java.util.logging

Plan

On ne trace pas une application avec `println`!!

`System.err.println` est à proscrire (en général)

- `println` ne doit pas être utilisé pour débuguer
- ni pour tracer, car cela ne permet pas de choisir ce qui sort
- ET il faut tout enlever au fur et à mesure, et à la fin...
- N'importe quel *linter* vous le dira (*e.g. sonarlint*)

`java.util.logging` est là pour ça

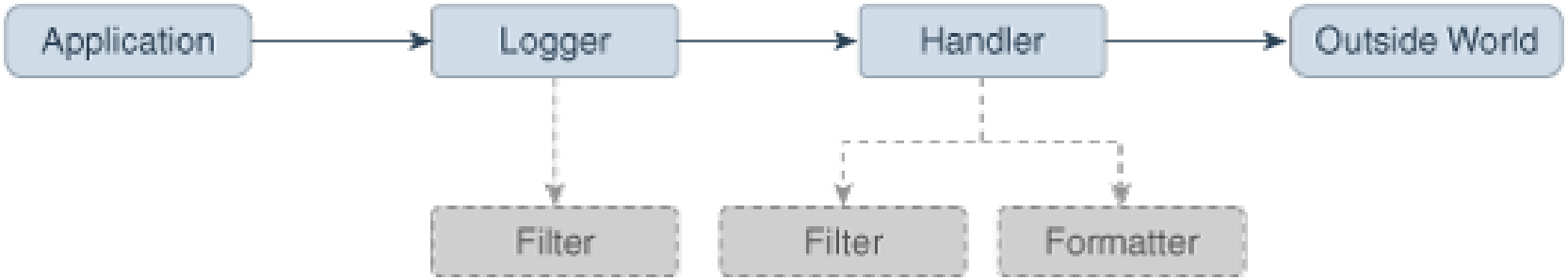
- permet de logger finement une application avec des sorties console/fichiers/réseau)
- $\Rightarrow$ administrateurs
- $\Rightarrow$ utilisateurs finaux

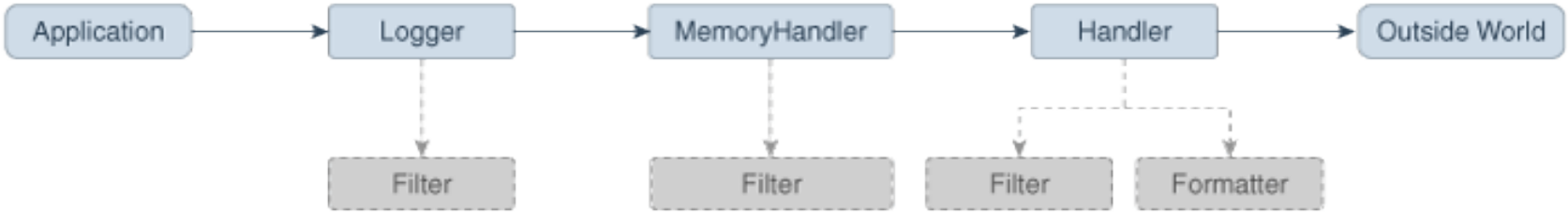
java.util.logging

Principales classes

- `Logger` : objet qui permettra de générer les logs
- `Level` : définit les différents niveaux de finesse des logs
- `LogManager` : gère la création et la gestion des `Loggers`
- `Handler` : exporte les logs vers le canal voulu, `e.g.` `FileHandler` pour écrire dans un fichier
- `Formatter` : formatte les logs au besoin
- `Filter` : affine le contrôle sur ce qui est loggé

flux de contrôle





Log Levels

Intérêt

- tous les logs possèdent un niveau de log associé
- ce niveau permet de définir la finesse et/ou l'importance du log
- la classe `Level` déinit des niveaux standards ▸ java.util.logging.Level

static Level	ALL	ALL indicates that all messages should be logged.
static Level	CONFIG	CONFIG is a message level for static configuration messages.
static Level	FINE	FINE is a message level providing tracing information.
static Level	FINER	FINER indicates a fairly detailed tracing message.
static Level	FINEST	FINEST indicates a highly detailed tracing message.
static Level	INFO	INFO is a message level for informational messages.
static Level	OFF	OFF is a special level that can be used to turn off logging.
static Level	SEVERE	SEVERE is a message level indicating a serious failure.
static Level	WARNING	WARNING is a message level indicating a potential problem.

java.util.logging.Logger

Création

- `Logger logger = LogManager.getLogger("package.subpackage");`
- le nom utilisé permet de créer un *namespace* hiérarchique
- le *root logger* a pour nom `""`
- les loggers héritent différents attributs de leur parent :
  - le *logging level*
  - les *handlers*
  - le *resource bundle* utilisé si le sien est `null`

java.util.logging.Logger

Utilisation

- méthodes `log(Level l, String message)` et des paramètres optionnels
- méthodes `logp` qui permettent de spécifier en plus les noms de la classe et de la méthode
- méthodes `logrb` : `logp` avec un *resource bundle* spécifique
- des méthodes alias pour chaque niveau de log standard (les plus utiles) : `warning, info, severe`, etc.
- ▸ java.util.logging.Logger

java.util.logging.Handler

Types

- `StreamHandler` : écrit les logs dans un `OutputStream`.
- `ConsoleHandler` : écrit les logs sur `System.err`
- `FileHandler` : écrit dans un fichier
- `SocketHandler` : écrit sur des ports TCP distants
- `MemoryHandler` : met en mémoire tampon les logs

