Java 8 言語での変更内容

ラムダ式、そしてインターフェース・クラスに加えられた変更によって Java 8 がどのような点で新しい言語になっているかを学ぶ

Dennis Sosnoski
Principal Consultant
Sosnoski Software Solutions Inc.

2014年 6月 26日

Java 8 には、より簡単にプログラムを作成できるようにする、重要な新しい言語機能が組み込まれています。ラムダ式が定義する、インライン・コード・ブロック用の新しい構文は、遥かに少ないボイラープレートで匿名内部クラスと同様の柔軟性を実現します。また、Java 8 で加えられたインターフェースに対する変更によって、既存のインターフェースに追加をしても、既存のコードをそのまま使用できるようになりました。この記事では、これらの Java 8 での変更内容をともに使用する方法を説明します。ラムダ式を Java 8 の Stream で使用する方法については、関連記事の「JVM の並行性: Java 8 での並行処理の基礎」を参照してください。

Java 8 で行われた最大の変更は、ラムダ式のサポートが追加されたことです。ラムダ式とは、参照によって渡すことができるコードのブロックです。ラムダ式は他のプログラミング言語で使われているクロージャーと似ています。クロージャーとは、関数を実装し、オプションで 1 つ以上の入力引数を取り、オプションで結果の値を返すコードのことです。クロージャーはあるコンテキストで定義され、そのコンテキストから値にアクセスすることができます (ラムダ式の場合、読み取り専用のアクセスとなります)。

クロージャーの知識が十分にないとしても、心配いりません。Java 8 のラムダ式は実質的に、Java 開発者であれば使い慣れているはずの匿名内部クラスを特殊化したものだからです。コードの 1 箇所だけでクラスが必要な場合、匿名内部クラスを使用することで、インターフェースまたは基底クラスのサブクラスを、インラインで実装することができます。ラムダ式も同じように使用しますが、ラムダ式では短縮した構文を使用するため、標準的な内部クラス定義よりも簡潔なコードになります。

この記事では、さまざまな状況でラムダ式を使用する方法と、ラムダ式に関連する Java 言語の `interface` 定義の拡張について説明します。連載「JVM の並行性」には、関連する記事として「JVM の並行性: Java 8 での並行処理の基礎」があり、その中ではラムダ式を Java 8 の Stream 機能で使用する方法をはじめとし、ラムダ式を扱う例がこの記事よりも数多く記載されているので参照してください。

ラムダ式の詳細

ラムダ式は常に、Java 8 で「関数型インターフェース」と呼ばれる、単一の抽象メソッドを定義する `interface` クラスの実装となります。ラムダ式の構文ではメソッド名を使用しないので、単一の抽象メソッドに限定されている点が重要です。ラムダ式では、メソッド名の代わりにダック・タイピング (引数に応じた戻り値の型を得られるようにする手法で、多くの動的言語で使用されている手法) を使用して、指定されたラムダ式が、要求されるインターフェース・メソッドに対応することを確実にします。

リスト 1 に記載する単純な例では、ラムダ式を使用して `Name` インスタンスをソートしています。`main()` メソッド内の最初のコード・ブロックでは、匿名内部クラスを使用して `Comparator<Name>` インターフェースを実装し、2 番目のブロックではラムダ式を使用して実装します (この記事の完全なサンプル・コードへのリンクについては、「参考文献」を参照してください)。

リスト 1. 匿名内部クラスとラムダ式の比較

```
public class Name {
    public final String firstName;
    public final String lastName;

    public Name(String first, String last) {
        firstName = first;
        lastName = last;
    }

    // only needed for chained comparator
    public String getFirstName() {
        return firstName;
    }

    // only needed for chained comparator
    public String getLastName() {
        return lastName;
    }

    // only needed for direct comparator (not for chained comparator)
    public int compareTo(Name other) {
        int diff = lastName.compareTo(other.lastName);
        if (diff == 0) {
            diff = firstName.compareTo(other.firstName);
        }
        return diff;
    }
    ...
}

public class NameSort {

    private static final Name[] NAMES = new Name[] {
        new Name("Sally", "Smith"),
        ...
    };

    private static void printNames(String caption, Name[] names) {
        ...
    }

    public static void main(String[] args) {

        // sort array using anonymous inner class
```

```
        Name[] copy = Arrays.copyOf(NAMES, NAMES.length);
        Arrays.sort(copy, new Comparator<Name>() {
            @Override
            public int compare(Name a, Name b) {
                return a.compareTo(b);
            }
        });
        printNames("Names sorted with anonymous inner class:", copy);

        // sort array using lambda expression
        copy = Arrays.copyOf(NAMES, NAMES.length);
        Arrays.sort(copy, (a, b) -> a.compareTo(b));
        printNames("Names sorted with lambda expression:", copy);
        ...
    }
}
```

