Logging and Debugging

Lecture 26

Motivation

- Ever had one of these to deal with?
  - JUnit red bar
  - a java.lang.NullPointerException exception
  - any unexpected and wrong behavior
- What do you do?
  - Stare at the code until you figure it out
  - Make random changes and try again
  - Ask someone for help
  - Insert many System.out.println()'s
- Problems with the last approach
  - Cluttered code ends up in deployment
  - If problems re-emerge, re-add the println's?
    "If the trace is useful now, it will be useful later"
- Better approaches, indicative of experience:
  - Use a real logging facility to save tracing information
  - Use a debugger to interactively inspect execution

Logging

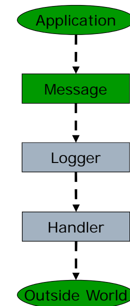
Computer Science and Engineering ■ The Ohio State University

- General framework for recording (during execution):
  - System information
  - Error messages
  - Fine-grain tracing output
- See java.util.logging
- Common in enterprise-scale, industrial-strength applications; uncommon in small programs
  - "Programming in the large"
- Flexibility and Customizability
  - Support for many output devices and formats
  - Dynamic control over output (no recompilation)

Taxonomy of java.util.logging

Computer Science and Engineering ■ The Ohio State University

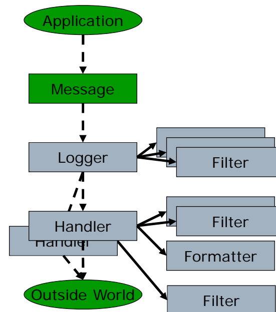
- Message
  - A string and a level of importance
- Logger
  - Client-side view of logging functionality
- Handler
  - Performs output
  - Different classes for sending to different destinations:
    - ConsoleHandler, FileHandler, SocketHandler



Extended Taxonomy

Computer Science and Engineering ■ The Ohio State University

- Logger can have multiple Handlers
  - Or none (more later)
- Filters
  - Optional for Loggers and Handlers
  - Fine control for squelching messages
  - Control usually done through *levels*
- Formatters
  - What output looks like
  - SimpleFormatter, XMLFormatter
- Defaults (no Filters, SimpleFormatter) usually sufficient



Message Levels

Computer Science and Engineering ■ The Ohio State University

- Logger discards messages *below* a certain level
  ```
  void setLevel(Level newLevel);
  ```
  - Default configuration shows INFO and higher:
  ```
  myLogger.setLevel(Level.INFO);
  ```
  - Handlers have similar controls
- 7 Levels, which are totally ordered:
  - For an end-user (ie suitable for general consumption)
    - SEVERE
    - WARNING
    - INFO
  - For a sys. admin (ie technical system information)
    - CONFIG
  - For a developer (ie can assume familiarity with code)
    - FINE
    - FINER
    - FINEST

Usage Guidelines

- SEVERE: significant or complete loss of some function
  `"Power lost - running on backup"`
  - Failure of application
  - Absence of a configuration file that completely debilitates the application (there is no good fall back)
- WARNING: problem adversely affecting operations
  `"Database connection lost, retrying..."`
- INFO: event within normal operation
  `"Startup complete"`
- FINE: significant events explaining flow/state of system
  `"Loading graphics package"`
  - Object creation
- FINER: major flow-of-control points in execution
  `"Building pie chart"`
  - Method entry/exit, or throwing exception
- FINEST: low-level debug tracing
  `"Starting bubble sort: value = " + size`
  - Intraprocedural tracing

Logger Creation

- Each Logger instance has a String name
- Created through a static factory, getLogger
  - Guarantees only one instance per name is created

    ```
    static Logger getLogger(String Name);
    ```
  - Can be cached in a field, or called in each method

    ```
    class Student {
      private static final Logger logger =
       Logger.getLogger(Student.class.getName());
      . . .
    }
    ```
- Usual practice: 1 Logger / class and package
  - Named following fully-qualified class name
    - Eg "edu.osu.cse.421.Student"

Logger Methods

- Basic method for adding a message

  ```
  void log(Level level, String msg);
  ```
  - Example

    ```
    logger.log(Level.FINEST, "Found target at
       position " + i);
    ```
- Convenience methods for each level
  - severe, warning, info, config, fine, finer, finest
  - Example

    ```
    logger.info("Configuration complete");
    ```
- Convenience methods for some events
  - entering, exiting, throwing
  - Associated log message has level FINER
  - Two string parameters: class name, method name
  - Example

    ```
    logger.entering("Student", "getValue");
    logger.entering(getClass().getName(),
       "getValue");
    ```

Example Code

```
package edu.osu.cse.421;

class Student {
  private static final Logger logger =
    Logger.getLogger(Student.class.getName());

  public boolean myMethod(int p1, Object p2) {
    logger.entering(getClass().getName(), "myMethod");
    logger.log(Level.FINER, "First argument: " + p1);
    logger.log(Level.FINER, "Second argument: " + p2);

    //Method body

    logger.exiting(getClass().getName(), "myMethod");
    logger.log(Level.FINER, "Returning: " + result);
    return result;
  }
}
```

Bad Practice: Logger.global

- Logger provides a convenience static field global
  - A globally visible logger
  - Does not need to be explicitly constructed
  - Simplifies quick and easy logging
- It might be tempting to
  - replace: System.out.println(s);
  - with: Logger.global.info(s);
- But benefit over println is marginal
  - Fine-grain control of output not possible

