11

SINGLETON

This pattern was previously described in GoF95.

DESCRIPTION

The Singleton pattern is an easy to understand design pattern. Sometimes, there may be a need to have one and only one instance of a given class during the lifetime of an application. This may be due to necessity or, more often, due to the fact that only a single instance of the class is sufficient. For example, we may need a single database connection object in an application. The Singleton pattern is useful in such cases because it ensures that there exists one and only one instance of a particular object ever. Further, it suggests that client objects should be able to access the single instance in a consistent manner.

WHO SHOULD BE RESPONSIBLE?

Having an instance of the class in a global variable seems like an easy way to maintain the single instance. All client objects can access this instance in a consistent manner through this global variable. But this does not prevent clients from creating other instances of the class. For this approach to be successful, all of the client objects have to be responsible for controlling the number of instances of the class. This widely distributed responsibility is not desirable because a client should be free from any class creation process details. The responsibility for making sure that there is only one instance of the class should belong to the class itself, leaving client objects free from having to handle these details.

A class that maintains its single instance nature by itself is referred to as a *Singleton* class.

EXAMPLE

Let us continue to work on the message logging utility example we have designed during the Factory Method pattern discussion in the previous chapter. One of the implementers of the `Logger` interface, the `FileLogger` class, logs incoming messages to the file `log.txt`. Having a singleton is helpful when there is only one physical instance of what the object represents. This is true in case of the

`FileLogger` because there is only one physical log file. In an application, when different client objects try to log messages to the file, there could potentially be multiple instances of the `FileLogger` class in use by each of the client objects. This could lead to different issues due to the concurrent access to the same file by different objects.

One of the solutions is to maintain an instance of the `FileLogger` class in a global variable within the application. This instance can be accessed in a consistent manner by all clients providing them with a single, global point of access to it. However this does not solve the problem fully.

- It does not prevent clients from creating new instances of the `FileLogger` class.
- It does not prevent multiple threads within the same client from executing the `log` method.

Another solution is to apply the monitor concept and declare the `log(String)` method as synchronized.

The monitor concept, discussed under Basic Patterns, ensures that no two threads are allowed to access the same object at the same time.

This does prevent multiple threads from entering the same method for execution, but does not prevent the client objects from creating multiple instances of the `FileLogger` class.

In addition to declaring the `log` method as synchronized, what is needed is a way to ensure that there exists one and only one instance of the `FileLogger` class during the lifetime of an application. This needs to be done in such a way that the client objects do not have to monitor the creation process or keep track of the number of `FileLogger` instances that exist.

To accomplish this using the Singleton pattern, the following changes can be made to the `FileLogger` class to make it a singleton class (Figure 11.1 and Listing 11.1).

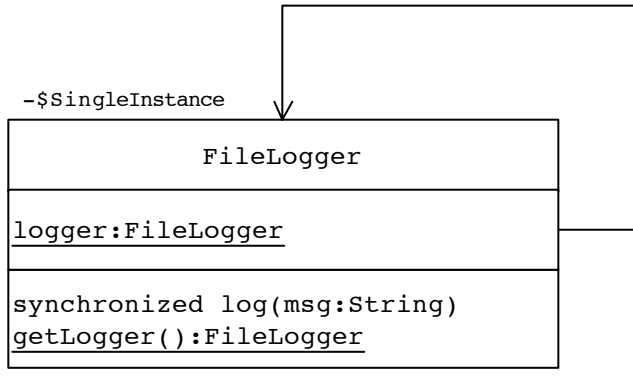

Figure 11.1 `FileLogger` Class as a Singleton

Listing 11.1 Singleton `FileLogger` Class

```
public class FileLogger implements Logger {
  private static FileLogger logger;
  //Prevent clients from using the constructor
  private FileLogger() {
  }
  public static FileLogger getFileLogger() {
    if (logger == null) {
      logger = new FileLogger();
    }
    return logger;
  }
  public synchronized void log(String msg) {
    FileUtil futil = new FileUtil();
    futil.writeToFile("log.txt",msg, true, true);
  }
}
```

Make the Constructor Private

Making the constructor private prevents client objects from creating `FileLogger` objects by invoking its constructor. At the same time, other methods inside `FileLogger` will have access to the private constructor.

