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

Plan

Java

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()
}
```

Java

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
```

Java

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)

Java

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();
}
```

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
```

Java

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.

Java

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.afficherNomComplet(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);
  }
}
```

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 );
  }
};
```

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);
}
```

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;
```

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;
};
```

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;
};
```

Java

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;
};
```

Java

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 ?

Java

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`

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();
```

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)
);
```

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;
}
```

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

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 personneToInt 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 personneToInt 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

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`

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 t1, T t2)`
- Paramètres d'entrée : deux variables de type `T`
- Valeur de retour : valeur de type `T`

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 t1, T t2)`
- 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 t1, T t2)`
- 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 t1, T t2)`
- 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 consommer l'objet reçu en paramètre (le modifier)

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 consommer 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 consommer 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.

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 `accept()`

```
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.

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 paramètre `t` et `r` respectent le test.

Exemple

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

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

```
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`

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`

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
```

Java

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

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;
  }
}
```

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]
  }
}
```

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]
  }
}
```

Java

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

Java

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

Java

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;
```

Java

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
- `...`

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
- `...`

Java

Considérons la liste suivante

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

Pour parcourir et afficher une liste

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

Affiche :
2
7
1
3

Java

Considérons la liste suivante

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

Pour parcourir et afficher une liste

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

Affiche :
2
7
1
3

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

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

Affiche :
2
7
1
3

Java

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

```
liste.stream()
     .map(elt -> elt + 2) // ajoute 2 a chaque élément
     .forEach(elt -> System.out.println(elt));
```

Affiche :
4
9
3
5

Java

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

```
liste.stream()
     .map(elt -> elt + 2) // ajoute 2 a chaque élément
     .forEach(elt -> System.out.println(elt));
```

Affiche :
4
9
3
5

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

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

Affiche :
4
9

5

Java

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

Java

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 = liste.stream()
                  .map(elt -> elt + 2)
                  .filter(elt -> elt > 3)
                  .reduce((a, b) -> a + b);
if (somme.isPresent())
  System.out.println(somme.get());
```

Affiche :
18

Java

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 = liste.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 de retourner un `Optional`, on peut initialiser le paramètre `a` à 0.

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

Affiche :

18

Java

Les valeurs de la liste n'ont pas été modifiées

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

Affiche :
2
7
1
3

Java

Les valeurs de la liste n'ont pas été modifiées

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

Affiche :
4
9

5

Java

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

Affiche :
1

Java

On peut aussi compter le nombre d'éléments

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

Affiche :
7

Java

On peut aussi utiliser `anyMatch` qui fonctionne comme `filter` mais qui retourne un booléen

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

Affiche :
true

Java

On peut aussi utiliser `anyMatch` qui fonctionne comme `filter` mais qui retourne un booléen

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

Affiche :
true

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

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

Affiche :
3
4
9

Java

Exercice 1 : Étant donnée la liste suivante

```
ArrayList<Integer> liste = new ArrayList(Arrays.
   asList(2, 7, 2, 1, 3, 9, 2, 4, 2));
```

Écrire un programme Java qui permet de supprimer l'avant dernière occurrence du chiffre 2 de la liste précédente

Java

Exercice 1 : Étant donnée la liste suivante

```
ArrayList<Integer> liste = new ArrayList(Arrays.
   asList(2, 7, 2, 1, 3, 9, 2, 4, 2));
```

Écrire un programme Java qui permet de supprimer l'avant dernière occurrence du chiffre 2 de la liste précédente

Solution

```
liste.remove(liste.subList(0,liste.lastIndexOf(2)).
   lastIndexOf(2));
liste.forEach(System.out::println);
```

Java

Exercice 2

- 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 + " apparait " + liste.stream
     ().filter(a -> a == j).count() + " fois");
  System.out.println("Saisir un entier");
}
```

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 méthodes de Stream se trouvent dans

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

Java

Avantages de l'API Date-Time

- Richesse en terme de fonctionnalité
- Clarté
- Simplicité

Java

Plusieurs classes pour le temps humain

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

Java

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
```

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 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
```

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

On peut récupérer un élément de la date

```
System.out.println(dateFormString.getDayOfMonth());
// affiche 30

System.out.println(dateFormString.getDayOfYear());
// affiche 211

System.out.println(dateFormString.getDayOfMonth());
// affiche 30

System.out.println(dateFormString.getMonthValue());
// affiche 7

System.out.println(dateFormString.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
```

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

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);
```

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
```

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
```

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
```

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]
```

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
```

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]
```

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 ?

```
ZonedDateTime dateHeureParis = ZonedDateTime.of(LocalDateTime.now(),
   paris);
ZonedDateTime losAngelesDateTime = dateHeureParis.withZoneSameInstant(
   michigan);
System.out.println(losAngelesDateTime);
// 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
```

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
```

Java

Pour obtenir l'heure universelle

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

Pourobtenir 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
```

Pourobtenir 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
```

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	`java.time.` **`Instant`**	`java.util.` ~~**`Date`**~~ `java.sql.` ~~**`Timestamp`**~~
Moment with offset-from-UTC (hours-minutes-seconds)	`java.time.` **`OffsetDateTime`**	(lacking)
Moment with time zone (`Continent/Region`)	`java.time.` **`ZonedDateTime`**	`java.util.` ~~**`GregorianCalendar`**~~
Date & Time-of-day (no offset, no zone) **Not a moment**	`java.time.` **`LocalDateTime`**	(lacking)

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