

Les nouveautés de Java 8

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

`elmouelhi.achref@gmail.com`



Plan

- 1 Classe anonyme
- 2 Implémentation par défaut
- 3 Interface fonctionnelle
- 4 Expression Lambda
- 5 Interfaces fonctionnelles prédéfinies
 - `Function<T, R>`
 - `BiFunction<T1, T2, R>`
 - `BinaryOperator<T>`
 - `Consumer<T>`
 - `Predicate<T>`
 - `BiPredicate<T, R>`
 - `IntFunction<T>`
 - `Supplier<T>`
 - **Autres interfaces fonctionnelles prédéfinies**
- 6 Références de méthodes

7 Quelques nouvelles méthodes pour les collections

8 API Stream

- `of()` **et** `stream()`
- `forEach()`
- `map()`
- `filter()`
- `reduce()`
- `collect()`
- `count()`, `max()` ...
- `...Match()`
- **Autres méthodes**

9 `IntStream`, `LongStream` **et** `DoubleStream`

10 API Date-Time

- Temps humain
- Temps machine

Classe anonyme ?

- Déclarée au moment de l'instanciation de sa classe mère, qui peut être
 - concrète
 - abstraite
 - interface
- Pouvant accéder aux attributs et méthodes de la classe englobante

Java

Dans un Java Project, commençons par créer la classe `Personne` suivante

```
package org.eclipse.model;

public class Personne {

    private String nom;
    private String prenom;
    private int age;

    public Personne(String nom, String prenom, int age) {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
    }

    // + getters et setters + toString()
}
```

Considérons la classe abstraite suivante

```
package org.eclipse.model;  
  
public abstract class IMiseEnForme {  
    public abstract void afficherDetails();  
}
```

Java

Créons une instance de la classe fille anonyme de la classe `IMiseEnForme` dans `Personne`

```
public class Personne {  
  
    private String nom;  
    private String prenom;  
    private int age;  
    public IMiseEnForme iMiseEnForme = new IMiseEnForme() {  
  
        @Override  
        public void afficherDetails() {  
            // TODO Auto-generated method stub  
            System.out.println("nom = " + nom + ", prenom = " + prenom);  
        }  
    };  
    public Personne(String nom, String prenom, int age) {  
        this.nom = nom;  
        this.prenom = prenom;  
        this.age = age;  
    }  
  
    // + getters et setters  
}
```

Java

Expliquons le code suivant

```
public IMiseEnForme iMiseEnForme = new IMiseEnForme() {  
  
    @Override  
    public void afficherDetails() {  
        // TODO Auto-generated method stub  
        System.out.println("nom = " + nom + ", prenom = " +  
            prenom);  
    }  
};
```

- `iMiseEnForme` : nom de l'instance de la classe fille anonyme dérivée de `IMiseEnForme`
- Une classe héritant d'une classe abstraite doit implémenter ses méthodes abstraites : pour cette, on a l'annotation `@Override` sur la méthode `afficherDetails()`

Java

Pour tester et vérifier que `iMiseEnForme` est une instance de la classe fille anonyme dérivée de `IMiseEnForme`

```
package org.eclipse.test;

public class Main {
    public static void main(String [] args) {
        Personne personne = new Personne("el mouelhi", "achref", 34);
        personne.iMiseEnForme.afficherDetails();
        System.out.println(personne.iMiseEnForme.getClass());
        System.out.println(personne.iMiseEnForme.getClass().
            getSuperclass());
    }
}
```

Java

Pour tester et vérifier que `iMiseEnForme` est une instance de la classe fille anonyme dérivée de `IMiseEnForme`

```
package org.eclipse.test;

public class Main {
    public static void main(String [] args) {
        Personne personne = new Personne("el mouelhi", "achref", 34);
        personne.iMiseEnForme.afficherDetails();
        System.out.println(personne.iMiseEnForme.getClass());
        System.out.println(personne.iMiseEnForme.getClass().
            getSuperclass());
    }
}
```

Le résultat sera

```
nom = el mouelhi, prenom = achref
class org.eclipse.model.Personne$1
class org.eclipse.model.IMiseEnForme
```

Remarques

À la compilation deux fichiers, relatifs à la classe `Personne`, ont été générés (d'extension `.class`)

- Le premier : `Personne`
- Le deuxième : `Personne$1` pour la classe anonyme \Rightarrow 1 étant l'indice de cette classe anonyme (les classes anonymes seront numérotées dans l'ordre et la première aura l'indice 1)



Remarques

À la compilation deux fichiers, relatifs à la classe `Personne`, ont été générés (d'extension `.class`)

- Le premier : `Personne`
- Le deuxième : `Personne$1` pour la classe anonyme \Rightarrow 1 étant l'indice de cette classe anonyme (les classes anonymes seront numérotées dans l'ordre et la première aura l'indice 1)

Remarques générales

- Une classe anonyme n'a pas d'identificateur et ne peut donc être instanciée qu'une seule fois (d'où son nom).
- Une classe anonyme est implicitement considérée comme finale (et ne peut donc pas être abstraite)

Java

Transformons la classe abstraite précédente en interface

```
public interface IMiseEnForme {  
    public abstract void afficherDetails();  
}
```

© Achref EL MOUELHI ©

Java

Transformons la classe abstraite précédente en interface

```
public interface IMiseEnForme {  
    public abstract void afficherDetails();  
}
```

Nous ne changeons rien dans le `main`

```
public class Main {  
    public static void main(String [] args) {  
        Personne personne = new Personne("el mouelhi", "achref", 34);  
        personne.iMiseEnForme.afficherDetails();  
        System.out.println(personne.iMiseEnForme.getClass());  
        System.out.println(personne.iMiseEnForme.getClass().getSuperclass());  
    }  
}
```

Java

Transformons la classe abstraite précédente en interface

```
public interface IMiseEnForme {  
    public abstract void afficherDetails();  
}
```

Nous ne changeons rien dans le `main`

```
public class Main {  
    public static void main(String [] args) {  
        Personne personne = new Personne("el mouelhi", "achref", 34);  
        personne.iMiseEnForme.afficherDetails();  
        System.out.println(personne.iMiseEnForme.getClass());  
        System.out.println(personne.iMiseEnForme.getClass().getSuperclass());  
    }  
}
```

Le résultat sera

```
nom = el mouelhi, prenom = achref  
class model.Personne$1  
class java.lang.Object // car une interface n'est pas réellement une  
    classe mère
```

Implémentation par défaut ?

- Jusqu'à Java 7, une interface ne peut contenir que des méthodes abstraites
- Depuis Java 8, une interface peut proposer une implémentation par défaut pour ces méthodes.

Ajoutons une méthode avec une implémentation par défaut à notre interface `IMiseEnForme`

```
public interface IMiseEnForme {  
    public void afficherDetails();  
    default public void afficherNomComplet(String nom,  
        String prenom) {  
        System.out.println(nom + " " + prenom);  
    }  
}
```

Java

Pour tester

```
public class Main {  
    public static void main(String [] args) {  
        Personne personne = new Personne("el mouelhi", "achref", 34);  
        personne.iMiseEnForme.afficherDetails();  
        personne.iMiseEnForme.afficherNomComplet(personne.getNom(),  
            personne.getPrenom());  
        System.out.println(personne.iMiseEnForme.getClass());  
        System.out.println(personne.iMiseEnForme.getClass().  
            getSuperclass());  
    }  
}
```

Java

Pour tester

```
public class Main {  
    public static void main(String [] args) {  
        Personne personne = new Personne("el mouelhi", "achref", 34);  
        personne.iMiseEnForme.afficherDetails();  
        personne.iMiseEnForme.afficherNomCompleet (personne.getNom(),  
            personne.getPrenom());  
        System.out.println(personne.iMiseEnForme.getClass());  
        System.out.println(personne.iMiseEnForme.getClass().  
            getSuperclass());  
    }  
}
```

Le résultat sera

```
nom = el mouelhi, prenom = achref  
el mouelhi achref  
class org.eclipse.model.Personne$1  
class java.lang.Object
```

Java

Interface fonctionnelle ?

- interface contenant une seule méthode abstraite
- pouvant contenir plusieurs méthodes avec une implémentation par défaut
- existe depuis Java 8
- utilisée souvent avec les expressions Lambda

Java

Interface fonctionnelle ?

- interface contenant une seule méthode abstraite
- pouvant contenir plusieurs méthodes avec une implémentation par défaut
- existe depuis Java 8
- utilisée souvent avec les expressions Lambda

Interface fonctionnelle, pourquoi ?

- code plus facile à lire, écrire et maintenir (une seule méthode abstraite par interface)
- supprimer le maximum de logique applicative d'un programme

Java

On peut ajouter une annotation Java 8 (`@FunctionalInterface`) pour vérifier si on a bien respecté la contrainte

```
@FunctionalInterface
public interface IMiseEnForme {
    public void afficherDetails();
    default public void afficherNomComplet(String nom, String prenom) {
        System.out.println(nom + " " + prenom);
    }
}
```

© Achref EL M...

