Java - Nouveautés des versions 8 à 16

Christopher Loisel

Autres apports de la JDK 9

1. Améliorations des streams Java 8
2. L'API HTTP 2
3. Nouvelles collections
4. Les Reactive Streams, les streams asynchrones
5. Méthodes privées d'interface
6. L'API CompletableFuture

Améliorations de l'API Stream

Dans ce cours, nous allons nous concentrer sur les améliorations intéressantes apportées à l'API Stream dans Java 9.

Stream Takewhile/Dropwhile

Des discussions sur ces méthodes sont apparues à plusieurs reprises sur StackOverflow (la plus populaire est celle-ci).

Imaginons que nous voulions générer un stream depuis un string. En ajoutant un character après l'autre, jusqu'à ce que la valeur dans ce Stream soit inférieure à 10.

Comment résoudre ce problème en Java 8 ? Nous pourrions utiliser une des opérations intermédiaires de court-circuitage comme limit, allMatch qui servent en fait à d'autres fins ou écrire notre propre implémentation de takeWhile basée sur un Spliterator qui, à son tour, complique un problème aussi simple.

Avec Java 9, la solution est simple :

```
Stream<String> stream = Stream.iterate("", s -> s + "s").takeWhile(s -> s.length() < 10);
```

L'opération takeWhile prend un prédicat qui est appliqué aux éléments pour déterminer le plus long préfixe de ces éléments (si un flux est ordonné) ou un sous-ensemble des éléments du flux (lorsqu'un flux est non ordonné).

Pour avancer, nous devons mieux comprendre ce que signifient les termes "le préfixe le plus long" et "le sous-ensemble d'un flux" :

- le préfixe le plus long est une séquence contiguë d'éléments du flux qui correspondent au prédicat donné.
- Le premier élément de la séquence est le premier élément de ce flux, et l'élément qui suit immédiatement le dernier élément de la séquence ne correspond pas au prédicat donné.

le sous-ensemble d'un flux est un ensemble de certains (mais pas tous) éléments du flux qui correspondent au prédicat donné.

Après avoir introduit ces termes clés, nous pouvons facilement comprendre une autre nouvelle opération dropWhile.

Elle fait exactement l'inverse de takeWhile. Si un flux est ordonné, dropWile renvoie un flux constitué des éléments restants de ce flux après avoir supprimé le plus long préfixe des éléments qui correspondent au prédicat donné.

Sinon, si un flux n'est pas ordonné, la fonction dropWile renvoie un flux constitué des éléments restants de ce flux après avoir supprimé un sous-ensemble d'éléments correspondant au prédicat donné.

Jetons les cinq premiers éléments en utilisant le Stream obtenu précédemment :

```
stream.dropWhile(s -> !s.contains("sssss"));
```

En d'autres termes, l'opération dropWhile supprime les éléments tant que le prédicat donné pour un élément renvoie vrai et cesse de supprimer les éléments lorsque le premier prédicat renvoie faux.

Stream Iterate

La nouvelle fonctionnalité suivante est la méthode iterate surchargée pour la génération de Streams finis. À ne pas confondre avec la variante finite qui renvoie un Stream ordonné infini produit par une fonction.

La nouvelle méthode iterate() modifie légèrement cette méthode en ajoutant un prédicat qui s'applique aux éléments pour déterminer quand le Stream doit se terminer. Son utilisation est très pratique et concise :

```
Stream.iterate(0, i -> i < 10, i -> i + 1).forEach(System.out::println);
```

Il peut être associé à l'instruction for correspondante :

```
for (int i = 0; i < 10; ++i) {
    System.out.println(i);
}
```

Stream Ofnullable

Il y a des situations où nous avons besoin de mettre un élément dans un Stream. Parfois, cet élément peut être un null, mais nous ne voulons pas que notre Stream contient de telles valeurs. Cela entraîne l'écriture d'une instruction if ou d'un opérateur ternaire qui vérifie si un élément est un null.

En supposant que les variables collection et map, ont été créées et remplies avec succès, regardez l'exemple suivant :

```
collection.stream().flatMap(s ->
{
    Integer temp = map.get(s);
    return temp != null ? Stream.of(temp) : Stream.empty();
}).collect(Collectors.toList());
```

Pour éviter ce genre de code passe-partout, la méthode ofNullable a été ajoutée à la classe Stream. Avec cette méthode, l'exemple précédent peut être simplement transformé en :

```
collection.stream()
  .flatMap(s -> Stream.ofNullable(map.get(s)))
  .collect(Collectors.toList());
```

L'API HTTP 2

Client HTTP/2 : Le client HTTP/2 est l'une des fonctionnalités de JDK 9 . HTTP/2 est la dernière version du protocole HTTP . À l'aide du client HTTP/2, à partir de l'application Java, nous pouvons envoyer la requête HTTP et nous pouvons traiter la réponse HTTP.

Avant JDK 9, pour envoyer une requête HTTP et traiter la réponse HTTP, nous utilisons la classe HttpURLConnection . Maintenant, vous vous demandez pourquoi le client HTTP/2 est introduit dans JDK 9 alors que nous avons HttpURLConnection pour résoudre le problème.

Mais la HttpURLConnection existante a peu de problèmes qui sont éliminés dans le client HTTP/2.

Avantages du client HTTP/2.0 :

1. Le client HTTP/2.0 est très léger et facile à utiliser. Le client HTTP/2.0 prend en charge HTTP/1.1 et HTTP/2.0 . HTTP/2 se concentre sur la manière dont les données sont structurées et transportées entre le serveur et le client. Dans HTTP/1.1, nous ne pouvons pas avoir plus de six connexions ouvertes à la fois, donc chaque requête doit attendre que les autres se terminent. La solution au problème ci-dessus est le multiplexage dans le client HTTP/2. Cela signifie qu'à l'aide de HTTP/2, vous pouvez envoyer plusieurs requests HTTP en parallèle sur une seule connexion TCP.
2. Dans HTTP/1.1, lorsque nous envoyons une requête HTTP contenant les informations de l'en-tête, HTTP/1.1 considère les données d'en-tête comme des données supplémentaires, ce qui augmente la bande passante. Cela peut être éliminé dans HTTP/2.0 en utilisant HPack pour la compression d'en-tête.
3. Le client HTTP/2.0 prend en charge les données textuelles et binaires pour le traitement.
4. Le client HTTP/2.0 fonctionne en mode synchrone et asynchrone. . Le client HTTP/2.0 offre de bien meilleures performances par rapport à HttpURLConnection.

API HTTP dans Java 9 : Dans Java 9, une nouvelle API a été introduite, facile à utiliser et qui ajoute également la prise en charge de HTTP/2. Trois nouvelles classes ont été introduites pour gérer la communication HTTP. Ces trois classes sont présentes dans le module jdk.incubator.httpclient et le package jdk.incubator.http dans le module. Comme jdk.incubator.httpclient n'est pas présent dans l'application java, nous devons l'importer explicitement dans le fichier module-info.java.

1. HttpClient : HttpClient est un conteneur pour les informations de configuration communes à plusieurs HttpRequests. Toutes les requests sont envoyées via un HttpClient. Les HttpClients sont immuables et créés à partir d'un générateur renvoyé par newBuilder(). Les générateurs de requests sont créés en appelant HttpRequest.newBuilder().
2. HttpRequest : HttpRequest représente une requête HTTP qui peut être envoyée à un serveur. Les requests HTTP sont construites à partir des générateurs HttpRequest. Les générateurs HttpRequest sont obtenus en appelant HttpRequest.newBuilder. L'URI, les en-têtes et le corps d'une requête peuvent être définis. Les corps de requête sont fournis via un HttpRequest. Objet BodyProcessor fourni aux méthodes DELETE, POST ou PUT. GET ne prend pas de corps. Une fois que tous les paramètres requis ont été définis dans le générateur, HttpRequest.Builder.build() est appelé pour renvoyer le HttpRequest.
3. HttpResponse : une HttpResponse est disponible lorsque le code d'état de la réponse et les en-têtes ont été reçus, et généralement après que le corps de la réponse a également été reçu.

