NG-CONSULTING

LES NOUVEAUTÉS DE JAVA 8 ET LA PROGRAMMATION FONCTIONNELLE

Java 8

Rédigé par

TOUNGA Franck

LES NOUVEAUTÉS DE JAVA 8 ET LA PROGRAMMATION FONCTIONNELLE

Introduction

J'ai découvert Java 8 dans sa version instable, lors de ma première mission chez BforBank en 2013. Chaque nouvelle version de Java est importante, mais celle-ci change radicalement la donne. Ce document est un guide autour des nouveautés de Java 8. Il commence par une introduction pratique aux expressions lambdas, ensuite, il couvre la nouvelle API *Stream* et montre comment vous pourrez l'utiliser pour rendre le code basé sur les collections radicalement plus facile à comprendre et à maintenir. Il explique également d'autres fonctions importantes de Java 8, y compris les méthodes par défaut au niveau des interfaces, les Optionals, les Futures pour tout ce qui touche à l'exécution asynchrone et enfin la nouvelle API Date et heure. Le code source des études de cas est disponible sur mon repository officiel github: https://github.com/ftounga/java8features

Chapitre 1. Java 8 : pourquoi devriez-vous vous en soucier ?

Voici les principaux concepts que vous devriez retenir de ce chapitre :

- Gardez à l'esprit l'idée d'un écosystème des langages et la pression d'évoluer qui en résulte. Bien que Java puisse être extrêmement compétitif en ce moment, rappelez-vous d'autres langages tels que COBOL qui n'ont pas évolué.
- Les ajouts de base à Java 8 fournissent de nouveaux concepts et fonctionnalités passionnantes pour faciliter l'écriture de programmes à la fois efficaces et concis.
- Les processeurs multicœurs ne sont pas entièrement pris en charge par la pratique de programmation Java existante.
- Les fonctions sont des valeurs de première classe ; rappelez-vous comment les méthodes peuvent être transmises en tant que valeurs fonctionnelles et comment les fonctions anonymes (lambdas) sont écrites.
- Le concept de Streams de Java 8 généralise de nombreux aspects des collections, mais permet à la fois un code plus lisible et permet de traiter en parallèle les éléments d'un flux de données.
- Vous pouvez utiliser une méthode par défaut dans une interface pour fournir un corps de méthode si une classe d'implémentation choisit de ne pas le faire.
- D'autres idées intéressantes de la programmation fonctionnelle incluent le traitement de la valeur nulle et l'utilisation du *pattern matching*.

Chapitre 2 : Passer du comportement en paramètre de fonction

Voici les principaux concepts que vous devriez retirer de ce chapitre :

- Le paramétrage du comportement est la capacité d'une méthode à prendre plusieurs comportements différents en paramètre et à les utiliser en interne pour accomplir différentes fonctions.
- Le paramétrage du comportement vous permet de rendre votre code plus adaptable à l'évolution des besoins et réduit les efforts d'ingénierie à venir.
- Le paramétrage du comportement est un moyen de donner de nouveaux comportements en tant qu'arguments à une méthode. Mais c'était verbeux avant Java 8. Les classes anonymes ont aidé un peu avant Java 8 à se débarrasser de la verbosité associée à la déclaration de plusieurs classes concrètes pour une interface et qui n'étaient utilisées qu'une seule fois.
- L'API Java contient de nombreuses méthodes pouvant être paramétrées avec différents comportements, notamment le tri, les threads et la gestion de l'interface graphique.

Chapitre 3. Expressions lambda

Voici les principaux concepts que vous devriez retirer de ce chapitre :

- Une expression lambda peut être comprise comme une sorte de fonction anonyme : elle n'a pas de nom, mais elle comporte une liste de paramètres, un corps, un type de retour et peut-être même une liste d'exceptions pouvant être lancées.
- Les expressions lambda vous permettent de passer du code de manière concise.
- Une interface fonctionnelle est une interface qui déclare exactement une méthode abstraite.
- Les expressions lambda ne peuvent être utilisées que lorsqu'une interface fonctionnelle est attendue.
- Les expressions lambda permettent de fournir directement la mise en œuvre de la méthode abstraite d'une interface fonctionnelle et de traiter l'expression entière comme une instance d'une interface fonctionnelle.
- Java 8 est fourni avec une liste d'interfaces fonctionnelles communes dans le package *java.util .function*, qui comprend *Predicate <T>, Function <T, R>, Supplier <T>, Consumer <T> et BinaryOperator <T>* décrit dans le tableau 3.2.
- Il existe des spécialisations de primitives d'interfaces fonctionnelles génériques communes telles que *Predicate <T> et Function <T, R>* qui peuvent être utilisées pour éviter les opérations de *autoboxing* : *IntPredicate, IntToLongFunction*, etc.
- Le pattern d'exécution around (c'est-à-dire, exécuter une fonctionnalité au milieu du code qui est toujours requis dans une méthode, par exemple l'allocation des ressources et le nettoyage) peut être utilisé avec lambdas pour gagner en flexibilité et réutilisation.
- Le type attendu pour une expression lambda est appelé le type cible.
- Les références de méthode vous permettent de réutiliser une implémentation de méthode existante et de la transmettre directement.
- Les interfaces fonctionnelles telles *que Comparator, Predicate et Function* ont plusieurs méthodes par défaut qui peuvent être utilisées pour combiner des expressions lambda.

Chapitre 4. Présentation des Stream

Voici quelques concepts clés à retirer de ce chapitre :

- Une Stream est une séquence d'éléments provenant d'une source qui prend en charge les opérations de traitement de données.
- Les flux utilisent l'itération interne : l'itération est rendue abstraite lors de l'utilisation des opérations telles que filtre, mapping et le tri.
- Il existe deux types d'opérations de flux : les opérations intermédiaires et les opérations terminales.
- Les opérations intermédiaires telles que le filtre et le mapping renvoient un flux et peuvent être chaînées ensemble. Elles sont utilisées pour mettre en place un pipeline d'opérations, mais ne produisent aucun résultat.
- Les opérations terminales telles que *for-Each* et *count* renvoient une valeur nonstream et traitent un pipeline de flux pour renvoyer un résultat.
- Les éléments d'un flux sont calculés à la demande.

