

Homework 1

CMSC 433
Programming Language Technologies and Paradigms
Fall 2002

Due September 24, 2002

Introduction

In this project, you will create a simple infrastructure for logging messages in your programs. The project has three goals:

- Create a simple debugging infrastructure for your own use.
- Get some experience with Java interfaces and the ideas of abstraction and code reuse.
- Create a simple server for later expansion.

The logging system will be built in a number of steps, outlined below. We will be building a simplification and variation of the Java 1.4 `java.util.logging` package. You may wish to refer to the generated documentation¹ for all of the classes that you will be using and writing in this assignment. All provided Java files for this assignment can be found at <http://www.cs.umd.edu/class/fall2002/cmsc433-0201/hw1/java/>. You can also get this assignment in PDF and postscript.

1 Logging events

A *log* collects and performs computations on notable events in a program, like method entry and exit, thread creation, or whatever the programmer finds of interest for a particular application. Before we can collect events, we have to define what they are. This part of the homework defines a class for noteworthy (loggable) events.

¹<http://www.cs.umd.edu/class/fall2002/cmsc433-0201/hw1/docs/>

1.1 The LogRecord class

Loggable events are implemented as objects of the class `LogRecord`. Essentially a `LogRecord` notes an event, stored as a string, along with the time the event took place and a number to “uniqueify” the event. The public methods of this class you must implement are shown below:

```
public class LogRecord {
    public LogRecord(String event);
    public String getEvent();
    public java.util.Date getTimestamp();
    public String toString();
    public void format(java.io.PrintWriter out);
    public static LogRecord fromFormat(java.io.BufferedReader in);
}
```

Documentation that describes this class is found at <http://www.cs.umd.edu/class/fall2002/cmsc433-0201/hw1/docs/LogRecord.html>. Some notes:

1. The event ID should be unique to all `LogRecords`, so you should use a global counter to ensure this.
2. The String returned by `toString` should have the following format:
date / #id / event
That is, output the `LogRecord` fields in order of timestamp, id, and event, and put a `/` in between each, and a `#` before the id. This results in output like the following:

```
Mon Aug 26 13:51:50 EDT 2002 / #2 / Exiting: HeapSortAlgorithm.downheap
```

3. The `format` and `fromFormat` methods are going to be used in part ?? to communicate `LogRecords` between a client and server. Here is the way we suggest you implement them:
 - For `format`, you will output the fields of the `LogRecord` one at a time, e.g. event ID followed by timestamp followed by event. Each one will be separated by an unambiguous *field separator* to allow them to be easily read in by `fromFormat`.
 - The field separator for all fields in the `LogRecord` will be the newline (so newlines are not allowed in ID or event strings)
 - For the purpose of formatting a `java.util.Date` object, use a String representation of the Date as value of type long that represents the number of milliseconds since the beginning of Unix time, January 1, 1970, 00:00:00 GMT. See `Date.getTime()` and the wrapper class `Long` for more on how to do this.

2 An Abstract Log

Now that we have defined a loggable event, we must create a log for collecting these events.

2.1 The Log interface

A log is a collection of a given size for storing and operating on events. We define a log using an interface, `Log`, so that we may implement logs in different ways, e.g. as an array in the program, as a remote server or database, etc. The `Log` interface is shown below:

```
public interface Log {
    public void add(LogRecord record);
    public LogRecord[] getAll(long windowMS);
    public void setFilter(String pattern);
}
```

The `add` method adds a `LogRecord` to the log. The `getAll` method returns all of the `LogRecords` stored in the log that were generated within (that is, whose timestamps fall within) the last `windowMS` milliseconds. Finally, the `setFilter` method sets the *inclusion filter* for the log as the given regular expression. In particular, a `LogRecord` will not be added to the `Log` unless its *event* portion matches the given regular expression. At the moment, we do not include inclusion filters for timestamps and/or event counters.

On the web we have provided an annotated version of `Log.java` with accompanying documentation. It describes all of its methods as above; feel free to use it in your assignment.

