Java™ magazine
By and for the Java community

JAVAFX：FXMLLOADER（続編） 40 | CLOJURE 49

2017年9月/10月

もっと、Java 9

05 モジュール：モジュールとは何か、モジュールをどう使うのか

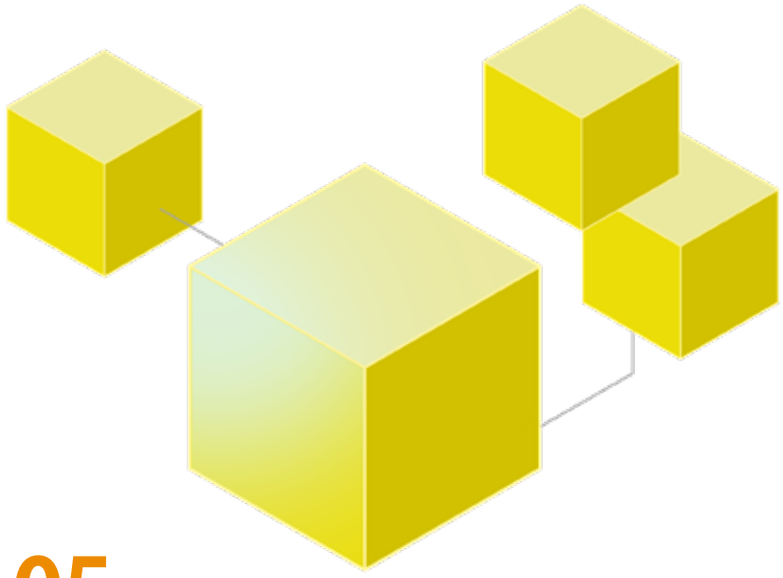
20 JAVA 9で更新されたコア・ライブラリ

30 JDKの内側：メソッド呼出しの仕組み

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /



05

JAVA 9モジュールを理解する

著者：Paul Deitel

モジュールとは何か、モジュールをどう使うのか

20

JAVA 9で更新されたコア・ライブラリ：OPTIONALとCOMPLETABLEFUTURE

著者：Raoul-Gabriel Urma、 Richard Warburton

OptionalとCompletableFutureの変更により、ビジネス・アプリケーションのエラー・ケースをより的確にモデリング

30

JAVAにおけるメソッド呼出しの仕組み

著者：Ben Evans

特殊なバイトコードでメソッドの呼出しを大幅に効率化。その仕組みを理解することで、JVMがコードを実行する方法を解き明かす

表紙画像：WES ROWELL

02

編集長より

OpenJDKとOracle JDKの集約：2つのリリース間に残された相違が近いうちになくなる見込み

40

JavaFX

FXMLLoaderでFXMLのカスタム動作を定義する

著者：Andrés Almiray

FXMLを用いた、JavaFXアプリケーションへのカスタム動作の注入

49

JVM言語

Clojure

著者：Brad Cypert

Lispのエレガントさを JVMのパフォーマンスで

59

Fix This

著者：Simon Roberts

中級者および上級者向けの言語知識を試せる最新のクイズ

03

Java注目の提案

提案中のJEP：コンテナ認識

19

ユーザー・グループ

Edinburgh JUG

OpenJDKとOracle JDKの集約

2つのリリース間の相違が近いうちになくなる見込み

突発的な重大ニュースでしたが、オラクルは、いくつかの重要なJavaの変更について発表しました。これらの変更は、どれも近日中に行われる予定です。9月初旬に行われたこの発表は、主に2つの内容に関わるものでした。1つはリリースの頻度について、もう1つはOpenJDKとオラクル製品との集約についてです。

前号の「編集長より」の記事では、メジャー・リリースの日程を決定する際にコミュニティが果たす役割について書きました。さらに、オラクルが新たなリリース周期への移行を模索している可能性にも触れました。新しいスケジュールに関する試案は、Java Platform Groupのチーフ・アーキテクトであるMark Reinholdがブログ投稿で詳しく説明しています。要約すると、現在の複数年周期に代わって、半年ごとに機能をリリースするというものです。Reinholdがこの新しいスケジュールの根拠を明確に説明したことは評価できます。ぜひ、この投稿の全文を読んでみることをお勧めします。

もし、Java 8のリリース後にこのスケジュールが実現していたとすれば、Java Magazineの本号と前号で紹介した機能の多くは、モジュールの実装の完成を待たずに公開されていたことでしょう。モジュールこそ、Java 9の提供が何か月も遅れた要因です。

Reinholdの提案はおおむね了承されるだろうというのが私の意見ですが、そうなった場合、開発者や、Javaを実行しているサイトはよい影響を受けることになるでしょう。新しい周期が承認され、発表された際には、その波及効果のいくつかについて、本欄または本編の記事で解説したいと思います。

もう1つの発表は、OpenJDKとOracle JDKのバイナリを交換可能にするという意向でした。ここでは、この発表に注目したいと思います。Oracle JDKの大部分は、オラクルからダウンロードできる無償版ですが、商用ライセン

写真：BOB ADLER/THE VERBATIM AGENCY

ORACLE

Java in the Cloud

Oracle Cloud delivers high-performance and battle-tested platform and infrastructure services for the most demanding Java apps.

Oracle Cloud.
Built for modern app dev.
Built for you.

Start here:
developer.oracle.com

#developersrule

スの対象となるいくつかのクローズド・ソース・ユーティリティも含まれています。その中には、評価の高いJava Flight Recorderなどのクローズド・ソース機能も含まれていますが、それらの機能は、OpenJDKのコントリビュータとの議論を経て、2018年中にオープン・ソース化される予定です。

このステップによって、すでにオープン・ソースとなっているOpenJDKは完全なものに近づきます。オラクルがOpenJDKをビルドし、簡単にOracle JDKと交換できるようになれば、クラウド・デプロイメントが容易になり、コンテナにデプロイできるのはどのJavaかを問う必要もなくなります。

Reinholdによると、この作業の目的は、「開発者にとってのOpenJDKビルドの魅力を高め、Oracle JDKとの差異を減らす」ことです。今までは、これら2つのJDKは異なるため、OpenJDKに全面的に頼ることはできないという疑念や心配がありました。そのため、この目的は価値あるものです。RedHat、Azul、Linaroなど、OpenJDK向けのビルドを提供しているベンダーは、長い間この疑念と闘わなくてはなりませんでした。しかし、今回ついにこの問題が解決され、Java SE開発キット全体が単一の統合技術ソースとしてオープン・ソース化されることになります。

1人のJava開発者として、私も新しいスケジュールとオープン・ソースへのさらなる貢献がもたらす、発展と今後のメリットの両方を楽しみにしています。

本号の主要記事をお読みいただければ、今までの複数段組みレイアウトが1段組みレイアウトに変更されていることにお気づきでしょう。この変更により、Java Magazineは、ブラウザでの読みやすさを維持しながら、モバイル機器で読みやすくなります。大きな画像を掲載できるようになることに加え、短い行に合わせてコードを再フォーマットする必要もなくなります。この新しいレイアウトが気に入ったか、気に入らないか、またはさらに読みやすくする提案はないか、ぜひお聞かせください。お待ちしています。

Editor in Chief、Andrew Binstock
javamag_us@oracle.com
@platypusguy

提案中のJDK拡張提案

Javaのコンテナ認識

マイクロサービスを実現する要素の1つに、サービスをコンテナにデプロイできることがあるのは間違いありません。コンテナとは、Dockerなどの簡易仮想環境を指します。オラクルのEmbedded Javaチームに所属するBob Vandetteによるこの JDK拡張提案（JEP）は、コンテナ内のJavaがコンテナを認識できるようにする技術を定義することを目指すものです。この提案で対処しようとしている主な問題は、Javaランタイムに影響する可能性があるコンテナ・ポリシーをどのようにしてJavaに認識させること（特に、リソースの制限について正確に伝達すること）ができるかというものです。

現在、JVMは起動時にCPUや環境についての情報を取得しています。Dockerコンテナ用のコマンドライン・スイッチで物理構成が設定されている場合、この情報は不正確である可能性があります。そのため、JVMが収集するデータから、使用可能なメモリが過大に見積もられる可能性があります。それによって内部的に誤った意思決定が行われることになり、最終的にはパフォーマンスの低下につながります。

この提案（なお、これはまだJEPの段階に達していません）では、コンテナのさまざまな構成詳細についての正確なデータを取得する新しいAPIを開発することが提唱されています。取得対象のデータとしては、メモリの制限および使用量、CPUの数、セット、および利用率、ブロックI/Oの詳細、デバイスI/Oの読取り速度および書込み速度、共有メモリサイズなどのホスト詳細といったものが検討されています。

どのような形になるにせよ、遅かれ早かれ、コンテナでのJava言語の使用がサポートされることになりそうだというのは明らかです。その場合、ここで提案されているようなJEPや、Java Specification Request（JSR）としてまずは登場することになるはずです。

モジュールを理解する 05
OPTIONALとCOMPLETABLEFUTUREの新機能 20
JAVA 8およびJAVA 9でのメソッド呼出し 30

もっと、Java 9

Java 9での進化はあまりに広範囲に及ぶため、何号かにわたってJava 9を紹介したとしてもすべてを網羅できないでしょう。Java 9は、JavaとJDKのあらゆる面でメジャー・リリースなのです。前号では、大きな変更点の多くや、Java 8からJava 9への移行方法について紹介しましたが、Java 9でもっとも重要なイノベーションであるモジュールに触れることはできませんでした。その理由は、前号の「編集長より」の記事でお伝えしました。Javaプラットフォームの新リリースは、オラクル、重要なパートナー、ユーザー・コミュニティの協働作業によるものですが、モジュールについての最終投票が行われたとき、リリースを進めるために必要な賛同が得られなかったのです。しかし、いくつかの細かい変更の結果、モジュールのリリースに向けた合意が得られました。仮にモジュールについて扱った場合、不正確な情報や不完全な情報を掲載してしまうというリスクを伴うこととなっていました。本号では、その埋め合わせとして、15ページにわたってモジュールを紹介します。有名なトレーナーであるPaul Deitel氏がモジュールとは何か、どう使うのかについて解説した記事です。

また、言語の変更点についても引き続き取り上げていきます。Raoul-Gabriel Urma氏とRichard Warburton氏による、OptionalとCompletableFutureの新機能の紹介です。この2つの機能はJava 8で脚光を浴び、今回のリリースで強化されています。さらに、JDK 9でのメソッド呼出しの仕組みも解説します。

加えて、Clojure（Lispに似たJVM言語）の初心者向け概要、JavaFXの特集の続き、そして、詳しい解説がついたいつものクイズも掲載しています。

なお、本号より、1段組みの新しいレイアウトに移行しています。特にモバイル機器で記事を読みやすくするためですが、いかがでしょうか。まだ改善の余地があると思われますか? あるとすれば、どのような点でしょうか? ぜひ、javamag_us@oracle.comまでご意見をお寄せください。お待ちしています。

画像：WES ROWELL

PAUL DEITEL

Java 9モジュールを理解する

モジュールとは何か、モジュールをどう使うのか

本記事では、Javaが誕生して以来もっとも重要なソフトウェア・エンジニアリング・テクノロジーとして新たに登場したJava 9プラットフォーム・モジュール・システム（JPMS）について紹介します。Project Jigsawの成果であるモジュール化によって、あらゆるレベルの開発者によるソフトウェア・システムのビルド、メンテナンス、改良の生産性が高まります。とりわけ、大規模システムにはこのことが当てはまります。

モジュールとは

モジュール化によって、パッケージの上位に新しい集まりが加わります。そこで鍵となる新しい言語要素がモジュールです。モジュールは、関連するパッケージ、リソース（イメージ・ファイルやXMLファイルなど）、モジュール記述子の集まりで、一意の名前が与えられており、再利用できます。モジュール記述子には、以下の内容を指定します。

- モジュールの名前
- モジュールの依存性（すなわち、そのモジュールが依存するモジュール）
- 明示的に他のモジュールから使用できるようにするパッケージ（モジュール内にある他のすべてのパッケージは、他のモジュールからは暗黙的に使用不可能になります）
- 提供するサービス
- 使用するサービス
- リフレクションを許可するモジュール

歴史

Java SEプラットフォームは、1995年に誕生しました。そして現在は、Internet of Things（IoT）や埋込みデバイスなど、リソースが限られたデバイスで動作する小規模アプリから、ビジネスに欠かせない大規模ミッション・クリティカル・システムまで、あらゆるものを構築するために使用されています。また、Java SEを使用している開発者は約1000万人にのぼります。そこには大量のレガシー・コードも存在しますが、現在に至るまでJavaプラットフォームはそもそもモノリシックであり、あらゆるものに対応する包括的なソリューションであり続けています。長年にわた

り、Javaのモジュール化に向けたさまざまな取り組みが行われてきましたが、普及したものはなく、Javaプラットフォーム自体をモジュール化するために使われたものもありませんでした。

Java SEプラットフォームのモジュール化の実装は難しく、その作業には長い年月を要しています。JSR 277：Javaモジュール・システムは、もともとJava 7のために2005年に提案されたものです。このJSRは、後にJSR 376：Javaプラットフォーム・モジュール・システムに取って代わられ、ターゲットはJava 8になりました。そして、Java 9ではJava SEプラットフォームがモジュール化されましたが、そのJava 9も2017年9月まで遅れました。

すべてのモジュールは、その依存性を明示的に宣言する必要があります。

目標

JSR 376によれば、Java SEプラットフォームのモジュール化が目標としているのは、主に以下の点です。

- 信頼性の高い構成：モジュール化により、モジュール間の依存性を明示的に宣言するメカニズムが提供されるため、コンパイル時と実行時の両方で依存性を認識できます。システムは、この依存性を横断的に確認することで、アプリをサポートするために必要となるすべてのモジュールで構成されるサブセットを特定できます。
- 強力なカプセル化：モジュール内のパッケージは、そのモジュールで明示的にエクスポートされた場合のみ、他のモジュールからアクセスできます。その場合でも、パッケージのエクスポート元モジュールの機能が必要であることを明示的に宣言しない限り、エクスポートされたパッケージを他のモジュールから使用することはできません。これにより、攻撃者がアクセスできるクラスが少なくなるため、プラットフォームのセキュリティが向上します。モジュール化を検討することで、より美しく論理的な設計を考え出せる可能性もあります。
- スケーラブルなJavaプラットフォーム：これまで、Javaプラットフォームは多数のパッケージで構成されたモノリシックなシステムでした。そのため、開発やメンテナンス、改良が難しく、簡単にサブセットを作成することはできませんでした。現在のところ、プラットフォームは95のモジュールに分割されており（この数は、Javaが進化するにつれて変わる可能性があります）、アプリや対象デバイスに必要なモジュールのみで構成されたカスタム・ランタイムを作ることができるようになっています。たとえば、デバイスがGUIをサポートしていない場合、GUIモジュールを含まないランタイムを作成することもできるため、ランタイムのサイズを大幅に削減できます。
- プラットフォームの整合性向上：プラットフォーム内にありながらも、アプリのクラスによって使われることが想定されていないクラスが多数あります。Java 9より前は、そのようなクラスも使うことができました。Java 9では、強力なカプセル化により、そういった内部APIは完全にカプセル化され、プラットフォームを使用するアプリには

隠蔽されます。ただし、内部APIに依存しているコードがある場合、そのレガシー・コードをJava 9モジュールに移行すると、問題が起きる可能性があります。

- パフォーマンスの向上：JVMでは、さまざまな最適化手法を使ってアプリケーションのパフォーマンスを改善しています。JSR 376に記載されているように、そういった最適化手法は、必要な型が特定のモジュール内にしかないことがあらかじめわかっている場合、さらに効果が上がります。

JDKのモジュールの一覧表示

Java 9の重要な側面の1つは、JDKがモジュールに分割され、さまざまな構成がサポートされている点です（「JEP 200：モジュール式JDK」を参照してください。Javaのモジュール化に関するすべてのJEPおよびJSRを、表1に示します）。まずは、JDKのbinフォルダから、次のように--list-modulesオプションを指定してjavaコマンドを使ってみます。

```
java --list-modules
```