Chapitre 5. Travailler avec des flux

Ce fut un chapitre long mais enrichissant. Vous pouvez maintenant travailler avec les collections plus efficacement. En effet, les flux vous permettent d'exprimer des requêtes sophistiquées de traitement de données de manière concise. De plus, les flux peuvent être parallélisés de manière transparente. Voici quelques concepts clés à retenir de ce chapitre :

- L'API Streams vous permet d'exprimer des requêtes de traitement de données complexes. Les opérations communes de Stream sont résumées dans le tableau 5.1.
- Vous pouvez filtrer et découper un flux à l'aide des méthodes *filter*, *distinct*, *skip* et *limit*.
- Vous pouvez extraire ou transformer des éléments d'un flux à l'aide des méthodes *map* et *flatMap*.
- Vous pouvez trouver des éléments dans un flux en utilisant les méthodes *findFirst* et *findAny*. Vous pouvez faire correspondre un prédicat donné dans un flux à l'aide des méthodes a*llMatch, noneMatch et anyMatch*.
- Ces méthodes utilisent le *short-circuiting* : un calcul s'arrête dès qu'un résultat est trouvé; il n'y a pas besoin de traiter tout le flux.
- Vous pouvez combiner itérativement tous les éléments d'un flux pour produire un résultat en utilisant la méthode *reduce*, par exemple, pour calculer la somme ou trouver le maximum d'un flux.
- Certaines opérations telles que *filter* et *map* sont sans état ; elles ne stockent aucun état. Certaines opérations telles que *reduce* stocke l'état afin de calculer une valeur. Certaines opérations telles que *sort* et *distinct* stockent également l'état car elles doivent mettre en mémoire tampon tous les éléments d'un flux avant de renvoyer un nouveau flux. De telles opérations sont appelées opérations statefull.
- Il existe trois spécialisations primitives de flux : *IntStream, DoubleStream et LongStream*. Leurs opérations sont également spécialisées en conséquence.

- Les flux peuvent être créés non seulement à partir d'une collection, mais également à partir de valeurs, de tableaux, de fichiers et de méthodes spécifiques telles que *iterate* et *genrate*.
- Un flux infini est un flux qui n'a pas de taille fixe.

Chapitre 6 : Collecter les données via l'API Stream

Voici les concepts clés que vous devriez retenir de ce chapitre :

- *Collect* est une opération terminale qui prend comme argument diverses recettes (appelées collecteurs) pour accumuler les éléments d'un flux dans un résultat récapitulatif.
- Les collecteurs prédéfinis incluent la réduction et la récapitulation des éléments de flux en une seule valeur, telle que le calcul du minimum, du maximum ou de la moyenne. Ces collecteurs sont résumés dans le tableau 6.1.
- Les collecteurs prédéfinis permettent de regrouper les éléments d'un flux avec les éléments *groupingBy* et de partitionner un flux avec *partitioningBy*.
- Les collecteurs se composent efficacement pour créer des regroupements, des partitions et des réductions à plusieurs niveaux.
- Vous pouvez développer vos propres collecteurs en implémentant les méthodes définies dans l'interface Collector.

Chapitre 7 : Traitement des données en parallèle et performance

Dans ce chapitre, vous avez appris ce qui suit :

- L'itération interne vous permet de traiter un flux en parallèle sans avoir besoin d'utiliser explicitement et de coordonner différents threads dans votre code.
- Même si le traitement d'un flux en parallèle est si facile, rien ne garantit que cela accélèrera vos programmes en toutes circonstances. Le comportement et la performance des programmes parallèles peuvent parfois être contre-productifs, et pour cette raison, il est toujours nécessaire de les mesurer et de s'assurer que vous ne ralentissez pas plutôt vos programmes.
- L'exécution parallèle d'une opération sur un ensemble de données, comme le fait un flux parallèle, peut fournir une amélioration des performances, en particulier lorsque le nombre d'éléments à traiter est énorme ou que le traitement de chaque élément est particulièrement long.
- Du point de vue des performances, l'utilisation de la bonne structure de données, par exemple, en utilisant des flux primitifs au lieu des flux non spécialisés autant que possible, est presque toujours plus important que d'essayer de paralléliser certaines opérations.

- La structure fork/join vous permet de fractionner récursivement une tâche parallélisable en tâches plus petites, de les exécuter sur des threads différents, puis de combiner les résultats de chaque sous-tâche afin de produire le résultat global.
- Les *Spliterators* définissent comment un flux parallèle peut diviser les données qu'il traverse.

Chapitre 10 : Utilisation de l'Optional

Dans ce chapitre, vous avez appris ce qui suit :

- L'ancienne classe j*ava.util.Date* et toutes les autres classes utilisées pour modéliser la date et l'heure en Java avant Java 8 présentaient de nombreuses incohérences et failles de conception, notamment leur mutabilité et certains décalages, valeurs par défaut et noms mal choisis.
- Les objets date-heure de la nouvelle API Date et heure sont tous immuables.
- Cette nouvelle API fournit deux représentations temporelles différentes pour gérer les différents besoins des humains et des machines lors de leur fonctionnement.
- Vous pouvez manipuler les objets de date et d'heure de manière absolue et relative, et le résultat de ces manipulations est toujours une nouvelle instance, laissant l'original inchangé.
- *TemporalAdjusters* vous permet de manipuler une date d'une manière plus complexe que de simplement changer une de ses valeurs, et vous pouvez définir et utiliser vos propres transformations de date personnalisées.
- Vous pouvez définir un formatteur pour imprimer et parser les objets date-heure dans un format spécifique. Ces formatteurs peuvent être créés à partir d'un modèle ou par programmation et ils sont tous thread-safe.
- Vous pouvez représenter un fuseau horaire, à la fois relatif à une région/un emplacement spécifique et en tant que décalage fixe par rapport à UTC / Greenwich, et l'appliquer à un objet date-heure afin de le localiser.
- Vous pouvez utiliser des systèmes de calendrier différents du système standard ISO-8601.

Chapitre 13. Techniques de programmation fonctionnelle

Voici les concepts clés que vous devriez retenir de ce chapitre :

- Les fonctions de première classe sont des fonctions qui peuvent être passées en arguments, renvoyées en tant que résultats et stockées dans des structures de données.
- Une fonction d'ordre supérieur est une fonction qui prend au moins une ou plusieurs fonctions en entrée ou renvoie une autre fonction. Les fonctions d'ordre supérieur typiques en Java comprennent *comparing, andThen, et compose.*
- Currying est une technique qui vous permet de moduler des fonctions et de réutiliser du code.
- Une structure de données persistante préserve sa version précédente lorsqu'elle est modifiée. En conséquence, elle peut empêcher la copie défensive inutile.
- Les flux en Java ne peuvent pas être auto-définis.
- Une liste paresseuse est une version plus expressive d'un flux Java. Une liste paresseuse vous permet de produire des éléments de la liste à la demande en utilisant un supplier qui peut créer d'avantages d'éléments de la structure de données.
- Le pattern matching est une fonctionnalité qui vous permet de unwraper des types de données. Cela peut être vu comme une généralisation de l'instruction *switch* de Java.
- La transparence référentielle permet aux calculs d'être mis en cache.
- Les combinateurs sont une idée fonctionnelle qui combine deux ou plusieurs fonctions ou d'autres structures de données.

