Java - Nouveautés des versions 8 à 16

christopher loisel

Rappel sur le package java.util.concurrent

Le package Java Concurrency fourni la concurrence, le multithreading et le parallélisme sur la plate-forme Java. Cela signifie qu'il peut vous aider à exécuter simultanément de nombreuses tâches ou applications différentes.

La concurrence Java tourne en grande partie autour de l'utilisation simultanée de threads, un processus léger qui possède son propre ensemble de fichiers et de piles pouvant accéder aux données partagées avec d'autres threads dans le même processus.

De manière asynchrone ou en parallèle, l'exécution de tâches chronophages peut améliorer le débit et l'interactivité du programme.

Java 5 a ajouté un nouveau package à la plate-forme Java → le package concurrent

Ce package contient un ensemble de classes et d'interfaces qui vous aideront à développer des applications concurrentes en Java. Si vous envisagez de créer vos propres classes d'utilitaires, ce package vous sera très utile car il fournit un ensemble de classes d'utilitaires pré-créées.

Executor : Un objet qui exécute des tâches Runnable.

ExecutorService : Un exécuteur qui fournit des méthodes pour gérer la terminaison et des méthodes qui peuvent produire un futur pour suivre la progression d'une ou plusieurs tâches asynchrones.

ScheduledExecutorService : un ExecutorService qui peut planifier l'exécution de commandes après un délai donné ou de façon périodique.

Future<V> : Un Future représente le résultat d'un calcul asynchrone.

CountDownLatch : Une aide à la synchronisation qui permet à un ou plusieurs threads d'attendre la fin d'un ensemble d'opérations effectuées dans d'autres threads.

CyclicBarrier : Une aide à la synchronisation qui permet à un ensemble de threads d'attendre les uns les autres pour atteindre un point de barrière commun.

Semaphore : Un sémaphore de comptage.

ThreadFactory : Un objet qui crée de nouveaux threads à la demande.

BlockingQueue<E> : Une file d'attente qui prend également en charge les opérations qui attendent que la file d'attente ne soit pas vide lors de la récupération d'un élément et qui attendent qu'un espace se libère dans la file d'attente lors du stockage d'un élément.

DelayQueue<E extends **Delayed**> : Une file d'attente bloquante non limitée d'éléments retardés, dans laquelle un élément ne peut être pris que lorsque son retard a expiré.

Phaser : Une barrière de synchronisation réutilisable, dont la fonctionnalité est similaire à celle de CyclicBarrier et CountDownLatch mais qui permet une utilisation plus flexible.

L’ Executor

Le package java.util.concurrent définit les interfaces utilisées pour effectuer les différentes tâches dans un environnement multithread. L'interface Executor est utilisée pour exécuter les tâches mentionnées dans le texte d'entrée.

```
public interface Executor
{
void execute(Runnable command);
}
```

Ainsi, une implémentation Executor prend l'instance Runnable donnée et l'exécute. On ne sait pas comment cela se produira réellement, mais la documentation suggère que vous devriez commencer à l'exécuter.

Une implémentation simple pourrait donc être :

```
public class MyExecutor implements Executor {
public void execute(Runnable r){
(new Thread(r)).start();
}
}
```

ExecutorService

ExecutorService est utile pour les tâches qui doivent être exécutées en arrière-plan et dont l'exécution est contrôlée par un thread de gestion. Ceci peut être réalisé en utilisant une classe Runnable.

```
public class Task implements Runnable {
    @Override
    public void run() {

        // task details
    }
}
```

Nous pouvons maintenant créer un objet/instance de cette classe et assigner la tâche. Nous devons spécifier la taille du pool de threads lors de la création d'une instance

```
// 20 is the thread pool size
ExecutorService exec = Executors.newFixedThreadPool(20);
```

Pour la création d'une instance ExecutorService monothread , nous pouvons utiliser newSingleThreadExecuter(ThreadFactory threadfactory) pour créer l'instance. Une fois l'exécuteur créé, nous pouvons soumettre la tâche.

```
public void execute() {
    executor.submit(new Task());
}
```

De plus, nous pouvons créer une instance Runnable pour la soumission de tâches.

```
executor.submit(() -> {
    new Task();
});
```

Deux méthodes de terminaison prêtes à l'emploi sont répertoriées comme suit :

1. shutdown() : il attend que toutes les tâches soumises soient terminées.
2. shutdownNow() : Il met immédiatement fin à toutes les tâches en cours d'exécution/en attente.

Il existe une autre méthode qui est awaitTermination() qui bloque avec force jusqu'à ce que toutes les tâches aient terminé leur exécution après qu'un événement d'arrêt déclenché ou un délai d'exécution se soit produit, ou que le thread d'exécution lui-même soit interrompu.

```
try {
    exec.awaitTermination( 50l, TimeUnit.NANOSECONDS );
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

ScheduledExecutorService

Le ScheduledExecutorService est comme ExecutorService sauf qu'il peut exécuter des tâches périodiquement. Vous avez besoin d'un Runnable ou Callable qui définit la tâche que vous souhaitez effectuer

```
public void execute() {
    ScheduledExecutorService execServ
      = Executors.newSingleThreadScheduledExecutor();

    Future<String> future = executorService.schedule(() -> {
        // ..
        return "Hello world";
    }, 1, TimeUnit.SECONDS);

    ScheduledFuture<?> scheduledFuture = execServ.schedule(() -> {
        // ..
    }, 1, TimeUnit.SECONDS);

    executorService.shutdown();
}
```

ScheduledExecutorService peut également programmer une tâche pour qu'elle se produise après un laps de temps prédéterminé.

```
executorService.scheduleAtFixedRate(() -> {
    // ..
}, 1, 20, TimeUnit.SECONDS);

executorService.scheduleWithFixedDelay(() -> {
    // ..
}, 1, 20, TimeUnit.SECONDS);
```

Ici,

- La méthode scheduleAtFixedRate() crée et exécute une action périodique qui est d'abord invoquée après le délai initial, puis périodiquement avec la période donnée jusqu'à ce que l'instance de temps indiquée par millisUntil s'épuise.
- La méthode scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit) crée et exécute une action périodique qui est invoquée en premier après le délai initial fourni. Il invoque ensuite à plusieurs reprises cette action avec le délai donné entre chaque invocation, en commençant toujours par le moment de la dernière invocation précédente.

Future

Représentant une opération asynchrone, les méthodes qu'elle contient vérifient si l'opération est terminée ou non et renvoient son résultat si c'est le cas.

L'API cancel(boolean isInterruptRunning) arrête l'opération, libère le thread et termine le thread si vrai. Sinon, les tâches sont terminées.

Cet extrait de code vous montre comment créer une instance de Future.

```
public void invoke() {
    ExecutorService executorService = Executors.newFixedThreadPool(20);

    Future<String> future = executorService.submit(() -> {
        // ...
        Thread.sleep(10000l);
        return "Hello";
    });
}
```

Calculez le résultat du futur et vérifiez si son exécution est terminée. Si le futur est fait, obtenez-en les données.

```
if (future.isDone() && !future.isCancelled()) {
    try {
        str = future.get();
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

Les opérations peuvent parfois prendre plus de temps que prévu. Mais si l'opération ne se termine pas dans un certain laps de temps, alors TimeoutException sera levée.

```
try {
    future.get(20, TimeUnit.SECONDS);
} catch (InterruptedException | ExecutionException | TimeoutException e) {
    e.printStackTrace();
}
```

CountDownLatch

La classe utilitaire synchrone bloque un ensemble de threads jusqu'à ce que certaines opérations soient terminées.

Un CountDownLatch (verrou de compte à rebours) est initialisé avec un compteur qui décrémentera au fur et à mesure que les threads termineront leur exécution. Une fois que le compteur atteint zéro, les threads dépendants sont libérés.