すると、JDKの一連のモジュールが一覧表示されます。その中には、Java言語SE仕様を実装する標準モジュール（名前がjavaで始まるもの）、JavaFXモジュール（名前がjavafxで始まるもの）、JDK固有のモジュール（名前がjdkで始まるもの）、Oracle固有のモジュール（名前がoracleで始まるもの）が含まれています。各モジュール名の後には、バージョン文字列が続きます。たとえば、@9はモジュールがJava 9に付属するものであることを示しています。

モジュールの宣言

前述のように、モジュールにはモジュール記述子が必要です。モジュール記述子は、モジュールの依存性や、他のモジュールから使用できるようにするパッケージなどを指定するメタデータです。モジュール記述子は、module-info.javaという名前のファイル内で定義した

JEP 200：	モジュール式JDK
JEP 201：	モジュール・ソース・コード
JEP 220：	モジュールのランタイム・イメージ
JEP 260：	大半の内部APIの隠蔽
JEP 261：	モジュール・システム
JEP 275：	モジュール式JAVAアプリケーションのパッケージ化
JEP 282：	JLINK：JAVAリンカー
JSR 376：	JAVAプラットフォーム・モジュール・システム
JSR 379：	JAVA SE 9

表1： Javaのモジュール化に関するJEPとJSR

モジュール宣言をコンパイルしたものです。それぞれのモジュール宣言では、moduleキーワードに続けて一意のモジュール名を記述し、波括弧内にモジュール本体を記述します。次に例を示します。

モジュール・システムの主な動機の1つに、強力なカプセル化があります。

```
module モジュール名 {
}
```

モジュール宣言の本体は、空になる場合も、さまざまなモジュール・ディレクティブが含まれる場合もあります。モジュール・ディレクティブには、requires、exports、provides…with、uses、opensなどがあります（それぞれについて詳しくは後述します）。後ほど説明しますが、モジュール宣言をコンパイルするとモジュール記述子が作成されます。モジュール記述子は、module-info.classという名前のファイルとしてモジュールのルート・フォルダに保存されます。ここで、それぞれのモジュール・ディレクティブについて簡単に紹介します。その後、実際のモジュール宣言について見ていきます。

exports、module、open、opens、provides、requires、uses、withの各キーワードは、限定キーワードです。後ほど紹介するtoとtransitiveも同様です。限定キーワードは、モジュール宣言の中でのみキーワードとして扱われます。コードの他の部分では、識別子として使うことができます。

requires：requiresモジュール・ディレクティブは、そのモジュールが別のモジュールに依存することを指定します。この関係をモジュール依存性と呼びます。すべてのモジュールは、その依存性を明示的に宣言する必要があります。モジュールAがモジュールBを必要とする場合、モジュールAがモジュールBを読み取り、モジュールBはモジュールAによって読み取られることになります。別のモジュールへの依存性を指定するためには、次のようにrequiresを使用します。

```
requires モジュール名;
```

また、requires staticというディレクティブもあります。このディレクティブは、コンパイル時にはモジュールが必要になるものの、実行時には省略可能であることを示します。これをオプション依存性と呼びますが、本記事では取り上げません。

requires transitive—暗黙の可読性：暗黙の可読性とは、別のモジュールへの依存性を指定しつつ、現在のモジュールを読み取る他のモジュールもその依存性を読み取れるようにすることです。暗黙の可読性

を指定するためには、次のようにrequires transitiveを使います。

requires transitive モジュール名;

java.desktopのモジュール宣言に含まれる、次のディレクティブについて考えてみます。

requires transitive java.xml;

この場合、java.desktopを読み取るすべてのモジュールは、暗黙的にjava.xmlを読み取ります。たとえば、java.desktopモジュールのあるメソッドがjava.xmlモジュールで定義されている型を返す場合、java.desktopを読み取るモジュールのコードはjava.xmlに依存することになります。java.desktopのモジュール宣言にrequires transitiveディレクティブがない場合、そのような依存モジュールは、明示的にjava.xmlを読み取らない限りコンパイルできません。

JSR 379によると、この例で示したような場合、Java SEの標準モジュールは必ず暗黙の可読性を付与することになっています。また、Java SE標準モジュールは非標準モジュールに依存する可能性がありますが、非標準モジュールに暗黙の可読性を付与することはありません。これによって、Java SE標準モジュールのみに依存するコードでは、Java SE実装間の移植性が保証されています。

exportsとexports…to：exportsモジュール・ディレクティブは、モジュールに含まれる1つのパッケージのpublic型（およびネストされているpublic型とprotected型）に対して、他のモジュールすべてのコードからアクセスできることを指定します。また、exports…toディレクティブを使うと、エクスポートされたパッケージにどのモジュールまたはモジュールのコードがアクセスできるかをカンマ区切りのリストによって厳密に指定できます。これを制限付きエクスポートと呼びます。

uses：usesモジュール・ディレクティブは、モジュールが使用するサービスを指定します。つまり、そのモジュールはサービス・コンシューマになります。サービスとは、usesディレクティブで指定されたインタフェースを実装するか、またはabstractクラスを拡張するクラスのオブジェクトのことを指します。

provides…with：provides…withモジュール・ディレクティブは、モジュールがサービスの実装を提供することを指定します。つまり、そのモジュールはサービス・プロバイダになります。ディレクティブのprovides部では、モジュールのusesディレクティブに記載されるインタフェースまたはabstractクラスを指定します。with部では、インタフェースを実装またはabstractクラスを拡張したサービス・プロバイダ・クラスの名前を指定します。

open、opens、opens…to：Java 9より前では、リフレクションを使ってパッケージ内のすべての型や

型のすべてのメンバーを参照できました。その際、この機能を許可したいかどうかにかかわらず、privateなメンバーもすべて参照できました。つまり、本当の意味でカプセル化されているものは何もありませんでした。

モジュール・システムの主な動機の1つに、強力なカプセル化があります。デフォルトでは、他のモジュールからモジュール内の型にアクセスすることはできません。アクセスできるようになるのは、型がpublicかつパッケージをエクスポートした場合であるため、公開したいパッケージのみを公開できます。Java 9では、リフレクションにもこの考え方が適用されます。

実行時にのみパッケージへのアクセスを許可する：opensモジュール・ディレクティブは、次の形式で記述します。

```
opensパッケージ
```

これにより、他のモジュールのコードから、特定のパッケージ内のpublic型（およびネストされているpublic型とprotected型）に対して実行時にのみアクセスできることを指定します。また、指定したパッケージのすべての型（およびその型のすべてのメンバー）にリフレクションを使ってアクセスすることもできます。

実行時にのみ特定のモジュールに対してパッケージへのアクセスを許可する：opens…toモジュール・ディレクティブは、次の形式で記述します。

```
opens パッケージ to カンマ区切りのモジュール・リスト
```

これにより、記載したモジュールのコードから、特定のパッケージ内のpublic型（およびネストされているpublic型とprotected型）に対して実行時にのみアクセスできることを指定します。指定したモジュールのコードでは、指定したパッケージのすべての型（およびその型のすべてのメンバー）にリフレクションを使ってアクセスすることもできます。

実行時にのみモジュール内のすべてのパッケージへのアクセスを許可する：すべてのモジュールに対して実行時にモジュール内のすべてのパッケージにリフレクションを使ってアクセスできるようにする場合は、次のようにしてモジュール全体をopenすることができます。

```
open module モジュール名 {
  // モジュール・ディレクティブ
}
```

リフレクションのデフォルト

デフォルトで、実行時にリフレクションを使ってパッケージにアクセスできるモジュールは、パッケージのpublic型（およびネストされているpublic型とprotected型）を参照できます。ただし、他のモジュールのコードは、公開されているパッケージのすべての型とその型のすべてのメンバーにアクセスできます。その際には、以前のバージョンのJavaと同じように、setAccessibleを使ってprivateメンバーにアクセスすることもできます。setAccessibleおよびリフレクションについて詳しくは、オラクルのドキュメントを参照してください。

コンパイル時に同じ名前のモジュールが複数見つかった場合、コンパイル・エラーが発生します。実行時に同じ名前のモジュールが複数見つかった場合、例外が発生します。

依存性注入：通常、リフレクションは依存性注入とともに使われます。その例の1つが、FXMLベースのJavaFXアプリです。[編集注：FXMLの使用について詳しくは、本号の「FXMLLoaderでFXMLのカスタム動作を定義する」をご覧ください。] FXMLアプリがロードされると、アプリが依存するコントローラ・オブジェクトとGUIコンポーネントが動的に作成されます。この仕組みを以下に示します。

- アプリはGUIとの相互作用を処理するコントローラ・オブジェクトに依存しているため、最初にFXMLLoaderが、実行中のアプリにコントローラ・オブジェクトを注入します。つまり、FXMLLoaderはリフレクションを使い、コントローラ・クラスを探してメモリにロードし、そのクラスのオブジェクトを作成します。
- 次に、コントローラはFXMLで宣言されているGUIコンポーネントに依存しているため、FXMLLoaderがFXMLで宣言されているGUIコントロールを作成してコントローラ・オブジェクトに注入します。その際に、各GUIコントロールは、コントローラ・オブジェクト内の対応する@FXMLインスタンス変数に割り当てられます。さらに、@FXML付きで宣言されているコントローラのイベント・ハンドラは、FXMLでの宣言のとおりに、対応するコントロールにリンクされます。

この処理が完了すると、コントローラとGUIがやり取りを行えるようになり、イベントに応答できるようになります。opens…toディレクティブを使うと、FXMLLoaderがカスタム・モジュール内にあるJavaFXアプリに対してリフレクションを使用できるようになります。

モジュールを使ったWelcomeアプリ

このセクションでは、モジュールの基礎について説明するために、単純なWelcomeアプリを作ってみます。ここでは、以下のことを行います。

- モジュールに格納するクラスを作成する
- モジュールを宣言する
- モジュール宣言とWelcomeクラスをコンパイルし、モジュールを作成する
- モジュールにmainを含むクラスを実行する

また、基本について紹介した後、以下のことも説明します。

- WelcomeアプリをモジュールJARファイルにパッケージ化する
- JARファイルからアプリを実行する

Welcomeアプリの構造：このセクションで紹介するアプリは、2つの.javaファイルで構成されています。Welcome.javaにはWelcomeアプリのクラスが、module-info.javaにはモジュール宣言が含まれています。慣例的に、モジュール化されたアプリは次のようなフォルダ構造になっています。

アプリのフォルダ
　src
　　モジュール名フォルダ
　　パッケージのフォルダ階層
　　Javaソース・コード・ファイル群
　　module-info.java

今回のアプリは、com.deitel.welcomeパッケージに定義するため、フォルダ構造は図1のようになります。

srcフォルダには、アプリのソース・コードをすべて格納します。具体的には、名前がモジュールと同じcom.deitel.welcomeである、モジュールのルート・フォルダが含まれています（モジュールのネーミングについては、後述します）。モジュールのルート・フォルダには、パッケージのディレクトリ構造を表す、com/deitel/welcomeというネストされたフォルダが含まれています。このフォルダは、com.deitel.welcomeパッケージに対応するものです。このフォルダに、Welcome.javaが含まれています。モジュールのルート・フォルダには、必須であるモジュール宣言module-info.javaも含まれています。

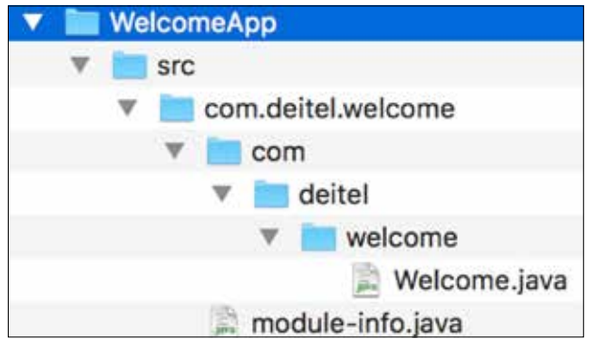


図1：Welcomeアプリのフォルダ構造

モジュールのネーミング規則：パッケージ名と同様

に、モジュール名も一意である必要があります。一意性を確保するため、通常はパッケージ名の最初を、組織のインターネット・ドメイン名を逆順にしたものにします。筆者が使っているドメイン名はdeitel.comです。そのため、パッケージ名の最初はcom.deitelとなります。慣習上、モジュール名にもドメイン名を逆順にしたものを使用します。

コンパイル時に同じ名前のモジュールが複数見つかった場合、コンパイル・エラーが発生します。実行時に同じ名前のモジュールが複数見つかった場合、例外が発生します。

今回の例では、モジュール内にあるパッケージは1つだけであるため、モジュールとそこに含まれるパッケージに同じ名前を使っています。これは必須ではありませんが、一般的な慣習です。モジュールを使うアプリでは、Javaはモジュール名をパッケージ名やパッケージ内の型の名前とは別に管理しています。そのため、モジュール名とパッケージ名が重複しても問題ありません。

通常、モジュールは関連するパッケージをまとめたものです。そのため、パッケージ名には共通部分があることがほとんどです。たとえば、モジュールに次のパッケージが含まれているとします。

```
com.deitel.sample.firstpackage;
com.deitel.sample.secondpackage;
com.deitel.sample.thirdpackage;
```

通常は、パッケージ名の共通部分をモジュールの名前にします。この例では、com.deitel.sampleです。共通部分がない場合は、モジュールの目的を示す名前を選びます。たとえば、java.baseモジュールには、Javaアプリに欠かせないと考えられているコア・パッケージ（java.lang、java.io、java.time、java.utilなど）が含まれています。また、java.sqlモジュールには、JDBCを経由したデータベースの操作に必要なパッケージ（java.sql、javax.sqlなど）が含まれています。ここで紹介したのは、多数ある標準モジュールのうちの2つだけです。java.baseなど、各標準モジュールのオンライン・マニュアルには、そのモジュールでエクスポートされているパッケージの完全なリストが掲載されています。

Welcomeクラス：Welcomeアプリのコードを次に示します。このアプリは、コマンドラインに文字列を表示するだけのものです。モジュール内に格納される型を定義する場合、すべての型をパッケージに格納する必要があります（コードの3行目を参照）。

```
// Welcome.java
// モジュールに格納されるWelcomeクラス
package com.deitel.welcome; // モジュール内のすべてのクラスはパッケージに格納する必要がある
```

```
public class Welcome {
  public static void main(String[] args) {
    // Systemクラスは、java.baseモジュールのjava.langパッケージ内に存在する
    System.out.println("Welcome to the Java Platform Module System!");
  }
}
```

module-info.javaのモジュール宣言：次のリストは、com.deitel.welcomeモジュールのモジュール宣言です。

```
// module-info.java
// com.deitel.welcomeモジュールのモジュール宣言
module com.deitel.welcome {
   requires java.base; // これはすべてのモジュールに暗黙的に含まれるため、省略可能
}
```

繰り返しになりますが、モジュール宣言はmoduleキーワードに続けてモジュール名を記述し、波括弧内に宣言本体を記述します。このモジュール宣言には、requiresモジュール・ディレクティブが含まれています。このモジュール・ディレクティブは、アプリがjava.baseモジュールで定義されている型に依存することを示しています。実際には、すべてのモジュールがjava.baseに依存するため、このrequiresモジュール・ディレクティブはすべてのモジュール宣言で暗黙的に行われています。そのため、このモジュール・ディレクティブを省略して次のように記述することも可能です。

```
module com.deitel.welcome {
}
```