Chapitre 15. Conclusions et la suite pour Java

Ce chapitre couvre

- Nouvelles fonctionnalités de Java 8 et leur effet évolutif sur le style de programmation
- Quelques idées inachevées lancées par Java 8
- Ce que Java 9 et Java 10 pourraient apporter

Nous avons couvert beaucoup de points dans ce tutoriel, et nous espérons que vous vous sentez aptes à commencer à utiliser les nouvelles fonctionnalités de Java 8 dans votre propre code, en vous basant peut-être sur nos exemples et nos quiz. Dans ce chapitre, nous passons en revue le parcours d'apprentissage de Java 8 et la poussée progressive vers la programmation de style fonctionnel. En outre, nous spéculons sur les futures améliorations et les nouvelles fonctionnalités qui pourraient être dans le pipeline de Java au-delà de Java 8.

16.1. Revue des fonctionnalités de Java 8

Un bon moyen de vous aider à comprendre Java 8 comme un langage pratique et utile consiste à revisiter les fonctionnalités à leur tour. Au lieu de simplement les énumérer, nous aimerions les présenter comme étant interconnectés pour vous aider à les comprendre non pas simplement comme un ensemble de fonctionnalités, mais comme une vue d'ensemble de haut niveau sur la conception du langage cohérente qu'est Java 8. Le second objectif est de souligner comment la plupart des nouvelles fonctionnalités de Java 8 facilitent la programmation de style fonctionnel en Java. Rappelez-vous, ce n'est pas un choix de conception capricieux, mais une stratégie de conception consciente, centrée sur deux tendances :

- Le besoin croissant d'exploiter la puissance des processeurs multicœurs maintenant que, pour des raisons de technologie du silicium, les transistors supplémentaires fournis annuellement par la loi de Moore ne se traduisent plus par des vitesses d'horloge plus élevées des cœurs de processeurs individuels. En d'autres termes, rendre votre code plus rapide nécessite du code parallèle.
- La tendance croissante à manipuler de manière concise des collections de données avec un style déclaratif pour traiter des données, comme prendre une source de données, extraire toutes les données correspondant à un critère donné et appliquer une opération au résultat – soit en le résumant ou en rassemblant les données dans une variable (résultat) pour un traitement ultérieur plus tard. Ce style est associé à l'utilisation d'objets et de collections immuables, qui sont ensuite traités pour produire d'autres valeurs immuables.

Ni l'une ni l'autre motivation n'est efficacement soutenue par l'approche traditionnelle, orientée objet, impérative, centrée sur la mutation des champs et l'application des itérateurs. La mutation des données sur un noyau et la lecture d'un autre est étonnamment coûteuse, sans parler de la nécessité d'un verrouillage sujet à erreur; De même, lorsque votre état d'esprit se concentre sur l'itération et la mutation des objets existants, l'idiome de programmation semblable à un flux peut vous sembler très étranger. Mais ces deux tendances sont facilement

supportées en utilisant des idées issues de la programmation fonctionnelle, ce qui explique pourquoi le centre de gravité de Java 8 s'est un peu éloigné de ce que l'on attend de Java.

Passons maintenant en revue, ce que vous avez appris de ce tutoriel, et voyez comment tout cela s'intègre dans la nouvelle tendance

16.1.1. Paramétrage du comportement (lambdas et références de méthode)

Pour pouvoir utiliser une méthode réutilisable telle que *filter*, vous devez être capable de présenter en argument une description du critère de filtrage. Bien que les experts Java avaient imaginé dans les versions antérieures de Java (en encapsulant le critère de filtrage dans une classe et transmettant une instance de cette classe), une solution permettant de transmettre du comportement, cette solution ne conviendrait pas à un usage général car elle était trop lourde à écrire.

Comme vous l'avez découvert aux chapitres 2 et 3, Java 8 fournit un moyen, emprunté à la programmation fonctionnelle, de passer un morceau de code à une méthode. Il fournit deux variantes de ceci :

Passer une lambda, un morceau de code tel que:

```
apple -> apple.getWeight() > 150
```

Passer une référence de méthode, à une méthode existante :

```
Apple::isHeavy
```

Ces valeurs ont des types tels que *Fonction<T, R>*, *Prédicat<T> et BiFunction <T, U, R>* et permettent d'utiliser les méthodes *apply, test*, etc. Les lambdas peuvent sembler être plutôt un concept de niche, mais c'est la façon dont Java 8 les utilise dans une grande partie de la nouvelle API *Streams* qui les propulse au centre de Java.

16.1.2. Streams

Les classes de collection en Java, avec les itérateurs et la construction pour chaque, nous ont servi honorablement pendant longtemps. Il aurait été facile pour les concepteurs de Java 8 d'ajouter des méthodes comme *filter* et la *map* aux collections, en exploitant les lambdas mentionnés précédemment pour exprimer des requêtes de type base de données. Mais ils ne l'ont pas fait – ils ont plutôt ajouté une toute nouvelle API Streams, qui fait l'objet des chapitres 4 à 7, et il vaut la peine de réfléchir pour savoir pourquoi.

Qu'est-ce qui ne va pas avec les collections qui exigent qu'elles soient remplacées ou améliorées avec une notion similaire mais différente qui est les *Stream* ? Nous allons le résumer ainsi : si vous avez une grande collection et que vous lui appliquez trois opérations, en mappant éventuellement les objets de la collection pour additionner deux de leurs champs, en filtrant les sommes répondant à certains critères, puis en triant le résultat, fera trois

traversées séparées de la collection. L'API Streams effectue paresseusement ces opérations dans un pipeline, puis exécute une seule traversée de flux en effectuant toutes les opérations ensemble. C'est beaucoup plus efficace pour les grandes quantités de données, et pour des raisons telles que les caches de mémoire. En effet, plus l'ensemble de données est grand, plus il est important de minimiser le nombre de traversées.