リスト 1 では、ラムダ式が簡単な匿名内部クラスの代わりとして使用されています。実際に、このような簡単な内部クラスはあまりにもよく使用されるため、ラムダ式は Java 8 プログラマーに直接的なメリットをもたらします (この例の内部クラスとラムダ式は、どちらも `Name` クラスで実装されたメソッドを使用して比較作業を行っています。ラムダ式に `compareTo()` メソッドをインライン化した場合には、これほど簡潔な式にはなりません)。

標準の function インターフェース

新しい `java.util.function` パッケージは、ラムダ式とともに使用するように意図された多種多様な関数型インターフェースを定義しています。これらのインターフェースは、次のカテゴリーに分かれています。

- Function: 引数を 1 つ取り、引数の値に基づいた結果を返します。
- Predicate: 引数を 1 つ取り、引数の値に基づいたブール値を結果として返します。
- BiFunction: 引数を 2 つ取り、引数の値に基づいた結果を返します。
- Supplier: 引数を取らずに、結果を返します。
- Consumer: 引数を 1 つ取り、結果を返しません (`void`)。

これらのカテゴリーのほとんどに、基本的なプリミティブ型引数や戻り値の型を扱うためのバリエーションがいくつか含まれています。インターフェースの多くは、インスタンスを結合するために使用できるメソッドを定義しています (リスト 2 を参照)。

リスト 2. 述部の結合

```
// use predicate composition to remove matching names
List<Name> list = new ArrayList<>();
for (Name name : NAMES) {
    list.add(name);
}
Predicate<Name> pred1 = name -> "Sally".equals(name.firstName);
Predicate<Name> pred2 = name -> "Queue".equals(name.lastName);
list.removeIf(pred1.or(pred2));
printNames("Names filtered by predicate:", list.toArray(new Name[list.size()]));
```

リスト 2 のコードでは、2 つの `Predicate<Name>` を定義しており、最初にファースト・ネーム Sally を突き合わせ、次にラスト・ネーム Queue を突き合わせています。`pred1.or(pred2)` メソッド呼び出しによって作成される結合述部は、2 つの述部を順に適用し、このうちのいずれかの述部が `true` に評価された場合 (Java での論理演算子 `||` と同じく早期出力を使用) に `true` を返すこ

とで定義されます。`List.removeIf()` メソッドは、この結合述部をリストに適用して、一致する名前をリストから削除します。

Java 8 では、多数の有用な `java.util.function` インターフェースの組み合わせを定義していますが、これらの組み合わせは一貫していません。述部のバリエーション (`DoublePredicate`、`IntPredicate`、`LongPredicate`、`Predicate<T>`) のすべてで、定義している構成メソッドと変更メソッドは `and()`、`negate()`、および `or()` で共通している一方、`Function<T>` のプリミティブ型バリエーションでは、構成メソッドも変更メソッドも定義していません。関数型プログラミング言語を使用した経験がある方にとっては、このような違いと省略は奇異に映ることでしょう。

`interface` の変更

`interface` クラスの構造 (例えば、リスト 1 のコードで使用した `Comparator`) は、Java 8 で変更されました。その目的の 1 つは、ラムダ式を使いやすくすることにあります。Java 8 より前のインターフェースに定義できたのは、定数と抽象メソッドだけで、抽象メソッドは後から実装しなければなりません。Java 8 では、インターフェースに `static` メソッドと `default` メソッドの両方を定義できるようになっています。インターフェースに定義された静的メソッドは、基本的に抽象クラスに定義された静的メソッドと同じです。デフォルト・メソッドは、むしろ旧式のインターフェース・メソッドに近いものがありますが、デフォルト・メソッドに指定された実装は、メソッドをオーバーライドしない限り使用されます。