Barrière Cyclique

CyclicBarrier est très similaire à CountDownLatch. Il peut être utilisé encore et encore, ce qui signifie que plusieurs threads attendront le compte à rebours jusqu'à ce qu'ils exécutent la tâche finale. Il manque cette fonctionnalité dans CountDownLatch

Nous devons créer un objet qui déclenchera l'événement.

```
public class Task implements Runnable {

    private CyclicBarrier barrier;

    public Task(CyclicBarrier barrier) {
        this.barrier = barrier;
    }

    @Override
    public void run() {
        try {
            LOG.info(Thread.currentThread().getName() +
              " is waiting");
            barrier.await();
            LOG.info(Thread.currentThread().getName() +
              " is released");
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }

}
```

Voici quelques fils de discussion pour discuter de la condition barrière :

```
public void start() {

    CyclicBarrier cyclicBarrier = new CyclicBarrier(3, () -> {
        // ..
        LOG.info("All previous tasks completed");
    });

    Thread t11 = new Thread(new Task(cyclicBarrier), "T11");
    Thread t12 = new Thread(new Task(cyclicBarrier), "T12");
    Thread t13 = new Thread(new Task(cyclicBarrier), "T13");

    if (!cyclicBarrier.isBroken()) {
        t11.start();
        t12.start();
        t13.start();
    }
}
```

Dans le code ci-dessus, isBroken() vérifie si l'un des threads a été interrompu pendant l'exécution.

Semaphore

Il est utilisé pour bloquer l'accès au niveau des threads à une partie de la ressource logique ou physique. Sémaphore contient un ensemble de permis. Lorsque le thread entre du code dans une section critique, le sémaphore lui indique si cette section est ouverte ou non. Si le permis n'est pas disponible, le fil ne peut pas entrer dans la partie critique du système.

La section critique est un terme qui fait référence à une ressource partagée et mutable. Deux threads ne peuvent pas accéder simultanément à la section critique. Un compteur garde une trace du nombre d'accès, donc lorsqu'un thread quitte la section critique et la libère pour que l'autre thread y entre, le compteur augmente.

Le code ci-dessous est ce qui est utilisé pour l'implémentation de Sémaphore.

```
public void execute() throws InterruptedException {

    LOG.info("Available : " + semaphore.availablePermits());
    LOG.info("No. of threads waiting to acquire: " +
      semaphore.getQueueLength());

    if (semaphore.tryAcquire()) {
        try {
            //
        }
        finally {
            semaphore.release();
        }
    }
}
```

Les sémaphores peuvent être utilisés pour

implémenter une structure de données de type Mutex .

ThreadFactory

Lorsqu'un nouveau thread est nécessaire, ThreadFactory peut le créer pour vous.

```java
public class GFGThreadFactory implements ThreadFactory {
    private int threadId;
    private String name;

    public GFGThreadFactory(String name) {
        threadId = 1;
        this.name = name;
    }

    @Override
    public Thread newThread(Runnable r) {
        Thread t = new Thread(r, name + "-Thread_" + threadId);
        LOG.info("created new thread with id : " + threadId +
            " and name : " + t.getName());
        threadId++;
        return t;
    }
}
```

BlockingQueue

BlockingQueue prend en charge le contrôle de flux en plus de la mise en file d'attente en introduisant le blocage si l'un des côtés de la file d'attente est plein ou vide.Un thread essayant de placer un élément dans une file d'attente complète est bloqué jusqu'à ce qu'un autre thread fasse de la place dans la file d'attente, soit en supprimant un ou plusieurs éléments de la file d'attente. Par exemple, s'il y a 10 éléments dans une file d'attente et que le dernier élément est supprimé par un autre thread, cela permettrait une mise en file d'attente immédiate. Il bloque également les threads essayant de supprimer d'une file d'attente lorsqu'elle est vide, afin qu'ils ne perdent pas leur temps. Si nous essayons de mettre en file d'attente une valeur nulle, une exception sera levée.

Phaseur

Cela signifie que le Phaser peut être réutilisé et mis à l'échelle de manière dynamique et qu'il est plus flexible qu'un CyclicBarrier ou un CountDownLatch. Un thread attendra sur une phase jusqu'à ce que le nombre de threads en attente soit inférieur ou égal à la taille de cette phase. Plusieurs phases d'exécution peuvent être coordonnées en réutilisant une instance d'un Phaser pour chaque phase du programme.

Join/Fork

Java 7 a introduit le framework fork/join. Il fournit des outils permettant d'accélérer le traitement parallèle en essayant d'utiliser tous les cœurs de processeur disponibles. Pour ce faire, il utilise l'approche "divide and conquer".

En pratique, cela signifie que le cadre commence par forks, en décomposant récursivement la tâche en sous-tâches indépendantes plus petites jusqu'à ce qu'elles soient suffisamment simples pour être exécutées de manière asynchrone.

Ensuite, la partie "join" commence. Les résultats de toutes les sous-tâches sont récursivement joints en un seul résultat. Dans le cas d'une tâche qui renvoie un résultat nul, le programme attend simplement que chaque sous-tâche s'exécute.

Pour assurer une exécution parallèle efficace, le cadre fork/join utilise un pool de threads appelé ForkJoinPool. Ce pool gère des threads de travail de type ForkJoinWorkerThread.

ForkJoinPool

Le ForkJoinPool est le cœur du framework. Il s'agit d'une implémentation de l'ExecutorService qui gère les threads de travail et nous fournit des outils pour obtenir des informations sur l'état et les performances du pool de threads.

Les threads de travail ne peuvent exécuter qu'une seule tâche à la fois, mais le ForkJoinPool ne crée pas un thread distinct pour chaque sous-tâche. Au lieu de cela, chaque thread du pool possède sa propre file d'attente à double extrémité (ou deque, prononcé "deck") qui stocke les tâches.

Cette architecture est essentielle pour équilibrer la charge de travail du thread à l'aide de l'algorithme de vol de travail.

Algorithme de détournement de travail

En termes simples, les threads libres essaient de "voler" du travail aux deques des threads occupés.

Par défaut, un thread travailleur reçoit des tâches de la tête de sa propre deque. Lorsque celle-ci est vide, le thread prend une tâche dans la queue de la deque d'un autre thread occupé ou dans la file d'entrée globale, car c'est là que les plus gros morceaux de travail sont susceptibles de se trouver.

Cette approche minimise la possibilité que les threads se disputent les tâches. Elle réduit également le nombre de fois où le thread devra aller chercher du travail, puisqu'il travaille d'abord sur les plus gros morceaux de travail disponibles.

Instanciation de ForkJoinPool

En Java 8, le moyen le plus pratique d'accéder à l'instance du ForkJoinPool est d'utiliser sa méthode statique commonPool(). Cette méthode fournit une référence au pool commun, qui est un pool de threads par défaut pour chaque ForkJoinTask.

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
```

Selon la documentation d'Oracle, l'utilisation du pool commun prédéfini réduit la consommation de ressources car cela décourage la création d'un pool de threads distinct par tâche.