Utilisation simple

```
package com.wombat;
import java.util.logging.*;

public class Nose {
    // Obtain a suitable logger.
    private static Logger logger = Logger.getLogger("com.wombat.nose");
    public static void main(String argv[]) {
        // Log a FINE tracing message
        logger.fine("doing stuff");
        try {
            Wombat.sneeze();
        } catch (Exception ex) {
            // Log the exception
            logger.log(Level.WARNING, "trouble sneezing", ex);
        }
        logger.fine("done");
    }
}
```

Ne log rien...

- car la configuration par défaut a été récupérée du fichier `java-home/conf/logging.properties`

java-home/conf/logging.properties

```
# "handlers" specifies a comma separated list of log Handler classes.
handlers= java.util.logging.ConsoleHandler

# To also add the FileHandler, use the following line instead.
#handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler

# Default global logging level.
.level= INFO

############################################################
# Handler specific properties.
# Describes specific configuration info for Handlers.
############################################################
# default file output is in user's home directory.
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
# Default number of locks FileHandler can obtain synchronously.
java.util.logging.FileHandler.maxLocks = 100
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

# Limit the message that are printed on the console to INFO and above.
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
```

Modifier la configuration du root logger ("") programmatiquement

```
public class ChangingConfig {

  public static void main(String[] args) throws SecurityException, IOException {
    Handler fh = new FileHandler("%t/wombat.log");// %t -> dans /tmp sous linux
    Logger logger = Logger.getLogger("");
    logger.addHandler(fh);
    fh.setLevel(Level.FINEST);
    Logger.getLogger("com.wombat").setLevel(Level.FINEST);

    Nose.main(args);
  }
}
```

Écraser la configuration par fichier

applicationLogging.properties

```
handlers=java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level=FINEST
```

Avec dans le code :

```
try (InputStream is = new FileInputStream("applicationLogging.properties")) {
  LogManager.getLogManager().readConfiguration(is);
} catch (IOException e) {
  e.printStackTrace();
}
```

ou en spécifiant sur la ligne de commande la propriété `java.util.logging.config.file`

```
java -Djava.util.logging.config.file="pathTto/applicationLogging.properties"  . . .
```

Ignorer la configuration globale

```
public class IgnoringGlobalConfiguration {//adding an handler to our logger

  private static Logger logger = Logger.getLogger("com.wombat.ignoring");

  public static void main(String argv[]) throws SecurityException, IOException {

    FileHandler fh = new FileHandler("mylog.txt");

    // Send logger output to our FileHandler.
    logger.addHandler(fh);

    // Request that every detail gets logged.
    logger.setLevel(Level.ALL);

    // Log a simple INFO message.
    logger.info("doing stuff");
    try {
      Wombat.sneeze();
    } catch (Exception ex) {
      logger.log(Level.WARNING, "trouble sneezing", ex);
    }
    logger.fine("done");
  }
}
```

Toujours préférer l'usage des méthodes utilisant des lambdas pour arguments

A set of methods alternatively take a "msgSupplier" instead of a "msg" argument. These methods take a `Supplier<String>` function which is invoked to construct the desired log message only when the message actually is to be logged based on the effective log level thus eliminating unnecessary message construction. For example, if the developer wants to log system health status for diagnosis, with the String-accepting version, the code would look like:

```
class DiagnosisMessages {
  static String systemHealthStatus() {
    // collect system health information
    ...
  }
}
...
logger.log(Level.FINER, DiagnosisMessages.systemHealthStatus());
```

Avec les lambdas

```
public class LambdaVersion {
  private static Logger logger = Logger.getLogger("com.wombat.lambda");

  public static void main(String argv[]) throws SecurityException, IOException {
    FileHandler fh = new FileHandler("mylog.txt");
    logger.addHandler(fh);
    logger.setLevel(Level.ALL);

    logger.info(() -> "doing stuff");

    try {
      Wombat.sneeze();
    } catch (Exception ex) {

      logger.log(Level.WARNING, ex, () -> "trouble sneezing");// NOT THE SAME
    }

    logger.fine(() -> "done");
  }
}
```

Conclusion

Sur le traçage

- c'est un point essentiel pour une application
- il faut le penser dès le début

java.util.logging

- permet de définir un namespace hiérarchique de loggers (optionnel)
- tous les aspects peuvent être configurés grâce à un fichier properties
- alternative : ▸ Log4j 2

Ce cours reprend le tutoriel d'Oracle : ▸ `java.util.logging`