Les autres, non moins importantes, concernent la possibilité de traiter des éléments en parallèle, ce qui est essentiel pour exploiter efficacement les processeurs multicœurs. Les flux, en particulier la méthode parallèle, permettent de marquer un flux comme approprié pour un traitement parallèle. Rappelons ici que le parallélisme et l'état mutable s'adaptent mal ensemble. Il devient évident que les concepts fonctionnels de bases (opérations sans effets secondaires et méthodes paramétrées avec lambdas et références de méthode qui permettent l'itération interne au lieu de l'itération externe, comme discuté au chapitre 4) sont essentiels à l'exploitation des flux en parallèle lors de l'utilisation de méthode telle que map, filter …etc

16.1.3. CompletableFuture

Java a fourni l'interface *Future* depuis Java 5. Les Futures sont utiles pour exploiter le multicœur car ils permettent à une tâche d'être générée sur un autre thread et de permettre à la tâche qui l'a générée de continuer à s'exécuter de façon asynchrone. Lorsque la tâche de génération a besoin du résultat, elle peut utiliser la méthode *get* pour attendre que la *Futur* se termine (produire sa valeur).

Le chapitre 11 explique l'implémentation Java 8 *CompletableFuture* de *Future*. Encore une fois, cela exploite les lambdas. On dit souvent « *Completable-Future is to Future as Stream is to Collection* ». Comparons:

- L'API Stream vous permet d'effectuer des opérations de pipeline et fournit un paramétrage de comportement avec *map*, *filter*, et ainsi de suite, évitant ainsi le code standard que vous devez généralement écrire en utilisant des itérateurs.
- De la même manière, *CompletableFuture* fournit des opérations telles que *thenCompose*, *thenCombine* et *allOf*, qui fournissent des règles de codage de design pattern courants, dans un style de programmation fonctionnelle impliquant *Futures*, et vous permettent d'éviter un code standard de style impératif.

Ce style d'opérations, bien que dans un scénario plus simple, s'applique également aux opérations Java 8 sur la classe *Optional*, que nous revisitons maintenant.

16.1.4. Optional

La bibliothèque Java 8 fournit la classe *Optional<T>*, qui permet à votre code de spécifier qu'une valeur est soit une valeur correcte de type T soit une valeur manquante renvoyée par la méthode statique *Optional.empty*. C'est génial pour la compréhension du programme et la documentation ; Elle fournit un type de données avec une valeur manquante explicite – au lieu de l'utilisation du pointeur *null* dans les versions précédentes pour indiquer des valeurs manquantes, dont nous ne pourrions jamais être sûrs si c'était une valeur manquante planifiée ou une erreur accidentelle résultant d'un calcul erroné.

Comme le chapitre 10 l'explique, si *Optional <T>* est utilisé de manière cohérente, les programmes ne doivent jamais produire de NullPointerExceptions. Encore une fois, vous pourriez voir cela comme un élément unique, sans rapport avec le reste de Java 8, et vous demander : « Comment le passage d'une forme de valeur manquante à une autre m'aide à écrire des programmes ?» Une inspection plus approfondie montre que la classe *Optional<T>* fournit *map*, *filter* et *ifPresent*. Ces méthodes ont un comportement similaire aux méthodes correspondantes dans la classe Streams et peuvent être utilisés pour enchaîner les calculs, toujours dans un style fonctionnel, avec les tests de valeur manquante qui sont effectués par la bibliothèque au lieu du code utilisateur. Ce test interne par rapport à un test externe est directement similaire à la façon dont la bibliothèque Streams effectue une itération interne par rapport à une itération externe dans le code utilisateur.

Le dernier sujet de cette section ne concerne pas la programmation de style fonctionnel, mais plutôt la prise en charge de Java 8 pour les extensions de bibliothèque compatibles avec les versions supérieures, dictées par les désirs de l'ingénierie logicielle.

16.1.5. Méthodes par défaut

Il y a d'autres ajouts à Java 8, dont aucun n'affecte particulièrement l'expressivité d'un programme individuel. Notamment, l'ajout de méthodes par défaut à une interface. Avant Java 8, les interfaces définissaient les signatures de méthodes ; maintenant, elles peuvent également fournir des implémentations par défaut pour les méthodes dont l'implémentation pourrait être absente.

C'est un nouvel outil formidable pour les concepteurs de bibliothèques, car cela leur permet de modifier une interface avec une nouvelle opération, sans avoir besoin de demander à tous les clients (classes implémentant cette interface) d'ajouter du code pour définir cette méthode. Par conséquent, les méthodes par défaut sont également pertinentes pour les utilisateurs des bibliothèques car elles les protègent des futures modifications de l'interface. Le chapitre 9 explique cela plus en détail.

Jusqu'à présent, nous avons résumé les concepts de Java 8. Nous nous penchons maintenant sur un sujet plus épineux. A savoir, ce que les futures améliorations et les nouvelles fonctionnalités peuvent être dans le pipeline de Java au-delà de Java 8.

16.2. Quel avenir pour Java ?

Regardons quelques-uns de ces points, dont la plupart sont discutés plus en détail sur le site Web du JDK Enhancement Proposal à http://openjdk.java.net/jeps/0. Ici, nous prenons soin d'expliquer pourquoi des idées apparemment sensibles ont des difficultés subtiles ou une interaction avec des fonctionnalités existantes qui inhibent leur incorporation directe dans Java.

16.2.1. Collections

Le développement de Java a été évolutif et non d'un coup. De nombreuses idées géniales ont été ajoutées à Java, par exemple, les tableaux étant remplacés par des collections et plus tard augmentés par la puissance des flux. Parfois, une nouvelle fonctionnalité est si nettement meilleure (par exemple, des collections sur des tableaux) que nous ne remarquons pas que

certains aspects de la fonctionnalité supplantée n'ont pas été transposés. Un exemple est les initialiseurs pour les conteneurs. Par exemple, les tableaux Java peuvent être déclarés et initialisés avec une syntaxe telle que :

```
Double [] a = {1.2, 3.4, 5.9};
```

qui est une abréviation pratique pour

```
Double [] a = new Double[]{1.2, 3.4, 5.9};
```

Les collections Java (via l'interface Collection) ont été introduites comme une meilleure mannière et plus uniforme de traiter des séquences de données telles que celles représentées par des tableaux. Mais leur initialisation a été plutôt négligée. Pensez à la façon dont vous initialisez un HashMap. Vous devriez écrire ce qui suit :

```
Map<String, Integer> map = new HashMap<>();
map.put("raoul", 23);
map.put("mario", 40);
map.put("alan", 53);
```

Ce que vous aimeriez pouvoir dire est quelque chose comme

```
Map<String, Integer> map = #{"Raoul" -> 23, "Mario" -> 40, "Alan" -> 53};
```

Où # {…} est un littéral de collection – une liste des valeurs qui doivent apparaître dans la collection. Cela semble plutôt utile comme fonctionnalité, mais elle ne fait pas encore partie de Java.

