Java 8

Mohamed DERKAOUI

02/06/2019

Plan

- Disparition du PermGen Space au profit du Metaspace
- API Time
- Expressions lambda
- Méthodes default dans une interface
- Interfaces fonctionnelles
- Moteur de scripts Nashorn

Metaspace

Metaspace

- JDK 8 HotSpot JVM utilise à présent une mémoire native pour la représentation des metadatas de classes.
- Similaire à celui d'Oracle JRockit et IBM JVM's.
- Intérêt : éviter l'exception java.lang.OutOfMemoryError: PermGen space problems.
- Plus besoin de faire du tuning ou du profiling sur PermGen space. Il disparaît complètement .
- Cette nouvelle fonction n'élimine pas (par magie) les classes et les fuites mémoires du classloader.
Vous devez pister les problèmes par d'autres approches.

Allocation mémoire

- Les options JVM : PermSize and MaxPermSize sont ignorées et un warning apparaît au démarrage.
- Modèle d'allocation du Metaspace :
- - La majorité des allocations des metadatas de classes sont effectuées hors de la mémoire native.
- Les classes décrivant les metadatas de classes ont été supprimées.

Capacité du Metaspace

- Par défaut, l'attribution de métadonnées de classe est limitée par la quantité de mémoire disponible (la capacité dépendra bien sûr si vous utilisez une JVM 32 bits contre 64 bits ainsi que la disponibilité de la mémoire virtuelle OS) .
- Une nouvelle option JVM (**MaxMetaspaceSize**) pour fixer une limite. Si l'option n'est pas définie, le Metaspace sera dynamiquement redimensionner en fonction des demandes de l'application (au runtime).
- La garbage collection du Metaspace est déclenchée une fois que l'utilisation des métadonnées de classe ait atteint le MaxMetaspaceSize.

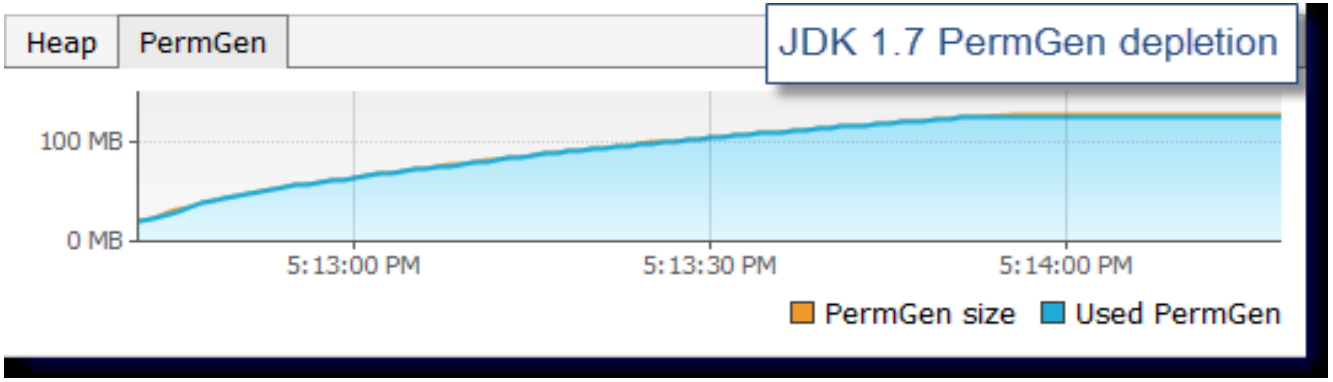
Monitoring

- Monitoring et tuning adéquats sont nécessaires pour limiter la fréquence ou le retard de la garbage collection.
- Les fréquentes garbage collections du Metaspace peuvent être un symptôme de fuites mémoire ou de dimensionnement insuffisant pour votre application.I
- Certaines données ont été déplacées vers le « Heap space ». Ce qui veut dire que vous observerez une augmentation de l'utilisation du heap memory.
- L'usage du Metaspace est disponible dans la sortie du GC log. JStat & JVisualVM ont été mis à jour pour afficher l'utilisation du metaspace.

