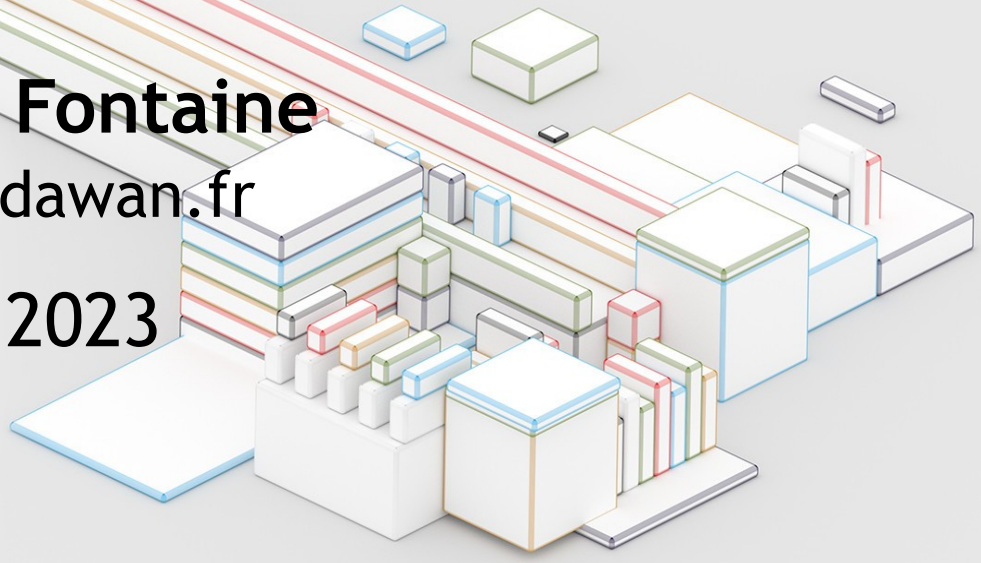
Nouveautés Java 8

Christophe Fontaine
cfontaine@dawan.fr

16/01/2023



API Time

En java 8, il existe 3 API de gestion du temps :

- **`java.util.Date`** (depuis le jdk 1.0)
- **`java.util.Calendar`** (depuis le jdk 1.1)
- **`java.time`** (depuis le jdk 8)

  Nouvelle API (JSR-310) inspiré par Joda Time

  Elle est basée sur 2 modèles de conception du temps :

  - **Temps Machine :** un entier augmentant depuis l'epoch (01 janvier 1970 00h00min00s0ms0ns)
  - **Temps Humain :** la succession de champs ayant une unité (année, mois, jours, heure, etc.)

Principes architecturaux

- **Immuabilité et thread safety**

  Les classes centrales de l'API sont immuables
  - Pas de problèmes de concurrence
  - Objets simples à créer, à utiliser et à tester
- **Chaînage**
  - Les méthodes chaînables rendent le code plus lisible
  - Des méthodes de type factory (`now()`, `from()` …) sont utilisées à la place de constructeurs
- **Clarté**
  - Chaque méthode définie clairement ce qu'elle fait
  - Passer un paramètre `null` à une méthode (sauf quelques cas particuliers) provoquera la levée d'un `NullPointerException`

Principes architecturaux

- Les méthodes de validation (qui prennent des objets en paramètre et retournent un booléen) retournent généralement false lorsque null est passé

- **Extensibilité**
  - Le design pattern Stratégie utilisé à travers l'API permet son extension en évitant toute confusion

    Par exemple: Les classes de l' API sont basées sur le système de calendrier ISO-8601 mais l'on peut utiliser les calendriers non-ISO comme le calendrier Impérial Japonais (inclus dans l'API) ou même créer son propre calendrier

Instant

Un point sur une timeline

- de **Instant.MIN** (-1Md) à **Instant.MAX** (+ 1Md) d'années
- **Instant.EPOCH** : 01 janvier 1970 00h 00min 00s 0ms 0ns
- **Instant.now() :** Instant actuelle
- **Instant.parse()** : Convertie une chaîne de caractère représentant un instant au format ISO-8601 (YYYY**-**MM**-**DD**T**HH**:**MM**:**SS.NN**Z**) en un objet de type Instant (Chaîne non valide → DateTimeParseException)
- **Instant.ofEpochSecond()/Instant.ofEpochMilliSecond()** : retourne un objet de type Instant à partir d'un offset de x secondes ou millisecondes par rapport à l'epoch
- **Instant.now().getNano()** : retourne le nombre de nanosecondes de l'instant actuel retourné par Instant.now()

