Formation Java : Nouveautés des versions 8 à 17

El Hadji Gaye

Auteur	El Hadji Gaye
Pour	Formation
Date	21/02/2024
Objet	Formation Java : Nouveautés des versions 8 à 17

I) Vocabulaire

- **JVM** : Java Virtual Machine. Cette dernière traduit le code source java en un code intermédiaire appelé « byte code ».

- **JRE** : Java Runtime Environment. Il est composé de la JVM et de quelques API (Application Programming Interface).

- **JDK** : Java Development Kit. Il est composé du JRE plus certains outils de base pour le développement (javac : Java Programming Language compiler, javap : Java Class File Disassembler, javadoc, etc.).

- **IDE** : Integrated Development Environment. C'est l'outil de développement. Il contient la JVM, la JRE, la JDK permettant ainsi de développer des applications java, de les compiler et de les exécuter.

- **Heap Space** : peut être traduit en français comme l'espace de tas. L'espace Java Heap est utilisé par le runtime Java pour allouer de la mémoire aux objets et aux classes JRE. Chaque fois que nous créons un objet, il est toujours créé dans l'espace Heap.

- **Stack Space** : peut être traduit en français comme mémoire de pile. La mémoire de pile en Java est utilisée pour l'allocation de mémoire statique et l'exécution d'un thread. Il contient des valeurs primitives spécifiques à une méthode et des références à des objets qui se trouvent dans un tas, référencés à partir de la méthode.

  L'accès à cette mémoire est dans l'ordre Last-In-First-Out (LIFO). Chaque fois qu'une nouvelle méthode est appelée, un nouveau bloc au-dessus de la pile est créé qui contient des valeurs spécifiques à cette méthode, comme des variables primitives et des références à des objets.

  Une fois l'exécution de la méthode terminée, le cadre de pile correspondant est vidé, le flux revient à la méthode appelante et de l'espace devient disponible pour la méthode suivante. Garbage Collection s'exécute sur la mémoire du tas pour libérer la mémoire utilisée par les objets qui n'ont aucune référence. Tout objet créé dans l'espace de tas a un accès global et peut être référencé à partir de n'importe quel endroit de l'application.

- **Projet Jigsaw** : Le projet **Jigsaw** vise à concevoir et à implémenter un système de modules standard pour la plate-forme Java SE et à appliquer ce système à la plate-forme elle-même et au JDK. Ses principaux objectifs sont de rendre les implémentations de la plate-forme plus facilement extensibles aux petits appareils, d'améliorer la sécurité et la maintenabilité, d'améliorer les performances des applications et de fournir aux développeurs de meilleurs outils pour la programmation en général.

- **API** : Signifie Application Programming Interface. Ce qui veut dire que c'est un ensemble de bibliothèques et librairies dédié pour implémenter une fonctionnalité donnée.

- **ORM** : Object-Relational Mapping (**MOR** : Mapping Objet-Relationnel en français) est une technique de programmation informatique qui crée l'illusion d'une base de données orientée

objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé.

- **JPA** : Java Persistence API (abrégée en JPA), est une interface de programmation Java permettant aux développeurs d'organiser des données relationnelles dans des applications utilisant la plateforme Java.

- **JPQL** : Le langage JPQL (**Java Persistence Query Language**) est un langage de requête orienté objet, similaire à SQL, mais au lieu d'opérer sur les tables et colonnes, JPQL travaille avec des objets persistants et de leurs propriétés. Il est très proche du langage SQL dont il s'inspire fortement mais offre une approche objet. La grammaire de ce langage est définie par la spécification J.P.A.

- **HQL** : **Hibernate Query Language** est aussi un langage de requête orienté objet au même titre que JPQL. La principale différence avec le langage JQL est que le en HQL le « **Select** » sur l'objet n'est pas nécessaire. En fin de compte pour le JPQL on aura : **Select person from Personne person** alors que pour le HQL on aura : **from Personne**.

- **Bean** : le « **Bean** » (ou haricot en français) est une technologie de composants logiciels écrits en langage Java. Les **Beans** sont utilisés pour encapsuler plusieurs objets dans un seul objet. Le « **Bean** » regroupe alors tous les attributs des objets encapsulés. Ainsi, il représente une entité plus globale que les objets encapsulés de manière à répondre à un besoin métier.

- **EJB** : **Enterprise Java Beans** (EJB) est une architecture de composants logiciels côté serveur pour la plateforme de développement Java EE. Cette architecture propose un cadre pour créer des composants distribués (c'est-à-dire déployés sur des serveurs distants) écrit en langage de programmation Java hébergés au sein d'un serveur applicatif permettant de représenter des données (EJB dit entité), de proposer des services avec ou sans conservation d'état entre les appels (EJB dit session), ou encore d'accomplir des tâches de manière asynchrone (EJB dit message). Tous les EJB peuvent évoluer dans un contexte transactionnel ce qui peut permettre de gérer les transactions avec les sources de données (fichier Xml, fichier Csv, fichier Json, base de données etc.…).

- **POJO** : POJO est un acronyme qui signifie Plain Old Java Object que l'on peut traduire en français par bon vieil objet Java. Cet acronyme est principalement utilisé pour faire référence à la simplicité d'utilisation d'un objet Java en comparaison avec la lourdeur d'utilisation d'un composant EJB. Ainsi, un POJO n'implémente pas d'interface spécifique à un Framework comme c'est le cas par exemple pour un composant EJB.

- **POJI** : POJI est un acronyme qui signifie Plain Old Java Interfaces que l'on peut traduire en français par bon vieil Interface Java correspond à une interface standard Java. Ils sont habituellement utilisés dans le contexte JEE pour fournir des services.

- **JNDI** : JNDI signifie Java Naming and Directory Interface, cette API permet d'accéder à différents services de nommage ou de répertoire de façon uniforme, d'organiser et rechercher des informations ou des objets par nommage (java naming and directory interface), de faire des opérations sur des annuaires (java naming and directory interface) tels que : LDAP : un

annuaire léger, X500 : normes d'annuaires lourdes à mettre en œuvre, NIS : annuaire obsolète.

- **Pool de connexions** : Un pool de connexions est un mécanisme permettant de réutiliser les connexions créées. En effet, la création systématique de nouvelles instances de Connection peut parfois devenir très lourd en consommation de ressources. Pour éviter cela, un pool de connexions ne ferme pas les connexions lors de l'appel à la méthode close(). Au lieu de fermer directement la connexion, celle-ci est « retournée » au pool et peut être utilisée ultérieurement.La gestion du pool se fait en général de manière transparente pour l'utilisateur

- **JSP** : Java Server Pages.

- **JSF** : Java Server Faces.

- **EJB** : Entreprise Java Bean.

- **EAR** : Un EAR (pour Enterprise Application Archive) est un format de fichier utilisé par Java EE pour empaqueter (en) un ou plusieurs modules dans une seule archive, de façon à pouvoir déployer ces modules sur un serveur d'applications en une seule opération, et de façon cohérente.

- **War** : WAR (Web Application Archive) est un fichier JAR utilisé pour contenir un ensemble de Java Server Pages, servlets, classes Java, fichiers XML, et des pages web statiques (HTML, JavaScript…), le tout constituant une application web. Cette archive est utilisée pour déployer une application web sur un serveur d'applications.