Java

On peut ajouter une annotation Java 8 (@FunctionalInterface) pour vérifier si on a bien respecté la contrainte

```
@FunctionalInterface
public interface IMiseEnForme {
    public void afficherDetails();
    default public void afficherNomComplet(String nom, String prenom) {
        System.out.println(nom + " " + prenom);
    }
}
```

En ajoutant une nouvelle méthode abstraite, une erreur (**Invalid '@FunctionalInterface' annotation; IMiseEnForme is not a functional interface**) s'affiche

```
@FunctionalInterface
public interface IMiseEnForme {
    public void afficherNomMajuscule();
    public void afficherDetails();
    default public void afficherNomComplet(String nom, String prenom) {
        System.out.println(nom + " " + prenom);
    }
}
```

Java

Le code permettant d'instancier la classe anonyme implémentant l'interface

IMiseEnForme suivant

```
public IMiseEnForme iMiseEnForme = new IMiseEnForme() {  
  
    @Override  
    public void afficherDetails() {  
        // TODO Auto-generated method stub  
        System.out.println("nom = " + nom + ", prenom = " + prenom );  
    }  
};
```

© Achref EL

Java

Le code permettant d'instancier la classe anonyme implémentant l'interface

IMiseEnForme suivant

```
public IMiseEnForme iMiseEnForme = new IMiseEnForme() {  
  
    @Override  
    public void afficherDetails() {  
        // TODO Auto-generated method stub  
        System.out.println("nom = " + nom + ", prenom = " + prenom );  
    }  
};
```

Peut être réécrit en utilisant les expressions Lambda

```
public IMiseEnForme iMiseEnForme = () -> System.out.println("nom = " +  
    nom + ", prenom = " + prenom);
```

Java

Le code permettant d'instancier la classe anonyme implémentant l'interface

IMiseEnForme suivant

```
public IMiseEnForme iMiseEnForme = new IMiseEnForme() {  
  
    @Override  
    public void afficherDetails() {  
        // TODO Auto-generated method stub  
        System.out.println("nom = " + nom + ", prenom = " + prenom );  
    }  
};
```

Peut être réécrit en utilisant les expressions Lambda

```
public IMiseEnForme iMiseEnForme = () -> System.out.println("nom = " +  
    nom + ", prenom = " + prenom);
```

- L'utilisation des expressions Lambda pour instancier une classe anonyme ne fonctionne que si cette dernière implémente une interface ayant une seule méthode abstraite ⇒ interface **fonctionnelle**
- `iMiseEnForme` n'est plus le nom d'un objet de la classe anonyme mais plutôt le nom de l'expression Lambda

Java

Expressions Lambda

- permettent d'implémenter et d'instancier les interfaces fonctionnelles (une sorte d'instance d'interface fonctionnelle)
- doivent avoir les mêmes paramètres et valeur de retour (signature) que la méthode abstraite de l'interface fonctionnelle
- existent depuis Java 8
- utilisent `->` pour séparer la partie paramètres et la partie traitement

Java

Expressions Lambda

- permettent d'implémenter et d'instancier les interfaces fonctionnelles (une sorte d'instance d'interface fonctionnelle)
- doivent avoir les mêmes paramètres et valeur de retour (signature) que la méthode abstraite de l'interface fonctionnelle
- existent depuis Java 8
- utilisent `->` pour séparer la partie paramètres et la partie traitement

Syntaxe

```
NomInterface nomLambdaExpression = ([arguments]) -> {  
    traitement  
};
```

Java

Considérons l'interface fonctionnelle suivante

```
@FunctionalInterface
public interface ICalcul {
    public int operationBinaire(int x, int y);
}
```

© Achref EL MOUËL

Java

Considérons l'interface fonctionnelle suivante

```
@FunctionalInterface
public interface ICalcul {
    public int operationBinaire(int x, int y);
}
```

Définissons une expression lambda dans `main` qui permettra de calculer la somme (le `return` est implicite ici)

```
ICalcul plus = (int x, int y) -> x + y;
```

Java

Considérons l'interface fonctionnelle suivante

```
@FunctionalInterface
public interface ICalcul {
    public int operationBinaire(int x, int y);
}
```

Définissons une expression lambda dans `main` qui permettra de calculer la somme (le `return` est implicite ici)

```
ICalcul plus = (int x, int y) -> x + y;
```

Pour calculer la somme de deux entiers 3 et 5, il faut faire

```
System.out.println(plus.operationBinaire(3, 5));
```

Java

La précision de type n'est pas obligatoire

```
ICalcul plus = (x, y) -> x + y;
```

© Achref EL MOUELHI ©

Java

La précision de type n'est pas obligatoire

```
ICalcul plus = (x, y) -> x + y;
```

Les accolades sont obligatoires au delà de deux instructions

```
ICalcul plus = (x, y) -> {  
    int resultat = x + y;  
    return resultat;  
};
```

Java

La précision de type n'est pas obligatoire

```
ICalcul plus = (x, y) -> x + y;
```

Les accolades sont obligatoires au delà de deux instructions

```
ICalcul plus = (x, y) -> {  
    int resultat = x + y;  
    return resultat;  
};
```

Une expression lambda peut utiliser une variable (ou un attribut) définie dans le contexte englobant

```
int i = 2, j = 3;  
ICalcul calculComplexe = (x, y) -> {  
    return x * i + y * j;  
};
```

Java

Une expression lambda ne peut redéfinir une variable (ou un attribut) définie dans le contexte englobant (**Ceci est faux car `i` a été déclaré et initialisé juste avant**)

```
int i = 2, j = 3;
ICalcul calculComplexe = (x, y) -> {
    int i = 5;
    return x * i + y * j;
};
```

© Achref EL

Java

Une expression lambda ne peut redéfinir une variable (ou un attribut) définie dans le contexte englobant (**Ceci est faux car i a été déclaré et initialisé juste avant**)

```
int i = 2, j = 3;
ICalcul calculComplexe = (x, y) -> {
    int i = 5;
    return x * i + y * j;
};
```

Une expression lambda ne peut modifier la valeur d'une variable (ou un attribut) définie dans le contexte englobant (**Ceci est faux**)

```
int i = 2, j = 3;
ICalcul calculComplexe = (x, y) -> {
    i++;
    return x * i + y * j;
};
```

Une solution possible pour le problème précédent

```
int i = 2, j = 3;  
ICalcul calculComplexe = (x, y) -> {  
    final int k = i + 1;  
    return x * k + y * j;  
};
```

Expressions Lambda

- Une expression lambda ne fonctionne pas sans interface fonctionnelle
- Faudrait-il créer une interface fonctionnelle chaque fois qu'on a besoin de définir et exécuter une expression lambda ?

© Achret

Expressions Lambda

- Une expression lambda ne fonctionne pas sans interface fonctionnelle
- Faudrait-il créer une interface fonctionnelle chaque fois qu'on a besoin de définir et exécuter une expression lambda ?

Solution

- Java 8 nous offre une quarantaine d'interfaces fonctionnelles
- Ces interfaces sont définies dans le package
`java.util.function`

Java

```
java.util.function.Function<T,R>
```

- Interface fonctionnelle possédant une méthode `apply` avec la signature `R apply(T t)`
- Paramètre d'entrée : variable de type `T`
- Valeur de retour : variable de type `R`

© Achref EL M...

Java

```
java.util.function.Function<T,R>
```

- Interface fonctionnelle possédant une méthode `apply` avec la signature `R apply(T t)`
- Paramètre d'entrée : variable de type `T`
- Valeur de retour : variable de type `R`

Considérons l'objet de type `Personne` suivant :

```
Personne personne = new Personne("el mouelhi", "achref", 34);
```

Java

```
java.util.function.Function<T,R>
```

- Interface fonctionnelle possédant une méthode `apply` avec la signature `R apply(T t)`
- Paramètre d'entrée : variable de type `T`
- Valeur de retour : variable de type `R`

Considérons l'objet de type `Personne` suivant :

```
Personne personne = new Personne("el mouelhi", "achref", 34);
```

Définissons une expression lambda qui prend comme entrée un objet de type `Personne` et qui retourne son nom concaténé à son prénom

```
Function<Personne, String> personneToString = (Personne p) -> p  
    .getNom() + " " + p.getPrenom();
```

Java

On peut simplifier l'expression précédente en supprimant les parenthèses et le type du paramètre d'entrée

```
Function<Personne, String> personneToString =  
    p -> p.getNom() + " " + p.getPrenom();
```

© Achref EL MOUËZ

Java

On peut simplifier l'expression précédente en supprimant les parenthèses et le type du paramètre d'entrée

```
Function<Personne, String> personneToString =  
    p -> p.getNom() + " " + p.getPrenom();
```

Pour exécuter

```
String nomComplet = personneToString.apply(personne);  
  
System.out.println(nomComplet);  
// affiche el mouelhi achref
```

Java

Considérons la liste de personne suivante

```
List<Personne> personnes = Arrays.asList(  
    new Personne("nom1", "prenom1", 35),  
    new Personne("nom2", "prenom2", 18),  
    new Personne("nom3", "prenom3", 27),  
    new Personne("nom4", "prenom4", 40)  
);
```

© Achref EL MOUËL

Java

Considérons la liste de personne suivante

```
List<Personne> personnes = Arrays.asList(  
    new Personne("nom1", "prenom1", 35),  
    new Personne("nom2", "prenom2", 18),  
    new Personne("nom3", "prenom3", 27),  
    new Personne("nom4", "prenom4", 40)  
);
```

Exercice

Écrire une méthode `static` appelée `listToStrings`

- prenant comme paramètre une liste de `Personne`
- utilisant l'interface fonctionnelle `Function` pour transformer un objet de type `Personne` en chaîne de caractère
- cette dernière correspond à l'attribut `nom` si l'attribut `age` est pair, à l'attribut `prénom` sinon.