Duration

Une durée entre 2 Instant

- **Duration.ZERO** : représente une durée nulle
- **Duration.parse()** : Convertie une chaîne de caractères représentant une durée au format ISO-8601 (**P**n**DT**n**H**n**M**n.n**S**) en un objet de type Duration

  D→jour, **H**→heure, **M**→minute, **S**→seconde

  (Chaîne non valide →DateTimeParseException)
- **Duration.ofMillis()** : retourne un objet de type Duration à partir d'une chaîne de caractères représentant une durée en millisecondes

Temps humain

- **LocalDate** → date
- **LocalTime** → heure
- **LocalDateTime** → date et heure
  - La méthode statique **now()** donne la date et l'heure courante
  - La méthode statique **of()** permet de créer une date ou une heure spécifique

```
LocalDate d1 = LocalDate.now() ;
LocalDate d2 = LocalDate.of(2014, Month.APRIL ,25);
LocalTime t2 = LocalTime.of(9,30,0);
```

- **ZoneId** → identificateur de fuseau horaire

```
ZoneId zoneid1 = ZoneId.of("Europe/Paris");
ZoneId zoneid2 = ZoneId.of("Etc/GMT-3");
ZoneId zoneid3 = ZoneId.systemDefault()
```

Temps humain

- **ZoneOffset** → représente un décalage de fuseau horaire par rapport à Greenwich / UTC

```
ZoneOffset zoneUTC = ZoneOffset.UTC;
ZoneOffset zoneGMTPlus5 = ZoneOffset.ofHours(5);
```

- **ZonedDateTime** → date et heure avec un fuseau horaire éviter les fuseaux horaires tant que l'on n'en a pas besoin

```
ZoneId paris = ZoneId.of("Europe/Paris");
LocalDateTime now= LocalDateTime.now() ;
ZonedDateTime zone = ZonedDateTime.of(now, paris);
```

- **OffsetDateTime** → date et heure avec un décalage de fuseau horaire par rapport à Greenwich / UTC

```
OffsetDateTime offset = OffsetDateTime.now();
OffsetDateTime value =  offset.minusDays(240);
```

Period

- Une durée entre 2 objets **`LocalDate`**

  Identique au type **`Duration`** (même méthodes)

```
LocalDate now = LocalDate.now() ;
LocalDate dateOfBirth = LocalDate.of(1970, 04 ,25);
Period p = dateOfBirth.until(now) ;
System.out.println(p.getYears()) ;
long l = dateOfBirth.until(now, ChronoUnit.DAYS) ;
System.out.println(l);
System.out.println(p.getDays());
```

Interopérabilité entre l'ancienne et la nouvelle API

- **Instant et java.util.Date**

```
Date date = Date.from(instant);
Instant instant = date.toInstant();
```

- **Instant et Timestamp**

```
Timestamp time = Timestamp.from(instant);
Instant instant = time.toInstant();
```

- **LocalDate et java.sql.Date**

```
Date date = Date.valueOf(localDate);
LocalDate localDate = date.toLocalDate();
```

- **LocalTime et java.sql.Time**

```
Time time = Time.valueOf(localTime1);
LocalTime localTime = time.toLocalTime();
```

Variable d'interface

1) Les variables d'interface sont supposées être **public**, **static** et **final**

2) La valeur d'une variable d'interface doit être définie lors de sa déclaration (**final**)

```
public interface CanSwim {
    int MAXIMUM_DEPTH = 100;
    final static boolean UNDERWATER = true;
    public static final String TYPE = "Submersible";
}
```