デフォルト・メソッドの重要な特徴の 1 つは、既存の `interaface` にデフォルト・メソッドを追加しても、そのインターフェースを使用する他のコードをそのまま変更せずに使用できることです (ただし、既存のコードが同じメソッド名を別の目的で使用していないことが条件です)。この強力な機能を利用して、Java 8 の設計者たちは既存の Java ライブラリーの多くにラムダ式のサポートを組み込みました。リスト 3 に、一例を示します。これはリスト 1 のコードに名前のソートを追加するための 3 つ目の形です。

リスト 3. キー抽出コンパレーターのチェーニング

```
// sort array using key-extractor lambdas
copy = Arrays.copyOf(NAMES, NAMES.length);
Comparator<Name> comp = Comparator.comparing(name -> name.lastName);
comp = comp.thenComparing(name -> name.firstName);
Arrays.sort(copy, comp);
printNames("Names sorted with key extractor comparator:", copy);
```

リスト 3 のコードが最初に示しているのは、定義されたキー抽出ラムダ式に基づいて、新規 `Comparator.comparing()` 静的メソッドでコンパレーターを作成する方法です (厳密には、このキー抽出ラムダ式は `java.util.function.Function<T,R>` インターフェースのインスタンスです。このインスタンスでは、作成されるコンパレーターの型は `T` との代入互換性があり、抽出されたキーの型 `R` が `Comparable` インターフェースを実装します)。このコードは、新規 `Comparator.thenComparing()` デフォルト・メソッドを使用してコンパレーターを結合する方法も示しています。リスト 3 の場合、このデフォルト・メソッドが返すのは、最初にラスト・ネームでソートし、次にファースト・ネームでソートする新しいコンパレーターです。

コンパレーターの作成は、次のようにインライン化できそうに思えるかもしれません。

```
Comparator<Name> comp = Comparator.comparing(name -> name.lastName)
    .thenComparing(name -> name.firstName);
```

残念ながら、この方法は Java 8 の型推論には使えません。コンパイラーに追加情報として、静的メソッドからの結果に求められる戻り値の型を指定する必要があるからです。それには、以下のいずれかの形式を使用します。

```
Comparator<Name> com1 = Comparator.comparing((Name name1) -> name1.lastName)
    .thenComparing(name2 -> name2.firstName);
Comparator<Name> com2 = Comparator.<Name,String>comparing(name1 -> name1.lastName)
    .thenComparing(name2 -> name2.firstName);
```

最初の形式では、`(Name name1) -> name1.lastName` として、ラムダ式にラムダ引数の型を追加します。この情報があれば、コンパイラーは残りの必要な処理を理解することができます。2 番目の形式では、コンパイラーに対し、`comparing()` メソッドに渡される関数インターフェース (この例では、ラムダ式で実装されるインターフェース) の型 `T` および `R` を伝えます。

コンパレーターを簡単に作成してチェーニングできることは Java 8 の有用な機能ですが、複雑さが追加されるという犠牲が伴います。Java 7 の `Comparator` インターフェースが定義しているメソッドは 2 つ (`compare()` と、すべてのオブジェクトに対して定義されることが保証された、偏在する `equals()`) であるのに対し、Java 8 バージョンが定義しているメソッドの数は 18 です (元の 2 つのメソッドに加え、9 つの新しい静的メソッドと 7 つの新しいデフォルト・メソッド)。このようにインターフェースが大量に膨れ上がってラムダ式が一緒に使われるというパターンは、Java 標準ライブラリーのかなりの部分で何度も目にするはずです。

既存のメソッドをラムダ式のように使用する方法

すでに目的を果たしている既存のメソッドがある場合、「メソッド参照」を使用して、そのメソッドを直接渡すことができます。リスト 4 に、この方法を示します。

リスト 4. 既存のメソッドをラムダ式のように使用する方法

```
...
// sort array using existing methods as lambdas
copy = Arrays.copyOf(NAMES, NAMES.length);
comp = Comparator.comparing(Name::getLastName).thenComparing(Name::getFirstName);
Arrays.sort(copy, comp);
printNames("Names sorted with existing methods as lambdas:", copy);
```

リスト 4 の処理内容はリスト 3 のコードと同じですが、ここでは既存のメソッドを使用しています。Java 8 の `ClassName::methodName` という形のメソッド参照構文を使用すれば、任意のメソッドをラムダ式であるかのように使用することができます。その効果は、そのメソッドを呼び出すだけのラムダ式を定義する場合とまったく同じです。メソッド参照を使用できる対象には、静的メソッド、特定のオブジェクトのインスタンス・メソッドまたはラムダ式への入力型のインスタンス・メソッドのいずれか (リスト 4 の場合、`getFirstName()` および `getLastName()` メソッドは、比較対象の `Name` 型のインスタンス・メソッドです)、そしてコンストラクターがあります。

