

JavaTM magazine

By and for the Java community 



05

JDK 9の9つの
新機能

11

JAVA 8からJAVA
9への移行

15

COLLECTION、STREAM、
ITERATORの機能拡張

21

JSHell:
新しいREPL

表紙画像：I-HUA CHEN

London Java Community

JDK 9 のインキュベータ・テクノロジーで HTTP 通信が大幅に簡素化



A middle-aged man with short, graying hair and glasses is walking towards the camera on a city sidewalk. He is wearing a light blue long-sleeved button-down shirt over a dark collared shirt, blue jeans, and a brown leather belt. He is holding a black folder or tablet under his left arm. The background is a blurred city street with other pedestrians, a green awning, and a building.

JDK 9のようなメジャー・リリースのデリバリが何度も遅れるのは、コミュニティが大切にされている場合にはよくあること

Javaコミュニティの一部の人は、この延期、そして今回の延期が何度も繰り返された遅延の後のさらなる延期であるという事実に、強い不満を抱きました。私は違います。

しかし、遅延の原因はコミュニティとの協働体制にあるため、私は文句を言わずにこの延期を受け入れようと思いました。業界の委員会に従事したことがある人に聞けば、協働作業は困難で雑音が多く、いらいらの極みだということ漏れなく教えてもらえます。このようなことは、おそらく育児を除けば他にありません。不満があつたとしても、すべての当事者は、努力を放棄するよりも協働作業の大変な部分を克服する方がよいと確信しています。自分の好みを主張するよりも協働する方が、全員にメリットがあるのです。

ORACLE®



Step up to modern cloud development. At the Oracle Code roadshow, expert developers lead labs and sessions on PaaS, Java, mobile, and more.

go.oracle.com/oraclecoderoadshow

developer.oracle.com

#developersrule



全体としての利益というのが理解されていることが、長く続いているJavaの歴史の原動力であり、Javaの成功に不可欠であることは疑う余地がありません。JavaとJVMの発展そのものが協働の成果であることは忘れられがちです。開発作業はパブリック・リポジトリの中でオープンソースとして行われ、どの機能を選択するか、どのようにコードを実装するか、リリース日はいつにするか、といった争点についての議論は公開メーリング・リストで行われます。実際、新しいリリース日に関する最近の決定や遅延の具体的な理由については、すべて公開フォーラム上で告知と応答が行われました。

では、ここで質問です。100名を超えるエンジニアを言語開発に携わらせていながらリリース日の問題をパートナーのコミュニティに委ねるような企業が主要スポンサーになっている言語を他にご存知ですか。言語開発にこれほど全力で取り組んでいる企業は、オラクル以外ではApple、Google、Microsoftの3社だけです。ただし、Javaのようなオープンかつ総意に基づくアプローチを採用している会社はオラクル以外にありません（私は上記3社の言語への取り組みや好みのアプローチを非難するつもりはなく、むしろ、Javaに対するオラクルのアプローチがいかに特殊なものであるかを強調しようとしています）。

協働という名目で遅延を受け入れたということは、私はJavaコミュニティを運営する部内者の立場で考えているということでしょう。多くのJava開発者は、リリース日をめぐる駆け引きには興味がなく、議論が公開の場で行われているかどうかなど気にもかけません。開発者は新



スできる状態になるまで待たなくても公開されるようになるでしょう。皮肉なことです。Java 9には特定の変更の影響を局所化できるモジュール化機能があるため、この戦略が可能になりました。

要するに、このリリースの遅延は、Javaの成功を支えているオープンな協働モデルに必要な産物だということです。オープンソースやオープンな協働作業を支持する方であれば、こうした遅延はプロセスに伴う犠牲であるということが、きっとおわかりになるでしょう。そういう意味で、これは一方的な決定によるリリースとは正反対のものです。とはいえ、モジュール化機能が導入されることで、今後は決められたスケジュールどおりにリリースできるようになり、高いレベルでの協働作業への取り組みも続いていくと思うと楽しみです。

Editor in Chief, Andrew Binstock
javamag_us@oracle.com
[@platypusquy](https://twitter.com/platypusquy)

しいテクノロジーがリリースされればよいのであり、だからこそ何度も延期されてきたことに不満を覚えているのです。

こうした観点から、想定されたスケジュールどおりに新技術が公開されるようにしようという取り組みが進んでいます。この新しいアプローチでは、公表した期日が到来したら、準備ができていたテクノロジーをすべて公開することを提案しています。オラクルでJava Platformグループの開発部門担当バイス・プレジデントを務めるGeorges Saabの発言によると、この新しいアプローチはJava 9のリリース後に採用される予定です。そうなれば、さほど大きくない機能改良なら、1つの中心機能がリリー

JDK 9の内側

JAVA 9の機能 05 | JAVA 8からJAVA 9へ 11 | COLLECTIONとSTREAM 15 | JSHELL 21 | HTTP/2 32

テーマを1つに絞った本号の『Java Magazine』では、新たにリリースされるJDK 9の、Javaプラットフォーム・モジュール・システム (JPMS) 以外の利点を重点的に取り上げています。モジュール機能はこのリリースの目玉としてもはやされることが多い機能ですが、この原稿を印刷に回した時点では、承認を受けた最終版がまだできてい

ませんでした。そのため、後で「誤りでした」ということになる可能性のある情報を先走って提供するのではなく、JDK 9のその他の部分を重点的に取り上げることにしました。その他の部分はすでに正式な承認を受け、完成しているため、モジュールが正式に承認されたタイミングでリリースされます。このリリース日は、現時点では9月下旬になる見込みです。この遅延の理由については、「編集長より」(2ページ)をお読みください。リリース日のすぐ後に発行される『Java Magazine』では、Java 9特集号の第2弾として、新しいモジュール・アーキテクチャとその上手な使用方法を詳しく紹介する予定です。

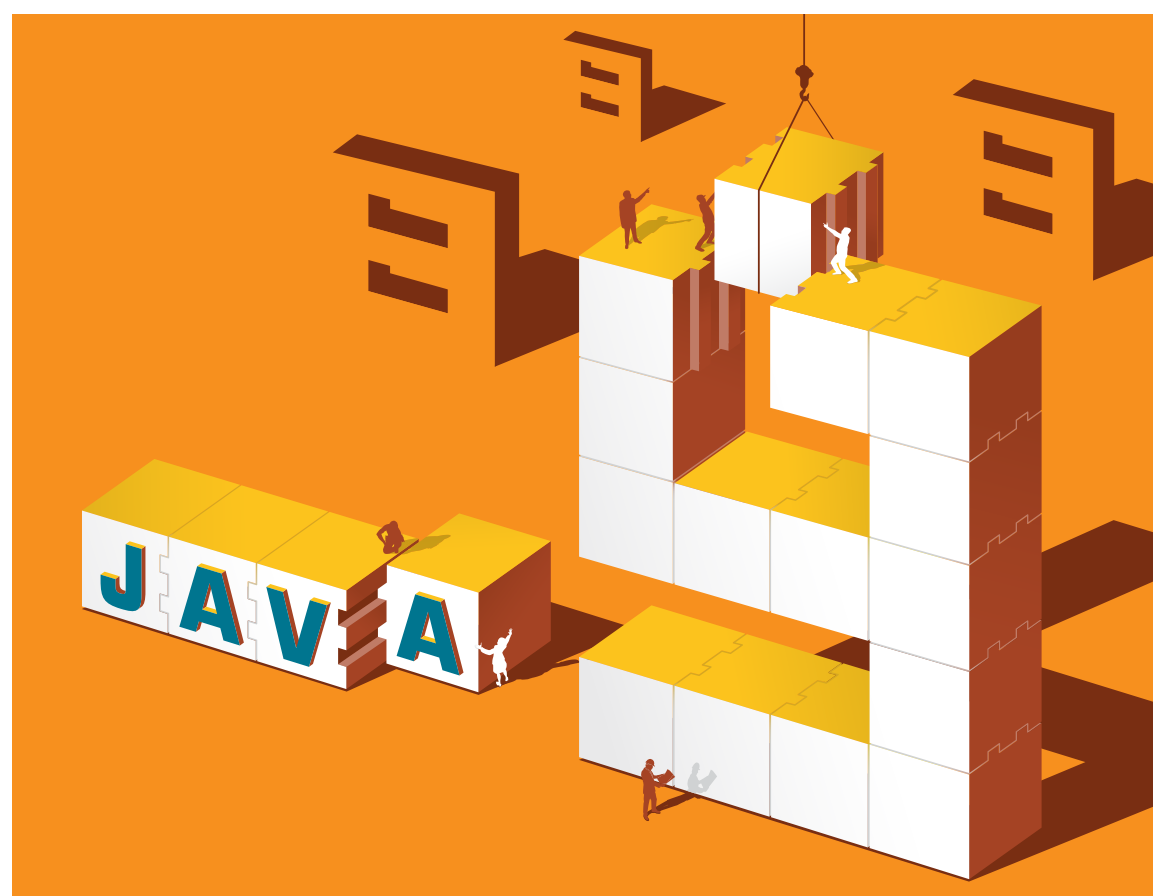
本号の各記事で説明しているように、Java 9にはモジュール以外の優れた機能が多数搭載されています。言語チームとプラットフォーム・チームは、Javaプログラミングをさらに簡潔で楽しめるものにする多くの便利な新機能を作成してきました。Simon Ritter氏の

記事(5ページ)では、そうした多数の便利な追加機能の概要を説明します。この記事の補足として、15ページから始まる記事では、Collection、Stream、Iteratorの新機能について詳しく説明します。また、モジュールを使用する予定がないとしても、Java 8のコードをJava 9でコンパイルして実行する方法をTrisha Gee氏が説明します(11ページ)。

Java 9のコードを実行する別のやり方は、JShellを使用する方法です。JShellは、

このリリースにバンドルされる新しいRead-Evaluate-Print Loop (REPL) です。JShellの入門記事(21ページ)では基本事項を紹介し、HTTP/2に関する記事(32ページ)ではJShellのその他の使用例を紹介します。HTTP/2テクノロジーはネットワーク・プログラミングを容易にする機能で、Java 9で導入される新しいインキュベータ・システムに含まれています。ここに見られるテクノロジーは、今後のリリースにバンドルされる可能性があります。HTTPをよく使用する方は、この記事をしっかりお読みください。

これらの記事の他に、恒例のJava言語クイズ、イベント・カレンダー、編集部へのご意見も掲載しています。どうぞお楽しみください。そして、Java 9に関して今後取り上げて欲しいテーマが他にあれば、お知らせください。





SIMON RITTER

開発者が喜ぶJDK 9の9つの新機能

このリリースに含まれるモジュール以外の多くの新機能

JDK 9の大きな新機能は、モジュール化されたJDKの導入と一体になったJavaプラットフォーム・モジュール・システムですが、他にも多くの新機能が搭載されています。本記事では、開発者にとって特に興味深いと思われるJDK 9の9つの新機能に焦点を当てて紹介したいと思います。さらに詳しい情報を調べることができるように、該当する場合は関連するJDK拡張提案 (JEP) の番号も掲載しています。

コレクションのファクトリ・メソッド (JEP 269)

コレクションは、アプリケーションでデータ項目のグループ (セットと書きかけましたが、この表現では誤解を生むかもしれません) をひとまとめにし、さまざまな便利な方法を使ってそのデータを操作するおなじみの方法です。

その最上位には、List、Set、Mapの概念を抽象化したインタフェースが存在します。

しかし、今まで問題だったのは、Javaにはあらかじめ定義されているデータを使ってコレクションを作成するシンプルな方法がなかったことです。構造的に不変なコレクション (すなわち、要素への参照を追加、削除、変更できないもの) を作る場合は、さらに多くの作業が必要になります。

JDK 8を使った簡単な例を見えます。

```
List<Point> myList = new ArrayList<>();
myList.add(new Point(1, 1));
myList.add(new Point(2, 2));
myList.add(new Point(3, 3));
myList.add(new Point(4, 4));
```

```
myList = Collections.unmodifiableList(myList);
```

このコードがひどいとは言えませんが、4つのPointを格納した不変リストを作成するために6行も必要としています。JDK 9では、コレクションのファクトリ・メソッドを使ってこの問題に対処しています。

この機能は、静的メソッドをインタフェースに含めることを可能にした、JDK 8で導入された変更点を利用したものです。この変更が意味するのは、インタフェースを実装する大量のクラスではなく、トップレベルのインタフェース (Set、List、Map) に対して、必要なメソッドを追加すればよいということです。

JDK 9を使って先ほどの例を書き換えてみます。

```
List<Point> list =
    List.of(new Point(1, 1), new Point(2, 2),
            new Point(3, 3), new Point(4, 4));
```

はるかにシンプルなコードになりました。

各コレクションを使う場合に適用されるルールは、(ご想像のとおり、) このファクトリ・メソッドを使う場合にも適用されます。そのため、Setを作成する場合は、重複した引数を与えることはできず、Mapを作成する場合は、重複したキーを与えることはできません。また、コレクションのファクトリ・メソッドで、値にnull値を使うことはできません。Javadocドキュメントには、メソッドの呼出し方法が詳しく記載されています。[編集注: コレクションについては、「Java 9で更新されたコア・ライブラリ: CollectionとStream」(21ページ) という記事で詳しく取り上げられています。]

Optionalクラスの拡張

Optionalクラスは、コードでNullPointerExceptionが起きる可能性がある場所を減らすため、JDK 8で導入されました（そして、Stream APIをさらに堅牢なものにするためによく使われています）。

JDK 9では、Optionalに4つの新しいメソッドが追加されます。

- `ifPresent(Consumer action)` : 値が存在する場合、その値を使用して`action`を実行します。
- `ifPresentOrElse(Consumer action, Runnable emptyAction)` : `ifPresent`と同様ですが、値がない場合は`emptyAction`を実行します。
- `or(Supplier supplier)` : このメソッドは、常にOptionalがあることを保証したい場合に便利です。
`or()`メソッドは、値が存在する場合は同じOptionalを、存在しない場合は`supplier`が作成する新しいOptionalを返します。
- `stream()`: 値が存在するかどうかにより、要素が0個または1個のストリームを返します。

Stream APIの機能強化

データのコレクションからストリームのソースを作成できると便利です。JDK 8では、Collections APIの外部でこの作成を行ういくつかのメソッドが提供されていました（たとえば、`BufferedReader.lines()`があります）。JDK 9では、いくつかの新しいソースが追加されます。たとえば、`java.util.Scanner`や`java.util.regex.Matcher`などです。

また、JDK 9では、Streamインタフェースに4つのメソッドが追加されます。

最初に紹介するのは、関連する2つのメソッド

takeWhile(Predicate)とdropWhile(Predicate)です。この2つのメソッドは、既存のlimit()メソッドとskip()メソッドを補完するものですが、固定の整数値ではなく、Predicateを使用します。takeWhile()メソッドは、Predicateのtest()メソッドがtrueを返すまで、入力ストリームから要素を取得して出力ストリームに渡し続けます。dropWhile()メソッドは、その逆の動作をします。つまり、Predicateのtest()メソッドがtrueを返すまで、入力ストリームからの要素を破棄します。その後、入力ストリームの要素のうち残されたものを出力ストリームに渡します。