2.2 The `Logger.java` class

Given an implementation of `Log` and the class `LogRecord`, we can put together a simple API for use in our programs. In design pattern terminology, this is a *Facade* pattern. We implement this API in the `Logger` class, whose methods are shown below:

```
public class Logger {
    public Logger(Log log);
    public Log getLog();
    public void printAll(long windowMS);
    public void addMethodEntry(Object instance, String method);
    public void addMethodExit(Object instance, String method);
}
```

On the web, we have provided an implementation of `Logger.java` for your use. In addition, we have created a sample file that makes use of the `Logger.java` API, called `HeapSortAlgorithm.java`. This is an implementation of heap sort, with inserted invocations to the logger `addMethodEntry` and `addMethodExit`

routines. Running this program will result in sorting an array, and printing out all of the logged events at the conclusion, resulting in output like the following:

```
Mon Aug 26 13:51:50 EDT 2002 / #0 / Entering: HeapSortAlgorithm.sort
Mon Aug 26 13:51:50 EDT 2002 / #1 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #2 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #3 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #4 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #5 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #6 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #7 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #8 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #9 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #10 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #11 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #12 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #13 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #14 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #15 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #16 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #17 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #18 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #19 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #20 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #21 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #22 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #23 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #24 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #25 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #26 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #27 / Entering: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #28 / Exiting: HeapSortAlgorithm.downheap
Mon Aug 26 13:51:50 EDT 2002 / #29 / Exiting: HeapSortAlgorithm.sort
```

To use `HeapSortAlgorithm.java`, you will need to actually implement the `Log` interface, and change the first line of the `main` method to be something like:

```
log = new Logger(new LocalLog(100));
```

where `LocalLog` is an implementation of `Log`, described next.

3 A Local Log: the `LocalLog` class

The `LocalLog` class is a local implementation of the `Log` interface, and has the following skeleton:

```
public class LocalLog implements Log {
```

```

public LocalLog(int logSize);
public void add(LogRecord record);
public LogRecord[] getAll(long windowMS);
public void setFilter(String strPat);
public String toString();
}

```

The middle three methods implement the `Log` interface, while the first is the sole constructor for the class, and the last is a convenience routine for printing the contents of the log.

You should implement `LocalLog` to keep track of records by storing them in some aggregation datastructure, such as an array, or an implementation of the `java.util.Collection` interface. Some notes:

1. So that the log does not grow without bound, you must fix its size; this size is set by the `logSize` parameter to the constructor. If an attempt is made to add a record to a log that is full, the oldest record should be discarded before the new record is added.
2. Choose your underlying datastructure so that you can easily implement `getAll` and `add`. That is, what datastructure allows you to easily discard the oldest record when the log is full? What datastructure allows you to easily determine and collect what records are of a certain age?

4 A Remote Log

For the final part of the assignment, you are to implement a *distributed* version of the `Log` interface. In particular, implement two classes `RemoteLogClient` and `RemoteLogServer`, where the former class implements the `Log` interface, but does so by communicating with the latter class, which is running in another process, possibly on a remote machine. It is in this process that the log records are stored and the commands are actually carried out.

4.1 The RemoteLogServer class

The skeleton of the `RemoteLogServer` class as follows:

```

public class RemoteLogServer {
    private Log log;
    RemoteLogServer(int port, int logSize);
    public static void main(String[] args);
}

```

The constructor for `RemoteLogServer` takes as its arguments a `port` on which to listen for TCP connections, and the size `logSize` of the log it is keeping. When the constructor is called, it does two things. First, it instantiates its private `Log` instance variable with a `LocalLog` having the specified size. This is the data to

which it administers the commands it receives. Second, the RemoteLogServer opens a socket on `localhost` at port number `port` to listen for TCP connections.

RemoteLogServer instances are single threaded, completely processing one command at a time as follows:

1. Accept a connection from RemoteLogClient on `port`.
2. Read the command text from the connection, followed by a newline. The text should be one of the following valid commands: `add`, `getAll`, or `setFilter`.
3. Read the arguments to the command, as follows:
 - if the command was `add`, read in a formatted LogRecord, using the `LogRecord.fromFormat` method.
 - if the command was `getAll`, read the text `windowSize n`, where *n* is a non-negative integer, followed by a newline.
 - if the command was `setFilter`, read some string *s* followed by a newline. This is the regular expression to filter for the log.
4. Process the command. This should be done by *forwarding* the command to the local `Log` instance. For example, if the command `setFilter` was received with a parameter `Entering.*`, then the RemoteLogServer was invoke the method `log.setFilter("Entering.*")`.
5. Return a response if the command was `getAll`. In this case all of the records that match the criteria in the `getAll`, as returned by `log.getAll`, are sent back to the client using the `LogRecord.format` method on each record.

