R2-01-03 : TP 5

Développement orienté objets & Qualité de développement

POUR COMMENCER **1**

EXERCICE 1 : UTILISATION D'UN LOGGER **1**

EXERCICE 1.1 : PRODUIRE DES LOGS 1
EXERCICE 1.2 : CONFIGURER LES LOGS 3

EXERCICE 2 : OPERATIONS MATHEMATIQUES **5**

EXERCICE 2.1 : TABLE D'OPERATION 5
EXERCICE 2.2 : MODE SANS ERREUR 8

Pour commencer

Continuer dans le projet créé pour les premiers TPs, pour se faire **créer** un package tp5.

Continuer à utiliser les outils mis à votre disposition vu en R1.01 et au début de ce module : les mécanismes automatiques dans l'IDE, le deboggueur, etc.

Exercice 1 : Utilisation d'un logger

Objectifs R2-03 : Afficher des « vrais » logs
Source : https://koor.fr/Java/Tutorial/

Dans ce TP, vous allez utiliser la librairie **`java.util.logging`**. Vous l'aurez compris grâce au nom du package, cette API, proposée de base, est utilisée pour l'affichage de logs sous différentes formes, différents formats et différentes granularités. Plus besoin d'utiliser les **`System.out.println()`** !

Présentation de l'API **`java.util.logging`** dans la javadoc :

https://docs.oracle.com/en/java/javase/11/core/java-logging-overview.html

L'exercice 1 est un exercice très simple pour vous entrainer à produire et configurer vos logs. Vous mettrez également en pratique dans l'exercice suivant : la table des opérations.

Exercice 1.1 : Produire des logs

Dans le paquetage **`tp5`**, créer la classe **`TestLogging`** avec une méthode statique **`main`**.

Dans un premier temps, vous allez récupérer le logger. Pour récupérer ce logger, il faut utiliser la méthode statique **`Logger.getLogger`** du package **`java.util.logging`**. Cette méthode accepte en paramètre une chaîne de caractères. Il s'agit du nom du logger. Vous pouvez mettre le nom qui vous convient. Dans l'exemple suivant, nous avons nommé le logger avec le nom du package contenant la classe **`TestLogging`**.

Récupération du logger :

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

public class TestLogging {

    // Récupération du logger
```

```
    private static Logger LOGGER =
Logger.getLogger(TestLogging.class.getPackageName());

    public static void main(String[] args) {

        // TODO: utiliser le logger
    }
}
```

Dans un deuxième temps, vous utiliserez le logger dans le main pour afficher un message : « Mon premier log ! ».

Utilisation du logger :

```
LOGGER.log( Level.INFO, "Mon premier log !" );
```

Affichage de la trace obtenu pour le premier run :

```
May 01, 2022 9:28:15 PM tp5.TestLogging main
INFO: Mon premier log !
```

Il existe différents niveaux de logs. Par ordre décroissant d'importance :

- SEVERE (highest value)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (lowest value)

Pour plus d'information, consultez la javadoc de la classe **LEVEL** :

https://docs.oracle.com/en/java/javase/11/docs/api/java.logging/java/util/logging/Level.html

Dans la suite, vous ajouterez des instructions pour **réaliser une division** avec :

- Un dividende random entre 0 et 9 (compris)
- Un diviseur random entre 0 et 2 (compris)

AIDE : Pour les allergiques aux maths, un peu de vocabulaire (source : wikipédia). La division fait apparaitre trois nombres :

- Le nombre qui est divisé s'appelle le dividende ;
- Le nombre qui divise s'appelle le diviseur ;
- Le résultat de l'opération s'appelle le quotient.

Dans 18 / 3 = 6, 18 est le dividende, 3 est le diviseur et 6 est le quotient (de 18 par 3).

Concernant les logs à ajouter :

- Un warning pour prévenir d'un éventuel problème avec la division : « Attention à une division par zéro peut se produire »

- Si la division se passe sans erreur, une log information sur le dividende, le diviseur et le quotient
- Si la division a provoquer une erreur, un log SEVERE affichant le message de l'exception (vous devez donc attraper l'exception pouvant être produire par la division)

Ci-dessous les traces attendues à la fin de votre programme :

Trace avec une division sans erreur :

