new file with the name*log4j2.properties* in *src/main/resources*source folder and type the below code in it,

1. *#Name of the Properties file*
2. name=LoggerConfigFile
3. *#Declaring logger for business logic*
4. logger.loggingdemo.name=com.infy
5. logger.loggingdemo.level=DEBUG
6. logger.loggingdemo.appenderRef.file.ref=LoggerAppender
7. logger.loggingdemo.additivity=false
8. *#File Appender*
9. appender.file.name=LoggerAppender
10. appender.file.type=File
11. appender.file.fileName=log/ErrorLog.log
12. *#Logging Pattern*
13. appender.file.layout.type=PatternLayout
14. appender.file.layout.pattern=%d{dd-MMM-yyyy HH:mm:ss} %level - %m%n
15. *#Please do not edit the below code. It is used for logging the data in the console*
16. *#Declaring logger for business console*
17. logger.tester.name=com.infy.userinterface
18. logger.tester.level=INFO
19. logger.tester.appenderRef.file.ref=ConsoleAppender
20. logger.tester.additivity=false
21. *# Console Appender*
22. appender.console.name=ConsoleAppender
23. appender.console.type=Console

**Step 4:** Create a new package, service in the *src/main/java* source folder, and create a new Java class *UserAuthentication.java*in it. Copy & paste the below code in the new java file with the name UserAuthentication.java,

1. package com.infy;
2. import org.apache.commons.logging.Log;
3. import org.apache.commons.logging.LogFactory;
4. public class UserAuthentication {
6. Log logger = LogFactory.getLog(UserAuthentication.class);
8. public void loginUser(String username, String password) throws Exception{
9. try {
10. if(username.isBlank() || password.isBlank()) {
11. throw new Exception("Service.INVALID\_CREDENTIALS");
12. }
13. logger.info(username+" logged in successfully");
14. } catch(Exception exception) {
15. logger.error(exception.getMessage(), exception);
16. throw exception;
17. }
18. }
19. }

**Step 5:**Now to test the functionality of the logger, we will create a new package userinterface, and in it, we will create a Tester.java class file.

1. package com.infy.userinterface;
2. import org.apache.commons.logging.Log;
3. import org.apache.commons.logging.LogFactory;
4. import com.infy.UserAuthentication;
5. public class Tester {
6. private static final Log LOGGER = LogFactory.getLog(Tester.class);
7. public static void main(String[] args) {
8. try {
9. UserAuthentication authentication = new UserAuthentication();
10. authentication.loginUser("Jon", "Asdf@123");
11. LOGGER .info("User has logged in successfully");
12. } catch(Exception exception) {
13. LOGGER .info(exception.getMessage());
14. }
15. }
16. }

Now observe the above code, there is no System.out.println() is used.

**Step 6:**Execute the Tester class and observe the output in the console and the logger file.

Output in console,

1. User has logged in successfully

Output in the Log file,

1. 10-Apr-2020 10:28:48 INFO - Jon logged in successfully

Note: To get the Log file, please refresh your project once.

Best Practice Logging:

As we know Logging is one of the important aspects of application development, its an efficient coding practice to log as much of the information as possible to keep track of the application health. At the same time, it is also important to check whether we are logging the redundant information which might cause problems while debugging.

In simple words, assume that in a Java application, there are 3 different classes **Tester**, **Service**, and **DAO**. Tester class has the main method which invokes the method of Service which in turn invokes another method of DAO, and in all these three classes, methods might throw any exceptions that are caught, logged, and re-thrown.

Assume that an exception occurred in the DAO class, here is the sequence of events that will follow,

* The exception thrown in DAO will be caught, logged, and re-thrown to the Service class. (Exception logged once)
* The exception thrown by the DAO will be caught in the Service class, logged, and re-thrown to the main method in Tester class (Exception logged twice)
* The exception thrown by the Service will be caught by the main method in the Tester class and logged, (Exception logged thrice)

Even though the exception information is the same, it is logged into the **error.log** file three times which is redundant and useless. Hence, it is always a good practice to log only once by avoiding the same information logged multiple times.

Here are a few solutions to avoid logging multiple times,

* Any exceptions handled in individual classes (**Service & DAO**) should be rethrown and propagated to the calling environment (**main** method in **Tester**) and logging should be done only in the calling environment (main method).
* Log only exceptions that are thrown by its own class (use if condition to check as below) but rethrow exceptions of all the other classes from the catch block as shown below.

1. *//Good practice*
2. public class Service{
3. public Integer SomeMethod(){
4. try{
5. *//Code to invoke method of DataProvider*
6. *//Code to throw service class exception,*
7. *//All exceptions thrown from service class will contain 'Service' in the exception message*
8. }catch(Exception e){
9. *//Good practice to check only Service class exceptions to log in its own class*
10. if(e.getMessage().contains("Service"){ *//Checking if exception is thrown by service class*
11. LogFactor.getLogger(this.getClass()).error(e.getMessage(),e);
12. }
13. throw e; *//Rethrowing exception*
14. }
15. }
16. }

**Problem Statement:**

You are given an implemented code that allows a user to buy pizzas. Log the required data and eliminate hard-coding where ever specified.

