

Exercice JVA

Exercice1 : les nouveautés

La plus emblématique des nouveautés est l'utilisation des lambda expressions comme une avancée des classes anonymes. Son but est de réduire le volume de code défini inutilement car seule une méthode est nécessaire.

La syntaxe de base d'une lambda expression est la suivante : « paramètres -> corps ». Le compilateur utilise généralement le contexte de la lambda expression pour définir l'interface fonctionnelle et le type des paramètres. Il y a 4 règles importantes au niveau de la syntaxe :

- 1- Déclarer le type des paramètres est optionnel,
- 2- Utiliser des parenthèses autour des paramètres est optionnel si vous avez seulement un paramètre,
- 3- Utiliser les accolades est optionnel autour du corps s'il y a seulement un traitement,
- 4- Placer le mot clé return est optionnel s'il y a seulement une seule expression qui retourne une valeur.

Etape1 : construire une classe `Etape1` avec des lambda expressions :

- un attribut de type `Runnable` pour faire un affichage « Exercice1.Etape1 », cette définition est faite sous la forme d'une lambda expression,
- une méthode `main` pour faire les tests et appeler cette lambda expression,
- ajouter à la méthode `main` une variable `strArray` qui est un tableau de `String` initialisée avec les prénoms des stagiaires de la salle. Ensuite, il est demandé de trier les éléments de ce tableau à l'aide de la classe `Arrays` et de sa méthode `sort`. Le second paramètre est un objet qui implémente l'interface `Comparable<E>`. Ce second paramètre est à fournir sous la forme d'une lambda expression énonçant le tri.

```
package fr.paris.java.exercice1;

import static java.lang.System.out;

import java.util.Arrays;

public class Etape1 {
    protected Runnable r1 = () -> out.println(this);

    public static void main(String[] args) {
        Etape1 etape1 = new Etape1();
        etape1.r1.run(); // exercice1.Etape1

        String[] strArray = {
            "Jean", "Philippe", "Gerard"
        };
        Arrays.sort(strArray, (String s1, String s2) -> s2.length() - s1.length());
        for (int i = 0; i < strArray.length; i++) {
```

```

        out.print(strArray[i] + ", ");
    }
}

public String toString() {
    return "exercice1.Etape1";
}
}

```

Parce que les lambda expressions sont un peu comme des méthodes sans objet support, il est utile de pouvoir se référer à des méthodes existantes lorsqu'elles existent au lieu d'employer des lambda expressions.

Etape2 : construire une classe `Etape2` avec des références à des méthodes :

- une méthode static `doWork()` qui permet d'afficher le nom du thread courant et attend une poignée de secondes avant de boucler 50 fois,
- une méthode `main` pour faire les points suivants :
 - o créer un thread dont la méthode exécutée est `doWork()` , le démarrer,
 - o créer un thread dont la méthode exécutée est donnée sous la forme d'une lambda expression qui appelle `doWork()` , le démarrer,
 - o créer un thread de manière traditionnelle avec une classe anonyme implémentant `Runnable` dont la méthode `run()` appelle `doWork()`.

```

package fr.paris.java.exercice1;

public class Etape2 {

    public static void main(String[] args) {
        new Thread(Etape2::doWork).start();
        new Thread(() -> doWork()).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                doWork();
            }
        }).start();
    }

    static void doWork() {
        String name = Thread.currentThread().getName();
        for (int i = 0; i < 50; i++) {
            System.out.printf("%s: %d%n", name, i);
            try {
                Thread.sleep((int) (Math.random() * 50));
            } catch (InterruptedException ie) {
            }
        }
    }
}

```

Une interface fonctionnelle est définie comme une interface avec exactement une méthode abstraite. Cela s'applique même aux interfaces Java déjà existantes dans les versions passées de Java.

Etape3 : construire une classe `Etape3` qui utilise une interface fonctionnelle :

- définir une interface fonctionnelle nommée `Formatter`, avec une obligation de codage nommée `format` qui prend en entrée une chaîne de caractères qui décrit un format et un nombre variable d'`Object` paramètres et qui retourne une chaîne de caractères,
- une méthode static nommée `forEach` ayant deux paramètres :
 - o une liste de chaînes de caractères
 - o un `formatter`

Son but est d'afficher les éléments de la liste en utilisant le formatteur,

- une méthode `main` pour faire les points suivants :
 - o créer une liste de chaînes de caractères,
 - o appeler la méthode static `forEach` 2 fois avec la liste. La 1^{ère} invocation utilisera une référence à la méthode `String::format`. La 2^{ème} invocation utilisera une lambda expression équivalente dont le corps invoque la méthode `format`.

```
package fr.paris.java.exercice1;

import java.util.Arrays;
import java.util.List;

@FunctionalInterface
interface Formatter
{
    String format(String fmtString, Object... arguments);
}

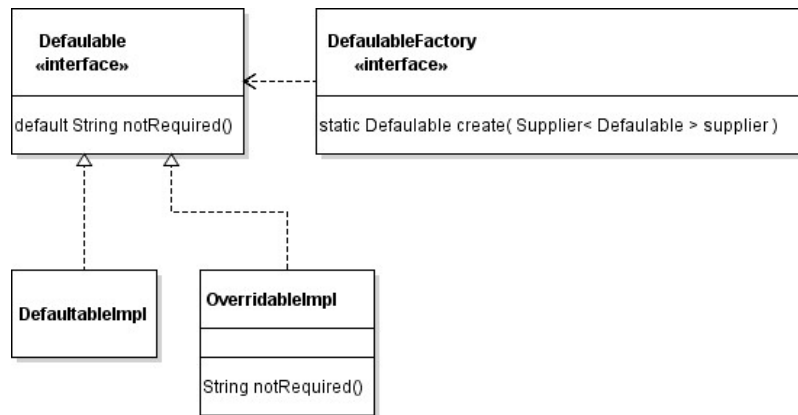
public class Etape3
{
    public static void main(String[] args)
    {
        List<String> names = Arrays.asList("Charlie Brown",
                                           "Snoopy",
                                           "Lucy",
                                           "Linus",
                                           "Woodstock");

        forEach(names, String::format);
        forEach(names, (fmt, arg) -> String.format(fmt, arg));
    }

    public static void forEach(List<String> list, Formatter formatter)
    {
        for (String item: list)
            System.out.print(formatter.format("%s\n", item));
        System.out.println();
    }
}
```

Des méthodes par défaut et des méthodes static peuvent être ajoutées à des interfaces existantes. Ainsi, toute classe qui implémentera une interface mais ne redéfinira pas toutes les méthodes, aura l'implémentation par défaut de celles qui ne sont pas redéfinies localement.

Etape4 : construire une classe Etape4 qui utilise une interface ayant des méthodes par défaut :



- définir une interface static `Defaultable` qui contient une méthode `notRequired` ayant une implémentation par défaut. Sa signature est telle qu'elle ne prend pas de paramètre en entrée et retourne une chaîne de caractères. Celle-ci sera la chaîne de votre choix.
- créer une classe static `DefaultableImpl` qui implémente l'interface précédente mais ne contient aucune méthode,
- faire une classe static `OverridableImpl` qui implémente l'interface précédente mais redéfinit la méthode nommée `notRequired`,
- créer une interface non static `DefaultableFactory` ayant une méthode static nommée `create` qui à partir d'un `Supplier` retourne un objet de classe `Defaultable`.
- une méthode `main` pour faire les points suivants :
 - o appeler la méthode `create` avec en paramètre une référence au `new` de la classe `DefaultableImpl`, pour avoir un objet respectant l'interface `Defaultable`,
 - o afficher le résultat de l'appel de la méthode `notRequired()` sur cet objet,
 - o appeler la méthode `create` avec en paramètre une référence au `new` de la classe `OverridableImpl`, pour avoir un objet respectant l'interface `Defaultable`,
 - o afficher le résultat de l'appel de la méthode `notRequired()` sur cet objet,

```
package fr.paris.java.exercice1;

import java.util.function.Supplier;

public class Etape4 {
    private interface Defaultable {
        // Interfaces now allow default methods, the implementer may or
        // may not implement (override) them.
        default String notRequired() {
            return "Default implementation";
        }
    }

    private static class DefaultableImpl implements Defaultable {
    }
}
```

```
private static class OverridableImpl implements Defaultable {
    @Override
    public String notRequired() {
        return "Overridden implementation";
    }
}

private interface DefaultableFactory {
    // Interfaces now allow static methods
    static Defaultable create( Supplier< Defaultable > supplier ) {
        return supplier.get();
    }
}

public static void main(String[] args) {
    Defaultable defaultable = DefaultableFactory.create( DefaultableImpl::new );
    System.out.println( defaultable.notRequired() );

    defaultable = DefaultableFactory.create( OverridableImpl::new );
    System.out.println( defaultable.notRequired() );
}
}
```

Exercice2 Utilisation des expressions lambda

Rappel : la méthode `compare` de l'interface `Comparator` doit retourner une valeur négative si la 1^{ère} entrée est plus petite que la 2^{ème} entrée et ainsi de suite. Elle retourne une valeur positive si la 1^{ère} entrée est plus grande que la 2^{ème}. Elle retourne 0 si les deux entrées sont égales.

Rappel : pour afficher un tableau, il n'est pas possible de le passer en paramètre directement à la méthode `println` de `PrintStream`. Aussi, il est conseillé de procéder à une transformation `System.out.println(Arrays.asList(tableau))` ; chaque élément du tableau sera séparé des autres par une virgule. De plus, c'est plus simple à écrire que de faire une boucle sur tous les éléments du tableau.

Etape1 : niveau de base

Construire une classe `Etape1` comprenant une méthode `main` où un tableau de `String` d'au moins 5 chaînes de caractères est créé et trié de plusieurs façons :

- Par longueur du plus petit au plus grand : utiliser la méthode `length()`,
- Par la propriété inverse (du plus grand au plus petit),
- Par ordre alphabétique sur le premier caractère (`charAt(0)` retourne le code numérique du 1^{er} caractère),
- Les chaînes de caractères qui contiennent un 'e' en premier, les autres en second.
- Dans ce dernier cas une seconde solution peut être fournie qui utilise une classe technique supplémentaire `StringUtils` qui possède une méthode statique `eChecker` avec 2 paramètres qui applique ce critère de tri. Une référence à cette méthode est ensuite utilisée.

```
package fr.paris.java.exercice2;

import java.util.Arrays;

public class Etape1 {
    public static void main(String[] args) {
        String[] words = { "Bart", "Homer", "Lisa", "Marge", "Maggie",
                           "Charles-Montgomery", "Carl", "Lenny" };
        System.out.println("Original array: " + Arrays.asList(words));

        Arrays.sort(words, (s1, s2) -> s1.length() - s2.length());
        System.out.println("Sorted by length ascending: " + Arrays.asList(words));

        Arrays.sort(words, (s1, s2) -> s2.length() - s1.length());
        System.out.println("Sorted by length descending : " + Arrays.asList(words));

        Arrays.sort(words, (s1, s2) -> s1.charAt(0) - s2.charAt(0));
        System.out.println("Sorted by first letter : " + Arrays.asList(words));

        Arrays.sort(words, (s1, s2) ->
            { int compareFlag = 0;
              if(s1.contains("e") && !s2.contains("e")) {
                  compareFlag = -1;
              }
              return compareFlag;
            });
    }
}
```

```

        } else if(s2.contains("e") && !s1.contains("e")) {
            compareFlag = 1;
        }
        return(compareFlag);
    });
    System.out.println("Sorted by whether it contains 'e' [v1] : " +
Arrays.asList(words));

    // Arrays.sort(words, (s1,s2) -> StringUtils.eChecker(s1,s2));
    Arrays.sort(words, StringUtils.eChecker);
    System.out.println("Sorted by whether it contains 'e' [v2] : " +
        Arrays.asList(words));
}
}

```

Etape2 : utilisation de lambda expressions dans des interfaces

Le but est de faire une méthode static appelée `betterString` dans la classe `StringUtils` qui prend 2 chaînes de caractères et une lambda expression qui désigne laquelle des deux est la meilleure sur un critère comme :

- a) La plus grande des deux en longueur,
- b) La première dans l'ordre alphabétique.