Java

Correction : la méthode `listToStrings`

```
public static List<String> listToStrings(List<Personne>personnes) {  
    Function<Personne, String> personneToString =  
        p -> p.getAge() % 2 == 0 ? p.getNom() : p.getPrenom();  
    List<String> noms = new ArrayList<String>();  
    for(Personne personne : personnes) {  
        noms.add(personneToString.apply(personne));  
    }  
    return noms;  
}
```

© Achret L

Java

Correction : la méthode `listToStrings`

```
public static List<String> listToStrings(List<Personne>personnes) {  
    Function<Personne, String> personneToString =  
        p -> p.getAge() % 2 == 0 ? p.getNom() : p.getPrenom();  
    List<String> noms = new ArrayList<String>();  
    for(Personne personne : personnes) {  
        noms.add(personneToString.apply(personne));  
    }  
    return noms;  
}
```

Pour tester

```
List<String> noms = listToStrings(personnes);  
  
for(String nom : noms) {  
    System.out.println(nom);  
}
```


Java

La méthode `andThen()`

- permet le chaînage des fonctions

© Achref EL MOUELHI ©

Java

La méthode `andThen()`

- permet le chaînage des fonctions

Considérons les deux expressions lambda suivantes

```
Function<Personne, String> personneToString = p -> p.getNom() + " " + p
    .getPrenom();
Function<String, Integer> strToInt = str -> str.length();
```

Java

La méthode `andThen()`

- permet le chaînage des fonctions

Considérons les deux expressions lambda suivantes

```
Function<Personne, String> personneToString = p -> p.getNom() + " " + p
    .getPrenom();
Function<String, Integer> strToInt = str -> str.length();
```

On peut définir une troisième qui est le chaînage des deux précédents

```
// on applique personneToString ensuite strToInt
Function <Personne, Integer> personneToInt = personneToString.andThen(
    strToInt);

int longueur = personneToInt.apply(personne);
System.out.println(longueur);
// affiche 17
```

Java

La méthode `andThen()`

- permet le chaînage des fonctions

Considérons les deux expressions lambda suivantes

```
Function<Personne, String> personneToString = p -> p.getNom() + " " + p
    .getPrenom();
Function<String, Integer> strToInt = str -> str.length();
```

On peut définir une troisième qui est le chaînage des deux précédents

```
// on applique personneToString ensuite strToInt
Function <Personne, Integer> personneToInt = personneToString.andThen(
    strToInt);

int longueur = personneToInt.apply(personne);
System.out.println(longueur);
// affiche 17
```

Type de la valeur de retour de la première = Type du paramètre d'entrée de la deuxième.

Java

On peut aussi utiliser `compose` qui permet le chaînage mais dans le sens inverse

```
// on applique personneToString ensuite strToInt
Function<Personne, Integer> personneToInt = strToInt
    .compose(personneToString);

int longueur = personneToInt.apply(personne);
System.out.println(longueur);
// affiche 17
```

Java

```
java.util.function.BiFunction<T1, T2, R>
```

- Interface fonctionnelle similaire à `Function` possédant une méthode `apply` avec la signature `R apply(T1 t1, T2 t2)`
- Paramètres d'entrée : variable de type `T1` et une deuxième de type `T2`
- Valeur de retour : variable de type `R`

© Achref EL MOU

Java

```
java.util.function.BiFunction<T1,T2,R>
```

- Interface fonctionnelle similaire à `Function` possédant une méthode `apply` avec la signature `R apply(T1 t1, T2 t2)`
- Paramètres d'entrée : variable de type `T1` et une deuxième de type `T2`
- Valeur de retour : variable de type `R`

Exemple

```
BiFunction <Integer, Integer, Integer> somme = (a, b) -> a + b;
```

Java

```
java.util.function.BiFunction<T1,T2,R>
```

- Interface fonctionnelle similaire à `Function` possédant une méthode `apply` avec la signature `R apply(T1 t1, T2 t2)`
- Paramètres d'entrée : variable de type `T1` et une deuxième de type `T2`
- Valeur de retour : variable de type `R`

Exemple

```
BiFunction <Integer, Integer, Integer> somme = (a, b) -> a + b;
```

Pour exécuter, on appelle la méthode `apply()`

```
int resultat = somme.apply(5, 7);  
System.out.println(resultat);  
// affiche 12
```


Java

```
java.util.function.BiFunction<T1, T2, R>
```

- Interface fonctionnelle similaire à `Function` possédant une méthode `apply` avec la signature `R apply(T1 t1, T2 t2)`
- Paramètres d'entrée : variable de type `T1` et une deuxième de type `T2`
- Valeur de retour : variable de type `R`

Exemple

```
BiFunction <Integer, Integer, Integer> somme = (a, b) -> a + b;
```

Pour exécuter, on appelle la méthode `apply()`

```
int resultat = somme.apply(5, 7);  
System.out.println(resultat);  
// affiche 12
```

Le chaînage est possible avec `andThen()`

Java

```
java.util.function.BinaryOperator<T>
```

- Similaire à `BiFunction` mais avec un seul type générique
- Interface fonctionnelle possédant une méthode `apply` avec la signature `T apply(T t)`
- Paramètres d'entrée : deux variables de type `T`
- Valeur de retour : valeur de type `T`

© Achref EL M...

Java

```
java.util.function.BinaryOperator<T>
```

- Similaire à `BiFunction` mais avec un seul type générique
- Interface fonctionnelle possédant une méthode `apply` avec la signature `T apply(T t)`
- Paramètres d'entrée : deux variables de type `T`
- Valeur de retour : valeur de type `T`

Exemple

```
BinaryOperator<Integer> somme = (a, b) -> a + b;
```

Java

```
java.util.function.BinaryOperator<T>
```

- Similaire à `BiFunction` mais avec un seul type générique
- Interface fonctionnelle possédant une méthode `apply` avec la signature `T apply(T t)`
- Paramètres d'entrée : deux variables de type `T`
- Valeur de retour : valeur de type `T`

Exemple

```
BinaryOperator<Integer> somme = (a, b) -> a + b;
```

Pour exécuter, on appelle la méthode `apply()`

```
System.out.println(somme.apply(5, 7));  
// affiche 12
```

Java

```
java.util.function.BinaryOperator<T>
```

- Similaire à `BiFunction` mais avec un seul type générique
- Interface fonctionnelle possédant une méthode `apply` avec la signature `T apply(T t)`
- Paramètres d'entrée : deux variables de type `T`
- Valeur de retour : valeur de type `T`

Exemple

```
BinaryOperator<Integer> somme = (a, b) -> a + b;
```

Pour exécuter, on appelle la méthode `apply()`

```
System.out.println(somme.apply(5, 7));  
// affiche 12
```

Le chaînage est possible avec `andThen()`

Java

```
java.util.function.Consumer<T>
```

- Interface fonctionnelle possédant une méthode `accept` avec la signature `void accept (T t)`
- Paramètre d'entrée : variable de type `T`
- Pas de valeur de retour : elle consomme l'objet reçu en paramètre (le modifier)

© Achref EL M...

Java

```
java.util.function.Consumer<T>
```

- Interface fonctionnelle possédant une méthode `accept` avec la signature `void accept (T t)`
- Paramètre d'entrée : variable de type `T`
- Pas de valeur de retour : elle consomme l'objet reçu en paramètre (le modifier)

Exemple

```
Consumer <Personne> ageIncrement =  
    p -> p.setAge (p.getAge () + 1);
```

Java

```
java.util.function.Consumer<T>
```

- Interface fonctionnelle possédant une méthode `accept` avec la signature `void accept (T t)`
- Paramètre d'entrée : variable de type `T`
- Pas de valeur de retour : elle consomme l'objet reçu en paramètre (le modifier)

Exemple

```
Consumer <Personne> ageIncrement =  
    p -> p.setAge(p.getAge() + 1);
```

Pour exécuter, on appelle la méthode `accept ()`

```
ageIncrement.accept(personne);  
System.out.println(personne);  
// affiche Personne [nom=el mouelhi, prenom=achref, age=35]
```


Java

```
java.util.function.Predicate<T>
```

- Interface fonctionnelle possédant une méthode `test` avec la signature `boolean test(T t)`
- Paramètre d'entrée : variable de type `T`
- Type de valeur de retour : un booléen précisant si le paramètre `t` respecte le test.

© Achref EL MOU

Java

```
java.util.function.Predicate<T>
```

- Interface fonctionnelle possédant une méthode `test` avec la signature `boolean test(T t)`
- Paramètre d'entrée : variable de type `T`
- Type de valeur de retour : un booléen précisant si le paramètre `t` respecte le test.

Exemple

```
Predicate <Personne> contrainteAge = p -> p.getAge() >= 18;
```

Java

```
java.util.function.Predicate<T>
```

- Interface fonctionnelle possédant une méthode `test` avec la signature `boolean test(T t)`
- Paramètre d'entrée : variable de type `T`
- Type de valeur de retour : un booléen précisant si le paramètre `t` respecte le test.