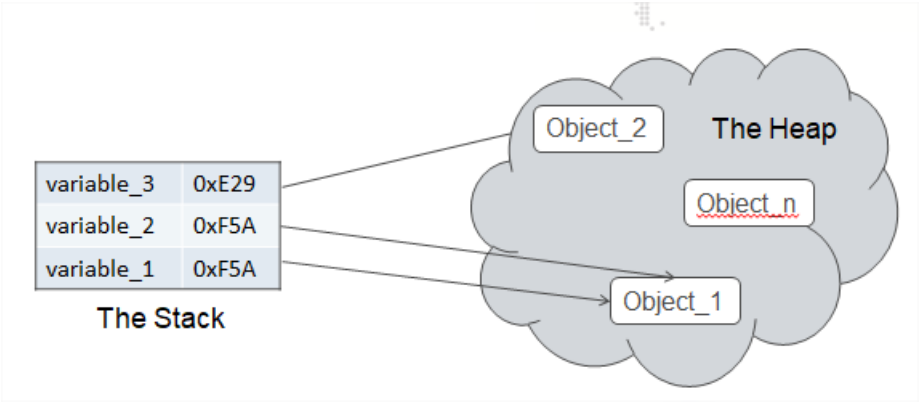
- **JSTL** : JSTL (Java Server page Standard Tag Library). C'est un ensemble de tags personnalisés et développés qui proposent des fonctionnalités souvent rencontrées dans les JSP.

- **JAR Hell** : JAR Hell est un terme similaire à Hell DLL utilisé pour décrire toutes les différentes manières dont le processus de chargement de classe peut finir par ne pas fonctionner.

II) Fonctionnement de la machine virtuelle (JVM)

Pour exécuter une application de manière optimale, JVM divise la mémoire en mémoire de pile (**Stack Memory**) et mémoire de tas (**Heap Space**). Chaque fois que nous déclarons de nouvelles variables et objets, appelons une nouvelle méthode, déclarons une chaîne ou effectuons des opérations similaires, JVM désigne la mémoire pour ces opérations à partir de la mémoire de pile ou de l'espace de tas.

Nous aborderons ces modèles de mémoire. Nous énumérerons quelques différences clés entre eux, comment ils sont stockés dans la RAM, les fonctionnalités qu'ils offrent et où les utiliser.



1. Mémoire de pile (Stack Memory)

La mémoire de pile ou **Stack Space** en Java est utilisée pour l'allocation de mémoire statique et l'exécution d'un thread. Il contient des valeurs primitives spécifiques à une méthode et des références à des objets qui se trouvent dans un tas, référencés à partir de la méthode.

Voici quelques autres fonctionnalités de la mémoire de pile :

- Il grandit et rétrécit au fur et à mesure que de nouvelles méthodes sont appelées et renvoyées respectivement
- Les variables à l'intérieur de la pile n'existent que tant que la méthode qui les a créées est en cours d'exécution
- Il est automatiquement alloué et dé-alloué lorsque la méthode termine l'exécution
- Si cette mémoire est pleine, Java lance **java.lang.StackOverFlowError**
- L'accès à cette mémoire est rapide par rapport à la mémoire de tas
- Cette mémoire est Thread Safe car chaque thread fonctionne dans sa propre pile

2. Mémoire de tas (Heap Space)

La mémoire de tas **Heap Space** est utilisée pour l'allocation dynamique de mémoire pour les objets Java et les classes JRE au moment de l'exécution. Les nouveaux objets sont toujours créés dans l'espace du tas et les références à ces objets sont stockées dans la mémoire de la pile. Ces objets ont un accès global et sont accessibles de n'importe où dans l'application.

Ce modèle de mémoire est divisé en parties plus petites appelées générations, ce sont:

- **Jeune génération** : c'est là que tous les nouveaux objets sont attribués et vieillis. Un **Garbage Collection** mineur se produit lorsque cela se remplit.

- **Ancienne génération ou génération permanente** : c'est là que sont stockés les objets qui ont survécu longtemps. Lorsque des objets sont stockés dans la jeune génération, un seuil pour l'âge de l'objet est défini et lorsque ce seuil est atteint, l'objet est déplacé vers l'ancienne génération.

- **Génération permanente** : il s'agit de métadonnées JVM pour les classes d'exécution (Class Loader) et les méthodes d'application.

Voici quelques autres caractéristiques de l'espace de tas:

- Il est accessible via des techniques complexes de gestion de la mémoire qui incluent la jeune génération, l'ancienne ou la génération permanente et la génération permanente

- Si l'espace du tas est plein, Java renvoie **java.lang.OutOfMemoryError**

- L'accès à cette mémoire est relativement plus lent que la mémoire de pile

- Cette mémoire, contrairement à Stack, n'est pas automatiquement désallouée. Il a besoin de Garbage Collector pour libérer les objets inutilisés afin de conserver l'efficacité de l'utilisation de la mémoire

- Contrairement à la pile, un tas n'est pas Thread Safe et doit être protégé en synchronisant correctement le code

III) L'architecture des Java Memory

1. *Permgen Space*

Tous ceux qui ont rencontrés le problème « **OutOfMemoryError : PermGen space** » pour la première fois ont certainement dû s'arracher les cheveux. Pourquoi donc ce processus Java se plaint-il de ne plus avoir de mémoire alors qu'il n'utilise pas la moitié de ce qui lui est alloué ?

La première partie de la réponse est assez simple… quand on l'a déjà vu ! La JVM de Sun définit la taille de la mémoire allouée en deux lots distincts.

1. La mémoire classique (« **Heap Space** ») configurée classiquement avec les options **-Xms128m** et **-Xmx256m**, respectivement pour l'initiale et la taille maxime.
2. La mémoire de la « **Permanent Generation** », contenant notamment les déclarations des objets Class (mais pas les instances !) et Method. La taille de cet espace est définit par les options **-XX:PermSize=96m** et **-XX:MaxPermSize=128m**, respectivement pour la taille initiale et la taille maximale

Il est tout à fait possible que le réglage par défaut (64Mo de taille maximale) ne suffise pas et qu'une « **OutOfMemoryError : PermGen space** » apparaisse après quelques minutes d'utilisation de l'application, à la suite de la compilation d'un volume trop important de JSP. Dans ce cas, augmenter l'espace « permanent générations » à 128Mo devrait résoudre le problème. Attention cependant, comme toutes les options commençant par XX, il s'agit de paramètres spécifiques à l'implémentation de la JVM, en l'occurrence celle de Sun. Ainsi la JVM BEA par exemple ne sépare pas la définition de ces deux espaces dans ses options de JVM.

Donc, grâce à ces options nous voilà débarrassés de cette erreur de mémoire. Pas forcément…

En effet, il est possible que cette erreur réapparaisse par exemple après plusieurs redéploiements d'une application web sous **Tomcat**. Dans ce cas, le réglage de la mémoire n'est pas en cause, mais il s'agit bel et bien d'une fuite mémoire. Plus précisément, les classes de l'application que l'on a déchargée ne sont pas recyclées par la JVM. En effet, tant qu'une référence subsiste vers une classe chargée par le **ClassLoader** de l'application web, ce **ClassLoader** et toutes les classes qu'il a définis ne pourront pas être recyclés.

Une multitude de causes peuvent aboutir à ce problème, parmi les plus classiques :

- Le chargement d'un driver JDBC dans le **ClassLoader** d'une application web : le driver (et donc le **ClassLoader** et toutes ses classes) reste référencé au niveau serveur par le DriverManager, on peut utiliser un ServletContextListener pour décharger le driver
- Une bibliothèque chargée au niveau serveur (type Hibernate) référence dans un cache des classes de l'application (comme par exemple des classes Proxy des classes du modèle)
- L'application web a fixé des données dans un objet ThreadLocal, liées à un thread du serveur (le thread appartient à un pool du conteneur et n'est jamais recyclé)

2. *MaxMetaspaceSize*

Dans la version 8 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.