Pour effectuer ce travail, il est demandé de définir une interface fonctionnelle `TwoStringPredicate` qui possède une obligation de codage nommée `isBetter` qui prend 2 `String` et retourne `true` ou `false`. Cette interface est utilisée dans la déclaration de la méthode `betterString` pour typer la lambda expression.

```

package fr.paris.java.exercice2;

@FunctionalInterface
public interface TwoStringPredicate {
    public boolean isBetter(String s1, String s2);
}

```

```

package fr.paris.java.exercice2;

public class StringUtils {
    public static int eChecker(String s1, String s2) {
        int compareFlag = 0;
        if (s1.contains("e") && !s2.contains("e")) {
            compareFlag = -1;
        } else if (s2.contains("e") && !s1.contains("e")) {
            compareFlag = 1;
        }
        return(compareFlag);
    }
}

```

```

public static String betterString(String s1, String s2, TwoStringPredicate tester) {
    if (tester.isBetter(s1, s2)) {
        return(s1);
    } else {
        return(s2);
    }
}

private StringUtils() {} // Uninstantiable class
}

```

```

package fr.paris.java.exercice2;

public class Etape2 {
    public static void main(String[] args) {
        String test1 = "Hello";
        String test2 = "Goodbye";
        String message = "Better of %s and %s based on %s is %s.%n";
        String result1 = StringUtils.betterString(test1, test2,
            (s1, s2) -> s1.length() > s2.length());
        System.out.printf(message, test1, test2, "length", result1);
        String result2 = StringUtils.betterString(test1, test2, (s1, s2) -> true);
        System.out.printf(message, test1, test2, "1st arg", result2);
    }
}

```

Etape3 : utilisation de lambda expressions dans des interfaces génériques

Utiliser des interfaces génériques permet de remplacer plusieurs interfaces non génériques où des lambda expressions sont utilisées. Ainsi, il est demandé de remplacer la méthode `betterString` par une méthode `betterEntry`. Pour ce faire, il faut en premier définir une nouvelle interface générique `TwoElementPredicate` contenant une obligation de codage nommée `isBetter` qui est une méthode générique. Ensuite, il est demandé de construire une classe `ElementUtils` qui possède une méthode static générique à 3 paramètres nommée `betterEntry` :

- 2 paramètres du type de généricité,
- Une lambda expression typée par l'interface fonctionnelle `TwoElementPredicate`.

Enfin, il faut construire une classe `Etape3` basé sur la classe `Etape2` mais utilisant la méthode `betterEntry`.

```

package fr.paris.java.exercice2;

@FunctionalInterface
public interface TwoElementPredicate<T> {
    public boolean isBetter(T element1, T element2);
}

```



```

package fr.paris.java.exercice2;

public class ElementUtils {
    public static <T> T betterElement(T element1, T element2,
        TwoElementPredicate<T> tester) {
        if (tester.isBetter(element1, element2)) {
            return (element1);
        }
        return (element2);
    }

    private ElementUtils() {
    } // Uninstantiatable class
}

```

```

package fr.paris.java.exercice2;

public class Etape3 {
    public static void main(String[] args) {
        String test1 = "Hello";
        String test2 = "Goodbye";
        String message = "Better of %s and %s based on %s is %s.%n";
        String result1 = StringUtils.betterString(test1, test2,
            (s1, s2) -> s1.length() > s2.length());
        System.out.printf(message, test1, test2, "length", result1);
        String result2 = StringUtils.betterString(test1, test2, (s1, s2) -> true);
        System.out.printf(message, test1, test2, "1st arg", result2);
        String result3 = ElementUtils.betterElement(test1, test2,
            (s1, s2) -> s1.length() > s2.length());
        System.out.printf(message, test1, test2, "length", result3);
        String result4 = ElementUtils.betterElement(test1, test2, (s1, s2) -> true);
        System.out.printf(message, test1, test2, "1st arg", result4);
        int result5 = ElementUtils.betterElement(1, 2, (n1, n2) -> n1 > n2);
        System.out.printf(message, 1, 2, "numeric size", result5);
    }
}

```

Maintenant, vous pouvez comparer le prix de deux voitures. Pour cela, il est demandé de définir une classe Car avec 2 attributs : name et price et ajouter une ligne à votre classe Etape3 pour comparer deux objet de classe Car.

```

ElementUtils.betterElement(car1, car2, (c1, c2) -> c1.getPrice() > c2.getPrice());

```

Rappel : `Arrays.asList` est un moyen simple pour faire une liste, par exemple

```
List<String> firstnames = Arrays.asList("Jean", "Philippe", "Gérard");
```

Rappel : `List` a une méthode `toString()` très utile qui peut être utilisée sur tous les containers qui implémentent cette interface, contrairement aux tableaux.

```
System.out.println(firstnames) ;
```

Rappel : `Predicate` et `Function` se trouvent dans le package `java.util.function`.

Etape4 : lambda expression et prédicats génériques

Il est demandé de construire une méthode `static` nommée `allMatches` dans la classe `StringUtilsils`. Cette méthode prend en paramètre un prédicat (`Predicate<String>`) et retourne une nouvelle liste de toutes les valeurs qui ont passées le test avec succès. Construire une classe `Etape4` pour valider ce développement avec les traitements suivants :

```
List<String> shortWords = StringUtilsils.allMatches(words, s -> s.length() <= 4);
```

```
List<String> wordsWithB = StringUtilsils.allMatches(words, s -> s.contains("B"));
```

```
List<String> evenLengthWords = StringUtilsils.allMatches(words, s -> (s.length() % 2) == 0);
```

```
package fr.paris.java.exercice2;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;

public class StringUtilsils {
    // ... précédents développements
    public static List<String> allMatches(List<String> candidates,
        Predicate<String> matchFunction) {
        List<String> results = new ArrayList<>();
        for (String possibleMatch : candidates) {
            if (matchFunction.test(possibleMatch)) {
                results.add(possibleMatch);
            }
        }
        return (results);
    }

    private StringUtilsils() {
    } // Uninstantiatable class
}
```

```
package fr.paris.java.exercice2;
```

```

import java.util.Arrays;
import java.util.List;

public class Etape4 {
    public static void main(String[] args) {
        List<String> firstnames = Arrays.asList("Bart", "Homer", "Lisa", "Marge",
                                                "Maggie", "Charles-Montgomery", "Carl", "Lenny");
        System.out.printf("Original words: %s.%n", firstnames);
        List<String> shortWords = StringUtils.allMatches(firstnames,
                                                         s -> s.length() <= 4);
        System.out.printf("Short words: %s.%n", shortWords);
        List<String> wordsWithB = StringUtils.allMatches(firstnames,
                                                         s -> s.contains("B"));
        System.out.printf("B words: %s.%n", wordsWithB);
        List<String> evenLengthWords = StringUtils.allMatches(firstnames,
                                                             s -> (s.length() % 2) == 0);
        System.out.printf("Even-length words: %s.%n", evenLengthWords);
    }
}

```

Il est maintenant demandé de définir une nouvelle méthode static `allMatches` afin que d'autres types de listes soient pris en compte. Dans ce but il est demandé d'enrichir la classe `ElementUtils` avec une méthode static générique `allMatches` qui applique une lambda expression sur les données correspondantes à son type de généricité. Enfin modifier le code de la classe `Etape4` pour valider ces développements en ajoutant des appels à `ElementUtils` dont

```

List<Integer> nums = Arrays.asList(1, 10, 100, 1000, 10000);

List<Integer> bigNums = ElementUtils.allMatches(nums, n -> n>500);

```

```

package fr.paris.java.exercice2;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;

public class ElementUtils {
    // ... précédents développements
    public static <T> List<T> allMatches(List<T> candidates,
                                         Predicate<T> matchFunction) {
        List<T> results = new ArrayList<>();
        for (T possibleMatch : candidates) {
            if (matchFunction.test(possibleMatch)) {
                results.add(possibleMatch);
            }
        }
        return (results);
    }

    private ElementUtils() {

```

```
} // Uninstantiatable class  
}
```

```
package fr.paris.java.exercice2;  
  
import java.util.Arrays;  
import java.util.List;  
  
public class Etape4 {  
    public static void main(String[] args) {  
        List<String> firstnames = Arrays.asList("Bart", "Homer", "Lisa",  
            "Marge", "Maggie", "Charles-Montgomery", "Carl", "Lenny");  
        System.out.printf("Original words: %s.%n", firstnames);  
        List<String> shortWords = StringUtils.allMatches(firstnames,  
            s -> s.length() <= 4);  
        System.out.printf("Short words: %s.%n", shortWords);  
        List<String> wordsWithB = StringUtils.allMatches(firstnames,  
            s -> s.contains("B"));  
        System.out.printf("B words: %s.%n", wordsWithB);  
        List<String> evenLengthWords = StringUtils.allMatches(firstnames,  
            s -> (s.length() % 2) == 0);  
        System.out.printf("Even-length words: %s.%n", evenLengthWords);  
        List<String> shortWords2 = ElementUtils.allMatches(firstnames,  
            s -> s.length() <= 4);  
        System.out.printf("Short words: %s.%n", shortWords2);  
        List<String> wordsWithB2 = ElementUtils.allMatches(firstnames,  
            s -> s.contains("B"));  
        System.out.printf("B words: %s.%n", wordsWithB2);  
        List<String> evenLengthWords2 = ElementUtils.allMatches(firstnames,  
            s -> (s.length() % 2) == 0);  
        System.out.printf("Even-length words: %s.%n", evenLengthWords2);  
        List<Integer> nums = Arrays.asList(1, 10, 100, 1000, 10000);  
        List<Integer> bigNums = ElementUtils.allMatches(nums, n -> n > 500);  
        System.out.printf("Nums bigger than 500: %s.%n", bigNums);  
    }  
}
```

Etape5 : lambda expression et fonctions génériques

Modifier la classe `StringUtils` afin de lui ajouter une méthode `transformedList` static. Cette méthode prend en paramètre une liste de `String` et une fonction qui s'applique sur des `String`, enfin elle retourne une liste contenant le résultat des transformations des éléments de la liste passée en paramètres. Dans ce but, il est demandé de faire une classe `StringUtils` par l'ajout d'une méthode et de faire une classe `Etape5` avec une nouvelle méthode principale avec par exemple

```
List<String> excitingWords = StringUtils.transformedList(words, s -> s + "!");  
  
List<String> eyeWords = StringUtils.transformedList(words, s ->  
s.replace("is", "eyes"));
```

```
List<String> upperCaseWords = StringUtils.transformedList(words,
String::toUpperCase);
```

```
package fr.paris.java.exercice2;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;

public class StringUtils {
// ... précédents développements
    public static List<String> transformedList(List<String> originals,
        Function<String, String> transformer) {
        List<String> results = new ArrayList<>();
        for (String original : originals) {
            results.add(transformer.apply(original));
        }
        return (results);
    }

    private StringUtils() {
    } // Uninstantiatable class
}
```

```
package fr.paris.java.exercice2;

import java.util.Arrays;
import java.util.List;

public class Etape5 {
    public static void main(String[] args) {
        List<String> firstnames = Arrays.asList("Bart", "Homer", "Lisa",
            "Marge", "Maggie", "Charles-Montgomery", "Carl", "Lenny");
        System.out.printf("Original words: %s.%n", firstnames);
        List<String> excitingWords = StringUtils.transformedList(firstnames, s -> s
            + "!");
        System.out.printf("Exciting words: %s.%n", excitingWords);
        List<String> eyeWords = StringUtils.transformedList(firstnames,
            s -> s.replace("is", "eyes"));
        System.out.printf("Eye words: %s.%n", eyeWords);
        List<String> upperCaseWords = StringUtils.transformedList(firstnames,
            String::toUpperCase);
        // SAME AS List<String> upperCaseWords =
        // StringUtils.transformedList(words, s -> s.toUpperCase());
        System.out.printf("Uppercase words: %s.%n", upperCaseWords);
    }
}
```

De même que précédemment, il est demandé de promouvoir cette méthode `allMatches` afin qu'elle fonctionne non seulement sur des listes de `String` mais plus généralement sur des listes de n'importe

quoi. Dans ce but il est demandé d'ajouter une méthode générique `allMatches` à la classe `ElementUtils`. Cette méthode a deux paramètres de généricité : un 1^{er} paramètre `T` qui qualifie le type des données d'entrée et un 2^{ème} paramètre `R` qui qualifie le type des données résultat. De plus cette méthode prend en paramètres :

- Une liste d'objet de type de généricité `T`,
- Une fonction générique à deux paramètres de généricité `T, R`, qui s'applique à un élément de la liste et retourne une donnée de type `R`.

Modifier `Etape5` afin de mettre en valeur vos développements :

```
List<Integer> wordLengths = ElementUtils.transformedList(words,
String::length);
```

```
package fr.paris.java.exercice2;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;

public class ElementUtils {
    // ... précédents développements
    public static <T, R> List<R> transformedList(List<T> originals,
        Function<T, R> transformer) {
        List<R> results = new ArrayList<>();
        for (T original : originals) {
            results.add(transformer.apply(original));
        }
        return (results);
    }

    private ElementUtils() {
    } // Uninstantiatable class
}
```

```
package fr.paris.java.exercice2;

import java.util.Arrays;
import java.util.List;

public class Etape5 {
    public static void main(String[] args) {
        List<String> firstnames = Arrays.asList("Bart", "Homer", "Lisa",
            "Marge", "Maggie", "Charles-Montgomery", "Carl", "Lenny");
        System.out.printf("Original words: %s.%n", firstnames);
        List<String> excitingWords = StringUtils.transformedList(firstnames, s -> s
            + "!");
        System.out.printf("Exciting words: %s.%n", excitingWords);
        List<String> eyeWords = StringUtils.transformedList(firstnames,
            s -> s.replace("is", "eyes"));
    }
}
```

```

        System.out.printf("Eye words: %s.%n", eyeWords);
        List<String> upperCaseWords = StringUtils.transformedList(firstnames,
            String::toUpperCase);
        // SAME AS List<String> upperCaseWords =
        // StringUtils.transformedList(words, s -> s.toUpperCase());
        System.out.printf("Uppercase words: %s.%n", upperCaseWords);
        List<String> excitingWords2 = ElementUtils.transformedList(firstnames,
            s -> s + "!");
        System.out.printf("Exciting words: %s.%n", excitingWords2);
        List<String> eyeWords2 = ElementUtils.transformedList(firstnames,
            s -> s.replace("is", "eyes"));
        System.out.printf("Eye words: %s.%n", eyeWords2);
        List<String> upperCaseWords2 = ElementUtils.transformedList(firstnames,
            String::toUpperCase);
        System.out.printf("Uppercase words: %s.%n", upperCaseWords2);
        List<Integer> wordLengths = ElementUtils.transformedList(firstnames,
            String::length);
        System.out.printf("Word lengths: %s.%n", wordLengths);
    }
}

```

Rappel : Il est possible d’avoir des warnings à la compilation lorsque l’on utilise des fonctions avec un nombre variable de paramètre. Aussi l’annotation `@SafeVarargs` permet d’éliminer ces désagréments.