Nouvelles collections

Les collecteurs ont été ajoutés dans Java 8 pour aider à accumuler les éléments d'entrée dans des conteneurs mutables tels que Map, List et Set.

Dans cet article, nous allons explorer deux nouveaux collecteurs ajoutés dans Java 9 : Collectors.filtering et Collectors.flatMapping utilisés en combinaison avec Collectors.groupingBy pour fournir des collections intelligentes d'éléments.

Collecteur filtrant

Le Collectors.filtering est similaire au Stream filter() ; il sert à filtrer les éléments d'entrée mais est utilisé pour des scénarios différents. Le filtre de Stream est utilisé dans la chaîne de flux alors que le filtrage est un Collector qui a été conçu pour être utilisé avec groupingBy.

Avec le filtre de Stream, les valeurs sont d'abord filtrées, puis regroupées. De cette façon, les valeurs qui sont filtrées disparaissent et il n'y a pas de trace de celles-ci. Si nous avons besoin d'une trace, nous devons d'abord grouper puis appliquer le filtrage, ce que fait Collectors.filtering.

Le Collectors.filtering prend une fonction pour filtrer les éléments d'entrée et un collecteur pour collecter les éléments filtrés :

```java
public void givenList_whenSatifyPredicate_thenMapValueWithOccurences()
{
    List<Integer> numbers = List.of(1, 2, 3, 5, 5);

    Map<Integer, Long> result = numbers.stream()
      .filter(val -> val > 3)
      .collect(Collectors.groupingBy(i -> i, Collectors.counting()));

    assertEquals(1, result.size());

    result = numbers.stream()
      .collect(Collectors.groupingBy(i -> i,
        Collectors.filtering(val -> val > 3, Collectors.counting())));

    assertEquals(4, result.size());
}
```

FlatMapping Collector

Le Collectors.flatMapping est similaire au Collectors.mapping mais a un objectif plus fin. Les deux collecteurs prennent une fonction et un collecteur où les éléments sont collectés mais la fonction flatMapping accepte un flux d'éléments qui est ensuite accumulé par le collecteur.

Voyons la classe modèle suivante :

```
class Blog {
    private String authorName;
    private List<String> comments;

    // constructor and getters
}
```

Collectors.flatMapping nous permet de sauter les collectes intermédiaires et d'écrire directement dans un seul conteneur qui est mis en correspondance avec le groupe défini par Collectors.groupingBy :

```java
public void givenListOfBlogs_whenAuthorName_thenMapAuthorWithComments() {
    Blog blog1 = new Blog("1", "Nice", "Very Nice");
    Blog blog2 = new Blog("2", "Disappointing", "Ok", "Could be better");
    List<Blog> blogs = List.of(blog1, blog2);

    Map<String,  List<List<String>>> authorComments1 = blogs.stream()
     .collect(Collectors.groupingBy(Blog::getAuthorName,
       Collectors.mapping(Blog::getComments, Collectors.toList())));

    assertEquals(2, authorComments1.size());
    assertEquals(2, authorComments1.get("1").get(0).size());
    assertEquals(3, authorComments1.get("2").get(0).size());

    Map<String, List<String>> authorComments2 = blogs.stream()
      .collect(Collectors.groupingBy(Blog::getAuthorName,
        Collectors.flatMapping(blog -> blog.getComments().stream(),
        Collectors.toList())));

    assertEquals(2, authorComments2.size());
    assertEquals(2, authorComments2.get("1").size());
    assertEquals(3, authorComments2.get("2").size());
}
```

Le Collectors.mapping fait correspondre tous les commentaires d'auteurs groupés au conteneur du collecteur, c'est-à-dire à la liste, alors que cette collecte intermédiaire est supprimée avec le flatMapping, qui fournit un flux direct de la liste de commentaires à mettre en correspondance avec le conteneur du collecteur.

Les Reactive Streams, les streams asynchrones

Dans cette partie, nous allons nous intéresser aux Reactive Streams de Java 9. En d'autres termes, nous pourrons utiliser la classe Flow, qui contient les principaux éléments de base pour construire une logique de traitement de stream réactif.

Pour construire un flux, nous pouvons utiliser trois abstractions principales et les composer dans une logique de traitement asynchrone.

Chaque flux doit traiter les événements qui lui sont publiés par une instance de l'éditeur ; l'éditeur dispose d'une méthode - subscribe().

Si l'un des abonnés souhaite recevoir les événements qu'il a publiés, il doit s'abonner à l'instance de l'éditeur en question.

Le récepteur des messages doit implémenter l'interface Subscriber. En général, c'est la fin de tout traitement de flux, car l'instance de cette interface n'envoie plus de messages.

Il possède quatre méthodes qui doivent être surchargées - onSubscribe(), onNext(), onError() et onComplete(). Nous les examinerons dans la section suivante.

Si nous voulons transformer un message entrant et le transmettre à l'abonné suivant, nous devons implémenter l'interface Processor. Celui-ci agit à la fois en tant qu'abonné, car il reçoit des messages, et en tant qu'éditeur, car il traite ces messages et les envoie pour un traitement ultérieur.

Publication et consommation de messages

Disons que nous voulons créer un flux simple, dans lequel nous avons un éditeur qui publie des messages, et un simple abonné qui consomme les messages au fur et à mesure qu'ils arrivent, un par un.

Créons une classe EndSubscriber. Nous devons implémenter l'interface Subscriber. Ensuite, nous allons surcharger les méthodes requises.

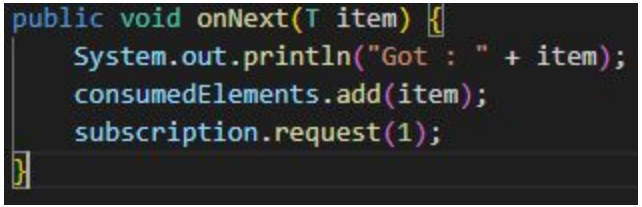
La méthode onSubscribe() est appelée avant le début du traitement. L'instance de l'Abonnement est passée comme argument. Il s'agit d'une classe qui est utilisée pour contrôler le flux de messages entre l'Abonné et l'Éditeur :

```
public class EndSubscriber<T> implements Subscriber<T> {
    private Subscription subscription;
    public List<T> consumedElements = new LinkedList<>();

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1);
    }
}
```

Nous avons également initialisé une liste vide de consumedElements qui sera utilisée dans les tests.

Maintenant, nous devons implémenter les méthodes restantes de l'interface Subscriber. La méthode principale est onNext() - elle est appelée chaque fois que l'éditeur publie un nouveau message :



Notez que lorsque nous avons lancé Subscription dans la méthode onSubscribe() et lorsque nous avons traité un message, nous devons appeler la méthode request() sur l'abonnement pour signaler que l'abonné actuel est prêt à consommer plus de messages.

Enfin, nous devons implémenter la méthode onError() - qui est appelée lorsqu'une exception est levée au cours du traitement, ainsi que la méthode onComplete() - appelée lorsque l'éditeur est fermé :

```
public void onError(Throwable t) {
    t.printStackTrace();
}

@Override
public void onComplete() {
    System.out.println("Done");
}
```

Écrivons un test pour le flux de traitement. Nous utiliserons la classe SubmissionPublisher - une construction de java.util.concurrent - qui implémente l'interface Publisher.