モジュールをコンパイルする：Welcomeアプリのモジュールをコンパイルするためには、コマンド・ウィンドウを開き、cdコマンドを使ってWelcomeAppフォルダに移動した後、次のように入力します。

```
javac -d mods/com.deitel.welcome ^
   src/com.deitel.welcome/module-info.java ^
```

src/com.deitel.welcome/com/deitel/welcome/Welcome.java

[注：記号^は、Microsoft Windowsの行継続文字です。上記のコマンドは、行継続文字を使用せずに1行で入力することもできます。LinuxおよびmacOSのユーザーが複数行にわたるコマンドを入力する場合は、コマンドライン内のキャレット（^）をバックスラッシュ行継続文字（\）に置き換える必要があります。] -dオプションを使うと、javacはコンパイルしたコードを指定されたフォルダに格納します。この例では、コンパイル済みモジュールを表すcom.deitel.welcomeという名前のサブフォルダを含むmodsフォルダに格納しています。慣習上、モジュールを含むフォルダにはmodsという名前が使用されます。

コンパイル後のWelcomeアプリのフォルダ構造：コードが正常にコンパイルされると、コンパイル済みコードがWelcomeAppフォルダのmodsサブフォルダ構造に格納されます（図2参照）。このフォルダは、展開済みモジュール・フォルダと呼ばれています。この名称は、このフォルダと.classファイルがJAR（Javaアーカイブ）ファイルに含まれていないことに由来しています。展開済みモジュールの構造は、前述した、アプリのsrcフォルダに類似しています。後ほど、このアプリをJARにパッケージ化します。展開済みモジュール・フォルダとモジュールJARファイルは、モジュール・アーティファクトと呼ばれます。モジュール・アーティファクトは、モジュール化したコードのコンパイル時および実行時にモジュール・パス（モジュール・アーティファクトの格納場所のリスト）に配置できます。

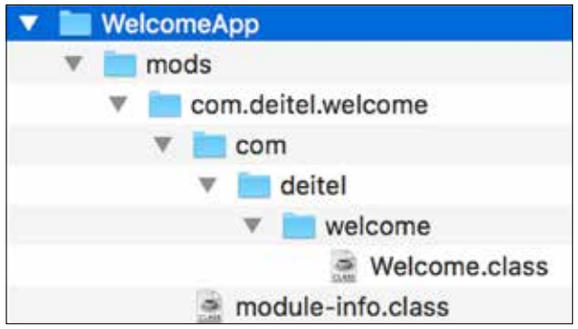


図2：WelcomeアプリのmodsフォルダのMの構造

モジュールの説明を参照する：javaコマンドの--describe-moduleオプションを使うと、com.deitel.welcomeモジュール記述子の情報を参照できます。WelcomeAppディレクトリから、次のコマンドを実行してください。

```
java --module-path mods --describe-module com.deitel.welcome
```

次の結果が出力されます。

```
com.deitel.welcome pathContainingModsFolder/mods/com.deitel.welcome/
requires java.base
```

contains com.deitel.welcome

まず、モジュール名とその格納場所が出力されています。続いて、このモジュールには標準モジュールjava.baseが必要で、パッケージcom.deitel.welcomeが含まれていることが出力されています。モジュールには上記のパッケージが含まれていますが、エクスポートはされていません。そのため、他のモジュールからパッケージの内容を使用することはできません。この例のモジュール宣言では、java.baseが必要であることを明示的に宣言したため、先ほどの出力には次の行が含まれています。

requires java.base

java.baseが必要であることを明示的に宣言しなかった場合、出力には代わりに次の行が含まれます。

requires java.base mandated

mandatedというモジュール・ディレクティブはありません。このモジュール・ディレクティブが--describe-moduleの出力に含まれるのは、単に、すべてのモジュールがjava.baseに依存していることを示すためです。

モジュールの展開済みフォルダからアプリを実行する：モジュールの展開済みフォルダからWelcomeアプリを実行するためには、WelcomeAppフォルダから次のコマンドを使用します（ここでも、macOS/Linuxユーザーは^を\に置き換えてください）。

```
java --module-path mods ^
  --module com.deitel.welcome/com.deitel.welcome.Welcome
```

--module-pathオプションでは、モジュールが格納されている場所を指定します。この場合は、modsフォルダになります。--moduleオプションでは、モジュール名と、アプリのエントリ・ポイントとなるクラス（mainを含むクラス）の完全修飾名を指定します。プログラムが実行されると、次の結果が表示されます。

Welcome to the Java Platform Module System!

先ほどのコマンドで、--module-pathは-p、--moduleは-mという省略形を使用できます。

モジュールをモジュールJARファイルにパッケージ化する：jarコマンドを使用すると、展開済みモジュール・フォルダをモジュールJARファイルにパッケージ化することができます。このファイルには、module-info.classファイルを含むモジュールの全ファイルが含まれます。module-info.classファイルは、JARのルート・フォルダに格納されます。アプリを実行するときは、モジュール・パスにあるJARファイルを指定します。jarコマンドを実行する前に、JARファイルを出力するフォルダが存在している必要があります。

モジュールにアプリのエントリ・ポイントが含まれている場合、jarコマンドの--main-classオプションでそのクラスを指定できます。次に例を示します。

```
jar --create -f jars/com.deitel.welcome.jar ^
  --main-class com.deitel.welcome.Welcome ^
  -C mods/com.deitel.welcome .
```

各オプションの意味は、次のとおりです。

- --createでは、新しくJARファイルを作成することを指定しています。
- -fでは、JARファイルの名前を指定しています。この例では、jarsという名前のフォルダにファイルcom.deitel.welcome.jarが作成されます。
- --main-classでは、アプリのエントリ・ポイント（mainメソッドを含むクラス）の完全修飾名を指定しています。
- -Cでは、JARファイルに含めるファイルが格納されているフォルダを指定しています。ドット（.）は、フォルダ内のすべてのファイルを含めることを示しています。

モジュールJARファイルからWelcomeアプリを実行する：エントリ・ポイントを指定してモジュールJARファイルにアプリを格納した後は、次のようにしてアプリを実行できます。

```
java --module-path jars com.deitel.welcome
```

省略形を用いて次のように記述することもできます。

```
java -p jars -m com.deitel.welcome
```

JARの作成時にエントリ・ポイントを指定しなかった場合でも、モジュール名と完全修飾クラス名を指定してアプリ

を実行できます。次に例を示します。

```
java --module-path jars ^
  com.deitel.welcome/com.deitel.welcome.Welcome
```

次のように記述することもできます。

```
java -p jars -m com.deitel.welcome/com.deitel.welcome.Welcome
```

クラスパスとモジュール・パス：Java 9より前のコンパイラとランタイムは、クラスパスを使って型を探していました。クラスパスは、コンパイル済みのJavaクラスを含むフォルダおよびライブラリ・アーカイブ・ファイルのリストです。以前のバージョンのJavaでのクラスパスは、CLASSPATH環境変数、JREの特殊なフォルダに格納された拡張機能、javacコマンドおよびjavaコマンドのオプションを組み合わせて定義されていました。

この仕組みでは、複数の場所から型をロードできたため、それぞれの場所を検索する順番がアプリの不安定さにつながることがありました。たとえば、何年も前に、筆者はサード・パーティ・ベンダー製のJavaアプリを自分のシステムにインストールしたことがあります。そのアプリのインストーラは、古いバージョンのサード・パーティ製JavaライブラリをJREの拡張機能フォルダに格納するものでした。そのアプリ以外に、筆者のシステムのJavaアプリで、そのライブラリの新しいバージョンに依存するものが複数ありました。新しいバージョンのライブラリでは、新しい型の追加や、古いライブラリの型の拡張が行われていました。しかし、JREの拡張機能フォルダ内にあるクラスは、クラスパスの他のクラスよりも先にロードされます。そのため、新しいバージョンのライブラリに依存するアプリを実行すると、NoClassDefFoundErrorやNoSuchMethodErrorが発生して動作が停止するようになりました。場合によっては、アプリが実行を始めてかなり経過してから停止することもありました（クラスのローディングについて詳しくは、「Understanding Extension Class Loading」を参照してください）。

モジュールおよびモジュール記述子で得られる信頼性の高い構成は、実行時におけるそのような多くのクラスパス問題の解決に役立ちます。また、すべてのモジュールで依存性が明示的に宣言されるようになり、その依存性もアプリの起動時に解決されます。モジュール・パスにモジュールを1つだけ格納し、すべてのパッケージを1つの

モジュールおよびモジュール記述子で得られる信頼性の高い構成は、実行時における多くのクラスパス問題の解決に役立ちます。

モジュール内で定義することも可能です。複数のモジュールが同じ名前である場合や、同じパッケージをエクスポートしている場合、ランタイムはプログラムを実行せずに、即座に終了します。</article>

Paul Deitel：Deitel & AssociatesのCEO兼最高技術責任者。MITの卒業後、コンピュータ業界で
35年働いている。Java Championであり、22年を超えるJavaプログラミング経験を持つ。共著者のHarvey M. Deitel博士とともに、世界でベストセラーとなっているプログラミング言語関連の本を執筆している。世界の産業界、政府、学術機関のクライアントに向けて、Java、Android、iOS、C#、C++、C、インターネットの各プログラミング・コースを開講している。

[編集注：本記事は、最近出版されたPaul Deitel氏およびHarvey Deitel氏の共著書『Java 9 for Programmers』から抜粋したものです。今後のJava Magazineに書評が掲載される予定です。]

learn more

Project Jigsaw：Module System Quick-Start GuidePaul Deitel氏のライブ・オンライン・コース「Introduction to Modularity with the Java 9 Platform Module System（JPMS）」を、SafariBooksOnline.comのサブスクライバーは追加費用なしで利用できます。

Sander MakおよびPaul Bakker著『Java 9 Modularity:Patterns and Practices for Developing Maintainable Applications』（O'Reilly Media、2017年）

EDINBURGH JUG

スコットランドの首都エディンバラは、エディンバラ・フェスティバルと世界遺産で有名ですが、テクノロジーの街でもあります。イギリスの中でも特に技術労働者が集中している場所であり、3つの大学、多数の金融サービス企業、イギリス最大の技術系起業支援会社、そして活発な新興企業コミュニティが存在しています。もちろん、そういった企業の多くがJavaをベースにしています。

もともとのEdinburgh Java User Group（EdJug）は、スコットランド中のJava開発者を集めるために2011年に設立されました。やがてその活動は停滞していきましたが、2014年に、エディンバラの地元の技術コミュニティに対象を絞って再始動しました。

2014年以来、JavaとJVMに関心を持つプロの開発者や学生などにより、EdJugの会員数は80ほどに増加しています。毎月最終水曜日にミーティングを開催しており、地元の講師や世界中の専門家を招いて講演が行われています。最近では、静的分析、Androidのセキュリティ、プロパティベースのテストなどのテーマが取り上げられました。

このJUGは、スポンサーの協力も得て、週末の無料トレーニング・コースなどの1回限りのイベントも開催しています。今までのコースでは、Scala、Clojure、Java 8などを扱っています。

近隣のグラスゴーやセント・アンドリューズとともに、エディンバラは、関数型プログラミングの専門家が集まる国際的な中心地の1つとなっています。JUGの活動にもこれが反映されており、地元の関数型プログラミング・グループとの協力のもと、HaskellやIdrisなどの言語のトレーニングやワークショップが開催されています。

RAOUL-GABRIEL URMA

RICHARD WARBURTON

Java 9で更新されたコア・ライブラリ：OptionalとCompletableFuture

OptionalとCompletableFutureの変更により、ビジネス・アプリケーションのエラー・ケースをより的確にモデリング

Java 9の変更点を取り上げた、前号の記事「Java 9で更新されたコア・ライブラリ：CollectionとStream」では、開発者の日常作業の生産性を向上させる多くの魅力的な機能について紹介しました。Java 9に含まれているのは、モジュールやJava 9 Read-Eval-Print Loop（REPL）などの大がかりな改善ばかりではありません。本記事では、コア・ライブラリの主な変更点に関する説明の締めくくりとして、OptionalとCompletableFutureに関するAPIの改善について取り上げます。

Optional

Optionalは、効率的にストリームを扱えるようにJava 8で導入された機能です。この機能もJava 9で更新されており、今回のリリースでは、以下で紹介するstream()、ifPresentOrElse()、or()が導入されています。

stream()： Java 8のStream APIをOptionalクラスと組み合わせて使ったことがある方なら、Optionalのストリームを値で置換したいと思う状況に遭遇したことがあるのではないでしょうか。たとえば、ユーザーが指定済みかもしれない設定のコレクションがあるとします。そして、ユーザーが設定を指定済みの場合にOptional<Setting>を返す、次のlookupSettingByName()メソッドが実装されているものとします。

```
List<Setting> settings =
  SETTING_NAMES.stream()
          .map(this::lookupSettingByName)
          .filter(Optional::isPresent)
          .map(Optional::get)
          .collect(toList());
```

この例では、Optionalの2つのメソッドを組み合わせて、やりたいことを実現しています。まず、空のOptionalを削除

するために、isPresent()メソッドでfilterをかけています。次に、get()メソッドを呼び出すことで値が存在するとわかったOptionalオブジェクトをアンボクシングしています。

Java 9では、このコードをもう少しきれいに書くことができます。Java 9のOptionalには、ストリームを返すメソッドが追加されています。おもしろいことに、その名はstream()です。このメソッドは、Optionalに要素があればその要素を含むStreamを、そうでなければ空のStreamを返します。この手法を使ったコードを見てみます。

```
List<Setting> settings =
  SETTING_NAMES.stream()
          .map(this::lookupSettingByName)
          .flatMap(Optional::stream)
          .collect(toList());
```

この機能が追加されたことにより、ストリームを扱うAPIでOptionalを使いやすくなります。

ifPresentOrElse()：新しく追加されたifPresentOrElseメソッドを使うと、Optionalに値が存在する場合にはある操作を、そうでない場合は別の操作を行うという、よくあるパターンをコードで表現できます。この機能を理解するために、飛行機にチェックインしようとする場合を例にとって、最初はnullチェックを、次にOptional型を使って、どのようにこのコードを書くか考えてみます。

まず、ユーザーから提示された予約を検索します。提示された予約が見つかればチェックインのページを表示し、見つからない場合は予約レコードが存在しない旨が記載されたページを表示します。

予約が見つからない場合にlookupBookingメソッドがnullを返すものとしたとき、チェックインのコードは次のようになるでしょう。

```
Booking booking = lookupBooking(bookingRef);
if (booking != null) {
  displayCheckIn(booking);
} else {
  displayMissingBookingPage();
}
```

Java 8では、Optional値を返すようにlookupBookingをリファクタリングしたかもしれません。この場合、予約が見つかったかどうかが明示的に示されることになります。さらに、Optionalを使うことで、予約が存在するときとしな

いときを明確に分けて考えることができます。この手法がnullチェックよりも優れているのは、明示的にドメイン・モデルに従って考えざるを得なくなるからです。もっとも単純な形でこのコードをリファクタリングすると、次のようになるでしょう。

```
Optional<Booking> booking = lookupBooking(bookingRefer);
If (booking.isPresent()) {
   displayCheckIn(booking.get());
} else {
   displayMissingBookingPage();
}
```

ここで使われているパターンは、Optionalを受け取ってisPresent()とget()を呼び出すというものです。しかし、コードが冗長になるため、あまり慣用的な方法とは言えません。しかも、このコードはnullチェックを行うコードと本質的に同じようなものです。Optionalに対して、ユースケースにふさわしいメソッドを呼び出せると理想的です。そうすれば、実質的に「求めるな、命じよ」というスタイルのコーディングに移行できます。Java 8のOptionalには、Optionalの内部に値が存在する場合にコールバックを呼び出すifPresentメソッドがあります。次に例を示します。

```
lookupBooking(bookingReference)
   .ifPresent(
      this::displayCheckIn);
```