Etape6 : lambda expression et fonctions à nombre variable de paramètres

Faire une méthode appelée `allPassPredicate` qui accepte un nombre quelconque de prédicat typé et retourne un seul prédicat. Son but est de retourner un prédicat qui est le ET de tous les prédicats passés en paramètre. Si aucun prédicat n’est fourni alors il retourne un prédicat qui est une tautologie.

Faire une méthode appelée `firstAllMatch` qui prend un Stream et un nombre quelconque de prédicats typés et retourne la 1^{ère} donnée du flux qui est valide pour tous les prédicats. Par exemple, si `words` est une liste de String, la ligne suivante trouvera le 1^{er} mot qui contient à la fois un ‘o’ et a une longueur plus grande que 5.

```
FunctionUtils.firstAllMatch(words.stream(), s -> s.contains("o"),
    s -> s.length() > 5);
```

Faire une classe `Etape6` pour valider ces développements.

```

package fr.paris.java.exercice2;

import java.util.function.Predicate;
import java.util.stream.Stream;

public class FunctionUtils {

    /** Returns a Predicate that is the result of ANDing all the argument Predicates.
     *  If no Predicates are supplied, it returns a Predicate that always returns

```

```

    * true.
    */

    @SafeVarargs
    public static <T> Predicate<T> allPassPredicate(Predicate<T>... tests) {
        Predicate<T> result = e -> true;
        for(Predicate<T> test: tests) {
            result = result.and(test);
        }
        return(result);
    }

    /** Returns first element that matches all of the tests, null otherwise. */

    @SafeVarargs
    public static <T> T firstAllMatch(Stream<T> elements, Predicate<T>... tests) {
        Predicate<T> combinedTest = allPassPredicate(tests);
        return(elements.filter(combinedTest)
            .findFirst()
            .orElse(null));
    }

    private FunctionUtils() {} // Uninstantiatable class
}

```

```

package fr.paris.java.exercice2;

import java.util.*;

public class Etape6 {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("hello", "hola", "goodbye", "adios");

        String word1 =
            FunctionUtils.firstAllMatch(words.stream(),
                s -> s.contains("o"),
                s -> s.length() > 5);
        System.out.println("First word with o and 5+ letters is " + word1);

        String word2 =
            FunctionUtils.firstAllMatch(words.stream(),
                s -> s.contains("o"),
                s -> s.length() > 8);
        System.out.println("First word with o and 8+ letters is " + word2);
    }
}

```

Faire une fonction appelée `anyPassPredicate` qui accepte un nombre quelconque de prédicats types et retourne un seul prédicat qui teste si l'argument passé satisfait au moins un d'entre eux.

Faire une méthode appelée `firstAnyMatch` qui prend un flux de donnée et un nombre quelconque de prédicats typés et retourne la 1^{ère} entrée du flux qui satisfait au moins un prédicat. Par exemple, si `words` est une liste de `String`, la ligne suivante trouvera le 1^{er} mot qui soit contient un 'o', soit a une longueur plus grande que 5.

```
FunctionUtils.firstAnyMatch(words.stream(), s -> s.contains("o"), s -> s.length() > 5);
```

Compléter la classe `Etape6` pour valider ces développements.

```
package fr.paris.java.exercice2;

import java.util.function.Predicate;
import java.util.stream.Stream;

public class FunctionUtils {
    // ... précédents développements
    /** Returns a Predicate that is the result of ORing all the argument Predicates.
     *  * If no Predicates are supplied, it returns a Predicate that always returns
     *  * false.
     */
    @SafeVarargs
    public static <T> Predicate<T> anyPassPredicate(Predicate<T>... tests) {
        Predicate<T> result = e -> false;
        for(Predicate<T> test: tests) {
            result = result.or(test);
        }
        return(result);
    }

    /** Returns first element that matches any of the tests, null otherwise. */

    @SafeVarargs
    public static <T> T firstAnyMatch(Stream<T> elements, Predicate<T>... tests) {
        Predicate<T> combinedTest = anyPassPredicate(tests);
        return(elements.filter(combinedTest)
            .findFirst()
            .orElse(null));
    }

    private FunctionUtils() {} // Uninstantiatable class
}
```

```
package fr.paris.java.exercice2;

import java.util.*;

public class Etape6 {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("hello", "hola", "goodbye", "adios");
```

```
String word1 =
    FunctionUtils.firstAllMatch(words.stream(),
                                s -> s.contains("o"),
                                s -> s.length() > 5);
System.out.println("First word with o and 5+ letters is " + word1);

String word2 =
    FunctionUtils.firstAllMatch(words.stream(),
                                s -> s.contains("o"),
                                s -> s.length() > 8);
System.out.println("First word with o and 8+ letters is " + word2);

String word3 =
    FunctionUtils.firstAnyMatch(words.stream(),
                                s -> s.contains("o"),
                                s -> s.length() > 5);
System.out.println("First word with o or 5+ letters is " + word3);

String word4 =
    FunctionUtils.firstAnyMatch(words.stream(),
                                s -> s.contains("o"),
                                s -> s.length() > 8);
System.out.println("First word with o or 8+ letters is " + word4);

String word5 =
    FunctionUtils.firstAnyMatch(words.stream(),
                                s -> s.contains("q"),
                                s -> s.length() > 8);
System.out.println("First word with q or 8+ letters is " + word5);
}
}
```

Exercice3 Utilisation des streams

Toutes les étapes de cet exercice sont basées une liste de chaîne de caractères de la forme :

```
List<String> firstnames = Arrays.asList("Lisa", "Bart", "Maggie", "Marge", "Homer",  
"Apu", "Mandula", ...);
```

Etape1 : utilisations élémentaires

- Boucler sur les prénoms et imprimer chacun sur une ligne séparée, avec deux espaces devant chaque prénom.
- Répétez le problème précédent, mais sans les deux espaces devant. Le but est d'utiliser une référence de méthode ici, par opposition à un lambda explicite dans la question d'avant,
- Dans l'exercice précédent, nous avons produit des listes de transformations de ce genre

```
List<String> excitingWords = StringUtils.transformedList(words, s -> s + "!");
```

```
List<String> eyeWords = StringUtils.transformedList(words, s ->  
s.replace("is", "eyes"));
```

```
List<String> upperCaseWords = StringUtils.transformedList(words,  
String::toUpperCase);
```

Produire les mêmes listes que ci-dessus, mais cette fois avec des streams et la méthode «map»,

- Dans l'exercice précédent, nous avons produit les listes filtrées comme ceci:

```
List<String> shortWords = StringUtils.allMatches(words, s -> s.length() <= 4);
```

```
List<String> wordsWithB = StringUtils.allMatches(words, s -> s.contains("B"));
```

```
List<String> evenLengthWords = StringUtils.allMatches(words, s -> (s.length()  
% 2) == 0);
```

Produire les mêmes listes que ci-dessus, mais cette fois avec des streams et la méthode «filter»,

- Changez les chaînes en majuscules, ne conserver que ceux qui sont moins de 4 caractères, de ce qui est reste, ne conserver que celles qui contiennent "E", et d'imprimer le premier résultat. Répétez le processus, sauf vérification d'un «Q» au lieu d'un "E". Lors de la vérification du "Q", essayez d'éviter de répéter tout le code à partir de quand vous avez coché pour un "E".
- L'exemple ci-dessus utilise l'évaluation paresseuse, mais il n'est pas facile de voir ce qu'il fait. Faire une variante de l'exemple ci-dessus qui prouve qu'il est en train de faire l'évaluation paresseuse. Le plus simple est de faire le suivi des entrées qui sont transformés en majuscules.
- Prenez l'un des exemples précédents où une liste est produite, mais cette fois le résultat final doit être un tableau au lieu d'une liste.

```
package fr.paris.java.exercice3;  
  
import java.util.*;  
import java.util.function.*;
```

```

import java.util.stream.Collectors;
//import java.util.stream.Stream; Not needed. See commented-out line below.

public class Etapel {
    public static void main(String[] args) {
        List<String> words =
            Arrays.asList("Lisa", "Bart", "Maggie", "Marge", "Homer", "Apu", "Mandula");
        // Stream<String> wordStream = words.stream(); Then, reuse the Stream. NO!! Why not?
        System.out.println("Words (with spaces):");
        words.stream().forEach(s -> System.out.println("  " + s));
        System.out.println("Words (no spaces):");
        words.stream().forEach(System.out::println);
        List<String> excitingWords = words.stream()
            .map(s -> s + "!")
            .collect(Collectors.toList());
        System.out.printf("Exciting words: %s.%n", excitingWords);
        List<String> eyeWords = words.stream()
            .map(s -> s.replace("is", "eyes"))
            .collect(Collectors.toList());
        System.out.printf("Eye words: %s.%n", eyeWords);
        List<String> upperCaseWords = words.stream()
            .map(String::toUpperCase)
            // or .map(s -> s.toUpperCase())
            .collect(Collectors.toList());
        System.out.printf("Uppercase words: %s.%n", upperCaseWords);
        List<String> shortWords = words.stream()
            .filter(s -> s.length() <= 4)
            .collect(Collectors.toList());
        System.out.printf("Short words: %s.%n", shortWords);
        List<String> wordsWithB = words.stream()
            .filter(s -> s.contains("B"))
            .collect(Collectors.toList());
        System.out.printf("B words: %s.%n", wordsWithB);
        List<String> evenLengthWords = words.stream()
            .filter(s -> (s.length() % 2) == 0)
            .collect(Collectors.toList());
        System.out.printf("Even-length words: %s.%n", evenLengthWords);
        String result1 = firstFunnyString(words, "E");
        System.out.println("Uppercase short word with 'E': " + result1);
        String result2 = firstFunnyString(words, "Q");
        System.out.println("Uppercase short word with 'Q': " + result2);
        Function<String, String> toUpper = s -> { System.out.println("Uppercasing " + s);
            return(s.toUpperCase());
        };
        String result3 = words.stream()
            .map(toUpper)
            .filter(s -> s.length() < 4)
            .filter(s -> s.contains("E"))
            .findFirst().orElse("No match");
        System.out.println("Uppercase short word with 'E': " + result3);
        String[] excitingWords2 = words.stream()
            .map(s -> s + "!")
            .toArray(String[]::new);
        System.out.printf("Exciting words as array: %s.%n", Arrays.asList(excitingWords2));
    }
}

```

```

public static String firstFunnyString(List<String> words, String containedTest) {
    String result =
        words.stream()
            .map(String::toUpperCase)
            .filter(s -> s.length() < 4)
            .filter(s -> s.contains(containedTest))
            .findFirst().orElse("No match");
    return(result);
}
}

```

Étape2 : utilisations avancées

- Produire une seule chaîne qui est le résultat de la concaténation des versions majuscules de toutes les chaînes. Utilisez une seule opération de réduction, sans utiliser de map,
- Produire la même chaîne que ci-dessus, mais cette fois par une opération de map qui transforme les mots en majuscules, suivie d'une opération de réduction qui les enchaîne.
- Produire une chaîne qui est tous les mots enchaînés ensemble, mais avec des virgules entre les deux. Notez qu'il n'y a pas de virgule au début, avant le 1^{er} prénom, et pas de virgule à la fin, après le dernier prénom. Indice majeur: il existe deux versions de réduction: l'une avec une valeur de départ, et l'autre sans une valeur de démarrage.
- Faites une méthode statique dans une classe `StreamUtils` qui produit une liste d'une longueur spécifiée par nombres aléatoires. Par exemple:

```

List<Double> nums = StreamUtils.randomNumberList(someSize);

// Result is something like [0.7096867136897776, 0.09894202723079482, ...]

```

- Faites une méthode statique dans la classe `StreamUtils` qui produit une liste de numéros qui vont dans l'ordre par une taille de pas. Par exemple :

```

List<Integer> nums = StreamUtils.orderedNumberList(50, 5, someSize);

// Result is [50, 55, 60, ...]