Nous allons soumettre N éléments à l'éditeur - que notre abonné final recevra :

```
public void whenSubscribeToIt_thenShouldConsumeAll()
  throws InterruptedException {

    // given
    SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
    EndSubscriber<String> subscriber = new EndSubscriber<>();
    publisher.subscribe(subscriber);
    List<String> items = List.of("1", "x", "2", "x", "3", "x");

    // when
    assertThat(publisher.getNumberOfSubscribers()).isEqualTo(1);
    items.forEach(publisher::submit);
    publisher.close();

    // then
     await().atMost(1000, TimeUnit.MILLISECONDS)
       .until(
         () -> assertThat(subscriber.consumedElements)
         .containsExactlyElementsOf(items)
     );
}
```

Notez que nous appelons la méthode close() sur l'instance de EndSubscriber. Elle invoquera le callback onComplete() sous chaque abonné de l'éditeur donné.

L'exécution de ce programme produira le résultat suivant :

```
Got : 1
Got : x
Got : 2
Got : x
Got : 3
Got : x
Done
```

Transformation des messages

Disons que nous voulons construire une logique similaire entre un *Publisher* et un *Subscriber*, mais aussi appliquer une certaine transformation.

Nous allons créer la classe TransformProcessor qui implémente Processor et extends SubmissionPublisher - car elle sera à la fois Publisher et Subscriber.

Nous allons passer dans une fonction qui transformera les entrées en sorties :

```java
public class TransformProcessor<T, R>
  extends SubmissionPublisher<R>
  implements Flow.Processor<T, R> {

    private Function<T, R> function;
    private Flow.Subscription subscription;

    public TransformProcessor(Function<T, R> function) {
        super();
        this.function = function;
    }

    public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1);
    }

    public void onNext(T item) {
        submit(function.apply(item));
        subscription.request(1);
    }

    public void onError(Throwable t) {
        t.printStackTrace();
    }

    public void onComplete() {
        close();
    }
}
```

Écrivons maintenant un test rapide avec un flux de traitement dans lequel l'éditeur publie des éléments de type String.

Notre TransformProcessor analysera la chaîne en tant qu'Integer - ce qui signifie qu'une conversion doit avoir lieu ici :

```
public void whenSubscribeAndTransformElements_thenShouldConsumeAll()
  throws InterruptedException {

    // given
    SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
    TransformProcessor<String, Integer> transformProcessor
      = new TransformProcessor<>(Integer::parseInt);
    EndSubscriber<Integer> subscriber = new EndSubscriber<>();
    List<String> items = List.of("1", "2", "3");
    List<Integer> expectedResult = List.of(1, 2, 3);

    // when
    publisher.subscribe(transformProcessor);
    transformProcessor.subscribe(subscriber);
    items.forEach(publisher::submit);
    publisher.close();

    // then
     await().atMost(1000, TimeUnit.MILLISECONDS)
       .until(() ->
         assertThat(subscriber.consumedElements)
         .containsExactlyElementsOf(expectedResult)
     );
}
```

Notez que l'appel de la méthode close() sur le Publisher de base provoquera l'appel de la méthode onComplete() sur le TransformProcessor.

Gardez à l'esprit que tous les éditeurs de la chaîne de traitement doivent être fermés de cette manière.

Contrôle de la demande de messages à l'aide de l'abonnement

Disons que nous voulons consommer uniquement le premier élément de l'abonnement, appliquer une certaine logique et terminer le traitement. Nous pouvons utiliser la méthode request() pour y parvenir.

Modifions notre EndSubscriber pour ne consommer que le nombre N de messages. Nous allons passer ce nombre comme argument du constructeur howMuchMessagesConsume :

```java
public class EndSubscriber<T> implements Subscriber<T> {

    private AtomicInteger howMuchMessagesConsume;
    private Subscription subscription;
    public List<T> consumedElements = new LinkedList<>();

    public EndSubscriber(Integer howMuchMessagesConsume) {
        this.howMuchMessagesConsume
          = new AtomicInteger(howMuchMessagesConsume);
    }

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1);
    }

    @Override
    public void onNext(T item) {
        howMuchMessagesConsume.decrementAndGet();
        System.out.println("Got : " + item);
        consumedElements.add(item);
        if (howMuchMessagesConsume.get() > 0) {
            subscription.request(1);
        }
    }
    //...

}
```

Nous pouvons demander des éléments aussi longtemps que nous le souhaitons.

Écrivons un test dans lequel nous voulons seulement consommer un élément de l'abonnement donné :

```
public void whenRequestForOnlyOneElement_thenShouldConsumeOne()
  throws InterruptedException {
    // given
    SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
    EndSubscriber<String> subscriber = new EndSubscriber<>(1);
    publisher.subscribe(subscriber);
    List<String> items = List.of("1", "x", "2", "x", "3", "x");
    List<String> expected = List.of("1");

    // when
    assertThat(publisher.getNumberOfSubscribers()).isEqualTo(1);
    items.forEach(publisher::submit);
    publisher.close();

    // then
    await().atMost(1000, TimeUnit.MILLISECONDS)
      .until(() ->
        assertThat(subscriber.consumedElements)
       .containsExactlyElementsOf(expected)
    );
}
```

Bien que *publisher* publie six éléments, notre *EndSubscriber* n'en consommera qu'un seul car il signale la demande de traitement de ce seul élément.

En utilisant la méthode request() sur *Subscription*, nous pouvons mettre en œuvre un mécanisme de contre-pression plus sophistiqué pour contrôler la vitesse de consommation du message.

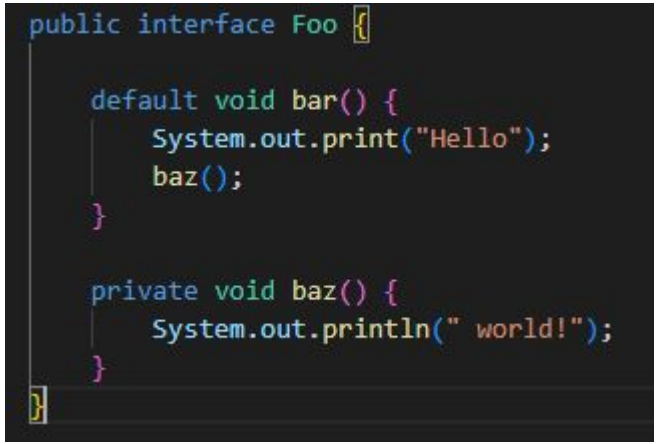
Méthodes privées d'interface

Depuis Java 9, des méthodes privées peuvent être ajoutées aux interfaces en Java. Dans ce court tutoriel, nous allons voir comment définir ces méthodes et quels sont leurs avantages.

Définir des méthodes privées dans les interfaces

Les méthodes privées peuvent être implémentées de manière statique ou non statique. Cela signifie que dans une interface, nous sommes en mesure de créer des méthodes privées pour encapsuler du code à partir de signatures de méthodes publiques statiques et par défaut.

Tout d'abord, voyons comment nous pouvons utiliser des méthodes privées à partir de méthodes d'interface par défaut :



```java
public interface Foo {

    default void bar() {
        System.out.print("Hello");
        baz();
    }

    private void baz() {
        System.out.println(" world!");
    }
}
```

bar() est capable d'utiliser la méthode privée baz() en l'appelant depuis sa méthode par défaut.

Ensuite, ajoutons une méthode privée définie de manière statique à notre interface Foo :

```
public interface Foo {

    static void buzz() {
        System.out.print("Hello");
        staticBaz();
    }

    private static void staticBaz() {
        System.out.println(" static world!");
    }
}
```

Au sein de l'interface, d'autres méthodes définies de manière statique peuvent utiliser ces méthodes statiques privées.