この2つのメソッドで、順序付けられていないストリームを扱うときは、十分注意してください。条件が一度満たされただけで、出力に渡される要素の状態が変わるため、予期しないストリーム要素を受け取る場合や、取得されると考えていた要素が存在しない場合があります。

3つ目の新しいメソッドは、`ofNullable(T t)`です。このメソッドは、渡される値が`null`かどうかにより、要素が0個または1個のストリームを返します。このメソッドを使うと、ストリームを構築する前の`null`チェックを省くことができ、非常に便利な場合があります。考え方は、先ほどのセクションで紹介した、`Optional`クラスの新しいメソッドである`stream()`によく似ています。

新しいストリーム・メソッドの最後を飾るのは、静的メソッド `iterate()` の新バージョンです。このメソッドのJDK 8バージョンは、シードとして1つのパラメータを受け取り、無限に続くストリームを出力するものでした。JDK 9では、3つのパラメータを受け取るメソッドがオーバーロードされています。このメソッドを使うと、ストリームで標準のforループ構文と同じことができます。たとえば、`Stream.iterate(0, i -> i + 1)`と記述すると、0から4までの整数のストリームが得られます。

Read-Eval-Print Loop : jshell (JEP 222)

JDK 9には、新しいRead-Eval-Print Loop (REPL) コマンドライン・ツールであるjshellが含まれます。IDEを使う場合、コードの編集、コンパイル、実行が必要になりますが、jshellではIDEとは異なり、Javaコードのインタラクティブな開発やテストが可能になります。jshellは絶えず、ユーザーの入力を読み取って評価し、入力値や、入力引き起こした状態変化の説明を出力します。そのため、開発者はすばやくコードの一部のプロトタイピングを行うことができます。

より簡単に利用してもらえるように、ishellには、編集可能な履歴、タ

JDK 9には、新しいRead-Eval-Print Loop (REPL) コマンドライン・ツールであるjshellが含まれます。jshellでは、Javaコードのインタラクティブな開発やテストが可能になります。

必要があります。今回の変更によって、既存の変数を（それが実質的に finalである限り）再割当てせずに使用できるようになります。次に例を示します。

```
try (Resource r = alreadyDefinedResource) ...
```

JDK 9では、次のように書くことができます。

```
try (alreadyDefinedResource) ...
```

- インタフェースでprivateメソッドを使用できるようになります。インタフェースは、JDK 8で大きく変わりました。まず、デフォルト・メソッドが導入され、下位互換性を損なうことなく既存のインタフェースに新しいメソッドを追加できるようになりました。これは、インタフェースに動作を含められるようになったことを意味した（Javaに動作の多重継承が初めて導入されました）ため、静的メソッドが追加されたことも理にかなっていました。JDK 9では、インタフェースにprivateメソッドが導入されるため、インタフェース内でカプセル化を維持したまま、一般的なコードをメソッドとして抽出できます。
- 匿名クラスでダイヤモンド演算子を使用できるようになります。JDK 8で改善され、JDK 9でさらに強化されるのが型推論です。このよい例として挙げられるのが、明示的にパラメータの型を指定しなくてもラムダ式を使用できることです。この機能を実現するためには、Javaコンパイラの型推論処理を大きく書き直す必要がありました。JDK 9でのダイヤモンド演算子の変更は、このコンパイラの変更を活用しています。ここでの問題は、今までもそうだったように、匿名クラスでダイヤモンド演算子を使うと、型推論によって型が一意に決まらない場合があります。コンパイラでは、一意に決まらない型を表現することもできますが、そういった型はクラスのシグネチャ属性を使ってJVMに伝えることはできません。JDK 9では、推論される型が一意に決まる場合に限り、ダイヤモンド演算子を匿名クラスで使うことができるようになります。次に例を示します。

```
List<String> myList = new ArrayList<>() {  
    // メソッドのオーバーライド  
};
```

- アンダースコア1つは、識別子として有効ではなくなります。筆者は、Java 9のプレゼンテーションを行う際に、変数名としてアンダースコア1つを使ったことがあるかをよく尋ねるのですが、幸いなことに、使ったことがあると答える方はほとんどいません。この変更が行われる理由は、今後のJDKリリース（おそらくJDK 10）で、アンダースコア1つが変数名として有効となるのが、ラムダ式のみになる予定であるためです。これは妥当なことです。ラムダ式の引数が1つだけで、その引数をラムダ式の本体で使用しない場合、アンダースコア1つを使うこともできます。次に例を示します。

```
__->getX()
```

(アンダースコアを変数名として使いたいという方は、2つ以上のアンダースコアを使うことができます。)

- @SafeVarargsアノテーションの使用が拡張されます。現在、このアノテーションは、コンストラクタ、静的メソッド、finalメソッドでのみ使用できます。これらはすべてオーバーライドできないメソッドです。privateメソッドはサブクラスでオーバーライドできないため、privateメソッドでもこのアノテーションを使用できるようになるのは妥当なことです。

スピン・ウェイト・ヒント (JEP 285)

この変更は小さなものであり、追加されるメソッドは1つだけです。しかし、このメソッドがThreadクラスに追加されることには大きな意味があります。実は、この機能には別の興味深い側面があります。というのも、オラクル以外の企業が提案したJEPで最初に採用されるものだからです（このJEPは、Azul Systemsによって提案されたものです）。

onSpinWait()メソッドは、スレッドが現在プロセッサのスピン・ループ中であることをヒントとしてJVMに伝えます。JVMやハードウェア・プラットフォームがスピン・ループ時の最適化をサポートしている場合、このヒントが使われる可能性があります。そうでない場合、この呼出しは無視されます。一般的には、スレッド間の待機時間の削減や、消費電力の削減といった最適化が行われます。

ProcessHandleインターフェイスは、ネイティブ・プロセスの識別と制御を行います。

まとめ

もうおわかりと思いますが、JDK 9には特に開発者を対象とした便利で強力な多くの新機能が含まれています。こういった機能を備えたこのリリースに、できるだけ早く開発を移行することを検討してはいかがでしょうか。 </article>

Simon Ritter (@speakjava) : Azul Systemsの副CTO。
イギリスのブルネル大学で物理学の学士号を取得し、1984年からITビジネスに携わる。1996年にSun Microsystemsに入社後、Javaの開発やコンサルタントに従事し、1999年からは、コアJavaプラットフォームやクライアント・アプリケーション、埋め込みアプリケーションを中心に、開発者向けにJavaテクノロジーについての講義を行っている。

注目のJDK拡張提案

今号のJava Magazineで詳しく特集しているように、JDK 9リリースには多くの魅力的な機能が含まれています。その一方で、JDK拡張提案241 ([JEP 241](#)) が承認されたため、jhatヒープ解析ツールがバンドルされなくなります。jhatは、もともとJava 6がリリースされた際にJDKに組み込まれるようになったものです。このツールは、HATと呼ばれていたプロジェクトに由来するもので、JDKに組み込まれる前は、すでに消滅したjavatoolsプロジェクトの一部でした。

jhatは最近のJDKリリースにも含まれていましたが、常に試験運用版のツールに分類されており、公式ドキュメントでもその扱いになっています。しかし、年月の経過とともに他のヒープ・プロファイラに先を越され、ほとんどメンテナンスもされていないため、長年にわたり衰退の一途をたどっています。

まだjhatを使っているという方は、別のツールを考慮すべきです。検討する価値がある選択肢には、[VisualVM](#)があります。また、プログラムによるレベルであれば、Java HotSpot Serviceability Agent APIも使うことができます。後者は、書籍『Java Performance Companion』で徹底解説されています。この本は、本誌英語版の2016年9月/10月号に書評が掲載されています。

商用のスタンドアロン版のツールも利用できます。特に有名なものには、[JProfiler](#)や[YourKit](#)などがあります。また、現在の優れたIDEには、開発者にとって使いやすいJVMヒープ調査ツールがバンドルされています。

いずれにしても、jhatに代わるよいツールを必ず見つけることができるでしょう。





TRISHA GEE

Java 8からJava 9への移行

段階的に作業を進めれば、モジュールを採用しなくても全部または一部の機能の導入が可能

本記事では、モジュールに移行せずにJava 9の新機能のいくつかを使ってJava 8のコードを実行する手順を紹介します。基本的には、コードをコンパイルして実行するために必要な方法について説明します。

はじめに

まず、JDK 9の最新バージョンをダウンロードしてインストールします。本記事の執筆時点では、まだEarly Accessリリースの段階ですが、[こちら](#)で確認できます（本記事の例では、build 9-ea+166を使用しています）。

おそらく皆さんは、システムに対するJava 9の影響がわかるまで、デフォルトのJavaバージョンはそのままだとのお考えでしょう。そこで、\$JAVA_HOMEを更新して新しくインストールしたバージョンを指すようにはせずに、新しい環境変数\$JAVA9_HOMEを作成することをお勧めします。本記事では、このアプローチを採用します。

また、こちらの詳細な[移行ガイド](#)を参照し、どのような手順が必要になるかを確認してください。

作業に着手する前に

Java 9への移行に必要なと見込まれる作業量は、Java 9のダウンロードやインストールをしなくても調べることができます。Java 8で利用できるツールがいくつかあり、そこから十分なヒントを得ることができます。

コンパイラの警告:最初に挙げられるのは、当然ではありますが、コンパイラの警告の確認です。この警告から、潜在的な問題についてわか

ることがあります（図1参照）。

筆者は、コードのコンパイルや警告の表示に使うIDEとしてIntelliJ IDEAを利用しています。ただし、図1に示すメッセージ・ボックスの内容は、コマンドラインからjavacを実行した場合に表示される警告と同じものです。NetBeansやEclipseなどの他のIDEにも同様の機能があります。図1には、アンダースコア1つのみをフィールド名として使っているコードがあるという警告が示されています。Java 9では、JDK拡張提案（JEP）213の一環として、アンダースコア1つのみの名前は有効な識別子の名前から除外されています。その理由は、Javaの今後のバージョンでラムダ式をさらに簡単に使用できるようにするためにアンダースコアが

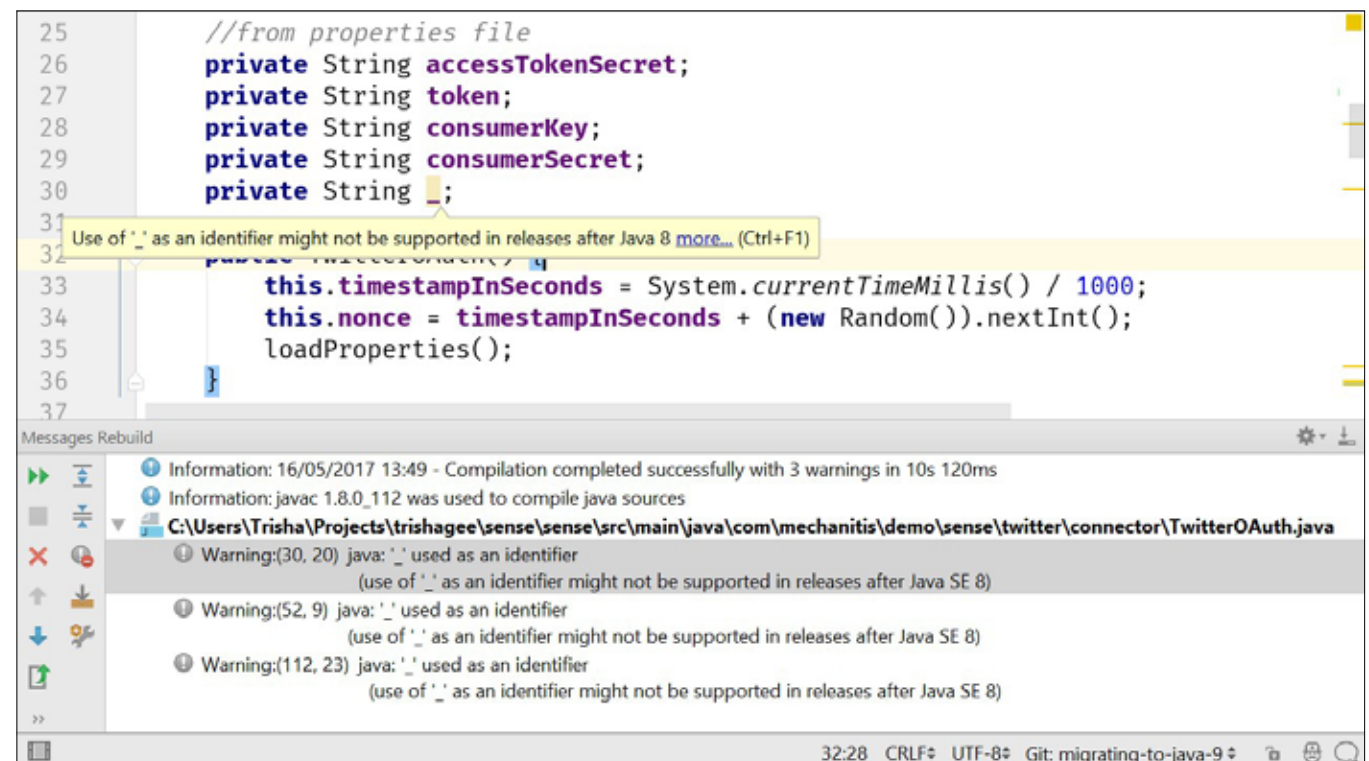


図1:発生する可能性がある問題に関するコンパイラ警告の例

使われる可能性があるからです。識別子の名前としてアンダースコア1つのみが使われている場合、その名前を変えることが必要になります。なお、_fieldNameのようにアンダースコアを含むフィールド名は引き続き有効です。問題になるのは、アンダースコアのみを含み、他の文字を含まない識別子の名前です。

jdepsで問題を引き起こす依存性の特定: 今回のリリースにおける目的の1つであり、Java 9でもっともよく知られた機能と言って間違いないのは、JDK開発者が内部実装を隠蔽できるようになったことです。これまで、Java開発者はJDK内の非サポートクラス、例えば sun パッケージ内のクラスに依存しないというガイドラインに従うよう求められていました。しかし、開発者が使うべきでないものを言語やフレームワーク、ライブラリの側で隠蔽できる方が、アプローチとしてはるかに優れています。Java 9では隠蔽される内部APIがコードに含まれていないことを確認するために、jdepsというツールを使用することができます。