メソッド参照は便利なだけでなく、場合によってはラムダ式を使用するよりも効率的です。また、より明確な型情報をコンパイラーに提供できます (これが、前のセクションでラムダ式の問題となった `.thenComparing` 構成体が、リスト 4 のメソッド参照では有効に機能する理由です)。既

存のメソッドへのメソッド参照を使用するか、ラムダ式を使用するかの選択が許される場合は、常にメソッド参照を使用するようにしてください。

キャプチャー型ラムダ式と非キャプチャー型ラムダ式

この記事でこれまでに記載したラムダ式の例は、すべて非キャプチャー型です。これは、インターフェース・メソッドのパラメーターに相当するものとして渡された値だけを使用する単純な式であることを意味します。Java 8 のキャプチャー型ラムダ式は、包含するコンテキストからの値を使用します。キャプチャー型ラムダ式は、他の JVM 言語 (Scala を含む) で使用されているクロージャーと似ています。ただし、Java 8では、包含するコンテキストからの値は事実上、final でなければならないという点が異なります。つまり、値は真に `final` であるか (これは、以前のバージョンの Java では、匿名内部クラスから参照される値が final でなければならないのと同じです)、あるいはコンテキスト内で値が決して変更されないことが基準となります。この基準は、ラムダ式で使用する値にも、匿名内部クラスで使用する値にも適用されます。

事実上の final でなければならないという制約を回避するには、いくつかの次善策があります。例えば、ラムダ式に含まれる特定の変数の現行値だけを使用するには、新しいメソッドを追加し、そのメソッドでそれらの現行値を引数として取り、キャプチャーした値を含めた (適切なインターフェース参照という形の) ラムダ式を返すという方法を使えます。包含するコンテキストからの値をラムダ式に変更させるには、その値を可変ホルダーでラップするという方法もあります。

非キャプチャー型ラムダ式は、キャプチャー型ラムダ式より効率的に処理できる可能性があります。非キャプチャー型ラムダ式であれば、コンパイラーが包含クラス内に静的メソッドとして生成することができるため、ランタイムのマジックによってメソッド呼び出しを直接インライン化することができます。キャプチャー型ラムダ式は効率性には劣るかもしれませんが、同じコンテキストでは匿名内部クラスと同じかそれ以上のパフォーマンスを発揮します。

ラムダ式の舞台裏

ラムダ式は、匿名内部クラスと同じように見えますが、実装方法は異なります。Java の内部クラスは大掛かりな構成体であり、バイトコード・レベルで、内部クラスごとに個別のクラス・ファイルが存在します。大半のデータは (大抵は、定数プールのエントリーという形で) 重複していて、わずかな量のコードを追加するだけでも、クラスのロードによってかなりのランタイム・オーバーヘッドが追加されます。

Java 8 は、ラムダ式に個別のクラス・ファイルを使用するのではなく、Java 7 で追加された `invokedynamic` バイトコード命令を使用するように作られています。`invokedynamic` が対象とするブートストラップ・メソッドが初めて呼び出されると、このメソッドによってラムダ式の実装が作成され、その後は返された実装が直接呼び出されるという仕組みです。この方法では、クラスのロードによるランタイム・オーバーヘッドのほとんどと、別個のクラス・ファイルを使用することによるスペース・オーバーヘッドが回避されます。ラムダ関数をどのように実装するかの詳細は、ブートストラップに任されます。現在、Java 8 によって生成されるブートストラップ・コードは、実行時にラムダ式の新規クラスを作成するようになっていますが、別の手法が使用されるかどうかは将来の実装次第です。

Java 8 には、`invokedynamic` によるラムダ式の実装を実際に有効に機能させるための最適化が組みまれています。Scala (2.10.x) をはじめとする他のほとんどの JVM 言語では、コンパイラーが生成

した内部クラスをクロージャーで使用しますが、これらの言語でも、Java 8 (および以降) の最適化を利用するために、今後のバージョンで `invokedynamic` の手法に移行することが考えられます。