Enfin, appelons les méthodes par défaut et statiques définies à partir d'une classe concrète :

```
public class CustomFoo implements Foo {

    public static void main(String... args) {
        Foo customFoo = new CustomFoo();
        customFoo.bar();
        Foo.buzz();
    }
}
```

La sortie est la chaîne "Hello world !" de l'appel à la méthode bar() et "Hello static world !" de l'appel à la méthode buzz().

Avantages des méthodes privées dans les interfaces

Parlons des avantages des méthodes privées maintenant que nous les avons définies.

Comme nous l'avons vu dans la section précédente, les interfaces peuvent utiliser des méthodes privées pour cacher les détails de l'implémentation aux classes qui implémentent l'interface. Par conséquent, l'un des principaux avantages de leur présence dans les interfaces est l'encapsulation.

Un autre avantage est (comme avec les méthodes privées en général) qu'il y a moins de duplication et plus de code réutilisable ajouté aux interfaces pour les méthodes ayant une fonctionnalité similaire.

L'API CompletableFuture

Java 9 a apporté quelques modifications à la classe CompletableFuture. Ces changements ont été introduits dans le cadre de la JEP 266 afin de répondre aux plaintes et suggestions courantes depuis son introduction dans le JDK 8, plus précisément, la prise en charge des délais et des temporisations, une meilleure prise en charge du sous-classement et quelques méthodes utilitaires.

Du point de vue du code, l'API comprend huit nouvelles méthodes et cinq nouvelles méthodes statiques. Pour permettre ces ajouts, environ 1500 lignes de code sur 2400 ont été modifiées (selon Open JDK).

Instance API Additions

Comme mentionné, l'API d'instance est livré avec huit nouveaux ajouts, ils sont :

1. Executor defaultExecutor()
2. CompletableFuture<U> newIncompleteFuture()
3. CompletableFuture<T> copy()
4. CompletionStage<T> minimalCompletionStage()
5. CompletableFuture<T> completeAsync(Supplier<? extends T> supplier, Executor executor)
6. CompletableFuture<T> completeAsync(Supplier<? extends T> supplier)
7. CompletableFuture<T> orTimeout(long timeout, TimeUnit unit)
8. CompletableFuture<T> completeOnTimeout(T value, long timeout, TimeUnit unit)

Méthode defaultExecutor()

Executor defaultExecutor()

Renvoie l'exécuteur par défaut utilisé pour les méthodes asynchrones qui ne spécifient pas d'exécuteur.

```
new CompletableFuture().defaultExecutor()
```

Ceci peut être surchargé par les sous-classes qui renvoient un exécuteur fournissant, au moins, un thread indépendant.

Méthode new Incomplète Future()

CompletableFuture<U> newIncompleteFuture()

Le newIncompleteFuture, également connu sous le nom de "constructeur virtuel", est utilisé pour obtenir une nouvelle instance de *completable future* du même type.

```
new CompletableFuture().newIncompleteFuture()
```

Cette méthode est particulièrement utile lors de la sous-classification de CompletableFuture, principalement parce qu'elle est utilisée en interne dans presque toutes les méthodes retournant un nouveau CompletionStage, permettant aux sous-classes de contrôler quel sous-type est retourné par ces méthodes.

Méthode *copy()*

CompletableFuture<T> copy()

Cette méthode renvoie un nouveau CompletableFuture qui :

- Lorsque celui-ci se termine normalement, le nouveau se termine aussi normalement.
- Lorsque le processus se termine exceptionnellement avec l'exception X, le nouveau processus se termine également exceptionnellement avec une CompletionException dont la cause est X.

```
new CompletableFuture().copy()
```

Cette méthode peut être utile comme une forme de "copie défensive", pour empêcher les clients de compléter, tout en étant capable d'organiser des actions dépendantes sur une instance spécifique de CompletableFuture.

Méthode minimalCompletionStage()

CompletionStage<T> minimalCompletionStage()

Cette méthode renvoie une nouvelle CompletionStage qui se comporte exactement de la même manière que celle décrite par la méthode de copie. Toutefois, cette nouvelle instance lève l'exception UnsupportedOperationException à chaque tentative de récupération ou de définition de la valeur résolue.

```
new CompletableFuture().minimalCompletionStage()
```

Un nouveau CompletableFuture avec toutes les méthodes disponibles peut être récupéré en utilisant la méthode toCompletableFuture disponible sur l'API CompletionStage.

Méthode *completeAsync()*

La méthode completeAsync doit être utilisée pour compléter le CompletableFuture de manière asynchrone en utilisant la valeur donnée par le *Supplier* fourni.

```
CompletableFuture<T> completeAsync(Supplier<? extends T> supplier, Executor executor)
CompletableFuture<T> completeAsync(Supplier<? extends T> supplier)
```

La différence entre ces deux méthodes surchargées est l'existence du deuxième argument, où l'exécuteur exécutant la tâche peut être spécifié. Si aucun n'est fourni, l'exécuteur par défaut (renvoyé par la méthode defaultExecutor) sera utilisé.

Méthode *orTimeout()*

```
new CompletableFuture().orTimeout(1, TimeUnit.SECONDS)
```

Résout l'exception CompletableFuture avec TimeoutException, sauf si elle est terminée avant le délai spécifié.

Méthode completeOnTimeout()

```
new CompletableFuture().completeOnTimeout(value,  1, TimeUnit.SECONDS)
```

Termine le CompletableFuture normalement avec la valeur spécifiée, à moins qu'il ne soit terminé avant le délai spécifié.

Travaux pratique

En utilisant le modèle et la classe réactive Streams, faite un programme qui contient un Publisher et un Subscriber, le Publisher enverra au Subscriber un élément d’une list, les un après les autres. Ils seront affichés à leur arrivée.

Apports de la JDK 10

- Inférence de type de variables locales
- Améliorations dans les API existantes

Inférence de type de variables locales

L'une des améliorations les plus visibles du JDK 10 est l'inférence de type des variables locales avec initialiseurs.

Ce cours fournit les détails de cette fonctionnalité avec des exemples.

Introduction

Jusqu'à Java 9, nous devions mentionner explicitement le type de la variable locale et nous assurer qu'il était compatible avec l'initialiseur:

```
String message = "Good bye, Java 9";
```

En Java 10, voici comment on peut déclarer une variable locale :

```
public void whenVarInitWithString thenGetStringTypeVar() {
    var message = "Hello, Java 10";
    assertTrue(message instanceof String);
}
```

Nous ne fournissons pas le type de données du message. Au lieu de cela, nous marquons le message comme une variable, et le compilateur déduit le type du message à partir du type de l'initialisateur présent sur le côté droit.

Dans l'exemple ci-dessus, le type du message serait String.

Notez que cette fonctionnalité est disponible uniquement pour les variables locales avec l'initialisateur. Elle ne peut pas être utilisée pour les variables membres, les paramètres de méthode, les types de retour, etc. L'initialisateur est nécessaire car sans lui, le compilateur ne pourra pas déduire le type.

Cette amélioration permet de réduire le code passe-partout, par exemple :

```
Map<Integer, String> map = new HashMap<>();
```

Ceci peut maintenant être réécrit comme :

```
var idToNameMap = new HashMap<Integer, String>();
```

Cela permet également de se concentrer sur le nom de la variable plutôt que sur son type.

Il convient également de noter que var n'est pas un mot-clé, ce qui garantit la rétrocompatibilité pour les programmes utilisant var, par exemple, comme nom de fonction ou de variable. var est un nom de type réservé, tout comme int.

Enfin, notez que l'utilisation de var n'entraîne pas de surcharge à l'exécution et ne fait pas de Java un langage dynamiquement typé. Le type de la variable est toujours déduit au moment de la compilation et ne peut être modifié ultérieurement.