```
public static ForkJoinPool forkJoinPool = new ForkJoinPool(2);
```

Nous pouvons obtenir le même comportement en Java 7 en créant un ForkJoinPool et en l'affectant à un champ statique public d'une classe utilitaire :

Maintenant, nous pouvons y accéder facilement :

```
ForkJoinPool forkJoinPool = PoolUtil.forkJoinPool;
```

Grâce aux constructeurs de ForkJoinPool, nous pouvons créer un pool de threads personnalisé avec un niveau de parallélisme, une fabrique de threads et un gestionnaire d'exceptions spécifiques. Ici, le pool à un niveau de parallélisme de 2, ce qui signifie que le pool utilisera deux cœurs de processeur.

ForkJoinTask<*V*>

Une ForkJoinTask est le type de base des tâches exécutées dans un ForkJoinPool. Au cœur de toutes ces implémentations se trouvent les méthodes abstraites RecursiveAction et RecursiveTask avec les méthodes compute(). La dernière de ces deux sous-classes (RecursiveTask) est pour les tâches qui renvoient un résultat, alors que le premier (RecursiveAction) se termine dans une boucle infinie s'il n'y a pas de tâche récursive à terminer.

RecursiveAction

Dans cet exemple, nous utilisons une chaîne appelée workload pour représenter l'unité de travail à traiter. Pour les besoins de la démonstration, la tâche n'a aucun sens : Elle met simplement en majuscule son entrée et l'enregistre.

Pour démontrer le comportement de fork du framework, l'exemple divise la tâche si workload.length() est supérieur à un seuil spécifié en utilisant la méthode createSubtask().

La chaîne est divisée récursivement en sous-chaînes, créant des instances CustomRecursiveTask basées sur ces sous-chaînes.

En conséquence, la méthode renvoie une List<CustomRecursiveAction>.

La liste est soumise au ForkJoinPool à l'aide de la méthode invokeAll() :

```java
public class CustomRecursiveAction extends RecursiveAction {

    private String workload = "";
    private static final int THRESHOLD = 4;

    private static Logger logger =
      Logger.getAnonymousLogger();

    public CustomRecursiveAction(String workload) {
        this.workload = workload;
    }

    @Override
    protected void compute() {
        if (workload.length() > THRESHOLD) {
            ForkJoinTask.invokeAll(createSubtasks());
        } else {
            processing(workload);
        }
    }

    private List<CustomRecursiveAction> createSubtasks() {
        List<CustomRecursiveAction> subtasks = new ArrayList<>();

        String partOne = workload.substring(0, workload.length() / 2);
        String partTwo = workload.substring(workload.length() / 2, workload.length());

        subtasks.add(new CustomRecursiveAction(partOne));
        subtasks.add(new CustomRecursiveAction(partTwo));

        return subtasks;
    }

    private void processing(String work) {
        String result = work.toUpperCase();
        logger.info("This result - (" + result + ") - was processed by "
          + Thread.currentThread().getName());
    }
}
```

Nous pouvons utiliser ce modèle pour développer nos propres classes RecursiveAction. Pour ce faire, nous créons un objet qui représente la quantité totale de travail, choisissons un seuil approprié, définissons une méthode pour diviser le travail et définissons une méthode pour effectuer le travail.

RecursiveTask<V>

Pour les tâches qui renvoient une valeur, la logique est similaire.

La différence est que le résultat de chaque sous-tâche est réuni en un seul résultat :

Dans cet exemple, nous utilisons un tableau stocké dans le champ arr de la classe CustomRecursiveTask pour représenter le travail. La méthode createSubtasks() divise récursivement la tâche en petits morceaux de travail jusqu'à ce que chaque morceau soit plus petit que le seuil. Ensuite, la méthode invokeAll() soumet les sous-tâches au pool commun et renvoie une liste de Future.

Pour déclencher l'exécution, la méthode join() est appelée pour chaque sous-tâche.

Nous avons réalisé cela ici en utilisant l'API Stream de Java 8. Nous utilisons la méthode sum() pour représenter la combinaison des résultats des sous-tâches dans le résultat final.

```java
public class CustomRecursiveTask extends RecursiveTask<Integer> {
    private int[] arr;

    private static final int THRESHOLD = 20;

    public CustomRecursiveTask(int[] arr) {
        this.arr = arr;
    }

    @Override
    protected Integer compute() {
        if (arr.length > THRESHOLD) {
            return ForkJoinTask.invokeAll(createSubtasks())
              .stream()
              .mapToInt(ForkJoinTask::join)
              .sum();
        } else {
            return processing(arr);
        }
    }

    private Collection<CustomRecursiveTask> createSubtasks() {
        List<CustomRecursiveTask> dividedTasks = new ArrayList<>();
        dividedTasks.add(new CustomRecursiveTask(
          Arrays.copyOfRange(arr, 0, arr.length / 2)));
        dividedTasks.add(new CustomRecursiveTask(
          Arrays.copyOfRange(arr, arr.length / 2, arr.length)));
        return dividedTasks;
    }

    private Integer processing(int[] arr) {
        return Arrays.stream(arr)
          .filter(a -> a > 10 && a < 27)
          .map(a -> a * 10)
          .sum();
    }
}
```

Soumettre des tâches au ForkJoinPool

Nous pouvons utiliser plusieurs approches pour soumettre des tâches au pool de threads.

Commençons par la méthode submit() ou execute() (leurs cas d'utilisation sont les mêmes) :

```
forkJoinPool.execute(customRecursiveTask);
int result = customRecursiveTask.join();
```

La méthode invoke() fork la tâche et attend le résultat, et n'a pas besoin d'être jointe manuellement :

```
int result = forkJoinPool.invoke(customRecursiveTask);
```

La méthode invokeAll() est le moyen le plus pratique de soumettre une séquence de ForkJoinTasks au ForkJoinPool. Elle prend les tâches comme paramètres (deux tâches, var args ou une collection), fork et renvoie ensuite une collection d'objets Future dans l'ordre dans lequel ils ont été produits.

Alternativement, nous pouvons utiliser des méthodes fork() et join() séparées. La méthode fork() soumet une tâche à un pool, mais ne déclenche pas son exécution. Nous devons utiliser la méthode join() à cette fin.

Dans le cas de RecursiveAction, la méthode join() ne renvoie que null ; pour RecursiveTask<V>, elle renvoie le résultat de l'exécution de la tâche :

```
customRecursiveTaskFirst.fork();
result = customRecursiveTaskLast.join();
```

Ici, nous avons utilisé la méthode invokeAll() pour soumettre une séquence de sous-tâches au pool. Nous pouvons faire le même travail avec fork() et join(), bien que cela ait des conséquences sur l'ordre des résultats.

Pour éviter toute confusion, c'est généralement une bonne idée d'utiliser la méthode invokeAll() pour soumettre plus d'une tâche au ForkJoinPool.

L'utilisation du cadre fork/join peut accélérer le traitement de tâches importantes, mais pour obtenir ce résultat, nous devons suivre certaines directives :

-Utilisez le moins de pools de threads possible. Dans la plupart des cas, la meilleure décision est d'utiliser un pool de threads par application ou système.

-Utilisez le pool de threads commun par défaut si aucun réglage spécifique n'est nécessaire.

-Utilisez un seuil raisonnable pour diviser ForkJoinTask en sous-tâches.

-Évitez tout blocage dans les ForkJoinTasks.

Exercice

Faire un programme utilisant le pattern Fork and join.

Ce programme fait la somme de tous les éléments d'un tableau, en utilisant le parallélisme pour traiter 50 éléments en parallèle.

Expression lambda

Les plus importants des apports de la JDK8 résidents dans les expressions lambda et les streams. Utilisés conjointement, ils vont offrir au langage des possibilités traditionnellement réservées à ce que l'on nomme des langages de programmation fonctionnelle. On pourra alors traiter une structure de données (telle une collection) en exprimant ce que l'on souhaite obtenir, sans avoir besoin d'expliciter comment y parvenir.

C'est précisément cet aspect qui permettra une meilleure optimisation du code et surtout sa parallélisation, c'est-à-dire le partage entre plusieurs processeurs du traitement d'une structure.

Nous commencerons par étudier en soit cette nouvelle notion d'expression lambda, ainsi que celle de référence qui s'y rattache. Nous verrons comment les utiliser pour "paramétrer" l'appel de méthode et, ainsi, remplacer avantageusement l'emploi de classe anonyme, puis nous étudierons les streams qui nous permettront d'exploiter avantageusement ses nouvelles spécificités de programmation fonctionnelle.

Introduction aux expressions lambda

Beaucoup de langages permettent de fournir une méthode (ou une fonction) en argument d'une autre méthode ou encore de manipuler des variables contenant des références de méthodes. On parle souvent de "paramétrisation des méthodes", de “méthode de rappel" ou encore de “fonctions d'ordre supérieur". Jusqu'à Java 7, de telles fonctionnalités pouvaient être mises en œuvre, de façon assez fastidieuse, en recourant à ce que nous avons appelé des "objets fonctions”, à savoir des objets implémentant une interface ne comportant qu'une seule méthode. Souvent, on était alors amené à instancier un tel objet pour ne l'utiliser finalement qu'une seule fois ; on pouvait alors recourir à une classe anonyme. Nous en avons rencontré des exemples avec les objets "comparateurs" fournis à certains algorithmes de la classe Collections.

Java 8 a amélioré la situation en introduisant à la fois la notion d'expression lambda et celle de référence à une méthode. Nous commencerons par introduire l'expression lambda sur quelques exemples, avant d'en voir les propriétés générales.

Premiers exemples d'expression lambda

Voyons sur un exemple comment l'emploi d'une expression lambda permet de remplacer

avantageusement le recours à une classe anonyme. Ce programme utilise une interface nommée Calculateur contenant une seule méthode nommée calcul. Nous implémentons sous

forme d'une classe anonyme dont nous plaçons la référence dans une variable carre. Puis nous utilisons à deux reprises cette variable pour provoquer un appel de la méthode calcul.