Exemple

```
Predicate <Personne> contrainteAge = p -> p.getAge() >= 18;
```

Pour exécuter, on appelle la méthode `test()`

```
if (contrainteAge.test(personne)) {  
    System.out.println("Vous êtes adulte");  
}  
// affiche Vous êtes adulte
```

Java

```
java.util.function.BiPredicate<T,R>
```

- Interface fonctionnelle possédant une méthode `test` avec la signature
`boolean test(T t, R r)`
- Paramètre d'entrée : une variable de type `T` et une de type `R`
- Type de valeur de retour : un booléen précisant si les paramètre `t` et `r` respectent le test.

© Achref EL MOU

Java

```
java.util.function.BiPredicate<T,R>
```

- Interface fonctionnelle possédant une méthode `test` avec la signature `boolean test(T t, R r)`
- Paramètre d'entrée : une variable de type `T` et une de type `R`
- Type de valeur de retour : un booléen précisant si les paramètre `t` et `r` respectent le test.

Exemple

```
BiPredicate <Personne, Integer> contrainteAge =  
    (p, x) -> p.getAge() >= x;
```

Java

```
java.util.function.BiPredicate<T,R>
```

- Interface fonctionnelle possédant une méthode `test` avec la signature
`boolean test(T t, R r)`
- Paramètre d'entrée : une variable de type `T` et une de type `R`
- Type de valeur de retour : un booléen précisant si les paramètres `t` et `r` respectent le test.

Exemple

```
BiPredicate <Personne, Integer> contrainteAge =  
    (p, x) -> p.getAge() >= x;
```

Pour exécuter, on appelle la méthode `test()`

```
if (contrainteAge.test(personne, 18)) {  
    System.out.println("Vous êtes adulte");  
}  
// affiche Vous êtes adulte
```

Java

```
java.util.function.IntFunction<T>
```

- Interface fonctionnelle possédant une méthode `apply` avec la signature `T apply(Integer t)`
- Paramètre d'entrée : variable de type `Integer`
- Valeur de retour : valeur de type `T`

© Achref EL MOU

Java

```
java.util.function.IntFunction<T>
```

- Interface fonctionnelle possédant une méthode `apply` avec la signature `T apply(Integer t)`
- Paramètre d'entrée : variable de type `Integer`
- Valeur de retour : valeur de type `T`

Exemple

```
IntFunction <String> parity = i -> (i % 2 == 0) ? "pair" : "impair";
```


Java

```
java.util.function.IntFunction<T>
```

- Interface fonctionnelle possédant une méthode `apply` avec la signature `T apply(Integer t)`
- Paramètre d'entrée : variable de type `Integer`
- Valeur de retour : valeur de type `T`

Exemple

```
IntFunction <String> parity = i -> (i % 2 == 0) ? "pair" : "impair";
```

Pour tester

```
System.out.println(parity.apply(4));  
// affiche pair  
  
System.out.println(parity.apply(5));  
// affiche impair
```

Java

```
java.util.function.Supplier<T>
```

- Interface fonctionnelle possédant une méthode `get` avec la signature `T get()`
- Paramètre d'entrée : aucun
- Valeur de retour : valeur de type `T`

© Achref EL MOU

Java

```
java.util.function.Supplier<T>
```

- Interface fonctionnelle possédant une méthode `get` avec la signature `T get()`
- Paramètre d'entrée : aucun
- Valeur de retour : valeur de type `T`

Exemple

```
Supplier<Double> reel = () -> Math.random() * 100;
```

Java

```
java.util.function.Supplier<T>
```

- Interface fonctionnelle possédant une méthode `get` avec la signature `T get()`
- Paramètre d'entrée : aucun
- Valeur de retour : valeur de type `T`

Exemple

```
Supplier<Double> reel = () -> Math.random() * 100;
```

Pour tester

```
System.out.println(reel.get());  
// affiche un nombre réel entre 0 et 100
```

Autres interfaces fonctionnelles prédéfinies

- `DoubleFunction`, `DoubleConsumer`,
`DoubleBinaryOperator`, `DoublePredicate`,
`DoubleSupplier`...
- `LongFunction`, `LongConsumer`, `LongBinaryOperator`,
`LongPredicate`, `LongSupplier`...
- `UnaryOperator`
- ...

Java

Références de méthodes

permet de définir une méthode abstraite d'une interface fonctionnelle

© Achref EL MOUELHI ©

Java

Références de méthodes

permet de définir une méthode abstraite d'une interface fonctionnelle

Syntaxe

```
FirstPart::secondPart
```

Java

Références de méthodes

permet de définir une méthode abstraite d'une interface fonctionnelle

Syntaxe

```
FirstPart :: secondPart
```

Explication

- `FirstPart` : le nom d'une classe, interface ou objet
- `SecondPart` : le nom d'une méthode

Java

Étant donné le contenu de la classe `Main` suivant

```
public class Main {  
    public static int somme(int x, int y) {  
        return x + y;  
    }  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
    }  
}
```

Java

Étant donné le contenu de la classe `Main` suivant

```
public class Main {  
    public static int somme(int x, int y) {  
        return x + y;  
    }  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
    }  
}
```

Hypothèse

On veut utiliser l'interface fonctionnelle `ICalcul` et implémenter la méthode abstraite `public int operationBinaire(int x, int y)` pour qu'elle retourne la somme de `x` et `y`.

Java

Un solution possible **mais redondante**

```
public class Main {  
    public static int somme(int x, int y) {  
        return x + y;  
    }  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        ICalcul iCalcul = (a, b) -> a + b;  
    }  
}
```

© Achref EL

Java

Un solution possible **mais redondante**

```
public class Main {  
    public static int somme(int x, int y) {  
        return x + y;  
    }  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        ICalcul iCalcul = (a, b) -> a + b;  
    }  
}
```

Avec Java 8, on a la possibilité de référencer une méthode existante

```
public class Main {  
    public static int somme(int x, int y) {  
        return x + y;  
    }  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        ICalcul iCalcul = Main::somme;  
    }  
}
```

Java

Exemple avec un Consumer

```
public class Main {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Consumer <Personne> afficher = System.out::print;  
        afficher.accept(personne);  
        // affiche Personne [nom=el mouelhi, prenom=achref, age=34]  
    }  
}
```

© Achref EL

Java

Exemple avec un Consumer

```
public class Main {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Consumer <Personne> afficher = System.out::print;  
        afficher.accept(personne);  
        // affiche Personne [nom=el mouelhi, prenom=achref, age=34]  
    }  
}
```

Exemple avec un constructeur

```
public class Main {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Supplier <Personne> constructeur = Personne::new;  
        Personne p = constructeur.get();  
        System.out.println(p);  
        // affiche Personne [nom=null, prenom=null, age=0]  
    }  
}
```

Quelques nouvelles méthodes pour les collections

- Plusieurs méthodes ont été ajoutées dans la hiérarchie des collections
- La plupart de ces méthodes utilisent les expressions lambda
- Objectif : éviter les itérations, simplifier l'écriture de code et améliorer la lisibilité

Java

Considérons la liste suivante

```
List<Integer> liste = new ArrayList<Integer>(Arrays.asList(2, 7, 1, 3))  
;
```

Pour parcourir une liste, on peut utiliser la méthode `forEach`

```
liste.forEach(elt -> System.out.println(elt));
```

Affiche :

2
7
1
3

Une deuxième écriture de `forEach` avec les références de méthodes

```
liste.forEach(System.out::println);
```

Affiche :

2
7
1
3

Java

Pour supprimer selon une condition, on utilise la méthode `removeIf()`

```
liste.removeIf(elt -> elt > 6);  
liste.forEach(System.out::println);
```

Affiche :

2
1
3

Pour modifier tous les élément de la liste, on utilise la méthode `replaceAll()`

```
liste.replaceAll(elt -> elt + 6);  
liste.forEach(System.out::println);
```

Affiche :

8
13
7
9

L'API Stream (disponible depuis Java 8)

- Nouvelle API pour la manipulation de données remplaçant l'API Iterator
- Simplifiant la recherche, le filtrage et la manipulation de données...
- Ne modifiant pas forcément la source de données

L'API Stream (disponible depuis Java 8)

- Nouvelle API pour la manipulation de données remplaçant l'API Iterator
- Simplifiant la recherche, le filtrage et la manipulation de données...
- Ne modifiant pas forcément la source de données

Les méthodes de Stream se trouvent dans

```
java.util.stream;
```

Un stream

- un chaînage d'opération
- 0 ou plusieurs opérations intermédiaires
 - retourne toujours un stream
 - n'est exécuté qu'en appelant l'opération terminale
- 1 opération terminale
 - consomme le stream
 - exécute les opérations intermédiaires

Java

Exemples d'opérations intermédiaires

- `map()` : permet d'effectuer un traitement sur une liste sans la modifier réellement
- `filter(Predicate)` : permet de filtrer des éléments selon un prédicat
- ...

© Achref EL MOUËL

Java

Exemples d'opérations intermédiaires

- `map()` : permet d'effectuer un traitement sur une liste sans la modifier réellement
- `filter(Predicate)` : permet de filtrer des éléments selon un prédicat
- ...

Exemples d'opérations finales

- `forEach(Consumer)`
- `reduce()` : permet de réduire une liste en une seule valeur
- `count()` : permet de compter le nombre d'élément d'une liste
- `collect()` : permet de récupérer le résultat des opérations successives sous une certaine forme à spécifier
- ...