図2に、筆者のプロジェクトで使っているコンパイル済みクラスの場合を示します。

多くのチュートリアルでは、JARに対してjdepsを使う方法が示されていますが、このツールはクラス・ファイルに対して使うこともできます。筆者の場合、コマンドラインを使ってクラス・ファイルの格納場所(図2から、%PROJECT%\%MODULE%\build\classesであることがわかります)に移動し、そこから-jdkinternalsフラグを指定してjdepsを実行して

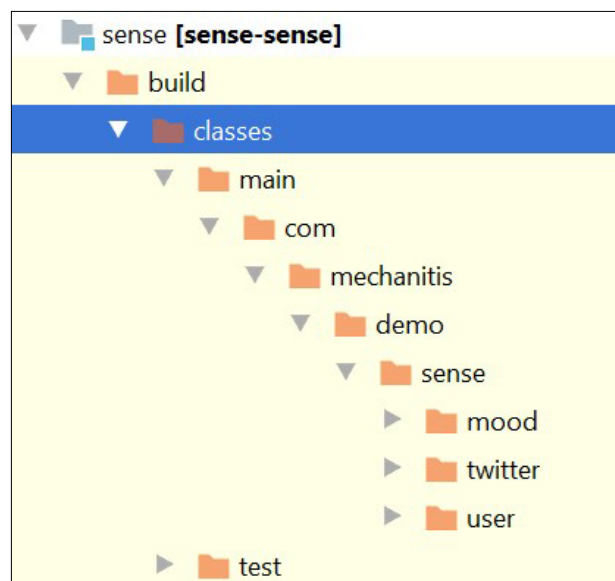


図2: プロジェクトでのコンパイル済みクラスの格納場所

います。筆者はWindowsを使っているため、実行するコマンドラインは次のようになります。mainは、クラスを含むディレクトリの名前です。

```
> %JAVA_HOME%\bin\jdeps -jdkinternals main
```

このコマンドを実行すると、図3に示す結果が得られます。

メッセージから、TwitterOAuthクラスで内部クラス sun.misc.BASE64Encoderとsun.misc.Unsafeが使われていることがわかります。エンコーダについては、Java 8で新しく追加されたjava.util.Base64に置き換えることがこの結果で提案されています。unsafeについては、この機能すべてを置き換える適切な機能がまだ存在しないため、特別なケースとなります。

JEP 260では、どの内部APIを隠蔽し、どのAPIをアクセス可能にするかという議論が行われています。JavaチームのメンバーであるMark Reinhold氏は、Javaチームがどのように各APIを扱うかについて、JDK関連のある主要メーリングリストへのメールの中で述べています。概要は次のとおりです。

- JDK 8で代替機能がサポートされている場合、JDK 9ではその機能をカプセル化する。
- JDK 8で代替機能がサポートされていない場合は、外部コードからアクセスできるようにするため、JDK 9ではその機能をカプセル化しない。

```
C:\Users\Trisha\Projects\trishagee\sense\sense\build\classes>"c:\Program Files\Java\jdk1.8.0_112\bin\jdeps" -jdkinternals main
main -> c:\Program Files\Java\jdk1.8.0_112\jre\lib\rt.jar
com.mechanitis.demo.sense.twitter.connector.TwitterOAuth (main)
-> sun.misc.BASE64Encoder          JDK internal API (rt.jar)
-> sun.misc.Unsafe                JDK internal API (rt.jar)

Warning: JDK internal APIs are unsupported and private to JDK implementation that are
subject to be removed or changed incompatibly and could break your application.
Please modify your code to eliminate dependency on any JDK internal APIs.
For the most recent update on JDK internal API replacements, please check:
https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool

JDK Internal API          Suggested Replacement
-----
sun.misc.BASE64Encoder    Use java.util.Base64 @since 1.8

C:\Users\Trisha\Projects\trishagee\sense\sense\build\classes>
```

図3: jdepsの実行結果

- JDK 9で代替機能がサポートされる場合、JDK 9では非推奨としてその機能をカプセル化する。JDK 10ではその機能を削除する可能性もある。

コンパイラの警告とjdepsでは、コード内でJava 9において問題を引き起こす可能性がある部分を示すことができ、場合によっては解決策を提示することも可能です。こういった問題は、以前のバージョンのJavaをまだ実行している間に修正することもできます（提示される代替クラスをそのバージョンで 사용할 ことができる場合）。そうしておく と、Java 9への移行が簡単になります。

Java 9での実行

先ほど紹介した移行ガイドでは、再コンパイルや変更を行う前に、アプリケーションをJava 9で実行することが推奨されています。ここで試しておきたいものの1つが継続的インテグレーション (CI) 環境です。Java 9でアプリケーションやテストを実行する部分以外は、現在のバージョンのJavaでビルドしたアーティファクトを使用できます。

ただし、本記事の執筆時点では、多くの一般的なCIサーバーやGradleなどの一部のビルド・ツールは、まだ完全にはJava 9をサポートしていません。プロジェクトでMavenを使っている場合は、TeamCityなどのいくつかのCIサーバーにおいて、Java 9でコードのコンパイルや実行ができるようになる予定です。しかし、Java 9の一環としてJDKとJREの構造が変わっているため、正しく動作させるために更新が必要なツールもあります。

そのため、チームでどのようなCIサーバーやビルド・ツールを使っているかを調査し、そういったサーバー、ツールや、ツールチェーンの関連部分が現時点でJava 9での実行をサポートしているかどうかを確認する必要があります。

Java 9でのコンパイル

先ほど紹介したとおりに作業を行ってきた場合、Java 9でコンパイルするのは簡

単なはずです。図1と図3で明らかになったエラーを修正せずに筆者のコードをJava 9でコンパイルすると、予想された2つのエラーが発生します。アンダースコア1つはフィールド名として使用できないというエラーと、BASE64Encoderにアクセスできなくなったというエラーです。いずれのエラーも簡単に修正できます。最初のエラーに対しては、フィールド名を変更します。2つ目のエラーに対しては、`java.util.Base64`をインポートして、次のコードを書き換えます。

```
String encodedString =
    new BASE64Encoder().encode(bytes);
```

新しいコードは次のようになります。

```
String encodedString =  
    Base64.getEncoder().encodeToString(bytes);
```

sun.misc.UnsafeにはJava 9でも引き続きアクセスできるため、このAPIを使用している部分については（現時点では）エラーも警告も表示されません。コードに影響を与える可能性があるその他の変更点の例については、「削除または変更されるAPI」を参照してください。

筆者のアプリケーションが依存している外部ライブラリはすべて Java 9 で動作するため、前述の2つの修正を行うだけで、すべてを正常にコンパイルして実行できます。ただし、使っているライブラリの一部が Java 9 に対応していないことがわかるかもしれません。[こちら](#)には、Java 9 でテストされている主なオープン・ソース・ライブラリのリストが掲載されています。ライブラリに問題がある場合は、Java 9 をサポートする更新版がないかを確認する必要があります。

想定外の動作

Java 9では、アプリケーションに影響を与える可能性があるいくつかの変更が行われています。中には、コンパイル・エラーにならず、違う動作をするものもあるかもしれません。前述の移行ガイドには、そういった変更点が含まれています。開発者が影響を受ける可能性がある変更点の一部を次に示します。動作の変更が疑われる場合にそれらの変更点を調べられるように、関連するJEP番号とともにリンクを以下に記載しました。

- JEP 231:起動時のJRE/バージョン選択の廃止

Java 9では、アプリケーションに影響を与える可能性があるいくつかの変更が行われています。中には、コンパイル・エラーにならず、違う動作をするものもあるかもしれません。

- [JEP 240: JVM TI hprofエージェントの削除](#)
- [JEP 241: jhatツールの削除](#)
- [JEP 260: 大半の内部APIのカプセル化](#)
- [JEP 289: Applet APIの非推奨化](#)
- [JEP 298: デモとサンプルの削除](#)
- [JEP 214: JDK 8で非推奨となったGCの組合せの削除](#)
- [JEP 248: G1ガベージ・コレクタのデフォルト化](#)
- [JEP 271: GCロギングの統合](#)
- [JEP 158: JVMロギングの統合](#)
- [JEP 223: 新しいバージョン文字列スキーム](#)
- [JEP 245: JVMコマンドライン・フラグ引数の検証](#)

まとめ

本記事では、モジュール機能に踏み込まずにJava 8のコードをJava 9に移植する方法を紹介しました。この説明は、Simon Ritter氏の記事「開発者が喜ぶJDK 9の9つの新機能」などで紹介されている多くの画期的なJava 9機能を使ってみたい読者を対象としたものです。[編集注: 本号を印刷に回した時点では、モジュール機能の仕様がJava Community Processで最終決定されていません。そのため、モジュール関連の機能については、今後のJava Magazineで取り上げたいと考えています。] [</article>](#)

Trisha Gee: Javaの高パフォーマンス・システムを専門とするJava Champion。Seville Java User Group (SVQJUG) のリーダーで、オープン・ソース開発も行っている。JetBrainsでIntelliJ IDEAのデベロッパ・アドボケートとして活躍。

ORACLE®



The Best Resource for Modern Cloud Dev

The Oracle Developer Gateway is the best place to jump-start modern cloud development. With free trials of PaaS and IaaS, documentation galore, piles of downloads, and tutorials for leveling up your skills, it's the resource of choice for developers working in Java, mobile, enterprise apps, and more.

Trials. Downloads. Tutorials.
Start here: developer.oracle.com

developer.oracle.com

#developersrule





RAOUL-GABRIEL URMA
RICHARD WARBURTON

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

Collection、Stream、Iteratorのすべてに新機能が追加

Java 9では、Stream、Collector、Optional、CompletableFutureなどのJava 8機能に多くの調整が加えられており、Collections APIも拡張されています。本記事では、Collection、Stream、Collectorの新機能の使い方に注目します。次号では、OptionalとCompletableFutureを題材に、信頼性について考えていきたいと思います。

コレクション・ファクトリ

Java 9のCollectionフレームワークには、一連の新しいファクトリ・メソッドが追加されています。まずは、このファクトリ・メソッドが解決しようとしている問題から見ていきます。最初に、いくつかのString値を格納したListのインスタンスを作成します。

```
List<String> values = new ArrayList<>();
values.add("Java 9");
values.add("is");
values.add("here");
```

コードを見ると、単純でよく行う作業にしては、長いものになっています。もっとも、Listのインスタンスを作成する方法は上記だけではありません。[Arrays.asList\(\)](#)はJava 5より前から存在していましたが、もともとは配列しか受け取ることができませんでした。この機能はJava 5で拡張されて可変長引数を受け取れるようになり、広く使われるようになりました。

```
List<String> values =
    Arrays.asList("Java 9", "is", "here");
```

この改善のおかげで便利な機能になってはいるものの、[Arrays.asList\(\)](#)が返すListは、少しばかり特殊なものです。Listに要素を追加しようとする、UnsupportedOperationException例外がスローされます。結局これは不変リストとなることから、特に問題ないという方もいるでしょう。しかし、早まってははいけません。実は、このリストは単に配列をラップしただけのものです。そのため、[set\(\)](#)操作を行うとバックエンドの配列が変更されます。実際には、ラップされる配列を保持している場合、その配列も変更されることがあります。SetやQueueを作りたいという方も残念でした。[Arrays.asSet\(\)](#)は存在しません。この問題を解決する一般的な方法は、オーバーロードされた、コレクションのコンストラクタを使うことです。

```
Set<String> values =
    new HashSet<>(Arrays.asList(
        "Java 9", "is", "here"));
```

最初の例と同様に、これはとても冗長です。プログラミング言語の中には、コレクション・リテラルを追加することによってこの問題を解決する機能を提供しているものもある点に注意してください。その場合、個々の値からコレクションをインスタンス化する何らかの構文が提供されます。次に示すのは、Groovyでの例です。

```
def values = ["Hello", "World", "from", "Java"] as Set
```



これを実装する方法の1つは、値からコレクションを作成する何らかのメソッドを提供して可変長引数のコンストラクタを使うことです。そうすることによって、コレクション・リテラルと同じように、短い構文でコレクションを作成できるようになります。Java 9では、このアプローチが使われています。そのため、次のようにしてはるかに簡潔に書くことができます。

```
List<String> list =  
    List.of("Java 9", "is", "here");  
Set<String> set =  
    Set.of("Hello", "World", "from", "Java");
```

Mapにも、同様のファクトリ・メソッドが追加されています。Mapの要素は単一の型ではなく、キーと値になっているため、仕組みは異なります。Mapでは、10エン트리分までのコンストラクタがオーバーロードされており、キーと値のペアを受け取ることができるようになっています。たとえば、次のようにして人と年齢のマップを作成できます。

```
Map<String, Integer> nameToAge
    = Map.of("Richard", 49, "Raoul", 47);
```

Mapで可変長引数を使うことは、少しばかり難しくなっています。キーと値の両方が必要ですが、Javaではメソッドに2つの可変長引数パラメータを持たせることはできません。そのため、`Map.Entry<K, V>`オブジェクトを作成する静的メソッド`entry()`と、可変長引数としてそれを受け取るメソッドを追加することによって、汎用的なケースに対応しています。

```
Map<String, Integer> nameToAge =  
    Map.ofEntries(entry("Richard", 49),  
                   entry("Raoul", 47));
```

ここでの目的は、冗長性だけでなく、プログラマーが間違える可能性も減らすことです。最近追加されたコレクションはすべて、nullを要素とすることができなくなっています。今回紹介しているコレクションもそれを踏襲しています。それによって、コレクションの内部でnull値が参照されることに関連するバグが起こりにくなります。さらに、内部実装もシンプルになります。

また、JDKのほとんどのコレクションと比べて大きく異なるのは、不変コレクションであることです。アプリケーションの一部がある部分の状態を書き換えた場合、別のコンポーネントがその状態に依存しているときは、問題が起きる可能性があります。不変オブジェクトではそのようなことは起こらないため、バグが起こりにくなります。不変性は、関数型プログラミングで長い間推奨されてきた考え方です。関数型プログラミングの話題が登場したところで、Java 9でのStream APIの更新について見ていきます。

Stream

Java 8で追加された大きな機能がストリームです。開発者がストリームを使って書くコードは、解決しようとする問題を表現するものになることが極めて多く、通常はコードの量も少なくて済みます。Java 9では、ストリームに細かい改善が行われています。

ofNullable: Streamインタフェースには、単一の値と可変長引数を受け取ることができるようにオーバーロードされた2つの**of()**というファクトリ・メソッドがあります。このメソッドを使うと、あらかじめ決められた値からストリームを作成できます。これは、ストリームのコードをテストしようとする場合や、いくつかの値を持つストリームをインスタンス化したい場合に大変便利です。Java 9では、**ofNullable()**ファクトリ・メソッドが追加されます。それでは、この機能の使い方について見ていきます。