PermGen vs Metaspace

- Exemple de simulation d'une grosse fuite mémoire
- JDK 1.7 @64 bits – PermGen depletion

Java heap space : 1024 MB
Java Max PermGen space : 128 MB



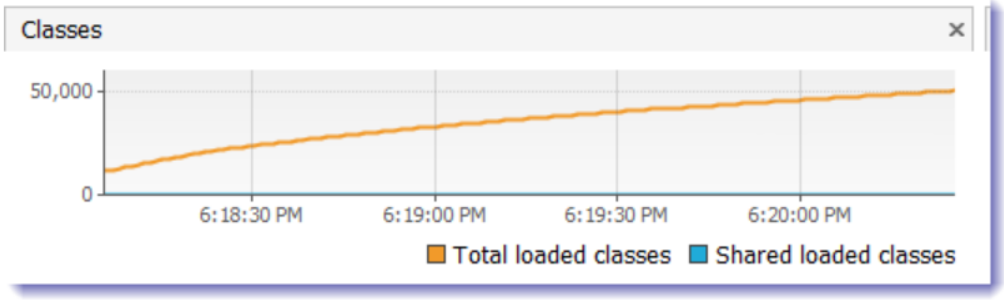
JDK 1.7 PermGen space depletion!

```
Heap
 PSYoungGen      total 316992K, used 0K [0x00
  eden space 281920K, 0% used [0x00000000eaab
  from space 35072K, 0% used [0x00000000fddc0
  to   space 32512K, 0% used [0x00000000fbe00
 ParOldGen       total 699072K, used 176774K
  object space 699072K, 25% used [0x00000000c
 PSPermGen       total 131072K, used 131071K
  object space 131072K, 99% used [0x00000000b
```

PermGen vs Metaspace (2)

- JDK 1.8 @64 bits
  Metaspace
  dynamic re-size

- Java heap space : 1024 MB
  Java Metaspace space : unbounded
  (default)




```
3.162: [GC (Metadata GC Threshold) [PSYoungGen: 199286K->29344K(305856K
3.219: [Full GC (Metadata GC Threshold) [PSYoungGen: 29344K->0K(305856K
6.153: [GC (Metadata GC Threshold) [PSYoungGen: 155324K->22688K(305856K
6.220: [Full GC (Metadata GC Threshold) [PSYoungGen: 22688K->0K(305856K
13.777: [GC (Metadata GC Threshold) [PSYoungGen:
13.881: [Full GC (Metadata GC Threshold) [PSYoun
26.105: [GC (Allocation Failure) [PSYoungGen: 26
36.925: [GC (Metadata GC Threshold) [PSYoungGen:
37.101: [Full GC (Metadata GC Threshold) [PSYoun
57.717: [GC (Allocation Failure) [PSYoungGen: 26
78.448: [GC (Allocation Failure) [PSYoungGen: 263904K->62848K(286656K)]
101.297: [GC (Metadata GC Threshold) [PSYoungGen: 277959K->62848K(22150
101.573: [Full GC (Metadata GC Threshold) [PSYoungGen: 62848K->0K(22150
121.502: [GC (Allocation Failure) [PSYoungGen: 158656K->24288K(254080K)
142.407: [GC (Allocation Failure) [PSYoungGen: 182944K->49824K(258176K)
Heap
 PSYoungGen      total 258176K, used 72641K [0x00000000eaab0000, 0x0000
  eden space 162880K, 14% used [0x00000000eaab0000,0x00000000ec0f8738,0
  from space 95296K, 52% used [0x00000000fa2f0000,0x00000000fd398030,0x
  to   space 91328K, 0% used [0x00000000f49c0000,0x00000000f49c0000,0x0
 ParOldGen       total 699072K, used 262735K [0x00000000c0000000, 0x000
  object space 699072K, 37% used [0x00000000c0000000,0x00000000d0093c10
 Metaspace total 304492K, used 191055K, reserved 335872K
  data space     204104K, used 160283K, reserved 233472K
  class space    100388K, used 30771K, reserved 102400K
```

JDK 1.8
Metaspace dynamic
re-size

from **20 MB**...**328 MB**

API Time

API Time

Nouvelle API 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 :** toutes les classes centrales de l'API Date and Time sont immuables, ce qui nous assure de ne pas avoir à nous soucier de problèmes de concurrence. De plus, qui dit objets immuables, dit des objets simples à créer, à utiliser et à tester.

- **Chaînage :** les méthodes chaînables rendent le code plus lisible et elles sont aussi plus simples à apprendre. Quant aux méthodes de type factory (par exemple: now(), from(), etc.) elles sont utilisées en lieu et place de constructeurs.