```

- Utilisez les streams, et calculez la somme de quelques entiers. Refaire en parallèle et vérifiez que vous obtenez la même réponse à chaque fois.
- Maintenant, utilisez les flux de même pour calculer le produit de quelques doubles. Il est nécessaire d'ajouter une méthode statique à `StreamUtils`. Montrez que les versions série et parallèle ne donnent pas toujours la même réponse.

```

package fr.paris.java.exercice3;

import java.util.*;

public class Etape2 {
    public static void main(String[] args) {

```

```

List<String> words = Arrays.asList("Lisa", "Bart", "Maggie", "Marge", "Homer",
"Apu", "Mandula");
System.out.printf("Original words: %s.%n", words);
String upperCaseWords = words.stream().reduce("",
    (s1, s2) -> s1.toUpperCase() + s2.toUpperCase());
System.out.printf("Single uppercase String: %s.%n", upperCaseWords);
String upperCaseWords2 = words.stream().map(String::toUpperCase)
    .reduce("", String::concat);
System.out.printf("Single uppercase String: %s.%n", upperCaseWords2);
String wordsWithCommas = words.stream()
    .reduce((s1, s2) -> s1 + "," + s2)
    .orElse("need at least two strings");
System.out.printf("Comma-separated String: %s.%n", wordsWithCommas);
System.out.printf("3 random nums: %s.%n",
    StreamUtils.randomNumberList(3));
System.out.printf("10 numbers starting at 50, by 5's: %s.%n",
    StreamUtils.orderedNumberList(50, 5, 10));
List<Integer> nums1 = Arrays.asList(1, 2, 3, 4, 5);
int sum1 = nums1.stream().reduce(0, Integer::sum);
System.out.printf("Serial sum of %s is %s.%n", nums1, sum1);
int sum2 = nums1.stream().parallel().reduce(0, Integer::sum);
System.out.printf("Parallel sum of %s is %s.%n", nums1, sum2);
System.out.println(Double.MIN_VALUE);
List<Double> nums2 = Arrays.asList(0.0000000000001, 200000000000.0,
    0.0000000000003, 400000000000.0);
double product1 = nums2.stream().reduce(1.0, (n1, n2) -> n1 * n2);
System.out.printf("Serial product of %s is %s.%n", nums2, product1);
double product2 = nums2.stream().parallel()
    .reduce(1.0, (n1, n2) -> n1 * n2);
System.out.printf("Parallel product of %s is %s.%n", nums2, product2);
}
}

```

```

package fr.paris.java.exercice3;

import java.util.*;
import java.util.stream.*;

public class StreamUtils {
    public static Stream<Double> randomNumberStream(int size) {
        return (Stream.generate(() -> Math.random()).limit(size));
    }

    public static List<Double> randomNumberList(int size) {
        return (randomNumberStream(size).collect(Collectors.toList()));
    }

    public static Stream<Integer> orderedNumberStream(int initialNum,
        int stepSize, int size) {
        return (Stream.iterate(initialNum, n -> n + stepSize).limit(size));
    }

    public static List<Integer> orderedNumberList(int initialNum, int stepSize,
        int size) {
    }
}

```

```
        return (orderedNumberStream(initialNum, stepSize, size)
                .collect(Collectors.toList()));
    }

    private StreamUtils() {
    } // Uninstantiatable class
}
```

Exercice4 Utilisation des IO et Files

Dans le cadre de cet exercice 4, il est demandé de prendre en compte un fichier texte nommé enable1-word-list.txt comme support. D'autres peuvent être fabriqués par les participants mais la tâche est fastidieuse.

Etape1 : utilisations de streams avec les fichiers

Ecrire une classe `fr.paris.java.exercice4.Etape1`,

- Imprimez le premier mot de 10 lettres trouvées dans le fichier. Pour ce faire il est souhaitable de faire une méthode static `nLetterWord` dans une classe `WordUtils`,
- Imprimez le premier mot de 6 lettres qui contient "a", "b" et "c". Pour ce faire il est souhaitable de faire une méthode static `abcWord` dans la classe `WordUtils`,
- Répétez le problème précédent, mais aussi ajouter la possibilité de casse mixte des mots dans le fichier. Astuce: faire quelque chose plus simple que de simplement ajouter trois tests de filtrage supplémentaires (pour "A", "B" et "C").
- Définir une méthode statique `isOoWord` dans la classe `WordUtils`, qui renvoie vrai seulement pour des mots qui ont au moins deux o consécutifs. Compte tenu de cette méthode, imprimer le premier mot qui a six ou plusieurs lettres, contient un "b", et est un mot oo. Pour ce faire il est souhaitable de faire une méthode static `isOoWord` dans une classe `StringUtils`,
- Faire un fichier appelé "twitter-words.txt" qui contient tous les mots de la liste de enable1 qui contiennent "wow" ou "cool". Les mots doivent être triés, en majuscules, et avoir un point à la fin d'exclamation. (Par exemple, "COOLER!"). Pour ce faire il est souhaitable de faire une méthode static `storeTwitterList` dans une classe `WordUtils`,
- Imprimez le nombre de fichiers dans votre projet. Les dossiers comptent comme des fichiers. Pour ce faire il est souhaitable de faire une méthode static `numPathsInTree` dans une classe `FolderUtils`,
- Créez un fichier contenant 17 doubles aléatoires entre 0 et 100, chacun avec exactement trois chiffres après la virgule. Pour ce faire il est souhaitable de faire une méthode static `writeNums` dans une classe `WritingUtils`,

```
package fr.paris.java.exercice4;

import java.util.*;
import java.util.function.*;

public class Etape1 {
    public static void main(String[] args) {
        String filename = "enable1-word-list.txt";

        List<String> testWords = Arrays.asList("foo", "bar", "baz1234567", "boo1234567");
        String s0 = WordUtils.nLetterWord(testWords.stream(), 10);
        System.out.printf("First 10-letter word in test words is '%s'.%n", s0);

        String s1 = WordUtils.nLetterWord(filename, 10);
        System.out.printf("First 10-letter word in file is '%s'.%n", s1);
    }
}
```



```

String s2 = WordUtils.abcWord(filename);
System.out.printf(
    "First 8-letter word with a, b, and c is '%s'. [approach 1]%n",      s2);

Predicate<String> abcTest = (word -> word.length() == 8);
abcTest = abcTest.and(word -> word.contains("a"))
    .and(word -> word.contains("b"))
    .and(word -> word.contains("c"));
String s3 = WordUtils.firstMatchingWord(filename, abcTest);
System.out.printf(
    "First 8-letter word with a, b, and c is '%s'. [approach 2]%n",      s3);

String s4 = WordUtils.abcWord2(filename);
System.out.printf(
    "First 8-letter word with a, b, and c is '%s'. [approach 3]%n",      s4);

String s5 = WordUtils.oWord(filename);
System.out.printf("First 6-letter O-word word with b is '%s'.%n", s5);

WordUtils.storeTwitterList(filename, "twitter-words.txt");

long n = FolderUtils.numPathsInTree(".");
System.out.printf("There are %s files in the project.%n", n);

String numberFilename = "random-nums.txt";
int numRandomNums = 17;
WritingUtils.writeNums(numberFilename, numRandomNums);
System.out.printf("Wrote %s random numbers to %s.%n", numRandomNums,
    numberFilename);
}
}

```

```

package fr.paris.java.exercice4;

import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class WordUtils {
    public static String nLetterWord(Stream<String> words, int n) {
        return(words.filter(word -> word.length() == n)
            .findFirst()
            .orElse(null));
    }

    public static String nLetterWord(String filename, int n) {
        try(Stream<String> words = Files.lines(Paths.get(filename))) {
            return(nLetterWord(words, n));
        } catch(IOException ioe) {
            System.err.println("IOException: " + ioe);
        }
    }
}

```

```

        return(null);
    }
}

/** One solution for finding the first 8-letter word with a, b, c,
 *  is to hardcode the tests.
 */
public static String abcWord(Stream<String> words) {
    return(words.filter(word -> word.length() == 8)
        .filter(word -> word.contains("a"))
        .filter(word -> word.contains("b"))
        .filter(word -> word.contains("c"))
        .findFirst()
        .orElse(null));
}

public static String abcWord(String filename) {
    try(Stream<String> words = Files.lines(Paths.get(filename))) {
        return(abcWord(words));
    } catch(IOException ioe) {
        System.err.println("IOException: " + ioe);
        return(null);
    }
}

/** Another solution for finding the first 8-letter word with a, b, c,
 *  is to just return the first word matching a test, and then
 *  pass in the AND of four tests (length 8, contains a, contains b, contains c).
 */
public static String firstMatchingWord(Stream<String> words, Predicate<String> test)
{
    return(words.filter(test)
        .findFirst()
        .orElse(null));
}

public static String firstMatchingWord(String filename, Predicate<String> test) {
    try(Stream<String> words = Files.lines(Paths.get(filename))) {
        return(firstMatchingWord(words, test));
    } catch(IOException ioe) {
        System.err.println("IOException: " + ioe);
        return(null);
    }
}

/** Like abcWord, but handles the possibility of mixed-case words
 *  in file by mapping to uppercase before testing.
 */
public static String abcWord2(Stream<String> words) {
    return(words.map(String::toLowerCase)
        .filter(word -> word.length() == 8)
        .filter(word -> word.contains("a"))
        .filter(word -> word.contains("b"))
        .filter(word -> word.contains("c"))
        .findFirst()

```

```

        .orElse(null));
    }

    public static String abcWord2(String filename) {
        try(Stream<String> words = Files.lines(Paths.get(filename))) {
            return(abcWord2(words));
        } catch(IOException ioe) {
            System.err.println("IOException: " + ioe);
            return(null);
        }
    }

    public static String oWord(Stream<String> words) {
        return(words.filter(StringUtils::isOoWord)
            // .filter(word -> Stringutils.isOoWord(word))
            .filter(word -> word.length() == 6)
            .findFirst()
            .orElse(null));
    }

    public static String oWord(String filename) {
        try(Stream<String> words = Files.lines(Paths.get(filename))) {
            return(oWord(words));
        } catch(IOException ioe) {
            System.err.println("IOException: " + ioe);
            return(null);
        }
    }

    /** Given a Stream of words, returns sorted uppercase List of
     *  words that contain "cool" or "wow". List is in uppercase
     *  with "!" at the end.
     */
    public static List<String> makeTwitterList(Stream<String> words) {
        return(words.filter(word -> word.contains("wow") || word.contains("cool"))
            .map(String::toUpperCase)
            .map(word -> word + "!")
            .sorted()
            .collect(Collectors.toList()));
    }

    /** Given a file of words, creates new file of twitter words
     *  (i.e., words containing "wow" or "cool", in uppercase with
     *  "!" at the end).
     */
    public static void storeTwitterList(String inputFile, String outputFile) {
        try(Stream<String> words = Files.lines(Paths.get(inputFile))) {
            List<String> twitterWords = makeTwitterList(words);
            Path outputPath = Paths.get(outputFile);
            Files.write(outputPath, twitterWords, Charset.defaultCharset());
            System.out.printf("Wrote %s words to %s.%n", twitterWords.size(),
outputPath.toAbsolutePath());
        } catch(IOException ioe) {
            System.err.println("IOException: " + ioe);
        }
    }

```

```
}

private WordUtils() {} // Uninstantiatable class
}
```

```
package fr.paris.java.exercice4;

public class StringUtils {

    public static boolean isOoWord(String word) {
        return(word.contains("oo"));
    }

    private StringUtils() {} // Uninstantiatable class
}
```

```
package fr.paris.java.exercice4;

import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;

public class WritingUtils {

    /** Writes n random numbers between 0-10 to given filename.
     * Uses 2 digits after decimal point.
     */
    public static void writeNums(String filename, int n) {
        Charset characterSet = Charset.defaultCharset();
        Path path = Paths.get(filename);
        try (PrintWriter writer =
            new PrintWriter(Files.newBufferedWriter(path, characterSet))) {
            for(int i=0; i<n; i++) {
                writer.printf("%.2f%n", 10 * Math.random());
            }
        } catch (IOException ioe) {
            System.err.println("IOException: " + ioe);
        }
    }
}
```

```
package fr.paris.java.exercice4;

import java.io.*;
import java.nio.file.*;
import java.util.stream.*;

public class FolderUtils {
    public static long numPathsInTree(String rootFolder) {
        try(Stream<Path> paths = Files.walk(Paths.get(rootFolder))) {
            return(paths.count());
        }
    }
}
```

```
    } catch(IOException ioe) {  
        System.err.println("IOException: " + ioe);  
        return(-1);  
    }  
}  
}
```

Exercice5 : les nouveautés de l'API Date

Ce chapitre porte sur la nouvelle API des Date et Time. Les classes les plus importantes sont :

Class	Description
<code>LocalDate</code>	Représente une date sans time
<code>LocalDateTime</code>	Gère à la fois date et time
<code>Period</code>	C'est une date basée sur une quantité de temps en termes d'année
<code>Duration</code>	Mesure le temps en secondes ou en nano secondes
<code>ChronoUnit</code>	Définit les unités utilisées en termes de temps. Par exemple, si vous avez besoin de trouver le nombre de jours entre 2 dates, <code>ChronoUnit.DAYS.between(localdate1, localdate2)</code>
<code>DateTimeFormatter</code>	Formatent pour l'affichage ou l'analyse d'objets de type date-time

Etape1 : construire une classe `Etape1` avec des essais premiers :

- Quelle classe utiliseriez-vous pour stocker votre anniversaire en années, mois, jour, secondes et nanosecondes?

