INSTRUCTION:
Read the description below and do the necessary.

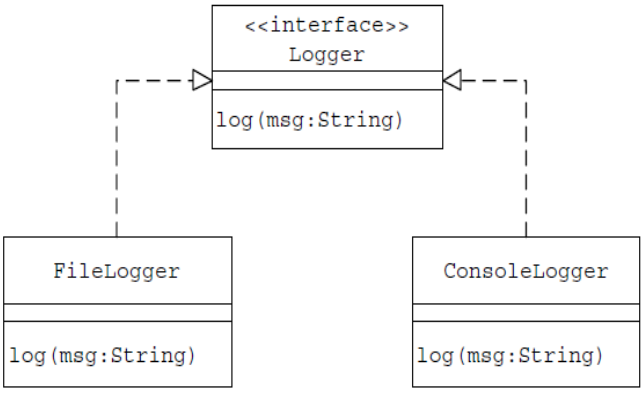
Case Study 1 (Part 1 - Introduction)

Logging messages is one of the most commonly performed tasks in software applications. Logging appropriate messages at appropriate stages can be extremely useful for debugging and monitoring applications. Because the message logging functionality could be needed by many different clients, it would be a good idea to keep the actual message logging functionality inside a common utility class so that client objects do not have to repeat these details.

A Java interface *Logger* that declares the interface to be used by the client objects to log messages can be defined. In general, an incoming message could be logged to different output media, in different formats. Different concrete implementer classes of the *Logger* interface can handle these differences in implementation. Two such implementers are described in the table below.

Implementer	*Functionality*
`FileLogger`	Stores incoming messages to a log file
`ConsoleLogger`	Displays incoming messages on the screen

The resulting message logging utility class hierarchy is depicted next.



Each of the Logger implementer classes offers the respective functionality inside the *log* method declared by the *Logger* interface.

Consider an application object *LoggerTest* that intends to use the services provided by the *Logger* implementers. Suppose that the overall application message logging configuration can be specified using the *logger.properties* property file.

Sample logger.properties file contents

```
FileLogging=OFF
```

Depending on the value of the *FileLogging* property, an appropriate *Logger* implementer needs to be used to log messages. For example, if the *FileLogging* property is set to ON, messages are to be logged to a file and hence a *FileLogger* object can be used to log messages. Similarly, if the *FileLogging* property is set to OFF, messages are to be displayed on the console and hence a *ConsoleLogger* object can be used.

To log messages, an application object such as the *LoggerTest* needs to:

- Identify an appropriate *Logger* implementer by reading the *FileLogging* property value from the *logger.properties* file

  Hints: You can write a method as below to return a boolean value which can be used to determine whether to use FileLogger or ConsoleLogger object.

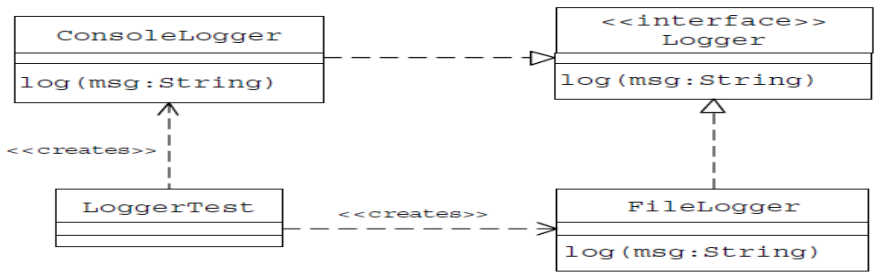
```
public boolean isFileLoggingEnabled() {
  Properties p = new Properties();
  try {
    p.load(ClassLoader.getSystemResourceAsStream(
      "Logger.properties"));
    String fileLoggingValue =
      p.getProperty("FileLogging");
    if (fileLoggingValue.equalsIgnoreCase("ON") == true)
      return true;
    else
      return false;
  } catch (IOException e) {
    return false;
  }
}
```

- Instantiate the *Logger* implementer and invoke the *log* method by passing the message text to be logged as an argument

This requires every application object to:

- Be aware of the existence and the functionality of all implementers of the *Logger* interface and their subclasses
- Provide the implementation required to select and instantiate an appropriate *Logger* implementer

The following figure depicts this design.



1. Create a Netbeans project named *yourMatricNumber-Lab4a-Part1* and implement the message logging utility (*Logger, ConsoleLogger and FileLogger*) in Java.

2. Create *logger.properties* file in the src directory of your Netbeans project with the following content:

   FileLogging=ON

3. Create the test class (LoggerTest.java) to test your implementation by logging the message "hello, how are you?" together with the timestamp. If the FileLogging is set to ON in the properties file, the message will be logged together with the timestamp in the *log.txt* file. If it is set to OFF, the message will be displayed on the console together with the timestamp.

   A sample output is as below:

   ```
   2018-09-21 15:45:47.478: hello, how are you?
   ```

Case Study 1 (Part 2 – Factory Method)

Apply the *Factory Method* design pattern to encapsulate the necessary implementation for selecting and instantiating an appropriate *Logger* implementer inside a separate *getLogger* method in a separate class *LoggerFactory.* As part of its implementation, the factory method *getLogger* checks the *logger.properties* property file to see if file logging is enabled and decides which *Logger* implementation is to be instantiated. The selected *Logger* implementer instance is returned as an object of type *Logger*.

1. With the *factory method* in place, client objects do not need to deal with the intricacies involved in selecting and instantiating an appropriate *Logger* implementer. Client objects do not need to know the existence of different implementers of the *Logger* interface and their associated functionality. Draw a UML class diagram to show your design for the *Message Logging Utility* system after you apply the *Factory Method* design pattern.

   Whenever a client object such as the *LoggerTest* needs to log a message, it can:
   - Invoke the factory method *getLogger*. When the factory method returns, the client object does not have to know the exact *Logger* subtype that is instantiated as long as the returned object is of the *Logger* type.
   - Invoke the *log* method exposed by the *Logger* interface on the returned object.

   In this example application design, the creator class *LoggerFactory* is designed as a concrete class with default implementation for the factory method *getLogger*. There can be variations in the way in which the class selection criterion is implemented. Such variations can be implemented by overriding the *getLogger* method in *LoggerFactory* subclasses.

2. Create a Netbeans project named *yourMatricNumber-Lab4a-Part2.* Copy the needed files from your Part 1 of this case study and implement the *LoggerFactory* class, and a *LoggerTestFactoryMethod* class to test your current implementation of the *Message Logging Utility* system.