```
interface Calculateur
{
    public int calcul (int n);
}

public class IntroLambda1
{
    public static void main (String args [])
    {
        int n1=5, n2 = 3;
        Calculateur carre = new Calculateur () { public int calcul(int n) {return n*n;}};
        int res = carre.calcul(n1);
        System.out.println("Carre de " + n1 +" " + res);
        System.out.println("Carre de " + n2 + " = " + carre.calculi(n2));
    }
}
```

Utilisation d'une classe anonyme pour paramétrer l'appel d'une fonction.

Avec Java 8, nous pouvons remplacer l’affectation :

par :

```
Calculateur carre = new Calculateur () { public int calcul(int n) {return n*n;}};
```

La notation :

```
Calculateur carre = x -> x * x;
```

x -> x * x

Est-ce que l'on nomme une expression lambda (ou, souvent, plus brièvement une "Lambda").

Elle représente en quelque sorte une méthode sans nom qui reçoit ici un argument nommé x et qui fournit en résultat la valeur de son carré.D'ores et déjà, on constate que nous n'avons précisé ni le type de l’objet fonction concerné, ni même le type de ses arguments. En fait, à la vue d'une telle affectation, le compilateur sait qu'il attend un objet du type de carré, c'est-à-dire Calculateur. Ce type ne définit qu'une seule méthode nommée calcul, il sait que la notation x -> x * x correspond à cette dernière. Il en déduit que x est de type int ct donc que la valeur de l’expression x * x est également de ce type.Enfin, aucune instruction return ne figure ici. Le compilateur prévoit simplement que c’est la valeur de l'expression ainsi calculée qui constitue la valeur de retour, de type inf, comme prévu dans l'interface Calculateur (si le type inf n'était pas compatible par affectation avec le type mentionné dans l'interface, on obtiendrait une erreur de compilation).

Voici, en définitive, l'adaptation du précédent programme :

```
interface Calculateur
{
    public int calcul (int n);
}

public class IntreLambda2
{
    public static void main (String args [])
    {
        int nl =5, n2 = 3;
        Calculateur carre = x-> x * x;
        int res = carre.calcul(ni);
        Syetem.out.println ("Carre de " + nl + " = " + res);
        System.out.printin ("Carre de " + m2 + " =" + carre.calcul(in2));
    }
}
```

Adaptation du programme précédent avec une expression lambda.

Ici, le corps de notre méthode *calcul* était très simple. Mais une expression lambda peut

comporter un corps plus élaboré, constitué de plusieurs instructions fournies alors plus

classiquement sous forme d’un bloc. Dans ce cas, si une valeur de retour est nécessaire, elle

sera spécifiée par une où plusieurs instructions return. Voici un exemple utilisant la même

interface Calculateur qui définit une expression lambda comportant un bloc d'instructions

(qui, en toute rigueur pourrait se ramener à une seule expression en utilisant deux opérateurs

conditionnels imbriqués) :

```
interface Calculateur { public int calcul(int n); }
public class CalculComplique
{ public static void main (String args [])
    { int n1 = 5, n2 = 4, n3 = -5;
        Calculateur complique = x -> { if (x > 0 && 2*(x/2)==x) return x;
            else if (x>0) return x+1;
            else return -x ;
        };
        int res = complique.calcul(n1);
        System.out.println ("Complique de " + n1 + "=" + res);
        System.out.println ("Complique de " + n2 + "=" + complique.calcul(n2));
        System.out.println ("Complique de " + n3 + "=" + complique.calcul(n3));
    }
}
```

Autre situation utilisant une expression lambda

Dans nos précédents exemples nous affection une expression lambda à une variable d'un type interface.

Mais, nous pouvons également utiliser une expression lambda pour donner une valeur a un argument transmis à une méthode, comme dans l'exemple suivant.

Il utilise la même interface calculateur que précédemment et il comporte une méthode statique *traite* qui reçoit un argument de type calculateur.

Lors de chacun des appels, cet argument est fourni sous forme d'une expression lambda.

```
interface Calculateur { public int calcul(int n); }
public class LambdaRappel
{ public static void main (String args [])
    {
        traite (5, x->x*x);
        traite (12, x ->2*x*x + 3*x +5);
    }
    public static void traite(int n, calculateur cal)
    {
        int res = cal.calcul(n);
        System.out.println("calcul (" + n + ") = " + res);

    }
}
```

Ce dernier exemple même mieux en évidence que les précédents le fait qu'une expression lambda permet en quelques sortes de définir un bloc de code qui sera utilisé ultérieurement par une méthode, situation qu'on traduit du souvent par le terme "fonction de rappels" noter que nous aurions pu également recourir à ce mécanisme en utilisant comme précédemment des variable de type calculateur.

Les interfaces fonctionnelles

Nous venons de voir comment utiliser une expression lambda pour implémenter une interface ne comportant qu'une seule méthode.

Ce mécanisme ne pourrait plus fonctionner si l'interface comportait plusieurs méthodes, puisque le compilateur ne pourrait plus savoir laquelle est implémentée par l'expression lambda.

Cependant avec Java 8,1 une interface peut disposer de méthode statique et de méthode par défaut, lesquels, de par leur nature, ne sont plus abstraits.

Bien tenir compte de cette particularité, il a été convenu que le mécanisme évoqué fonctionne encore dans ce cas, à condition naturellement que l'interface ne prévoit qu'une seule méthode abstraite.C'est cette dernière qui se trouvera implémentée par l'expression lambda.

Les interfaces répondant à cette condition seront nommés “interface fonctionnel” et la méthode abstraite correspondante sera dite “la méthode fonctionnelle”.

Une interface fonctionnelle doit contenir exactement une méthode abstraite, laquelle sera la méthode fonctionnelle.

Parmi les interfaces existant dans Java 7, certaines sont fonctionnels. Parmi celles déjà rencontré on peut citer : iterables, closeable, est comparable. Java en introduit beaucoup d'autres dans le but de simplifier l'utilisation des expressions lambda. Ainsi, dans l'exemple du paragraphe précédent, nous avions défini cette interface fonctionnelle:

