FICHE · JAVA · LOGGERS 1/2

Journaliser avec java.util.logging

Remplacer System.out.println par le logger fourni avec le JDK

Frédéric Guyomarch · novembre 2026

Quand un programme ne fait pas ce qu'on attend, le premier réflexe est souvent d'ajouter des `System.out.println` pour voir ce qui se passe. Ce réflexe dépanne en TP, mais il ne tient pas dans une vraie application : ces lignes s'affichent toujours, elles ne disent ni quand ni où elles ont été écrites, et pour s'en débarrasser il faut modifier le code. La journalisation, ou *logging*, répond à ce besoin de façon propre : le programme écrit des messages classés par gravité, et c'est la configuration, en dehors du code, qui décide lesquels sont conservés, où ils vont et sous quelle forme.

Cette fiche est la **première de deux**. Elle n'utilise que `java.util.logging`, le système fourni avec le JDK, et pose tout le vocabulaire (logger, handler, formatter, niveaux, configuration) ainsi que les bons réflexes. La seconde fiche présentera les façades et les frameworks des applications professionnelles : `System.Logger`, SLF4J, Logback et Spring Boot. Tous les exemples ont été exécutés avec Java 21.

1. Pourquoi ne pas utiliser System.out ?

```
void payer(Commande c) {
  System.out.println("debut payer");
  System.out.println("c = " + c);
  ...
  System.out.println("on passe ici");
}
```

Ce code pose trois problèmes. Il n'y a **pas de niveau** : rien ne distingue une trace de mise au point d'une erreur grave, tout a la même importance. Il n'y a **pas de contexte** : la ligne n'indique ni l'heure, ni le thread, ni la classe qui l'a écrite ; sur un serveur qui traite des centaines de requêtes, elle est inexploitable. Enfin il n'y a **pas de réglage** : c'est tout ou rien, toujours vers la console, et pour faire taire ces messages il faut retoucher le code et recompiler.

Un logger règle ces trois points. Chaque message porte un niveau de gravité, il est automatiquement daté et accompagné de son origine, et un fichier de configuration choisit, au lancement, ce qui doit être affiché. `java.util.logging`, souvent abrégé JUL, est inclus dans le JDK depuis Java 1.4 : il n'y a aucune dépendance à ajouter.

2. Le trajet d'un message

JUL repose sur une chaîne de trois objets :

- Le code appelle un **logger**, par exemple `LOG.info("Commande 42 validée")`. Chaque classe a en général son propre logger, nommé d'après elle. Le logger compare le niveau du message à son propre seuil ; si le message n'est pas assez grave, il est ignoré, et cela ne coûte presque rien.
- Sinon, il est transmis à un ou plusieurs **handlers**, qui choisissent la destination : `ConsoleHandler` pour la console, `FileHandler` pour un fichier, `SocketHandler` pour un serveur.
- Chaque handler utilise un **formatter**, qui met le message en forme : `SimpleFormatter` produit du texte, `XMLFormatter` du XML.

Le résultat est une ligne comme celle-ci, où l'on lit la date, l'heure, le niveau, le nom du logger et le message :

```
2026-11-13 11:10:20 INFO    fr.iut.panier.Panier - Commande 42 validée
```

3. Les niveaux de gravité

Chaque message a un niveau, défini dans la classe `java.util.logging.Level` avec une valeur numérique :

Niveau	Valeur	Quand l'utiliser
`SEVERE`	1000	Une opération a échoué, quelqu'un doit agir
`WARNING`	900	Situation anormale, mais gérée par le programme
`INFO`	800	Événement métier important : démarrage, commande validée
`CONFIG`	700	Paramètres de configuration lus au démarrage
`FINE`	500	Détails utiles au développeur
`FINER`, `FINEST`	400, 300	Suivi très fin : entrée et sortie de méthodes

Le niveau d'un logger fonctionne comme un seuil : réglé sur `INFO`, il laisse passer `INFO` et tout ce qui est plus grave (`WARNING`, `SEVERE`), et ignore `FINE` et en dessous. Les niveaux spéciaux `Level.ALL` et `Level.OFF` laissent tout passer ou bloquent tout. En développement on descend souvent à `FINE` ; en production on reste en général sur `INFO`.

Pour s'entraîner à choisir, on peut classer quelques messages : « connexion à la base de données impossible » relève de `SEVERE` ; « fichier de configuration absent, valeurs par défaut utilisées » de `WARNING` ; « commande 42 validée » d'`INFO` ; « valeur de la variable i » de `FINE` ou `FINER`.

4. Déclarer et utiliser un logger

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class Panier {
  private static final Logger LOG =
      Logger.getLogger(Panier.class.getName());

  void ajouter(Article a) {
    LOG.fine(() -> "Ajout de " + a);
    if (a.prix() < 0)
      LOG.warning("Prix négatif : " + a.id());
  }
}
```

On déclare un logger par classe, en constante `private static final`, nommé avec le nom complet de la classe, ici `fr.iut.panier.Panier`. Ces noms pointés forment une **hiérarchie** : le logger `fr.iut.panier.Panier` a pour parent `fr.iut.panier`, lui-même enfant de `fr.iut`, et ainsi de suite jusqu'au logger racine, dont le nom est vide. Un réglage posé sur `fr.iut` s'applique donc à toutes les classes du paquetage et de ses sous-paquetages, sauf si un réglage plus précis le remplace.

La méthode `fine` reçoit ici une lambda plutôt qu'une chaîne. La différence est importante : avec une chaîne, la concaténation est faite à chaque appel, même si le niveau `FINE` est désactivé ; avec une lambda, le message n'est construit que s'il doit être écrit.

5. Configurer avec logging.properties

Sans configuration, JUL écrit les messages de niveau `INFO` et plus sur la console, sur deux lignes par message. Pour changer ce comportement, on écrit un fichier de propriétés :