Méthode d'interface par défaut

- Une méthode par défaut dans une interface définit une méthode abstraite avec une implémentation par défaut
- La classe qui implémente l'interface peut remplacer la méthode par défaut. Si elle ne le fait pas l'implémentation par défaut est utilisée
- La méthode par défaut permet aux développeurs d'ajouter de nouvelles méthodes à une interface sans casser les implémentations existantes de cette interface

1) Une méthode par défaut ne peut être déclarée que dans une interface
2) Une méthode par défaut doit être marquée avec le mot-clé **`default`**

   Dans ce cas on doit fournir un corps de méthode

Méthode d'interface par défaut

3) Une méthode par défaut ne peut pas être **`static`**, **`final`** ou **`abstract`**

4) Comme toutes les méthodes d'une interface, une méthode par défaut est supposée être public

- Quand une interface hérite d'un autre interface qui contient une méthode par défaut, il peut:
  - choisir d'ignorer la méthode par défaut
    ↳ l'implémentation par défaut sera utilisée
  - redéfinir la méthode par défaut
  - redéclarer la méthode comme abstraite

Méthode d'interface par défaut


```
public interface HasFins {
    public default int getNumberOfFins() {
        return 4;
    }
    public default double getLongestFinLength() {
        return 20.0;
    }
    public default boolean doFinsHaveScales() {
        return true;
    }
}

public interface SharkFamily extends HasFins {
    public default int getNumberOfFins() {
        return 8;
    }
    public double getLongestFinLength(); // force la classe
    // qui implémentera l'interface à redéfinir la méthode
    public boolean doFinsHaveScales() { // Ne compile pas
        return false;                   // il manque default
    }
}
```

Méthode d'interface par défaut

- Si une classe implémente deux interfaces qui ont des méthodes par défaut avec le même nom et la même signature, le compilateur générera une erreur

  Sauf si la sous-classe remplace les méthodes par défaut en double et supprime l'ambiguïté

```
public interface Walk {
    public default int getSpeed() {
        return 5;
    }
}

public interface Run {
    public default int getSpeed() {
        return 10;
    }
}
```

```
public class Cat implements Walk, Run {
    public int getSpeed() {
        return 1;
    }

    public static void main(String[] args) {
        System.out.println(new Cat().getSpeed());
    }
}
```

Méthode static d'interface

Les méthode statiques d'une interface sont identiques aux méthodes statiques défini dans les classes. Mais elles ne sont héritées d'aucune classe qui implémente l'interface

1) Comme toutes les méthodes d'une interface, une méthode statique est supposée être **`public`**
2) Pour référencer la méthode statique, une référence au nom de l'interface doit être utilisée

```
public interface Hop {
    static int getJumpHeight() {
        return 8;
    }
}
public class Bunny implements Hop {
    public void printDetails() {
        System.out.println(Hop.getJumpHeight());
        System.out.println(getJumpHeight()); // Ne compile pas
    }
}
```

Styles de programmation

- **Impérative**

  On écrit l'algorithme avec le flux de contrôle et des mesures explicites

- **Déclarative**

  On déclare ce qui doit être fait sans préoccupation pour le flux de contrôle

- **Fonctionnelle**

  Un paradigme de programmation déclaratif qui traite le calcul comme une série de fonctions et évite des données d'état et mutables pour faciliter la concurrence

Expressions Lambda

- Une expression lambda peut être assimilée à une fonction anonyme, ayant potentiellement accès au contexte (variables locales et/ou d'instance) du code appelant
- Le code de l'expression lambda sert d'implémentation pour une méthode abstraite de l'interface

  On peut les utiliser avec n'importe quel code Java utilisant une telle interface, à condition que les signatures de la méthode correspondent à celle de l'expression lambda
- **Syntaxe**

```
(paramètres) -> code
(paramètres) -> {code}
```

L'opérateur `->` sépare le(s) paramètre(s) du bloc de code

Expressions Lambda (Règle de syntaxe)

- Le type des paramètres peut être déclaré explicitement ou inféré par le compilateur
- S'il y a un paramètre, les parenthèses sont facultatives
- S'il n'y a pas de paramètre, les parenthèses sont obligatoires
- Le corps peut retourner une valeur ou pas
- S'il n'y a qu'une seule instruction, les accolades sont facultatives
- le retour doit être explicité si plusieurs instructions
- Une expression lambda peut avoir accès aux variables définies dans son contexte englobant (**seules les variables dont la valeur ne change pas peuvent être accédées**)