```
interface Calculateur { public int calcul(int n); }
```

La méthode fonctionnelle calcul reçoit un argument de type int est fourni un résultat de type int une telle interface existe deux façons standard dans java 8, elle se nomme IntUnaryOperator; elle représente une fonction recevant un int et dont la méthode fonctionnelle se nomme applyAsInt. Voici comment nous pourrions adapter notre exemple dans ce sens:

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

public class LambdaRappel2
{ public static void main (String args[])
    {
        traite (5, x->x*x);
        traite (12, x ->2*x*x + 3*x +5);
    }
    public static void traite(int n, IntUnaryOperator cal)
    {
        int res = cal.calcul(n);
        System.out.println("calcul (" + n + ") = " + res);
    }
}
```

D'une manière générale, dans le paquetage java.util.function, il existe beaucoup d'interface standard représentant des fonctions, des prédicats... Non seulement elles sont définies pour les trois types numériques de base une, long et double mais, de surcroît, on trouve des versions générique représentant les mêmes fonctionnalités appliquées à des types classe quelconque. Par exemple, IntUnaryOperator qu'on vient d'utiliser dispose d'une version générique UnaryOperator<Int>, avec toutefois deux petites différences: d'une part, on aurait utilisé un type enveloppe integer au lieu d'un type de base. D'autre part, la méthode fonctionne elle se nommerait appeler apply et non à applyAsInt.

Voici tout d'abord les principales interfaces standard générique est la méthode fonctionnelle correspondante:

Interface fonctionnelle	Type arguments	Type valeur de retour	Méthode fonctionnelle
Supplier<T>	aucun	T	get
Consumer<T>	T	void	accept
BiConsumer<T, U>	T, U	void	accept
Function<T,R>	T	R	apply
BiFunction<T, U, R>	T,U	R	apply
UnaryOperator<T>	T	T	apply
BinaryOperator<T>	T,T	T	apply
Predicate<T>	T	boolean	test
BiPredicate<T, U>	T, U	boolean	test

La syntaxe des expressions lambda

Une expression lambda constitue donc une notation abrégé d'une méthode fonctionnel d'une interface fonctionnelle.

Comme toute méthode, une expression lambda de ou non renvoyer une valeur. Dans le cas où elle se limite à une seule expression, la valeur or envoyée n'est rien d'autre que celle de l'expression. Dans le cas d'un bloc, elle sera fourni le cas échéant par une ou plusieurs instructions return classique.

Par ailleurs, alors que nos exemples ne comportait qu'un argument, une expression lambda pour comporter plusieurs comme dans : `(x, y) -> x * x + x * y + y * y`

Noter qu'alors la syntaxe leur impose d'être entre parenthèses.

Il est possible de préciser le type d'un ou plusieurs arguments comme dans:

```
(Point P, float x) -> {System.out.println("valeur =" + x); p.affiche();}
```

La liste d'arguments peut-être vide, auquel cas elle doit obligatoirement être placée entre parenthèses:

```
() -> System.out.println("bonjour")
```

En définitive, la syntaxe générale d'une expression lambda est la suivante :

```
liste_d_arguments   ->   corps
```

Syntaxe d'une expression lambda

Contexte d'utilisation d'une expression lambda

D'une manière générale une expression lambda peut intervenir dans différents contextes permettant aux compilateur de lui attribuer un type, à savoir:

1. Déclaration de variable;
2. Affectation;
3. Argument de méthode;
4. Instruction return;
5. Corps d'une autre expression lambda (composition d'expression lambda);
6. Initialiseur de tableau
7. Expression conditionnel

Tableau d’expression lambda

Voici un exemple d'utilisation d'expression lambda dans lequel nous définissons un tableau de calculateur fourni chacun sous forme d'une expression lambda :

```
interface Calculateur { public int calcul(int n); }
public class LambdaRappel2
{ public static void main (String args[])
    {
        Calculateur [] tabCalc = {x-> x*x, x->2*x, x->(int)Math.sqrt(x)};
        for (Calculateur calc : tabCalc) traite(15, calc);
    }
    public static void traite(int n, calculateur cal)
    {
        int res = cal.calcul(n);
        System.out.println("calcul (" + n + ") = " + res);
    }
}
```

Références de méthodes

Nous avons vu comment une expression lambda permet d'exprimer une méthode fonctionnelle d'une interface fonctionnelle en introduisant le code correspondant à l'emplacement où on en a besoin. Les références de méthode vont offrir une autre sorte de raccourci dès lors qu'une méthode existante peut jouer le rôle de la méthode fonctionnelle attendue. Comme nous allons le voir, il existe plusieurs sortes de référence qui peuvent s'appliquer à des méthodes statistiques, à des méthodes de classe, à des méthodes associant un objet particulier ou à des constructeurs.

Références de méthodes statique

Considérons à nouveau l'exemple du paragraphe précédent, en supposant que nous disposons d'une méthode statique carré définit ainsi:

```
public static int carre (int n)
{
    return n*n;
}
```

Dans l'appel de la méthode traite, a la place de l'expression lambda x->x*x, nous pouvons fournir la référence à cette méthode carré sous la forme suivante

RefStat::carre

Notre appel de la méthode traite s'écrira alors : traite(5, RefStat::carre);

On notera que, bien que cette méthode possède un nom, le compilateur est capable de la traiter comme une méthode fonctionnelle en se basant uniquement sur le type des arguments et de la valeur de retour.

```java
interface Calculateur { public int calcul(int n); }
public class RefStat
{
    public static int carre (int n)
    {
        return n*n;
    }

    public static int trinome (int n)
    {
        return 2*n*n + 3*n + 5;
    }

    public static void main (String args[])
    {
        traite(5, RefStat::carre);
        traite(12, RefStat::carre);
    }

    public static void traite(int n, calculateur cal)
    {
        int res = cal.calcul(n);
        System.out.println("calcul (" + n + ") = " + res);
    }
}
```

Présentation des streams

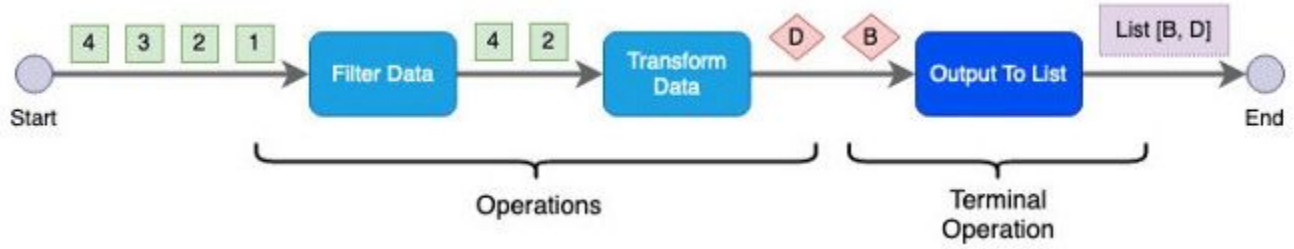
Nous l'avons dit en introduction de ce chapitre, les streams apportent donc à Java des possibilités traditionnellement réservées au langage fonctionnel, dans lesquels le programmeur se contente d'exprimer "ce qu'il veut obtenir" sans avoir à préciser les moyens d'y parvenir.
Cette particularité va ouvrir la voie à l'automatisation du calcul parallèle.

Un stream décrit une succession d'opérations destinées à être appliquées à une structure de données telle une collection et ,plus généralement, à toute structure sur laquelle il est possible d'iterer.

Par exemple supposons que nous disposions d'une liste : `ArrayList<Integer> liste;`

Considérons cette " instruction" qui, en définitive, affiche les éléments de liste, dont la valeur est positive:

```
liste.stream().filter(ee-> ee>0).forEach(ee -> System.out.print(ee + " "));
```



La méthode stream(prévu dorénavant dans l'interface collection) crée un objet de type stream<Integer>, associé aux éléments de l'objet l'ayant appelé, c'est-à-dire ici de liste.