Un Stream peut être construit à partir de

- une suite de données
- un tableau
- une liste

Java

Pour construire un Stream à partir d'une suite de données

```
Stream stream = Stream.of(2, 7, 1, 3);
```

© Achref EL MOUELHI ©

Java

Pour construire un Stream à partir d'une suite de données

```
Stream stream = Stream.of(2, 7, 1, 3);
```

On peut utiliser la méthode `of()` pour construire un Stream à partir d'un tableau

```
int[] tab = { 2, 7, 1, 3 };  
Stream stream = Stream.of(tab);
```

Java

Pour construire un Stream à partir d'une suite de données

```
Stream stream = Stream.of(2, 7, 1, 3);
```

On peut utiliser la méthode `of()` pour construire un Stream à partir d'un tableau

```
int[] tab = { 2, 7, 1, 3 };  
Stream stream = Stream.of(tab);
```

On peut utiliser la méthode `stream()` pour construire un Stream à partir d'une liste

```
List<Integer> liste = Arrays.asList(2, 7, 1, 3);  
Stream stream = liste.stream();
```

Java

Si les données sont de même type, on peut utiliser la généricité

```
List<Integer> liste = Arrays.asList(2, 7, 1, 3);  
Stream<Integer> stream = liste.stream();
```

Java

Pour parcourir et afficher une liste

```
stream.forEach(elt -> System.out.println(elt));
```

Affiche :

2

7

1

3

© Achref EL MOUELHI ©

Java

Pour parcourir et afficher une liste

```
stream.forEach(elt -> System.out.println(elt));
```

Affiche :

2
7
1
3

Une deuxième écriture possible de `forEach` avec les références de méthodes

```
stream.forEach(System.out::println);
```

Affiche :

2
7
1
3

Java

Exemple avec une opération intermédiaire `map` et une opération finale `forEach`

```
stream.map(elt -> elt + 2)  
      .forEach(elt -> System.out.println(elt));
```

Affiche :

4
9
3
5

Java

Exemple avec deux opérations intermédiaires `map` et `filter`

```
stream.map(elt -> elt + 2)
      .filter(elt -> elt > 3)
      .forEach(elt -> System.out.println(elt));
```

Affiche :

4
9
5

L'opération `reduce`

- permet de réduire le stream en une seule valeur (opération finale)
- retourne le résultat sous forme d'un `Optional`
- s'exprime avec une expression lambda avec deux paramètres d'entrée :
 - le premier correspond à la valeur de retour de la l'application précédente et
 - le deuxième contient l'élément courant

La méthode `reduce` retourne la somme si la liste filtrée n'est pas vide sinon elle ne retourne rien. Il faut donc tester la présence d'un résultat avant de l'afficher

```
Optional<Integer> somme = stream.map(elt -> elt + 2)
    .filter(elt -> elt > 3)
    .reduce((a, b) -> a + b);
if (somme.isPresent())
    System.out.println(somme.get());
// affiche 18
```

© Achref EL MOUELHI

La méthode `reduce` retourne la somme si la liste filtrée n'est pas vide sinon elle ne retourne rien. Il faut donc tester la présence d'un résultat avant de l'afficher

```
Optional<Integer> somme = stream.map(elt -> elt + 2)
    .filter(elt -> elt > 3)
    .reduce((a, b) -> a + b);
if (somme.isPresent())
    System.out.println(somme.get());
// affiche 18
```

Pour éviter le test, on peut faire

```
Optional<Integer> somme = stream.map(elt -> elt + 2)
    .filter(elt -> elt > 3)
    .reduce((a, b) -> a + b);
somme.ifPresent(System.out::print);
// affiche 18
```

La méthode `reduce` retourne la somme si la liste filtrée n'est pas vide sinon elle ne retourne rien. Il faut donc tester la présence d'un résultat avant de l'afficher

```
Optional<Integer> somme = stream.map(elt -> elt + 2)
    .filter(elt -> elt > 3)
    .reduce((a, b) -> a + b);
if (somme.isPresent())
    System.out.println(somme.get());
// affiche 18
```

Pour éviter le test, on peut faire

```
Optional<Integer> somme = stream.map(elt -> elt + 2)
    .filter(elt -> elt > 3)
    .reduce((a, b) -> a + b);
somme.ifPresent(System.out::print);
// affiche 18
```

Ou aussi

```
stream.map(elt -> elt + 2)
    .filter(elt -> elt > 3)
    .reduce((a, b) -> a + b)
    .ifPresent(System.out::print);
// affiche 18
```

Java

Pour éviter de retourner un `Optional`, on peut initialiser le premier paramètre `a` à 0.

```
int somme = stream.map(elt -> elt + 2)
                .filter(elt -> elt > 3)
                .reduce(0, (a, b) -> a + b);
System.out.println(somme);
// affiche 18
```

© Achref EL MOU

Java

Pour éviter de retourner un `Optional`, on peut initialiser le premier paramètre `a` à 0.

```
int somme = stream.map(elt -> elt + 2)
                .filter(elt -> elt > 3)
                .reduce(0, (a, b) -> a + b);
System.out.println(somme);
// affiche 18
```

Ou aussi

```
int somme = stream.map(elt -> elt + 2)
                .filter(elt -> elt > 3)
                .reduce((a, b) -> a + b)
                .orElse(0);
System.out.println(somme);
// affiche 18
```

Java

Les valeurs de la liste n'ont pas été modifiées

```
stream.map(elt -> elt + 2)
        .filter(elt -> elt > 3);
liste.forEach(elt -> System.out.println(elt));
```

Affiche :

2
7
1
3

© Achref EL MOUELHI ©

Java

Les valeurs de la liste n'ont pas été modifiées

```
stream.map(elt -> elt + 2)
    .filter(elt -> elt > 3);
liste.forEach(elt -> System.out.println(elt));
```

Affiche :

2
7
1
3

Pour enregistrer les modifications de `map` et `filter`

```
liste = stream.map(elt -> elt + 2)
    .filter(elt -> elt > 3)
    .collect(Collectors.toList());
liste.forEach(elt -> System.out.println(elt));
```

Affiche :

4
9
5

Remarque

Il est possible d'obtenir un résultat sous forme

- d'un `Set` avec la méthode `toSet ()`
- d'un `map ()` avec la méthode `toMap ()`
- ...

Java

On peut aussi compter le nombre d'éléments

```
long nbr = stream.map(elt -> elt + 2)
                .filter(elt -> elt > 5)
                .count();
System.out.println(nbr);
```

Affiche :

1

© Achref EL MOUËL

Java

On peut aussi compter le nombre d'éléments

```
long nbr = stream.map(elt -> elt + 2)
                .filter(elt -> elt > 5)
                .count();
System.out.println(nbr);
```

Affiche :

1

Pour chercher le min ou le max

```
int nbr = stream
        .max(Comparator.naturalOrder()).get();
System.out.println(nbr);
```

Affiche :

7

Pour tester s'il existe un élément qui vérifie une condition, on peut utiliser `anyMatch(Predicate)` qui retourne un booléen

```
boolean result = stream
    .map(elt -> elt + 2)
    .anyMatch(element -> element == 9);
System.out.println(result);
// affiche true
```

© Achref EL MOUELHI ©

Pour tester s'il existe un élément qui vérifie une condition, on peut utiliser `anyMatch(Predicate)` qui retourne un booléen

```
boolean result = stream
    .map(elt -> elt + 2)
    .anyMatch(element -> element == 9);
System.out.println(result);
// affiche true
```

Pour tester si tous les éléments vérifient une condition, on peut utiliser `allMatch(Predicate)` qui retourne un booléen

```
boolean result = stream
    .map(elt -> elt + 2)
    .allMatch(element -> element == 9);
System.out.println(result);
// affiche false
```

Pour tester s'il existe un élément qui vérifie une condition, on peut utiliser `anyMatch(Predicate)` qui retourne un booléen

```
boolean result = stream
    .map(elt -> elt + 2)
    .anyMatch(element -> element == 9);
System.out.println(result);
// affiche true
```

Pour tester si tous les éléments vérifient une condition, on peut utiliser `allMatch(Predicate)` qui retourne un booléen

```
boolean result = stream
    .map(elt -> elt + 2)
    .allMatch(element -> element == 9);
System.out.println(result);
// affiche false
```

Pour tester aucun élément ne vérifie une condition, on peut utiliser `noneMatch(Predicate)` qui retourne un booléen

```
boolean result = stream
    .map(elt -> elt + 2)
    .noneMatch(element -> element >= 15);
System.out.println(result);
// affiche true
```

Pour limiter le nombre d'éléments sélectionnés et trier la sortie

```
stream.map(elt -> elt + 2)
        .limit(3)
        .sorted()
        .forEach(System.out::println);
```

affiche :

3
4
9

© Achref EL MOUELHI ©

Pour limiter le nombre d'éléments sélectionnés et trier la sortie

```
stream.map(elt -> elt + 2)
        .limit(3)
        .sorted()
        .forEach(System.out::println);
```

affiche :

3
4
9

Pour calculer la moyenne

```
Double moyenne = stream.collect(Collectors.averagingInt(Integer::
    intValue));
System.out.println(moyenne);
// affiche 3.25
```

Pour limiter le nombre d'éléments sélectionnés et trier la sortie

```
stream.map(elt -> elt + 2)
        .limit(3)
        .sorted()
        .forEach(System.out::println);
```

affiche :

3
4
9

Pour calculer la moyenne

```
Double moyenne = stream.collect(Collectors.averagingInt(Integer::
    intValue));
System.out.println(moyenne);
// affiche 3.25
```