Utilisation illégale de var

Comme mentionné précédemment, var ne fonctionnera pas sans l'initialisation : `var n; // error: cannot use 'var' on variable without initializer`

Il ne fonctionnerait pas non plus s'il était initialisé avec null : `var emptyList = null; // error: variable initializer is 'null'`

Cela ne fonctionnera pas pour les variables non locales : `public var = "hello"; // error: 'var' is not allowed here`

L'expression lambda nécessite un type cible explicite, et donc var ne peut pas être utilisé : `var p = (String s) -> s.length() > 10; // error: lambda expression needs an explicit target-type`

Il en va de même pour l'initialiseur de tableau : `var arr = { 1, 2, 3 }; // error: array initializer needs an explicit target-type`

Directives pour l'utilisation de var

Il existe des situations où var peut être utilisé légalement, mais peut ne pas être une bonne idée de le faire.

Par exemple, dans des situations où le code pourrait devenir moins lisible : `var result = obj.prcoess();`

Ici, bien qu'il s'agisse d'une utilisation légale de var, il devient difficile de comprendre le type renvoyé par le process()ce qui rend le code moins lisible.

java.net a un article dédié sur les directives de style pour l'inférence de type des variables locales en Java qui explique comment nous devrions faire preuve de jugement lorsque nous utilisons cette fonctionnalité.

Une autre situation où il est préférable d'éviter var est dans les flux avec un long pipeline :

```
var x = emp.getProjects.stream().findFirst().map(String::length).orElse(0);
```

L'utilisation de var peut également donner un résultat inattendu.

Par exemple, si nous l'utilisons avec l'opérateur diamant introduit dans Java 7 : `var empList = new ArrayList<>();`

Le type de empList sera ArrayList<Object>et non List<Object>. Si nous voulons que ce soit ArrayList<Employee>, nous devrons être explicites : `var empList = new ArrayList<Employee>();`

L'utilisation de var avec des types non-dénotables pouvait provoquer une erreur inattendue.

Par exemple, si nous utilisons var avec l'instance de classe anonyme :

```
public void whenVarInitWithAnonymous_thenGetAnonymousType()
{
    var obj = new Object() {};
    assertFalse(obj.getClass().equals(Object.class));
}
```

Maintenant, si nous essayons d'assigner un autre objet à obj, nous obtiendrons une erreur de compilation :

```
obj = new Object(); // error: Object cannot be converted to <anonymous Object>
```

Ceci est dû au fait que le type inféré de obj n'est pas Object.

Améliorations dans les API existantes

- copyOf()
- toUnmodifiable*()
- Optional*.orElseThrow()

copyOf()

java.util.List, java.util.Map et java.util.Set ont chacune une nouvelle méthode statique copyOf(Collection).

Elle renvoie la copie non modifiable de la collection donnée :

```
@Test(expected = UnsupportedOperationException.class)
public void whenModifyCopyOfList_thenThrowsException() {
    List<Integer> copyList = List.copyOf(someIntList);
    copyList.add(4);
}
```

Toute tentative de modification d'une telle collection entraînerait une exception de type java.lang.UnsupportedOperationExceptionruntime.

toUnmodifiable*()

java.util.stream.Collectors obtiennent des méthodes supplémentaires pour collecter un *Stream* dans une *List*, une *Map*ou un *Set*:

```
@Test(expected = UnsupportedOperationException.class)
public void whenModifyToUnmodifiableList_thenThrowsException() {
    List<Integer> evenList = someIntList.stream()
      .filter(i -> i % 2 == 0)
      .collect(Collectors.toUnmodifiableList());
    evenList.add(4);
}
```

Toute tentative de modification d'une telle collection entraînerait une exception de type java.lang.UnsupportedOperationExceptionruntime.

Optional*.orElseThrow()

java.util.Optional, java.util.OptionalDouble, java.util.OptionalIntand java.util.OptionalLongeach a obtenu une nouvelle méthode orElseThrow() qui ne prend aucun argument et lance NoSuchElementExceptionsi aucune valeur n'est présente :

```
@Test
public void whenListContainsInteger_OrElseThrowReturnsInteger() {
    Integer firstEven = someIntList.stream()
      .filter(i -> i % 2 == 0)
      .findFirst()
      .orElseThrow();
    is(firstEven).equals(Integer.valueOf(2));
}
```

Elle est synonyme de la méthode get() et constitue désormais l'alternative préférée à la méthode existante get().

Apports de la JDK 11

- Inférences de type pour les lambda expressions
- Simplification des "run" de programmes
- Améliorations dans les classes String, StringBuilder et StringBuffer
- Améliorations des Predicate Java 8
- Classes imbriquées et visibilité des attributs
- Suppression des modules JEE, JavaJX et CORBA

Inférences de type pour les lambda expressions

Une des principales fonctionnalités introduites dans Java 10 était l'inférence de type des variables locales. Elle permettait l'utilisation de var comme type de la variable locale au lieu du type réel. Le compilateur déduit le type sur la base de la valeur affectée à la variable.

Cependant, nous ne pouvions pas utiliser cette fonctionnalité avec les paramètres lambda. Par exemple, considérons le lambda suivant. Ici, nous spécifions explicitement les types des paramètres : `(String s1, String s2) -> s1 + s2`

Nous pourrions sauter les types de paramètres et réécrire le lambda comme : `(s1, s2) -> s1 + s2`

Même Java 8 le permettait. L'extension logique de ceci dans Java 10 serait : `(var s1, var s2) -> s1 + s2`

Améliorations dans les classes Strings

Java 11 ajoute quelques nouvelles méthodes à la classe String : isBlank, lines, strip, stripLeading, stripTrailing et repeat.

Voyons comment nous pouvons utiliser ces nouvelles méthodes pour extraire les lignes non vides d'une chaîne de caractères à plusieurs lignes :

```
String multilineString = "chris helps \n \n developers \n explore Java.";
List<String> lines = multilineString.lines()
  .filter(line -> !line.isBlank())
  .map(String::strip)
  .collect(Collectors.toList());
assertThat(lines).containsExactly("chris helps", "developers", "explore Java.");
```

Ces méthodes permettent de réduire la quantité de texte passe-partout nécessaire à la manipulation des chaînes de caractères et nous évitent d'avoir à importer des bibliothèques.

Dans le cas des méthodes de dépouillement, elles offrent une fonctionnalité similaire à la méthode de découpage plus familière, mais avec un contrôle plus fin et un support Unicode.

java.lang.StringBuffer / java.lang.StringBuilder

Ces deux classes ont désormais accès à une nouvelle méthode compareTo() qui prend en argument un StringBuffer/StringBuilder et retourne un int. La logique de comparaison suit le même ordre lexicographique que pour la nouvelle méthode de la classe CharSequence.

Simplification des "run" de programmes

Un changement majeur dans cette version est que nous n'avons plus besoin de compiler les fichiers sources Java avec javac explicitement :

```
$ javac HelloWorld.java
```

```
$ java HelloWorld
```

```
Hello Java 8!
```

Au lieu de cela, nous pouvons exécuter directement le fichier en utilisant la commande java :

```
$ java HelloWorld.java
```

```
Hello Java 11!
```

Améliorations des Predicate Java 8

Tout d'abord, voyons comment nous avons réussi à nier un prédicat avant Java 11.