Javaアプリケーションで設定ファイルを配置できる場所を探そうとする場合について考えてみます。いくつかのプロパティ、ここでは、`app.config`と`app.home`を使うものとします。Java 8では、次のようなコードを書くことになります。

```
String configurationDirectory =  
    Stream.of("app.config", "app.home", "user.home")  
        .flatMap(key -> {  
            final String property =  
                System.getProperty(key);  
            if (property == null)  
            {  
                return Stream.empty();  
            }  
            else
```



```
String configurationDirectory =  
    Stream.of("app.config", "app.home", "user.home")  
        .flatMap(key ->  
            Stream.ofNullable(System.getProperty(key)))
```

```
.findFirst()  
.orElseThrow(IllegalStateException::new);
```

takeWhileとdropWhile: eコマースのWebサイトで支払いを処理するアプリケーションについて考えてみます。その日に行われたすべての支払いが金額の降順で並べ替えられたリストを維持管理しています。ビジネス要件として、毎日の終わりにその日に発生した500ポンド以上の支払いについてレポートを生成することを求められているとします。Java 8のStreamを使ってこのコードを書く場合、普通は次のようになります。

```
List<Payment> expensivePayments = paymentsByValue
    .stream()
    .filter(transaction ->
        transaction.getValue() >= 500)
    .collect(Collectors.toList());
```

このアプローチの残念な点は、その日に行われた大量の取引に対する処理を始めると、**filter**操作が入力リストにあるすべての取引に適用されてしまうことです。入力リストが取引額の降順で並べ替えられていることはわかっているため、条件に該当しない取引を見つけたら、それ以降の取引はすべて除外できます。取引のリストのサイズが非常に大きくなると、処理が完了するまでにますます時間がかかり、無用の効率低下を招くことになります。Java 9では、**takeWhile**操作を追加することによって、この問題が解決されます。

```
List<Payment> expensivePayments = paymentsByValue
    .stream()
    .takeWhile(transaction ->
        transaction.getValue() >= 500)
    .collect(Collectors.toList());
```

`filter`は条件に一致するすべてのストリーム要素を保持しますが、`takeWhile`は条件に一致しない要素が見つかったところで処理が終わります。`dropWhile`操作はその逆の動作になります。最初の要素から始めて、条件がfalseである限り要素を破棄し続けます。

ここまでは、要素の順番、すなわち出現順序が決まっているストリームについて見てきました。ストリームの順番は、ソースで定義できます。た

例えば、値のリストから生成したストリームでは、リストの順番が出現順序になります。`sorted()`のように、パイプライン内で出現順序を決めるストリーム操作もあります。すべてとは言いませんが、`takeWhile()`と`dropWhile()`を実際に使用するほとんどのケースでは、入力ストリームが決まった出現順序になっている必要があります。

順序付けられていないストリームに`takeWhile()`を適用するユースケースの1つとしては、ストリーム操作を終了できるようにしたい場合があります。たとえば、無限に続くストリームのデータを処理するストリーム操作があり、アプリケーションがシャットダウンされた際にストリームを停止できるようにしたい場合や、ユーザーがストリーム・パイプラインをキャンセルする必要がある場合が考えられます。`volatile boolean`のフラグなどのような、外部状態を読み取る`takeWhile()`操作を使うと、これを実現できます。ストリーム・パイプラインを停止したければ、その状態を`false`にするだけで、できます。

iterate: これに関連するアップデートとして、ストリームを作成する `iterate()` メソッドに別のバージョンが追加された点が挙げられます。Java 8で導入された従来の `iterate` メソッドには、初期値と、ストリームの次の値を生成する関数を渡します。次の例をご覧ください。

```
IntStream.iterate(3, x -> x + 3)
    .filter(x -> x < 100)
    .forEach(System.out::println);
```

このコードでは、3で割り切れる100未満の数字をすべて出力します。3で割り切れる最初の数字である3から始めて、反復のたびに3を加算しています。その後、100未満の数字のみが残るようにフィルタを適用し、メソッド参照を使って結果を出力しています。単純なコードのように見えるかもしれませんが、実行してみると大きな問題があることに気づきます。ぜひ実行してみてください。

そうです。このプログラムは終了することなく、無限ループで3を加算し続けます。この原因は、数が増え続けることをフィルタが認識できない点にあります。Java 9の新しい`iterate`には、2番目の引数として、反復処理を停止する時点を示す条件を渡すことができます。これを使うと、この問題を解決できます。早速、コードを次のように書き換えてみます。

```
IntStream.iterate(3, x -> x < 100, x -> x + 3)
```

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

プログラムは、99を出力してから終了するようになります。このサンプル・コードでは、プリミティブの`int`値を操作しているため、`IntStream`インタフェースを使っていますが、`iterate()`メソッドはプリミティブのインタフェースでも通常の`Stream`インタフェースでも利用できます。ストリームは、単独でJava 8に追加されたわけではなく、強力なCollectorクラスとともに登場しました。次に紹介するように、このクラスもJava 9で改善されています。

Collector

Java 8で追加されたもう1つの重要な機能がCollectorです。Collectorでは、ストリームの要素をMap、List、Setなどのさまざまなコンテナに集約することにより、データ処理クエリを指定できます。たとえば、Collectorsクラスの[groupingBy](#)コレクタと[summingLong](#)コレクタを使うと、年ごとに費用を合計したマップを作成できます。本記事の以降の部分では、Collectorsクラスから静的メソッドを参照している部分について、静的インポートが行われているものとします。

```
Map<Integer, Long> yearToSum
    = purchases.stream()
        .collect(groupingBy(Expense::getYear,
            summingLong(Expense::getAmount)));
```

それでは、Java 9の新機能とは何でしょうか。Java 9では、Collectorsユーティリティ・クラスに2つの新しいCollector、[Collectors.filtering](#)と[Collectors.flatMapping](#)が追加されています。

本記事の以降の部分では、例を使ってfilteringとflatMapingという新機能の使用方法を続けて紹介します。次に示すのは、これから使用するExpenseクラスとTagクラスの定義です。

```
public class Expense {
    private final long amount;
    private final int year;
    private final List<Tag> tags;
```

```
public Expense(long amount, int year,  
               List<Tag> tags) {  
    this.amount = amount;  
    this.year = year;  
    this.tags = tags;  
}
```

```
public long getAmount() {
    return amount;
}
```

```
public int getYear() {
    return year;
}
```

```
public List<Tag> getTags() {
    return tags;
}
}
```

```
public class Tag {
    private final String content;

    public Tag(String content) {
        this.content = content;
    }
}
```

Collectors.filtering: もう一度「Collector」のセクションの冒頭で示した例について考えてみます。今度は、1,000 ポンドを超える費用について、年ごとの費用リストのマップを作る必要があるとします。

先ほどの説明にもあったため、おわかりとは思いますが、年ごとの費用リストのMapは次のようにして作ります。

```
Map<Integer, List<Expense>> yearToExpenses =
```

```
purchases.stream()  
    .collect(groupingBy(Expense::getYear));
```

ストリームにフィルタを次のように追加することも考えられます。

```
Map<Integer, List<Expense>> yearToExpenses =  
    purchases.stream()  
        .filter(expense ->  
            expense.getAmount() > 1_000)  
        .collect(groupingBy(Expense::getYear));
```

しかしこれでは、ある年の費用がすべて1,000ポンド未満だった場合、マップには、その年のエントリが含まれない（つまり、キーも値もない）ことになってしまいます。

代わりに次のように filtering コレクタを使用して、その年も結果の Map に保持され、空のリストが生成されるようにすることができます。レポートを読むユーザーにとって、年が抜けているのは混乱の元です。その年はデータがないのか、単なるソフトウェアのバグなのかがわからないからです。その特定の年に関してフィルタ条件に該当するエントリがないことをはっきりさせるため、空のリストを返します。

```
Map<Integer, List<Expense>> yearToExpensiveExpenses =
    purchases.stream()
        .collect(
            groupingBy(
                Expense::getYear,
                filtering(expense ->
                    expense.getAmount() > 1_000,
                    toList())));
```

flatMap : flatMappingコレクションは、mappingコレクションの兄のような存在です。年ごとの費用から、年とタグのセットのマップを作成する、つまり、`Map<Integer, Set<Tag>>`を作成する必要があるとします。

まずは、次のようなコードを思いつくかもしれません。


```
expenses.stream()
    .collect(
        groupingBy(
            Expense::getYear,
            mapping(Expense::getTags, toSet())));
```

残念ながら、このクエリは`Map<Integer, Set<List<Tag>>>`を返します。

`flatMap`を使うと、中間のリストをフラット化して単一のコンテナを生成することができます。`flatMap`コレクタは、2つの引数を受け取ります。最初の引数は1つの要素から要素のストリームに変換する関数で、次の引数は単一のフラットなストリームをコンテナに格納するダウンストリームCollectorです。これを使うと、クエリを次のように解決できます。

```
Map<Integer, Set<String>> =
    expenses.stream()
        .collect(
            groupingBy(
                Expense::getYear,
                flatMapping(expense ->
                    expense.getTags().stream(),
                    toSet())));
```

`flatMap`コレクタは、Stream APIの`flatMap`メソッドに関連しています。このメソッドは、入力ストリームの各要素について0個以上の要素を持つストリームを生成する関数を受け取ります。そのため、結果はフラットな単一のストリームになります。

まとめ

Java 9では、Collection、Stream、Collectorに新しい操作が追加されることで、パターンを改善する多くの魅力的な新機能が実現しています。こういった新機能を使うと、解決する問題に近く、読みやすいコードを書くことができます。大きな新リリースがある場合、開発者はもっとも重要な機能のみに注目しがちですが、実のところ、最近のリリースで拡張された機能には、開発者の生産性を向上させるさまざまな改善が施されています。Java 8は素晴らしいリリースでした。そしてJava 9でも、よく使われるパターンがさらに使いやすくなっています。 [</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。



JShell: Javaプラットフォーム用 Read-Evaluate-Print Loop

JShellと呼ばれる新しいRead-Evaluate-Print Loop (REPL) がJDK 9に導入されます。プロジェクトKullaが主導するJShell (JEP 222) は、Javaプログラミング言語の宣言、文、および式を評価するAPIと対話型ツールを開発者に提供することを目的としています。

ように出力されるはずです。

```
java version "9-ea"
```

Java(TM) SE Runtime Environment (build 9-ea+173)

Java HotSpot(TM) 64-Bit Server VM...

JShellを実行するためには、コマンドラインで**jshell**と入力します。

JShellは、コマンドライン・インタフェースを使用した基本的なシェルをJavaで使えるようにする、JDK 9の新しいツールです。Javaプラットフォーム向けの最初の正式なREPL実装でもあります。とはいえ、REPLという概念はすでに多くの言語（GroovyやLispなど）やサード・パーティ製ツール（Java REPLやBeanShellなど）に存在していました。JShellは新しい言語ではなく、Java向けのIDEや新しいコンパイラでもありません。

JShellはUNIXシェルのように動作します。つまり、命令を読み取り、評価し、実行結果を出力し、プロンプトを表示して新しいコマンドを待機します。JShellは、スニペット、状態、ラッピング、命令の修正、前方参照、スニペットの依存性など、いくつかの概念を中心に構築されています。これらの概念について、以下で説明します。

JShellを実行するためには、最新のJDK 9 Early Accessプレビュー・ビルドをダウンロードし、お使いの環境にインストールする必要があります。次に、JAVA_HOME環境変数を設定し、正しくインストールできていることを確認するために`java -version`を実行します。このコマンドからは、次の

```
[pandaconstantin@localhost ~]$ jshell
```

```
| Welcome to JShell -- Version 9-ea
```

| For an introduction type /help intro

プロンプトが表示されているときにコマンドラインに `/help` と入力すると、複数の便利なコマンドのヘルプが表示されます。図1に、このコマンドからの出力の一部を示します。ここでは、多くの重要なコマンドが簡潔に説明されています。

JShellがどのように動作するかを理解するために、いくつかのコード・スニペットを見てみます。スニペットとは、標準Java構文を利用した命令で、1つの式、文、または宣言を表します。簡単なスニペットを次に示します。コマンドの次の行に示されているのは、JShellからの出力です。

```
System.out.println("My JShell snippet");
```

My JShell snippet

(本記事の例では、コマンドラインからJShellに入力したテキストを青色の文字で表し、結果の出力を黒色の等幅文字で表します。)

Javaコードと同様に、JShellでも変数、メソッド、クラスを宣言できます。

```
int x, y, sum
```

$$x \geq 0$$
$$y \geq 0$$

```
int sum ==> 0
```

```
x = 10 ; y = 20 ; sum = x + y;
```

$x ==> 10$

```

/list [<name or id>|-all|-start]
    list the source you have typed
/edit <name or id>
    edit a source entry referenced by name or id
/drop <name or id>
    delete a source entry referenced by name or id
/save [<all>|-history|-start] <file>
    Save snippet to a source file.
/open <file>
    open a file as source input
/vars [<name or id>|-all|-start]
    list the declared variables and their values
/methods [<name or id>|-all|-start]
    list the declared methods and their signatures
/types [<name or id>|-all|-start]
    list the declared types
/imports
    list the imported items
/exit
    exit the jshell
/history
    History of what you have typed
/!
    Re-run last snippet

```

図1: JShellコマンドのリストの一部

$$y == > 20$$

sum ==> 30

```
System.out.println("Sum of " + x + " and " + y +  
    " = " + sum);
```

Sum of 10 and 20 = 30

次に示すのは、有効なクラスの例です。後ほどこのクラスを使用します。

```
class Student {
    private String name ;
    private String classRoom ;
    private double grade ;

    public Student() {

    }

    public String getName() {
        return name ;
    }

    public void setName(String name) {
        this.name = name ;
    }

    public String getClassRoom() {
        return classRoom ;
    }

    public void setClassRoom(String classRoom) {
        this.classRoom = classRoom ;
    }

    public double getGrade() {
        return grade ;
    }
}
```


Successful
Successful

ラッピング

変数の宣言やメソッドの定義を必ずしもクラス内で行う必要はありません。クラス、変数、メソッド、式、文は合成用のクラス内で（見せかけのブロックとして）展開されるため、好みに応じて、トップレベル・コンテキストまたはクラス・ボディ内のどちらかで定義できます。また、簡潔さを重視するのであれば、セミコロンを省略できる場合もあります。

```
String firstName, lastName
firstName ==> null
lastName ==> null

String concatName(String firstName,
String lastName) {
return firstName + lastName ;
}
| created method concatName(String,String)
```

次に示すコードでは、トップレベル・コンテキストで変数とメソッドを宣言しています。前述のとおり、トップレベルではクラスを修正できませんが、次のコードを見ればわかるように、クラス内のメソッドは修正できます。

```
class Person {  
  
    private String firstName ;  
    private String lastName ;  
  
    public String concatName(String firstName,  
        String lastName) {  
        return firstName + lastName;  
    }  
  
}  
| created class Person
```