- **Clarté :** Chaque méthode définie clairement ce qu'elle fait. De plus, hormis dans quelques cas particuliers, passer un paramètre nul à une méthode provoquera la levée d'un NullPointerException. Les méthodes de validation prenant des objets en paramètre et retournant 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, bien que les classes de l' API soient basées sur le système de calendrier ISO-8601, nous pouvons aussi utiliser les calendriers non-ISO - tel que le calendrier Impérial Japonais - qui sont inclus dans l'API, ou même créer notre propre calendrier.

Instant

- Un point sur une timeline -1Md à + 1Md d'années :

Instant 0 = 1er Janvier 1970

Instant en cours : Instant instant = Instant.now();

- **Instant.EPOCH :** représente l'epoch.
- **Instant.MIN :** représente la plus petite valeur (pour les dates avant Jésus-Christ).
- **Instant.MAX :** représente la plus grande valeur possible (31 décembre un trillion).
- **Instant.parse() :** retourne un objet de type Instant à partir d'une chaîne de caractère représentant une date au format ISO-8601. Si la chaîne de caractères ne représente pas une valeur valide, une exception de type java.time.format.DateTimeParseException sera levée (le standard ISO-8601 spécifie que la lettre “T” désigne l'heure qu'elle précède et “Z” une data UTC).
- **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() : retourne un objet de type Instant représentant la date UTC actuelle.
- **Instant.now().getNano() :** retourne le nombre de nanosecondes de la date actuelle retournée par **Instant.now()**.

Duration

- Durée entre 2 Instant :

```
Instant start = Instant.now() ;
….
Instant end = Instant.now() ;
Duration elapsed = Duration.between(start,end);
long millis = elapsed.toMillis() ;
```

- **Duration.ZERO :** représente une durée nulle.
- **Duration.parse() :** retourne un objet de type Duration à partir d'une chaîne de caractères représentant une durée au format ISO-8601. Si la chaîne de caractères ne représente pas une valeur valide, une exception de type java.time.format.DateTimeParseException sera levée (selon le standard ISO-8601 une durée commence P - pour Period- et est suivie d'une valeur comme 4DT11H9M8S - pour Days, Hours, Minutes et Seconds - où la date et l'heure sont séparées par la lettre “T” - pour Time). Le standard permet aussi de définir un nombre d'années et de mois, ce que ne permet pas la méthode parse().
- **Duration.ofMillis() :** retourne un objet de type Duration à partir d'une chaîne de caractères représentant une durée en millisecondes.

Temps humain

- LocalDate

```
LocalDate d1 = LocalDate.now() ;
LocalDate d2 = LocalDate.of(2014, Month.APRIL ,25);
```

- LocalTime
- LocalDateTime
- ZoneId
- ZoneOffset
- ZonedDateTime
- OffsetDateTime

Period

- Durée entre 2 objets LocalDate
- Identique au type Duration (même méthodes)

```
LocalDate now = LocalDate.now() ;
LocalDate dateOfBirth = LocalDate.of(1970, Month.APRIL ,25);
Period p = dateOfBirth.until(now) ;
System.out.println(p.getYears()) ;

Period p2 = dateOfBirth.until(now, ChronoUnit.DAYS) ;
System.out.println(p.getDays()) ;
```

Interopérabilité entre l'ancienne et la nouvelle API

- Instant & Date :

```
Instant instant1 = Instant.now() ;
Date date = Date.from(instant1) ;
Instant instant2 = date.toInstant() ;
```

- Instant & Timestamp :

```
Timestamp time = TimeStamp.from(instant1) ;
Instant instant = time.toInstant() ;
```

- LocalDate & Date :

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

- LocalTime & Time :

```
Time time = Time.from(localTime) ;
LocalTime localTime = time.toLocalTime();
```

Expressions Lambda

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.

Définition

- 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 donc 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**
ou **(paramètres) -> {code}**
le retour doit être explicité si plusieurs instructions

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"));
```

Utilisation (2)

- 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).

Méthodes default

Mot-clé « default »

- On peut définir un corps dans une méthode d'une interface en la marquant **default**.