Pour commencer, créons une classe Personne avec un champ âge et une méthode isAdult() :

```
public class Person {
    private static final int ADULT_AGE = 18;

    private int age;

    public Person(int age) {
        this.age = age;
    }

    public boolean isAdult() {
        return age >= ADULT_AGE;
    }
}
```

Imaginons maintenant que nous ayons une liste de personnes :

```
List<Person> people = Arrays.asList(new Person(1),  new Person(18), new Person(2));
```

Et nous voulons récupérer tous les adultes. Pour réaliser cela en Java 8, nous pouvons :

```
people.stream().filter(Person::isAdult).collect(Collectors.toList());
```

Cependant, que se passe-t-il si nous voulons récupérer les personnes non adultes à la place ? Dans ce cas, nous devons nier le prédicat :

```
people.stream().filter(person -> !person.isAdult()).collect(Collectors.toList());
```

Malheureusement, nous sommes obligés de laisser tomber la référence à la méthode, même si nous la trouvons plus facile à lire.

Une solution de contournement possible consiste à créer une méthode isNotAdult() sur la classe Person, puis à utiliser une référence à cette méthode :

```
people.stream().filter(Person::isNotAdult).collect(Collectors.toList());
```

Mais peut-être ne voulons-nous pas ajouter cette méthode à notre API, ou peut-être ne pouvons-nous pas le faire parce que la classe n'est pas la nôtre. C'est alors que Java 11 arrive avec la méthode Predicate.not(), comme nous allons le voir dans la section suivante.

La méthode Predicate.not()

La méthode statique Predicate.not() a été ajoutée à Java 11 afin d'annuler un Predicate existant.

Reprenons notre exemple précédent et voyons ce que cela signifie.

Au lieu d'utiliser un lambda ou de créer une nouvelle méthode sur la classe Person, nous pouvons simplement utiliser cette nouvelle méthode : `people.stream().filter(Predicate.not(Person::isAdult)).collect(Collectors.toList());`

De cette façon, nous n'avons pas à modifier notre API et nous pouvons toujours compter sur la lisibilité des références de méthodes.

Nous pouvons rendre cela encore plus clair avec une importation statique :

```
people.stream().filter(not(Person::isAdult)).collect(Collectors.toList());
```

Gestion de la visibilité des attributs des classes imbriquées

Java autorise la déclaration de plusieurs classes dans un seul fichier source, telles que les classes imbriquées (Nested Class). Du point de vue de l'utilisateur, elles sont toutefois généralement considérées comme appartenant à la "même classe".

Et, par conséquent, les utilisateurs s'attendent à ce qu'elles partagent un régime d'accès commun aux attributs ou méthodes de la classe mère.

Pour préserver ces attentes, les compilateurs doivent élargir l'accès des attributs privés aux classes du même package en ajoutant des ponts d'accès. Une invocation d'un membre privé est compilée en un appel d'une méthode (getter) générée par le compilateur dans la classe cible, qui à son tour accède au membre privé prévu.

Par exemple, dans le cas d'une classe NestedClass, imbriquée à l'intérieur d'une classe NestingClass, qui a besoin d'accéder à un des attributs privés de la classe hôte :

```
public class NestingClass {
    private int nestingInt;
    class NestedClass {
        public void printNestingInt() {
            System.out.println("Nesting Int = " + nestingInt);
        }
    }
}
```

Le compilateur découple les deux classes et crée une méthode d'accès publique à nestingInt utilisée par la classe NestedClass :

```
public class CompiledNestingClass {
    private int nestingInt;
    public int proxyToGetNestingInt() {
        return nestingInt;
    }
}
class CompiledNestedClass {
    CompiledNestingClass nestingClass;
    public void printNestingInt() {
        System.out.println("Nesting Int = " + nestingClass.proxyToGetNestingInt());
    }
}
```

Ces ponts subvertissent l'encapsulation (*private* ne revêt plus exactement le même sens) et peuvent dérouter les utilisateurs et les outils.

Une notion formelle d'un groupe de fichiers de classe formant un nid (ou nest), où les partenaires de nid partagent un mécanisme de contrôle d'accès commun, permet d'obtenir directement le résultat souhaité de manière plus simple, plus sécurisée et plus transparente.

Pour relier facilement classes imbriquées et hôtes, en Java 11 deux nouveaux attributs ont été ajoutés aux classes : NestHost (hôte du nid) et NestMembers (membres du nid).

On a aussi l'ajout de 3 méthodes à java.lang.Class :

- Class getNestHost()
- Class[] getNestMembers()
- boolean isNestmateof(Class)

Removed and Deprecated Modules

À mesure que Java évolue, nous ne pouvons plus utiliser aucune de ses fonctionnalités supprimées et devons cesser d'utiliser les fonctionnalités dépréciées. Jetons un coup d'œil rapide aux plus importantes d'entre elles.

Java EE and CORBA

Des versions autonomes des technologies Java EE sont disponibles sur des sites tiers ; il n'est donc pas nécessaire de les inclure dans Java SE.

Java 9 avait déjà déprécié certains modules Java EE et CORBA. Dans la version 11, il les a maintenant complètement supprimés :

- Java API for XML-Based Web Service*s (java.xml.ws*)
- Java Architecture for XML Binding *(java.xml.bind*)
- JavaBeans Activation Framework *(java.activation*)
- Common Annotations *(java.xml.ws.annotation*)
- Common Object Request Broker Architecture *(java.corba)*
- JavaTransaction API *(java.transaction*)

JMC and JavaFX

JDK Mission Control (JMC) n'est plus inclus dans le JDK. Une version autonome de JMC est désormais disponible en téléchargement séparé.

Il en va de même pour les modules JavaFX ; JavaFX sera disponible sous la forme d'un ensemble distinct de modules en dehors du JDK.

Deprecated Modules

En outre, Java 11 a déprécié les modules suivants :

le moteur JavaScript Nashorn, y compris l'outil JJS

Schéma de compression Pack200 pour les fichiers JAR

Travaux pratique

exercice 1:

Créer une classe player qui contient une string “name”, un int “age”, avec des getter et des setter pour chaque objet. Avec une méthode isAdult() renvoyant un booléen. La classe

Faite un string de Player, contenant plusieurs players, certains adultes, d’autres enfants.

Utiliser les streams et predicate pour filtrer le stream en fonction de la methode isadult;

Nous utiliserons les vars pour stocker deux List de player, une adulte et une enfant.

Nous afficherons ensuite le nom et l'âge de chaque list en utilisant un foreach et une lambda.

Nous compilerons bien entendu avec la nouvelle méthode.

Apports de la JDK 12 et 13

- Le switch comme instruction
- Les blocs de texte
- Le mot-clé "yield"
- Les outils apportés par la JDK 12

Switch Expressions

La fonctionnalité la plus populaire introduite dans Java 12 est les expressions Switch.

À titre de démonstration, comparons l'ancienne et la nouvelle expression Switch. Nous les utiliserons pour distinguer les jours ouvrables des jours de week-end en fonction de l'enum DayOfWeek de l'instance LocalDate.

Tout d'abord, regardons l'ancienne syntaxe :

```
DayOfWeek dayOfWeek = LocalDate.now().getDayOfWeek();
String typeOfDay = "";
switch (dayOfWeek) {
    case MONDAY:
    case TUESDAY:
    case WEDNESDAY:
    case THURSDAY:
    case FRIDAY:
        typeOfDay = "Working Day";
        break;
    case SATURDAY:
    case SUNDAY:
        typeOfDay = "Day Off";
}
```

Et maintenant, voyons la même logique avec les expressions de commutation :

```
case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> {
    // more logic
    System.out.println("Working Day")
}
```

Les nouvelles instructions switch ne sont pas seulement plus compactes et plus lisibles. Elles suppriment également le besoin d'instructions break. L'exécution du code ne sera pas interrompue après la première correspondance.

Une autre différence notable est que nous pouvons assigner une instruction switch directement à la variable. Ce n'était pas possible auparavant.