文や式はそれぞれ固有のネームスペースに作成されるため、コードの全体的な機能を妨げずにいつでも修正できます。

前方参照と依存性

クラスのボディ内では、後から定義されるメンバーを参照できます。コードを評価すると、この参照はエラーになりますが、JShellでは逐次処理されるため、実際にスニペットをコールする前に、不足しているメンバーを記述すれば問題を解決できます。

スニペットAが2つ目のスニペットBに依存している場合、スニペットBに対する変更は即座にスニペットAに伝播されます。また、依存するスニペットを更新すると、mainスニペットも更新されます。依存するスニペットが無効な場合は、mainスニペットも無効になります。

変数を宣言して初期化したら、それらは`list`コマンドを使用して確認できます。次に例を示します。

```
String firstname;  
firstname ==> null  
String lastname;  
lastname ==> null
```

```
double grade;  
grade ==> 0.0
```

```
String getStudentFullName(String fn, String ln) {  
    return fn + " " + ln ;  
}  
| created method getStudentFullName(String,String)
```

```
firstname = "Wolfgang";  
firstname ==> "Wolfgang"
```

```
lastname = "Mozart";  
firstname ==> "Mozart"
```

```
System.out.println("Hello " +  
getStudentFullName(firstname,lastname));
```

Hello Wolfgang Mozart

listコマンドの出力として次の内容が表示されます。

```
1 :String firstname ;
2 :String lastname ;
3 : double grade ;
4 :String getStudentFullName
(String firstname, String lastname) {
    return firstname + " " + lastname ;
}
5 :firstname = "Wolfgang" ;
6 :lastname = "Mozart" ;
7 :System.out.println("Hello" +
getStudentFullName(firstname,lastname));
```

出力に含まれる番号はスニペットの識別子です。スニペットを操作（編集、削除など）する場合にこの番号が役立ちます。コードに含まれる変数、メソッド、クラスをすべてリストすることもできます。すべての変数をリストする例を次に示します。

```
/vars
| String firstname = "Wolfgang"
| String lastname = "Mozart"
| double grade = 0.0
```

変数の値を変更する場合や特定のスニペットを編集する場合は、スニペットの識別子を付けて/editを実行します。次に例を示します。

/edit 6

図2のダイアログ・ボックスが表示され、そこで値を変更できます。
このエディタを使ってlastnameを筆者の姓に変更すると、次のような結果になります。

```
lastname ==> "Drabo"
```


また、`firstname`を筆者の名に変更し、スニペットの識別子（この場合は7番）を使って出力関数を再実行すると、次のようになります。

```

/7
System.out.println("Hello" +
getStudentFullName(firstname, lastname));
Hello Constantin Drabo

```

`/save`コマンドを実行すると、ファイルにスニペットを保存できます。`/open`コマンドを実行すると、そのファイルを開いて実行できます。

```
/save StudentName.js
/open StudentName.js
Hello Wolfgang Mozart
Hello Constantin Drabo
```

JShellではいくつかのキーボード・ショートカットも使用できます。ナビゲーション履歴は、上/下矢印キーを使用して取得できます。[Tab]キーを使うと、IntelliSense機能と同じように、その時点までに入力したテキストに対応する選択肢が表示されます。

まとめ

JShellには、さまざまな用途があります。まず、初心者は、完全なプログラムを書かずにJavaコードを試してみることができます。この意味では、すばらしい学習用ツールだと言えます。また、Webサービスが利用できることを確認する場合や、返される値を確かめる場合など、小さな機能を試してみる際にとても役立つツールです。さらに、JavaFXで簡単なレイアウトを試してみる場合には、とても便利なツールです。

コマンドラインから使用するにせよ、プログラムで使用するにせよ、いずれにしてもJShellはJDK 9の機能の中で特に多用されるものの1つとなるでしょう。

Constantin Drabo：ブルキナファソ在住のソフトウェア・エンジニア。NetBeans Dream Teamのメンバーであり、FedoraProjectのFedoraアンバサダーとしても活動。

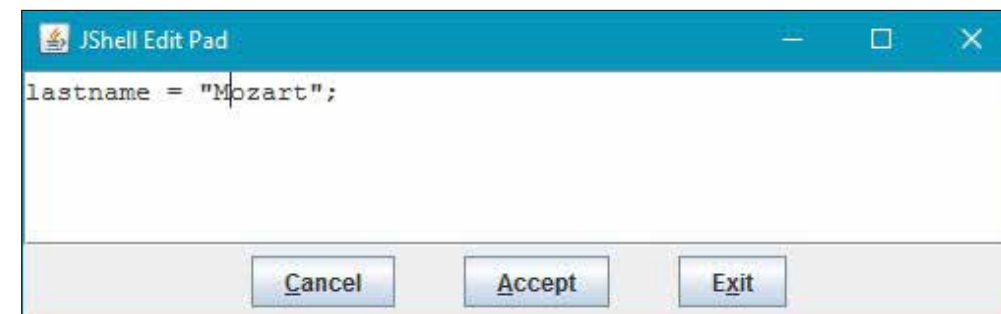


図2: 組み込みのスニペット・エディタ

ブルキナファソ (旧称オートボルタ) 初のJavaユーザー・グループ、FasoJUGの設立者でもある。

[編集注：本記事は、本誌2016年7月/8月号に掲載されたJShell 9チュートリアルを大幅に改訂したものです。]



JIM LASKEY

JDK 9のJavaScriptエンジン Nashorn

便利な追加機能とES6のサポートでNashornがさらに充実

今回の新リリースでは、JDKの組み込みJavaScriptエンジンである Nashornにもさまざまな改善が施されています。まずは、機能拡張について確認する前に、その背景について簡潔に説明するところから始めます。そうすることによって、なぜ今回の変更が行われたのか、納得していただけるはずです。

背景

筆者がNashornの開発を始めたのは2010年後半であり、新しく追加されたinvokedynamicバイトコード命令 (JSR 292) の実験を行う方法を探していた時のことでした。その後、JVMチームはinvokedynamicの試験台としてNashornを採用し、Nashornプロジェクトはinvokedynamicの実装のために大幅なパフォーマンス改善を行いました。

それと並行して、Javaグループでは、クライアントサイドの開発でどの程度JavaScriptが優勢になる可能性があるのかについてや、クライアントでもサーバーでもJavaScriptとJavaの統合が重要な要素になることが話し合われていました。当時のJDKがJavaScript向けに提供していたのは、Mozilla Foundationによるオープンソース実装のRhinoでした。Rhinoは古くなり始めており、開発は縮小される方向に向かっていました。そして、2012年11月にJDK拡張提案 (JEP) 174 「Nashorn JavaScriptエンジン」が承認されました。この承認によって、Javaプラットフォームで実行され、強固かつ安全ですべての機能を搭載した、ECMAScript-262 Edition 5.1 (ES5) の新たな実装の提供に向けて、本格的な作業を始めることが可能になりました。

当初、Nashornは、アプリケーション・サーバー、JavaFXアプリケーション、ユーティリティ、シェル・スクリプト、埋込みシステムなどさまざまなアプリケーションに使われていました。

Nashornは現在も広く使われているものの、その用途は主に3つの領域に落ち着いているように見受けられます。

- クライアントとサーバーの両方で実行できるJavaScriptアプリケーションの開発: これはJavaScriptの世界でアイソモーフィック開発と呼ばれる方式で、デスクトップとモバイルの両方でサービスを提供するフロントエンドを構築したい小規模な店舗に多大なメリットをもたらします。共通のコードベースを持つ1つのプログラミング言語を使って短時間で開発できるからです。アイソモーフィック開発は、大規模システムにも拡張できます。
- ランタイム適応コーディング (ダイナミック・コーディング): 筆者が好んで使う用語は、ソフト・コーディングです。これは、アプリケーションやサーバーのデプロイ後にもアプリケーションの各部を変更できることを意味し、データベースのストアド・プロシージャからアプリケーションの構成管理まで、あらゆることに使われています。
- シェル・スクリプト: 従来はbashやPythonが使われていた領域でJavaScriptを使うことです。

GoogleのV8などのプラットフォームで実行されるJavaScriptネイティブのパフォーマンスと比べた場合の、JVM上で実行されるNashornのパフォーマンスについては、多くの議論が交わされています。NashornはJavaScriptをバイトコードに変換しているため、動作の開始が遅くなります。その後のNashornのパフォーマンスは、ネイティブ・コードを実行するHotSpot次第です。このアプローチは、長時間にわたって実行を続けるサーバー・アプリケーションには向いていますが、一度だけ実行される小さなスクリプトにはあまり適していません。そもそも、Nashornが開発された主な理由は、JavaScriptとJavaをシームレスに同時実行することでした。ほとんどのNashornアプリケーションは、JavaScriptとJavaを協調させるという仕事をうまくこなしています。V8はこのような目的で設計されたものではありません。

ES6のサポート

アイソモフィック開発を行うJavaScript開発者にとってもっとも重要な問題は、クライアントとサーバーのソース・コードに互換性を持たせる必要があることです。ほとんどのブラウザがJavaScript言語サポートの標準レベルとしてECMAScript 6 (ES6) を採用しているため、NashornにもES6をサポートさせることは必須でした。Nashornチームには、JDK 9リリースで完全なES6機能セットを提供するだけの時間はありませんでしたが、今後の更新では、さらに多くのES6機能がサポートされる予定です。ES6サポートを有効化するためには、コマンドラインで`--language=es6`を指定して下さい。

新しいキーワード: 最初に気づくのは、新しいキーワードとして、ES6のブロック・スコープに従う`let`と`const`が実装されていることです。`const`は、不変ローカル変数を作成します。下位互換性のため、`var`はES5.1のセマンティクスを維持しています。

```
const a = 10;  
a = 20; // 定数に設定しようとしているため、TypeError
```

```
let x = 10;
{
  let x = 20; // スコープが異なる
  print(x); // 20を出力
}
print(x); // 外側のスコープの10を出力
```

シンボルのサポート:

シンボルは、ES6で導入された、文字列に似たオブジェクトです。シンボルはプリミティブですが、値ではなく参照を使って比較されます。つまり、すべてのシンボルが一意になるため、開発者がプライベートなオブジェクト・プロパティを作成できるようになります。

```
// 新しい一意なシンボル
let unique = Symbol('optionalName');
myobj[unique] = 'foo';
// 指定した名前の共有シンボル
let shared = Symbol.for('name');
myobj[shared] = 3;
```

最初の例では、個別のシンボルを作成しています。これは、現在のスコープに対してプライベートなプロパティを作成するために使うことができます。2番目の例は、コンテキスト全体でシンボルを共有できるようにする (internする) 方法を示しています。ES6では、iteratorとデフォルトの `toString` 関数の定義に共有シンボルが使われています。シンボルをキーとしたプロパティは、リフレクション操作を使っても参照することはできません。

新しいiteratorプロトコル: ES6では、オブジェクトに対して反復処理を行う新しいプロトコルが提供されています。

```
// Symbol.iteratorプロパティに
// iterator関数を割当て
myobj[Symbol.iterator] = function() {
    return {
        next: function() { ... }
    }
};

// for..of文を使ってmyobjに対して反復処理を実行
for (let id of myobj) {
    print(myobj[id]);
}
```

`for..of`ループが導入されていることに注意してください。Nashornの `for each` 構文もまだ使える見込みですが、同じ機能であっても、`for..of`に切り替えることをNashornチームは推奨しています。

新しいコレクション: ES6およびNashornには、新しいコレクション・クラスであるMapとSetが追加されています。この2つのコレクション・クラスには、新しいiteratorプロトコルが実装されています。

```
let map = new Map();
map.set('foo', 'bar');
map.get('foo'); // -> 'bar'
map.clear();
map.get('foo'); // -> 未定義
```

MapとSetの弱参照バージョンであるWeakMapとWeakSetも実装さ

れています。この2つのコレクションでは、エントリ値が他のどの変数やオブジェクトからも参照されなくなると、ガベージ・コレクタによって削除されます。

テンプレート:テンプレートは、文字列リテラルの新たな形態で、バッククォートが区切り文字になっています。テンプレートを使うと、式の埋込みや、複数行にわたる文字列の記述を行うことができます。Nashornは、関数を使って出力する、ES6の「タグ付き」文字列もサポートしています。

```
// 複数行の文字列
'string text line 1
 string text line 2'
// 式の埋込み
'string text ${expression} string text'
// 「tag」関数を使って出力
tag 'string text ${expression} string text'
```

なお、-scriptingモードでは、従来どおり\$EXEC式の簡略表記としてバッククォートが使われることに注意してください。

バイナリと8進数値:さらに、バイナリと8進の数値リテラルに新しい構文が追加されています。

```
// バイナリ数値リテラル
0b111110111 === 503
// 8進数値リテラル
0o767 === 503
```

このようなES6の機能に加え、Nashornの利便性を向上させる新機能も追加されています。

JavaScript Parsing API

JDK 9では、NashornのParser APIが拡張され、ES6の構文が完全にサポートされています。次に示すのは、サンプルのJavaScriptを解析する例です。

```
// Nashornリソースからparser.jsをロード
//// パーサー・ライブラリをロード
// ES6のクラス宣言を含むサンプル・スクリプト
```

```
var script = "class XYZ {}";
```

```
// スクリプトを解析し、抽象構文木を構築
var json = parse(script);
```

```
// 抽象構文木をJSONに変換して出力
print(JSON.stringify(json, undefined, 4));
```

この結果、次のJSONが出力されます。

```
{
  "type": "Program",
  "body": [
    {
      "type": "VariableDeclaration",
      "declarations": [
        {
          "type": "VariableDeclarator",
          "id": {
            "type": "Identifier",
            "name": "XYZ"
          },
          "init": {
            // ...
          }
        }
      ]
    }
  ]
}
```

Parsing APIは、コードの分析や計測機能のインジェクションにも使用できるため、開発者は高い柔軟性を実現できます。NetBeansは、JavaScript機能の処理のほとんどでこのAPIを使用しています。

\$EXEC 2.0

Nashornで特に評判のよい機能は、シンプルなバッククォート式を使ってプロセスをフォークできることです。珍しいこの機能によって、開発者はシェル・スクリプトをJavaScriptで書くことができます。

```
var listing = `ls -l`;
```

この例では、`ls -l` コマンドから出力される文字列が `listing` に格納されます。

バッククォート式は、\$EXEC関数の簡略表記です。つまり、``ls -l'`は`$EXEC('ls -s')`を簡潔に表したものです。

JDK 9の\$EXECは、以前のバージョンから大きく改善されています。最初の引数を文字列配列で渡すことができるようになったため、空白などの特殊文字を含む引数を渡せるようになりました。

```
$EXEC(["/bin/echo", "my.java", "your java"]);
```

\$EXEC関数には、throwOnErrorプロパティも追加されています。これをtrueに設定すると、コマンドが失敗した際にRangeErrorが発生します。今までと同じように、\$EXITグローバルを使ってコマンドのステータスを確認することも可能です。

```
$EXEC.throwOnError = true
`javac my.java`
<shell>:1 RangeError:
  $EXEC returned non-zero exit code:2
```