16.2.2. Amélioration du système de type

Nous discuterons de deux améliorations possibles du système de type en Java : la variance de site de déclaration et l'inférence de type de variable locale.

Variance de site de déclaration

Java prend en charge les caractères génériques en tant que mécanisme flexible pour permettre le sous-typage des génériques (plus généralement appelé variance de site d'utilisation). C'est pourquoi l'affectation suivante est valide :

```
List<? extends Number> numbers = new ArrayList<Integer>();
```

Mais l'affectation suivante, omettant le *? extends*, donne une erreur de compilation :

```
List<Number> numbers = new ArrayList<Integer>();    <-- Incompatible types
```

De nombreux langages de programmation tels que C# et Scala prennent en charge un mécanisme de variance différent appelé variance de site de déclaration. Ils permettent aux programmeurs de spécifier la variance lors de la définition d'une classe générique. Cette fonctionnalité est utile pour les classes qui sont intrinsèquement différentes. L'*Itérator*, par exemple, est intrinsèquement covariant et le Comparateur est intrinsèquement contravariant. Vous ne devriez pas avoir besoin de penser en termes de *? extends* ou *? super* quand vous les utilisez. C'est pourquoi l'ajout d'une variance de site de déclaration à Java serait utile car ces spécifications apparaissent à la déclaration des classes. En conséquence, cela réduirait certaines préoccupations pour les programmeurs. Notez qu'au moment de la rédaction de ce document (Aout 2017), la variance du site de déclaration peut déjà être disponible pour Java 9.

Encore plus d'inférence de type

À l'origine en Java, chaque fois que nous introduisions une variable ou une méthode, nous donnions son type en même temps. Par exemple,

```
double convertUSDToGBP(double money) { ExchangeRate e = ...; }
```

contient trois types ; ceux-ci donnent le type du résultat *convertUSDToGBP*, le type de son argument *money*, et le type de sa variable locale *e*. Au fil du temps, cela a été simplifié de deux façons. Tout d'abord, vous pouvez omettre les paramètres de type de génériques dans une expression lorsque le contexte les détermine. Par exemple,

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

peut être abrégé à ce qui suit depuis Java 7 :

```
Map<String, List<String>> myMap = new HashMap<>();
```

Deuxièmement, en utilisant la même idée – en propageant le type déterminé par le contexte dans une expression – une expression lambda telle que

```
Function<Integer, Boolean> p = (Integer x) -> booleanExpression;
```

peut être raccourci à

```
Function<Integer, Boolean> p = x -> booleanExpression;
```

en omettant les types. Dans les deux cas, le compilateur déduit les types omis.

L'inférence de type donne quelques avantages quand un type consiste en un seul identifiant, le principal étant le travail d'édition réduit lors du remplacement d'un type par un autre. Mais à mesure que la taille des types augmente, les génériques sont paramétrés par d'autres types génériques, l'inférence de type peut faciliter la lisibilité. Les langages Scala et C# permettent de remplacer un type dans une déclaration initialisée par une variable locale par le mot-clé *var*, et le compilateur remplit le type approprié du côté droit. Par exemple, la déclaration de *myMap* montrée précédemment en utilisant la syntaxe Java pourrait être représentée comme ceci :

```
var myMap = new HashMap<String, List<String>>();
```

Cette idée est appelée inférence de type sur variable locale ; vous pouvez vous attendre à des développements similaires en Java car cela réduit le fouillis causé par la répétition de type redondante.

Cependant, il y a une petite cause d'inquiétude ; Considérons une classe *Car* qui hérite d'une classe *Vehicule*, puis fait la déclaration :

```
var x = new Vehicle();
```

Déclarez-vous que *x* a un type *Car* ou plutôt *Vehicule* ? Dans ce cas, une explication simple sur le fait que le type manquant est le type de l'initialiseur (ici Véhicule) est parfaitement claire. L'explication peut être soutenue par le fait que le mot clé *var* ne peut pas être utilisé lorsqu'il n'y a pas d'initialiseur.

16.2.3. Pattern matching

Comme nous l'avons vu au chapitre 14, les langages de style fonctionnel fournissent généralement une forme de pattern matching – une forme améliorée de *Switch* dans laquelle vous pouvez demander : « Cette valeur est-elle une instance d'une classe donnée ?» Et, de manière récursive, ses champs ont certaines valeurs.

Il convient de vous rappeler ici que la conception orientée objet traditionnelle décourage l'utilisation de *switch* et encourage plutôt des modèles tels que le pattern de visiteur où le flux de contrôle dépendant du type de données est effectué par une méthode de dispatch plutôt que par *switch*. Ce n'est pas le cas à l'autre extrémité du spectre des langages de programmation – dans la programmation de style fonctionnel où la correspondance de modèle sur les valeurs des types de données est souvent le moyen le plus pratique de concevoir un programme.

L'ajout en général du pattern matching du style de Scala à Java semble un travail assez important, mais après la généralisation récente pour passer à Strings, vous pouvez imaginer une extension de syntaxe plus modeste, qui permette au *switch* de fonctionner sur des objets, en utilisant la syntaxe *instanceof* . Ici, nous revoyons notre exemple de la section 14.4 et supposons une classe *Expr*, qui est sous-classée dans *BinOp* et *Number* :

```
switch (someExpr) {
      case (op instanceof BinOp):
         doSomething(op.opname, op.left, op.right);
      case (n instanceof Number):
         dealWithLeafNode(n.val);
      default:
         defaultAction(someExpr);
}
```

Il y a un certain nombre de choses à noter. Nous empruntons du pattern matching l'idée que dans *case (op instanceof BinOp):, op* est une nouvelle variable locale (de type *BinOp*), qui devient liée à la même valeur que *someExpr* ; de même, dans le cas *Number*, n devient une variable de type *Number*. Dans le cas par défaut, aucune variable n'est liée. Cette proposition évite beaucoup de code boilerplate par rapport à l'utilisation de chaînes de *if-then-else* et de casting vers des sous-types. Un concepteur orienté objet classique argumenterait probablement qu'un tel code de répartition de type de données serait mieux exprimé en utilisant des méthodes de type visiteur surchargées dans les sous-types, mais pour les yeux de programmation fonctionnelle, le code associé serait dispersé sur plusieurs définitions de classe. C'est une dichotomie de conception classique discutée dans la littérature sous le nom de « problème d'expression ». –> http://en.wikipedia.org/wiki/Expression_problem.