Très probablement, vous souhaitez utiliser la classe `LocalDateTime`. Pour prendre un fuseau horaire particulier en compte, vous devez utiliser la classe `ZonedDateTime`. Les 2 classes date et l'heure à la nanoseconde précision et les deux classes piste, lorsqu'il est utilisé en conjonction avec la période, donner un résultat en utilisant une combinaison d'unités humaines base, comme année, mois et jour.

- Étant donné une date aléatoire, comment voulez-vous trouver la date du précédent jeudi?

Vous pouvez utiliser la méthode `previous` d'un `TemporalAdjuster`:

```
LocalDate date = ...;

System.out.printf("The previous Thursday is: %s%n",
    date.with(TemporalAdjuster.previous(DayOfWeek.THURSDAY)));
```

- Quelle est la différence entre un `ZoneId` et une `ZoneOffset`?

Les 2 suivent un décalage par rapport à Greenwich / temps UTC, mais `ZoneOffset` suit seulement le décalage absolu de Greenwich / UTC. La classe `ZoneId` utilise également les `ZoneRules` pour déterminer comment un décalage varie d'un moment particulier de l'année et par région.

- Comment voulez-vous convertir un `Instant` en `ZonedDateTime`? Comment voulez-vous convertir un `ZonedDateTime` en `Instant`?

Vous pouvez convertir un Instant en ZonedDateTime en utilisant la méthode

`ZonedDateTime.ofInstant`. Vous aurez besoin de fournir un `ZoneId`:

```
ZonedDateTime zdt = ZonedDateTime.ofInstant(Instant.now(),
ZoneId.systemDefault());
```

Autrement vous pouvez aussi utiliser la méthode `Instant.atZone`:

```
ZonedDateTime zdt = Instant.now().atZone(ZoneId.systemDefault());
```

Vous pouvez utiliser la méthode `toInstant` **de l'interface** `ChronoZonedDateTime`, **implémentée par la classe** `ZonedDateTime`, **pour convertir depuis un** `ZonedDateTime` **en** `Instant`:

```
Instant inst = ZonedDateTime.now().toInstant();
```

Etape2 : construire une classe `Etape2` avec des essais avancés :

- Ecrire un exemple qui, pour une année donnée, rapporte la longueur de chaque mois au sein de cette année. Pour ce faire, il est demandé de faire une classe `MonthsInYear` avec une méthode `static display`,

```
package fr.paris.java.exercice5;
/**
 * Display the numnber of days in each month of the specified year.
 */
import java.time.Month;
import java.time.Year;
import java.time.YearMonth;
import java.time.DateTimeException;

import java.io.PrintStream;
import java.lang.NumberFormatException;

public class MonthsInYear {
    public static void main(String[] args) {
        int year = 0;

        if (args.length <= 0) {
            System.out.printf("Usage: MonthsInYear <year>%n");
            throw new IllegalArgumentException();
        }

        try {
            year = Integer.parseInt(args[0]);
        } catch (NumberFormatException nexc) {
            System.out.printf("%s is not a properly formatted number.%n",
                args[0]);
            throw nexc;        // Rethrow the exception.
        }
    }
}
```

```

    }

    try {
        Year test = Year.of(year);
    } catch (DateTimeException exc) {
        System.out.printf("%d is not a valid year.%n", year);
        throw exc;    // Rethrow the exception.
    }

    System.out.printf("For the year %d:%n", year);
    for (Month month : Month.values()) {
        YearMonth ym = YearMonth.of(year, month);
        System.out.printf("%s: %d days%n", month, ym.lengthOfMonth());
    }
}
}

```

- Ecrire un exemple qui, pour un mois donné de l'année courante, répertorie tous les lundis dans ce mois. Pour ce faire, il est demandé de faire une classe `ListMondays` avec une méthode `static display`,

```

package fr.paris.java.exercice5;
/**
 * Display all of the Mondays in the current year and the specified month.
 */
import java.time.Month;
import java.time.Year;
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.DateTimeException;

import java.time.temporal.TemporalAdjuster;
import java.time.temporal.TemporalAdjusters;

import java.io.PrintStream;
import java.lang.NumberFormatException;

public class ListMondays {
    public static void main(String[] args) {
        Month month = null;

        if (args.length < 1) {
            System.out.printf("Usage: ListMondays <month>%n");
            throw new IllegalArgumentException();
        }

        try {
            month = Month.valueOf(args[0].toUpperCase());
        } catch (IllegalArgumentException exc) {
            System.out.printf("%s is not a valid month.%n", args[0]);
            throw exc;    // Rethrow the exception.
        }
    }
}

```



```

        System.out.printf("For the month of %s:%n", month);
        LocalDate date = Year.now().atMonth(month).atDay(1).
            with(TemporalAdjusters.firstInMonth(DayOfWeek.MONDAY));
        Month mi = date.getMonth();
        while (mi == month) {
            System.out.printf("%s%n", date);
            date = date.with(TemporalAdjusters.next(DayOfWeek.MONDAY));
            mi = date.getMonth();
        }
    }
}

```

- Ecrire un exemple qui teste si une date donnée est un vendredi 13. Pour ce faire, il est demandé de faire une classe `FridayThirteenQuery` avec une méthode static `queryFrom(TemporalAccessor date)`. Puis, il est demandé d'ajouter une classe `Superstitious` avec une méthode `isDangerous` ?

```

package fr.paris.java.exercice5;
import java.time.Month;
import java.time.Year;
import java.time.LocalDate;
import java.time.DateTimeException;

import java.time.temporal.TemporalQuery;
import java.time.temporal.TemporalAccessor;
import java.time.temporal.ChronoField;

import java.io.PrintStream;
import java.lang.Boolean;

public class FridayThirteenQuery implements TemporalQuery<Boolean> {

    // Returns TRUE if the date occurs on Friday the 13th.
    public Boolean queryFrom(TemporalAccessor date) {

        return ((date.get(ChronoField.DAY_OF_MONTH) == 13) &&
            (date.get(ChronoField.DAY_OF_WEEK) == 5));
    }
}

```

```

package fr.paris.java.exercice5;
import java.time.Month;
import java.time.Year;
import java.time.LocalDate;
import java.time.DayOfWeek;
import java.time.DateTimeException;

import java.time.temporal.TemporalQuery;
import java.time.temporal.TemporalAccessor;

import java.io.PrintStream;

```

```
public class Superstitious {

    public static void main(String[] args) {
        Month month = null;
        LocalDate date = null;

        if (args.length < 2) {
            System.out.printf("Usage: Superstitious <month> <day>%n");
            throw new IllegalArgumentException();
        }

        try {
            month = Month.valueOf(args[0].toUpperCase());
        } catch (IllegalArgumentException exc) {
            System.out.printf("%s is not a valid month.%n", args[0]);
            throw exc;        // Rethrow the exception.
        }

        int day = Integer.parseInt(args[1]);

        try {
            date = Year.now().atMonth(month).atDay(day);
        } catch (DateTimeException exc) {
            System.out.printf("%s %s is not a valid date.%n", month, day);
            throw exc;        // Rethrow the exception.
        }

        System.out.println(date.query(new FridayThirteenQuery()));
    }
}
```

Exercice6 : le nouveau moteur Javascript Nashorn

Nashorn est le successeur de Rhino, le moteur JavaScript créé par Mozilla. Il est basé sur l'ECMAScript-262 et est implémenté entièrement en Java en utilisant – entre autre – invokedynamic (JSR 292), une nouvelle instruction introduite avec Java 7 notamment pour les lambdas et pour les langages dynamiques.

Etape1 : comment appeler du JavaScript depuis du code Java et inversement:

- créer une commande externe pour utiliser `jdk/bin/jjs` et connaître la version de Nashorn,

```
jjs -version
```

`jjs` permet d'exécuter des fichiers `.js` sans avoir à les appeler directement depuis du code Java (bien que ce soit ce que fait `jjs`). `jjs` est aussi un interpréteur JavaScript en ligne de commande. Quand le prompt `jjs` est actif et il faut saisir `quit()` pour le stopper.

- Il est demandé de saisir le code Javascript suivant lorsque le prompt `jjs` est actif.

```
jjs> var x = 10, y = 5
jjs> var z = x + y
jjs> z
15
jjs> print(z)
10
jjs> quit()
```

- On s'intéresse à l'appelle de code Javascript depuis du Java. Ecrire une méthode `nashornString` static dans la classe `Etape1` dont le but est d'imprimer la somme de 15 et 10 en JavaScript, puis modifier la méthode principale de la classe.
Pour récupérer une instance de `ScriptEngine`, il est aussi possible de passer "JavaScript" or "ECMAScript" en paramètre de la méthode `getEngineByName()`, à la place de "nashorn". Le résultat sera identique. La méthode `eval()` quant à elle évalue la chaîne de caractère et l'exécute.

```
public static void nashornString() throws ScriptException {
    final ScriptEngineManager factory = new ScriptEngineManager();
    final ScriptEngine engine = factory.getEngineByName("nashorn");
    engine.eval("print(15 + 10)");
}
```

- Bien évidemment la méthode `eval()` ne sera pas utilisée avec des chaînes de caractère que sur des cas très particuliers. La solution à privilégier est d'avoir le code JavaScript dans un fichier séparé. Il est demandé de faire un fichier `simple.js` qui contient la fonction `sum` de deux entiers. Puis ajouter une méthode static `nashornFile()` à la classe `Etape1`, elle interprète

le fichier simple.js.

Il est demandé de lire depuis le code Java une variable déclarée en JavaScript, puis il est demandé d'invoquer la méthode `sum` depuis le corps de la fonction

```
/*
 * simple.js
 */
var myVariable = "jsVariable";

function sum(a, b) {
    return a + b;
}
```

```
public static void nashornFile() throws ScriptException, NoSuchMethodException {
    final ScriptEngineManager factory = new ScriptEngineManager();
    final ScriptEngine engine = factory.getEngineByName("nashorn");

    // Build a Reader
    final InputStream inputStream =
Thread.currentThread().getContextClassLoader().getResourceAsStream("simple.js");
    final InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
    engine.eval("var mySecondeVariable = 10");
    engine.eval(inputStreamReader);

    // Get a variable from a JavaScript file
    System.out.println("jsVariable = " + engine.get("myVariable"));

    // Invoke a function from a JavaScript file
    final Invocable invocable = (Invocable)engine;
    final int sum = (Integer)invocable.invokeFunction("sum", 30, 20);
    System.out.println("50 == " + sum);

    // Get a variable
    System.out.println("new Integer(10) " + engine.get("mySecondeVariable"));
}
```

- On s'intéresse maintenant à l'appel de Java depuis du Javascript. Il est demandé d'ajouter la méthode suivante à la classe `Etape1`. Appeler cette méthode depuis la méthode principale.

```
public void timerTask() throws InterruptedException {
    final TimerTask task = new TimerTask() {
        @Override
        public void run() {
            System.out.println("Hello!");
        }
    };

    new Timer().schedule(task, 0, 1000);
    Thread.sleep(10000);
    task.cancel();
}
```

L'objet `Packages` a les propriétés `java`, `javax` et `javafx`, permettant de simuler l'utilisation de packages telle que nous pourrions le faire en Java. Il est demandé d'écrire un nouveau script JavaScript nommé `jjs_Packages.js` qui fait la même chose que cette méthode et qui sera exécuté depuis le prompt `jjs`. Effectuer un test avec `jjs`.

```
// Create a variable java to simulate the utilization of package before each class.
// Classes from java.lang need this prefix too.
// Packages.java et Packages.javafx is available too.
var java = Packages.java;

var task = new java.util.TimerTask() {
  run: function() {
    print('Hello!');
  }
}

new java.util.Timer().schedule(task, 0, 1000);
java.lang.Thread.sleep(10000);
task.cancel();
```

- Au lieu de répéter `java.util` et `java.lang` de l'exemple précédent, il est possible de simuler l'import de packages et assigner à des variables JavaScript le chemin complet des différentes classes utilisées dans le code. Il est demandé de réécrire un nouveau script nommé `jjs_ShortPackages.js`. Effectuer un test avec `jjs`.

```
// Simulates java imports at the top of a class.
var javaPackage = Packages.java;
var TimerTask = java.util.TimerTask;
var Timer = java.util.Timer;
var Thread = java.lang.Thread;

var task = new TimerTask() {
  run: function() {
    print('Hello!');
  }
}

new Timer().schedule(task, 0, 1000);
Thread.sleep(10000);
task.cancel();
```