Performance Consideration

- Entering/Exiting methods overloaded

  ```
  void entering (String, String, Object[]);
  ```
  - Used to display value of parameters (and possibly this object too)
- Concern: Stringifying these objects can be expensive
- Solution: Short-circuit check whether message level is too fine to matter anyway

  ```
  boolean isLoggable(Level level)
  ```
  - Returns true if and only if level messages would be passed on by logger
  - Handler might still filter them out of course

Example Code


```
package edu.osu.cse.421;
class Student {
  private static final Logger logger =
    Logger.getLogger(Student.class.getName());

  public boolean myMethod(int p1, Object p2) {
    if (logger.isLoggable(Level.FINER)) {
      logger.entering(getClass().getName(), "myMethod",
                      new Object[]{Integer.valueOf(p1), p2});
    }

    // Method body

    if (logger.isLoggable(Level.FINER)) {
       logger.exiting(getClass().getName(), "myMethod",
                      Boolean.valueOf(result));
    }
    return result;
  }
}
```

Handlers

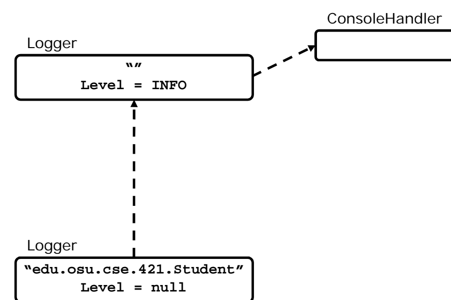
- Recall:
  - Handlers do the work of publishing messages to a device/destination
  - One Logger can have multiple Handlers
- Predefined Handlers in java.util.logging:
  - ConsoleHandler, FileHandler, StreamHandler, SocketHandler
- Default configuration uses ConsoleHandler
  - Output goes to screen
- To associate a Handler with a Logger
  - Use Logger method addHandler()

```
FileHandler h = new FileHandler("test.log");
logger.addHandler(h);
```

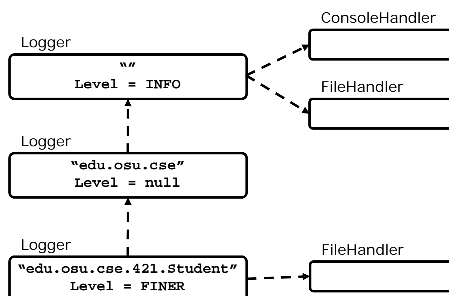
Logging Hierarchy

- Every logger has a parent logger
  - Follows naming scheme
    - "edu.osu.cse.421", if it exists, is parent of "edu.osu.cse.421.Student"
  - Default logging level is null
    - Receives parent's logging level
- When message meets logger's level
  - Passed along to associated handlers
  - Passed up to parent's handlers
    - Ignores parent's logging level
- Root logger
  - Named "" (the empty string)
  - By default, has level INFO, and has 1 handler (a ConsoleHandler)

Logging Hierarchy: Default



Logging Hierarchy: General



Logger Organization: Alternative

- One logger/class simplifies controlling output based on *structural* concerns
- A different segmentation would be based on *functional* concerns
- Example
  - AppLog: General application events
  - SQLLog: SQL-related processing activities
  - ThreadLog: Events related to managing the thread pool
  - RequestLog: Requests into the system, including the time to fulfill the request
  - DbConnectLog: Events related to managing the database connection pool

Eclipse Support

- Lots of boiler-plate code
- Approach 1: Modify method body template
  - Window > Preferences > Java > Code Style > Code Templates > Method Body
- Approach 2: Create new code template
  - Window > Preferences > Java
  - Editor > Templates > New
    - Name: logger
    - Pattern: private static final Logger logger = Logger.getLogger(${enclosing_type}.class.getName());
  - Now you can type "logger" inside any class, then use content-assist to fill in the rest

Configuration

- Default set in an external properties file
  - ${JDK_HOME}/jre/lib/logging.properties
- Defaults can be overridden
  - Provide a new file, eg mylog.prop
  - Run program with command-line argument
    - -Djava.util.logging.config.file=mylog.prop

Example Properties File


```
# Specify the handlers to create in the root logger
# (all loggers are children of the root logger)
# The following creates two handlers
handlers = java.util.logging.ConsoleHandler,
    java.util.logging.FileHandler

# Set the default logging level for the root logger
.level = ALL

# Set the default logging level for new ConsoleHandler instances
java.util.logging.ConsoleHandler.level = INFO

# Set the default logging level for new FileHandler instances
java.util.logging.FileHandler.level = ALL

# Set the default formatter for new ConsoleHandler instances
java.util.logging.ConsoleHandler.formatter =
    java.util.logging.SimpleFormatter

# Set the default logging level for the logger named edu.osu.cse.421
edu.osu.cse.421.level = ALL
```

Debugging

- Debuggers give us a way to stop a program and examine its contents
- Breakpoint: A stop sign
  - Whenever execution reaches that point, it stops
- Viewing state
  - Examine value of variables, fields, memory
    - A good toString method helps!
  - Watch certain variables or expressions
  - Change the value of variables
- Advancing execution
  - Step-into/over/return to take a small step forward (into next method / one line / out of method)
  - Resume to continue (until next breakpoint)

Summary

- Logging components from java.util.logging
  - Messages, Loggers, Handlers
  - (Also Filters and Formatters)
- Message Levels
  - End-users: SEVERE, WARNING, INFO
  - Administrators: CONFIG
  - Developers: FINE, FINER, FINEST
- Logger
  - Creation with static factory
  - Basic methods (log, info/fine/etc, entering/etc)
  - Eclipse support for boiler-plate code
- Configuration with external properties file
- Debugging in Eclipse