残念なことに、これでは今回のニーズは満たされません。値が存在しないことにより「予約が存在しません」ページを表示する場合を処理しないからです。このようなユースケースでは、Java 9で改善されたOptionalが役立ちます。元のコードは、次のようにリファクタリングできます。

```
lookupBooking(bookingReference)
   .ifPresentOrElse(
      this::displayCheckIn,
      this::displayMissingBookingPage);
```

or()：Java 9でOptionalに追加されたもう1つのメソッドは、簡潔な名前のor()メソッドです。このメソッドは、Optionalを作成する関数を引数として受け取ります。呼出しの対象となるオブジェクトに値が存在する場合、その値が返されます。そうでない場合は、引数として渡された関数が呼び出され、その結果が返されます。

これは、Optionalを返すメソッドが複数あり、値が存在する最初のOptionalを返したい場合に特に便利です。会社番号などの識別子を使って顧客情報を検索したい場合を考えてみます。まず、既存の顧客データストアをチェックし、その会社がすでに登録されているかどうかを確認します。そこに存在しない場合、別のサービスでその会社についての情報を検索し、新しい顧客を作成します。

提供されたIDはユーザーのタイプミスである可能性もあります。その場合、IDの検索は成功しません。値が存在しないことを示すためにメソッドからnullを返す場合、次のようなコードを書くことになるでしょう。

```
Client client = findClient(companyId);
if (client == null) {
  client = lookupCompanyDetails(companyId);
}
// ここでもclientはnullの可能性がある
```

findClient()をリファクタリングしてOptionalを返すようにする場合、Java 8のorElseGet()メソッドを使うことができます。そうすると、Optionalに値が存在しない場合にのみlookupCompanyDetails()メソッドを呼び出すことになります。次に例を示します。

```
Client client =
  findClient(companyId)
  .orElseGet(() ->
      lookupCompanyDetails(companyId));
```

しかし、これでもまだ今回のユースケースが正確にモデリングされているわけではありません。companyIdが実際の会社の識別子に対応していない場合があるため、依然としてlookupCompanyDetails()メソッドが失敗する可能性があります。Optional型を使ってモデリングの失敗を表そうとするなら、そのメソッドもOptionalを返すようにするべきです。

そんなときに力を発揮するのが新しいor()メソッドです。このメソッドを使って、Optional<Client>を返すことができます。データベース内の顧客の検索が成功すれば、存在する値がその顧客になります。有効な新しい会社であれば、その会社が返されます。タイプミスだった場合、Optionalは空になります。次にそのコードを示します。

```
Optional<Client> client =
  findClient(companyId)
  .or(() -> lookupCompanyDetails(companyId));
```

ここまでは、Java 8のOptionalがJava 9でどのように改善されたかについて見てきました。欠けていたユースケースに対応できるように、的を絞った一連のメソッドが追加されています。なお、プリミティブに特化したOptionalクラス（OptionalIntなど）は、同じ扱いにはなっていません。それらのクラスには、stream()とifPresentOrElse()は追加されているものの、or()は追加されていません。

CompletableFuture

Java 8では、Future<T>の拡張としてCompletableFuture<T>が導入されました。このクラスを使うと、コールバック駆動スタイルを用いて、別々のタスクからの情報の流れを表現できます。本記事の以降の部分では、Java 9でのFutureの機能強化について説明します。最初に、1つの典型的な問題を紹介します。そして、Java 8でその問題に対処するいくつかの方法を紹介した後、最後のセクションで、Java 9を使ったより優れた解決策について説明します。

2つのサービスの組合せ：ネットワーク経由で使用する2つのサービスの結果を組み合わせる場合を考えてみます。1つのサービスは航空運賃の最適価格を探すもの、もう1つは米ドル（USD）を英ポンド（GBP）に換算するものです。いずれのサービスも、結果を返すまで多少の遅延が発生します。この遅延は、サービスとのネットワーク通信の負荷によるものです。

この問題は、次のようにCompletableFutureを利用して解決することができます（上級者向けの注：デフォルトで、CompletableFutureは共通スレッド・プールを使いますが、オーバーロードされたsupplyAsyncを使ってパラメータにExecutorを指定することもできます）。

```
BigDecimal amount =
  CompletableFuture.supplyAsync(
```

```
    () -> findBestPrice("LDN - NYC"))
  .thenCombine(CompletableFuture.supplyAsync(
    () -> queryExchangeRateFor("GBP")),
  this::convert)
  .get();
```

このコードで、convertメソッドはfindBestPriceとqueryExchangeRateForの結果として2つのBigDecimalを受け取り、最終的な価格を計算します。

タイムアウト・メカニズム：上記のコードには、いくつかの問題があります。1点目は、get()がブロッキング・コールであることです。つまり、メイン・スレッドは結果の準備が整うまで待機する必要があり、続きの実行が可能になるのはその後になります。理想を言えば、バックグラウンドで結果の計算を行っている間、メイン・スレッドには別の実用的な仕事をさせる方が望ましいでしょう。2点目は、タイムアウトが指定されていないため、メイン・スレッドが無限にブロックされる可能性があることです。いずれかのサービスが過負荷のため応答しない場合を考えてみてください。タイムアウト・メカニズムを追加するためには、Futureから継承された別バージョンのgetを使用します。このメソッドは、結果を返すためにパイプライン全体でかかった時間が、指定した長さを超えた場合、TimeoutExceptionをスローします。

```
BigDecimal amount =
  CompletableFuture.supplyAsync(
    () -> findBestPrice("LDN - NYC"))
  .thenCombine(CompletableFuture.supplyAsync(
    () -> queryExchangeRateFor("GBP")),
        this::convert)
  .get(1, TimeUnit.SECONDS);
```

しかし、手直ししたこのコードでもブロックが発生するため、メイン・スレッドに同時に別の実用的な仕事をさせることはできません。この問題に対処するために、thenAcceptを使って、結果を使用できるようになったときに実行されるコールバックを渡すようにコードをリファクタリングすることができます。次のコードの最終行に注目してください。

```
CompletableFuture.supplyAsync(
      () -> findBestPrice("LDN - NYC"))
   .thenCombine(CompletableFuture.supplyAsync(
      () -> queryExchangeRateFor("GBP")),
         this::convert)
   .thenAccept(amount ->
      System.out.println(
         "The price is: " + amount + "GBP"));
```

ただし、このアプローチを使った場合、タイムアウト機能が失われてしまいます。ノンブロッキング・メソッドを使いつつ、タイムアウトも指定できるのが理想的です。Java 8には、この問題を解決するエレガントな手法は組み込まれていません。Java 8で使えるソリューションは、少しばかりややこしい方法です。たとえば、CompletableFutureと合わせてacceptEither()またはapplyToEitherを使い、さらに別のCompletableFutureを使って結果を待機するというものがあります。そのCompletableFutureは、指定した時間が経過した後にTimeoutExceptionをスローするScheduledThreadPoolExecutorをラップします。

```
CompletableFuture.supplyAsync(
  () -> findBestPrice("LDN - NYC"))
.thenCombine(CompletableFuture.supplyAsync(
  () -> queryExchangeRateFor("GBP")),
  this::convert)
.acceptEither(
  timeoutAfter(1, TimeUnit.SECONDS),
  amount -> System.out.println(
    "The price is: " + amount + "GBP"));
```

timeoutAfterのシンプルな実装は次のようになります。変数delayerには、ScheduledThreadPoolExecutorのインスタンスを格納します。

```
public <T> CompletableFuture<T> timeoutAfter(
```

```
    long timeout,
    TimeUnit unit) {
      CompletableFuture<T> result =
        new CompletableFuture<T>();

    delayer.schedule(() ->
      result.completeExceptionally(
        new TimeoutException()), timeout, unit);
    return result;
}
```

timeoutAfterメソッドは、指定した時間が経過した後にTimeoutExceptionをスローするスケジュールが設定されたCompletableFutureを返します。その後、このメソッドとacceptEitherメソッドを組み合わせて、最初に完了したCompletableFutureを選択します。つまり、実際のサービスまたはタイムアウトのいずれかが選択されます。ただ、おわかりのように、これを実装するためには大量のコードを書く必要があります。

Java 9での改善：Java 9のCompletableFutureには、いくつかの新しいメソッドが導入されています。その中に、orTimeoutやcompleteOnTimeoutがあります。タイムアウトの処理が組込みでサポートされているため、自前で仕組みを実装する必要はありません。

orTimeoutメソッドのシグネチャは次のようになっています。

```
public CompletableFuture<T> orTimeout(
  long timeout, TimeUnit unit)
```

これは、内部的にScheduledThreadExecutorを使用しており、指定したタイムアウト時間が経過するとTimeoutExceptionをスローしてCompletableFutureが完了します。また、このメソッドは別のCompletableFutureを返します。つまり、さらに計算パイプラインを連鎖させて、TimeoutExceptionが発生した場合にはわかりやすいメッセージを表示できます。次に例を示します。

```
CompletableFuture.supplyAsync(
    () -> findBestPrice("LDN - NYC"))
```

```
    .thenCombine(CompletableFuture.supplyAsync(
        () -> queryExchangeRateFor("GBP")),
          this::convert)
    .orTimeout(1, TimeUnit.SECONDS)
    .whenComplete((amount, error) -> {
        if (error == null) {
            System.out.println(
                "The price is: " + amount + "GBP");
        } else {
            System.out.println(
              "Sorry, we could not return you a result");
  }
});
```

completeOnTimeoutメソッドのシグネチャは次のようになっています。

```
public CompletableFuture<T> completeOnTimeout(
   T value, long timeout, TimeUnit unit)
```

こちらも内部的にScheduledThreadExecutorを使っていますが、orTimeoutとは異なり、CompletableFutureパイプラインがタイムアウトした場合に使用されるデフォルト値を指定します。考え方としては、orTimeoutメソッドは、java.util.OptionalのorElse()メソッドに似ています。

次のコードでは、サービスがなかなか応答しない場合、デフォルトの価格が使われます。そうでない場合は、2つのサービスから返された値を組み合わせた結果が使われます。

```
CompletableFuture.supplyAsync(
      () -> findBestPrice("LDN - NYC"))
   .thenCombine(CompletableFuture.supplyAsync(
      () -> queryExchangeRateFor("GBP")),
          this::convert)
   .completeOnTimeout(DEFAULT_PRICE, 1, SECONDS)
   .thenAccept(amount -> {
```

```
    System.out.println(
        "The price is: " + amount + "GBP");
});
```

まとめ

Java 9では、OptionalおよびCompletable Futureデータ型を使用する場合に見られる、いくつかの煩雑なパターンを回避できるようになっています。そういった変更点を活用すれば、実際のビジネス・アプリケーションで扱う必要があるエラー・ケースをより的確にモデリングできるようになり、信頼性がより高いコードを書けるようになります。Optionalの改善によって、以前は難しかった方法でOptionalを使えるようになっています。また、CompletableFutureクラスを使って、外部操作のタイムアウトをとても簡単に実装できるようになっています。</article>

Raoul-Gabriel Urma（@raoulUK）：イギリスのデータ・サイエンティストや開発者の学習コミュニティをリードするCambridge SparkのCEO/共同創業者。若いプログラマーや学生のコミュニティであるCambridge Coding Academyの会長/共同創設者でもある。ベストセラーとなったプログラミング関連書籍『Java 8 in Action』（Manning Publications、2015年）の共著者として執筆に携わった。ケンブリッジ大学でコンピュータ・サイエンスの博士号を取得している。

Richard Warburton（@richardwarburto）：Java Championであり、ソフトウェア・エンジニアの傍ら、講師や著述も行う。ベストセラーとなった『Java 8 Lambdas』（O'Reilly Media、2014年）の著者であり、Iteratr LearningとPluralsightで開発者の学習に貢献し、数々の講演やトレーニング・コースを実施している。ウォーリック大学で博士号を取得している。

1段組みの新しい記事レイアウトはお気に召しましたか?
ぜひ、JAVAMAG_US@ORACLE.COMまでご意見をお寄せください。

BEN EVANS

Javaにおけるメソッド呼出しの仕組み

特殊なバイトコードでメソッドの呼出しを大幅に効率化。その仕組みを理解することで、JVMがコードを実行する方法を解き明かす

本記事では、Java 8とJava 9でJVMがメソッドを実行する方法について説明します。このトピックは、JVMの内部に関する基本的な内容です。JVMのJust-In-Time（JIT）コンパイラを理解したい方や、アプリケーションの実行をチューニングしたい方にとって、欠かせない背景知識です。

バイトコードの内部

まずは、簡単なJavaコードを見てみます。

```
long time = System.currentTimeMillis();
HashMap<String, String> hm = new HashMap<>();
hm.put("now", "bar");
Map<String, String> m = hm;
m.put("foo", "baz");
```

Javaコンパイラがこのコードから生成したバイトコードを確認するためには、-cスイッチを付けてjavapツールを使用します。実行すると、逆コンパイルされたコードが表示されます。

```
0: invokestatic #2 // Method
                   java/lang/System.currentTimeMillis:()J
3: lstore_1
4: new #3 // class java/util/HashMap
7: dup
```

写真：JOHN BLYTHE

```
8: invokespecial #4 // Method java/util/HashMap."<init>":()V
11: astore_3
12: aload_3
13: ldc #5 // String now
15: ldc #6 // String bar
17: invokevirtual #7 // Method java/util/HashMap.put:
            (Ljava/lang/Object; Ljava/lang/Object;)
            Ljava/lang/Object;
20: pop
21: aload_3
22: astore 4
24: aload 4
26: ldc #8 // String foo
28: ldc #9 // String baz
30: invokeinterface #10, 3 // InterfaceMethod java/util/Map.put:
            (Ljava/lang/Object;Ljava/lang/Object;)
            Ljava/lang/Object;
35: pop
```

[編集注：インデントされた行は、前の行の続きです。] 実際のJavaのメソッド呼出しは、何種類かある、invoke*という形式のバイトコードの1つに変換されます。JVMレベルのコードを見るのは初めてだというJavaプログラマーの方は、このことを知って驚かれるかもしれません。

それでは、逆コンパイルされたコードの最初の部分を詳しく見てみます。

```
0: invokestatic #2 // Method java/lang/System.currentTimeMillis:()J
3: lstore_1
```

System.currentTimeMillis()メソッドの静的呼出しは、invokestatic命令コードに変換され、バイトコードの0の位置に格納されています。このメソッドにはパラメータがないため、呼出し前に評価スタックにロードするものはありません。

次に、バイト・ストリームに2つのバイト00 02が登場します。この2つは組み合わされて16ビット数値（この場合は#2）となり、クラス・ファイル内の表（この表を定数プールと呼びます）に対するオフセットとして使用されます。定数プールのインデックスはすべて16ビットであるため、命令コードがプール内のエントリを参照する場合は常に、エントリのオフセットをエンコードした連続する2バイトが必要です。

便利なことに、逆コンパイラはオフセット#2に対応するメソッドが何かを示すコメントを追加します。今回の例では、予想どおりSystem.currentTimeMillis()メソッドとなっています。javapでは、逆コンパイルされた出力の中に、呼び出されたメソッドの名前、メソッドが受け取るパラメータの型（括弧内）、メソッドの戻り型が表示されます。

呼出し先から戻る際には、呼出しの結果がスタックに配置されます。オフセット3の位置には、引数のない単独の命令コードlstore_1があります。この処理は、戻り値をlong型のローカル変数に保存するものです。

もちろん、人間が読めば、この値は二度と使われないことを理解できます。しかし、Javaコンパイラにおける設計上の目標の1つは、論理的に意味をなすかどうかにかかわらず、Javaソース・コードの内容をできる限り忠実に表現する点にあります。そのため、System.currentTimeMillis()の戻り値は、この後のプログラムで使われることがなくても保存されます。

それでは、逆コンパイルされたコードの次の部分を見てみます。

```
4: new #3 // class java/util/HashMap
7: dup
8: invokespecial #4 // Method java/util/HashMap."<init>":()V
11: astore_3
12: aload_3
13: ldc #5 // String now
15: ldc #6 // String bar
17: invokevirtual #7 // Method java/util/HashMap.put:
                (Ljava/lang/Object;Ljava/lang/Object;)
                Ljava/lang/Object;
20: pop
```