```
May 01, 2022 9:53:41 PM tp5.TestLogging main
INFO: Mon premier log !
May 01, 2022 9:53:41 PM tp5.TestLogging main
WARNING: Attention à une division par zéro peut se produire
May 01, 2022 9:53:41 PM tp5.TestLogging main
INFO: dividende = 3, diviseur = 2 et quotient = 1
```

Trace avec une division qui provoque une erreur :

```
May 01, 2022 9:48:09 PM tp5.TestLogging main
INFO: Mon premier log !
May 01, 2022 9:48:09 PM tp5.TestLogging main
WARNING: Attention à une division par zéro peut se produire
May 01, 2022 9:48:09 PM tp5.TestLogging main
SEVERE: Le message d'exception : java.lang.ArithmeticException: divide by
zero
```

AIDE : Pour faciliter votre affichage dans les logs, vous pouvez utiliser la séquence d'instruction ci-dessous.

```
Object[] data = { dividende, diviseur, quotient };
LOGGER.log( Level.INFO, "dividende = {0}, diviseur = {1} et quotient = {2}", data );
```

Exercice 1.2 : Configurer les logs

L'API **`java.util.logging`** ne permet pas simplement l'affichage de logs. Vous allez pouvoir configurer finement leur affichage en fonction de vos besoins.

Modifier la granularité d'affichage des logs en utilisant la méthode **`setLevel(…)`** du logger. Cette méthode permettra de n'afficher que les logs du même niveau d'importance que précisé ou plus haut, sans enlever les logs inférieurs. Dans l'exemple ci-dessous, le programme n'affichera que les logs WARNING et SEVERE, sans enlever les logs INFO mis précédemment :

```
// Récupération du logger
private static Logger LOGGER =
Logger.getLogger(TestLogging.class.getPackageName());

// Configuration du logger
static {
    LOGGER.setLevel(Level.WARNING);
}
```

Les granularités d'affichage sont les suivants : **`OFF / SEVERE / WARNING / INFO / CONFIG / FINE / FINER / FINEST / ALL`**

Modifier l'affichage des logs :

```
// Configuration du logger
static {
```

```
    System.setProperty("java.util.logging.SimpleFormatter.format", "[%1$s]
%4$-10s | (%3$s) %2$-15s | %5$s\n");
    LOGGER.setLevel(Level.WARNING);
}
```

Par défaut, la classe **SimpleFormatter** affiche un log sur deux lignes avec un format bien précis. Avec cette ligne de configuration, nous changeons le format des logs produits. La syntaxe de définition du format ressemble à celle utilisée par la méthode System.out.printf avec l'ajout d'index (entre les caractères % et $). Ces index correspondent à :

1. la date de production du log
2. la source (nom du type et nom de la méthode) si elle est connue ou bien le nom du logger
3. le nom du logger
4. le niveau du log (INFO, SEVERE...). Dans l'exemple ci-dessus, je l'affiche sur 10 caractères et aligné par la gauche (-) : %4$-10s.
5. le message du log
6. l'exception associée si elle est spécifiée

Pour plus d'informations :

https://docs.oracle.com/en/java/javase/11/docs/api/java.logging/java/util/logging/SimpleFormatter.html

Trace obtenue après modification de l'affichage :

```
[2022-05-01T22:28:34.048006+02:00[Europe/Paris]] WARNING    | (tp5) tp5.TestLogging main | Attention à la division par zéro
[2022-05-01T22:28:34.103760+02:00[Europe/Paris]] SEVERE     | (tp5) tp5.TestLogging main | Le message d'exception :
java.lang.ArithmeticException: divide by zero
```

Dans les deux exemples ci-avant, la configuration se fait directement dans la classe. Nous allons privilégier la création d'un fichier de configuration.

Créer un dossier **conf** à la racine de votre projet et, dans ce dossier, un fichier **debug-logging.properties**. Dans ce fichier, vous allez mettre la configuration de votre logger.

Exemple de fichier de configuration :

```
# On log sur la console et dans un fichier.
handlers=java.util.logging.ConsoleHandler, java.util.logging.FileHandler

# On peut configurer le ConsoleHandler, mais ici j'utilise sa configuration par
défaut.
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter

# On configure notre FileHandler (il utilise lui aussi un SimpleFormatter).
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter
java.util.logging.FileHandler.pattern=R2.01.02-%u-%g.log

# On change le format des logs pour notre SimpleFormatter.
java.util.logging.SimpleFormatter.format=[%1$s] %4$-10s | (%3$s) %2$-15s | %5$s\n