IV) Le JDK Modulaire

Le JDK est divisé en un ensemble de modules qui peuvent être combinés au moment de la compilation, de la construction et de l'exécution dans une variété de configurations incluses, on peut aussi dénombrer :

- Configurations correspondant à la plate-forme Java SE complète, au JRE complet et au JDK complet

- Configurations à peu près équivalentes en contenu à chacun des profils compacts définis dans Java SE 8

- Configurations personnalisées qui ne contiennent qu'un ensemble spécifié de modules éventuellement augmentés par des modules externes de bibliothèque et d'application, et les modules requis de manière transitoire par tous ces modules.

La définition de la structure modulaire doit faire une distinction claire entre les modules standards, dont les spécifications sont régies par le **Java Community Process**, et les modules spécifiques au JDK. Il doit également distinguer les modules inclus dans la spécification de la plate-forme Java SE, et donc rendus obligatoires dans chaque implémentation de plate-forme, de tous les autres modules.

Le projet **Jigsaw** vise à concevoir et à implémenter un système de modules standard pour la plate-forme Java SE et à appliquer ce système à la plate-forme elle-même et au JDK. Ses principaux objectifs sont de rendre les implémentations de la plate-forme plus facilement extensibles aux petits appareils, d'améliorer la sécurité et la maintenabilité, d'améliorer les performances des applications et de fournir aux développeurs de meilleurs outils pour la programmation en général.

La structure modulaire du JDK met en œuvre les principes suivants :

1) Les modules standards, dont les spécifications sont régies par le JCP, ont des noms commençant par la chaîne "java.".

2) Tous les autres modules font simplement partie du JDK et ont des noms commençant par la chaîne "jdk.".

3) Si un module exporte un package qui contient un type qui contient un membre public ou protégé qui, à son tour, fait référence à un type d'un autre module, alors le premier module doit accorder une lisibilité implicite au second, via requiert transitive. (Cela garantit que le chaînage d'invocation de méthode fonctionne de manière évidente.)

4) Un module standard peut contenir des packages d'API standard et non standard. Si un module standard exporte un package API standard, l'exportation peut être qualifiée ; si un module standard exporte un package d'API non standard, l'exportation doit être qualifiée. Dans les deux cas, si un module standard exporte un package avec qualification, l'exportation doit se faire vers un sous-ensemble des modules du JDK. Si un module standard est un module Java SE, c'est-à-dire qu'il est

inclus dans la spécification de la plate-forme Java SE, il ne doit pas exporter de packages d'API non SE, du moins pas sans qualification.

5) Un module standard peut dépendre d'un ou plusieurs modules non standards. Il ne doit pas accorder de lisibilité implicite à un module non standard. S'il s'agit d'un module Java SE, il ne doit accorder une lisibilité implicite à aucun module non SE.

6) Un module non standard ne doit exporter aucun package d'API standard. Un module non standard peut accorder une lisibilité implicite à un module standard.

Une conséquence importante des principes 4 et 5 est que le code qui ne dépend que des modules Java SE dépendront uniquement des types Java SE standard et sera donc portable pour toutes les implémentations de la plate-forme Java SE.

V) Le Java Platform Module System spécifie

Le Java Platform Module System spécifie un format de distribution pour les collections de code Java et les ressources associées. Il spécifie également un référentiel pour stocker ces collections de code Java, ou modules, et identifient comment ils peuvent être découverts, chargés et vérifiés pour leur intégrité. Il inclut des fonctionnalités telles que les espaces de noms dans le but de corriger certaines des lacunes du format JAR existant, en particulier le JAR Hell, ce qui peut entraîner des problèmes tels que le chemin de classe et les problèmes de chargement de classe.

Le Java Module System était initialement développé dans le cadre du **Java Community Process** en tant que **JSR 277** et devait être publié avec **Java 7**.

JSR 277 plus tard a été mis en attente et Project Jigsaw a été créé pour modulariser le JDK. Ce JSR a été remplacé par JSR 376 (Java Platform Module System).

Project **Jigsaw** était à l'origine destiné à Java 7 (2011) mais a été reporté à Java 8 (2014) dans le cadre du Plan B, et de nouveau reporté à une version Java 9 en 2017. Java 9, y compris Java Module System, a été publié le 21 septembre 2017.

VI) L'API Java Stream

Un Stream en Java peut être défini comme une séquence d'éléments provenant d'une source. La source des éléments fait ici référence à une collection ou à un tableau qui fournit des données au flux.

- Les Stream Java sont conçus de telle manière que la plupart des opérations de flux (appelées opérations intermédiaires) renvoient un Stream. Cela permet de créer une chaîne d'opérations de flux. C'est ce qu'on appelle un pipeline de flux.

- Les Stream Java prennent également en charge les opérations d'agrégation ou de terminal sur les éléments. Les opérations d'agrégation sont des opérations qui nous permettent d'exprimer rapidement et clairement des manipulations courantes sur des éléments de flux, par exemple, trouver l'élément max ou min, trouver le premier élément correspondant à des critères donnés, etc.

- Non pas qu'un Stream conserve le même ordre des éléments que l'ordre dans la source du Stream.

1. *Qu'est-ce qu'un Stream ?*

Nous avons tous regardé des vidéos en ligne sur YouTube. Lorsque nous commençons à regarder une vidéo, une petite partie du fichier vidéo est d'abord chargée sur notre ordinateur et commence à jouer. Nous n'avons pas besoin de télécharger la vidéo complète avant de commencer à la regarder. C'est ce qu'on appelle le streaming vidéo. À un niveau très élevé, nous pouvons considérer les petites portions du fichier vidéo comme un flux et la vidéo entière comme une collection.

Au niveau granulaire, la différence entre une collection et un flux réside dans le moment où les éléments sont calculés. Une collection est une structure de données en mémoire qui contient toutes les valeurs de la structure de données. Chaque élément de la collection doit être calculé avant de pouvoir être ajouté à la collection. Alors qu'un Stream est conceptuellement un pipeline dans lequel les éléments sont calculés à la demande.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Using a Stream to filter even numbers and then double them
List<Integer> evenNumber = numbers.stream()
   .filter(n -> n % 2 == 0)   // Filter even numbers
   .toList(); // Collect the results into a new list

System.out.println("Even Numbers List: " + evenNumber); // [2, 4, 6, 8, 10]
```

Ce concept donne lieu à des avantages de programmation importants. L'idée est qu'un utilisateur extraira uniquement les valeurs dont il a besoin d'un flux, et ces éléments seront produits de manière invisible pour l'utilisateur, au fur et à mesure des besoins. Il s'agit d'une forme de relation producteur-consommateur.

En Java, l'interface java.util.Stream représente un flux sur lequel une ou plusieurs opérations peuvent être effectuées.

Les opérations de flux sont soit intermédiaires, soit terminales. Les opérations du terminal renvoient un résultat d'un certain type et les opérations intermédiaires renvoient le flux lui-même afin que nous puissions enchaîner plusieurs méthodes d'affilée pour effectuer l'opération en plusieurs étapes.

Les flux sont créés sur une source, par ex. un java.util.Collection comme List ou Set. La Map n'est pas supportée directement, nous pouvons créer un flux de clés, de valeurs ou d'entrées de map.

Les opérations de flux peuvent être exécutées séquentiellement ou en parallèle. Lorsqu'il est effectué en parallèle, on parle de flux parallèle.

Sur la base des points ci-dessus, nous pouvons dire qu'un Stream est :

- Conçu pour les lambdas ou la programmation fonctionnelle
- Pas une structure de données pour stocker des objets
- Ne prend pas en charge l'accès indexé
- Peut facilement être regroupé sous forme de tableaux ou de listes
- Accès paresseux pris en charge

- Parallélisable

2. Créer un Stream

Les méthodes indiquées ci-dessous sont les différentes manières les plus populaires de créer des flux à partir de collections.

a) Stream.of()

Dans l'exemple donné, nous créons un flux d'un nombre fixe d'entiers.