バイトコード4から10では、新しいHashMapのインスタンスを作成しています。その後の命令11では、そのコピーをローカル変数に保存しています。次に、命令12から16で、HashMapオブジェクトと、put()の呼出しに使う引数が配置されたスタックを準備しています。put()メソッドの実際の呼出しは、命令17から19で行っています。

今回使われているinvoke命令コードは、invokevirtualです。これは、メソッド呼出しの対象となるインスタンス（レシーバ・オブジェクトと呼ばれることもあります）が存在しない静的呼出しとは異なります（インスタンスを呼び出すバイトコードでは、レシーバと、呼出しに使うすべての引数を評価スタックに配置してから、invoke命令を発行する必要があります）。今回の例では、put()の戻り値は使われていないため、命令20で戻り値を破棄しています。

21から25までのバイト列は、最初は少し奇妙に見えるかもしれません。

```
21: aload_3
22: astore 4
24: aload 4
26: ldc #8 // String foo
28: ldc #9 // String baz
30: invokeinterface #10, 3 // InterfaceMethod java/util/Map.put:
                    (Ljava/lang/Object;Ljava/lang/Object;)
                    Ljava/lang/Object;
35: pop
```

バイトコード4で作成され、命令11でローカル変数3に保存されたHashMapインスタンスが、ここでスタック上に再ロードされ、その参照のコピーがローカル変数4に保存されています。この処理によって、インスタンスがスタックから削除されるため、このインスタンスを使う前に（変数4から）再ロードする必要があります。

このような変数のシャッフルが起こるのは、元のJavaコードで、追加のローカル変数（HashMap型ではなくMap型）が作成されているためです。ただし、この変数は常に元の変数と同じオブジェクトを参照しています。これは、バイトコードが元のソース・コードにできる限り近くなっているもう1つの例と言えます。Javaがこの「単純なバイトコード」と言われるアプローチを採用している主な理由の1つは、JVMのJITコンパイラへの入力フォーマットを簡単なものにすることです。

スタックと変数がシャッフルされた後、命令26から29でMap内に配置される値がロードされます。これでレシーバと引数が格納でき、スタックの準備ができました。続く命令30では、put()が呼び出されます。今回の命令コー

ドはinvokeinterfaceですが、実際に呼び出されるメソッドは同じです。ここでも、命令35のpopでput()の戻り値は破棄されています。

ここまでの部分で、Javaコンパイラは呼出しの状況に応じてinvokestatic、invokevirtual、invokeinterfaceを生成し分けていることがおわかりいただけたかと思います。

メソッド呼出しに使われるJVMバイトコード

それでは、メソッド呼出しに使われる可能性がある5つのJVMバイトコードすべてについて見ていきます（表1参照）。いずれのケースでも、バイトコードb0とb1が組み合わされて定数プールのオフセットc1となります。

命令コード名	引数	説明
invokevirtual	b0 b1	CP#C1に見つかったメソッドを仮想ディスパッチで呼び出す
invokespecial	b0 b1	CP#C1に見つかったメソッドを「特殊」な（正確な）ディスパッチで呼び出す
invokeinterface	b0 b1 x0 00	CP#C1に見つかったインタフェース・メソッドをインタフェース・オフセット検索で呼び出す
invokestatic	b0 b1	CP#C1に見つかった静的メソッドを呼び出す
invokedynamic	b0 b1 00 00	呼び出すメソッドを動的に検索して呼び出す

表1：メソッド呼出しに使われるJVMバイトコード

実際にJavaコードを書いて生成されたJavaクラスをjavapで逆アセンブルし、それぞれの形式のバイトコードがどのような状況で生成されるのかを確認してみると、よい演習になるでしょう。

もっとも一般的なメソッド呼出しは、仮想ディスパッチを示すinvokevirtualです。仮想ディスパッチという用語は、呼び出されるメソッドが実行時に完全に決定されることを意味しています。これを理解するためには、実行中のアプリケーション内に存在する各クラスに対して、その型に対応するメタデータを格納するメモリ領域がJVM内に確保されていることを知る必要があります。この領域は、（少なくとも、Java HotSpot VMでは）klassと呼ばれており、JVMでの型情報の表現と考えることができます。

Java 7以前では、klassメタデータは、permgenと呼ばれるJavaヒープに格納されていました。Javaヒープ内の

オブジェクトには、必ずオブジェクト・ヘッダー（oopと呼ばれています）が存在するため、klassはklassOopと呼ばれていました。Java 8とJava 9では、klassメタデータがJavaヒープからネイティブ・ヒープに移動したため、オブジェクト・ヘッダーは不要になっています。Javaプログラマーは、型に対応するClass<?>オブジェクトを使ってklassの情報の一部を参照できますが、これは別の概念です。

klass内の領域で特に重要なのがvtableです。実質的に、この領域は、その型で定義されているメソッドの実装を指す関数ポインタの表だと言えます。invokevirtualでインスタンス・メソッドが呼び出されると、JVMはvtableを参照して、実行する必要があるコードを正確に特定します。klassにメソッドの定義がない場合、JVMはスーパークラスに対応するklassへのポインタをたどり、再度呼出しを試みます。

これがJVMでのメソッドのオーバーライド処理の基本となりますが、この処理を効率的に行うことができるように、vtableは決まった方法でレイアウトされています。各klassのvtableは、親の型で定義されているメソッドが最初に来るようになっており、親の型とまったく同じ順番でレイアウトされます。親クラスで宣言されておらず、その型で新しく追加されたメソッドはvtableの最後に配置されます。

つまり、サブクラスでメソッドをオーバーライドした場合、オーバーライドした実装とオーバーライドされた実装でvtableのオフセットが同じ値になります。この仕組みによって、vtableのオフセットが親のオフセットと同じになるため、オーバーライドされたメソッドのルックアップがとても簡単になります。

図1は、Pet、Cat、Bearの各クラスとFurryインタフェースが定義されている例です。

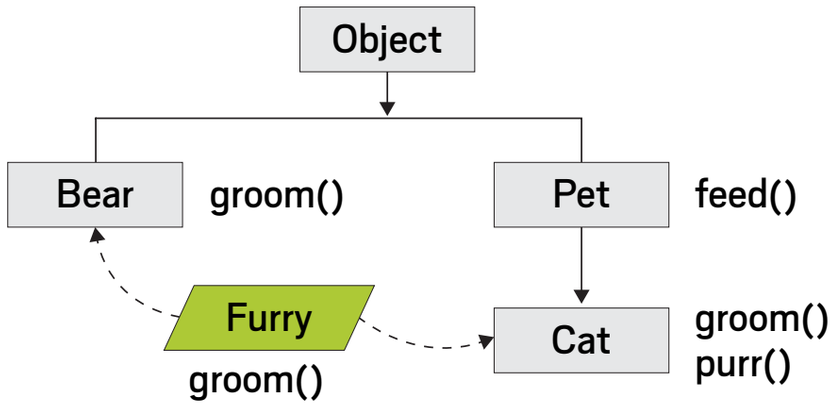


図1：単純な継承階層

Java 7では、これらのクラスのvtableは図2のようなレイアウトになります。なお、この図はJava 7のpermgen内のレイアウトを示しているため、参照先はklassOopとなっています。そのため、2ワードのオブジェクト・ヘッダー（図では、mおよびkkで示されています）が存在します。前述のとおり、Java 8とJava 9ではこのエントリは存在しませんが、その点を除けば、同じ図が適用されます。

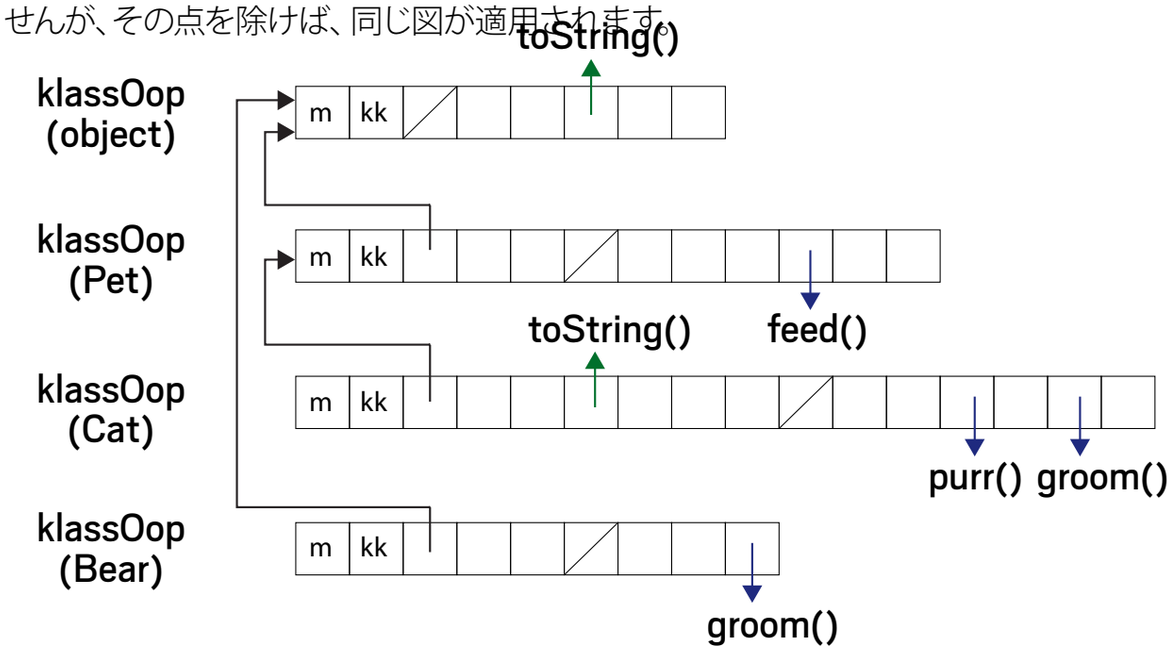


図2： 図1に示したクラスのvtableの構造

Cat::feedを呼び出す場合を考えてみます。JVMは、Catクラスでオーバーライドされたメソッドを見つけられないため、Petのklassへのポインタをたどります。このklassにはfeed()の実装が存在するため、そのコードが呼び出されることになります。このvtable構造がうまく動作するのは、Javaがクラスの単一継承しか実装していないためです。つまり、どの型にも、直接的なスーパークラスは1つしかありません（スーパークラスのないオブジェクトは例外です）。

invokeinterfaceが使われる場合は、もう少し複雑になります。たとえば、すべてのFurryの実装でgroom()メソッドが必ずしもvtableの同じ場所にあるとは限りません。Cat::groomとBear::groomでオフセットが異なるのは、それぞれのクラスの継承階層が異なるためです。そのため、コンパイル時にインタフェース型しかわからないオブジェクトのメソッドを呼び出す場合、追加でルックアップを行う必要があります。

インタフェースの呼出しの際に、追加のルックアップ作業が若干必要にはなりますが、インタフェースの使用を

避けて微少な最適化を試みるべきではないという点に注意してください。JVMにはJITコンパイラが搭載されているため、この2つのケースに実質的なパフォーマンスの差はないことを覚えておいてください。

メソッド呼出しの例

別の例を見てみます。次のコードについて考えてみます。

```
Cat tom = new Cat();
Bear pooh = new Bear();
Furry f;

tom.groom();
pooh.groom();

f = tom;
f.groom();

f = pooh;
f.groom();
```

このコードから、次のバイトコードが生成されます。

```
Code:
0: new          #2          // class scratch/Cat
3: dup
4: invokespecial #3          // Method scratch/Cat."<init>":()V
7: astore_1
8: new          #4          // class scratch/Bear
11: dup
12: invokespecial #5         // Method scratch/Bear."<init>":()V
15: astore_2
16: aload_1
```

```
17: invokevirtual #6          // Method scratch/Cat.groom:()V
20: aload_2
21: invokevirtual #7          // Method scratch/Bear.groom:()V
24: aload_1
25: astore_3
26: aload_3
27: invokeinterface #8,  1     // InterfaceMethod scratch/Furry.groom:()V
32: aload_2
33: astore_3
34: aload_3
35: invokeinterface #8,  1     // InterfaceMethod scratch/Furry.groom:()V
```

27と35で行われている2つの呼出しは同じように見えますが、実際には異なるメソッドを呼び出しています。27の呼出しではCat::groomが、35の呼出しではBear::groomが呼ばれます。

これまでに説明したinvokevirtualとinvokeinterfaceを知っておけば、invokespecialの動作が理解しやすくなるでしょう。invokespecialでメソッドが呼び出されると、JVMは仮想ルックアップを実行せずに、リクエストされたメソッドの正確な場所のみをvtable上で検索します。つまり、invokespecialは、privateメソッドの呼出し、スーパークラスのメソッドの呼出し、コンストラクタ本体の呼出し（バイトコードでは、<init>というメソッドに変換されます）という3つの場合に使用されます。3つの場合とも、オーバーライドされる可能性が明示的に排除されており、仮想ルックアップは必要ありません。

finalメソッド

もう1つ、触れておくべき特殊なケースが、finalメソッドです。一見したところでは、finalメソッドの呼出しもinvokespecial命令に変換されると思われるかもしれません。しかし、Java言語仕様のセクション13.4.17には、finalメソッドについて「finalと宣言されたメソッドをfinalと宣言されていないメソッドに変更しても、既存のバイナリとの互換性は失われない」と述べられています。

それでは、コンパイラがfinalメソッドの呼出しをinvokespecialにコンパイルした場合を考えてみます。メソッドがfinal以外に変更されると、サブクラスでオーバーライドされる可能性があります。ここで、コンパイルされたコードにサブクラスのインスタンスが渡されたとします。すると、invokespecialが実行され、誤ったメソッドの実装が

呼び出されることになります。これは、Javaのオブジェクト指向のルールに違反します（厳密に言えば、リスコフの置換原則に違反することになります）。

そのため、finalメソッドの呼出しはinvokevirtual命令にコンパイルされる必要があります。実際のJavaのHotSpot VMは、finalメソッドを検出して非常に効率よく実行できるように最適化されています。

まとめ

ここまでで、JVMがサポートする5つの呼出し命令のうち4つを説明してきました。残るinvokedynamicは、これだけで1本の記事が必要になるほど奥深く興味深いテーマです。2回シリーズの2回目となる次の記事では、invokedynamicとそれに関連する題材を扱いたいと思います。</article>

Ben Evans（@kittylyst）：Java ChampionおよびjClarityの創業者/技術フェローであり、London Java Community（LJC）主催者。Java SE/EE Executive Committeeのメンバーでもある。近刊の『Optimizing Java』（O'Reilly）を含む4冊の本を執筆している。

learn more

javap
JavaのJVM命令セット

ANDRÉS ALMIRAY

FXMLLoaderでFXMLのカスタム動作を定義する

FXMLを用いた、JavaFXアプリケーションへのカスタム動作の注入

本シリーズの1回目の記事では、FXMLフォーマットの基本のほか、FXMLLoaderという便利なユーティリティについて説明しました。名前からもわかるように、FXMLLoaderはFXMLリソースを読み込み、その内容を解析し、FXMLファイルに含まれる定義に基づいてSceneGraphを構築します。しかし、FXMLLoaderでできることはそれだけにとどまりません。特に、最近のJava開発で一般的に利用される技術である依存性注入（Dependency Injection /DI）と組み合わせれば、その可能性は大きく広がります。FXMLLoaderになじみがないという方は、本記事の内容が理解しづらい可能性があるため、1回目の記事をご覧になることをお勧めします。

設定の中には、プログラム的なアプローチで調整したいと開発者が考えるものもあります。FXMLフォーマットの設計者は、そのことを認識していました。たとえば、UIが作成された直後、ユーザーがアプリケーションを操作できるようになる前に、ListViewやTableViewに一連の動的な項目を挿入したい場合がその例です。このようなタスクを実行するためには、アプリケーションの初期化時に特定のタイミングで呼び出されるカスタム動作をJavaFX開発者が定義できることが必要です。また、カスタマイズの対象となるターゲット・ウィジェット（ListViewなど）を見つける方法も必要です。この動作は、最近のDIフレームワークが提供するものに非常によく似ています。FXMLの設計者が実現しているのは、まさにその機能です。ここからは、その仕組みについて見ていきます。