Il est également possible d'exécuter du code dans des expressions switch sans renvoyer de valeur :

```
switch (dayOfWeek) {
    case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> System.out.println("Working Day");
    case SATURDAY, SUNDAY -> System.out.println("Day Off");
}
```

Les logiques plus complexes doivent être entourées d'accolades :

```
case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> {
    // more logic
    System.out.println("Working Day")
}
```

Notez que nous pouvons choisir entre l'ancienne et la nouvelle syntaxe. Les expressions de commutation de Java 12 ne sont qu'une extension, et non un remplacement.

Les blocs de texte

La deuxième fonction de prévisualisation concerne les blocs de texte pour les chaînes de caractères de plusieurs lignes telles que JSON, XML, HTML, etc. intégrés.

Auparavant, pour intégrer JSON dans notre code, nous le déclarions comme un littéral String :

```
String JSON_STRING  = "{\r\n" + "\"name\" : \"chris\",\r\n" + "\"website\" :
\"https://www.%s.com/\"\r\n" + "}";
```

Maintenant, écrivons le même JSON en utilisant les blocs de texte String :

```
String TEXT_BLOCK_JSON = """
{
    "name" : "chris",
    "website" : "https://www.%s.com/"
}
""";
```

Comme on peut le constater, il n'est pas nécessaire d'échapper aux guillemets doubles ou d'ajouter un retour chariot. En utilisant des blocs de texte, le JSON intégré est beaucoup plus simple à écrire et plus facile à lire et à maintenir.

En outre, toutes les fonctions de String sont disponibles :

```
public void whenTextBlocks_thenStringOperationsWorkSame() {
    assertThat(TEXT_BLOCK_JSON.contains("Baeldung")).isTrue();
    assertThat(TEXT_BLOCK_JSON.indexOf("www")).isGreaterThan(0);
    assertThat(TEXT_BLOCK_JSON.length()).isGreaterThan(0);
}
```

En outre, java.lang.String dispose désormais de trois nouvelles méthodes pour manipuler les blocs de texte :

- stripIndent() - imite le compilateur pour supprimer les espaces blancs superflus.
- translateEscapes() - Traduit les séquences d'échappement telles que "\\\t" en "\t".
- formatted() - fonctionne de la même manière que String::format, mais pour les blocs de texte.

Jetons un coup d'œil rapide à un exemple de String::formatted :

```
assertThat(TEXT_BLOCK_JSON.formatted("baeldung").contains("www.baeldung.com")).isTrue();
assertThat(String.format(JSON_STRING,"baeldung").contains("www.baeldung.com")).isTrue();
```

Étant donné que les blocs de texte sont une fonction de prévisualisation et qu'ils peuvent être supprimés dans une version ultérieure, ces nouvelles méthodes sont marquées pour la dépréciation.700

Yield

La syntaxe des breaks de Java 12 n'est plus compilée dans Java 13, elle utilise plutôt yield.

```
private static int getValueViaBreak(String mode) {
int result = switch (mode) {
    case "a":
    case "b":
        break 1;
    case "c":
        break 2;
    case "d":
    case "e":
    case "f":
        break 3;
    default:
        break -1;
};
return result;
```

En Java 13, nous pouvons utiliser yield pour retourner une valeur

```
private static int getValueViaYield(String mode) {
int result = switch (mode) {
    case "a", "b":
        yield 1;
    case "c":
        yield 2;
    case "d", "e", "f":
        // do something here...
        System.out.println("Supports multi line block!");
        yield 3;
    default:
        yield -1;
};
return result;
```

Les outils apportés par la JDK 12

Ajout d'un nouvel algorithme de garbage collection (GC) nommé Shenandoah qui réduit les temps de pause GC en effectuant le travail d'évacuation simultanément avec les threads Java en cours d'exécution. Les temps de pause avec Shenandoah sont indépendants de la taille du tas, ce qui signifie que vous aurez les mêmes temps de pause cohérents que votre tas fasse 200 Mo ou 200 Go.

Travaux pratique

Créer une classe Agenda, qui possède une variable Month contenant un int représentant un mois.

nous allons coder une méthode toSaison() qui renvoi un booléen, si le mois n’est pas compris dans les 12 mois de l'année la méthode renverra false, sinon elle renverra true.

La méthode affichera quelle est la saison de Month, si le mois est incorrect nous afficherons une erreur.

Vous devrez bien sûr utiliser les notions vues précédemment.

Apports de la JDK 14

- Changements dans les switch
- Clarifications du NullPointerException
- Le Live Monitoring
- Changements dans le "instanceof"
- Nouveaux outils associés à la JDK 14

Le Live Monitoring

JDK Flight Recorder (JFR) prend désormais en charge la surveillance continue d'une application Java en permettant la consommation dynamique des événements à l'aide d'une nouvelle API située dans le paquet jdk.jfr.consumer. La fonctionnalité est toujours activée lors de l'utilisation de JFR, ce qui signifie que les données enregistrées jusqu'à la dernière seconde sont disponibles pour la consommation en cours de processus et hors processus.

Clarifications du NullPointerException

Auparavant, la trace de la pile d'une exception NullPointerException n'avait pas grand-chose à dire, si ce n'est qu'une valeur était nulle à une ligne donnée d'un fichier donné.

Bien qu'utile, cette information ne faisait que suggérer une ligne à déboguer au lieu de brosser un tableau complet pour qu'un développeur puisse le comprendre, simplement en regardant le journal.

Aujourd'hui, Java a facilité les choses en ajoutant la possibilité d'indiquer ce qui était exactement nul dans une ligne de code donnée.

Par exemple, considérez ce simple extrait :

```
int[] arr = null;
arr[0] = 1;
```

Auparavant, en exécutant ce code, le journal indiquait :

```
Exception in thread "main" java.lang.NullPointerException
at com.baeldung.MyClass.main(MyClass.java:27)
```

Mais maintenant, avec le même scénario, le journal pourrait dire :

```
java.lang.NullPointerException: Cannot store to int array because "a" is null
```

Comme nous pouvons le voir, nous savons maintenant précisément quelle variable a causé l'exception.

Changements dans les switch

Elles ont été introduites pour la première fois en tant que fonction de prévisualisation dans le JDK 12, et même dans Java 13, elles sont restées des fonctions de prévisualisation uniquement. Mais aujourd'hui, les expressions de commutation ont été normalisées de sorte qu'elles font partie intégrante du kit de développement.

Cela signifie concrètement que cette fonctionnalité peut désormais être utilisée dans le code de production, et non plus seulement en mode preview pour être expérimentée par les développeurs.

À titre d'exemple, considérons un scénario dans lequel nous désignons les jours de la semaine comme étant soit un jour de semaine, soit un week-end.

Avant cette amélioration, nous l'aurions écrit comme suit :

```
boolean isTodayHoliday;
switch (day) {
    case "MONDAY":
    case "TUESDAY":
    case "WEDNESDAY":
    case "THURSDAY":
    case "FRIDAY":
        isTodayHoliday = false;
        break;
    case "SATURDAY":
    case "SUNDAY":
        isTodayHoliday = true;
        break;
    default:
        throw new IllegalArgumentException("What's a " + day);
}
```

Avec les expressions switch, nous pouvons écrire la même chose de manière plus succincte :

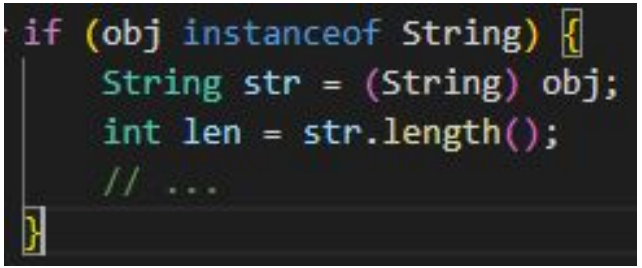
```
boolean isTodayHoliday = switch (day) {
    case "MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY" -> false;
    case "SATURDAY", "SUNDAY" -> true;
    default -> throw new IllegalArgumentException("What's a " + day);
};
```