ラムダ式の制約

記事の冒頭で説明したように、ラムダ式は常に、ある特定の関数インターフェースの実装となります。ラムダ式を渡すには、必ずインターフェース参照として渡さなければなりません。また、他のインターフェース実装の場合と同じく、特定のインターフェースとなるように作成されたラムダ式は、そのインターフェースとしてのみ使用することができます。リスト 5 では、(名前を除いて) まったく同じ関数型インターフェースのペアを使用して、この制約について示しています。Java 8 のコンパイラーは、両方のインターフェースのラムダ式実装として `String::length` メソッドを受け入れますが、ラムダ式が最初のインターフェースのインスタンスとして定義された後は、そのラムダ式を 2 番目のインターフェースのインスタンスとして使用することはできません。

リスト 5. ラムダ式の制約

```
private interface A {
    public int valueA(String s);
}
private interface B {
    public int valueB(String s);
}
public static void main(String[] args) {
    A a = String::length;
    B b = String::length;

    // compiler error!
    // b = a;

    // ClassCastException at runtime!
    // b = (B)a;

    // works, using a method reference
    b = a::valueA;
    System.out.println(b.valueB("abc"));
}
```

Java インターフェースの観点で考えるとしたら、Java インターフェースは常に上記リストのような方法で機能してきたため (ただし、最後の部分で使用しているメソッド参照は、Java 8 で新しく追加されたものです)、リスト 5 のコードに意外な点は何もありません。けれども、Scala などの関数型プログラミング言語を扱ったことがある開発者にとって、このインターフェースの制約は直観に反しているはずです。

関数型プログラミング言語では、変数を定義するのにインターフェースではなく、関数型を使用します。このような言語では、高階関数を扱うのが一般的です。高階関数は、関数を引数として渡したり、関数を値として返したりするので、関数をビルディング・ブロックとして使用して他の関数を作成できるなど、ラムダ式より大幅に柔軟なプログラミング・スタイルが実現します。Java 8 では関数型を定義していないため、同じようにしてラムダ式を作成することはできません。(リスト 3 の例で示したように) インターフェースを作成することはできますが、それは、関連するその特定のインターフェースで機能するようにコードを作成した場合に限られます。新しい `java.util.function` パッケージ 1 つをとっても、ラムダ式専用の 40 のインターフェースが設

定されています。これらのインターフェースと既存の数百のインターフェースを合わせると、インターフェースを作成する方法は常に非常に限られてしまうことがわかります。

関数型を追加するのではなく、インターフェースを使用するという選択は、慎重に検討された結果です。こうすることにより、Java ライブラリーを大幅に変更する必要がなくなると同時に、ラムダ式を既存のライブラリーで使用することができます。その一方、Java 8 が真の関数型プログラミングではなく、いわゆる「インターフェース・プログラミング」(関数型のようなプログラミング) 以上にはならないという欠点があります。けれども、JVM では他の豊富な言語 (関数型言語を含む) を使用できるため、これは切迫した制約ではありません。

まとめ

ラムダ式は Java 言語の主要な拡張機能です。アプリケーションが Java 8 に移行されるにつれ、ラムダ式とその兄弟メソッド参照は、あらゆる Java 開発者にとって瞬く間に不可欠なツールとなるはずです。ラムダ式は、Java 8 の Stream と組み合わせて使用すると特に効果を発揮します。ラムダ式と Stream が連動することで並行プログラミングが単純化され、アプリケーションのパフォーマンスが向上する仕組みを理解するには、「JVM の並行性: Java 8 での並行処理の基礎」を参照してください。

著者について

Dennis Sosnoski

Dennis Sosnoski は、スケーラブルなシステムの開発経験が豊富にある、Java および Scala の開発者です。XML と Web サービスの分野で有名な彼のバックグラウンドとしては、JiBX XML データ・バインディングの開発や、いくつかのオープンソース Web サービス・フレームワーク (一番最近のものでは Apache CXF) に関する取り組みなどがあります。Dennis は Java ユーザー・グループや Java カンファレンスで頻繁にプレゼンターを務めており、人気のある連載「Java Web サービス」をはじめとし、developerWorks の数多くの記事を執筆しています。彼が行っている Web サービスのトレーニングと、コンサルティング作業について Sosnoski Software Associates Ltd サイトで詳しい情報を得てください。また、彼が現在行っている JVM に関する並行プログラミングの探求を Scalable Scala サイトでチェックして読んでください。