Static Public Interface to Access an Instance

Provide a public interface, in the form of a static method `getInstance`, for clients to be able to get access to an instance of the `FileLogger` class. This public method must be static for a client to be able to access this method without having to instantiate the class.

Inside the `getInstance` method, create and return an instance of the `FileLogger` class by accessing its private constructor. This is done *only during the first invocation* of the `getInstance` method. Every subsequent call to the `getInstance` method returns the *same* `FileLogger` instance that is created during the first invocation. A new instance of the class is *not* created again.

A design like this ensures that there exists only one instance of the `FileLogger` class and that no two threads are allowed to execute the `log` method at the same time. This solves the problems the earlier design approach has posed.

In the case of clients expecting to use the singleton `FileLogger` object (the `LoggerTest` in this case), nothing really changes in the way they interact with the singleton `FileLogger` object.

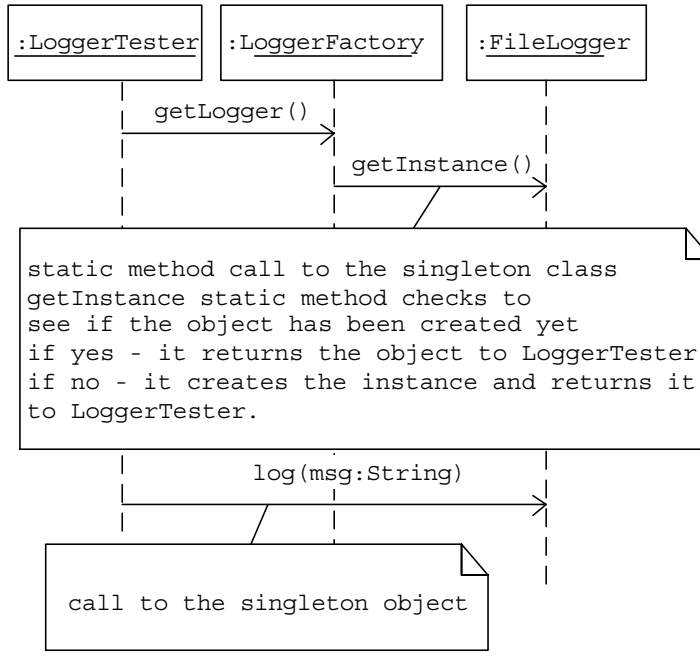

Figure 11.2 Message Flow When a Client Accesses the Singleton `FileLogger` and Invokes Its Method to Log a Message

```
//client code
public class LoggerTest{
  public static void main(String[] args){
     LoggerFactory factory=new LoggerFactory();
     //factory method call
     Logger logger=factory.getLogger();
     logger.log("A Message to Log");
  }
}
```

Figure 11.2 shows the message flow when the client `LoggerTest` accesses the singleton `FileLogger`.

The only change is for those client objects that attempt to create an instance of the `FileLogger` class (the `LoggerFactory` in this case). Instead of invoking the constructor method, they will have to use the public method `getInstance` to get an instance of the `FileLogger` class as in Listing 11.2.

Listing 11.2 `LoggerFactory` Class: Revised

```
public class LoggerFactory {
  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;
    }
  }
  public Logger getLogger() {
    if (isFileLoggingEnabled()) {
      return FileLogger.getFileLogger();
    } else {
      return new ConsoleLogger();
    }
  }
}
```

PRACTICE QUESTIONS

1. Besides the approach adopted in the example above, there can be different ways to ensure the singleton nature of an object. Think of other ways of accomplishing it.
2. Design and implement a database connection class as singleton.