- Outre l'objet `Packages`, il y a l'objet `JavaImporter` qui permet d'indiquer les packages auxquels appartiennent les classes utilisées dans le code, tout comme par exemple `java.util.*`. Il est demandé de réécrire un nouveau script nommé `jjs_JavaImporter.js`. Effectuer un test avec `jjs`.

```
// Create an object containing all the packages from where the classes
// we are going to use in the program are defined.
var java = new JavaImporter(java.lang, java.util);
```

```
// All the code included in this block doesn't need to indicate the packages explicitly
// if defined in JavaImporter of course.
with(java) {
    var task = new TimerTask() {
        run: function() {
            print('Hello!');
        }
    }

    new Timer().schedule(task, 0, 1000);
    Thread.sleep(10000);
    task.cancel();
}
```

- Le JavaScript permettant d'utiliser des fonctions anonymes – sans la contrainte d'interfaces fonctionnelles – il est possible de remplacer la redéfinition de la méthode `run()` de la classe `TimerTask` – qui est une classe abstraite – par une fonction anonyme. Il est demandé de réécrire un nouveau script nommé `jjs_ShortPackagesAndClosures.js`. Effectuer un test avec `jjs`.

```
// Simulates java imports at the top of a class.
var javaPackage = Packages.java;
var TimerTask = java.util.TimerTask;
var Timer = java.util.Timer;
var Thread = java.lang.Thread;

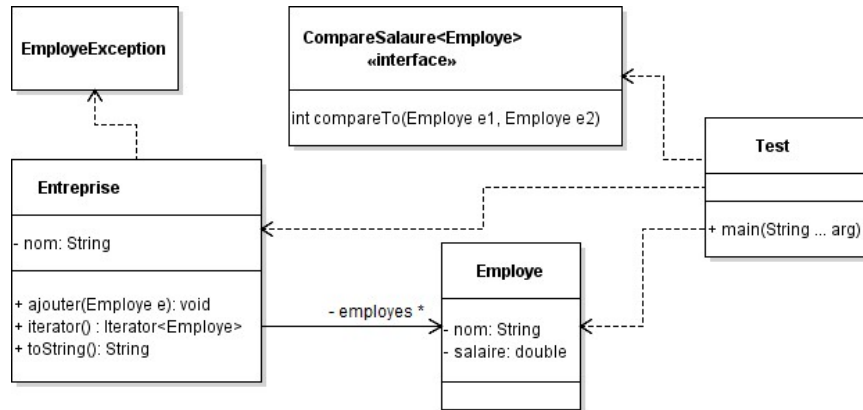
var timer = new Timer();

// JavaScript allows anonymous function the use of a Functional Interface.
// Therefore we can simply all the preceding examples as follow.
// TimerTask being an abstract class, the following code wouldn't be valid in Java.
timer.schedule(
    function() {
        print('Hello!');
    }, 0, 1000);

Thread.sleep(10000);
timer.cancel();
```

Exercice7 : code refactoring

Soit le code ci-dessous concernant l'utilisation de collections Java. Nous disposons d'une classe `Employe`, `Entreprise` ainsi que d'une classe de comparaison de salaire notée `CompareSalaire`, une classe d'exception `EmployeeException` et d'un jeu de test `Test`.



```
package fr.paris.java.exercice7;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Test {
    public static void main(String[] args) throws EmployeeException {
        Entreprise e1 = new Entreprise("LO");
        e1.ajouter(new Employee("Cyril", 5000));
        e1.ajouter(new Employee("Laurent", 6000));
        e1.ajouter(new Employee("Rima", 5700));
        e1.ajouter(new Employee("Victorien", 4300));
        System.out.println(e1);

        // Remplit une liste avec les employés
        List<Employee> l = new ArrayList<>();
        for (Employee e : e1) {
            l.add(e);
        }

        // Tri par salaires croissants
        CompareSalaire compareur = new CompareSalaire();
        Collections.sort(l, compareur);
        System.out.println("Employés de " + e1.getNom()
            + " par ordre croissant des salaires");
        for (Employee employee : l) {
            System.out.println(employee.getNom() + " gagne "
                + employee.getSalaire());
        }
    }
}
```

```
package fr.paris.java.exercice7;

public class Employe {
    private String nom;
    private double salaire;

    protected Employe(String nom, double salaire) {
        this.nom = nom;
        this.salaire = salaire;
    }

    public String getNom() {
        return nom;
    }

    public double getSalaire() {
        return salaire;
    }
}
```

```
package fr.paris.java.exercice7;

public class EmployeException extends Exception {
    private Employe emp;
    public EmployeException(String message, Employe emp) {
        super(message);
        this.emp = emp;
    }
}
```

```
package fr.paris.java.exercice7;

import java.util.ArrayList;
import java.util.Iterator;

public class Entreprise implements Iterable<Employe> {
    private String nom;
    private ArrayList<Employe> employees;

    public Entreprise(String nom) {
        this.nom = nom;
        employees = new ArrayList<Employe>();
    }

    public String getNom() {
        return nom;
    }

    public void ajouter(Employe emp) throws EmployeException {
        if (! employees.add(emp)) {
            throw new EmployeException("Employé déjà dans cette entreprise", emp);
        }
    }
}
```



```

    }

    @Override
    public Iterator<Employe> iterator() {
        return employes.iterator();
    }

    @Override
    public String toString() {
        StringBuffer sb = new StringBuffer(nom);
        for (Employe employe : employes) {
            sb.append("\n    " + employe.getNom());
        }
        return sb.toString();
    }
}

```

```

package fr.paris.java.exercice7;

import java.util.Comparator;

/**
 * Compateur d'employés. Utilise le salaire pour comparer 2 employés.
 */
public class CompareSalaire implements Comparator<Employe> {

    /**
     * Compare 2 employés suivant leur salaire.
     * @return un nombre positif si le salaire de e1 est supérieur au
     * salaire de e2, 0 si les 2 salaires sont les mêmes, et négatif sinon.
     */
    public int compare(Employe e1, Employe e2) {
        return Double.compare(e1.getSalaire(), e2.getSalaire());
    }
}

```

Etape1 : Remplacez CompareSalaire par une expression lambda pour trier par salaire croissant.
 Remarquez la simplification par rapport à l'ancien code, et aussi l'amélioration de la lisibilité.

```

package fr.paris.java.exercice7;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Etape1 {
    public static void main(String[] args) throws EmployeeException {
        Entreprise e1 = new Entreprise("IBM");
        e1.ajouter(new Employe("Cyril", 5000));
        e1.ajouter(new Employe("Laurent", 6000));
        e1.ajouter(new Employe("Rima", 5700));
        e1.ajouter(new Employe("Victorien", 4300));
        System.out.println(e1);
    }
}

```

```

// Remplit une liste avec les employés
List<Employe> l = new ArrayList<>();
for (Employe e : e1) {
    l.add(e);
}

// Tri par salaires croissants
Collections.sort(l,
    (emp1, emp2) -> Double.compare(emp1.getSalaire(),
        emp2.getSalaire()));

System.out.println("Employés de " + e1.getNom()
    + " par ordre croissant des salaires");
for (Employe employe : l) {
    System.out.println(employe.getNom() + " gagne "
        + employe.getSalaire());
}
}
}

```

Etape2 : Capture d'une variable du contexte englobant

Ecrivez une méthode static `add` qui prend un entier en paramètre et qui retourne une fonction qui prend un entier en paramètre et qui retourne un `Long` (fonction de type `Integer -> Long` ; utilisez l'interface fonctionnelle `ToLongFunction`).

Utilisez cette méthode pour afficher la somme de 3 et de 5.

```

package fr.paris.java.exercice7;

import java.util.function.ToLongFunction;

public class Etape2 {

    public static ToLongFunction<Integer> add(Integer param) {
        ToLongFunction<Integer> result = (x) -> x + param;
        return result;
    }

    public static void main(String[] args) {
        // définition d'une fonction qui ajoute 3
        ToLongFunction<Integer> add = add(3);

        System.out.println(add.applyAsLong(5));
    }
}

```

Créez une liste d'entiers. Utilisez ensuite cette méthode `add` pour transformer cette liste en une autre liste dont tous les éléments sont égaux aux éléments de la première liste + 3. Avec `forEach` faites afficher tous les éléments de la nouvelle liste.

```

package fr.paris.java.exercice7;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.ToLongFunction;

public class Etape2 {
    static List<Long> entiersPlus3 = new ArrayList<Long>();

    public static ToLongFunction<Integer> add(Integer param) {
        ToLongFunction<Integer> result = (x) -> x + param;
        return result;
    }

    public static void main(String[] args) {
        // définition d'une fonction qui ajoute 3
        ToLongFunction<Integer> add = add(3);
        System.out.println(add.applyAsLong(5));
        System.out.println();
        List<Integer> entiers = new ArrayList<Integer>();
        entiers.addAll(Arrays.asList(1, 3, 5, 7, 9, 8, 6, 4, 2, 0));

        entiers.forEach(entier -> entiersPlus3.add(add.applyAsLong(entier)));
        entiersPlus3.forEach(entier -> System.out.println(entier));
    }
}

```

Etape3 : Reprenez la classe `Employe` précédente.

Ecrivez une classe `Etape3` qui ajoute plusieurs employés à une liste, qui filtre les employés qui gagnent plus de 15000, puis les trie par ordre de salaire croissant, et finalement affiche une seule ligne du type

21000 : Bob, 23000 : Fred, 31000 : Pierre

Vous utiliserez l'API des streams, ainsi que les méthodes `filter`, `sorted`, `map` (pour avoir un stream dont les éléments sont du type "23000 : Fred", finalement la méthode `collect` qui prendra en paramètre `Collector.joining(", ")`.

Exercice8 : Nouveaux outils du jdk 8

Cet exercice a l'intention de donner un aperçu de l'outil de surveillance et de diagnostic JMC : Oracle Java Mission Control, et comment il peut être utilisé pour étudier les performances des applications Java.

Etape1 : connexion à une JMC instance de SampleConsoleServer s'exécutant sur un hôte distant

Java Mission Control est gratuit pour toute utilisation de développement et de test, mais nécessite une licence lorsqu'il est utilisé à des fins de production.

Java Mission Contrôle est une suite d'outils pour la gestion, le suivi, le profilage et le dépannage de vos applications Java. JMC a été inclus dans le SDK Java standard depuis la version 7u40. JMC se compose de la console JMX et de Java Flight Recorder. Plus de plug-ins peuvent être facilement installés au sein de Mission Control. JMC peut aussi être installé dans l'IDE Eclipse.

Java Mission Control utilise JMX pour communiquer avec les processus Java distants. La console JMX est un outil pour le suivi et la gestion d'une instance en cours d'exécution JVM. L'outil présente des données en direct sur la mémoire et l'utilisation du processeur, le ramassage des ordures, l'activité des threads, et plus encore. Il comprend également un navigateur JMX MBean entièrement que vous pouvez utiliser pour surveiller et gérer des MBeans dans la JVM et dans votre application Java.

Pour activer l'agent de gestion à distance sur un processus Java, nous allons ajouter les paramètres suivants lorsque vous démarrez :

```
-Dcom.sun.management.jmxremote=true  
-Dcom.sun.management.jmxremote.port=3614  
-Dcom.sun.management.jmxremote.authenticate=false  
-Dcom.sun.management.jmxremote.ssl=false
```

Localement ces options ne sont pas nécessaires, mais dans cet exercice nous allons exécuter TCPServer sur un serveur de test à distance.