コントローラ

1回目の記事で紹介したJavaFXビューには、アプリケーション固有の動作はありませんでした。しかし、役割が分離されているという意味で、これはよいことだと言えます。ビューの要素はUIの外見を定義すべきですが、動作を定義すべきではありません。動作を定義するのは、コントローラの役割です。FXMLの用語では、DIに関与し、関連付けられたビューで定義されたUI要素がトリガーするイベントに反応できるオブジェクトをコントローラと呼びます。JavaFXアプリケーションのメイン・エントリ・ポイント

は、特定のクラス（javafx.application.Application）を拡張する必要があります。それとは違い、コントローラ・クラスは、どんな型でも構いません。また、実装する必要があるマーカー・インタフェースもありません。そのため、必要に応じて自由に型やコントローラ階層を作ることができます。

まずは、ボタンとListViewからなるUIを備えた単純なアプリケーションから始めます。リストの内容はプログラムで初期化し、ボタンをクリックするたびに新しい項目が追加されるものとします。このUIのFXML定義は次のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.ListView?>
<?import javafx.scene.layout.VBox?>
<VBox xmlns="http://javafx.com/javafx"
   xmlns:fx="http://javafx.com/fxml"
   fx:controller="sample.AppController"
   prefHeight="300.0" prefWidth="100.0">
   <Button text="Add item" onAction="#addItem"/>
   <ListView fx:id="theList"/>
</VBox>
```

このファイルでは、本シリーズの記事で初めて扱ういくつかの機能が使われています。まずは、UIのルート要素に適用されているfx:controller属性です。この属性は、指定されたクラスのオブジェクトのインスタンスを作成し、そのインスタンスをこのビューのコントローラに設定するようにFXMLLoaderに指示するものです。このようにインスタンス化されるオブジェクトのクラスには、引数なしのpublicなコンストラクタが必要です。

2つ目の機能は、ボタンの定義で使われているonActionという名前の属性です。この値のフォーマットに注目してください。文字#の後に識別子が記述されています。これは、コントローラ・クラスに定義されているメソッドです。

最後に、ListViewでのfx:idの使い方を確認してください。これは、属性名を変数名として用いてListViewへの内部参照を保持するようにFXMLLoaderに指示するものです。この変数は、コントローラにListViewへの参照を注入する際に使われます。次に、コントローラを見てみます。

```
// sample/AppController.java
package sample;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.ListView;

public class AppController {
   @FXML private ListView<String> theList;

   public void initialize() {
      for (int i = 0; i < 5; i++) {
         theList.getItems().add("Item " + i);
      }
   }

   public void addItem(ActionEvent ignored) {
      theList.getItems().add("Item "
         + theList.getItems().size());
   }
}
```

おわかりのように、コントローラは通常のJavaクラスのように見えます。ただし、本シリーズの記事では初めて扱う要素である@FXMLアノテーションが使われています。これは、FXMLLoaderがターゲット・コントローラのインスタンスにDIを行うことができるようにするために必要なリンクです。このアノテーションでは、ListViewを参照するために定義した変数名を利用しています。フィールド名が変数名と同じであり、ListViewに含めることができる要素の型（この場合は、単純なJava文字列）が定義されている点に注目してください。FXMLLoaderは、このネーミング規則に従ってフィールド名と変数名、およびフィールドの型とウィジェットの型を照合します。そのため、フィールドがChoicebox<String>として定義されている場合は、実行時例外が発生します。お使いのIDEがFXMLの規約を

認識するものであれば、こういったエラーは避けることができます。

2つ目の特徴は、引数がないinitializeという名前のpublicメソッドが定義されている点です。このメソッドは、すべてのウィジェットの注入が行われた後、FXMLLoaderによって呼び出されます。この時点で、ウィジェットの参照が完全に解決され、注入も完了しています。そのため、データの初期化には絶好のタイミングです。今回の例では、まさにこの処理によってListViewにデフォルトのデータが提供されています。メソッドが別の名前である場合、FXMLLoaderに呼び出されることはありません。これは、FXMLLoaderの規約によって定義されている強い制約です。

JSR 330では、Javaプログラムで依存性注入を実行する際の**基本原理が定義されています。**

最後に、少しだけアクション・ハンドラを見てみます。このハンドラは、ボタンに関連付けられているonActionハンドラの値に一致する名前のメソッドとして定義されています。FXMLファイルを見れば、おわかりいただけるでしょう。このハンドラは、ボタンのクリックに反応します。そのため、引数としてActionEventを受け取ります。この型は、コードを反応させたいイベントの種類によって変わるため、注意してください。たとえば、コードをマウスの移動に反応させたい場合は、引数の型はMouseEventにする必要があります。このイベント・ハンドラは、ボタンをクリックすると即時に呼び出され、UIスレッド内でも呼び出されます。このちょっとしたアプリケーションでは、ハンドラがリストに新しい項目を追加しているだけであるため、問題にはなりませんが、この設定には注意が必要です。通常、アプリケーションでは、UIスレッドの外側で行う必要がある、ネットワーク接続のオープン、データベースやファイルへのアクセスや、さまざまな計算を行うからです。データの準備ができたら、UIスレッド内のUI要素にデータを戻す必要があります。したがって、JavaFXが提供するスレッド構造には十分注意してください。スレッド処理を適切に行っていない場合、実行時例外、アプリケーションの無応答、データの破損や、さらに深刻な事態につながる可能性があります。

以上の内容をまとめておきます。コントローラには、FXMLビューの動作を定義する役割があります。ビューとコントローラは、FXMLビューのルート・ノードにあるfx:controller属性を使って関連付けます。FXMLLoaderは、自動的にコントローラのインスタンスを作成します。大半の場合、これが問題になることはありませんが、別の方法でコントローラのインスタンスを作成しなければならない場合はどうでしょうか。たとえば、@FXMLアノテーションが付加されたすべての要素の注入が完了する前に、追加のカスタマイズを行わなければならないような場合です。そのような場合でも、心配は無用です。これは、DIフレームワークを使って行うことができます。次のセクションでは、その方法を説明します。

JSR 330と依存性注入

JSR 330では、JavaプログラムでDIを実行する際の基本原理が定義されています。そのための実装は何種類かありますが、説明を簡潔にするため、個別のDIフレームワークの設定に関する説明は行いません。以降の例では、JSR 330で提供される標準APIを使用します。

それでは、先ほどの例で、単純なforループを使ってListViewにデータを挿入する代わりに、データ・プロバイダ・メカニズムを使ってみます。このデータ・プロバイダ・メカニズムは、次のようなインタフェースを使って定義します。

```
// sample/DataProvider.java
package sample;

import java.util.Collection;

public interface DataProvider<T> {
  Collection<T> getData();
}
```

このインタフェースは、アプリケーションでの必要に応じて自由に実装できます。たとえば、ファイルや、JDBCなどによるデータベース接続からデータを取得できます。次に、アップデートしたコントローラ・クラスを示します。

```
// sample/AppController.java
package sample;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.ListView;

import javax.inject.Inject;

public class AppController {
  @FXML private ListView<String> theList;
```

```
private final DataProvider<String> dataProvider;

@Inject
public AppController(DataProvider<String> dp) {
   this.dataProvider = dp;
}

public void initialize() {
   theList.getItems().addAll(dataProvider.getData());
}

public void addItem(ActionEvent ignored) {
    theList.getItems().add("Item "
     + theList.getItems().size());
   }
}
```

コントローラに単一引数のコンストラクタが定義されており、さらに@Injectアノテーションが付加されている点に注意してください。ただし、この単純な変更は、コントローラのインスタンスを作成するためのFXMLLoaderの規約と矛盾します。FXMLLoaderとコントローラ・クラスを連携させるためには、別の方法が必要になります。幸いなことに、その方法が用意されています。FXMLLoaderにControllerFactoryを設定すると、別の戦略を使ってコントローラのインスタンスを作成することができます。そのことを知っていれば、後はアプリケーションのエントリ・ポイントに戻り、FXMLLoaderのインスタンスにその新しい情報を設定するだけです。次に例を示します。

```
// sample/AppMain.java
package sample;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
```

```
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.net.URL;

public class AppMain extends Application {
  @Override
  public void start(Stage primaryStage) throws Exception {
    // DIフレームワーク経由で初期化する
    AppController controller = ...
    URL location = getClass().getResource("app.fxml");
    FXMLLoader fxmlLoader = new FXMLLoader(location);

    // ControllerFactoryを設定する
    fxmlLoader.setControllerFactory(param -> controller);

    VBox vbox = fxmlLoader.load();
    Scene scene = new Scene(vbox);
    primaryStage.setScene(scene);
    primaryStage.sizeToScene();
    primaryStage.show();
  }
}
```

これで、コントローラのインスタンスがDIコンテナからフェッチされます。このフェッチ操作はフレームワークによって異なるため、例示は省略していますが、簡単なControllerFactoryを定義するラムダ式を適用することもできます。AppControllerクラスがDIコンテナによって管理されるようになったことにより、メソッドに@PostConstructアノテーションを付加するなどの他の機能も利用できるため、コントローラをさらにカスタマイズできます。ただし、@PostConstructアノテーションが付加されたメソッドは、コントローラのインスタンスに@FXMLアノテーションが付加されたフィールドが注入される前に呼び出されるという点は覚えておいてくださ

い。なお、この設定の副作用として、fx:controllerの値は使われなくなります。それでも、DIコンテナによって提供されるコントローラの実際の型を指定しておくとよいでしょう。そうしておくと、@FXMLアノテーションで注入できるウィジェットの型および名前と、コントローラが提供するイベント・ハンドラの名前に関するヒントがIDEに表示されるためです。別の方法として、コントローラのプロパティをFXMLLoaderで直接設定することもできます。しかし、値の遅延計算を行う際に役立つと思われるため、今回はファクトリによるアプローチを紹介しました。

ここまでは、ビューとコントローラについて説明してきました。しかし、ご存知のように、UIアプリケーションの設計に使用できるデザイン・パターンがいくつかあります。その多くはモデル・ビュー・コントローラ（MVC）パターンから派生したもので、モデル・ビュー・プレゼンタやモデル・ビュー・ビューなどがあります。これらのパターンを使うと、役割によって各メンバーを明確に区別できるため、整然とした実装になり、アプリケーションの開発もやりやすくなります。

今回説明した例で、どこがモデルに相当するのかと疑問に思う方もいらっしゃるかもしれません。今回の例では、モデルはコントローラ内に暗黙的に存在しています。ObservableList<String>のような形ですべての項目の状態を保持する必要がある場合、このリストを配置する場所として最初に思いつくのはコントローラ自体でしょう。今回の例のような小さなアプリケーションの場合、その選択でも誤りとは言えません。しかし、アプリケーションの機能が増加し、複雑になってくると、この方式にはスケーラビリティが不足していることにすぐ気づくでしょう。そのため、MVCパターンやその派生パターンに基づいて考えることを強くお勧めします。つまり、動作はコントローラのみに配置し、データおよび状態はモデル・クラスに残しておきます。このクラスのインスタンスは、手動でもDIコンテナ経由でも作成できます。特定のアプリケーションの要件に従って自由に決めて構いません。

FXMLは、JavaFXのUIを宣言的に定義する方法を提供しますが、動作の定義に使うことはできません。

まとめ

FXMLは、JavaFXのUIを宣言的に定義する方法を提供しますが、動作の定義に使うことはできません。必要になる動作は、コントローラを指定して記述できます。コントローラは、DIを使用して、設定が必要なUIコンポーネントにアクセスすることができます。また、FXMLファイルで特殊なフォーマットを使って、UIコンポーネント内に直接フック

できるイベント・ハンドラを定義することも可能です。デフォルトの戦略を使うと、FXMLLoaderにコントローラのインスタンスを作成させることができます。しかし、別のインスタンス化メカニズムが必要な場合は、この戦略を変更することもできます。</article>

Andrés Almiray：JavaおよびGroovyの開発者で、Java Champion。17年を超えるソフトウェアの設計および開発の経験があり、Javaの初期の頃から、Webアプリケーションやデスクトップ・アプリケーションの開発に関わっている。熱心なオープン・ソースの信奉者で、Groovy、JMatter、Asciidoctorなどの有名プロジェクトに参加。Griffonフレームワークの創設メンバーであり、現在のプロジェクト・リードである。JSR 377の仕様リードも務めている。

1段組みの新しい記事レイアウトはお気に召しましたか?
ぜひ、JAVAMAG_US@ORACLE.COMまでご意見をお寄せください。

learn more

オラクルによるFXMLに関するチュートリアル

BRAD CYPERT

Clojure

Lispのエレガントさを JVMのパフォーマンスで

Clojureは、JVMベースの汎用的かつ動的な関数型プログラミング言語です。スクリプト言語の手軽さとインタラクティブ開発を、マルチスレッド・プログラミング向けの効率的で安定したインフラストラクチャと組み合わせている点が特徴です。筆者は、Clojureで初めてLisp方言と言われる言語に触れました。よく耳にする不満の中には丸括弧の多用があるものの、言語自体については、美しく書くことができ、エレガントに読むことができるものだと感じました。Clojureによる、おなじみのHello Worldコードは次のようになります。

```
(ns demo.hello-world)

; これはコメントです
(println "Hello World")
```

このコードを実行するためには、ClojureのJARファイルが必要です。このJARファイルは、こちらから、またはMavenから取得できます。JARファイルをダウンロードしたら、先ほどのコードをhello.cljに保存して、次を実行します。

```
java -cp clojure.jar clojure.main hello.clj
> Hello World
```

以降のソースの例で、式の下にあるコメントは、その式からの出力または戻り値を示しています。上記の例では、demo.hello-worldという新しいネームスペースを定義し、そのネームスペースでHello Worldを出力しています。とても簡単なサンプルですね。

これは貴重な第一歩ですが、文を出力するだけでは大半の言語の力を示すことにはなりません。Clojureも同様です。もう少し難しい問題を扱う方が適切でしょう。そこで、整数のリストに含まれるすべての整数を2倍して合計

を求める場合を考えてみます。これは、単純なマップ/リデュース問題だと気づいた方もいらっしゃるかもしれません。Clojureでは、まさにその考え方を構文で表現できます。

```
(ns demo.hello-lists)

(def my-list '(1 2 3 4 5))