In the project given, the Validator, PizzaService and the PizzaShoppie classes have been implemented for you. You have to modify them as specified below. The configuration.properties and log4j2.properties file have to be implemented by you.

**Validator Class:**

Log all the exceptions thrown from the methods of this class under the level ERROR.

**PizzaService Class:**

* Log all the exceptions thrown from the Service class methods.
* In the purchasePizzas() method, log the name of the Foodie and the total cost before the return statement under the level INFO.

**PizzaShoppie Class:**

* In this class, avoid any hard coding of data. (Make use of configuration.properties file)
* When any exception is thrown from the service method, an irrelevant exception message gets printed. Make sure the output is according to the data given below,
  + If the walletBalance is invalid, the output should be, **Your wallet is empty or has invalid amount.**
  + If the pizzasOrdered is null or empty, the output should be, **The pizzas have not yet been ordered. Please add any pizza to order.**
  + If the walletBalance is not sufficient, the output should be, **You, unfortunately, do not have enough money. Please come back when you have it.**
* **Note:**None of the outputs should be hardcoded in the Java classes.

**configuration.properties:**

* This is a file you will have to completely implement.
* Based on the exceptions thrown and the relevant messages shown above, create key-value pairs.
* Similarly, create a key-value pair for the success message also.

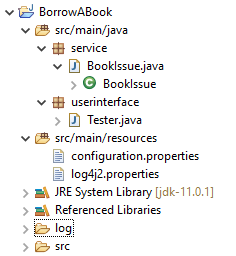
**log4j2.properties:**

* This is a file you will have to completely implement.
* Make sure that the logging levels up to DEBUG can be logged.
* All the data should get logged in an external file having the name, LogOutput.log.
* Make sure to log only the date and the message.

**Note**: Check the project using SonarLint to maintain the coding standards.

Assignment:

A Reader wants to issue few books from a book store. We have already set up a project for you as such:



All you need to do is code the BookIssue.java class.

configuration.properties file:

1. BOOK\_AVAILABLE=You can successfully borrow the book:
2. BOOK\_UNAVAILABLE=Sorry! The book is currently unavailable!
3. INVALID\_BOOKNAME=This book is invalid!
4. B1\_NAME=The Book Thief
5. B1\_VALUE=5
6. B2\_NAME=The Kite Runner
7. B2\_VALUE=0

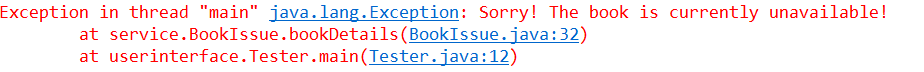
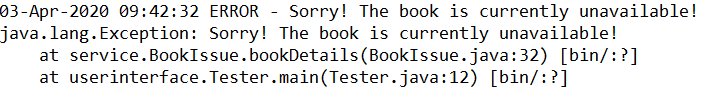
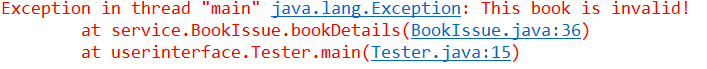
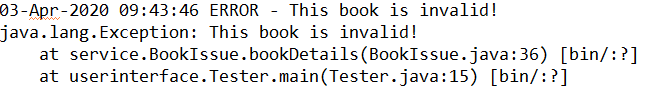
log4j2.properties file:

1. name=LoggerConfigFile
2. rootLogger.level=info
3. rootLogger.appenderRef.file.ref=LoggerAppender
4. appender.file.name=LoggerAppender
5. appender.file.type=File
6. appender.file.fileName=log/ErrorLog.log
7. appender.file.layout.type=PatternLayout
8. appender.file.layout.pattern=%d{dd-MMM-yyyy HH:mm:ss} %level - %m%n

Tester.java class:

1. package userinterface;
2. import service.BookIssue;
3. public class Tester {
4. public static void main(String[] args) throws Exception {
5. *// Uncomment after implementing BookIssue class*
6. */\**
7. *\* BookIssue b1 = new BookIssue(); b1.bookDetails("The Book Thief");*
8. *\*/*
9. */\**
10. *\* BookIssue b2= new BookIssue(); b2.bookDetails("The Kite Runner");*
11. *\*/*
12. */\**
13. *\* BookIssue b3 = new BookIssue(); b3.bookDetails("Greek Goddesses");*
14. *\*/*
15. }
16. }

BookIssue.java class:

* Provide the necessary configurations for Logger and Properties object to fetch the corresponding values from the configuration.properties file.
* Check the value for the correcting book by fetching its value from the configuration.properties file.
* If the value is greater than 0, display and log the appropriate success message by fetching it from the configuration.properties file.  
  Console output:  
    
  Log file:  
  
* If the value is 0, throw a new exception with a message "BOOK\_UNAVAILABLE".  
  Console output:  
    
  Log file:  
  
* For any other value, throw a new exception with a message "INVALID\_BOOKNAME".  
  Console output:  
    
  Log file:  
  
* Log and re-throw any exception that is caught.
* **Note**: Check the project using SonarLint to maintain the coding standards.