[編集注:最後の2行は1行で表示されます。]管理性も向上しており、コマンドの引数として標準のリダイレクト・ディレクティブを指定し、コマンドのI/O操作を行うこともできます。

```
$EXEC("echo 'my argument has spaces' > tmp.txt");
```

\$EXECの最後の3つの引数に入力ストリームと出力ストリームを渡し、stdin、stdout、stderrをオーバーライドすることもできます。

また、セミコロンや改行を使って、複数のコマンドを実行できます。

```
$EXEC(<<EOD);
echo this ; echo that
echo whatever you want
EOD
```

縦棒記号を使って、コマンドの結果を次のコマンドにパイプすることができます。

```
$EXEC("echo 'my argument has spaces' | cat");
```

さらに、組込みコマンドである疑似cd、setenv、unsetenvを使って、環境変数の値を即座に変更することもできます。

```
$EXEC("setenv PATH ~/bin:${ENV.PATH}; mycmd");
$EXEC("cd ~/bin; ls -l");
```

REPLとしてのjjs

JDK 8版のNashornでは、新しいコマンドライン・ツールJava JavaScript (jjs) が追加されました。jjsにより、開発者はJavaコードを書かずにNashorn機能のテストやJavaScriptアプリケーションの起動を簡単に行えるようになります。

JDK 9では、`java.io.jline`の入出力APIが新しいコンソール入力処理ライブラリ `jline2`に置き換えられています。この変更によって、開発者は、シェルの使用時に、想定されたすべての標準制御を利用できるようになります。次に例を示します。

- 左右の矢印で入力を移動、オプション・キーを押しながら矢印を押すとシンボル単位でジャンプ
- 前方削除および後方削除機能
- [Ctrl]キーを押しながら[k]キーを押すと行の以降の部分を削除、[Ctrl]キーを押しながら[y]キーを押すとその行を復元
- 上下矢印で入力履歴をスクロール表示
- タブ補完をグローバル、プロパティ、Java型にも拡張
- VT100エスケープ・シーケンスによる画面フォーマット

こういった細かい変更すべてによって、作業が少し楽にできるようになっ



GASTÓN HILLAR

新しいHTTP/2クライアントを使う

JDK 9のインキュベータ・テクノロジーでHTTP通信が大幅に簡素化

Java 9では、Java Enhancement Proposal (JEP) 110で定義されている新しいHTTPクライアントAPIが導入され、このAPIによってHTTP/2 と WebSocketが実装されます。この新しいHTTPクライアントは、従来の `HttpURLConnection` APIを置き換えることを目的としたインキュベータ・モジュールとしてJDK 9に含まれます。本記事では、この新しいインキュベータHTTPクライアントが提供する非同期APIを使う方法を紹介します。特に、JDK 9に含まれる新しいread-eval-printループ (REPL) であるJShellで新しいHTTPクライアントAPIを使う方法について説明します。

まずは、基本となる同期版APIの使用方を説明するところから、この新しいクライアントの紹介を始めます。次に、非同期版APIを使って基本的なGETリクエストを実行してみた後、HTTP/2 over TLSを実行する方法を紹介します。この過程により、新しいHTTPクライアントを使用して、ノンブロッキング動作を提供する言語機能がもたらすメリットを活用する方法を確認できます。

最新のHTTPクライアント

JavaでHTTPクライアントが必要になった場合は、サード・パーティ製クライアントを使う方法が一般的でした。しかし、この新しいインキュベータHTTPクライアントでは、HTTP/1.1とHTTP/2を使うことができます。HTTP/2 over TLS (h2) や、クリアテキストを使用したHTTP/2 over TCP (h2c) を扱うことも可能です。

新しいクライアントでは、HTTP/1.1からHTTP/2 over TCPへのアップグレードをリクエストするために、h2cトークンを含むアップグレード・ヘッダー・フィールドを追加できます。サーバーがh2cをサポートしない場合、アップグレードは行われず、すべてHTTP/1.1で動作します。h2cは「HTTP/2 cleartext」の略です。そのため、h2cは暗号化されないという重要なポイントは簡単に覚えることができます。

現在のところ、HTTP/2をサポートするほとんどの構成では、h2のみ

がサポートされています。したがって、クライアントが最新バージョンのTLSで動作するように設定する方法を理解しておくことが重要です。このシナリオは重要であるため、本記事にはその完全な例を含めています。

新しいクライアントとTLSスタックのアップグレードを組み合わせると、アプリケーション層プロトコル・ネゴシエーション (ALPN) をサポートできるため、クライアントはこのTLS拡張機能を使って少ないラウンドトリップでHTTP/2ネゴシエーションを行うことができますようになります。HTTPクライアントで使用する設定に応じて、先ほど説明したようにHTTP/1.1からh2cへのアップグレードのネゴシエーションを行うことも、ゼロからHTTP/2 (h2) を選択することもできます。また、このクライアントは、RFC 6455に準拠したWebSocketもサポートしていますが、本記事では、HTTP機能の例について重点的に説明します。

新しいIncubator Moduleは、JDK 9のjdk.incubator.httpclientに含まれています。このインキュベータ・モジュールは、今後のJDKのバージョンで別のモジュールに移される可能性がある点を考慮しておくことが非常に重要です。以前のプレリリース版のJDKでは、このモジュールは別の名前になっていました。以降のコード・サンプルが期待どおりに動作するためには、確実に最新バージョンのJDK 9を実行する必要があります。

本記事では、このHTTPクライアントの使い方を簡単に説明できるように、JShellを使用します。この方法では、ビルド・システムやIDEでJDK 9を動作させるための個別設定を行う必要はありません。本記事の執筆時点では、多くのIDEやビルド・システムはまだ完全にはJDK 9に対応していません。そのため、予期しない問題が発生する可能性があります。しかし、IDEやビルド・システムがJDK 9に完全に対応すれば、JavaアプリケーションをビルドするIDEに関係なく、本記事のコード・サンプルを使用できるようになるはずです。

次のコマンドは、`--add-modules`オプションの値に`jdk.incubator.httpclient`モジュールを指定してJShellを起動するものです。こうすること

によって、JShellはjdk.incubator.httpclientを解決し、JShellセッションからこのモジュールを使用できるようになります。複数のバージョンのJDKをインストールしており、かつJDK 9がパスに含まれていない場合は、JDK 9のbinフォルダからコマンドを実行する必要があります。macOSやLinuxでは、jshellではなく./jshellと記述する必要があるかもしれません。

```
jshell --add-modules=jdk.incubator.httpclient
```

JShellを使う場合、文末のセミコロン (;) は不要です。しかし、Javaコードとの互換性を持たせて実際のアプリケーションの構築に使えるように、筆者はセミコロンを使うようにしています。それでは、JShellに次のimport文を入力してください。

```
import jdk.incubator.http.*;
```

JShell は、デフォルトでたくさんの `import` 文を利用することが出来ます。しかし、JShell を使わない場合は、サンプル・コードを動作させるために以下の `import` 文を追加する必要があります。

```
import java.lang.*;
import java.net.URI;
import java.net.URISyntaxException;
```

新しいモジュールでは、リクエストとレスポンスが分離されています。以下に示すのは、サンプル・コードでHTTP操作を行うために使用する主なクラスです。

- **HttpClient** : 特定の設定が行われている不変HTTPクライアントを示します。リクエストの送信とレスポンスの受信が可能です。
- **HttpRequest** : HTTPリクエストを示します。
- **HttpResponse** : HTTPレスポンスを示します。

APIは、インスタンスの作成や、さまざまな部品の設定を行うビルダーを提供します。ビルダーは、`new`という接頭辞で始まる静的メソッドになっています。ただし、`HTTPHeader`クラスで表されるヘッダーのビルダーは提供されていません。また、残念なことに、URIは依然として`java.net.URI`インスタンスとして指定されています。そのため、問合せパラメータが必要な場合は、フォーマットするか、文字列を連結しなければなりません。

下記の行では、`client`という名前で`HttpClient`インスタンスを作成しています。最初に`HttpClient.newBuilder`メソッドを呼び出して新しい`HttpClient`ビルダーを作成し、そこにいくつかのメソッド呼出しをチェーンさせています。具体的には、引数に`HttpClient.Redirect.ALWAYS`を指定して`followRedirects`メソッドを呼び出し、クライアントが常にリダイレクトに従うように指定しています。今回は、`http://www.oracle.com`というURIに対してHTTP GETリクエストを行おうとしているため、リダイレクトに従うものとしています。クライアントに対して希望するHTTPのバージョンを指定していないことから、クライアントはHTTP/1.1クライアントとして作成されます。このクライアントは、リクエスト・ヘッダーでHTTP/2へのアップデートをリクエストしません。最後にチェーンされている`build`メソッドの呼出しによって、ビルダーのコードが終了しています。

```
HttpClient client = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build();

System.out.println(client.version());

URI uri = new URI("http://www.oracle.com");

HttpRequest request = HttpRequest
    .newBuilder()
    .uri(uri)
    .GET()
    .build();
```

HttpClientインスタンスを作成した後、`client.version()`メソッドを呼び出してその結果を出力しています。デフォルトの設定を使っているため、JShellにはHTTP_1_1と表示されます。デフォルト値はHttpClient.Version.HTTP_1_1であり、この場合、クライアントはHTTP/1.1でのみ動作します。

コードの次の行では、HTTP GETリクエストを行うURIを使って新しいURIインスタンスを作成し、`uri`に保存しています。次に、新しい[HttpRequest](#)ビルダーを作成する[HttpRequest.newBuilder](#)メソッドを呼び出した後にいくつかのメソッド呼出しをチェーンし、`request`という名前の[HttpRequest](#)インスタンスを作成しています。

引数にuriを指定してuriメソッドを呼び出している部分では、リ

クエストを行うURIを指定しています。次に、GETメソッドをチェーンします。このメソッドは、指定されたURIにHTTP GETリクエストを行うものです。最後にチェーンされているbuildメソッドの呼出しによって、ビルダーのコードが終了しています。これは非常に読みやすいコードです。HttpRequestのインスタンスを作成するコードでは、HTTPのGET動詞がメソッドとしてとてもわかりやすい形で示されています。

次に、以下に示すコードで、client.sendメソッドを呼び出します。このメソッドは同期的に実行され、レスポンスが取得できるまで実行がブロックされます。引数として、先ほど作成したHttpRequestインスタンスであるrequestと、HttpResponseBodyHandler.asString()が渡されています。

```
HttpResponse<String> response =
    client.send(request,
        HttpResponse.BodyHandler.asString());
System.out.println(
    String.format("Status code: %d",
        response.statusCode()));
System.out.println(
    String.format("Body length: %d",
        response.body().length()));
```

2番目の引数には、ここで使用するレスポンス本文のハンドラを指定します。本文のハンドラは、レスポンスのステータス・コードとレスポンス・ヘッダーを受け取り、HttpResponseBodyProcessorのインスタンスを返します。この例では、HttpResponseBodyHandler.asString()が本文のプロセッサを返します。このプロセッサは、デフォルトの文字セットを使ってレスポンスの本文を文字列として格納します。HTTP GETリクエストの処理が成功すると、sendメソッドはHttpResponse<String>を返し、それがresponseに保存されます。

最後の部分では、レスポンスのHTTPステータス・コードを返すstatusCode()メソッドと、文字列として取得した本文の長さを返すbody().length()メソッドの結果を出力しています。

(JShellでは、このとおりに入力しても動作しません。各文は1行で入

力する必要があります。残念ながら、本記事執筆時点では、JShellでは複数行のコードを入力できず、予期しないエラーがスローされます。しかし、ここに掲載するすべてのコードを1行にすれば、非常に読みにくくなってしまいます。)

先ほどのコードでは、HTTP/1.1を使ってHTTP GETリクエストで取得したWebページの本文を文字列として取得しています。これが、新しいHTTPクライアントのおそらくもっともシンプルな使い方です。再度コードを読んでも、HttpClientとHttpRequestの両方が同じような方法で構築されていることに気づくでしょう。また、BodyHandlerを指定できるため、HttpResponseを文字列として取得することがわかりやすくなっています。つまり、ビルダーの使用に関する部分について考え、目的に応じてそれを設定しさえすればよいということです。APIは簡潔で、チェーンさせることができます。

最近のJavaコードになじんでいる方なら、このAPIが使いやすいことがわかりでしょう。JShellでわずかなコードを書くだけで、非常に基礎的な設定とその同期APIによって、新しいHTTPクライアントを使うことができました。

非同期で実行する

このクライアントでHTTP/2とTLSを使う方法を紹介する前に、非同期APIについて触れておきます。JShellは、デフォルトでたくさんのimport文を実行してくれます。しかし、JShellを使わない場合は、後述のサンプル・コードを動作させるために以下のimport文を追加する必要があります。

```
import java.util.concurrent.CompletableFuture;
```

同期的に実行した先ほどの例では、client.sendメソッドを呼び出しました。今回は、HttpRequestを作成するコードは同様ですが、client.sendAsyncメソッドを呼び出します。このメソッドは、

JShellでわずかなコードを書くだけで、非常に基礎的な設定とその同期APIによって、新しいHTTPクライアントを使うことができました。



次のコードでは、`response.whenComplete`を呼び出し、例外

```
HttpClient client = HttpClient.newBuilder()
```

```

        .build();
        URI uri = URI.create("http://localhost:8080/");
    }
}

```

JShellでこのようにわずかなコードを書くだけで、非常に基礎的な設定とそ

この HTTP クライアントは、標準の Java TLS メカニズムを使って HTTP/2 over TLS

また、簡単に証明書をロードするために、Bouncy Castleライブラリ

特定のビルド・システムは利用せず、引き続き、JShellでの例を紹介しま

Bouncy Castleのサイトから次のJAR

- bcmail-jdk15on-157.jar
- bcpkix-jdk15on-157.jar
- bcprov-jdk15on-157.jar

modulesオプションの値にjdk.incubator.httpclientモジュールを、--class-pathオプションの値に先ほど挙げたJARファイルを指定してJShellを起動しています。こうすることによって、JShellはjdk.incubator.httpclientを解決できます。また、指定したクラス・ファイルもロードされるため、Bouncy Castleライブラリを使用できるようになります。なお、JShellはJARファイルを保存したパスから起動するようにしてください。パスにjshellが含まれていない場合は、フルパスを指定する必要があります。

次のコードは、すべての import 文を記載したものです。JShell を使わない場合にもコードを実行しやすいように、JShell がデフォルトで実行する多くの import 文も含めています。JShell が自動的に読み込む import 文と重複しても、エラーは発生しないようになっています。

JULY/AUGUST 2017

TLSContextHelper クラスには、TLSv1.2 とともに使用できる多くの静的メソッドが宣言されています。このクラスのコードは 150 行ほどあるため、Java Magazine の[ダウンロード・サイト](#)からダウンロードしてください。このコードに含まれる重要なメソッドについて説明します。