```
public interface InterfaceA {
        default void m1() {
                System.out.println("Hello A")

           }
                void m2() ;
     }
```

- La classe implémentant l'interface n'a pas l'obligation de redéfinir (@Override) une méthode default.

- Une classe implémentant 2 interfaces ayant la même méthode définie en default doit impérativement (sinon erreur de compilation) faire un @Override pour éviter l'héritage en étoile.

Interfaces fonctionnelles

Définition

- 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.
- Package : **java.util.function**

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));
```

java.util.function

- 4 catégories (43 interfaces) :

```java
@FunctionalInterface
public interface Supplier<T> {

    T get();
}
```

```java
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);
}
```

```java
@FunctionalInterface
public interface BiConsumer<T, U> {

    void accept(T t, U u);
}
```

```java
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);
}
```

```java
@FunctionalInterface
public interface Predicate<T, U> {

    boolean test(T t, U u);
}
```

```java
@FunctionalInterface
public interface Function<T, R> {

    R apply (T t);
}
```

```java
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply (T t, U u);
}
```

Références

- On peut faire référence à une méthode de classe ou d'instance :
System.out::println
stringLenComparator::compare

On peut appeler un constructeur :
NomClasse::new

```
Random r = new Random();
Supplier<Double> random = Math::random;
Supplier<Double> random2 = r::nextDouble;
Function<Random,Double> random3 = Random::nextDouble;
Function<String, Thread> factory = Thread::new;
Supplier<Double> random = () -> Math.random();
Supplier<Double> random2 = () -> r->nextDouble();
Function<Random,Double> random3 = (Random random) -> random.nextDouble();
Function<String, Thread> factory = (String name) -> new Thread(name);
```

Streams

Stream

- Un **iterator** limité qui permet de parcourir des collections. Il ne modifie pas la source des données.

- On peut enchaîner des opérations. L'enchaînement est appelé pipeline.

- Il existe des méthodes intermédiaires et des méthodes terminales. Ces dernières doivent être appelées pour clôturer le pipeline.

- Stream<T>

- ParellelStream

Utilisation

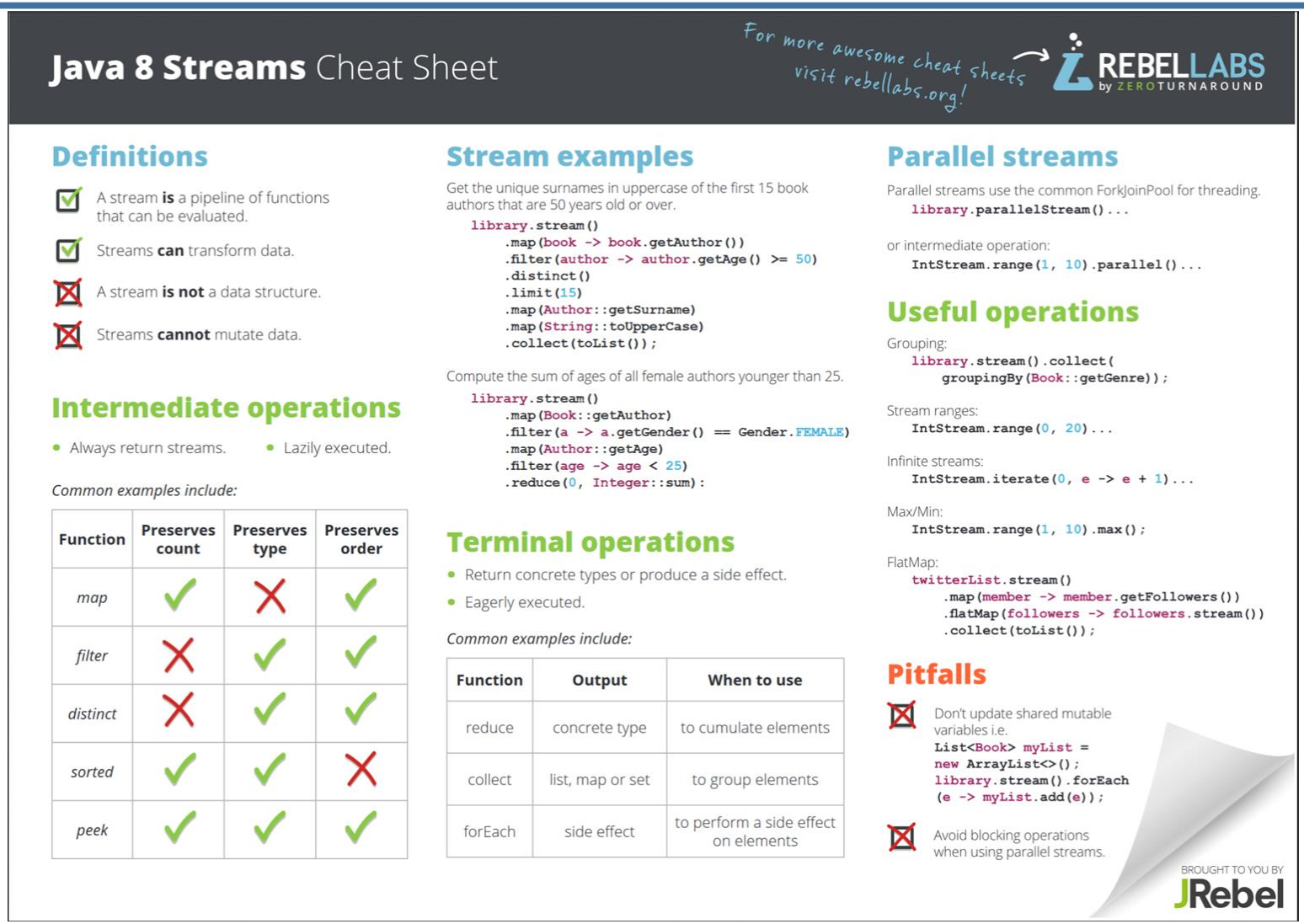
```
long nbAdmins = users.stream()
                                        .filter(a -> a.getRole().equals("admin"))
                                        .count();


chaines.stream().filter(x -> x.startsWith("cha"))
                .map(x -> x.substring(0, 1).toUpperCase() + x.substring(1))
                .sorted()
                                .forEach(System.out::println);


double chiffresAffaires = commandes.stream()
                          .collect(Collectors.summingDouble(Commande::getTotal));


String s = "hello DAWAN";
Stream stream = s.chars()
                                 .map(String::toUpperCase)
                                 .forEach(System.out::println);
```