```
Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);
stream.forEach(p -> System.out.println(p));
```

b) Stream.of(array)

Dans l'exemple donné, nous créons un flux à partir du tableau. Les éléments du flux sont extraits du tableau.

```
Stream<Integer> stream = Stream.of( new Integer[]{1,2,3,4,5,6,7,8,9} );
stream.forEach(p -> System.out.println(p));
```

c) List.stream()

Dans l'exemple donné, nous créons un flux à partir de la liste. Les éléments du flux sont extraits de la liste.

```
List<Integer> list = new ArrayList<Integer>();

for(int i = 1; i< 10; i++){
    list.add(i);
}

Stream<Integer> stream = list.stream();
stream.forEach(p -> System.out.println(p));
```

d) Stream.generate() ou Stream.iterate()

Dans l'exemple donné, nous créons un flux à partir d'éléments générés. Cela produira un flux de 20 nombres aléatoires. Nous avons limité le nombre d'éléments à l'aide de la fonction **limit**().

```
Stream<Integer> randomNumbers = Stream
    .generate(() -> (new Random()).nextInt(100));

randomNumbers.limit(20).forEach(System.out::println);
```

e) Stream of String chars ou tokens

Dans l'exemple donné, nous créons d'abord un flux à partir des caractères d'une chaîne donnée. Dans la deuxième partie, nous créons le flux de jetons reçus lors de la séparation d'une chaîne.

```
IntStream stream = "12345_abcdefg".chars();
stream.forEach(p -> System.out.println(p));

//OR

Stream<String> stream = Stream.of("A$B$C".split("\\$"));
stream.forEach(p -> System.out.println(p));
```

Il existe également d'autres moyens, tels que l'utilisation de **Stream.Builder** ou l'utilisation d'opérations intermédiaires. Nous en apprendrons davantage sur eux dans des articles séparés de temps en temps.

3. Stream Collectors

Après avoir effectué les opérations intermédiaires sur les éléments du flux, nous pouvons à nouveau collecter les éléments traités dans une collection à l'aide des méthodes stream Collector.

a) Collecter des éléments de flux dans une liste

Dans l'exemple donné, nous créons d'abord un flux sur les entiers 1 à 10. Ensuite, nous traitons les éléments du flux pour trouver tous les nombres pairs.

Enfin, nous rassemblons tous les nombres pairs dans une liste.

```
for(int i = 1; i< 10; i++){
    list.add(i);
}
Stream<Integer> stream = list.stream();
List<Integer> evenNumbersList = stream.filter(i -> i%2 == 0)
                            .collect(Collectors.toList());
System.out.print(evenNumbersList);
```

b) Collecter des éléments de flux dans un tableau

L'exemple donné est similaire au premier exemple présenté ci-dessus. La seule différence est que nous collectons des nombres pairs dans un tableau.

```
List<Integer> list = new ArrayList<Integer>();

for(int i = 1; i< 10; i++){
    list.add(i);
}
Stream<Integer> stream = list.stream();
Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::new);
System.out.print(evenNumbersArr);
```

Il existe également de nombreuses autres façons de collecter des flux dans un ensemble, une carte ou de plusieurs manières. Suivez simplement le cours de collectionneurs et essayez de les garder à l'esprit.

4. *Operations sur Streams*

L'abstraction de flux a une longue liste de fonctions utiles. Examinons-en quelques-uns.

Avant d'aller de l'avant, construisons au préalable une liste de chaînes. Nous construirons nos exemples sur cette liste afin qu'elle soit facile à comprendre et à comprendre.

```
List<String> memberNames = new ArrayList<>();
memberNames.add("Amitabh");
memberNames.add("Shekhar");
memberNames.add("Aman");
memberNames.add("Rahul");
memberNames.add("Shahrukh");
memberNames.add("Salman");
memberNames.add("Yana");
memberNames.add("Lokesh");
```

Ces méthodes de base ont été divisées en 2 parties ci-dessous :

a) Opérations intermédiaires

Les opérations intermédiaires renvoient le flux lui-même afin que vous puissiez enchaîner plusieurs appels de méthodes d'affilée. Apprenons les plus importants.

Stream.filter()

La méthode filter() accepte un Predicate pour filtrer tous les éléments du flux. Cette opération est intermédiaire, nous permettant d'appeler une autre opération de flux (par exemple forEach()) sur le résultat.

```
memberNames.stream().filter((s) -> s.startsWith("A"))
                .forEach(System.out::println);

Sortie du programme :

Amitabh
Aman
```

Stream.map()

L'opération intermédiaire map() convertit chaque élément du flux en un autre objet via la fonction donnée.

L'exemple suivant convertit chaque chaîne en chaîne MAJUSCULE. Mais nous pouvons également utiliser map() pour transformer un objet en un autre type.

```
memberNames.stream().filter((s) -> s.startsWith("A"))
                .map(String::toUpperCase)
                .forEach(System.out::println);

Sortie du programme :

AMITABH
AMAN
```

Stream.sorted()

La méthode sorted() est une opération intermédiaire qui renvoie une vue triée du flux. Les éléments du flux sont triés dans l'ordre naturel, sauf si nous transmettons un comparateur personnalisé.

```
memberNames.stream().sorted()
                .map(String::toUpperCase)
                .forEach(System.out::println);

Sortie du programme :

AMAN
AMITABH
LOKESH
RAHUL
SALMAN
SHAHRUKH
SHEKHAR
YANA
```

Veuillez noter que la méthode sorted() crée uniquement une vue triée du flux sans manipuler l'ordre de la collection source. Dans cet exemple, l'ordre des chaînes dans les noms de membres n'est pas modifié.

b) Opérations terminales

Les opérations du terminal renvoient un résultat d'un certain type après avoir traité tous les éléments du flux.

Une fois l'opération du terminal invoquée sur un Stream, l'itération du Stream et de l'un des flux chaînés commencera. Une fois l'itération terminée, le résultat de l'opération du terminal est renvoyé.

Stream.forEach()

La méthode forEach() permet de parcourir tous les éléments du flux et d'effectuer certaines opérations sur chacun d'eux. L'opération à effectuer est passée sous forme d'expression lambda.

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

Stream.collect()

La méthode collect() est utilisée pour recevoir des éléments de Steam et les stocker dans une collection.

```
List<String> memNamesInUppercase = memberNames.stream().sorted()
                    .map(String::toUpperCase)
                    .collect(Collectors.toList());

System.out.print(memNamesInUppercase);
```

```
Sortie du programme :

[AMAN, AMITABH, LOKESH, RAHUL, SALMAN, SHAHRUKH, SHEKHAR, YANA]
```

Stream.match()

Diverses opérations de correspondance peuvent être utilisées pour vérifier si un prédicat donné correspond aux éléments du flux. Toutes ces opérations de correspondance sont terminales et renvoient un résultat booléen.