```
# logging.properties
handlers = java.util.logging.ConsoleHandler,\
           java.util.logging.FileHandler
.level = INFO
fr.iut.panier.level = FINE
java.util.logging.ConsoleHandler.level = FINE

java.util.logging.FileHandler.pattern = app%u.log
java.util.logging.FileHandler.formatter =\
           java.util.logging.SimpleFormatter
java.util.logging.SimpleFormatter.format =\
           %1$tF %1$tT %4$-7s %3$s - %5$s%6$s%n
```

et on l'indique au lancement du programme :

```
java -Djava.util.logging.config.file=logging.properties fr.iut.panier.App
```

La ligne `.level` règle le logger racine, et `fr.iut.panier.level` abaisse le seuil pour notre paquetage seulement. Le piège classique est le suivant : on passe le logger en `FINE` et rien ne

s'affiche de plus. C'est que le message doit franchir **deux filtres** : le niveau du logger, puis celui du handler. Or le `ConsoleHandler` est réglé sur `INFO` par défaut ; il faut donc l'abaisser lui aussi.

Le `FileHandler` écrit dans un fichier ; sans indication, il produit du XML, d'où la ligne qui lui associe un `SimpleFormatter`. Le format de ce dernier suit la syntaxe de `String.format` : l'argument 1 est la date, 3 le nom du logger, 4 le niveau, 5 le message et 6 l'éventuelle exception. Tout se règle au lancement : on change ce qui est affiché sans toucher au code.

6. Journaliser une exception

Deux façons de faire sont à éviter :

```
catch (NumberFormatException e) {
  e.printStackTrace();           // écrit sur System.err
  LOG.severe("Erreur : " + e);   // ne garde que la 1re ligne
}
```

`printStackTrace` écrit directement sur la sortie d'erreur, hors de toute configuration ; concaténer l'exception dans le message ne garde que son nom et son message, sans la pile d'appels. La bonne façon est de passer l'objet exception en argument de `log` :

```
catch (NumberFormatException e) {
  LOG.log(Level.SEVERE, "Quantité illisible : " + saisie, e);
}

// Ou, avec un message construit à la demande
LOG.log(Level.WARNING, e, () -> "Saisie : " + saisie);
```

La trace complète est alors écrite, causes comprises. Le message doit aussi donner le **contexte** utile pour reproduire le problème : la valeur fautive, l'identifiant de la commande, l'utilisateur concerné. Enfin, on journalise une exception **une seule fois** : si on la journalise puis la relance, la couche du dessus la journalisera à son tour et la même erreur apparaîtra plusieurs fois dans les logs.

7. Les bonnes pratiques avec JUL

```
// À éviter
LOG.fine("Panier : " + panier);
LOG.info("Connexion de " + login + " / " + motDePasse);

// Recommandé
LOG.fine(() -> "Panier : " + panier);
LOG.log(Level.INFO, "Connexion réussie pour {0}", login);
if (LOG.isLoggable(Level.FINE)) {
  // plusieurs lignes de détail…
}
```

- **Choisir le bon niveau.** `INFO` pour les événements métier, `FINE` pour les détails, `SEVERE` seulement quand quelqu'un doit agir. Si tout est en `SEVERE`, plus personne ne regarde les erreurs.

- **Ne pas calculer pour rien.** Une lambda, ou des paramètres `{0}`, `{1}` selon la syntaxe de `MessageFormat` ; le test `isLoggable` quand tout un bloc de code ne sert qu'à la journalisation.
- **Ne jamais journaliser de secrets.** Ni mot de passe, ni jeton d'authentification, ni numéro de carte, et pas de données personnelles inutiles : les logs sont copiés, archivés, envoyés vers d'autres outils et lus par d'autres personnes. Le RGPD s'applique aux données personnelles qu'ils contiennent.
- **Écrire des messages utiles.** Une ligne par événement, avec les identifiants qui permettent de retrouver la situation, sans journaliser dans une boucle exécutée des millions de fois.
- **Garder la configuration hors du code.** Les niveaux et les destinations se règlent dans `logging.properties`, pas par des appels dans le programme.

8. À retenir

1. Un logger `static final` par classe, plutôt que `System.out.println`.
2. Logger, handler, formatter : qui filtre, vers où, sous quelle forme.
3. Deux seuils : celui du logger et celui du handler.
4. `logging.properties` règle tout au lancement, sans recompiler.
5. L'exception en argument, un message construit à la demande, jamais de secrets.

Conseil : reprendre la SAÉ et remplacer ses `println` par un logger JUL, puis changer le niveau dans `logging.properties` et constater, sans recompiler, que les messages `FINE` apparaissent ou disparaissent.

Pour aller plus loin : la documentation du paquetage `java.util.logging` dans la Javadoc du JDK. La suite, avec SLF4J, Logback et Spring Boot, fera l'objet de la seconde fiche.