# Rappels sur les niveaux :
# OFF / SEVERE / WARNING / INFO / CONFIG / FINE / FINER / FINEST / ALL
# On limite tous les logs des autres composants (des autres packages) à l'affichage
des erreurs.
.level=SEVERE

# On active les logs du package tp5 sur WARNING (et donc SEVERE).
tp5.level=WARNING
```

Ce fichier de configuration permet de faire des logs sur la console et dans un fichier (le fichier de logs sera créé à la racine de votre projet, dans notre cas R2.01.02-0-0.log) et de limiter l'affichage des logs à SEVERE pour l'ensemble du projet, sauf dans le package tp5 ou le niveau d'affichage commence à WARNING : tp5.level=WARNING.

Comment utiliser ce fichier de configuration ? Remplacer les configurations faites avant par :

```
// Récupérarion du gestionnaire de logs.
private static final LogManager logManager = LogManager.getLogManager();

// Configuration du logger
// EditConfiguration > Modify options > add VM options :
// -Djava.util.logging.config.file=conf/debug-logging.properties
static{
    try {
        logManager.readConfiguration( new FileInputStream("conf/debug-
logging.properties") );
    } catch ( IOException exception ) {
        LOGGER.log( Level.SEVERE, "Cannot read configuration file",
exception );
    }
}
```

OU en ajoutant une option à la configuration de lancement du programme : Run > Edit Configurations > Modify options > add VM options et ajouter l'option VM suivante « -Djava.util.logging.config.file=conf/debug-logging.properties »

IMPORTANT : le fichier de configuration n'est à appeler qu'une seule fois ! Soit dans la classe avec le main soit dans les options de configuration de lancement du programme. Dans les autres classes, il faut juste récupérer le logger seulement :