Soit le code serveur TCP ci-dessous:

```
package fr.paris.java.exercice8;  
import java.io.*;  
import java.net.*;  
  
class TCPServer  
{  
    public static void main(String argv[]) throws Exception  
    {  
        String clientSentence;  
        String capitalizedSentence;  
        ServerSocket welcomeSocket = new ServerSocket(6789);  
  
        while(true)  
        {  
            Socket connectionSocket = welcomeSocket.accept();  
            BufferedReader inFromClient =
```

```

        new BufferedReader(new
InputStreamReader(connectionSocket.getInputStream()));
        DataOutputStream outToClient = new
DataOutputStream(connectionSocket.getOutputStream());
        clientSentence = inFromClient.readLine();
        System.out.println("Received: " + clientSentence);
        capitalizedSentence = clientSentence.toUpperCase() + '\n';
        outToClient.writeBytes(capitalizedSentence);
    }
}
}

```

Un client TCP est donné pour exemple ci-après

```

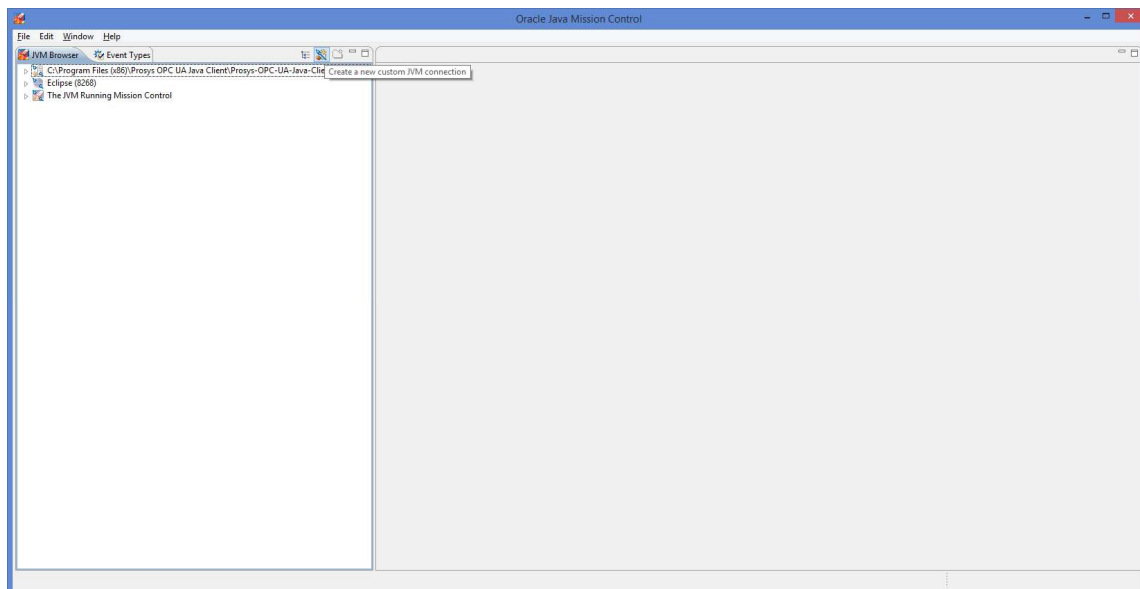
package fr.paris.java.exercice8;
import java.io.*;
import java.net.*;

class TCPClient
{
    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;
        BufferedReader inFromUser = new BufferedReader( new InputStreamReader(System.in));
        Socket clientSocket = new Socket("localhost", 6789);
        DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
        BufferedReader inFromServer = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
        sentence = inFromUser.readLine();
        outToServer.writeBytes(sentence + '\n');
        modifiedSentence = inFromServer.readLine();
        System.out.println("FROM SERVER: " + modifiedSentence);
        clientSocket.close();
    }
}

```

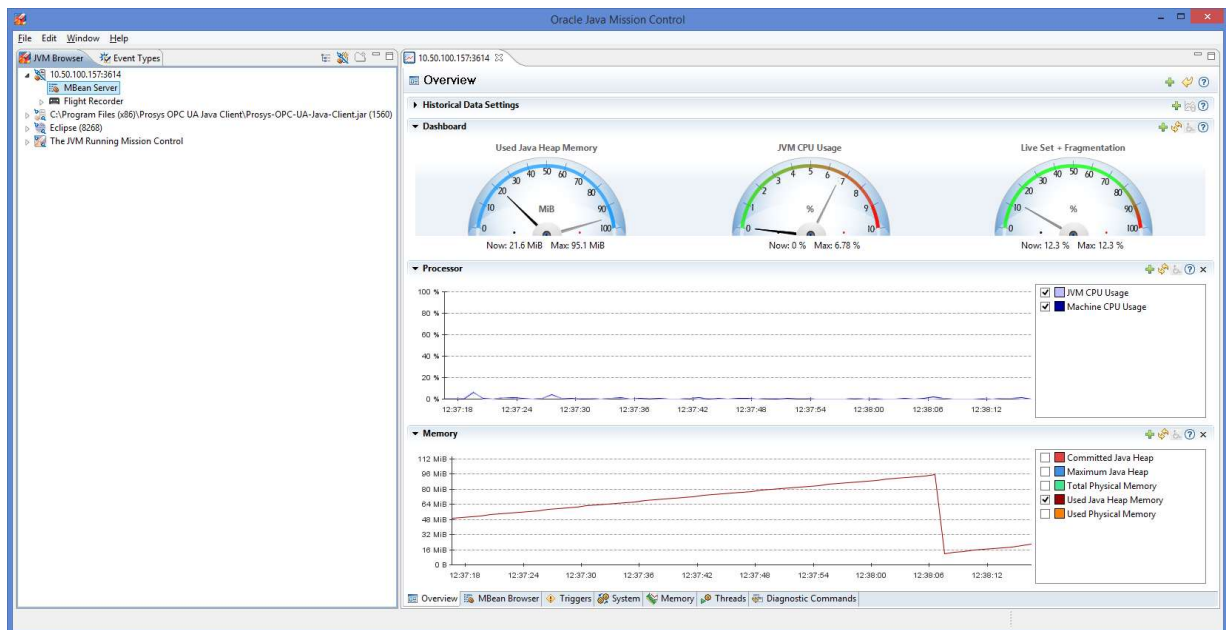
Maintenant, quand le serveur est démarré, il autoriser les connexions JMX au port 3614. Le numéro de port a été attribué par hasard.

JMC est regroupé avec JDK et peut être trouvé dans "C: \ Program Files \ Java \ jdk1.xx_xx \ bin \ jmc.exe" sur les machines Windows.

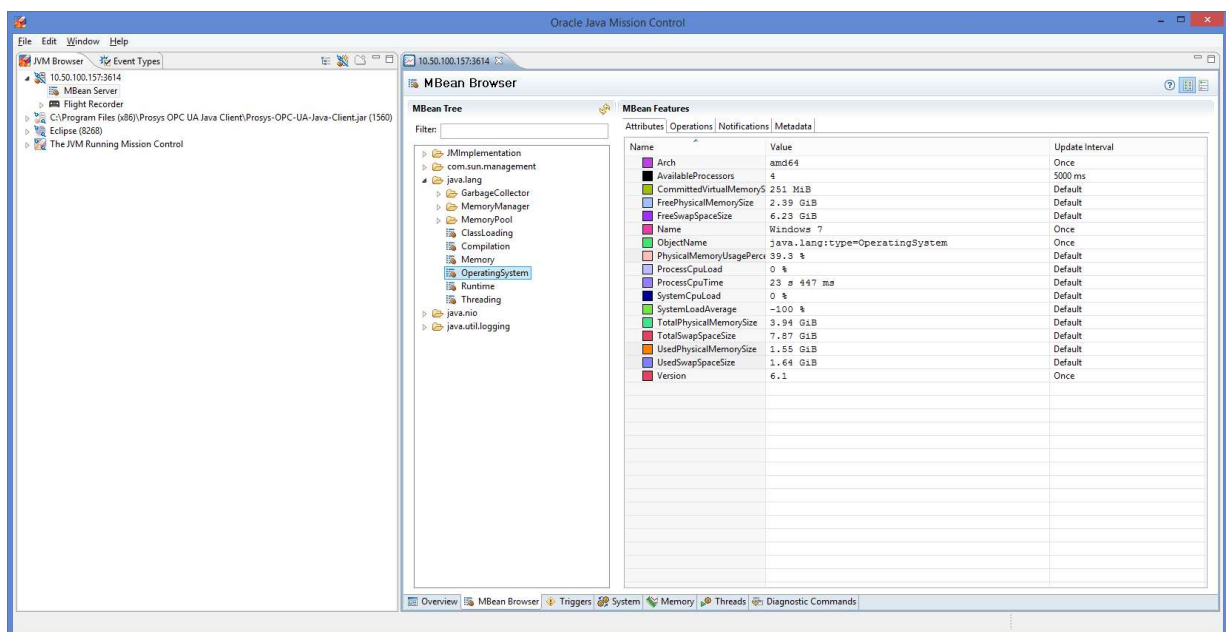


Sélectionnez “Create a new custom JVM connection”

Entrez le nom d'hôte / adresse IP et le port et cliquez sur Terminer.



Copie de l'onglet « Overview »



Surveillez votre application Java à travers MBeans dans le navigateur MBean.

Modifier la classe `TCPServer` afin d'ajouter

- une propriété de type String nommée `label` qui est utilisée dans l'impression à l'écran (`out.println`),
- une propriété booléenne nommée `running` qui est utilisée comme condition d'arrêt dans la boucle `while`.

Définir une interface `TCPServerMBean` qui ait les obligations de codage pour les getters et setters de ces attributs.

Modifier la classe `TCPServer` afin qu'elle implémente ce contrat de codage. Restructurer cette classe afin de faire apparaître un constructeur qui comprend le contenu du main.

Une fois que le serveur a été instrumenté par Mbeans, la gestion de la ressource est réalisée par un agent JMX. Il est demandé de faire une classe principale qui publie cet agent JMX auprès du serveur de composant JMX.

```
package fr.paris.java.exercice8;

import java.lang.management.*;
import javax.management.*;

public class Main {

    public static void main(String[] args)
        throws Exception {

        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
        ObjectName name = new ObjectName("fr.paris.java.exercice8:type=TCPServer");
        TCPServer server= new TCPServer();
        mbs.registerMBean(server, name);

        ...

        System.out.println("Waiting forever...");
        Thread.sleep(Long.MAX_VALUE);
    }
}
```

Après exécution de ce code, il est demandé d'observer le bean dans la console JMC et de modifier la chaîne de caractères pour comprendre l'impact dans l'affichage du serveur. Enfin une modification de la propriété `running` stoppe le serveur.

Etape2 : connexion à des données d'exécution avec Java Flight Recorder et de l'analyser avec Java Mission Control.

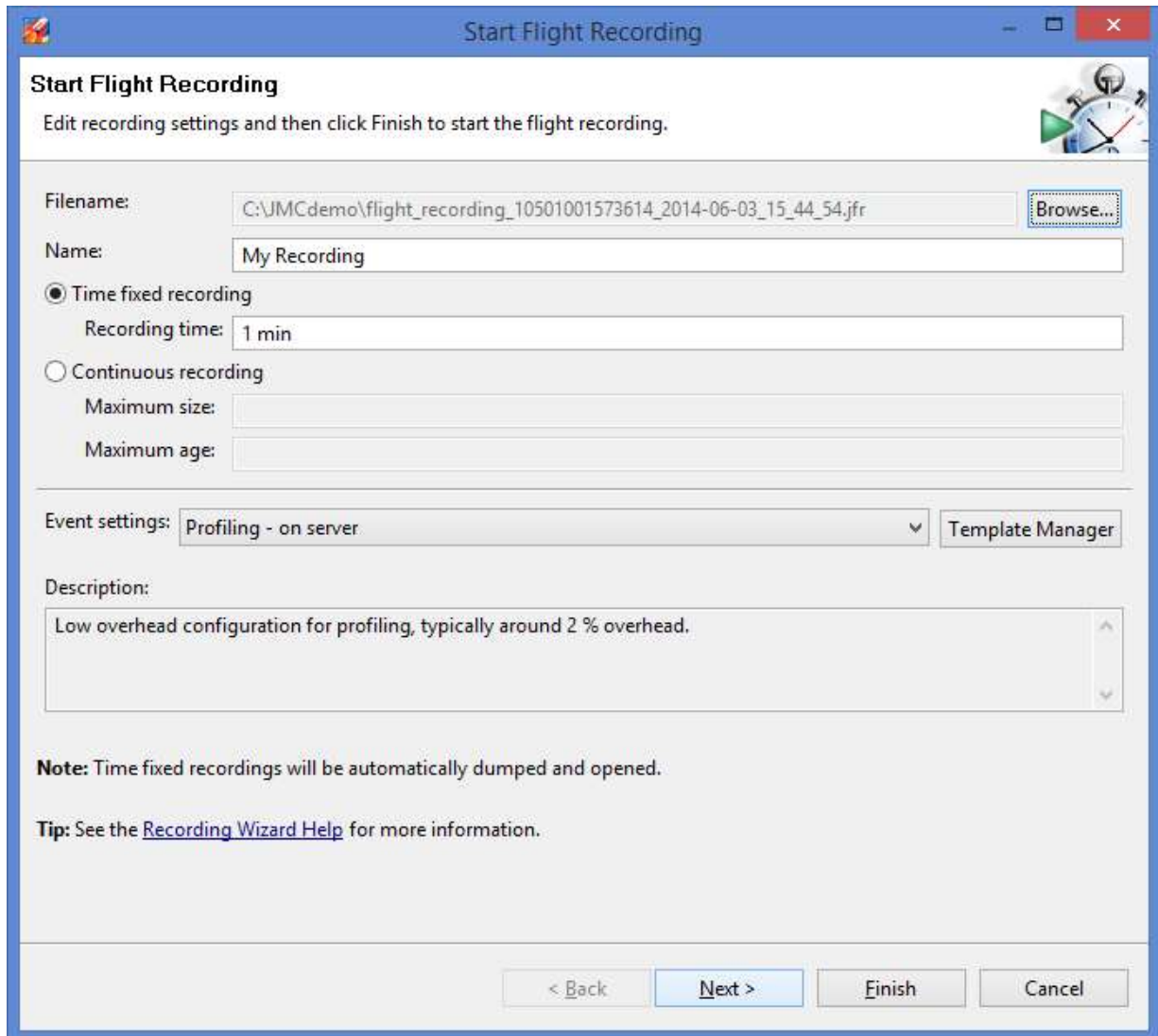
Java Flight Recorder est un framework de profilage et collection d'événements intégré à l'Oracle JDK. Java Flight Recorder peut être utilisé pour recueillir des enregistrements sans utiliser JMC, mais dans cet exemple nous utiliserons ces outils ensemble.