Pour calculer la somme

```
Integer somme = stream.collect(Collectors.summingInt(Integer::intValue)
    );
System.out.println(somme);
// affiche 13
```


Exercice 1

- Écrire un programme qui demande à l'utilisateur de saisir un entier
- tant que la valeur saisie est positive, on la rajoute dans un `ArrayList`, on affiche son nombre d'occurrence et on demande une nouvelle saisie

Java

Solution

```
ArrayList<Integer> liste = new ArrayList();
Scanner scanner = new Scanner(System.in);
int i;
System.out.println("Saisir un entier");
while((i = scanner.nextInt()) > 0) {
    final int j = i;
    liste.add(i);
    System.out.println(i + " apparaît " + liste.stream
        ().filter(a -> a == j).count() + " fois");
    System.out.println("Saisir un entier");
}
```

Remarque

Si toutes les données d'un Stream sont des nombres, on peut utiliser des Stream particuliers :

- IntStream
- DoubleStream
- LongStream

Java

Si toutes les valeurs sont de type entier, on peut utiliser `IntStream`

```
IntStream stream = IntStream.of(1, 2, 3);  
stream.forEach(System.out::println);
```

Affiche :

1
2
3

© Achref EL MOUELHI ©

Java

Si toutes les valeurs sont de type entier, on peut utiliser `IntStream`

```
IntStream stream = IntStream.of(1, 2, 3);  
stream.forEach(System.out::println);
```

Affiche :

1
2
3

On peut aussi obtenir un `IntStream` à partir d'un tableau prédéfini

```
int [] tab = { 1, 2, 3 };  
IntStream stream = Arrays.stream(tab);  
stream.forEach(System.out::println);
```

Affiche :

1
2
3

Java

On peut aussi obtenir un `IntStream` à partir d'une liste prédéfinie

```
List<Integer> liste = Arrays.asList(1, 2, 3);  
IntStream stream = liste.stream().mapToInt(Integer::intValue);  
stream.forEach(System.out::println);
```

Affiche :

1
2
3

Arrays.stream() VS Stream.of()

- La méthode `Arrays.stream()` fonctionne seulement avec les tableaux de type primitif `int[]`, `long[]`, et `double[]`.
- Sa valeur de retour est respectivement `IntStream`, `LongStream` et `DoubleStream`.
- La méthode `Stream.of()` retourne un `Stream` générique de type `T Stream`. Par conséquent, et contrairement à `Arrays.stream()`, elle accepte tous les types.

Java

Pour convertir `IntStream` **en** `Stream<Integer>`

```
IntStream stream = IntStream.of(1, 2, 3);  
Stream<Integer> ints = stream.boxed();  
ints.forEach(System.out::println);
```

Affiche :

1
2
3

Java

Exemple avec la méthode `range()`

```
var stream = IntStream.range(1, 3);  
stream.forEach(System.out::println);
```

Affiche :

1
2

© Achref EL MOUELHI ©

Java

Exemple avec la méthode `range()`

```
var stream = IntStream.range(1, 3);  
stream.forEach(System.out::println);
```

Affiche :

1
2

Exemple avec la méthode `rangeClosed()`

```
var stream = IntStream.rangeClosed(1, 3);  
stream.forEach(System.out::println);
```

Affiche :

1
2
3

Java

On peut aussi utiliser la méthode `iterate(valeurInitiale, expressionLambda)`

```
var stream = IntStream.iterate(1, i -> i + 1).limit  
    (3);  
stream.forEach(System.out::println);
```

Affiche :

1
2
3

Java

On peut aussi convertir IntStream en DoubleStream

```
IntStream stream = IntStream.of(1, 2, 3);  
DoubleStream stream1 = stream.mapToDouble(num -> num * 3);  
  
stream1.forEach(System.out::println);
```

Affiche :

3
6
9

© Achref EL MOUELHI

Java

On peut aussi convertir `IntStream` **en** `DoubleStream`

```
IntStream stream = IntStream.of(1, 2, 3);  
DoubleStream stream1 = stream.mapToDouble(num -> num * 3);  
  
stream1.forEach(System.out::println);
```

Affiche :

3
6
9

On peut aussi ignorer quelques valeurs d'un Stream avec `skip()`

```
IntStream stream = IntStream.range(0, 6);  
stream.skip(3).forEach(System.out::println);
```

Affiche :

3
4
5

Java

Pour calculer la somme des valeurs d'un `IntStream`

```
IntStream stream = IntStream.range(0, 6);  
System.out.println(stream.sum());
```

Affiche :

15

© Achref EL MOUELHI

Java

Pour calculer la somme des valeurs d'un `IntStream`

```
IntStream stream = IntStream.range(0, 6);  
System.out.println(stream.sum());
```

Affiche :

15

Et pour calculer le maximum

```
IntStream stream = IntStream.range(0, 6);  
System.out.println(stream.max().orElse(-1));
```

Affiche :

5

Java

L'API Date-Time (disponible depuis Java 8)

- Nouvelle API pour la manipulation de date
- Simplifiant la manipulation, la recherche et la comparaison des dates
- Offrant une possibilité de travailler
 - soit sur le temps machine (Timestamp ou le nombre de secondes écoulées depuis 01/01/1970) : **Instant**
 - soit sur le temps humain (jour, mois, année, heure, minute, seconde...) : **LocalDateTime**, **LocalDate**, **LocalTime**...

Java

L'API Date-Time (disponible depuis Java 8)

- Nouvelle API pour la manipulation de date
- Simplifiant la manipulation, la recherche et la comparaison des dates
- Offrant une possibilité de travailler
 - soit sur le temps machine (Timestamp ou le nombre de secondes écoulées depuis 01/01/1970) : **Instant**
 - soit sur le temps humain (jour, mois, année, heure, minute, seconde...) : **LocalDateTime**, **LocalDate**, **LocalTime**...

Les classes de l'API Time se trouvent dans

```
java.time.*;
```

Avantages de l'API Date-Time

- Richesse en terme de fonctionnalité
- Clarté
- Simplicité

Plusieurs classes pour le temps humain

- `LocalDateTime` : **date** + **heure**
- `LocalDate` : **date**
- `LocalTime` : **heure**
- `ZonedDateTime` : **date** + **heure** + **fuseau horaire**
- ...

Pour obtenir une date, on fait appel à

- `now()` : pour obtenir une instance date et/ou l'heure courante
- `of()` : pour obtenir une instance à partir des données passées en paramètres
- `parser()` : pour obtenir une instance à partir d'une chaîne de caractère
- `from()` : pour obtenir une instance en convertissant des données passées en paramètres
- ...

Pour obtenir la date et l'heure actuelle

```
LocalDateTime localDateTime = LocalDateTime.from(ZonedDateTime.now());  
System.out.println("Date et heure actuelle : " + localDateTime);  
// affiche Date et heure courante : 2018-07-30T03:09:01.874
```

© Achref EL MOUELHI ©

Pour obtenir la date et l'heure actuelle

```
LocalDateTime localDateTime = LocalDateTime.from(ZonedDateTime.now());  
System.out.println("Date et heure actuelle : " + localDateTime);  
// affiche Date et heure courante : 2018-07-30T03:09:01.874
```

Ou tout simplement

```
LocalDateTime localDateTime = LocalDateTime.now();  
System.out.println("Date et heure actuelle : " + localDateTime);  
// affiche Date et heure courante : 2018-07-30T03:09:01.874
```

© Achref EL ME

Pour obtenir la date et l'heure actuelle

```
LocalDateTime localDateTime = LocalDateTime.from(ZonedDateTime.now());  
System.out.println("Date et heure actuelle : " + localDateTime);  
// affiche Date et heure courante : 2018-07-30T03:09:01.874
```

Ou tout simplement

```
LocalDateTime localDateTime = LocalDateTime.now();  
System.out.println("Date et heure actuelle : " + localDateTime);  
// affiche Date et heure courante : 2018-07-30T03:09:01.874
```

Pour obtenir la date (sans l'heure)

```
LocalDate localDate = localDateTime.toLocalDate();  
System.out.println("Date actuelle : " + localDate);  
// affiche Date actuelle : 2018-07-30
```

Pour obtenir la date et l'heure actuelle

```
LocalDateTime localDateTime = LocalDateTime.from(ZonedDateTime.now());  
System.out.println("Date et heure actuelle : " + localDateTime);  
// affiche Date et heure courante : 2018-07-30T03:09:01.874
```

Ou tout simplement

```
LocalDateTime localDateTime = LocalDateTime.now();  
System.out.println("Date et heure actuelle : " + localDateTime);  
// affiche Date et heure courante : 2018-07-30T03:09:01.874
```

Pour obtenir la date (sans l'heure)

```
LocalDate localDate = localDateTime.toLocalDate();  
System.out.println("Date actuelle : " + localDate);  
// affiche Date actuelle : 2018-07-30
```

Pour obtenir l'heure (sans la date)

```
LocalTime localTime = localDateTime.toLocalTime();  
System.out.println("Heure actuelle : " + localTime);  
// affiche Heure actuelle : 03:14:40.495
```


Java

Il est possible de construire une date en passant les différents paramètres : année, mois, jour, heure, minute... (plusieurs surcharges possibles pour cette méthode)

```
LocalDateTime dateHeureNaissance = LocalDateTime.of(1985, 07, 30, 01,
    30, 20);
System.out.println(dateHeureNaissance);
// affiche 1985-07-30T01:30:20
```