Un tel stream ce n'est pas une nouvelle structure de données; aucune donnée ni stocker. Il ne s'agit que d'établir une sorte de canal de traitement de l'information.

La méthode filter s'applique à un stream; on lui fournit un prédicat c'est-à-dire une méthode recevant un argument du type des éléments du stream(ici Integer), et fournissant à résultat booléen; il permet de sélectionner les seuls éléments satisfaisant à la condition précisée.

Le résultat est un stream (de même type, ici stream<integer>) c'est-à-dire un " canal" permettant d'accéder aux seuls éléments sélectionnés, ici les positifs de la liste.

Aucun traitement n'est encore mis en place.

La méthode foreach s'applique elle aussi un stream.

On lui fournit un argument de type *consumer* qui précise une action à réaliser sur chacun des éléments concernés.

C'est seulement l'exécution de cette méthode qui entraîne la mise en œuvre du parcours de la liste et l'application des actions prévues dans le stream, ici sélection et affichage.

C'est précisément cette particularité permet ce que l'on homme une " exécution paresseuse" dans laquelle ne sont réalisées que les opérations strictement nécessaires.

Une méthode tel que *filter* et dit intermédiaire tandis qu'une méthode tel que for each et terminale.

On peut appliquer un stream un nombre quelconque de méthode intermédiaire alors pompe plus appliquer une seule méthode terminale puisque c'est elle qui déclenche le traitement proprement dit.

Un stream ne peut être utilisée qu'une seule fois ainsi un petit exemples précédents nous aurions pu déclarer:

```
Stream<Integer> str = liste.stream();
Stream<Integer> str1 = liste.stream(ee-> ee>0);
```
avant de réaliser l'appel.

```
str1.foreach(ee->System.out.print(ee + " "));
```

Nous aurions obtenu les mêmes résultats mais l'exécution de for each aurait provoqué la fermeture du stream STR et pas seulement str1!

Une tentative d'exécution de St Victor ou de Esther un. For each aurait provoqué une erreur d'exécution.

Voici un autre exemple fondé sur une liste d'objets de type supposé disposer des méthodes nécessaire getX et getY fournissant des résultats de type int.

```
listePoint.stream().map(pp->pp.getX() + pp.getY()).filter(xx-> xx > 0).forEach(xx ->System.out.print(xx + " "));
```

Ici la méthode map. Fais correspondre à chaque. Un entier et égal à la somme de ses coordonnées. Elle reçoit donc en entrée un stream<Point> et elle fournit en sortie un stream dentier, nous verrons qu'il s'agira précisément d'un intstream.

La méthode foreach s'applique à ce stream d'éléments entier; ici nous nous contentons d'afficher les valeurs ainsi obtenues. On notera qu'il ne sera pas possible d'accéder au point de la collection de départ

Différentes façon de créer un stream

Pour créer un flux, la source de données doit être accessible. Ces sources sont ce qui permet de fournir les différents éléments à la demande d'un flux pour qu'il les traite.

Il existe différentes manières de créer un flux à partir de différentes sources :

- Collection
- Tableau
- Ensemble de données
- Fichier
- ...

Vous pouvez également utiliser une fonction pour générer un nombre infini d'éléments en tant qu'entrée de données.

Exemple

X.stream()	Stream<String> chaines = Arrays.asList("a1", "a2", "a3").stream();
Stream<T> Arrays.stream(T[])	Stream<String> chaines = Arrays.stream({"a1", "a2", "a3"});
Stream<T> Stream.of(T)	pour un élément
Stream<Integer> stream = Stream.of(1,2,3,4); //	pour plusieur éléments
IntStream.range(int, int)	Les valeurs entières incluse entre les bornes inférieure et supérieure exclue fournies
Stream<T> Stream.iterate(T, UnaryOperator<T>)	Stream<Integer> stream = Stream.iterate(0, (i)-> i + 1);
Stream.empty()	Un empty stream
Stream<String> Files.lines	les lignes du fichier
Stream<String> BufferedReader.lines()	les lignes du fichier
...	...

La création d'un Stream à partir de ses fabriques

L'interface Stream propose diverses méthodes de fabrication pour créer un flux.

```
Stream<Integer> stream = Stream.of(new Integer[] { 1, 2, 3});
```

Attention, ce sont des objets qui sont attendus. Il n'est pas possible de fournir un tableau de type entier primitif en paramètre.

L'obtention d'un Stream à partir d'une collection

Le traitement des données sur une collection peut être fait en itérant explicitement à travers chaque élément. Stream a fourni une API pour l'itération, qui fait tout le travail pour vous. L'interface Collection a deux façons d'itérer appelées : "stream()" et "parallelStream()".

L'interface Collection propose deux méthodes pour obtenir un Stream dont la source sera la collection :

Méthode	Rôle
default Stream<E> stream()	Renvoyer un Stream dont les traitements seront exécutés de manière séquentielle sur les éléments de la collection
default Stream<E> parallelStream()	Renvoyer un stream dont les traitements sont exécutés en parallèle sur les éléments de la collection

Une façon d'obtenir un flux à partir de n'importe quelle collection consiste à utiliser la méthode stream().

```
List<String> elements = new ArrayList<String>();
elements.add("element1");
elements.add("element2");
elements.add("element3");
Stream<String> stream = elements.stream();
```

Les opérations intermédiaires

Une opération intermédiaire fait toujours référence à un Stream

Dans un pipeline d'opérations de flux, le nombre d'opérations intermédiaires peut être défini sur zéro, 1 ou plus.

Les opérations intermédiaires peuvent être regroupées en deux parties :

- Les opérations sans état ne préservent pas l'état des éléments par rapport au suivant, ce qui signifie que chaque élément est traité indépendamment des autres.
- Une opération avec état gardera une trace de sa progression par rapport aux éléments qu'elle a parcourus. Par exemple, un panier d'achat peut garder une trace du montant que chaque client a déjà dépensé pour son achat total.Certaines opérations avec état doivent traiter tous les éléments avant de pouvoir produire leur résultat. Le traitement d'un flux avec un traitement parallèle peut nécessiter

L'API Stream comporte certaines opérations intermédiaires définies :

Opération		Rôle
filter	Stateless	Stream<T> filter(Predicate<? super T> predicate) Filtrer tous les éléments pour n'inclure dans le Stream de sortie que les éléments qui satisfont le Prédicat
map	Stateless	<R> Stream<R> map(Function<? super T,? extends R> mapper) Renvoyer un Stream qui contient le résultat de la transformation de chaque élément de type T en un élément de type R
mapToxxx(Int, Long or Double)	Stateless	xxxStream mapToxxx(ToxxxFunction<? super T> mapper) Renvoyer un Stream qui contient le résultat de la transformation de chaque élément de type T en un type primitif xxx
flatMap	Stateless	<R> Stream<R> flatMap(Function<T,Stream<? extends R>> mapper) Renvoyer un Stream avec l'ensemble des éléments contenus dans les Stream<R> retournés par l'application de la Function sur les éléments de type T. Ainsi chaque élément de type T peut renvoyer zéro, un ou plusieurs éléments de type R.