(reduce
  +
  (map #(* % 2) my-list))
; 30
```

Clojureでは、キーワードという考え方がサポートされています。これをJavaなどの他の言語における予約キーワードという考え方（staticやfinalなど）と混同しないようにしてください。

上記のコードは、いくつかのシンプルな考え方を実際に示したものです。

- まず、値1、2、3、4、5が格納された新しいリストを定義します。Clojureのリストは、一重引用符の後に、リストに格納する値を丸括弧のペアで囲んで記述することによって定義できます。
- 次に、リデュース操作を行う式を記述します。reduceは、関数とリストを受け取ります。ここでは、+記号にバインドされている関数と、map関数から返されるリストを渡しています。
- 最初の行で定義したリストを使い、リストの各値に対してそれぞれを2倍した値をマップします。mapの最初の引数には、匿名関数を渡しています。%は、そこに引数が渡されることを示すプレースホルダです。複数の引数がある場合は、%1や%2のようにインデックスを付けて記述できます。

ほとんどの関数型言語と同様に、Clojureも問題に対して大まかなアプローチをとっています。関数型言語は、多くの場合、個々の手順に対して処理を行うコードが必要になるというよりは、リストの反復処理といった実装の詳細は気にせずに、解決しようとしている問題を説明するコードを書く方法だとみなされています。

Java開発者のためのClojure

Clojureは、コンパイル言語であるものの、強力なスクリプト言語のように感じられる、ある種のダイナミックさをJVMにもたらすものです。Clojureのすべての機能は、コンパイル時だけでなく、実行時にもサポートされます。さらに、Clojureを使う場合、お気に入りのJavaライブラリを使えるという快適さを捨てる必要はありません。ClojureはJavaの相互運用性を完全にサポートしているため、既存のJavaコードを活用できます。つまり、完全にコードを書き直さなくてもClojureを試してみることができるということでもあります。JavaコードとともにClojureコードを使うこ

とができ、その逆も可能です。

基本は不変性

関数型プログラミング言語であるClojureには、データ構造の不変性と永続性に関する豊富な機能が搭載されています。可変状態が必要な場合は、Clojureのソフトウェア・トランザクション・メモリ・システムとリアクティブ・エージェント・システムを活用して、クリーンかつ正確なマルチスレッド設計を実現できます。

マルチスレッドと言えば、Clojureでは標準ライブラリの一部としてcore.asyncという非同期ツールキットが提供されています。Clojureは大まかにGoの同時実行性モデルに基づいているため、Clojureを使うことによってチャネルを使いこなせるようになります。Clojureでチャネルを使うと、チャネルからの値の取り出しと、チャネルへの値の格納という主に2つのことができます。

Clojureの非同期処理を深く理解するためには、1冊の本まではいかなくとも、1つの記事が必要になるほどです。ここでは非同期処理について深く取り上げることはしませんが、Clojureの非同期ライブラリはスレッドよりもエレガントな抽象化を行う方法であることは強調しておきたいと思います。

ツールとコミュニティ

Clojureコミュニティの先頭に立っているのは、Clojureの発案者で、CognitectのCTOであるリッチ・ヒッキー氏です。ヒッキー氏はベテランのJava開発者で、評判の高い講演を数多く行っています。中には、有名な「Simple Made Easy」という講演もあります。実際、ライブラリの構築、初心者の援助、創造力の育成などにおいて、コミュニティ全体でこの「Simple Made Easy」という考え方が取り入れられています。この考え方を意識すると、Clojureコミュニティで「フレームワーク」が避けられる傾向にあることに気づくでしょう。初心者が大きなフレームワークを提示されることはあまりないでしょう。それよりも、ニーズに合わせていくつかのライブラリを組み合わせるべきだという助言を受けることの方がはるかに多いはずです。Clojure言語の初心者、大規模なフレームワークを使ってきた方は特に、このことに対して不安を感じるかもしれません。しかし、少しばかり時間をかけて努力すれば、解決方法を探していた問題を解決できるだけでなく、Clojure自体を深く理解できることでしょう。

非常によく使用されているのは、Clojureの事実上の標準ビルド・ツールであるLeinengenです。このツールは、コミュニティが構築したもので、JavaでGradleが解決したものと同様の問題を解決することを狙ったものですが、実際はそれ以上のことを行ってくれます。Leinengenには、プロジェクトをすぐに開始できるテンプレート・プロジェクトが搭載され、プロジェクトのアドオン（プロファイルと呼ばれます）のサポートも提供されています。たとえば、PostgreSQLがサポートされたluminusテンプレートを土台にして新しいプロジェクトを作りたい場合、ターミナ

ルを開いて次のコマンドを実行すればいいだけです。

```
lein new luminus myapp +postgres
```

覚えておいていただきたいのは、各テンプレートには独自のプロファイルが含まれているため、別のテンプレートを使う場合は、postgresプロファイルをベースにできない場合があることです。

Leinengenは、build.gradleファイルやpackage.jsonファイルに相当するproject.cljファイルを作成してくれます。このファイルの中で、依存性の定義、プロジェクトのメタデータの設定、Leinengenプラグインの利用、ビルド・パイプラインの作成を行うことができます。

また、Leinengenは、豊富な機能を持つRead-Eval-Print Loop（REPL）も備えています。PythonやNodeを使って開発を行っている方なら、Python統合開発学習環境（IDLE）やNode REPLでさまざまなアイデアを試してみたことがあるでしょう。LeinengenのREPLも同じようなものです。Leinengenをインストールすると、コマンドラインからlein replを実行するだけで、新しく思いついたことをClojureで簡単にテストできます。クリーンなClojure環境が開き、そこで新しいアイデアを試してみることができます。

Clojureでのテスト

Clojureには、clojure.testというネームスペースを持つ軽量テスト・ライブラリが付属しています。このライブラリの中核をなすのが、任意の式のアサーションを定義できるisマクロです。Clojureでは、2つのパターンでテスト・ケースを記述できます。1つ目は、with-testマクロを使ってソース・コード内にインラインでテスト・ケースを記述する方法です。2つ目は、deftestマクロを使って別のネームスペースにテストを配置する方法です。シンプルな算術式のテストを次のようにして容易に行うことができます。

```
(ns demo.hello-tests)

(deftest math-tests
  (is (= 7 (+ 4 3)))
```

テストとは本来、単にコンピュータのもっとも基本的な処理の境界値をテストするものではなく、ビジネス・ロジックをテストするものです。もう少し的確な例として、独自のネームスペースで実行されるテスト・ケースが考えられま

す。次に例を示します。

```
(ns app.test.modules.users
  (:require [clojure.test :refer :all]
            [app.modules.users :refer [get-user]]))

(deftest user-tests
  (testing "Fetching a user"
    (let [user (get-user 1)]
      (is (some? (:email user))))))
```

上記のコードは、今まで紹介してきたものの中で一番複雑であるため、詳しく見てみることにします。ここでは、新しいネームスペースを設定して1件のテストを定義しています。そのテストでは、キーに1を指定してget-userを呼び出したときに、電子メールが定義されているデータ構造が返されることを確認しています。なお、get-userは、「email」というキーとメールアドレスの値で構成される、キーと値のペアを返すことに注意してください。

ここでは、以下のことを行っています。

- 新しいネームスペースapp.test.modules.usersを定義します。
- このネームスペースでは、いくつかのインポートを行います。これは、clojure.testとapp.modules.usersにアクセスする必要があるためです。ここではclojure.test内の関数をすべて参照していることにお気づきになると思いますが、app.modules.usersで行っているように、テストしようとするもののみを公開することも可能です。
- user-testsという新しいテストを定義します。
- testingマクロを使って新しいスコープを作成し、そのテスト・コンテキストにFetching a userというラベルを設定します。
- let式を使い、(get-user 1)で得られたキーと値のペアを、userという名前のローカル・バインディングにバインドします。
- 最後に、isマクロでアサーションを定義します。このマクロの本体には、some?関数が含まれています。この関数は、渡された値がnil（nullを表す値）でない場合にtrueを、それ以外の場合はfalseを返します。その後、先ほどバインドした、ユーザーを表すデータ構造で:emailキーの値を検索します。

コロンの意味

Clojureでは、キーワードという考え方がサポートされています。これをJavaなどの他の言語における予約キーワードという考え方（staticやfinalなど）と混同しないようにしてください。キーワードは、マップやパターン・マッチングの条件をわかりやすく記述する優れた方法です。また、出力される際には、コロンを除いた文字列として評価されます。Clojureでコードを書く場合、頻繁にキーワードを使うことになるでしょう。キーワードは、記述するコードの意図の表現や宣言に便利で、ユーザー・データのマップのキーや、リストの値を定義する際に使用できます。

```
(ns demo.hello-keywords)

(def brads-details
 {:name "Brad"
  :languages '(:clojure :java :javascript)})
```

キーワードは、Javaの列挙型とよく比較されます。列挙型と同じように使用されることもありますが、コンパイル時または実行時に定義されるキーワードに要件はないことについても触れておくべきでしょう。事実、「keyword」関数を使って、実行時に文字列からキーワードを作成することもできます。先ほどの例では、name、languages、clojure、java、javascriptというキーワードが使われており、これらのキーワードはすべてコロンで始まっていることにお気づきになると思います。

関数型プログラミングという考え方

Clojureの大部分は、関数型プログラミング言語という考え方に基づいています。そのため、Clojureはコードを一連の数式として扱い、状態やデータの変更を避けようとします。Clojureは、関数プログラミングでとてもよく使われる手法によってこのパラダイムにアプローチしています。すなわち、Clojureでは不変の永続データ構造と第一級関数群が使われています。このアプローチとJavaのマップを比較してみます。まずは、Javaのコードを示します。

```
import java.util.HashMap;

HashMap<String, Integer> map = new HashMap<>();
map.put("java", 7);
System.out.println(map);
map.put("java", 8);
```

```
System.out.println(map);
```

ここで何をしているかは容易におわかりいただけるでしょう。最初にマップを出力した際は、javaという名前のキーおよび値7が出力されます。マップのjavaキーに新しい値を設定してから出力した際には、javaという名前のキーおよび値8のマップが出力されます。Clojureでは、この操作は次のようになります。

```
(ns demo.hello-immutability)
(def map {:java 7})
(println map)
(assoc map :java 8)
(println map)
```

このClojureコードでは、両方とも{:java 7}が出力されます。これは、Clojureのマップが不変であるためです。Clojureでは、ほとんどのデータ構造が不変です。(assoc map :java 8)という行で実際に返されるのは、元のマップを複製して作られた{:java 8}という新しいマップです。並列プログラミングには、デッドロックやデータ競合といったありがちな落とし穴があります。そのような落とし穴を避けることができるようにするために、Clojureの中核には不変性という考え方が据えられています。

Clojureの関数は第一級です。すなわち、関数をデータ構造に格納することや、引数として渡すことが可能です。実際、関数の適用（関数をリスト内のすべての値に適用するという考え方）という処理を行っているのは、別の関数とリストを引数として受け取る関数です。たとえば、applyを使って、指定したリスト内のそれぞれの値に関数を適用することができます。パラメータには、匿名関数や、ネームスペースまたはローカル・バインディングにすでにバインドされている関数を使うことができます。最初に示した、値を2倍する例は、次の2つの方法のいずれでも記述できます。

```
(ns demo.hello-apply)

(def double [x] (* x 2))

(apply double (list 1 2 3 4))
; (2 4 6 8)
```

```
(apply #(* % 2) (list 1 2 3 4))
; (2 4 6 8)
```

Lispやメタ言語を使ったことのある方なら、関数型プログラミング言語の分割代入というエレガントな仕組みを期待することでしょう。これは、キーと値のペアやリスト（ネストしている場合もあります）に格納されたデータから複数の値を抽出する際に便利な方法です。分割代入の本質は、データ構造に含まれる、使用またはバインドしたい一部のデータを宣言的に定義できることにあります。分割代入はClojureですぐに使用できるため、複雑なデータ構造をはるかに簡単に扱うことができます。キーワードのリストを受け取り、その最初のキーワードを文字列として返す関数は、次のように記述できます。

```
(ns demo.hello-transform)

(defn first-keyword->string
  [my-list]
  (name
    (first my-list)))
```

関数名のわかりやすさに注目してください。特殊文字（ハイフンや大なり記号など）を関数名に使用できるため、簡潔でわかりやすい関数名を付けることができます。データをある構造から別の構造に変換するような場合には、特にそれが当てはまります。この関数は、1つのパラメータmy-listを受け取り、引数を文字列に変換するname関数を呼び出します。この場合、引数はmy-listの最初の項目になります。

このコードは、分割代入を使って次のように書き換えることもできます。

```
(ns demo.hello-transform)

(defn first-keyword->string
  [[head tail]]
  (name head))
```

このコードこそ、とてもわかりやすいものだと思いませんか? Clojureでは、リスト、マップ、ベクターなどの多くのもの

に対して分割代入を行うことができます。特に、リストやマップなどの大規模なデータ構造がどのくらい使い回されるかを考慮すると、Clojureでの開発において分割代入はすぐに標準的な手法となります。

マクロ・システム

他のLisp言語から色濃く継承した特徴として、先ほども触れたClojureのマクロ・サポートが挙げられます。キャリアの大半でJavaを使ってきたという方にとって、マクロという概念はおなじみではないかもしれません。マクロはコンパイル時に評価され、コードに展開されます。つまり、コンパイラ拡張やCのプリプロセッサと同じようなものです。

マクロは非常に強力な機能で、Clojureではとてもよく使用されます。実は、Clojureの中核をなす構成要素の多くは単なるマクロです。たとえば、Web開発でとてもよく使用されるCompojureというライブラリでは、URLルートを定義し、そのルートに一致する着信リクエストを関数にディスパッチできるようになっていますが、その際にマクロが使われています。コードベースにCompojureへの依存性を追加すると、次のような構文でルートを容易に定義できるようになります。

```
(ns demo.hello-compojure)

(def get-user-by-id
  [id]
  (str "getting user for id: " id))

(defroutes my-routes
  (GET "/user/:id" [id] get-user-by-id)
```

この例では、Compojureを使ってWebサーバーの/user/1へのルートを設定しています。curlまたはWebブラウザを使ってこのリソースをリクエストすると、getting user for id: 1が返されます。実際には、get-user-by-idをもう少し意味のある処理で置き換える必要がありますが、マクロの威力はこの例からわかっていただけるでしょう。マクロを使うと、わかりやすく簡潔な表記法を使ってたくさんのコードを書くことができます。

この例では、Clojureを使ってHTTPサーバーを抽象化するRingというWebサーバーを使っています。筆者がClojureを使った経験のほとんどは、Web中心の事例です。その際に、Ringによる抽象化を使用しています。Ringは長い間使われており、Clojureコミュニティの多くのメンバーがサポートおよびコミットを行ってきています。

まとめ

その構文と長所から、Clojureはすばらしく強力な言語であることが証明されています。筆者もClojureを使ったコーディングが大好きです。触れるべきことはまだたくさんありますが、本記事では、皆さんがこの言語の魅力を感じ、試したいと思っていただけると筆者が考える一部の要点のみを紹介しました。Clojureについてもっと学んでみたいという方は、ホーム・サイト、筆者がClojureに目覚めた無料のオンライン・チュートリアル、または筆者の記事をご覧ください。</article>

Brad Cypert：米国ケンタッキー州ルイビル在住のソフトウェア・エンジニア。Clojure、EdTech、JVMに情熱を傾けている。LinkedIn、Carfaxなどの大手技術企業で働いた経験を持ち、空いた時間には、より快適にポッドキャストを聞けることを目指すWebアプリケーションをClojureとTypeScriptで構築する自身のプロジェクト、Poriosに取り組んでいる。

SIMON ROBERTS

クイズに挑戦

中級者、上級者向けの問題をさらに出題

いつもこのクイズをお読みくださっている皆さんであれば、以下の設問が2種類の認定資格試験の難易度を想定していることはご存知でしょう。「中級者向け」と書かれた設問は、Oracle Certified Associate試験に相当するものです。この試験には、基本的なレベルの内容が含まれます。「上級者向け」と書かれた設問は、1Z0-809 Programmer II試験に相当するものです。1Z0-809 Programmer II試験は、Java 8のプログラミングの基本的知識を有することがすでに認められており、さらに高度な専門知識を有することを証明しようとしている開発者向けの認定資格試験です。

念のため、今回の設問はJava 8のものであることを強調しておきます。今後Java 9を取り上げる際は、もちろん、移行したことがはっきりとわかるようにします。

設問1（中級者向け）. 次のコードについて：

```
int i1 = 1;
Integer io = new Integer(i1);
// line n1
Integer ib1 = i1;
Integer ib2 = 1;

System.out.println(
   (i1 == io) + " " +
   (ib1 == ib2));
```

どのような結果になりますか。

a. true true

b. true false

c. false true
d. false false
e. line n1の次の行でコンパイルが失敗する

設問2（中級者向け）. 次の2つのクラスについて:

```
class Base {
 int x = 1;
}

class Sub extends Base {
 int y = 1;