```
// Récupération du logger
private static Logger LOGGER =
Logger.getLogger(TableDOperation.class.getPackageName());
```

Exercice 2 : Opérations mathématiques

Objectifs R2-01 : objet, classe, héritage, interaction et gestion des erreurs
Objectifs R2-03 : utilisation des logs

Cet exercice s'inspire de la table de multiplication vu dans le TP4. Vous allez réaliser une table d'opérations mathématiques : soit des additions, soit des soustractions, soit des multiplications. Au gros, c'est presque le même exercice mais plus générique.

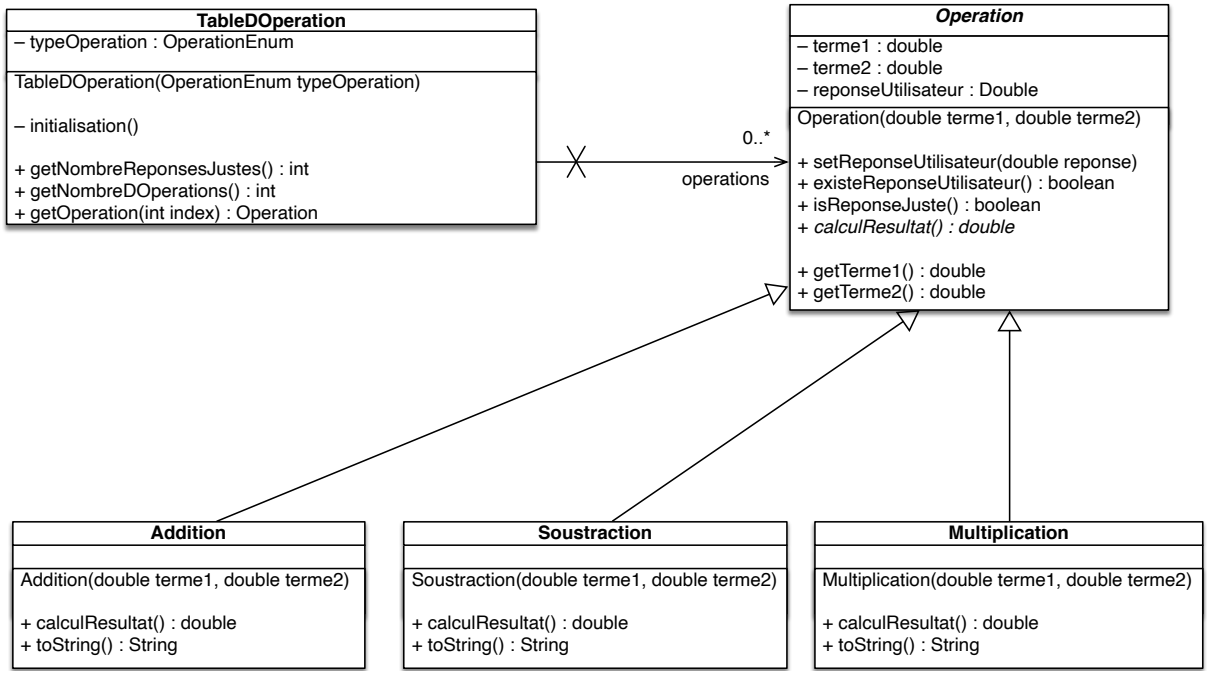
Donc, dans la suite, vous réaliserez une application sur le terminal pour vous exercer aux opérations d'addition, de soustraction et de multiplication. Cette application proposera à l'utilisateur de travailler sur une suite de 5 opérations d'un même type.

VOUS DEVEZ AJOUTER DES LOGS COMME VU DANS L'EXERCICE PRÉCÉDENT !

Exercice 2.1 : Table d'opération

Dans le paquetage **`tp5`**, créer le package **`tabledoperation`** et les classes **`TableDOperation`**, **`Operation`**, **`Addition`**, **`Soustraction`**, **`Multiplication`**, **`OperationUtilitaire`** et **`TestTableDOperation`**. La classe **`OperationUtilitaire`** est disponible à la fin de cet exercice.

Compléter les classes en respectant le diagramme UML ci-après.



Explication de certaines méthodes de la classe `TableDOperation` :

- La méthode **`initialisation`**() initialise la liste des opérations. Pensez à mettre des constantes dans la classe.
- La méthode **`getNombreDeReponsesJustes`**() retourne le nombre de réponses justes données par l'utilisateur.

Explication de certaines méthodes de la classe ABSTRAITE `Operation` :

- L'attribut **`reponseUtilisateur`** est un objet de type **`Double`** pour avoir la possibilité de le laisser **`null`** au début. Si l'attribut est **`null`**, l'utilisateur n'a pas répondu (un attribut de type **`double`** aurait eu une valeur par défaut de 0.0). L'utilisation d'un objet dans ce cas précis permet d'avoir une information supplémentaire.
- La méthode **`calculResultat`**() est une méthode abstraite. Les classes filles de la classe **`Operation`** devront l'implémenter.
- La méthode **`existeReponseUtilisateur`**() retourne vrai si l'utilisateur a donné une réponse.

Explication de certaines méthodes des classes filles d'`Operation` :

- La méthode **`calculResultat`**() retourne le résultat de l'opération entre le terme 1 et le terme 2.
- La méthode **`toString`**() retourne la description de l'opération « 2 + 5 = »

Compléter la méthode main de la classe **`TestTableDOperation`** pour correspondre à la trace ci-dessous. Pensez à gérer les erreurs de saisies de l'utilisateur, pour le moment seulement des nombres en dehors des attentes. **IMPORTANT**, ajouter au fur et à mesure les logs, voir juste après !

Trace attendue : les réponses en rouge sont les entrées clavier de l'utilisateur. En gras, la gestion des erreurs de saisies (on redemande à l'utilisateur) :

```
Addition 1 ou Soustraction 2 ou Multiplication 3 ? 4
Merci de répondre par 1 ou 2 ou 3 ? 1
Donner les réponses aux opérations :
2.6 + 9.3 = 11.9
7.5 + 18.3 = 25.8
0.8 + 5.5 = 6.3
10.8 + 8.6 = 19.4
9.3 + 10.5 = 19.8
Nombre de réponses justes : 5
```

Compléter les différentes classes et méthodes pour obtenir des logs comme ci-après. Un log donne plusieurs indications, la classe et la méthode dont il est issue (<init> signifie constructeur) et bien sûr le niveau de log. On peut remarquer qu'un WARNING est remonté lorsque l'utilisateur a donné une mauvaise réponse (log obtenu dans la méthode **`getNombreDeReponsesJustes()`** de la classe **`TableDOperation`**). Vous pouvez ajouter d'autres logs pour vous aider.

Exemple de logs attendus (extrait du fichier de log) :