flatMapToxxx (Int, Long or Double)		xxxStream flatMapToxxx(Function<? super T,? extends xxxStream> mapper) Renvoyer un Stream avec l'ensemble des éléments contenus dans les xxxStream retournés par l'application de la Function sur les éléments de type T. Ainsi chaque élément de type T peut renvoyer zéro, un ou plusieurs éléments de type primitif xxx.
distinct	Stateful	Stream<T> distinct() Renvoyer un Stream<T> dont les doublons ont été retirés. La détection des doublons se fait en invoquant la méthode equals()
sorted	Stateful	Stream<T> sorted() Stream<T> sorted(Comparator<? super T>) Renvoyer un Stream dont les éléments sont triés dans un certain ordre. La surcharge sans paramètre tri dans l'ordre naturel : le type T doit donc implémenter l'interface Comparable car c'est sa méthode compareTo() qui est utilisée pour la comparaison des éléments deux à deux La surcharge avec un Comparator l'utilise pour déterminer l'ordre de tri.
peek	Stateless	Stream<T> peek(Consumer <? super T>) Renvoyer les éléments du Stream et leur appliquer le Consumer fourni en paramètre
limit	Stateful Short-Circuiting	Stream<T> limit(long) Renvoyer un Stream qui contient au plus le nombre d'éléments fournis en paramètre

skip	Stateful	Stream<T> skip(long) Renvoyer un Stream dont les n premiers éléments ont été ignorés, n correspondant à la valeur fournie en paramètre
sequential		Stream<T> sequential() Renvoyer un Stream équivalent dont le mode d'exécution des opérations est séquentiel
parallel		Stream<T> parallel() Renvoyer un Stream équivalent dont le mode d'exécution des opérations est en parallèle
unordered		Stream<T> parallel() Renvoyer un Stream équivalent dont l'ordre des éléments n'a pas d'importance
onClose		Stream<T> onClose(Runnable) Renvoyer un Stream équivalent dont le handler fourni en paramètre sera exécuté à l'invocation de la méthode close(). L'ordre d'exécution de plusieurs handlers est celui de leur déclaration. Tous les handlers sont exécutés même si un handler précédent à lever une exception.

Les opérations terminales

Les flux sont traités dès que leur seule opération terminale est appelée.

Une seule opération de terminal ne peut pas être "invoquée" sur le même Stream : une fois cela fait, le Stream ne sera plus utilisable. Afin d'animer le flux, les utilisateurs doivent obtenir un nouveau flux de votre source pour chaque traitement. On ne peut invoquer le pipeline que sur un seul flux à la fois, sinon il lèvera une exception.

```
String[] fruits = { "orange", "citron", "pamplemousse", "banane","fraise",
"groseille", "raisin", "pomme", "poire", "abricot", "cerise", "peche", "clementine" };

Stream<String> stream = Stream.of(fruits);
stream.filter(s -> s.startsWith("p"))
  .forEach(System.out::println);
try {
stream.filter(s -> s.startsWith("c"))
    .forEach(System.out::println);
} catch (IllegalStateException e) {
e.printStackTrace(System.out);
}
```

Il n'est pas nécessaire de créer un nouveau flux pour chaque ensemble de données, car le flux pointe simplement vers sa source et ne la réplique pas. Vous pouvez utiliser un fournisseur pour générer des flux pour différents ensembles.

```
 Supplier<Stream<String>> streamSupplier = () -> Stream.of(fruits);
  streamSupplier.get()
                .filter(s -> s.startsWith("p"))
                .forEach(System.out::println);
  streamSupplier.get()
                .filter(s -> s.startsWith("c"))
                .forEach(System.out::println);
```

Contrairement aux opérations intermédiaires qui renvoient toujours un Stream, les opérations terminales renvoient une valeur qui correspond au résultat de l'exécution du pipeline d'opérations sur les données. Ce résultat peut être :

- Une valeur d'un type primitif
- Un élément ou une instance de type Optional
- Une collection ou un tableau d'éléments
- void

Il est possible qu'il n'y ait pas de résultat à l'issu des traitements d'un Stream. Java 8 propose la classe Optional qui encapsule une valeur ou l'absence de valeur.

Certaines opérations d'un Stream renvoie donc un objet de type java.util.Optional qui permet de préciser s'il y a un résultat ou non. C'est notamment le cas dans les opérations de réduction, des opérations de recherche (findXXX) ou de recherche de correspondance (XXXMatch) sur un Stream vide.

L'API Stream propose plusieurs opérations terminales dont les principales sont :

Méthode	Rôle
forEach	void forEach(Consumer<? super T> action) Exécuter le Consumer sur chacun des éléments du Stream
forEachOrdered	void forEachOrdered(Consumer<? super T> action) Exécuter le Consumer sur chacun des éléments du Stream en respectant l'ordre de éléments si le Stream en défini un
toArray	Object[] toArray() Renvoyer un tableau contenant les éléments du Stream <A> A[] toArray(IntFunction<A[]> generator) Renvoyer un tableau contenant les éléments du Stream : le tableau est créé par la fonction fournie
Reduce	Optional<T> reduce(BinaryOperator<T> accumulator) Réaliser une opération de réduction qui accumule les différents éléments du Stream grâce à la fonction fournie T reduce(T identity, BinaryOperator<T> accumulator) Réaliser une opération de réduction qui accumule à partir de la valeur fournie les différents éléments du Stream grâce à la fonction <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) Réaliser une opération de réduction avec les fonctions fournies en paramètres

collect	<R,A> R collect(Collector<? super T,A,R> collector) Réaliser une opération de réduction avec le Collector fourni en paramètre <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) Réaliser une opération de réduction avec les fonctions fournies en paramètres
min	Optional<T> min(Comparator<? super T> comparator) Renvoyer le plus petit élément du Stream selon le Comparator fourni
max	Optional<T> max(Comparator<? super T> comparator) Renvoyer le plus grand élément du Stream selon le Comparator fourni
count	long count() Renvoyer le nombre d'éléments contenu dans le Stream
anyMatch	boolean anyMatch(Predicate<? super T> predicate) Retourner un booléen qui indique si au moins un élément valide le Predicate

allMatch	boolean allMatch(Predicate<? super T> predicate) Retourner un booléen qui indique si tous les éléments valident le Predicate
noneMatch	boolean noneMatch(Predicate<? super T> predicate) Retourner un booléen qui indique si aucun élément ne valide le Predicate
findFirst	Optional<T> findFirst() Retourner un Optional qui encapsule le premier élément validant le Predicate s'il existe
findAny	Optional<T> findAny() Retourner un Optional qui encapsule élément validant le Predicate s'il existe
iterator	Iterator<T> iterator() Renvoyer un Iterator qui permet de réaliser une itération sur tous les éléments en dehors du Stream
spliterator	Spliterator<T> spliterator() Renvoyer un Spliterator pour les éléments du Stream

Certaines de ces méthodes sont de type short-circuiting, par exemple findFirst().

Il est possible d'exporter les éléments d'un Steam dans une collection ou un tableau en utilisant certaines de ses méthodes :

Méthode	Rôle
collect(Collectors.toList())	Obtenir une collection de type List qui contient les éléments du Stream `Stream<Integer> intStream = Stream.of(1,2,3); List<Integer> intList = intStream.collect(Collectors.toList()); System.out.println(intList);` Copy

toArray(TypeDonnees[]::new)	Obtenir un tableau qui contient les éléments du Stream
	`Stream<Integer> intStream = Stream.of(1, 2, 3);`
	`Integer[] intArray = intStream.toArray(Integer[]::new); System.out.println(Arrays.deepToString(intArray));`
	Copy

Définissez votre flux de données, puis utilisez une opération de réduction pour traiter les éléments du flux. Les opérations de réduction typiques sur des nombres entiers sont, par exemple, la sommation de valeurs ou la recherche de la moyenne. Quelques autres exemples examinent quel est le plus petit nombre ou le plus grand nombre.

Une opération de réduction commence toujours par une identité, puis par le premier élément. Il répète le processus pour chacun des éléments suivants et les combine à travers un traitement fixe. Le résultat est simplement une répétition de ces traitements effectués sur chaque élément.

