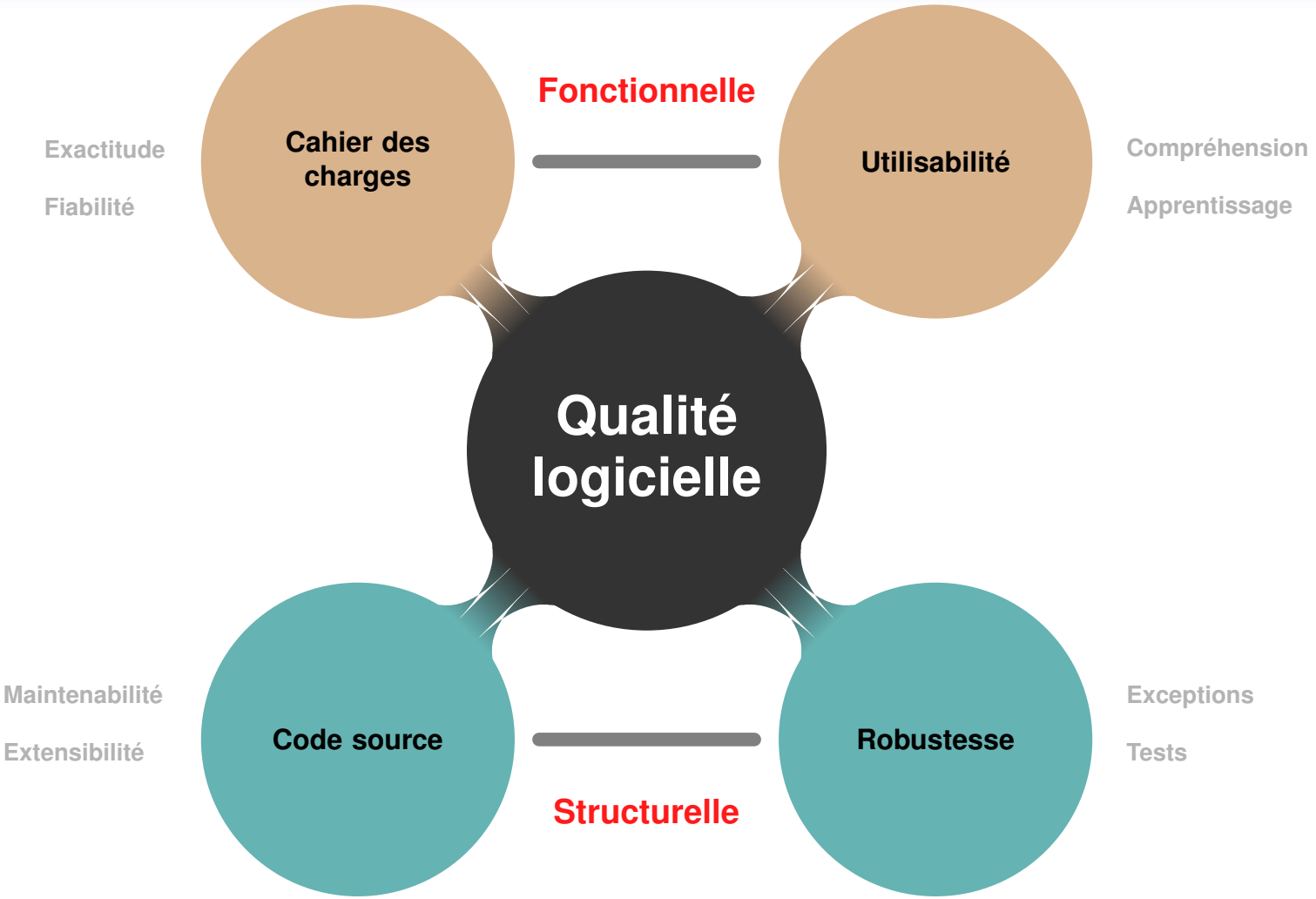