16.2.4. Formes plus riches de génériques

Cette section traite de deux limitations des génériques Java et examine une évolution possible pour les atténuer.

Génériques réifiés

Lorsque les génériques ont été introduits dans Java 5, elles devaient être rétrocompatibles avec la JVM existante. À cette fin, les représentations d'exécution de *ArrayList <String>* et *ArrayList <Integer>* sont identiques. C'est ce qu'on appelle le modèle d'effacement du polymorphisme générique. Certains coûts d'exécution sont associés à ce choix, mais l'effet le plus significatif pour les programmeurs est que les paramètres des types génériques ne peuvent être que des objets. Supposons que Java autorise, disons, *ArrayList <int>*. Vous pouvez ensuite affecter un objet ArrayList sur le tas contenant une valeur primitive telle que int 42, mais le conteneur Array-List ne contient aucun indicateur indiquant s'il contient une valeur Object telle qu'une chaîne ou une valeur int primitive telle que 42.

A un certain niveau cela semble inoffensif – si vous obtenez une primitive 42 d'une *ArrayList <int>* et un objet String « abc » d'une ArrayList <String>, pourquoi devriez-vous vous inquiéter du fait que les conteneurs *ArrayList* soient indiscernables ? Malheureusement, la réponse est garbage collection, car l'absence d'informations de type au runtime sur le contenu de ArrayList laisserait la JVM incapable de déterminer si l'élément de votre *ArrayList* était une référence *Integer* (à suivre et marqué comme « en cours d'utilisation » par GC) ou une valeur int primitive (certainement pas à suivre).

Dans le langage C#, les représentations d'exécution de *ArrayList <String>, ArrayList <Integer> et ArrayList <int>* sont toutes différentes en principe. Mais même si elles sont identiques, des informations de type suffisantes sont conservées au moment de l'exécution

pour permettre, par exemple au garbage collector de déterminer si un champ est une référence ou une primitive. C'est ce qu'on appelle le modèle réifié du polymorphisme générique ou, plus simplement, des génériques réifiés. Le mot réification signifie « rendre explicite quelque chose qui autrement serait simplement implicite ».

Les génériques réifiés sont clairement souhaitables ; ils permettent une unification plus complète des types primitifs et de leurs types d'objets correspondants – quelque chose que vous verrez comme problématique dans les sections suivantes. La principale difficulté pour Java est la rétrocompatibilité, à la fois dans la JVM et dans les programmes existants qui utilisent la réflexion et qui s'attendent à ce que les génériques soient supprimés.

Flexibilité syntaxique supplémentaire dans les génériques pour les types de fonctions

Les génériques se sont révélés être une merveilleuse caractéristique lorsqu'elles ont été ajoutées à Java 5. Elles sont également parfaites pour exprimer le type de nombreuses références de méthode et de lambda Java 8. Vous pouvez exprimer une fonction à un argument :

```
Function<Integer, Integer> square = x -> x * x;
```

Si vous avez une fonction à deux arguments, vous utilisez le type *BiFunction <T, U, R>,* où T est le type du premier paramètre, U le second et R le résultat. Mais il n'y a pas de TriFunction sauf si vous le déclarez vous-même.

De même, vous ne pouvez pas utiliser la *fonction <T, R>* pour les références à des méthodes prenant zéro argument et renvoyant le type de résultat R ; vous devez utiliser le *Supplier<R>* à la place.

En substance, Java 8 lambdas a enrichi ce que vous pouvez écrire, mais le système de type n'a pas suivi la flexibilité du code. Dans de nombreux langages fonctionnels, vous pouvez écrire, par exemple, le type *(Integer, Double) => String*, pour représenter ce que Java 8 appelle *BiFunction <Integer, Double, String>,* avec *Integer => String* pour représenter *Function <Integer, String>*, et même *() => String* pour représenter le Supplier<String>. Vous pouvez comprendre => comme une version infixe de *Function*, *BiFunction*, *Supplier* etc. Une simple extension de la syntaxe Java pour les types le permettrait, ce qui donnerait des types plus lisibles, comme dans le cas de Scala, comme on l'à vu au chapitre 14.

Spécialisations primitives et génériques

En Java, tous les types primitifs (int, par exemple) ont un type d'objet correspondant (ici *java.lang.Integer*) ; souvent, nous nous référons à ceux-ci comme des types boxed (wrappés, encapsulés) et unboxed. Bien que cette distinction ait l'objectif louable d'augmenter l'efficacité de l'exécution, les types peuvent devenir source de confusion. Par exemple, pourquoi en Java 8 écrit-on *Predicate <Apple>* au lieu de *Function <Apple, Boolean>* ? Il s'avère qu'un objet de type *Predicate <Apple>*, lorsqu'il est appelé en utilisant la méthode *test*, renvoie un booléen primitif.

En revanche, comme tous les génériques, une fonction ne peut être paramétrée que par des types d'objets, ce qui dans le cas de la fonction <Apple, Boolean> est le type *Boolean*, pas le

type primitif booléen. *Le prédicat <Apple>* est donc plus efficace car il évite de mettre en boîte le prinmitif booléen pour faire un objet *Boolean*. Ce problème a conduit à la création de plusieurs interfaces similaires, telles que *LongToIntFunction* et *BooleanSupplier*, qui ajoutent une surcharge conceptuelle supplémentaire. Un autre exemple concerne la question des différences entre void, qui ne peut qualifier que les types de retour de méthode et n'a pas de valeurs, et le type d'objet *Void*, qui a pour seule valeur null – une question qui apparaît régulièrement sur les forums. Les cas spéciaux de Fonction tels que *Supplier <T>,* qui pourrait être écrit *() => T* dans la nouvelle notation proposée précédemment, attestent en outre des ramifications causées par la distinction entre types primitifs et types d'objets. Nous avons discuté plus tôt de la façon dont les génériques réifiés pouvaient résoudre nombre de ces problèmes.

16.2.5. Un soutien plus profond pour l'immuabilité

Certains lecteurs experts ont peut-être été un peu contrariés quand nous avons dit que Java 8 avait trois formes de valeurs :

- Valeurs primitives
- Objets
- Fonctions

À un certain niveau, nous allons nous en tenir à nos arguments et dire : « Ce sont les valeurs qu'une méthode peut prendre comme arguments et retourner comme résultats.» Mais nous souhaitons également admettre que c'est un peu problématique : Dans quelle mésure pouvez vous estimer renvoyer une valeur (mathématique) lorsque vous renvoyez une référence à un tableau mutable ? Une String ou un tableau immuable est clairement une valeur, mais le cas est beaucoup moins clair pour un objet ou un tableau modifiable – votre méthode peut retourner un tableau avec ses éléments dans l'ordre croissant, mais un autre code peut changer un de ses éléments plus tard.