換する `java.key.KeyFactory` インスタンスを返します。

- `createX509CertificateFromFile` : 証明書のファイル名を受け取ってファイルの内容を読み込み、そのファイルから `X509Certificate` インスタンスを生成して返します。
- `createPrivateKeyFromPemFile` : PEM 形式の鍵ファイル名を受け取ってファイルの内容を読み込み、`java.Security.PrivateKey` インスタンスを生成して返します。
- `createKeyManagerFactory` : 証明書のファイル名、クライアントの鍵ファイル名、クライアントの鍵パスワードを受け取り、`createX509CertificateFromFile` メソッドと `createPrivateKeyFromPemFile` メソッドを呼び出します。続いて、これらのメソッドから返されたインスタンスを使用し、`java.net.ssl.KeyManagerFactory` クラスのインスタンスを作成して返します。
- `createTrustManagerFactory` : 認証局の証明書ファイル名を受け取り、`createX509CertificateFromFile` メソッドを呼び出します。続いて、`X509Certificate` のインスタンスを使用し、`java.net.ssl.TrustManagerFactory` クラスのインスタンスを作成して返します。
- `createAndInitTLS12Context` : 認証局の証明書ファイル名、クライアントの証明書ファイル名、クライアントの鍵ファイル名を受け取り、目的の TLS バージョン (TLSv1.2) の `SSLContext` インスタンスを作成して初期化します。続いて、`BouncyCastleProvider` を使用し、先ほど説明した `createKeyManagerFactory` メソッドと `createTrustManagerFactory` メソッドを呼び出します。

次のコードの `certificatesPath` 変数では、この例を実行するために必要な証明書のベース・パスを宣言しています。この文字列の内容は、認証局の証明書ファイル、クライアントの証明書ファイル、およびクライアントの鍵ファイルが格納されているパスに置き換えてください。処理する HTTP リクエストに対応したファイルを使用する必要があります。ここでは例として、Windows のパス `D:\JavaMagazine\http2` を使用しています。なお、認証局の証明書、クライアントの証明書、およびクライアントの鍵の

ファイル名を指定する変数を定義するコードは、Linux、Oracle Solaris、macOS など、コードを実行しているすべてのプラットフォームに対応します。このコードでは、[String.join](#) を呼び出し、[java.io.File.separator](#) を使用して先ほど説明したパスとファイル名を組み合わせることによって、フルパスのファイル名を作成しています。ca.crt、server.crt、server.key は、忘れずに適切なファイル名に置き換えてください。最後の行では、[HttpClient](#) で h2 を使用するために用いる [SSLContext](#) を作成して初期化しています。

```
String certificatesPath = "D:\\JavaMagazine\\http2";
String caCertificateFileName =
    String.join(java.io.File.separator,
        certificatesPath,
        "ca.crt");
String clientCertificateFileName =
    String.join(java.io.File.separator,
        certificatesPath,
        "server.crt");
String clientKeyFileName =
    String.join(java.io.File.separator,
        certificatesPath,
        "server.key");
SSLContext sslContext =
    SecurityHelper.createAndInitSSLContext(
        caCertificateFileName,
        clientCertificateFileName,
        clientKeyFileName);
```

下記の行では、`client` という名前で `HttpClient` インスタンスを作成しています。最初に `HttpClient.newBuilder` メソッドを呼び出して新しい `HttpClient` ビルダーを作成し、そこにいくつかのメソッド呼出しをチェーンさせています。前述の例に似ているコードも含まれていますが、今回は `sslContext` メソッド（先ほど作成した、`sslContext` という名前の `SSLContext` インスタンスを引数として渡しています）を呼び出して、

HTTP/2 over TLSv1.2 を使えるようにしている点に注意してください。そして、`HttpClient.Version.HTTP_2` を引数として `version` メソッドを呼び出し、HTTP/2 の使用を強制しています。ここでは `SSLContext` もチェーンさせているため、HTTP/2 over TLSv1.2 が使用されることになります。

```
HttpClient client = HttpClient.newBuilder()
    .sslContext(sslContext)
    .version(HttpClient.Version.HTTP_2)
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build();
System.out.println(client.version());
URI uri =
    new URI("https://your-rest-api-url-for-get-method");
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(uri)
    .GET()
    .build();
CompletableFuture<HttpResponse<String>> response =
    client.sendAsync(request,
        HttpResponse.BodyHandler.asString());
response.whenComplete((HttpResponse<String> response,
    Throwable exception) -> {
    if (exception == null) {
        System.out.println(
            String.format("Status code: %d",
                response.statusCode()));
        System.out.println(String.format(
            "Body length: %d",
            response.body().length()));
    } else {
        System.out.println(String.format(
            "Something went wrong. %s",
            exception.getMessage()));
    }
}
```

 $\});$

`HttpClient` インスタンスを作成した次の行では、`client.version()` メソッドを呼び出した結果を出力しています。HTTP/2 の使用を強制しているため、JShell には HTTP_2 と表示されます。なお、<https://your-rest-api-url-for-get-method> は、GET メソッドを使用可能であり、HTTP/2 over TLSv1.2 でレスポンスを返す REST API の URI に置き換える必要があります。また、証明書を使っているため、誤った証明書を使うと TLS ハンドシェイク（古い Java の例外名では、SSL ハンドシェイクと報告されます）が失敗する点に注意してください。わずか数行を追加するだけで、先ほどの例と同じように非同期で、HTTP GET リクエストを実行できるようになりました。ただし今回は、REST API が HTTP/2 over TLSv1.2 に対応している場合、`HttpClient` は HTTP/1.1 ではなく HTTP/2 over TLSv1.2 で動作します。

```
HttpClient client = HttpClient.newBuilder()
    .sslContext(sslContext)
    .version(HttpClient.Version.HTTP_2)
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build();

System.out.println(client.version());

URI uri = new URI("https://your-rest-api-url-for-get-method");
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(uri)
    .GET()
    .build();

CompletableFuture<HttpResponse<String>> response =
    client.sendAsync(request,
        HttpResponse.BodyHandler.asString());

response.whenComplete((HttpResponse<String> response,
    Throwable exception) -> {
    if (exception == null) {
        System.out.println(
            String.format("Status code: %d",
```

```
        response.statusCode());  
        System.out.println(String.format(  
            "Body length: %d",  
            response.body().length()));  
    } else {  
        System.out.println(String.format(  
            "Something went wrong. %s",  
            exception.getMessage()));  
    }  
});
```

まとめ

本記事ではシンプルな例を用いて、JShell から新しいインキュベータ・モジュールの HTTP/2 クライアントで HTTP/2 over TLS を使う方法を説明しました。このモジュールでは、JShell から利用でき、読みやすいコードを書ける多くの追加機能が提供されています。ソフトウェア開発タスクでは、この新しい HTTP/2 クライアントが非常に便利であることに気づくはずです。JShell などのインタラクティブ REPL を使う必要がある場合は、特にそうであると言えます。ただし、現在の欠点は、このモジュールがインキュベータとして含まれており、今後変更される可能性があることです。とはいえ、API に何らかの変更が生じて構わないのであれば、この新機能は十分試してみる価値があると言えるでしょう。 </article>

Gastón Hillar (@gastonhillar) : Java が初めてリリースされた頃からの Java 関連ソフトウェア・アーキテクトで、ソフトウェアの設計や開発に 20 年ほど携わっており、ソフトウェア開発、ハードウェア、電子工学、Internet of Things に関する多くの著書がある。Intel Black Belt Software Developer Award を 8 回受賞している。

learn more

[HTTP/2のホームページ](#)

[JEP 110: HTTP/2クライアント \(インキュベータ\)](#)
[JEP 222: jshell: Java ShellRFC 6455 \(WebSocket
 プロトコル\)](#)

Bouncy Castle Crypto API

クイズに挑戦

中級者、上級者向けの問題

以下の設問は、2種類の認定資格試験の難易度を想定しています。「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、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:

```
interface ParentIF {}
interface ChildIF extends ParentIF {}
interface OtherIF {}
class ParentCL {}
class ChildCL extends ParentCL {}
class OtherCL {}
```

さらに、次のコードについて:

```
ChildIF ci = null;
ParentIF pi = null;
OtherIF oi = null;
ChildCL cC = null;
ParentCL pC = null;
OtherCL oC = null;
```

```
cl = (ChildIF)ol; // line n1
```

```
cC = (ChildCL)pC; // line n2
cC = (ChildCL)oC; // line n3
cl = (ChildIF)oC; // line n4
```

正しいものはどれですか。1つ選んでください。

- a. line n1とline n3は、いずれもコンパイルに失敗する。
- b. line n1とline n3は、いずれもキャストが不要である。
- c. line n3は、コンパイルに失敗する。
- d. line n2とline n4は、いずれもキャストが不要である。
- e. line n4は、コンパイルに失敗する。

設問2 (中級者向け) . 次のコードについて:

```
class P {
    private int value;
    // line n1
    public P(int v) {
        value = v;
    }
}
```

```
class S extends P {  
    private int value;  
    // line n2  
    public S(int v, int u) {  
        // line n3  
        value = u;  
    }  
}
```


正しい説明文はどれですか。1つ選んでください。

- a. コードはエラーなしにコンパイルできる。
- b. line n2に`public S(int v) {}`を追加すると、コードはエラーなしにコンパイルできる。
- c. line n3に`this(v);`を追加すると、コードはエラーなしにコンパイルできる。
- d. line n3に`super(v);`を追加すると、コードはエラーなしにコンパイルできる。
- e. line n1に`private P(){}`を追加すると、コードはエラーなしにコンパイルできる。

設問3 (上級者向け) . 新しいバージョンのファイルが書き込まれた場合や、ファイルの追加や削除が発生した場合のような、ディレクトリ内の変更に応答する必要があるプログラムを書いているとします。

この要件に対応するためには、標準Java SE APIのどの機能を使いますか。1つ選んでください。

- a. `java.nio.file.Path`
- b. `java.nio.file.Files`
- c. `java.nio.file.FileVisitor`
- d. `java.nio.channels.AsynchronousChannel`
- e. `java.io.File`

設問4 (上級者向け) . 次のコードについて：

```
public static void delay() {
    try { Thread.sleep((int) (Math.random() * 10)); }
    catch (InterruptedException ie) {}
}

public static void main(String[] args) {
    int[] x = {0};
    boolean[] hold = {true};
    new Thread() -> {
        delay();
        x[0] = 99;
        hold[0] = false;
    }.start();
    new Thread() -> {
        delay();
        while (hold[0])
            ;
        System.out.println("value is " + x[0]);
    }.start();
}
```

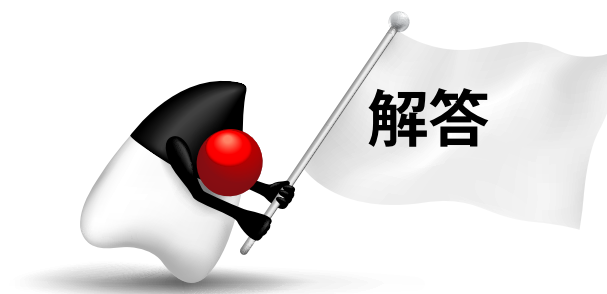
```
}).start();
new Thread() -> {
    delay();
    while (hold[0])
        ;
    System.out.println("value is " + x[0]);
}.start();
}
```

さらに、次の動作から選択する場合：

- 1. プログラムはvalue is 0を出力する。
- 2. プログラムはvalue is 99を出力する。
- 3. JVMが終了する (プログラムが停止する)。
- 4. JVMが終了しない (プログラムが停止しない)。

仕様上、起こりうるすべての結果を述べたものはどれですか。1つ選んでください。

- a. 2と3が起きる。
- b. 1と2は起こらず、3が起きる。
- c. 1か2のいずれかと、3が起きる。
- d. 2と3の両方が起きる。または、1と2は起こらず、4が起きる。
- e. 3と、1か2のいずれかが起きる。または、1と2は起こらず、4が起きる。



設問1: 正解は選択肢Cです。この設問は、代入の互換性について問うものです。

手短かに言えば、コンパイラは代入の際に3つの状況を認識します。まずは、右辺の式が代入される式の型とis-a関係にある状況です。この右辺の式は、代入対象となるクラスの厳密なインスタンスでも、そのサ

line n1は、`OtherIF`への参照を`ChildIF`にキャストしようとしています。この2つのインタフェース型には何の関係もありますが、コンパイラにとって重要なのは、キャストされる参照が`ChildIF`インタフェースの実装である可能性があるかどうかです。一般的に、このような形でインタフェース型にキャストするのは妥当です。たとえば次のようなクラス

こう考えると、`oC`という参照がこういったオブジェクトの1つを指している可能性があることは明かです。この場合、要求された`ChildIF`型へのキャストは成功します。そのため、選択肢Eは誤りです。

参考までに、OtherCLの定義がfinalだった場合を考えてみます。この場合、OtherCLにはサブクラスが存在しないことが保証され、上記で説明したようなシナリオが起こりえなくなるため、line n4のコンパイルは失敗します。

設問2: 正解は選択肢Dです。この設問は、サブクラスの初期化の一部として実行される、親クラスの要素の初期化について問うものです。Java は非常に苦勞して、正しい初期化が行われることを保証しています（正しい初期化は必ずしも強制できるとは限りませんが、できる限りそれを目指すようになっています）。

この設問には、3つのルールが関係しています。まず、サブクラスをインスタンス化するときは、親が正しく初期化されるように、親クラスのコンストラクタを呼んで必要な引数を渡す必要があります。

次に、クラスのソース・コードで明示的にコンストラクタが定義されていないという特別な場合、コンパイラがデフォルトのコンストラクタを作成します。このコンストラクタに引数はなく、親クラス（スーパークラス）の引数なしコンストラクタを呼びます。親クラスがObjectの場合も同様です。

3つ目は、コンストラクタでスーパークラスのコンストラクタに制御を渡す方法が明示的に宣言されていない場合です。この場合、コンパイラはコンストラクタ内に親クラスの引数なしコンストラクタを呼び出すコードを暗黙的に生成します。このルールから、Sのコンストラクタは次のコードと等価ということになります。

```
public S(int v, int u) {
    super();
    value = u;
}
```

今回の例では、設問に記載されているように、子クラスのSから親クラスPのコンストラクタを明示的に呼び出していません。そのため、親クラスの引数なしコンストラクタが呼ばれることになります。しかしPには、1つの引数を受け取る明示的なコンストラクタがあります。まとめると、子クラスのコンストラクタがPの引数なしコンストラクタ（このコンストラクタは存在しません）を暗黙的に呼び出しているため、子クラスのコンストラクタはコンパイルできません。したがって、選択肢Aは誤りです。