© Achref EL MOUELH

Java

Il est possible de construire une date en passant les différents paramètres : année, mois, jour, heure, minute... (plusieurs surcharges possibles pour cette méthode)

```
LocalDateTime dateHeureNaissance = LocalDateTime.of(1985, 07, 30, 01, 30, 20);  
System.out.println(dateHeureNaissance);  
// affiche 1985-07-30T01:30:20
```

Ou en utilisant les énumérations

```
LocalDateTime dateHeureNaissance = LocalDateTime.of(1985, Month.JULY, 30, 01, 30, 20);  
System.out.println(dateHeureNaissance);  
// affiche 1985-07-30T01:30:20
```

Java

Il est possible de construire une date en passant les différents paramètres : année, mois, jour, heure, minute... (plusieurs surcharges possibles pour cette méthode)

```
LocalDateTime dateHeureNaissance = LocalDateTime.of(1985, 07, 30, 01, 30, 20);  
System.out.println(dateHeureNaissance);  
// affiche 1985-07-30T01:30:20
```

Ou en utilisant les énumérations

```
LocalDateTime dateHeureNaissance = LocalDateTime.of(1985, Month.JULY, 30, 01, 30, 20);  
System.out.println(dateHeureNaissance);  
// affiche 1985-07-30T01:30:20
```

Il est possible de construire que la date (ou que l'heure)

```
LocalDate dateNaissance = LocalDate.of(1985, Month.JULY, 30);  
System.out.println(dateNaissance);  
// affiche 1985-07-30
```

Java

Il est possible de construire une date à partir d'une chaîne de caractère

```
String dateString = "1985-07-30";  
LocalDate dateFromString = LocalDate.parse(dateString);  
System.out.println(dateFromString);  
// affiche 1985-07-30
```

© Achref EL MOUELHI ©

Java

Il est possible de construire une date à partir d'une chaîne de caractère

```
String dateString = "1985-07-30";  
LocalDate dateFromString = LocalDate.parse(dateString);  
System.out.println(dateFromString);  
// affiche 1985-07-30
```

On peut récupérer un élément de la date

```
System.out.println(dateFromString.getDayOfMonth());  
// affiche 30  
  
System.out.println(dateFromString.getDayOfYear());  
// affiche 211  
  
System.out.println(dateFromString.getDayOfMonth());  
// affiche 30  
  
System.out.println(dateFromString.getMonthValue());  
// affiche 7  
  
System.out.println(dateFromString.getYear());  
// affiche 1985
```

Java

Pour obtenir le mois ou le jour en toutes lettres

```
System.out.println(dateFormString.getDayOfWeek());  
// affiche TUESDAY
```

```
System.out.println(dateFormString.getMonth());  
// affiche JULY
```

© Achref EL MOUELHI

Java

Pour obtenir le mois ou le jour en toutes lettres

```
System.out.println(dateFormString.getDayOfWeek());  
// affiche TUESDAY
```

```
System.out.println(dateFormString.getMonth());  
// affiche JULY
```

Pour obtenir le mois ou le jour en toutes lettres en français

```
System.out.println(dateFormString.getDayOfWeek().getDisplayName(  
    TextStyle.FULL, Locale.FRANCE));  
// affiche mardi  
  
System.out.println(dateFormString.getMonth().getDisplayName(TextStyle.  
    FULL, Locale.FRANCE));  
// affiche juillet
```

Java

Pour obtenir le mois ou le jour en toutes lettres

```
System.out.println(dateFormString.getDayOfWeek());  
// affiche TUESDAY  
  
System.out.println(dateFormString.getMonth());  
// affiche JULY
```

Pour obtenir le mois ou le jour en toutes lettres en français

```
System.out.println(dateFormString.getDayOfWeek().getDisplayName(  
    TextStyle.FULL, Locale.FRANCE));  
// affiche mardi  
  
System.out.println(dateFormString.getMonth().getDisplayName(TextStyle.  
    FULL, Locale.FRANCE));  
// affiche juillet
```

Pour le jour, on peut faire aussi

```
System.out.println(DayOfWeek.from(dateFormString).getDisplayName(  
    TextStyle.FULL, Locale.FRANCE));  
// affiche mardi
```


Java

On peut aussi incrémenter une date en ajoutant un nombre de jours, mois, années...

```
System.out.println(dateFormString.plus(2, ChronoUnit.DAYS));  
// affiche 1985-08-01  
  
System.out.println(dateFormString.plus(1, ChronoUnit.WEEKS));  
// affiche 1985-08-06  
  
System.out.println(dateFormString.plus(30, ChronoUnit.YEARS));  
// affiche 2015-07-30  
  
System.out.println(dateFormString.plus(1, ChronoUnit.MONTHS));  
// affiche 1985-08-30
```

Java

On peut aussi incrémenter une date en ajoutant un nombre de jours, mois, années...

```
System.out.println(dateFormString.plus(2, ChronoUnit.DAYS));  
// affiche 1985-08-01  
  
System.out.println(dateFormString.plus(1, ChronoUnit.WEEKS));  
// affiche 1985-08-06  
  
System.out.println(dateFormString.plus(30, ChronoUnit.YEARS));  
// affiche 2015-07-30  
  
System.out.println(dateFormString.plus(1, ChronoUnit.MONTHS));  
// affiche 1985-08-30
```

On peut aussi décrémenter une date en retirant un nombre de jours, mois, années...

```
System.out.println(dateFormString.minus(2, ChronoUnit.DAYS));  
// affiche 1985-07-28  
  
System.out.println(dateFormString.minus(1, ChronoUnit.WEEKS));  
// affiche 1985-07-23
```

Java

On peut aussi utiliser la classe `TemporalAdjusters` pour aller à un jour précis

```
LocalDate lundiSuivant = dateFromString.with(TemporalAdjusters.  
    next (DayOfWeek.MONDAY) );  
System.out.println(lundiSuivant);  
// affiche 1985-08-05
```

© Achref EL MOU

Java

On peut aussi utiliser la classe `TemporalAdjusters` pour aller à un jour précis

```
LocalDate lundiSuivant = dateFormString.with(TemporalAdjusters.  
    next (DayOfWeek.MONDAY) );  
System.out.println(lundiSuivant);  
// affiche 1985-08-05
```

Ou aussi

```
LocalDate lundiPrecedent = dateFormString.with(  
    TemporalAdjusters.previous (DayOfWeek.MONDAY) );  
System.out.println(lundiPrecedent);  
// affiche 1985-07-29
```

Considérons les deux dates suivantes

```
LocalDateTime dateHeureNaissance = LocalDateTime.of(1985, Month.JULY,  
    30, 01, 30, 20);  
LocalDateTime dateDuJour = LocalDateTime.of(2018, Month.JULY, 30, 03,  
    59, 20);
```

© Achref EL MOUELHI ©

Considérons les deux dates suivantes

```
LocalDateTime dateHeureNaissance = LocalDateTime.of(1985, Month.JULY,  
    30, 01, 30, 20);  
LocalDateTime dateDuJour = LocalDateTime.of(2018, Month.JULY, 30, 03,  
    59, 20);
```

Pour comparer deux dates, on peut utiliser la méthode `compareTo()` qui commence par comparer les années, puis les mois...

```
System.out.println(dateHeureNaissance.compareTo(dateDuJour));  
// affiche -33 (différence d'année) car dateHeureNaissance < dateDuJour  
System.out.println(dateDuJour.compareTo(dateHeureNaissance));  
// affiche 33 car dateHeureNaissance > dateDuJour  
System.out.println(dateHeureNaissance.compareTo(dateHeureNaissance));  
// affiche 0 car dateHeureNaissance = dateDuJour
```

Considérons les deux dates suivantes

```
LocalDateTime dateHeureNaissance = LocalDateTime.of(1985, Month.JULY,
    30, 01, 30, 20);
LocalDateTime dateDuJour = LocalDateTime.of(2018, Month.JULY, 30, 03,
    59, 20);
```

Pour comparer deux dates, on peut utiliser la méthode `compareTo()` qui commence par comparer les années, puis les mois...

```
System.out.println(dateHeureNaissance.compareTo(dateDuJour));
// affiche -33 (différence d'année) car dateHeureNaissance < dateDuJour
System.out.println(dateDuJour.compareTo(dateHeureNaissance));
// affiche 33 car dateHeureNaissance > dateDuJour
System.out.println(dateHeureNaissance.compareTo(dateHeureNaissance));
// affiche 0 car dateHeureNaissance = dateDuJour
```

On peut aussi utiliser les méthodes `isBefore()`, `isAfter()` ou `isEqual()`

```
System.out.println(dateHeureNaissance.isAfter(dateDuJour));
// affiche false
System.out.println(dateDuJour.isBefore(dateHeureNaissance));
// affiche false
System.out.println(dateHeureNaissance.isEqual(dateHeureNaissance));
// affiche true
```

Java

Pour connaître la différence (par exemple en années) entre deux dates

```
System.out.println(ChronoUnit.YEARS.between(  
    dateHeureNaissance, dateDuJour));  
// affiche 33
```

© Achref EL MOU

Java

Pour connaître la différence (par exemple en années) entre deux dates

```
System.out.println(ChronoUnit.YEARS.between(  
    dateHeureNaissance, dateDuJour));  
// affiche 33
```