Si nous nous intéressons vraiment à la programmation fonctionnelle en Java, il est nécessaire d'avoir un support linguistique pour dire « valeur immuable ». Comme indiqué au chapitre 13, le mot-clé *final* n'atteint pas vraiment cet objectif. Il interdit juste toute modification de primitives ; considérons ceci :

```
final int[] arr = {1, 2, 3};
final List<T> list = new ArrayList<>();
```

Le premier interdit une autre assignation *arr = ...* mais n'interdit pas *arr [1] = 2* ; ce dernier interdit les assignations à la liste mais n'interdit pas aux autres méthodes de changer le nombre d'éléments dans la liste. Le mot-clé *final* fonctionne bien pour les valeurs primitives, mais pour les références aux objets, il donne souvent un faux sentiment de sécurité.

Voici ce que nous faisons : étant donné que la programmation fonctionnelle met l'accent sur la non-mutation de la structure existante, il existe un argument fort pour un mot-clé tel que *transitively_final*, qui peut qualifier les champs de référence et qui garantit qu'aucune modification ne peut avoir lieu sur cet attribut ou à tout objet directement ou indirectement accessible via ce champ.

De tels types produisent une intuition à propos des valeurs : les valeurs sont immuables, et seules les variables (qui contiennent des valeurs) peuvent être mutées pour contenir une valeur immuable différente. Comme nous l'avons remarqué en tête de cette section, les auteurs Java, y compris nous-mêmes, avons parfois parlé de façon inconsistante de la possibilité qu'une valeur Java soit un tableau mutable. Dans la section suivante, discutons de l'idée d'une *valeur type* ; celles-ci ne peuvent contenir que des valeurs immuables, même si des variables de valeur type peuvent encore être mises à jour, à moins d'être qualifiées avec *final*.

16.2.6. Valeurs types

Dans cette section, nous discuterons de la différence entre types primitifs et types d'objets, en orientant la discussion sur le désir de valeurs type, qui vous aident à écrire des programmes fonctionnels, de la même mannière que les types d'objets sont nécessaires pour la programmation orientée objet. Bon nombre des problèmes dont nous discuterons sont interreliés, il n'y a donc pas de moyen facile d'expliquer un problème de façon isolé. Au lieu de cela, nous identifions le problème par ses différentes facettes.

Le compilateur ne peut-il pas traiter *Integer* et *int* de manière identique ?

Étant donné toutes les options implicites de boxing et unboxing que Java a lentement acquises depuis Java 1.1, vous pouvez vous demander s'il est temps pour Java de traiter, par exemple, *Integer* et *int* identiquement et de s'appuyer sur le compilateur Java pour optimiser la JVM.

Ce serait une excellente idée en principe, mais considérons les problèmes liés à l'ajout du type *Complex* à Java pour voir pourquoi l'autoboxing est problématique. Le type *Complex*, qui modélise des nombres dits complexes ayant des parties réelles et imaginaires, est naturellement introduit comme ceci:

```
class Complex {
    public final double re;
    public final double im;
    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
     public static Complex add(Complex a, Complex b) {
        return new Complex(a.re+b.re, a.im+b.im);
     }
}
```

Mais les valeurs de type *Complex* sont des types de référence, et chaque opération sur un *Complexe* doit faire une allocation d'objet, ce qui induit des coûts à chaque addition. Ce dont nous avons besoin, c'est d'un type primitif, qui peut être appelé *complexe*.

Le problème ici est que nous voulons un « objet sans boîte », et ni Java ni la JVM n'a de réel support pour cela. Maintenant, nous pouvons continuer à nous plaindre, « Oh, mais sûrement le compilateur peut optimiser cela. » Malheureusement, c'est beaucoup plus difficile qu'il n'y paraît ; Bien qu'il existe une optimisation du compilateur basée sur ce qu'on appelle l'analyse d'échappement (escape analysis), qui peut parfois déterminer que l'unboxing est correct, son applicabilité est limitée par les suppositions de Java sur les *objets*, qui sont présents depuis Java 1.1. Considérez le casse-tête suivant :

```
double d1 = 3.14;
double d2 = d1;
Double o1 = d1;
Double o2 = d2;
Double ox = o1;
System.out.println(d1 == d2 ? "yes" : "no");
System.out.println(o1 == o2 ? "yes" : "no");
System.out.println(o1 == ox ? "yes" : "no");
```

Le résultat est « oui », « non », « oui ». Un programmeur Java expert dirait probablement : « Quel code stupide, tout le monde sait que vous devriez utiliser *equal* sur les deux dernières lignes au lieu de ==. » Même si toutes ces primitives et objets contiennent la valeur immuable 3.14 et devraient vraiment être indiscernables, les définitions de *o1* et *o2* créent de nouveaux objets, et l'opérateur == (comparaison d'identité) peut les distinguer. Notez que sur les primitives, la comparaison d'identité fait une comparaison bit à bit mais sur les objets elle est faite sur la référence. Très souvent, nous créons accidentellement un nouvel objet *Double* distinct, que le compilateur doit respecter car la sémantique d'Object, dont *Double* hérite, l'exige. Vous avez déjà vu cette discussion, à la fois dans la discussion précédente sur les valeurs type et dans le chapitre 14, où nous avons discuté de la transparence référentielle des méthodes qui mettent à jour de manière fonctionnelle les structures de données persistantes.