Vous pouvez générer le résultat souhaité en répétant cette combinaison d'actions :

- Un résultat unique (la somme, la moyenne, la plus petite/grande valeur, ...)
- Une collection (List, Set, ...) qui contient les éléments
- Une Map qui contient des paires clé/valeur extraites des données du Stream

L'API Stream propose différentes opérations pour vous aider à organiser vos données :

- Spécialisée : count(), max(), min(), ...
- Générique : reduce(), collect()

Travaux pratiques

1)Utilisez les Streams pour compter le nombre de mots dans une liste de string.

2)Faire un Intstream créé avec la méthode range, compris entre 0 et 150, sélectionner uniquement les int compris entre 5 et 35, et les affiche.

3)Dans une *List* de string, sélectionner uniquement les string commençant par une voyelle, et pratiquer un foreach pour inverser celles-ci.

4)Trier une *List* d'int dans l'ordre croissant, retirer ensuite les doublons, et faire un foreach sur le résultat pour afficher celui-ci.

L'API DateTime

Il était autrefois difficile de travailler avec des dates en Java. L'ancienne bibliothèque de dates fournie par le JDK ne comprenait que trois classes : java.util.Date, java.util.Calendar et java.util.Timezone.

Ces classes ne convenaient que pour les tâches les plus élémentaires. Pour toute tâche un tant soit peu complexe, les développeurs devaient soit utiliser des bibliothèques tierces, soit écrire des tonnes de code personnalisé.

Java 8 a introduit une toute nouvelle API de date et d'heure (java.util.time.*) qui est vaguement basée sur la bibliothèque Java populaire appelée JodaTime. Cette nouvelle API simplifie considérablement le traitement de la date et de l'heure et corrige de nombreuses lacunes de l'ancienne bibliothèque de date.

Le premier avantage de la nouvelle API est sa clarté : elle est très claire, concise et facile à comprendre.

Elle ne présente pas les nombreuses incohérences de l'ancienne bibliothèque, comme la numérotation des champs (dans le calendrier, les mois sont basés sur des zéros, mais les jours de la semaine sont basés sur des unités).

Un autre avantage est la flexibilité - travailler avec plusieurs représentations du temps.

L'ancienne bibliothèque de dates ne comprenait qu'une seule classe de représentation du temps - java.util.Date, qui, malgré son nom, est en fait un timestamp.

Elle ne stocke que le nombre de millisecondes écoulées depuis l'époque Unix.

La nouvelle API propose plusieurs représentations temporelles différentes, chacune convenant à des cas d'utilisation différents :

Instant - représente un point dans le temps (timestamp)

LocalDate - représente une date (année, mois, jour)

LocalDateTime - identique à LocalDate, mais inclut le temps avec une précision de l'ordre de la nanoseconde.

OffsetDateTime : identique à LocalDateTime, mais avec un décalage de fuseau horaire.

LocalTime : heure avec une précision de l'ordre de la nanoseconde, sans information sur la date.

ZonedDateTime : même principe que OffsetDateTime, mais avec un décalage de fuseau horaire.

OffsetLocalTime : identique à LocalTime, mais avec un décalage de fuseau horaire.

MonthDay : mois et jour, sans année ni heure.

YearMonth : mois et année, sans jour ni heure.

Duration - durée représentée en secondes, minutes et heures. La précision est de l'ordre de la nanoseconde.

Period - période de temps représentée en jours, mois et années.

Un autre avantage est que toutes les représentations temporelles de l'API Date et Heure de Java 8 sont immuables et donc à l'abri des erreurs de threads.

Toutes les méthodes de mutation renvoient une nouvelle copie au lieu de modifier l'état de l'objet original.

Les anciennes classes telles que java.util.Date n'étaient pas thread-safe et pouvaient introduire des bogues de programmation concurrente très subtils.

Chaînage de méthodes

Toutes les méthodes de mutation peuvent être enchaînées, ce qui permet de mettre en œuvre des transformations complexes en une seule ligne de code.

```java
import java.time.*;
import java.time.temporal.*;

public class Main
{
    Run | Debug
    public static void main(String[] args)
    {
        ZonedDateTime nextFriday = LocalDateTime.now().plusHours(hours: 1).
        with(TemporalAdjusters.next(DayOfWeek.FRIDAY)).
        atZone(ZoneId.of(zoneId: "Europe/Paris"));
        System.out.println(nextFriday);
    }
}
```

Les exemples ci-dessous montrent comment effectuer des tâches courantes avec l'ancienne et la nouvelle API.

Obtenir l'heure actuelle:

```
// Old
Date now = new Date();

// New
ZonedDateTime now = ZonedDateTime.now();
```

représenter une date spécifique:

```
// Old
Date birthDay = new GregorianCalendar(1990, Calendar.DECEMBER, 15).getTime();

// New
LocalDate birthDay = LocalDate.of(1990, Month.DECEMBER, 15);
```

Extraction de champs spécifiques

```
// Old
int month = new GregorianCalendar().get(Calendar.MONTH);

// New
Month month = LocalDateTime.now().getMonth();
```

Additionner et soustraire le temps

```
// Old
GregorianCalendar calendar = new GregorianCalendar();
calendar.add(Calendar.HOUR_OF_DAY, -5);
Date fiveHoursBefore = calendar.getTime();

// New
LocalDateTime fiveHoursBefore = LocalDateTime.now().minusHours(5);
```

Modification de champs spécifiques

```
// Old
GregorianCalendar calendar = new GregorianCalendar();
calendar.set(Calendar.MONTH, Calendar.JUNE);
Date inJune = calendar.getTime();

// New
LocalDateTime inJune = LocalDateTime.now().withMonth(Month.JUNE.getValue());
```

Tronquer : La troncature remet à zéro tous les champs de temps plus petits que le champ spécifié. Dans l'exemple ci-dessous, les minutes et tout ce qui est inférieur seront mis à zéro

```
// Old
Calendar now = Calendar.getInstance();
now.set(Calendar.MINUTE, 0);
now.set(Calendar.SECOND, 0);
now.set(Calendar.MILLISECOND, 0);
Date truncated = now.getTime();

// New
LocalTime truncated = LocalTime.now().truncatedTo(ChronoUnit.HOURS);
```

Conversion des fuseaux horaires

```
// Old
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTimeZone(TimeZone.getTimeZone("CET"));
Date centralEastern = calendar.getTime();

// New
ZonedDateTime centralEastern = LocalDateTime.now().atZone(ZoneId.of("CET"));
```

Obtenir l'intervalle de temps entre deux points dans le temps

```
// Old
GregorianCalendar calendar = new GregorianCalendar();
Date now = new Date();
calendar.add(Calendar.HOUR, 1);
Date hourLater = calendar.getTime();
long elapsed = hourLater.getTime() - now.getTime();

// New
LocalDateTime now = LocalDateTime.now();
LocalDateTime hourLater = LocalDateTime.now().plusHours(1);
Duration span = Duration.between(now, hourLater);
```

Formatage et analyse de l'heure : DateTimeFormatter est un remplacement de l'ancien SimpleDateFormat qui est sûr pour les threads et offre des fonctionnalités supplémentaires.

```
// Old
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
Date now = new Date();
String formattedDate = dateFormat.format(now);
Date parsedDate = dateFormat.parse(formattedDate);

// New
LocalDate now = LocalDate.now();
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
String formattedDate = now.format(formatter);
LocalDate parsedDate = LocalDate.parse(formattedDate, formatter);
```

Nombre de jours dans un mois

```
// Old
Calendar calendar = new GregorianCalendar(1990, Calendar.FEBRUARY, 20);
int daysInMonth = calendar.getActualMaximum(Calendar.DAY_OF_MONTH);

// New
int daysInMonth = YearMonth.of(1990, 2).lengthOfMonth();
```

Exercice Pratique

Faite un programme qui a pour but d’afficher votre age, a partir de votre date de naissance