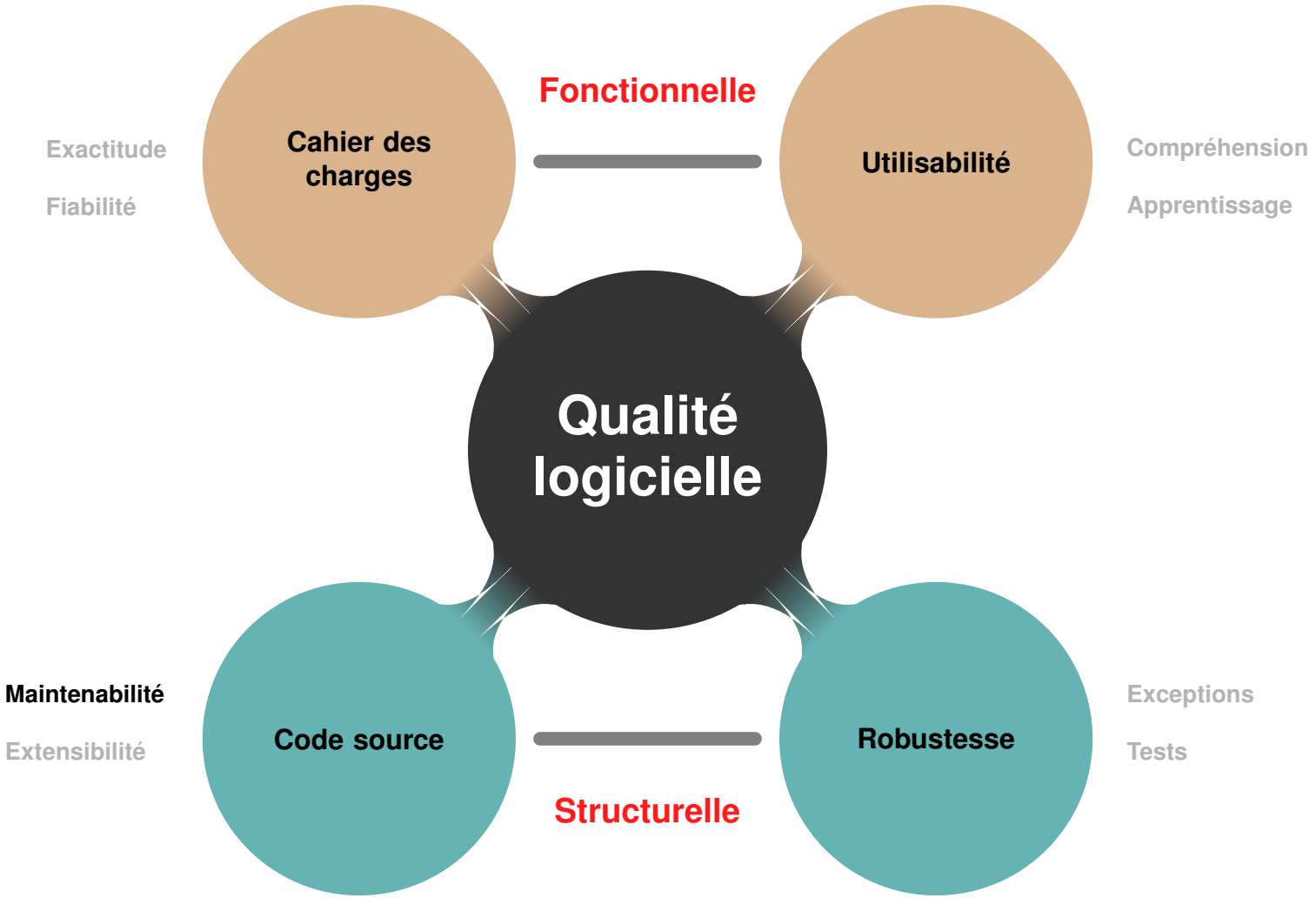
Traçage d'une application