Valeur type – tout n'est pas un primitif ou un objet

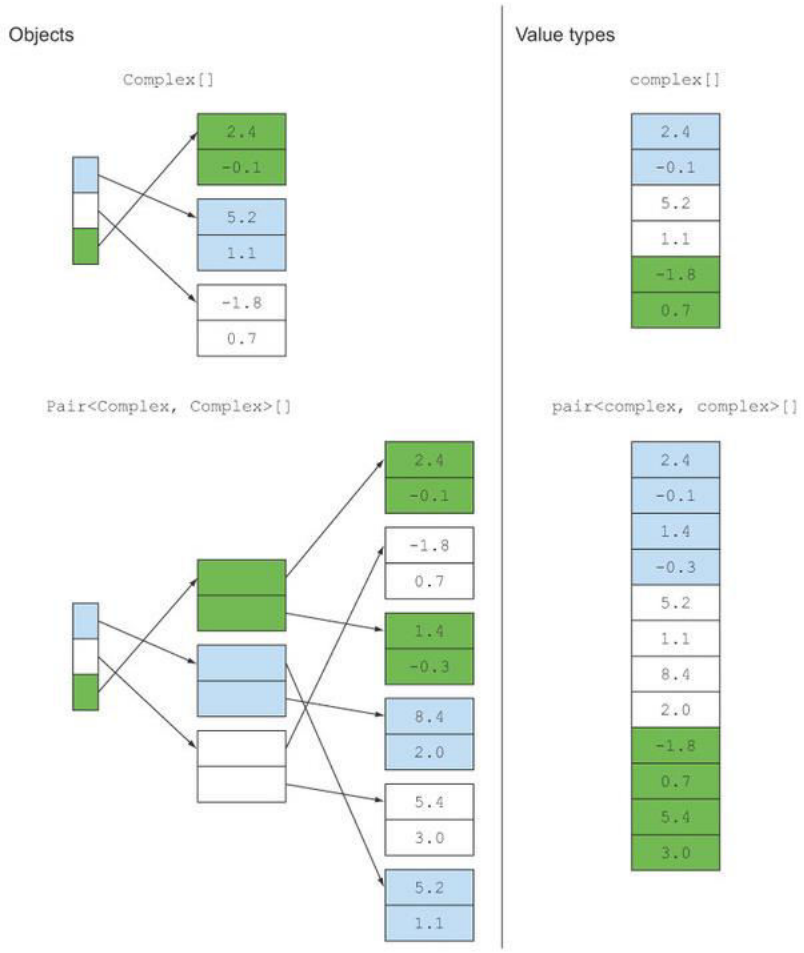
Nous suggérons que la résolution de ce problème consiste à retravailler les suppositions Java selon lesquelles que tout ce qui n'est pas une primitive est un objet et donc hérite d'*Object*, et que toutes les références sont des références à des objets.

Le développement commence comme ça. Il existe deux formes de valeurs : celles de type *Objet* qui ont des champs mutables à moins qu'elles ne soient interdites avec *final*, et celles d'identité, qui peuvent être testées avec ==. Il y a aussi des valeurs type, qui sont immuables et qui n'ont pas d'identité de référence ; les types primitifs sont un sous-ensemble de cette notion plus large. Nous pourrions alors autoriser des valeurs type définies par l'utilisateur (en commençant peut-être par une lettre minuscule pour souligner leur similarité avec les types primitifs tels que *int* et *boolean*). Sur les valeurs type, == effectuerait par défaut une comparaison élément par élément de la même manière que la comparaison matérielle sur *int* effectue une comparaison bit par bit. Vous pouvez voir que cela est remplacé pour plutôt la comparaison à virgule flottante, qui effectue une opération un peu plus sophistiquée. Le type *Complex* serait un exemple parfait de valeur type (non primitif); ces types ressemblent à des structures *C# (structs)*.

De plus, les types de valeur peuvent réduire les besoins de stockage car ils n'ont pas d'identité de référence. La figure 16.1 illustre un tableau de taille trois, dont les éléments 0, 1 et 2 sont respectivement gris clair, blanc et gris foncé. Le diagramme de gauche montre une exigence de stockage typique lorsque *Pair* et *Complex* sont des objets et la droite montre la meilleure disposition lorsque *Pair* et *Complex* sont des valeurs type (notez que nous les avons appelés paire et complexe en minuscule dans le diagramme pour souligner leur similarité avec les types primitifs). Notez également que les valeurs type sont également susceptibles d'améliorer les performances, non seulement pour l'accès aux données (plusieurs niveaux

d’indirection de pointeur remplacés par une seule instruction d’adressage indexée) mais aussi pour l’utilisation du cache matériel (contiguité des données).

Figure 16.1. Objets et valeurs type



Notez que parce que les valeurs types n’ont pas d’identité de référence, le compilateur peut alors les wrapper et les unboxer à son choix. Si vous passez un complexe en argument d’une fonction à une autre, le compilateur peut naturellement le passer comme deux *doubles* séparés. (Bien sûr, le renvoyer sans boxing est plus compliqué dans la JVM, car la JVM ne fournit que des instructions de retour de méthode en passant des valeurs représentables dans un registre machine 64 bits.) Mais si vous passez une valeur type plus grande (peut-être un grand tableau immuable), le compilateur peut à la place, de manière transparente pour l’utilisateur, la passer en référence une fois qu’elle a été *wrapper*. Une technologie similaire existe déjà en *C#* ;

Les structures peuvent sembler similaires aux classes, mais il existe des différences importantes dont vous devez être conscient. Tout d’abord, les classes sont des types de référence [C #] et les structures sont des valeurs type. En utilisant des structures, vous pouvez créer des objets qui se comportent comme les types [primitifs] intégrés et profiter de leurs avantages.

Au moment de la rédaction de cet article (juillet 2015), il existe une proposition concrète pour les valeurs type en Java.

Boxing, génériques, valeurs type-le problème d'interdépendance

Nous aimerions avoir des valeurs type en Java, car les programmes fonctionnels traitent des valeurs immuables qui n'ont pas d'identité. Nous aimerions voir les types primitifs comme un cas particulier des valeurs type, mais le modèle d'effacement des génériques, que Java a actuellement, signifie que les valeurs type ne peuvent pas être utilisés avec des génériques sans boxing. Les versions d'objets (par exemple, *Integer*) des types primitifs (par exemple, int) continuent d'être vitales pour les collections et les génériques Java en raison de leur modèle d'effacement, mais maintenant leur classe mère *Objet* (et donc l'égalité de référence) est maintenant vu comme un inconvénient. S'attaquer à l'un de ces problèmes signifie les aborder tous.

16.3. Le dernier mot

Ce livre a exploré les nouvelles fonctionnalités ajoutées par Java 8; Celles-ci représentent peut-être la plus grande étape d'évolution de Java – la seule étape d'évolution relativement importante était l'introduction, il y a 10 ans, des génériques dans Java 5. Dans ce chapitre, nous avons également examiné les pressions pour l'évolution de Java. En conclusion, nous proposons l'énoncé suivant :

Java 8 est un excellent endroit pour faire une pause mais pas pour s'arrêter.

Nous espérons que vous avez apprécié l'aventure qu'est Java 8, et que nous avons suscité votre intérêt à explorer la programmation fonctionnelle et l'évolution de Java.