```
boolean matchedResult = memberNames.stream()
      .anyMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);    //true

matchedResult = memberNames.stream()
      .allMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);    //false

matchedResult = memberNames.stream()
      .noneMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);    //false
```

Stream.count()

Le count() est une opération de terminal renvoyant le nombre d'éléments dans le flux sous forme de valeur longue.

```
long totalMatched = memberNames.stream()
   .filter((s) -> s.startsWith("A"))
   .count();

System.out.println(totalMatched);     //2
```

Stream.reduce()

La méthode reduce() effectue une réduction sur les éléments du flux avec la fonction donnée. Le résultat est un facultatif contenant la valeur réduite.

Dans l'exemple donné, nous réduisons toutes les chaînes en les concaténant à l'aide d'un séparateur #.

```
Optional<String> reduced = memberNames.stream()
       .reduce((s1,s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);

Sortie du programme :

Amitabh#Shekhar#Aman#Rahul#Shahrukh#Salman#Yana#Lokesh
```

5. *Opérations de court-circuit*

Bien que les opérations de flux soient effectuées sur tous les éléments d'une collection satisfaisant un prédicat, il est souvent souhaité d'interrompre l'opération chaque fois qu'un élément correspondant est rencontré au cours de l'itération.

En itération externe, nous ferons avec le bloc if-else. Dans les itérations internes telles que dans les flux, nous pouvons utiliser certaines méthodes à cette fin.

Stream.anyMatch()

anyMatch() retournera true une fois qu'une condition passée comme prédicat sera satisfaite. Une fois qu'une valeur correspondante est trouvée, plus aucun élément ne sera traité dans le flux.

Dans l'exemple donné, dès qu'une chaîne commençant par la lettre « A » est trouvée, le flux se terminera et le résultat sera renvoyé.

```
boolean matched = memberNames.stream()
      .anyMatch((s) -> s.startsWith("A"));
System.out.println(matched);    //true
```

Stream.findFirst()

La méthode findFirst() renverra le premier élément du flux et ne traitera ensuite plus d'éléments.

```
String firstMatchedName = memberNames.stream()
        .filter((s) -> s.startsWith("L"))
        .findFirst().get();
System.out.println(firstMatchedName);    //Lokesh
```

6. *Parallel Streams*

Avec le framework **Fork/Join** ajouté à Java SE 7, nous disposons de machines efficaces pour implémenter des opérations parallèles dans nos applications.

Mais implémenter un framework **fork/join** est une tâche complexe, et si elle n'est pas bien faite ; c'est une source de bugs multithread complexes qui peuvent potentiellement faire planter l'application. Avec l'introduction des itérations internes, nous avons eu la possibilité d'effectuer des opérations en parallèle plus efficacement.

Pour activer le parallélisme, tout ce que nous avons à faire est de créer un flux parallèle, au lieu d'un flux séquentiel. Et à notre grande surprise, c'est vraiment très simple.

Dans l'un des exemples de flux répertoriés ci-dessus, chaque fois que nous souhaitons effectuer un travail particulier en utilisant plusieurs threads dans des cœurs parallèles, il nous suffit d'appeler la méthode parallelStream() au lieu de la méthode stream().

```
List<Integer> list = new ArrayList<Integer>();
for(int i = 1; i< 10; i++){
 list.add(i);
}
//Here creating a parallel stream
Stream<Integer> stream = list.parallelStream();

Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::new);
System.out.print(evenNumbersArr);
```

L'un des principaux moteurs des API Stream est de rendre le parallélisme plus accessible aux développeurs. Bien que la plate-forme Java fournisse déjà un solide support pour la concurrence et le parallélisme, les développeurs sont confrontés à des obstacles inutiles lors de la migration de leur code du séquentiel vers le parallèle, selon les besoins.

Par conséquent, il est important d'encourager les idiomes à la fois séquentiels et parallèles. Ceci est facilité en déplaçant l'accent vers la description du type de calcul qui doit être effectué plutôt que de la manière dont il doit être effectué.

Il est également important de trouver un équilibre entre faciliter le parallélisme et ne pas aller jusqu'à le rendre invisible. Rendre le parallélisme transparent introduirait le non-déterminisme et la possibilité de courses aux données là où les utilisateurs ne s'y attendraient pas.

VII) Les apports en programmation concurrente (JDK 7 et plus)

1. Applications multithreads avec API Java Concurrency

a) Introduction

L'API Java Concurrency est un ensemble de packages et de classes Java développés pour créer des applications multithreads. Il a été introduit dans Java 5 et vise à faciliter l'écriture de code concurrent et parallèle en Java. L'API Java Concurrency propose des classes et des utilitaires qui permettent aux développeurs de créer et de gérer des threads, de synchroniser l'accès aux ressources partagées et d'exécuter des algorithmes parallèles. Il comprend des fonctionnalités telles que des pools de threads, des verrous, des variables de condition et des variables atomiques, qui aident à gérer la synchronisation et la coordination des threads. Cependant, la mise en œuvre de systèmes multithread peut être complexe et sujette aux erreurs, et l'utilisation de l'API Java Concurrency nécessite une bonne compréhension de ses fonctionnalités et de ses meilleures pratiques. Cet article vise à fournir une introduction à l'API Java Concurrency et à ses fonctionnalités de base pour aider les développeurs à se lancer dans l'écriture de programmes simultanés et parallèles en Java.

b) Comprendre les threads en Java

Les threads constituent un concept fondamental de la concurrence Java et il est nécessaire de comprendre leur fonctionnement pour concevoir des applications multithreads avec l'API Java Concurrency.

Un thread peut être considéré comme un chemin d'exécution indépendant au sein d'une application Java. Chaque thread possède sa propre pile et son propre compteur de programme et peut exécuter du code simultanément avec d'autres threads. Les applications Java démarrent généralement avec un seul thread d'exécution appelé thread principal, qui est responsable de l'exécution de la méthode main().

En Java, les développeurs peuvent soit étendre la classe Thread, soit implémenter l'interface Runnable pour créer un nouveau thread. La classe Thread inclut une méthode run() qui sert de point d'entrée au thread et peut être remplacée par le développeur pour définir le comportement du thread. Alternativement, l'interface Runnable fournit une seule méthode run() que le développeur peut implémenter et envoyer en tant que paramètre au constructeur Thread.

Après avoir créé un thread, il peut être démarré avec la méthode start(). Cette méthode lance l'exécution du thread et appelle la méthode run(). Lorsque la méthode run() se termine, le thread se termine.

Les threads Java peuvent être mis en pause, repris et terminés à l'aide de méthodes telles que sleep(), wait() et interrompu(). En outre, les threads peuvent communiquer entre eux et synchroniser l'accès aux ressources partagées en utilisant des structures de synchronisation telles que des blocs synchronisés, des verrous et des variables de condition.

Il est important de se rappeler que la programmation multithread peut être complexe et sujette à des erreurs telles que des conditions de concurrence critique et des blocages. Il est essentiel de suivre les meilleures pratiques lors du développement d'applications multithread en Java, comme éviter l'état mutable partagé, utiliser des objets immuables et minimiser la synchronisation.

c) Concept de synchronisation et de sécurité des threads

Synchronisation

La synchronisation est le processus de contrôle de l'accès aux ressources partagées dans un environnement multithread.

La synchronisation est implémentée à l'aide de constructions telles que des méthodes synchronisées, des instructions synchronisées et des verrous.

Lorsqu'une méthode ou une instruction est synchronisée, un seul thread peut y accéder à la fois. Tous les threads suivants qui tentent d'accéder à la même méthode ou instruction sont bloqués jusqu'à ce que le thread initial libère le verrou.

La synchronisation est requise pour éviter les conditions de concurrence, qui se produisent lorsque deux ou plusieurs threads accèdent à une ressource partagée en même temps, entraînant une corruption des données.

Par exemple, considérons un scénario dans lequel deux threads tentent de mettre à jour simultanément une variable de compteur partagée. Si les deux threads lisent la valeur du compteur, l'incrémentent et la réécrivent, le résultat peut être incorrect car chaque thread n'est pas au courant de la mise à jour de l'autre.

Sécurité des threads

La sécurité des threads fait référence à la capacité d'un programme à fonctionner correctement dans un environnement multithread, sans corruption de données, conditions de concurrence critique ou autres bogues liés à la concurrence. La sécurité des threads est importante pour les structures de données partagées, telles que les listes, les cartes et les ensembles, auxquelles plusieurs threads peuvent accéder simultanément. Java inclut diverses classes de collection thread-safe, telles que ConcurrentHashMap et CopyOnWriteArrayList, qui garantissent un accès simultané sûr et efficace. Les développeurs doivent prêter une attention particulière à la synchronisation et à la sécurité des threads lors du développement d'applications multithread, car une synchronisation incorrecte peut entraîner des blocages, des livelocks et d'autres problèmes liés à la concurrence. Les meilleures pratiques en matière de synchronisation et de sécurité des threads incluent la réduction de l'état mutable partagé, l'utilisation d'objets immuables et l'utilisation d'abstractions de haut niveau comme le package java.util.concurrent.

d) Création et gestion de threads en Java

La création et la gestion de threads en Java sont une étape nécessaire pour développer des applications multithreads qui utilisent l'API Java Concurrency. Voici quelques exemples pour expliquer le processus :

- Les threads Java peuvent être créés en étendant la classe Thread ou en implémentant l'interface Runnable.

- L'extension de la classe Thread implique de remplacer la méthode run(), qui détermine le comportement du thread. L'implémentation de l'interface Runnable nécessite l'implémentation de la méthode run() et sa transmission en tant que paramètre au constructeur Thread. Cela contribue également à la surcharge du constructeur.

- Après avoir créé un thread, il peut être démarré à l'aide de la méthode start(). Cela lance l'exécution du thread et appelle la méthode run().

- Le thread peut également être mis en pause à l'aide de la méthode wait(), ce qui le fait attendre jusqu'à ce qu'un autre thread lui informe qu'il est prêt à redémarrer.

- Les threads Java peuvent communiquer entre eux et utiliser des constructions de synchronisation telles que des blocs synchronisés, des verrous et des variables de condition pour synchroniser l'accès aux ressources partagées.

- L'interruption des threads Java est également possible avec la méthode interrupt (), qui définit un indicateur indiquant que le thread doit cesser de s'exécuter.

- L'API Java Concurrency inclut de nombreuses abstractions de haut niveau pour la gestion des threads, telles que le framework Executor, qui simplifie la gestion des pools de threads et l'exécution des tâches.

Le développement d'applications Java multithread nécessite de prêter une attention particulière à la sécurité des threads et à la synchronisation, car une utilisation inappropriée des constructions de concurrence peut entraîner des erreurs subtiles et difficiles à déboguer.

e) Pools de threads et exécuteurs en Java

- Thread Pool (Pool de threads) : un pool de threads est un groupe de threads qui peuvent être réutilisés pour effectuer plusieurs tâches simultanément. Un pool de threads maintient un pool de threads qui peuvent être utilisés pour exécuter des tâches plutôt que de créer et de détruire des threads à chaque fois qu'une tâche doit être exécutée. Le pool de threads alloue des tâches aux threads disponibles dans le pool, réduisant ainsi la surcharge liée à la création et à la destruction des threads.
- Executor (Exécuteur) : Un exécuteur est un objet qui gère l'exécution de tâches dans un pool de threads. Les exécuteurs de l'API Java Concurrency créent et gèrent des threads abstraits et fournissent une API de haut niveau pour exécuter des tâches dans un environnement multithread.

Avantages de l'utilisation du pool de threads et de l'exécuteur

- Utilisation efficace des ressources : les pools de threads et les exécuteurs permettent une utilisation efficace des ressources système en réutilisant les threads au lieu de créer et de détruire des threads à plusieurs reprises.

- Performances améliorées : les pools de threads et les exécuteurs peuvent améliorer les performances des applications en exécutant des tâches simultanément, réduisant ainsi le temps nécessaire pour terminer les tâches.

- Simplifiez la gestion de la concurrence : les pools de threads et les exécuteurs éliminent la complexité de la gestion des threads, facilitant ainsi l'écriture de code simultané qui est à la fois thread-safe et évolutif.

f) Meilleures pratiques pour développer des applications multithread en Java

Le développement d'applications multithread en Java peut s'avérer difficile en raison de la complexité impliquée dans la gestion des threads et la garantie de la sécurité des threads. Voici quelques bonnes pratiques pour écrire des applications multithreads efficaces, évolutives et sécurisées pour les threads en Java.

Utiliser des pools de threads et des exécuteurs

Les pools de threads et les exécuteurs aident à la gestion et à la réutilisation efficaces des threads. Ils résument la complexité de la gestion des threads et fournissent une API de haut niveau pour exécuter des tâches dans un environnement multithread. L'utilisation de pools de threads et d'exécuteurs peut vous aider à écrire du code évolutif et efficace.

Évitez l'état mutable partagé

Un état mutable partagé se produit lorsque deux ou plusieurs threads partagent une structure de données et qu'au moins un thread peut la modifier. L'état mutable partagé peut entraîner des conditions de concurrence critique, des blocages et d'autres problèmes de concurrence. Pour surmonter ces problèmes, utilisez des structures de données immuables ou synchronisez l'accès aux états mutables partagés.

Utiliser des variables atomiques

Les variables atomiques sont des variables thread-safe qui peuvent être mises à jour de manière atomique. Ils fournissent un moyen d'éviter les conditions de concurrence lorsque plusieurs threads accèdent à la même variable. Les variables atomiques sont disponibles dans l'API Java Concurrency et peuvent être utilisées pour garantir la sécurité des threads.

Utiliser des variables volatiles

Les variables volatiles sont des variables qui sont toujours lues et écrites à partir de la mémoire principale plutôt que du cache local d'un thread. Les variables volatiles vous permettent de garantir que la valeur d'une variable est visible par tous les threads. L'utilisation de variables volatiles peut vous aider à éviter les problèmes de concurrence tels que les conditions de concurrence.

Utiliser la synchronisation et les verrous

La synchronisation et les verrous permettent de garantir qu'un seul thread à la fois peut accéder à une section critique de code. Vous devez utiliser la synchronisation et les verrous chaque fois que vous avez besoin d'accéder à un état mutable partagé.

Utiliser des classes Thread-Safe

L'API Java Concurrency comprend diverses classes thread-safe pour écrire du code thread-safe. Par exemple, la classe ConcurrentHashMap est une implémentation thread-safe de l'interface Map. L'utilisation de classes thread-safe vous permet de créer du code thread-safe sans vous soucier des problèmes de concurrence sous-jacents.

Utiliser des classes immuables

Les classes immuables sont des classes dont les instances ne peuvent pas être modifiées une fois créées. Ils sont du fait de leur conception thread-safe et peuvent être utilisés pour faciliter la gestion de la concurrence.

Utiliser des constructions de concurrence de haut niveau

L'API Java Concurrency comprend diverses constructions de concurrence de haut niveau, telles que CountDownLatch, CyclicBarrier et Semaphore, qui peuvent être utilisées pour faciliter la gestion de la concurrence. Ces constructions vous permettent de coordonner plusieurs threads et de garantir qu'ils s'exécutent dans un ordre spécifique.

g) Conclusion

- Développer des applications multithreads est essentiel pour améliorer les performances et la réactivité des applications.

- L'API Java Concurrency fournit une large gamme d'outils et de mécanismes pour développer des applications multithreads, notamment la création et la gestion de threads, la synchronisation, la sécurité des threads, les pools de threads et les exécuteurs.

- Comprendre les bases de la programmation multithread et de l'API Java Concurrency est crucial pour développer des applications efficaces, fiables et évolutives.

- Il est important de suivre les meilleures pratiques, telles que minimiser l'utilisation de variables globales, utiliser des collections thread-safe, éviter les blocages et gérer correctement la synchronisation des threads.

- Les variables atomiques et les objets de verrouillage sont des mécanismes importants fournis par l'API Java Concurrency pour garantir la sécurité des threads et éviter les problèmes de concurrence.

- Le développement d'applications multithread avec l'API Java Concurrency nécessite une planification, des tests et un débogage minutieux pour garantir que l'application est évolutive, fiable et efficace.

- Avec une utilisation appropriée de l'API Java Concurrency, les développeurs peuvent créer des applications capables de gérer de grandes quantités de données, de traiter plusieurs requêtes simultanément et d'offrir une meilleure expérience utilisateur.

2. *Le package java.util.concurrent*

Le package Java Concurrency couvre la concurrence, le multithreading et le parallélisme sur la plate-forme Java. La concurrence est la capacité d'exécuter plusieurs ou plusieurs programmes ou applications en parallèle. L'épine dorsale de la concurrence Java est constituée de threads (un processus léger, qui possède ses propres fichiers et piles et peut accéder aux données partagées à partir d'autres threads du même processus). Le débit et l'interactivité du programme peuvent être améliorés en effectuant des tâches chronophages de manière asynchrone ou en parallèle. Java 5 a ajouté un nouveau package à la plate-forme Java → package java.util.concurrent. Ce package contient un ensemble de classes et d'interfaces qui aident au développement d'applications simultanées (multithreading) en Java. Avant ce package, il faut créer soi-même les classes utilitaires correspondant à ses besoins.
Vous trouverez ci-dessous les principaux composants/utilitaires du package java.util.concurrent

c) Executor

L'exécuteur est un ensemble d'interfaces qui représente un objet dont l'implémentation exécute des tâches. Cela dépend de l'implémentation si la tâche doit être exécutée sur un nouveau thread ou sur un thread actuel. Par conséquent, nous pouvons découpler le flux d'exécution des tâches du mécanisme d'exécution des tâches réel, en utilisant cette interface. L'exécuteur n'exige pas que l'exécution de la tâche soit asynchrone. Le plus simple de tous est l'interface exécutable.

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

Afin de créer une instance d'exécuteur, nous devons créer un invocateur.