Plan

Qualité fonctionnelle vs. structurelle



Savoir produire des logs est fondamental



Plan

Produire des logs est essentiel

Le traçage d'une application participe de la qualité d'un logiciel, et cela pour plusieurs aspects de son cycle de vie

Intérêts

- Diagnostic des problèmes par les utilisateurs finaux et les administrateurs système
- Diagnostic des problèmes par les ingénieurs du service
- Diagnostic par l'équipe de développement
- Production de données formatées dans des fichiers de logs
- ...

TOUS LES LANGAGES
possèdent une librairie dédiée au logging

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*)

Le package `java.util.logging` est là pour ça

- permet de tracer finement une application Java avec des sorties console/fichiers/réseau

java.util.logging

Package java.util.logging

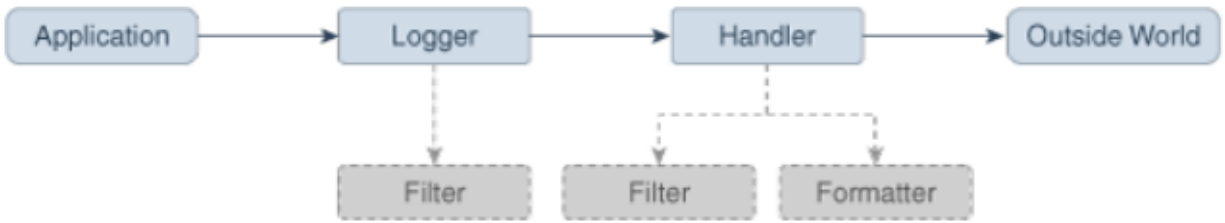
Principales classes du package

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

Plan

logique du flux de traçage

Figure 8-1 Java Logging Control Flow

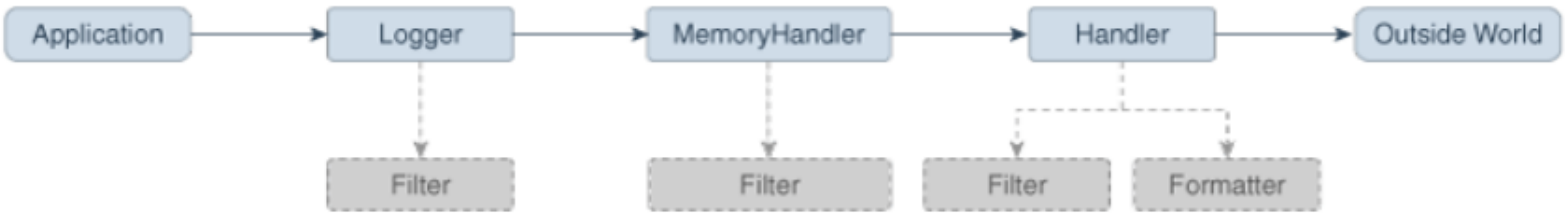


Each `Logger` object keeps track of a set of output `Handler` objects. By default all `Logger` objects also send their output to their parent `Logger`. But `Logger` objects may also be configured to ignore `Handler` objects higher up the tree.

Some `Handler` objects may direct output to other `Handler` objects. For example, the `MemoryHandler` maintains an internal ring buffer of `LogRecord` objects, and on trigger events, it publishes its `LogRecord` object through a target `Handler`. In such cases, any formatting is done by the last `Handler` in the chain.

logique du flux de traçage

Figure 8-2 Java Logging Control Flow with MemoryHandler



The APIs are structured so that calls on the Logger APIs can be cheap when logging is disabled. If logging is disabled for a given log level, then the Logger can make a cheap comparison test and return. If logging is enabled for a given log level, the Logger is still careful to minimize costs before passing the `LogRecord` to the `Handler`. In particular, localization and formatting (which are relatively expensive) are deferred until the `Handler` requests them. For example, a `MemoryHandler` can maintain a circular buffer of `LogRecord` objects without having to pay formatting costs.