 // line n1
}
```

line n1に挿入される静的メソッドについて正しいものはどれですか。ただし、いずれのクラスにも他に何も追加しないものとします。

a. フィールドint xにもフィールドint yにもアクセスできない。
b. フィールドint xにはアクセスできないが、適切に参照すればフィールドint yにはアクセスできる。
c. 適切に参照すれば、フィールドint xにもフィールドint yにもアクセスできる。
d. this.xおよびthis.yという形式を使う場合にのみ、フィールドint xおよびint yにアクセスできる。
e. this.xおよびthis.yという形式を使ってフィールドint xおよびフィールドint yにアクセスできるが、他の形式も使うことができる。

設問3（上級者向け）. 次のコードについて:

```
public void kickOff() {
 ExecutorService es = Executors.newFixedThreadPool(4);
 Callable<String> c = new Callable<String>() {
  public String call() {return "Done";}
```

```
  };
  String result = es.submit(c).get(); // line n1
  // ...
}
```

正しいものはどれですか。 2つ選んでください。

a. コードは問題なく、line n1で変数resultに値"Done"が割り当てられる。

b. callメソッドの宣言を次のように変更した場合、line n1をコンパイルできるようにするためには、適切な例外ハンドリングを追加する必要がある。

```
public String call() throws MyException
```

c. callメソッドの宣言を次のように変更した場合、メソッドをコンパイルできなくなる。

```
public String call() throws MyException
```

d. InterruptedExceptionがハンドリングされていないため、line n1のコードはコンパイルできない。

e. ExecutionExceptionがハンドリングされていないため、line n1のコードはコンパイルできない。

設問4（上級者向け）. 次のコードについて：

```
class Parent {}
class Child extends Parent {}

public void doBatch(/* 仮パラメータの型 */ data) {
  for (Parent p : data) {
  }
}

/* 引数の型 */ lc = null;
```

doBatch(lc);

コードをコンパイルできるようにするためには、コメントされている部分をどの仮パラメータと実引数の型のペアで置き換えればよいですか。 2つ選んでください。

a. 仮パラメータ:Parent[]
   引数:Child[]
b. 仮パラメータ:List<Parent>
   引数:List<Child>
c. 仮パラメータ:List<? extends Parent>
   引数:List<Child>
d. 仮パラメータ:List<? super Child>
   引数:List<Parent>
e. 仮パラメータ:List<Parent>
   引数:List<? extends Parent>

設問1: 正解は選択肢Aです。この設問は、等価性の範囲と数値ラッパー型に関するいくつかの明確な考え方について問うものです。現実的で妥当な近似表現で言うなら、==演算子は、その左辺と右辺において式の値のビット・パターンが同じかどうかをテストするものです。すなわち、プリミティブ式が対象の場合、==演算子は2つの式が同じ数値（昇格やアンボクシングの結果である場合もあります）を表していればtrueを返します。オブジェクト参照式の場合、==は2つの式が同じオブジェクトを参照していればtrueを返します。

それでは実際のところ、設問のコードにある2つのテストは何を意味するのでしょうか。まず、i1 == ioについて考えてみます。このテストでは、intプリミティブ値とIntegerオブジェクトを比較しています。Integerオブジェクトは、同じint値をラップして明示的に作成されます。この場合の演算子の動作は、オブジェクトにラップされてい

るint値をプリミティブとして抽出し、2つのプリミティブ値を比較するというものです。両辺の値は共に1であるため、この比較はtrueを返します。Java言語仕様のセクション15.21.1には、プリミティブがボクシングされて新しいIntegerオブジェクトが作成されるのではなく、ラッパーのプリミティブ値が抽出されると明示されていることに特に注意してください。ちなみに、仮に新しいオブジェクトが作成された場合、結果はfalseになったでしょう。

次のib1 == ib2では、2つのオブジェクト参照が比較されます。つまり、比較によってtrueが返されるのは、2つの参照ib1およびib2が同じオブジェクトを参照している場合のみです。new演算子を呼び出してオブジェクトを作成した場合、別々のオブジェクトになったでしょう。しかし、今回のように小さな整数をボクシングする場合は特殊で、基盤システムによって、プールから取得したオブジェクトが共有されます。そのため、この2つの参照は、同じ意味合いを持つ2つのオブジェクト（すなわち、それぞれに同じ値がボクシングされた2つのオブジェクト）を単に指すものではなく、メモリ内の同じオブジェクトを指すものになります。そのため、このテストの結果もtrueとなります。

Javaでは、+演算子を用いて任意のデータ型を Stringに連結できます。

それでは、ここで少しばかり紙面を割いて、ボクシングに使われるオブジェクトのプールについて説明します。最初に注目すべき点は、Javaは特別な範囲の数値を規定しており、次の代入文がある場合、テストa == bはtrueになることです。

```
Integer a = 3;
Integer b = 3;
```

これが可能で安全なのは、ラッパー・オブジェクトが不変であるためです。したがって、これらが変更されることはないため、共有しても安全です。この措置は、メモリの使用量を低下させ、その結果としてガベージ・コレクタの負荷を減らすためのもので、実際に効果もあります。仕様には、この措置が行われるのは-128から+127までの値とBoolean値の場合であると記載されています（セクション3.10のサブパラグラフ）。

以上の内容を踏まえると、設問にある2つの比較式はいずれもtrue値になることは明らかです。Javaでは、+演算子を用いて任意のデータ型をStringに連結できるため、出力結果は、trueという単語の後に空白が追加され、その後にもう一度trueという単語が追加されるというものになります。そのため、選択肢Aが正解で、選択肢B、C、D

は誤りです。

プリミティブがオブジェクト参照に割り当てられている部分でコンパイル・エラーになる可能性も考えられるかもしれません。もちろん、JavaはInteger型も含めてプリミティブ値へのオブジェクト参照を許可しないため、一般的にこのようなコードは疑われることになります。しかし、（ずっと以前のJava 5で）ボクシング機能が追加されているため、当然ながら何もしなくても必要な変換が行われます。そのため、line n1の後の2つの代入には何の問題もなく、エラーも起こりません。そのため、選択肢Eは誤りです。

設問2：正解は選択肢Cです。この設問は、静的メソッドの性質と、インスタンス・メソッドとの違いについて問うものです。

次のメソッドが設問のサンプル・コードに追加された場合を考えてみます。

```
public static void doStuff() {
  y = 99;
}
```

このコードのコンパイルは失敗します。変数yへのアクセスが原因です。興味深いことに、筆者がJava開発で使っているIDEでは、「staticでない変数yをstaticコンテキストから参照することはできません」という不正確で誤解を生むメッセージが表示されます。「staticメソッドからインスタンス・フィールドを参照することはできません」といった、表面だけをとらえたメッセージが表示されることもよくあります。

一方、OracleのJDK 8に付属するコマンドライン・コンパイラでは、わずかに異なるメッセージが表示されます。しかし、その意味はまったく異なっています。表示されるのは、「staticでない変数thisをstaticコンテキストから参照することはできません」というメッセージです。これこそ的確なメッセージです。暗黙の参照thisは、静的でないコンテキストにのみ存在します。そのため、この設問で静的メソッドが挿入される場所でthisを使うことはできません。

サンプルの静的メソッド内にある修飾子のない変数yが暗黙的にthis.yを参照していることを考えれば、このような形式の参照が機能しないことは明らかです。これを踏まえると、選択肢DとEは誤りであることがわかります。確実に言えるのは、this.yとthis.xは機能しないということです。

次に考慮すべき点は、静的メソッドがインスタンス変数にアクセスできるかどうかです。アクセスすることは簡単であり、必要となるのはインスタンスへの有効な参照のみです。thisという参照は使用できませんが、引数リスト

を使って静的メソッドにインスタンスへの参照を渡すのは簡単です。そこで、次のメソッドをline n1に挿入したと考えてみます。

```
public static void doStuff(Sub self) {
  System.out.println("x is " + self.x + " y is " + self.y);
}
```

この場合、doStuffを呼び出す際に実パラメータとして渡されたオブジェクトのxフィールドとyフィールドの値が出力されます。考えられる別の方法は、静的メソッドで自らを含むクラスのインスタンスを作成することです。これはよくある動作で、静的ファクトリという考え方の元にもなっています。つまり、選択肢Cは正解です。フィールドにアクセスする際に必要となる唯一の要件は、有効な参照が存在することです。そのため、これらのフィールドにアクセスできないとする選択肢AとBは誤りです。

さらに注意しなければならないのは、選択肢Bで、フィールドを使用できるかどうかは、フィールドがクラス内にあるか親クラス内にあるかによるとされている点です。しかし、この点は無関係です。たとえば、親クラスのフィールドがprivateである場合、そのような制限が生じる可能性もあります。しかし、この設問にはそのような前提はありません。また、対象のフィールドにデフォルト・アクセスが指定されており、かつ子クラスが別のパッケージにある場合にも、子クラスから親のフィールドへのアクセスは制限されます。ただし、この設問では、パッケージについての情報は何も与えられていません。そして、試験のガイダンスには次のように書かれています。

getメソッドのインフラストラクチャは、すべてのCallableから発生する可能性がある、あらゆる例外を処理する必要があります。その例外が特定の実装で実際に存在するかどうかは関係ありません。

「**package文およびimport文の欠落：**サンプル・コードにpackage文やimport文が含まれておらず、省略されている文について設問で明示的に言及されていない場合は、すべてのサンプル・コードが同じパッケージにあり、コードを適切に動作させるimport文が含まれていると仮定します」

そのため、親クラスと子クラスは同じパッケージにあると考える必要があります。

設問3：正解は選択肢DとEです。ExecutorServiceインタフェースは、スレッド・プールをカプセル化するために使われます。このようなスレッド・プールは、プールに提供されるコードの実行をサポートします。実行対象のコードには、RunnableインタフェースまたはCallableインタフェースを使います。Runnableインタフェースでは、何のチェック例外も宣言されていないrunメソッドが定義されていますが、Callableインタフェースでは、Exceptionをスローするcallメソッドが定義されています。そのため、選択肢Cは誤りです。callメソッドでは、状況に応じて任意の例外をスローできます。もちろん、そのような例外は適切に処理する必要があり、プール・インフラストラクチャ側でハンドリング方法の意味合いを判断することはできません。

ExecutorServiceのsubmitメソッドは、Futureオブジェクトを返します。Futureオブジェクトは、実際のところ、ジョブの送信後にそのジョブとインタラクションを行ういくつかの便利な方法を提供する、単なる「ジョブ・ハンドル」にすぎません。そのような方法の1つがgetメソッドです。このメソッドを使って、完了したジョブの結果を取得します。ジョブのcallメソッドから例外がスローされた場合、getメソッドはコール元にその例外を報告する必要があります。

また、getメソッドを呼び出すと、ジョブが完了するまで（必要に応じて）処理がブロックされます。一般的な慣習として、ブロッキング・メソッドは中断できるようになっています。この中断が発生すると、ブロッキング・コールは異常終了し、コール元にInterruptedExceptionがスローされます。getメソッドもその慣習に従います。InterruptedExceptionはもちろんチェック例外であるため、getを呼び出すコードは、Java例外メカニズムの通常のルールに従って対処する必要があります。そのため、選択肢Dは正解であると考えることができます。

なお、Callableインタフェースのcallメソッドはthrows Exceptionと宣言されているため、実装からInterruptedExceptionがスローされたとしても問題はありません。とはいえ、getメソッドは独自の理由でチェック例外をスローするため、getメソッドから2つの理由で同じ例外がスローされる可能性がある場合、ややこしいことになります。そのため、get操作の結果を処理するコードを全般的に簡略化するという意味も踏まえて、callメソッドで発生する例外は、すべてExecutionExceptionでラップされてからgetメソッドにスローされるようになっています。このExecutionExceptionもチェック例外です。そのため、選択肢Eも正解です。

選択肢DとEが正解であるということは、選択肢Aは誤りだということになります。
さらに、callメソッドで発生する可能性があるすべての例外を処理するために必要なものはすでに存在しているため、選択肢Bも誤りです。getの呼出しには例外ハンドリングが必要ですが、callメソッドが変更されているかどうかには関係ありません。getメソッドのインフラストラクチャは、すべてのCallableから発生する可能性がある、あら

ゆる例外を処理する必要があります。その例外が特定の実装で実際に存在するかどうかは関係ありません。

コンパイラが安全性を証明できない場合、ジェネリクス・システムはコードのコンパイルを拒否しなければなりません。ただし、ジェネリクスの構文による回避策が用意されています。

設問4：正解は選択肢AとCです。この設問は、一般的なジェネリクスとコレクションの関係を問うものです。List<Parent>はList<Child>で置換できると考え、選択肢Bは有効だと判断してしまいがちです。このコードには、実際に実行時に問題を起こすような要素は何も含まれていませんが、コンパイルすることはできません。その理由は、一般的にこのような置換は安全でないためです。doBatchメソッドで、受け取ったListに項目を追加しようとする場合を考えてみます。ParentオブジェクトをList<Parent>に追加しても問題ありませんが、List<Child>に追加することは安全ではありません。つまり、コンパイラは選択肢Bに記載されているペアを許可してはならないことになります。許可した場合、実行時エラーが発生する可能性が生じるからです。そのため、選択肢Bは誤りです。

ただし、何かのコレクションをメソッドに渡す場合、その多くはコレクションの各メンバーに何らかの反復処理を行いたい場合であり、コレクションに項目を追加することはあまりないと考えてよいでしょう。これを安全に行うことができるとした場合、Parent型を受け取るメソッドに複数のChildオブジェクトを渡すことができるととても便利です。実際、「複数のオブジェクト」が配列であれば、この操作はある程度安全であると言えます。コレクションとは違い、配列ではベース型がわかっているからです。つまり、Childの配列にParentオブジェクトを格納しようとすると、例外（具体的にはArrayStoreException）が発生します。
コンパイラがこのようなミスを防いでくれればさらによいかもしれませんが、不適切なオブジェクトがコレクションに入ってしまうことは受け入れられないにしろ、挿入時に確実に例外が発生すれば問題ないという考え方もあるでしょう。そのため、Javaは選択肢Aに記載されているペアの呼出しを常に許可してきており、この選択肢は正解です。

しかし、ジェネリクスのメカニズム、すなわちCollections APIでは、実行時の安全性に対する考え方が異なっています。コンパイラが安全性を証明できない場合、ジェネリクス・システムはコードのコンパイルを拒否しなければなりません。ただし、ジェネリクスの構文を使った回避策が用意されています。List<?...構文を使うと、「何のリストなのかは正確にはわからないが...」と宣言したうえで、その後に知っていることを記述できます。それでは、これを

動作させるために何を知っている必要があるでしょうか。ここで起こるのは、?のオブジェクトを読み取るということです。つまり、この状況で疑問符が表すものが何であったとしても、その型のオブジェクトを読み取り、そのオブジェクトをParent型の変数に割り当てることになります。この場合、forループ内で変数pへの代入が起こります。それでは、Parentの参照にその型のオブジェクトを安全に割り当てられるようにする場合、疑問符は何を表せばよいのでしょうか。答えは、ParentまたはParentのサブタイプです。これは、List<? extends Parent>という構文を使って表現できます。以上より、選択肢Cは正解です。

選択肢Dももっともらしく見えるかもしれませんが、このペアを使用できるのは、引数として渡されたリストにChildオブジェクトを格納するメソッドだけです。この形式を使う場合、リストの内容をObject型以外の型に割り当てようとすると、メソッドは失敗します（? super Childは、リストの要素がChildの親クラスであればObjectを含め、文字どおり何でも構わないという意味になるためです）。そのため、このメソッドの本体を考慮すると、選択肢Dは誤りであることがわかります。

同様に、選択肢Eでも、Parentオブジェクトを受け入れる準備ができていないListにParentオブジェクトが挿入される可能性が生じます。そのため、選択肢Eも誤りです。</article>

Simon Roberts：Sun Microsystemsがイギリスで初めてJavaの研修を行う少し前にSunに入社し、Sun認定Javaプログラマー試験とSun認定Java開発者試験の作成に携わる。複数のJava認定ガイドを執筆し、現在はフリーランスでシリコン・バレーや世界中の多くの大企業の研修を行う。OracleのJava認定プロジェクトにも継続的に関わっている。