Streams (2)



Java 8 Streams Cheat Sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

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.

BROUGHT TO YOU BY JRebel

Source : RebelLabs

StringJoiner

- Création d'une chaîne de caractères avec un séparateur

```
StringJoiner sj = new StringJoiner(", ", "{", "}");
sj.add("one").add("two").add("three");
String s = sj.toString();
System.out.println(s); //one, two, three
```

```
StringJoiner sj = new StringJoiner(", ");
sj.add("one").add("two").add("three");
String s = sj.toString();
System.out.println(s); // {one, two, three}
```

Moteur de scripting Nashorn

Nashorn

- Moteur de Javascript intégré dans le jdk (outil : jjs).
- On peut interpréter du code javascript depuis Java et passer également des paramètres Java vers le code javascript.
- Récupération du moteur :

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
```

- Évaluation d'une expression ou d'un appel de fonctions :

```
engine.eval("print('Hello Dawan!');");
```

- Chargement d'un fichier :

```
BufferedReader reader = new BufferedReader(new FileReader("script.js"));
engine.eval(reader);
reader.close();
Invocable invocable = (Invocable) engine;
Object result = invocable.invokeFunction("fun1", "Mohamed DERKAOUI");
```

Objets Java / JS

- On peut instancier des objets Java dans le JS :

```
var IntArray = Java.type('int[]');
var myObj = Java.type('fr.dawan.formation.Participant');
var ArrayList = Java.type('java.util.ArrayList') ;
```

- Le package java.util est importé
(pas besoin d'une déclaration explicite)

- On peut récupérer des objets JS dans le code Java :
objet : ScriptObjectMirror

```
public static void funcJava3(ScriptObjectMirror mirror) {
        System.out.println(mirror.getClassName() + ": " + Arrays.toString(mirror.getOwnKeys(true)));
        System.out.println(mirror.get("nom"));
        System.out.println(mirror.get("tel"));
    }

    public static void funcJava4(ScriptObjectMirror person) {
        System.out.println("Full Name is: " + person.callMember("getFullName"));
    }
```

Plus d'informations sur http://www.dawan.fr
Contactez notre service commercial au **09.72.37.73.73**(prix d'un appel local)

DAWAN Paris, 11 Rue Antoine Bourdelle, 75015 PARIS
DAWAN Nantes, 28, rue de Strasbourg, 44000 NANTES
DAWAN Lyon, Bâtiment de la banque Rhône Alpes, 235 cours Lafayette, 69006 LYON
DAWAN Lille , 16 place du général de Gaulle, 6ème étage, 59800 Lille
formation@dawan.fr