この問題を解決する方法は2つあります。親クラスPに適切な引数なしコンストラクタを作成して子クラスからの呼出しが成功するようにするか、子クラスのコンストラクタを変更して親クラスに存在するコンストラクタを明示的に呼び出すようにするかのいずれかです。既存のコンストラクタを呼ぶことも、Pに引数の異なる別のコンストラクタを作ってSから明示的に呼び出すこともできますが、後者を実現する選択肢は存在しないため、この方法は無視して構いません。

選択肢Bは、子クラスに引数が1つのコンストラクタを追加していますが、これは実のところ役に立ちません。新しいコンストラクタにもコンストラクタの明示的な呼出しはないため、親クラスに引数なしコンストラクタが必要になることに変わりありません。つまり、子クラスにこのコンストラクタを追加しても、状況は変わりません。この場合、コンストラクタのシグネチャ、すなわち引数リストが一致しているだけでは意味がない点に注意してください。そのため、選択肢Bは誤りです。

選択肢Cは実際に有効な構文ですが、これも問題を解決することにはなりません。 `this(v);` は、1つの引数を受け取るコンストラクタに処理を委譲しようとするものですが、対象のコンストラクタは、親クラスではなく、同じクラスにある必要があります。そのため、選択肢Cは誤りです。なお、インスタンス・メソッドと違ってコンストラクタは継承されないことに注意してください。そのため、実際には `this(v);` の呼出し先は存在しないことになります。

選択肢Dは、1つの引数を受け取る、親クラスのコンストラクタを呼び出す正しい構文です。**v**の値を親クラスのコンストラクタに渡しているため、引数が1つであるという要求も満たしています。この値は、**P**のメンバーであるprivate変数**value**に格納されます。そのため、選択肢Dはコンパイルが成功し、正解ということになります。

親に引数なしコンストラクタを作成する選択肢Eは一見正しそうに見えますが、この選択肢Eのコンストラクタはprivateになっています。引数なしコンストラクタがprivateであるため、子クラスからはアクセスできません。そのため、SのコンストラクタがPの引数なしコンストラクタを呼び出そうとしているという本質的な問題は解決されなままです。Sは対象のコンストラクタにアクセスする必要がありますが、privateなコンストラクタではアクセスに失敗します。そのため、選択肢Eは誤りです。privateなコンストラクタというのは奇妙に思えるかもしれませんが、オブジェクトの作成方法を制御する場合にはとて

も便利です。たとえば、シングルトンや静的ファクトリ、ビルダー・パターンなどの実装に役立ちます。

また、PとSの両方に存在するprivateフィールドvalueについても考えてみるとおもしろいでしょう。これは本当に不適切なことなのでしょうか。また、この2つは、実際には同じフィールドになるのでしょうか。実は、いずれもそうではありません。フィールドがprivateであるため、名前が衝突することはありません。各フィールドは、含まれているクラスの中からしか見えません。この2つのフィールドは、実質的にクラスの外側からは見えないため、何に影響を与えることもなく、同じ名前の別の変数と衝突することもないのです。また、この2つは完全に別のフィールドをそれぞれ定義しており、同じフィールドになることはありません。両方のクラスを同時に見れば奇妙に感じられるかもしれませんが、予測に反し、コードに問題はありません。両方のソース・コードを同時に見なければ、そのことにも気づかないでしょう。ただし、これらのフィールドがクラスの外側からアクセスできる場合（デフォルトのアクセスを許可する場合など）、コードはとても恐ろしいものになります。通常、このような状況はシャドーイングや変数の隠蔽と呼ばれます。注意して構文を利用すれば回避できることですが、この状況を許可するのは、みずからメンテナンス上の問題を生み出すようなもので、さまざまな誤解が生じることにつながります。

設問3: 正解は選択肢Aです。これは、悩ましい「APIを暗記する」系の設問ですが、この設問ではメソッド名を覚える必要はなく、APIが提供可能な機能さえ覚えていれば対応できます。実のところ、これは時間の無駄になるものではありません。こういったことを覚えていないと、すでに提供されている機能を重複して作ってしまうことになるからです。つまり、この設問は不適切なものではありません。

それでは、ここで示されている各機能は何をするものなのでしょうか。これらの機能は、ディスク・ストレージに関連するものです。最初のjava.nio.file.Pathインタフェースは、Javaプログラムで「パス

プログラマーは、コンピュータがどのように動作するかについてのメンタル・モデルが必要であり、それが日々の作業の根拠となります。

とファイル名」を表現する新しい方法です。かつて、このような作業は `java.io.File` クラスを使って行われていましたが、このクラスには、汎用的ではないファイル・システムの機能（パーミッションなど）を表現する力に欠けていました。注目すべき点は、`Path` はインタフェースであり、異なるオペレーティング・システムや、場合によってはファイル・システムの種類ごとに完全に独立した実装が可能になっていることです。一方の `File` はクラスであるため、このような柔軟性を実現することは困難でした。`Path` インタフェースでは、パスのセグメントの操作や、相対パスと絶対パスの取り扱いのための便利でわかりやすい機能が数多く提供されています。さらに、2種類の `register` メソッドも定義されており、ファイル・システムが変更されたタイミングをコードで簡単に判定できます。以上の説明から、選択肢Aが正解であることは明らかです。また、`File` クラスは機能が限られた古いものだと考えられていることから、選択肢Eが誤りであることも推測できます。

Filesクラスは静的メソッドのコンテナで、ファイルのコピーと移動、ファイルとディレクトリの作成と削除、パーミッションの読取りと操作、ディレクトリ・ツリーのトラバースといった便利な操作を実行できます。このクラスでは、ファイル内のデータへのアクセスを簡素化できるユーティリティ・メソッドも提供されています。たとえば、linesメソッドはテキスト・ファイルを直接Stream<String>として読み取ります。これは非常に便利なクラスであり、まだじっくり見たことがないという方はぜひ確認してみてください。ただし、ディレクトリ構造の変更を監視するという問題を解決するために直接使用することはできないため、選択肢Bは誤りです。

FileVisitorインタフェースは、Files.walkFileTreeメソッドなどの機能と合わせて使用します。FileVisitorは、ディレクトリ・ツリー内の一部または全部のファイルやディレクトリを操作する際に使います。つまり、このインタフェースはディレクトリの内容を調べることができますが、それは一度限りであるため、変更の監視に直接利用できるものではありません。よって、選択肢Cは誤りです。

AsynchronousChannelインタフェースはベース・インタフェースであり、ここからいくつかの興味深いクラスが間接的に派生しています。これらのクラスは、「コールバック」型の非同期I/O操作を実現するもので、高度に並列化されたシステムにおいて、利用するスレッドの数を最小限にとどめたい場合に役立つ可能性があります。ただし、この機能は、ディレクトリの変更を監視するという問題に直接の関係はありません。そのため、選択肢Dは誤りです。

最後に、Pathインタフェースではファイル・システム上の変更の監視を簡単に実現できるregisterメソッドが定義されていますが、このメソッドは単独で使えるものではないことをお伝えしておきます。このメソッドには、WatchServiceのインスタンスを渡すことも必要です。WatchServiceのインスタンスは、静的ファクトリ・メソッドFileSystem.newWatchService()から取得できます。

設問4: 正解は選択肢Eです。この設問は、Java仕様の中でも誤解されることがとても多いメモリ・モデルについて問うものです。この仕様は、スレッド化されたコードの動作について規定しています。提示されたプログラムの動作に影響するのは、仕様によって規定されている2つの点、すなわち書き込まれたデータの可視性と、書き込み操作が認識される順番です。この2つのうち1つは、コードを実行した際に表面化する可能性がほとんどないことに注意してください。しかし、だからといって答えが誤りであるということではなく、単に皆さんのシステムがたまたまそのように動作しないというだけのことです。

プログラマーには、コンピュータがどのように動作するかについてのメンタル・モデルが必要であり、それが日々の作業の根拠となります。一般的に、日々の作業で役立っているモデルは大幅に簡素化するものであるため、並列処理が関係してくる場合には、それが問題になる可能性があります。ハードウェア・エンジニアは、プロセッサを早く動作させるために、いくつかの奥の手を持っています。Javaがよいパフォーマンスを保証するためには、実行されるあらゆるホストでそういった奥の手をできるだけ多く使えるようにしなければなりません。そのため、Javaの仕様は実装についてではなく、メモリに関するどのような効果を信頼できるのかについて記述される形になっています。この仕様では、あるスレッドがデータを書き込んだ場合、別のスレッドからも見える必要があるデータはどれか、またどのタイミングで見えるようになる必要がある

るかを推論できるようにするために、happens-before関係と呼ばれる考え方を使っています。

奇妙なこともかもしれませんが実のところ、happens-before関係は、何かが別のことの前に起きることを言っているものではありません。少なくとも皆さんが想定するような形ではなく、実際には、実行順序に関係するわけでも、それを定めているわけでもありません。関係するのは、ある効果の可視性だけです。これは驚くべきことかもしれませんが、その根本的な理由は、何十年もの間、コンパイラの最適化によって命令の並べ替えが行われてきた点にあります。ある状況下では、命令の並べ替えは安全に行うことができます。簡単な例を見てみます。

```
double x = heavyComputation();
double y = otherComputation();
if (x > 3) doSomethingWith(x);
```

この場合、最初の2行の順番を変えても、結果に違いはない点に注意してください。しかし、この並べ替えによって、コンパイラがさらに効率のよいコードを生成できる可能性があります。たとえば、`x`を保存しておいて3番目の命令で再取得するのではなく、まず`otherComputation`を実行して結果を保存しておき、次に`heavyComputation`を実行して直後にそのまま`x`の値を使用することもできるでしょう。これは小さな改善のように思えるかもしれませんが、小さな改善の積み重ねが大きな改善につながる可能性があります。

それでは、happens-before関係からわかることは何でしょうか。AがBの前に起こり、かつAが書き込んだものをBが読み取る場合、このhappens-before関係からわかるのは、Aが書き込んだ値をBが読み取ることです。重要なのは、happens-before関係と「観測」の両方が存在しなければ、何も保証されないという点です。さらに、ある状況（たとえば、2つのスレッドT1とT2）でhappens-before関係が存在するというだけでは、そのような関係が存在しない場所（たとえば、3つ目のスレッドT3）では、効果について何の可視性も保証されません。当然ながら、happens-before関係はシングルスレッドで実行されるコードの行の順番によって生まれるものです。また、happens-before関係には推移性があります。そのため、AがBの前に起こり、BがCの前に起こる場合、AはCの前に起こります。この推移的效果はシングルスレッドに限られることはありませんが、この行単位での単純な仮定は、シングルスレ

ッドの場合にのみ成り立ちます。スレッド間で必要な関係を生み出すには、特殊な手順を経る必要があります。

設問のコード例では、`main`メソッド内で2つのスレッドを作成し、開始しています。ただし、2つのスレッド間に`happens-before`関係は定義されていません。つまり、2番目のスレッド（読取りを行う側）は、配列`x`が0から99に変更されたことが見える可能性もあれば、見えない可能性もあります。同じように、配列`hold`が`true`から`false`に変更されたことが見える可能性も、見えない可能性もあります。両方が変更されたことが見えるかもしれませんが、いずれも見えないかもしれません。そして、いずれかのみの変更が見え、もう片方の変更は見えない可能性もあります（多くの場合、これは開発者にとって驚きです）。そのため、2番目のスレッドは、99は見えて`false`への変更は見えない可能性があります。また、`true`への変更は見えて99への変更は見えない可能性もあります。後者の場合、プログラムは`value is 0`と出力して終了しますが、通常、これは予期しない動作です（公正を期するために言うなら、この結果は通常の計算ハードウェアではかなり起こりにくいものですが、仕様上は許可されています。つまり、これはコードのバグであり、結果はOKではないということです）。

以上のことから、起こりうる結果について考えてみます。ここで重要になるのは、2番目のスレッドに何が見えるかです。boolean値が変更されたことが見える場合、プログラムは何かを出力してから終了します。しかし、booleanの変更が見えない場合、プログラムは何も出力せず、停止することもあります。ここから、選択肢A、B、Cを除外できます。これらの選択肢は、プログラムが停止しない可能性を認めていないからです。

残された2つの可能性を考えるためには、2番目のスレッドでbooleanの値の変更が見える場合に、どの値が出力されるかを考える必要があります。ここまでの説明で、value is 0とvalue is 99のいずれかが正しく出力される場合もあることはわかりでしょう。選択肢Dは、プログラムが停止する場合はvalue is 99が出力されるものの、プログラムが停止しない可能性もあると言っています。そのため、この選択肢は誤りであることがわかります。選択肢Eは、値を出力して停止するか、値を何も出力せず停止もしないかのいずれかの可能性があると言っています。これが正解です。

「仕様上許可されている」という考え方には、少しばかり納得がいかないと思う方もいらっしゃるかもしれません。そこで、実際のハード

ウェアで何が起る可能性があるのかについて、1つの例を見てみます。ただし最初に、ハードウェア実装の点からJavaのメモリの動作について考えることは危険であり、誤解につながる場合も多くあると認識することが重要です。唯一の信頼できるアプローチは、メモリ・モデルを考慮し、それを使って考えることです。いずれにせよ、「でもそんなことは絶対に起こらない」というしつこい感情はひとまず押しとどめます。実際のハードウェアでも前述のようなことが起きる可能性が1つあります。まず、`main`メソッドで作成された2つのスレッドが別々のCPU上で実行され、2つのCPUが独立したキャッシュを持っていると考えてください（これは実際にありうることです）。次に、書込みスレッドが実行され、メイン・メモリではなく、キャッシュに99と`false`が書き込まれたとします。スレッド間では何のhappens-before関係も強制されていないことを考えると、基盤となるハードウェアにもJVMにも、各スレッドのキャッシュの一貫性を保つ義務はないと言えます。これらの値が最終的にメイン・メモリに格納されるのか、別のCPUのキャッシュに移動するかは定義されていません。そのため、boolean値がたまたま99より先にキャッシュからフラッシュされることも考えられます（通常、キャッシュは行単位でフラッシュされますが、たまたまこの2つのデータ項目が別の行に格納される可能性もあります。その場合、書込みとは逆の順番でフラッシュされることもありえます）。このような状況では、あるスレッドが「順番」に実行した書込み操作が、別のスレッドからは異なる順番に見えることもあります。以上の説明で、Javaで信頼性のあるコードやないコードがどんなものであるかを考える際に、メモリ・モデルが唯一の安全な方法である理由がわかったのではないかと思います。

最後に、どうすればこのコードを意図どおりに動作させられるかについて、簡単に説明しておきます。ここでは、あえて意図する動作と言いました。意図する動作とは、ソース・コードから読み取れる動作、すなわち、value is 99を出力して停止するという動作です。これを実現するためには、1つ目のスレッドによる[hold\[0\]](#)への書込みと、2つ目のスレッドによるその値の読取りとの間に、happens-before関係を作成する必要があります。