Classe interne `Logger.Level`

▸ java.util.logging.Logger.Level

Intérêt : modélise le niveau d'un log

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

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

java.util.logging.Logger

```
public class Logger
extends Object
```

A Logger object is used to log messages for a specific system or application component. Loggers are normally named, using a hierarchical dot-separated namespace. Logger names can be arbitrary strings, but they should normally be based on the package name or class name of the logged component, such as java.net or javax.swing. In addition it is possible to create "anonymous" Loggers that are not stored in the Logger namespace.

Logger objects may be obtained by calls on one of the getLogger factory methods. These will either create a new Logger or return a suitable existing Logger. It is important to note that the Logger returned by one of the `getLogger` factory methods may be garbage collected at any time if a strong reference to the Logger is not kept.

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

Principales méthodes

- `addHandler(Handler handler)` : ajoute un canal pour les logs
- `setLevel(Level newLevel)` : règle le niveau des logs générés
- `setResourceBundle(ResourceBundle bundle)` : pour l'internationalisation des logs
- `log(Level l, String message)` crée un log avec un niveau
- `logp(Level level, String sourceClass, String sourceMethod, String msg)` crée et formate un log avec en plus les noms de la classe et de la méthode
- `logrb` : `logp` avec un *resource bundle* spécifique
- `setFilter` : spécifie le `Filter` implémentant la logique de filtrage sur les logs
- des alias pour chaque niveau de log standard (les plus utiles) : `info(String msg)`, `warning(String msg)`, etc.

java.util.logging.Handler

java.util.logging.Handler

```
public abstract class Handler
extends Object
```

A `Handler` object takes log messages from a `Logger` and exports them. It might for example, write them to a console or write them to a file, or send them to a network logging service, or forward them to an OS log, or whatever.

A `Handler` can be disabled by doing a `setLevel(Level.OFF)` and can be re-enabled by doing a `setLevel` with an appropriate level.

`Handler` classes typically use `LogManager` properties to set default values for the `Handler`'s `Filter`, `Formatter`, and `Level`. See the specific documentation for each concrete `Handler` class.

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

Chaque handler possède son propre niveau de log

⇒ permet d'avoir différents niveau de log suivant les canaux

Plan

Exemple

```
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) { // just for the sake of this example; don't do that catch!
            // 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 {
    Logger logger = Logger.getLogger("");

    Handler fh = new FileHandler("%t/wombat.log");// %t -> dans /tmp sous linux
    logger.addHandler(fh);
    fh.setLevel(Level.FINEST);

    Logger.getLogger("com.wombat").setLevel(Level.FINEST);

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

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");
  }
}
```

Écraser la configuration avec un 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"  . . .
```

Changer le formatage des logs

ex : mise en place d'un formatage pour la sortie sur fichier

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

    public static void main(String[] argv) throws SecurityException, IOException {
      FileHandler fh = new FileHandler("mylog.txt");
      fh.setFormatter(new Formatter() {

      @Override
      public String format(LogRecord record) {
        return record.getLevel()+" : "+record.getMessage()+"\n";
      }

      });

        logger.addHandler(fh);
        logger.setLevel(Level.ALL);
        logger.info(() -> "doing stuff");
        try {
            Wombat.sneeze();
        } catch (Exception ex) {
            logger.log(Level.WARNING, ex, () -> "trouble sneezing");
        }
        logger.fine("done");
    }
}
```

Générer des fichiers de données

ex : création d'un fichier csv

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

    public static void main(String[] argv) throws SecurityException, IOException {
      FileHandler fh = new FileHandler("experiment.csv");
      fh.setFormatter(new Formatter() {

        int count = 0;

      @Override
      public String format(LogRecord record) {
        count++;
        return count + " ; " + record.getMessage()+"\n";
      }

      });
        logger.addHandler(fh);
        logger.setLevel(Level.INFO);

        for (int i = 0; i < 100; i++) {
      logger.info(() -> ""+Math.random()*100);
    }
    }
}
```

Plan

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, quel que soit le langage

- 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
- il est possible de régler finement le niveau de log pour chaque handler
- alternative : ▸ Log4j 2

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