```
INFO        | (tp5.tabledoperation) tp5.tabledoperation.TestTableDOperation
main | Type d'operation : ADDITION
INFO        | (tp5.tabledoperation) tp5.tabledoperation.TestTableDOperation
main | Mode sans erreur : false
INFO        | (tp5.tabledoperation) tp5.tabledoperation.TableDOperation
initialisation | Initialisation des opérations
INFO        | (tp5.tabledoperation) tp5.tabledoperation.Operation <init> |
Création d'opération avec les termes 7.3 et 6.6
INFO        | (tp5.tabledoperation) tp5.tabledoperation.Operation <init> |
Création d'opération avec les termes 9.6 et 9.7
INFO        | (tp5.tabledoperation) tp5.tabledoperation.Operation <init> |
Création d'opération avec les termes 9.5 et 13.5
INFO        | (tp5.tabledoperation) tp5.tabledoperation.Operation <init> |
Création d'opération avec les termes 1 et 10.4
INFO        | (tp5.tabledoperation) tp5.tabledoperation.Operation <init> |
Création d'opération avec les termes 2.1 et 11
INFO        | (tp5.tabledoperation) tp5.tabledoperation.TableDOperation
getNombreDeReponsesJustes | Une réponse juste de l'utilisateur
INFO        | (tp5.tabledoperation) tp5.tabledoperation.TableDOperation
getNombreDeReponsesJustes | Une réponse juste de l'utilisateur
INFO        | (tp5.tabledoperation) tp5.tabledoperation.TableDOperation
getNombreDeReponsesJustes | Une réponse juste de l'utilisateur
WARNING     | (tp5.tabledoperation) tp5.tabledoperation.TableDOperation
getNombreDeReponsesJustes | Une réponse fausse de l'utilisateur
WARNING     | (tp5.tabledoperation) tp5.tabledoperation.TableDOperation
getNombreDeReponsesJustes | Une réponse fausse de l'utilisateur
```

La classe OperationUtilitaire :

```java
import java.util.Random;

public class OperationUtilitaire {

    private static final Random random = new Random();
    private static final int RANGE_MAX = 200;
    private static final int ARRONDI = 10;

    // Retourne une valeur de type double entre 0 et RANGE_MAX/ARRONDI
    public static double randomDouble() {
```

```
        return (double)random.nextInt(RANGE_MAX)/ARRONDI;
    }

    // Retourne la valeur double arrondi
    // Evite les erreurs de calcul du au double en Java
    public static double arrondir(double valeur) {
        return (double)Math.round(valeur * ARRONDI) / ARRONDI;
    }
}
```

Exercice 2.2 : Mode sans erreur

Objectifs R2-03 : création et traitement d'une exception

Nous souhaitons rajouter à l'application un mode sans erreur. C'est-à-dire redemander à l'utilisateur une réponse tant que la sienne n'est pas la bonne, voir trace ci-dessous.

Pour ce faire, vous utiliserez le mécanisme d'exception :

- Créer une classe **`ErreurOperationException`**. Cette classe héritera de la classe **`Exception`**. Elle ne contiendra rien mais son nom sera significatif (comme **`NullPointerException`**)
- La méthode **`setReponseUtilisateur`**(…) provoquera une exception de type **`ErreurOperationException`** lorsque la réponse de l'utilisateur ne correspondra pas à la solution.
- L'exception sera « remontée » jusqu'au test pour être traitée.

Ajouter des logs pour vous aider.

AIDE :

Modifier les constructeurs de **`TableDOperation`** et **`Operation`** pour prendre en compte le mode sans erreur.

Modifier la classe **`TestTableDOperation`** pour ajouter une méthode de demande de réponse utilisateur qui se rappellera en cas d'erreur de calcul de l'utilisateur :

```
private static void demandeReponseUtilisateur(Operation  operation){

    // Affichage de l'opération
    System.out.print(operation);

    // Demander la réponse utilisateur
    double reponseUtilisateur = entree.nextDouble();
    entree.nextLine();

    // Enregistrer la réponse utilisateur
    // Mode sans erreur : attraper l'exception si la réponse de
    // l'utilisateur n'est pas la bonne et redemander la reponse
    // utilisateur
    // A COMPLETER
}
```

Trace attendue : les réponses en rouge sont les entrées clavier de l'utilisateur :

```
Addition 1 ou Soustraction 2 ou Multiplication 3 ? 1
Mode sans erreur true ou false ? true
Donner les réponses aux opérations :
5.6 + 1.5 = 6
```

```
Votre réponse n'est pas correcte, réessayer !
5.6 + 1.5 = 7.1
…
```