Changements dans le "instanceof"

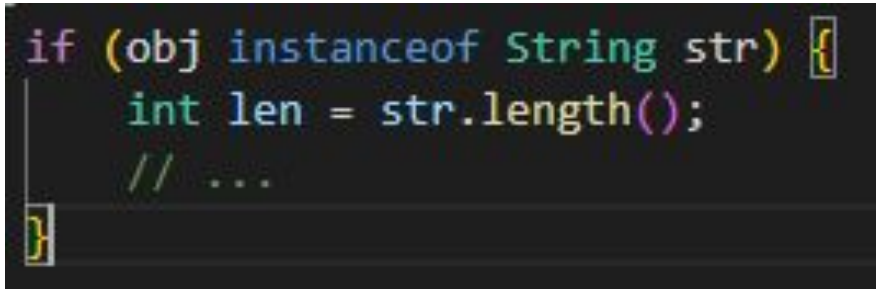
Le JDK 14 a introduit la correspondance de motifs pour instanceof dans le but d'éliminer le code passe-partout et de faciliter un peu plus la vie du développeur.

Pour comprendre cela, prenons un exemple simple.

Avant cette fonctionnalité, nous avons écrit :



Maintenant, nous n'avons pas besoin d'autant de code pour commencer à utiliser obj comme String :



Dans les prochaines versions, Java va proposer le filtrage pour d'autres constructions telles que les commutateurs.

Nouveaux outils associés à la JDK 14

- *[Incubator] Packaging Tool (JEP343) :* Le nouvel outil permettant de générer un installeur CLI pour n'importe quel OS (deb/rpm pour Linux, pkg/dmg pour MacOS, et exe/msi pour Windows)

Apports de la JDK 15

- Les classes "Sealed"
- Fonctionnalités dépréciées

Sealed Classes

Actuellement, Java ne fournit aucun contrôle fin sur l'héritage. Les modificateurs d'accès tels que public, protected, private, ainsi que le package-private par défaut, fournissent un contrôle à très gros grain.

À cette fin, l'objectif des classes scellées est de permettre aux classes individuelles de déclarer quels types peuvent être utilisés comme sous-types. Ceci s'applique également aux interfaces et à la détermination des types qui peuvent les implémenter.

Les classes scellées impliquent deux nouveaux mots-clés - scellé et permis :

```
public abstract sealed class Person
    permits Employee, Manager {

    //...
}
```

Dans cet exemple, nous avons déclaré une classe abstraite appelée Person. Nous avons également spécifié que les seules classes qui peuvent l'étendre sont Employee et Manager. L'extension de la classe scellée se fait comme aujourd'hui en Java, à l'aide du mot-clé extends :

```
public final class Employee extends Person {
}

public non-sealed class Manager extends Person {
}
```

Il est important de noter que toute classe qui étend une classe scellée doit elle-même être déclarée scellée, non scellée ou finale. Cela garantit que la hiérarchie des classes reste finie et connue par le compilateur.

Cette hiérarchie finie et exhaustive est l'un des grands avantages de l'utilisation des classes scellées. Voyons un exemple de ce principe en action :

```
if (person instanceof Employee) {
    return ((Employee) person).getEmployeeId();
}
else if (person instanceof Manager) {
    return ((Manager) person).getSupervisorId();
}
```

Sans une classe scellée, le compilateur ne peut pas raisonnablement déterminer que toutes les sous-classes possibles sont couvertes par nos instructions if-else. Sans une clause else à la fin, le compilateur émettrait probablement un avertissement indiquant que notre logique ne couvre pas tous les cas.

Fonctionnalités dépréciées

- Le mécanisme d'activation RMI a été déprécié et pourrait être supprimé dans une future version de la plate-forme. L'activation RMI est une partie obsolète de RMI qui est facultative depuis Java 8. Elle permet de démarrer ("activer") les JVM du serveur RMI à la réception d'une demande d'un client, au lieu de demander aux JVM du serveur RMI de fonctionner en permanence. D'autres parties de RMI ne sont pas dépréciées.

- Après une mise à niveau du SDK macOS utilisé pour construire le JDK, le comportement des propriétés apple.awt.brushMetalLook et textured Swing a changé. Lorsque ces propriétés sont définies, le titre du cadre est toujours visible. Il est recommandé de définir la propriété apple.awt.transparentTitleBar sur true pour que le titre du cadre redevienne invisible. La propriété apple.awt.fullWindowContent peut également être utilisée. Veuillez noter que la prise en charge des fenêtres texturées était mise en œuvre en utilisant la valeur NSTexturedBackgroundWindowMask de NSWindowStyleMask. Toutefois, cette valeur a été supprimée dans macOS 10.12, de même que NSWindowStyleMaskTexturedBackground, qui a été supprimé dans macOS 10.14.

L'outil "jpackage"

La communauté OpenJDK a publié une préversion de la JEP 343 : Packaging Tool. Aussi connue sous le nom de jpackage, ce nouvel outil est capable de packager des applications Java auto-porteuses, incluant un environnement d'exécution Java (JRE).

Le groupe de l'Open JDK a publié une pré-version de la JEP 343 : Outil de Packaging. L'outil, connu sous le nom de jpackage est un nouvel utilitaire qui permet de packager des applications Java auto-porteuses, contenant donc un Java Runtime Environment (JRE). Le prototype s'appuie sur l'outil javapackager de JavaFX et est destiné au développeurs qui s'intéressent à jpackage.

Historiquement, les développeurs Java ont toujours souhaité avec un moyen de construire des applications qu'ils puissent installer nativement sur des plateformes, plutôt que d'installer un Java Runtime Environment (JRE), puis de distribuer des fichiers JAR et de configurer un classpath. Avec jpackage les applications Java peuvent dorénavant être installées et désinstallées en utilisant la façon classique liée à chaque plateforme - jpackage supporte les formats msi et exe sur Windows, pkg et dmg sur MacOS, ainsi que deb et rpm sur Linux.

L'outil jpackage a pour objectif de compléter les manques laissés par les autres technologies telles que :

- javapackager : un outil de packaging distribué avec l'Oracle JDK 8, retiré d'Oracle JDK 11 avec le reste de JavaFX ;
- Java Web Start, qui a été déprécié en Java 9, en même temps que le visualiseur d'Applets et JNLP, et retiré d'Oracle JDK 11 ;
- pack200, un outil créé pour compresser les fichiers JAR, qui a été déprécié en JDK 11 et sera retiré dans une version future.

L'outil jpackage supporte :

- les applications modulaires à un environnement d'exécution propre que l'on a intégré à l'aide de jlink ;
- les applications modulaires distribuées sous forme de modules JAR ou de fichiers JMOD ;
- de même que les applications legacy qui s'exécutent sur le classpathet sont composées de plusieurs fichiers JAR.

L'outil jpackage produit l'image d'une application Java qui contient toutes les dépendances nécessaires à son fonctionnement. Cette image est enregistrée dans un répertoire unique sur le système de fichiers. Elle peut inclure les éléments suivants :

- le lanceur de l'application, natif ;
- une image du JRE, qui inclut les modules de l'application si elle a été modularisée ;
- les ressources de l'application sous forme de JAR, fichier ico ou png ;
- des fichiers de configuration : plist, cfg, properties.

Il est prévu d'inclure l'outil jpackage dans la distribution du JDK 13 dans un nouveau module jdk.jpackage. L'interface de ligne de commande (CLI) sera conforme à la JEP 293 : Guidelines for JDK Command-line Tool Options. En outre, il est possible d'accéder à jpackage au traver le l'API ToolProvider (java.util.spi.ToolProvider) sous le nom jpackage.