Finally, the `public static main()` method for your server should get the port number to listen on from the command line (i.e., start the server with `java -Dport=portnum myLoggingServer`). The port number can be extracted in your server with the `Integer.getInteger()` method. For testing purposes, so everyone uses different ports, test your server using port `x33yy`, where *x* is your section number and *yy* is your account number. You could set this as the default port in case none is passed on the command line.

You will want to familiarize yourself with the `java.net.ServerSocket` interface for implementing this class.

4.2 The RemoteLogClient class

Here is the skeleton for the RemoteLogClient class:

```
public class RemoteLogClient implements Log {
    RemoteLogClient (java.net.InetAddress addr, int port);
    public void add(LogRecord record);
    public LogRecord[] getAll(long windowMS);
}
```

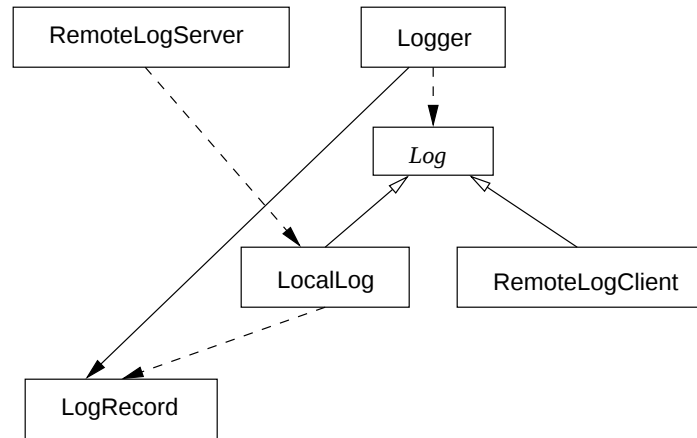


Figure 1: Class/Interface relationship for Logging package

```

    public void setFilter(String pattern);
}

```

The constructor for **RemoteLogClient** takes as its arguments a `java.net.InetAddress` `addr` as the machine to which to connect, and a `port` to use for TCP connections. These correspond to the machine name on which the **RemoteLogServer** is running and the port on which it is listening for connections. These arguments are simply stored in the **RemoteLogClient** when the constructor is called.

When one of the remaining methods is called, **RemoteLogClient** will connect to the **RemoteLogServer**, send a command, retrieve the result (if any), and close the connection. Connections only remain open for the duration of a single command. The format of commands is as described for the **RemoteLogServer** class above.

For the `getAll` method, the retrieved result will be a list of formatted **LogRecords**. You should therefore invoke the `LogRecord.fromFormat` method to turn these into **LogRecord** instances, until no further data is available.

Finally, you should allow the client to get the host and port number of the server to connect to from the command line. That is, you should be able to start your program with arguments `-Dhost=hostname -Dport=portnum`. Again, you can use `Integer.getInteger()` for this.

You will want to familiarize yourself with the `java.net.Socket` class for implementing the client.

Final Notes

The relationships among the classes you will be writing is shown in Figure ?? . The **Log** interface (shown in italics) is implemented by the classes **LocalLog**

and `RemoteLogClient`, as visualized by the hollow arrows. The `LocalLog` class stores instances of `LogRecord`, as visualized by a dotted arrow, while the `Logger` interface creates `LogRecords`, visualized by the unbroken arrow, to store in a `Log`. Finally, the `RemoteLogServer` stores an instance of a `LocalLog` for actually managing the `LogRecords` sent to it from `RemoteLogClient`'s. This diagram does not depict the socket-based interaction between `RemoteLogClient` and `RemoteLogServer`.

It is worth considering at this point what you have learned. What was the benefit of making `Log` an interface as opposed to a class? How might you extend this infrastructure to be more useful for debugging? One obvious example would be to add more event types to the `Logger` interface, outside of method entry and method exit. In what ways could the design you have implemented here be improved? In particular, what changes would make it easier to use? Or make it more efficient?

We will be considering the answers to these questions during the course.