Expressions Lambda (Utilisation)

- **Paramètres : types implicites / explicites**

```
Arrays.asList("a","b","d").forEach(e -> System.out.print(e + "\t"));
Arrays.asList("a","b","d").forEach((String e) -> System.out.print(e+"\t"));
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

- **Référence vers une variable**

```
String sep = ",";
Arrays.asList("a", "b", "d").forEach((String e)->System.out.print(e sep));
```

- **Retour explicite**

```
Arrays.asList("a", "b", "d").sort((e1, e2) -> {
    int result = e1.compareTo(e2);
    return result;
});
```

- **Méthode anonyme**

```
Runnable r = () -> System.out.println("hello Dawan");
Thread t = new Thread(r); // ou :
Thread t = new Thread(() -> System.out.println("hello Dawan"));
```

Expressions Lambda (Utilisation)

- Une expression lambda n'est pas un **`Object`**

  C'est un objet sans identité

- Elle n'hérite pas d'**`Object`**, on ne peut donc pas appeler de méthode **`equals`**() ou autre dessus

- Pas d'instanciation (mot-clé new dessus)

Interfaces fonctionnelles

- Interface fonctionnelle = une interface définissant quoi accomplir dans une tâche mais pas la façon de le faire
- Interface annotée **`@FunctionalInterface`** avec **une seule méthode abstraite** (les méthodes `default` sont autorisées)

```
@FunctionalInterface
public interface Creator<T> {
    T create(String s1, String s2);
}
```

- Son implémentation est exprimée avec une expression lambda ou une référence de méthode
- Package : **java.util.function**

Interfaces fonctionnelles

Le package `java.util.function` fournit 43 interfaces fonctionnelles regrouper en 4 catégories :

- **Function** : une fonction qui a un ou plusieurs arguments et qui retourne un résultat

  méthode → **`apply`**`()`

  `Function, BiFunction, DoubleFunction, DoubleToIntFunction, DoubleToLongFunction, IntFunction, IntToDoubleFunction ...`

- **Consumer** : une fonction qui a un ou plusieurs arguments et qui ne retourne pas de valeur

  méthode → **accept**()

  `Consumer, BiConsumer, DoubleConsumer, IntConsumer, LongConsumer, ObjDoubleConsumer, ObjIntConsumer, ObjLongConsumer`

Interfaces fonctionnelles

- **Predicate** : une fonction qui a un ou plusieurs paramètres et qui retourne un booléen

  méthode → **`test()`**

  `Predicate, BiPredicate, DoublePredicate, IntPredicate, LongPredicate`

- **Supplier** : une fonction qui n'a pas de paramètre et qui retourne une valeur

  méthode → **`get()`**

  `Supplier, BooleanSupplier, DoubleSupplier, IntSupplier et LongSupplier`

Les interfaces qui existaient avant java 8 est qui ne comporte qu'une seule méthode (`Comparator, Runable …`) peuvent aussi être utilisé comme interface fonctionnelle

Interfaces fonctionnelles

- **Utilisation**

```
public class NameParser<T> {
    public T parse(String fullName, Creator<T> creator) {
        String[] tokens = fullName.split(" ");
        return creator.create(tokens[0], tokens[1]);
    }
}
```

Avant Java 8

```
Name name1 = parser.parse("Mohamed DERKAOUI", new Creator<Name>() {
    @Override
    public Name create(String firstName, String lastName) {
        return new Name(firstName, lastName);
    }
});
```

Avec Java 8