Ou aussi en utilisant la méthode `between` de la classe `Period`

```
Period periode = Period.between(dateHeureNaissance.  
    toLocalDate(), dateDuJour.toLocalDate());  
System.out.println(periode.getYears());  
// affiche 33
```

Java

Pour connaître la différence (par exemple en secondes) entre deux heures

```
System.out.println(ChronoUnit.SECONDS.between(dateHeureNaissance.  
    toLocalTime(), dateDuJour.toLocalTime()));  
// affiche 8940
```

© Achref EL MOUELHI ©

Java

Pour connaître la différence (par exemple en secondes) entre deux heures

```
System.out.println(ChronoUnit.SECONDS.between(dateHeureNaissance.  
    toLocalTime(), dateDuJour.toLocalTime()));  
// affiche 8940
```

Sans convertir en `LocalTime`, le résultat n'est pas le même car on compare deux dates et non seulement deux heures

```
Duration duree = Duration.between(dateHeureNaissance, dateDuJour);  
System.out.println(duree.getSeconds());  
// affiche 1041388140
```

Java

Pour connaître la différence (par exemple en secondes) entre deux heures

```
System.out.println(ChronoUnit.SECONDS.between(dateHeureNaissance.  
    toLocalTime(), dateDuJour.toLocalTime()));  
// affiche 8940
```

Sans convertir en `LocalTime`, le résultat n'est pas le même car on compare deux dates et non seulement deux heures

```
Duration duree = Duration.between(dateHeureNaissance, dateDuJour);  
System.out.println(duree.getSeconds());  
// affiche 1041388140
```

Une deuxième solution avec la méthode `between` de la classe `Duration`

```
Duration duree = Duration.between(dateHeureNaissance.toLocalTime(),  
    dateDuJour.toLocalTime());  
System.out.println(duree.getSeconds());  
// affiche 8940
```

Java

Pour connaître le fuseau horaire actuel

```
System.out.println("Fuseau horaire par défaut : " +  
    ZoneId.systemDefault());  
// affiche Fuseau horaire par défaut : Europe/Paris
```

© Achref EL MOUAD

Java

Pour connaître le fuseau horaire actuel

```
System.out.println("Fuseau horaire par défaut : " +  
    ZoneId.systemDefault());  
// affiche Fuseau horaire par défaut : Europe/Paris
```

Pour connaître les règles appliquées aux heures

```
System.out.println("Règles appliquées aux heures : "  
    + ZoneId.systemDefault().getRules());  
// affiche Règles appliquées aux heures : ZoneRules[  
    currentStandardOffset=+01:00]
```

Java

Pour obtenir une date avec le fuseau horaire

```
ZonedDateTime zonedDateTime = ZonedDateTime.now();  
System.out.println(zonedDateTime);  
// affiche 2019-09-22T10:33:18.496+02:00[Europe/  
Paris]
```

© Achref EL MOUL

Java

Pour obtenir une date avec le fuseau horaire

```
ZonedDateTime zonedDateTime = ZonedDateTime.now();  
System.out.println(zonedDateTime);  
// affiche 2019-09-22T10:33:18.496+02:00[Europe/  
Paris]
```

Pour obtenir une date selon un motif défini

```
DateTimeFormatter dateTimeFormatter =  
    DateTimeFormatter.ofPattern("'Le' dd '/' MMMM '/'  
    yyyy");  
System.out.println(zonedDateTime.format(  
    dateTimeFormatter));  
// affiche Le 22 / septembre / 2019
```


Java

Comment connaître tous les fuseaux horaires ?

```
ZoneId.getAvailableZoneIds().forEach(System.out::println);  
// affiche une liste longue de fuseaux horaires
```

© Achref EL MOUELHI ©

Java

Comment connaître tous les fuseaux horaires ?

```
ZoneId.getAvailableZoneIds().forEach(System.out::println);  
// affiche une liste longue de fuseaux horaires
```

Pour obtenir la date et l'heure de Paris, on peut faire

```
ZoneId paris = ZoneId.of("Europe/Paris");  
ZonedDateTime dateHeureParis = ZonedDateTime.of(LocalDate.now(),  
    paris);  
System.out.println(dateHeureParis);  
// affiche 2019-09-22T20:09:27.949+02:00[Europe/Paris]
```

Java

Comment connaître tous les fuseaux horaires ?

```
ZoneId.getAvailableZoneIds().forEach(System.out::println);  
// affiche une liste longue de fuseaux horaires
```

Pour obtenir la date et l'heure de Paris, on peut faire

```
ZoneId paris = ZoneId.of("Europe/Paris");  
ZonedDateTime dateHeureParis = ZonedDateTime.of(LocalDateTime.now(),  
    paris);  
System.out.println(dateHeureParis);  
// affiche 2019-09-22T20:09:27.949+02:00 [Europe/Paris]
```

Comment connaître la date et l'heure exacte de Michigan aux États-Unis ?

```
ZoneId michigan = ZoneId.of("US/Michigan");  
ZonedDateTime dateHeureMichigan = dateHeureParis.withZoneSameInstant(  
    michigan);  
System.out.println(dateHeureMichigan);  
// affiche 2019-09-22T14:09:27.949-04:00 [US/Michigan]
```

Java

Comment connaître le décalage horaire (Offset) par rapport à l'heure universelle

```
System.out.println(dateHeureParis.getOffset());  
// affiche +02:00
```

© Achref EL MOUËL

Java

Comment connaître le décalage horaire (Offset) par rapport à l'heure universelle

```
System.out.println(dateHeureParis.getOffset());  
// affiche +02:00
```

Pour construire une date avec le décalage horaire, on utilise la classe `OffsetDateTime`

```
OffsetDateTime dateAvecOffset = OffsetDateTime.of(  
    LocalDateTime.now(), ZoneOffset.ofHours(6));  
System.out.println(dateAvecOffset);  
// affiche 2019-09-22T20:34:34.138+06:00
```

Java

Pour obtenir l'heure universelle

```
Instant maintenant = Instant.now();  
System.out.println(maintenant);  
// affiche 2019-09-22T05:12:06.030Z
```

© Achref EL MOUELHI ©

Java

Pour obtenir l'heure universelle

```
Instant maintenant = Instant.now();  
System.out.println(maintenant);  
// affiche 2019-09-22T05:12:06.030Z
```

Pour obtenir le timestamp (nombre de secondes écoulées depuis 01/01/1970)

```
System.out.println(maintenant.getEpochSecond());  
// affiche 1569135019
```

Java

Pour obtenir l'heure universelle

```
Instant maintenant = Instant.now();  
System.out.println(maintenant);  
// affiche 2019-09-22T05:12:06.030Z
```

Pour obtenir le timestamp (nombre de secondes écoulées depuis 01/01/1970)

```
System.out.println(maintenant.getEpochSecond());  
// affiche 1569135019
```

Pour ajouter du temps à notre instant

```
Instant demain = maintenant.plus(1, ChronoUnit.DAYS);  
System.out.println(demain);  
// affiche 2019-09-23T06:56:22.998Z
```


Java

Pour soustraire du temps à notre instant

```
Instant hier = maintenant.minus(1, ChronoUnit.DAYS);  
System.out.println(hier);  
// affiche 2019-09-21T07:00:16.265Z
```

© Achref EL MOUELHI ©

Java

Pour soustraire du temps à notre instant

```
Instant hier = maintenant.minus(1, ChronoUnit.DAYS);  
System.out.println(hier);  
// affiche 2019-09-21T07:00:16.265Z
```

Remarque

Seules les constantes `ChronoUnit.DAYS` et `ChronoUnit.HALF_DAYS` fonctionnent avec les instants

Java

Pour soustraire du temps à notre instant

```
Instant hier = maintenant.minus(1, ChronoUnit.DAYS);  
System.out.println(hier);  
// affiche 2019-09-21T07:00:16.265Z
```

Remarque

Seules les constantes `ChronoUnit.DAYS` et `ChronoUnit.HALF_DAYS` fonctionnent avec les instants

Pour comparer deux instants, on peut utiliser la méthode `compareTo()`

```
System.out.println(maintenant.compareTo(hier));  
// affiche 1 car maintenant > hier  
System.out.println(hier.compareTo(maintenant));  
// affiche -1 car maintenant < hier  
System.out.println(maintenant.compareTo(maintenant));  
// affiche 0 car maintenant = hier
```

Java

Pour comparer deux instants, on peut aussi utiliser les méthodes `isBefore()`, `isAfter()` ou `isEqual()`

```
System.out.println(maintenant.isAfter(hier));  
// affiche true
```

```
System.out.println(maintenant.isBefore(hier));  
// affiche false
```

```
System.out.println(maintenant.isEqual(maintenant));  
// affiche true
```

Java

Date-time classes in Java	Modern class	Legacy class
Moment in UTC	<code>java.time. Instant</code>	<code>java.util. Date</code> <code>java.sql. Timestamp</code>
Moment with offset-from-UTC (hours-minutes-seconds)	<code>java.time. OffsetDateTime</code>	(lacking)
Moment with time zone (`Continent/Region`)	<code>java.time. ZonedDateTime</code>	<code>java.util. GregorianCalendar</code>
Date & Time-of-day (no offset, no zone) <u>Not</u> a moment	<code>java.time. LocalDateTime</code>	(lacking)

Les dates après et avant Java 8 (Source : Stack OverFlow)