```
public class Invoker implements Executor {
  @Override
  public void execute(Runnable r) {
     r.run();
  }
}
```

Désormais, pour l'exécution de la tâche, nous pouvons utiliser cet invocateur.

```
public void execute() {
  Executor exe = new Invoker();
  exe.execute( () -> {
     // task to be performed
  });
}
```

Si l'exécuteur ne peut pas accepter la tâche à exécuter, il lancera une exception **RejectedExecutionException**.

d) ExecutorService

ExecutorService est une interface et force uniquement l'implémentation sous-jacente à implémenter la méthode execute(). Il étend l'interface **Executor** et ajoute une série de méthodes qui exécutent des threads qui renvoient une valeur. Les méthodes pour fermer le pool de threads ainsi que la possibilité de les mettre en œuvre pour le résultat de l'exécution de la tâche.

Nous devons créer une cible **Runnable** pour utiliser **ExecutorService**.

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

    // task details
  }
}
```

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

```
// 20 est la taille du pool de threads
ExecutorService executor = Executors.newFixedThreadPool(20);
```

Pour la création d'une instance ExecutorService à thread unique, 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());
}
```

Nous pouvons également 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 :

- shutdown() : il attend que l'exécution de toutes les tâches soumises soit terminée.
- shutdownNow() : Il termine immédiatement toutes les tâches en exécution/en attente.

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

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

e) ScheduledExecutorService

Il est similaire à **ExecutorService**. La différence est que cette interface peut effectuer des tâches périodiquement. Les fonctions Runnable et Callable sont utilisées pour définir la tâche.

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

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

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

  executorService.shutdown();
}
```

ScheduledExecutorService peut également définir une tâche après un certain délai fixe.

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

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

- **scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)** : cette méthode crée et exécute une action périodique qui est d'abord invoquée après le délai initial, puis avec la période donnée jusqu'à l'arrêt de l'instance de service.

- **scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)** : cette méthode crée et exécute une action périodique qui est invoquée d'abord après le délai initial fourni et à plusieurs reprises avec le délai donné entre la fin de l'exécution et l'invocation de l'action le prochain.

f) Future

Il représente le résultat d'une opération asynchrone. Les méthodes qu'il contient vérifient si l'opération asynchrone est terminée ou non, obtiennent le résultat terminé, etc. L'API cancel(boolean isInterruptRunning) annule l'opération et libère le thread en cours d'exécution. Si la valeur de isInterruptRunning est vraie, le thread exécutant la tâche sera terminé instantanément. Sinon, toutes les tâches en cours sont terminées.

L'extrait de code ci-dessous crée une instance de Future.

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

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

Le code pour vérifier si le résultat du futur est prêt ou non et récupère les données lorsque le calcul est terminé.

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

Spécification du délai d'attente pour une opération donnée. Si le temps nécessaire est supérieur à ce temps, TimeoutException est levée.

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

g) CountDownLatch

Il s'agit d'une classe utilitaire qui bloque un ensemble de threads jusqu'à ce que certaines opérations soient terminées. Un **CountDownLatch** est initialisé avec un compteur (qui est de type Integer). Ce compteur décrémente à mesure que l'exécution des threads dépendants est terminée. Mais une fois que le compteur revient à zéro, d'autres threads sont libérés.

h) CyclicBarrier

CyclicBarrier est presque identique à **CountDownLatch** sauf que nous pouvons le réutiliser. Il permet à plusieurs threads de s'attendre en utilisant wait() avant d'invoquer la tâche finale et cette fonctionnalité n'est pas dans CountDownLatch.

Nous devons créer une instance de Runnable Task pour lancer la condition de barrière.

```
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();
    }
  }
```

Maintenant, invoquons quelques threads pour répondre à la condition de 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, la méthode isBroken() vérifie si l'un des threads a été interrompu pendant le temps d'exécution.

i) Semaphore

Il est utilisé pour bloquer l'accès au niveau du thread à une partie de la ressource logique ou physique. Le sémaphore contient un ensemble de permis. Chaque fois que le thread tente d'entrer la partie code d'une section critique, le sémaphore donne la permission si le permis est disponible ou non, ce qui signifie que la section critique est disponible ou non. Si le permis n'est pas disponible, le thread ne peut pas accéder à la section critique.

Il s'agit essentiellement d'une variable nommée compteur qui maintient le nombre de threads entrants et sortants de la section critique. Lorsque le thread en cours d'exécution libère la section critique, le compteur augmente.