L'application qui est à analyser doit toujours être lancée avec des options

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

Nous allons également lancer un script de connexion aux clients, la lecture des valeurs provenant de `TCPServer` et déconnexion pour obtenir un ensemble de données généré intéressant.

Paramètres de Flight Recording peuvent être obtenus en double-cliquant sur l'icône " Flight Recording " dans le navigateur de la JVM.



Start Flight Recording

Edit recording settings and then click Finish to start the flight recording.

Filename: C:\JMCdemo\flight_recording_10501001573614_2014-06-03_15_44_54.jfr Browse...

Name: My Recording

☒ Time fixed recording

Recording time: 1 min

☐ Continuous recording

Maximum size:

Maximum age:

Event settings: Profiling - on server Template Manager

Description: Low overhead configuration for profiling, typically around 2 % overhead.

Note: Time fixed recordings will be automatically dumped and opened.

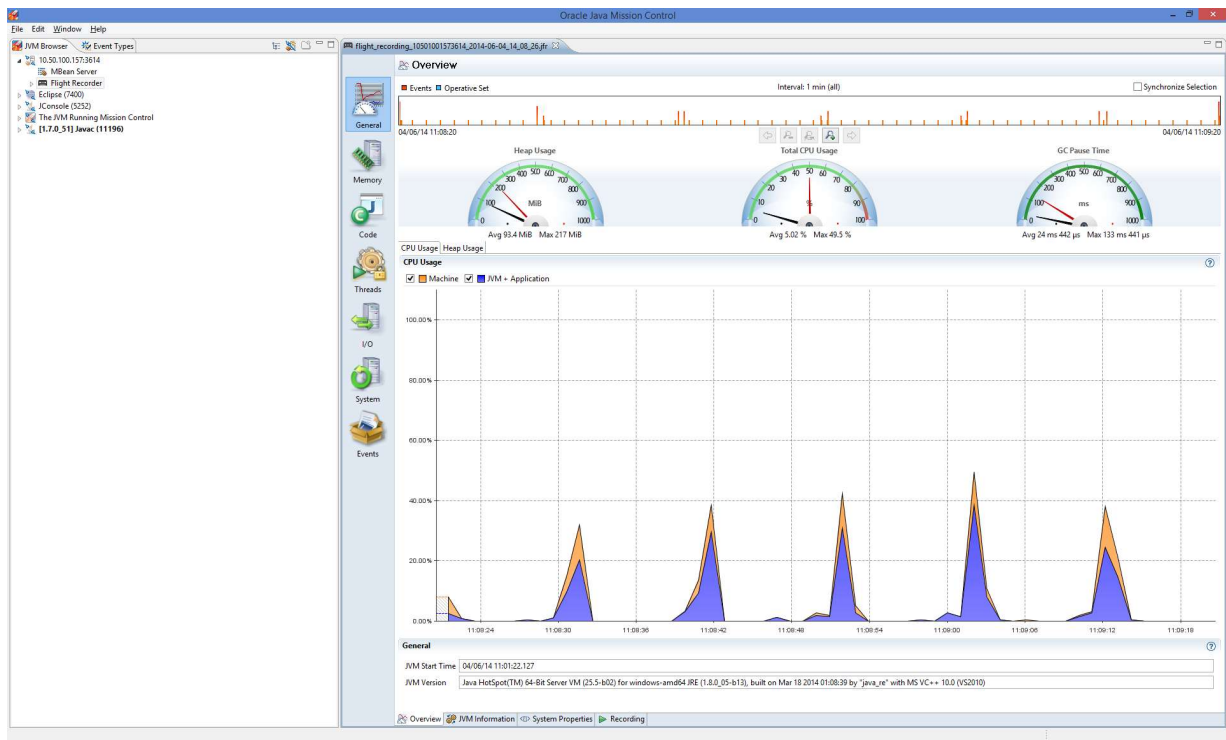
Tip: See the [Recording Wizard Help](#) for more information.

< Back **Next >** Finish Cancel

Il existe deux types d'enregistrement: enregistrements alternatifs fixes et enregistrements continus. Enregistrements continus n'ont pas de temps de fin défini et ils doivent être explicitement fournis.

Après avoir cliqué sur "Terminer", le bord inférieur de l'écran affiche la progression de l'enregistrement. Dans cet exercice, nous allons utiliser l'enregistrement de temps fixe et lancez le Flight Recording pendant 1 minute. Après les fins d'enregistrement, les résultats peuvent être analysés avec JMC.

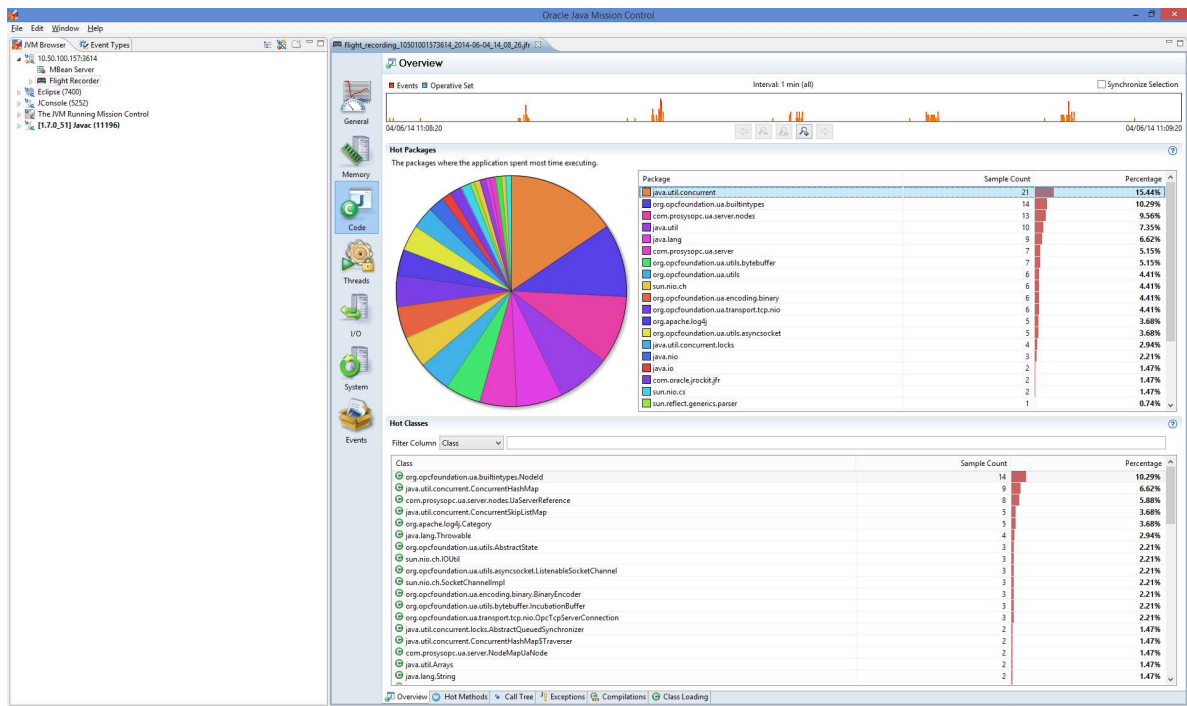
L'onglet général de Flight Recording ressemble à l'écran suivant :



Le Flight Recording de Java fournit des informations sur le système d'exploitation, la JVM et l'application Java s'exécutant dans la JVM.

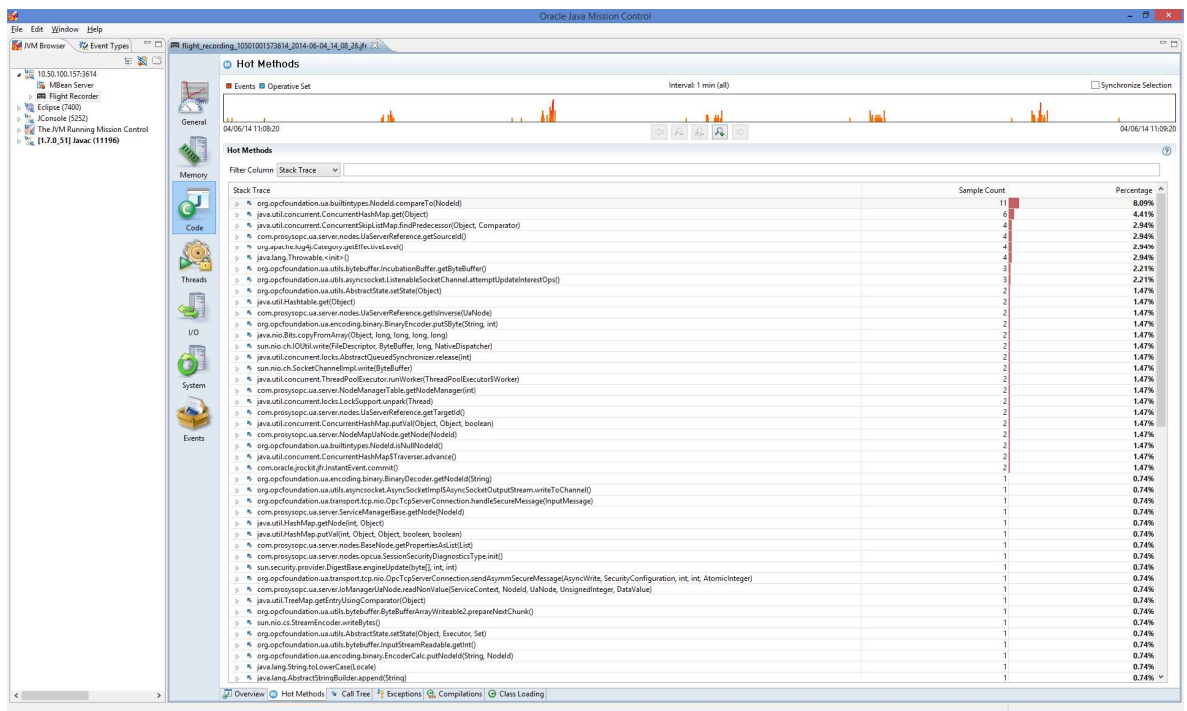
Il est clairement visible depuis les pics d'utilisation du CPU que le script a connecté et déconnecté à un ensemble de clients cinq fois au cours de la période de temps d'une minute d'enregistrement de vol. Au début de l'enregistrement dernier lot de clients précédents étaient également sur le point de se déconnecter du serveur.

Ensuite, nous passons à l'onglet «Code»:



Cet onglet nous montre l'activité des différents packages et classes. On peut en déduire que `java.util.concurrent` a été le package le plus utilisé lors de cette exécution et `org.opcfoundation.ua.builtintypes.NodeId` est la classe la plus utilisée.

Passons sur l'onglet "Méthod Hot" dans le bas de l'écran:



Ce point de vue nous montre l'activité de méthodes spécifiques et l'origine de l'endroit où ces méthodes ont été appelées. Nous pouvons voir que "org.opc.foundation.ua.builtin.types.NodeId.compareTo (NodeId)" consommé le plus de temps dans notre contexte d'exécution.

Etape3 : Utilisation de jdeps.

C'est un nouvel outil du jdk8 pour évaluer les dépendances de vos codes.

A la suite de notre étude, il est demandé de rechercher les dépendances de la classe TCPServer

```
jdeps -verbose:class -cp classes fr.paris.java.exercice8.TCPServer
```

```
Usage: jdeps <options> <classes...>
where <classes> can be a pathname to a .class file, a directory, a JAR file,
or a fully-qualified classname or wildcard "*". Possible options include:
-s          --summary          Print dependency summary only
-v          --verbose          Print additional information
-V <level>  --verbose-level=<level> Print package-level or class-level dependencies
Valid levels are: "package" and "class"
-c <path>   --classpath=<path> Specify where to find class files
-p <pkg name> --package=<pkg name> Restrict analysis to classes in this package
(may be given multiple times)
-e <regex>  --regex=<regex>    Restrict analysis to packages matching pattern
(-p and -e are exclusive)
-P          --profile          Show profile or the file containing a package
-R          --recursive        Recursively traverse all dependencies
--version          Version information
```

Utilisez le -R ou une option -recursive d'analyser les dépendances transitives

Références

1. <https://leanpub.com/whatsnewinjava8/read>
2. <http://www.javacodegeeks.com/2014/05/java-8-features-tutorial.html>
3. <https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>
4. <http://www.informit.com/articles/article.aspx?p=2191424&seqNum=2>
5. <http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html>
6. <http://richard.grin.free.fr/ufr/cours/tic409/tp/java8/tp/lambda/index.html>
7. <http://www.leveluplunch.com/blog/2014/02/06/more-to-java8-than-lambdas-date-time-api/>
8. <http://www.selikoff.net/2015/01/25/using-nashorns-jjs-for-experimenting/>
9. <http://java-buddy.blogspot.fr/2014/04/nashorn-javascript-exercise-call-java.html>
10. <http://www.java.meximas.com/nashorn-javascript-exercise-call-java-method-from-javascript/>
11. <http://www.javaworld.com/article/2078809/java-concurrency/java-101-the-next-generation-java-concurrency-without-the-pain-part-1.html>
- 12.