```
Name name2 = parser.parse("Mohamed DERKAOUI",
(firstName, lastName) -> new Name(firstName, lastName));
```

Références de méthodes

- Les références de méthodes offrent un raccourci syntaxique pour créer une expression lambda dont le but est d'invoquer une méthode statique ou non ou un constructeur
- **Syntaxe**

  `qualificateur::identifiant`

  - Un qualificateur est
    - un type pour les méthodes statiques et les constructeurs
    - un type ou une expression pour les méthodes d'instances
  - Un identifiant précise le nom de la méthode ou l'opérateur **`new`** pour un constructeur

Références de méthodes

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

Supplier<Double> random = Math::random;

Random r = new Random();

Supplier<Double> random2 = () -> r.nextDouble();

Supplier<Double> random2 = r::nextDouble;

Function<Random,Double> random3 = (Random random) ->
random.nextDouble();

Function<Random,Double> random3 = Random::nextDouble;

Function<String, Thread> factory = (String name) -> new
Thread(name);

Function<String, Thread> factory = Thread::new;
```

Méthode forEach

- Java fournit une nouvelle méthode **`forEach()`** pour itérer les éléments
- Elle est défini dans les interfaces **`Iterable`** et **`Stream`**
- Les classes **`Collection`** qui héritent l'interface Iterable peuvent utiliser la méthode **`forEach()`** pour itérer les éléments
- Cette méthode prend un seul paramètre qui est une interface fonctionnelle

```
List<String> names = Arrays.asList("Larry","Steve","John");
names.forEach(name -> {
    System.out.println(name);
});
names.forEach(System.out::println);
```

Stream

- Un **iterator** limité qui permet de parcourir des collections qui ne modifie pas la source des données
- On peut enchaîner des opérations →pipeline
- Il existe des méthodes :
  - **Intermédiaires :** elles renvoie un stream
  - **Terminales :** elles sont appelées pour clôturer le pipeline et renvoient le résultat
- `Stream<T>`
- `ParellelStream`

Stream

Opérations intermédiaires

- **`map()`** → renvoie un flux constitué des résultats de l'application d'une fonction donnée aux éléments de ce flux

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
list.stream().map(x -> x + 1).forEach(System.out::print);
// affiche -> 23456
```

- **`filter()`** → sélectionne les éléments selon le prédicat passé en argument

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
list.stream().filter(num -> num%2 == 0 ).forEach(System.out::print);
// affiche → 24
```

Stream

- **`sorted()`** → trier un flux

```
List<Integer> list = Arrays.asList(6, 2, 9, 1, 7);
list.stream().sorted().forEach(System.out::println);
// affiche -> 12679
```

- **`distinct()`** → supprimer les doublons

```
List<Integer> list = Arrays.asList(1, 2, 2, 3, 3, 4, 5);
list.stream().distinct().forEach(System.out::print);
// affiche → 12345
```

- **`limit()`** → limiter le nombre d'éléments

```
List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6);
entiers.stream().limit(3).forEach(System.out::print);
// affiche -> 1,2,3
```

Stream

- **`skip()`** → ignorer un certain nombre d'éléments

```
List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6);
entiers.stream().skip(3).forEach(System.out::println);
// 4,5,6
```

- **`peek()`** → tous les éléments du Stream en appliquant le Consumer sur chacun des éléments

Opérations terminales

- **`collect()`** → pour renvoie le résultat des opérations intermédiaires effectuées sur le flux

```
List<Integer> nbr = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> pair = nbr.stream()
                        .filter(x -> x % 2 == 0)
                        .collect(Collectors.toList());
System.out.println(pair); // affiche -> [2, 4]
```

Stream

- **`Reduce()`** → réduire les éléments d'un flux à une seule valeur
  prend un `BinaryOperator` comme paramètre

```
int[] nbr = {1, 2, 3};
int somme = Arrays.stream(nbr).reduce(0, (a, b) -> a + b);
System.out.println(somme);
// affiche -> 6
```

- **`forEach()`** → parcourir chaque élément du flux

…

Stream

Definitions

- ☑ A stream **is** a pipeline of functions that can be evaluated.
- ☑ Streams **can** transform data.
- ☒ A stream **is not** a data structure.
- ☒ Streams **cannot** mutate data.

Intermediate operations

- Always return streams.
- Lazily executed.