Le code ci-dessous est utilisé pour l'implémentation de Semaphore :

```
static Semaphore semaphore = new Semaphore(20);

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.

j) ThreadFactory

Il agit comme un pool de threads qui crée un nouveau thread à la demande. ThreadFactory peut être défini comme :

```
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;
  }
}
```

k) BlockingQueue

L'interface BlockingQueue prend en charge le contrôle de flux (en plus de la file d'attente) en introduisant le blocage si BlockingQueue est plein ou vide. Un thread essayant de mettre un élément dans une file d'attente pleine est bloqué jusqu'à ce qu'un autre thread libère de l'espace dans la file d'attente, soit en retirant un ou plusieurs éléments de la file d'attente, soit en effaçant complètement la file d'attente. De même, il bloque un thread essayant de supprimer d'une file d'attente vide jusqu'à ce que d'autres threads insèrent un élément. BlockingQueue n'accepte pas de valeur nulle. Si nous essayons de mettre l'élément nul en file d'attente, il lève NullPointerException.

l) DelayQueue

DelayQueue est une file d'attente prioritaire spécialisée qui classe les éléments en fonction de leur temps de retard. Cela signifie que seuls les éléments dont le temps est expiré peuvent être retirés de la file d'attente. L'en-tête DelayQueue contient l'élément qui a expiré depuis le moins de temps. Si aucun délai n'a expiré, alors il n'y a pas de tête et le sondage renverra null. DelayQueue accepte uniquement les éléments qui appartiennent à une classe de type Delayed. DelayQueue implémente la méthode getDelay() pour renvoyer le temps de retard restant.

m) Lock

Il s'agit d'un utilitaire permettant d'empêcher d'autres threads d'accéder à un certain segment de code. La différence entre Lock et un bloc synchronisé est que nous avons les API de verrouillage lock() et unlock() dans des méthodes distinctes alors qu'un bloc synchronisé est entièrement contenu dans les méthodes.

n) Phaser

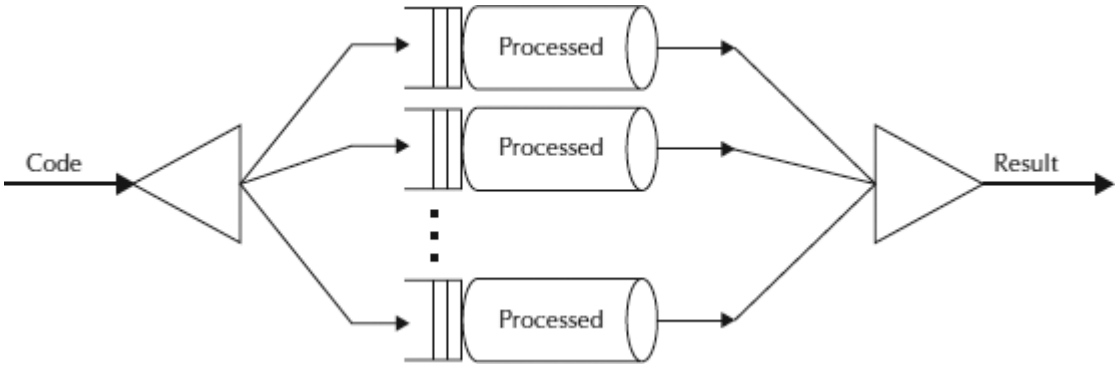
Il est plus flexible que CountDownLatch et CyclicBarrier. Phaser est utilisé pour agir comme une barrière réutilisable sur laquelle le nombre dynamique de threads doit attendre avant que l'exécution ne continue. Plusieurs phases d'exécution peuvent être coordonnées en réutilisant l'instance d'un Phaser pour chaque phase du programme.

3. Le Fork and Join

L'utilisation efficace de cœurs parallèles dans un programme Java a toujours été un défi. Peu de frameworks développés en interne répartiraient le travail sur plusieurs cœurs, puis les rejoindraient pour renvoyer l'ensemble de résultats. **Java 7** a incorporé cette fonctionnalité en tant que framework Fork and Join.

a) Framework Fork/Join

Le **Fork-Join** divise la tâche à accomplir en sous-tâches jusqu'à ce que la mini-tâche soit suffisamment simple pour la résoudre sans autres ruptures. C'est comme un algorithme diviser pour régner. Un concept crucial dans ce cadre est qu'aucun thread de travail n'est inactif. Ils mettent en œuvre un algorithme de vol de travail dans lequel les travailleurs inactifs volent le travail des travailleurs occupés.



Il est basé sur les travaux de Doug Lea, un leader d'opinion en matière de concurrence Java. **Fork/Join** gère les problèmes de threading ; vous indiquez au framework quelles parties du travail peuvent être divisées et traitées de manière récursive.

```
Result solve(Problem problem) {
   if (problem is small)
      directly solve problem
   else {
      split problem into independent parts
      fork new subtasks to solve each part
      join all subtasks
      compose result from subresults
   }
}
```

b) Core Classes

Les classes principales prenant en charge le mécanisme **Fork-Join** sont **ForkJoinPool** (https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html) et **ForkJoinTask** (https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html).

Découvrons leurs rôles en détail.

ForkJoinPool

ForkJoinPool est une implémentation spécialisée **d'ExecutorService** implémentant l'algorithme de vol de travail discuté ci-dessus. Nous créons une instance de **ForkJoinPool** en fournissant le niveau de parallélisme cible, c'est-à-dire le nombre de processeurs.

Si vous utilisez un constructeur sans argument, il crée par défaut un pool de taille égale au nombre de processeurs disponibles obtenus à l'aide de la technique donnée.

```
var numberOfProcessors = Runtime.getRunTime().availableProcessors();

ForkJoinPool pool = new ForkJoinPool(numberOfProcessors);
```

Même si vous spécifiez une taille de pool initiale, celui-ci ajuste sa taille de manière dynamique pour conserver suffisamment de threads actifs à tout moment. Une autre différence significative par rapport aux autres **ExecutorService** est que ce pool n'a pas besoin d'être explicitement arrêté à la sortie du programme car tous ses threads sont en mode démon.

Il existe trois manières différentes de soumettre une tâche à **ForkJoinPool**.

- **execute () –** Exécution asynchrone souhaitée ; appelez sa méthode fork pour diviser le travail entre plusieurs threads.
- **invoke () –** Attendre d'obtenir le résultat ; appelez la méthode d'invocation sur le pool.

- **submit () –** Renvoie un objet Future que vous pouvez utiliser pour vérifier l'état et obtenir le résultat une fois terminé.

ForkJoinTask

ForkJoinTask est une classe abstraite permettant de créer des tâches qui s'exécutent dans un **ForkJoinPool**. **RecursiveAction** et **RecursiveTask** sont les deux seules sous-classes directes et connues de **ForkJoinTask**.

La seule différence entre ces deux classes est que **RecursiveAction** ne renvoie pas de valeur tandis que **RecursiveTask** a une valeur de retour et renvoie un objet du type spécifié.

Dans les deux cas, vous devrez implémenter la méthode **compute** () dans votre sous-classe qui effectue le calcul principal souhaité par la tâche.

La classe **ForkJoinTask** fournit plusieurs méthodes pour vérifier l'état d'exécution d'une tâche.

- **isDone()** : renvoie vrai si une tâche se termine d'une manière ou d'une autre.
- **isCompletedNormally()** : renvoie true si une tâche se termine sans annulation ni rencontre d'exception.
- **isCancelled()** : renvoie true si la tâche a été annulée.
- **isCompletedabnormally()** : renvoie vrai si la tâche a été annulée ou a généré une exception.