Common examples include:

Function	Preserves count	Preserves type	Preserves order
map	✓	✗	✓
filter	✗	✓	✓
distinct	✗	✓	✓
sorted	✓	✓	✗
peek	✓	✓	✓

Stream examples

Get the unique surnames in uppercase of the first 15 book authors that are 50 years old or over.

```
library.stream()
     .map(book -> book.getAuthor())
     .filter(author -> author.getAge() >= 50)
     .distinct()
     .limit(15)
     .map(Author::getSurname)
     .map(String::toUpperCase)
     .collect(toList());
```

Compute the sum of ages of all female authors younger than 25.

```
library.stream()
     .map(Book::getAuthor)
     .filter(a -> a.getGender() == Gender.FEMALE)
     .map(Author::getAge)
     .filter(age -> age < 25)
     .reduce(0, Integer::sum);
```

Terminal operations

- Return concrete types or produce a side effect.
- Eagerly executed.

Common examples include:

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach	side effect	to perform a side effect on elements

Parallel streams

Parallel streams use the common ForkJoinPool for threading.

```
library.parallelStream()...
```

or intermediate operation:

```
IntStream.range(1, 10).parallel()...
```

Useful operations

Grouping:

```
library.stream().collect(
    groupingBy(Book::getGenre));
```

Stream ranges:

```
IntStream.range(0, 20)...
```

Infinite streams:

```
IntStream.iterate(0, e -> e + 1)...
```

Max/Min:

```
IntStream.range(1, 10).max();
```

FlatMap:

```
twitterList.stream()
     .map(member -> member.getFollowers())
     .flatMap(followers -> followers.stream())
     .collect(toList());
```

Pitfalls

☒ Don't update shared mutable variables i.e.

```
List<Book> myList =
new ArrayList<>();
library.stream().forEach
(e -> myList.add(e));
```

☒ Avoid blocking operations when using parallel streams.

Source : REBELLABS

Optional

- **`Optional`** permet d'encapsuler un objet qui peut avoir ou pas une valeur, cela permet d'éviter l'utilisation d'un null pour marquer l'absence de valeur et est risquer une exception **`NullPointerException`**
- **Créer un objet Optional**

  avec les méthode de classe :
  - **`empty()`** pour créer un objet **`Optional`** vide

```
Optional<String> empty = Optional.empty();
```

  - **`of()`** pour créer un objet **`Optional`**, l'argument passé à la méthode, ne peut pas être `null`, sinon une exception `NullPointerException` est lancée

```
String str = "hello world";
Optional<String> opt = Optional.of(str);
```

Optional

- **`ofNullable()`** : idem **`of()`** mais accepte une valeur null, dans ce cas on crée un Objet **Optional** vide

```
String str = null;
Optional<String> opt = Optional.of(str)
```

- **Vérifier la présence d'une valeur**

  La méthode **`isPresent()`** retourne `true`, si il y a une valeur dans l'optional

```
Optional<String> opt = Optional.of("hello world");
System.out.println(opt.isPresent()); // affiche true
opt = Optional.ofNullable(null);     // affiche false
System.out.println(opt.isPresent());
```

- **Retour de la valeur avec la méthode `get()`**

```
Optional<String> opt = Optional.of("hello world");
String name = opt.get();
```

Optional

- **Action conditionnel avec `ifPresent()`**

  La méthode **`ifPresent()`** permet d'exécuter du code sur la valeur de l'objet Optional, si elle est différente de null

```
Optional<String> opt = Optional.of("hello world");
opt.ifPresent(str -> System.out.println(str.length()));
```

- **Valeur par défaut**
  - avec **`orElse()`**

    retourne la valeur contenu dans un objet Optional, si elle est présente

    Sinon elle revoie la valeur du paramètre de la méthode, qui agit comme une valeur par défaut

```
String str = null;
String res = Optional.ofNullable(str).orElse("défaut");
```

Optional

- avec **orElseGet()**

  Si la valeur facultative n'est pas présente, la méthode **orElseGet()** prend en paramètre une interface fonctionnelle et renvoie son résultat

```
String str = null;
String name = Optional.ofNullable(str)
                      .orElseGet(() -> "john");
```

- Exceptions avec **orElseThrow()**

  Si la valeur facultative n'est pas présente, la méthode **orElseThrow()** lance une exception

```
String str = null;
String strRep = Optional.ofNullable(str)
       .orElseThrow(IllegalArgumentException::new);
```