

# Java™ magazine

By and for the Java community 

## ライブラリ 適切なライブラリを選ぶ

05

iTEXTによる  
PDFの作成と  
操作

17

j2HTMLによる  
HTML高速生成

28

ASMでバイト  
コードを生  
成、変換する

42

ライブラリを  
JAVAモジュールに移行する



05

## 複雑なPDFを簡単に作成

著者: Bruno Lowagie、Joris Schellekens

人気のJavaライブラリiTextで、  
PDFファイルを作成して操作する方法

## その他の記事

### 55 Stateデザイン・パターン

著者: Ian Darwin

巨大なswitch文や大量のif文を  
使わず、デザイン・パターンで状態  
遷移をエレガントに管理

### 65 Fix This

著者: Simon Roberts、Mikalai Zaikin

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

## 特集記事

17

### J2HTML: HTML5 ジェネレータ・ ライブラリ

著者: Mert Çaliskan

Javaのタイプ保証と  
おなじみのツールを  
活用し、動的に生成  
される小さなWeb  
アプリケーションを  
簡単に作成する

28

### ASMによるバイト コード処理の実際

著者: Ben Evans

ASMライブラリを使  
用して、実行時にバイ  
トコードをスキャン、  
検査、生成、変換す  
る

42

### ライブラリを JAVAモジュールに 移行する

著者: Nicolai Parlog

モジュールへの移行  
に必要なのは、「早と  
ちり」を避けた、慎重  
な計画と丹念な実行

54

### 便利な ライブラリ集

ここ数年で取り上げ  
たさまざまなJava  
ライブラリ

## 常設記事

### 02 編集長より

大規模アプリにスクリプト・エンジ  
ンを埋め込むという確かな主張

### 53 Java注目の提案

JEP 296: 単一リポジトリへの  
JDKソースの統合

### 76 Java注目の提案

JEP 335: Nashorn JavaScript  
エンジンの非推奨化

//from the editor /



# エキスパート・ユーザーの望みをかなえる

アプリにスクリプト・エンジンを埋め込むという主張

**現**在、「ユーザー・エクスペリエンス」と言えば、コンシューマーや非エキスパート・ユーザーの好みに焦点を当てたものがほとんどです。しかし、エキスパート・ユーザーを対象とするベンダーにとって、アプリケーションのユーザビリティとは、エキスパートが必要な作業をソフトウェアで行うことができるかを問うものになります。多くのアプリケーションやパッケージにおいて、その答えはあらゆる種類の制約に阻まれています。そういった制約は、エキスパートのニーズに対するベンダーの認識から生まれます。しかし、過小評価されがちですが、多くの場合にエキスパートが確実に作業を行えるようにする選択肢が1つあります。それが埋込みスクリプト・エンジンです。スクリプトを使うという選択肢をユーザーに提供しているアプリケーションは、エキスパートに優しいアプリケーションです。提供していないものは、エキスパートに優しいとは言えません。

スクリプトは、熟練したユーザーの領域です。プログラミングに関するいくつかの基礎知識が求められ、ツールの内部構造の詳細を学ぶための時間も必要になるからです。アプリケーションでプログラミングを行えることのメリッ

トは、長いこと認識されています。もっとも一般的な例は、間違いなくVisual Basic for Applications (VBA) です。VBAを使うと、Microsoft Excel (およびその他のMicrosoftデスクトップ・アプリ) で洗練されたマクロを書くことができます。他のソフトウェアも、より洗練されたスクリプト言語に依存してきました。たとえば、TclはElectronic Design Automation (EDA) やCADツールで使われる主要なスクリプト・ツールです。UI設計をはじめとする他の分野では、埋込みスクリプト言語Luaが広く使われています。

実際、長年にわたり、埋込みスクリプトという概念はとても一般的でした。そのため、デザイン・パターンをソフトウェアに初めて導入した「Gang of Four」(四人組) 本には、Interpreter/パターンが含まれていました。今日から考えると、これはいい意味で奇妙なことのようには思います。このソリューションの人気の頂点に達したのは、およそ10年前です。ドメイン固有言語 (DSL) という形で長いこと注目を集めました。が、(汎用ソリューションではなく) ドメイン固有性という利点によって難しい要件が課されることに開発者が気づくと、下火になっていきました。

写真: BOB ADLER/THE VERBATIM AGENCY

ORACLE®



## Level Up at Oracle Code

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

Get on the list for event updates:

[go.oracle.com/oraclecoderoadshow](https://go.oracle.com/oraclecoderoadshow)

[developer.oracle.com](https://developer.oracle.com)

#developersrule





---

iTEXT PDFライブラリ 05

J2HTML: 実行時にHTMLを生成 17

ASM: JAVAバイトコードの読取り、生成、変換 28

ライブラリをJAVAモジュールに変換する 42

2015年以降に取り上げたライブラリ集 54

---

# 便利なライブラリを 探して使う

# 開

ラリの宝庫です。

発者がJavaエコシステムを讃えるとき、特に挙げられる2つの特長は、優れた開発ツールの数々と、膨大なサードパーティ製ライブラリです。現在は、必要とされることのほぼすべてを行うためのライブラリが公開されています。その大部分はオープンソースで、自由に利用できます。Javaアーティファクトの主要なリポジトリの1つであるMaven Centralを見てみると、300万件を超えるエントリがあり、そのうち30万件近くが固有のものとなっています。まさにライブ

このような多くのアーティファクトを扱う際に役立てていただくため、Java Magazineでは定期的にライブラリ関連の記事を掲載し、年に一度はライブラリの特集号としています。本号には、ここ数年で取り上げたライブラリを簡単な説明とともにまとめた一覧 ([54ページ](#)) も掲載しました。この一覧には、暗号通貨に関するものからJVMの

内部を扱うものまで、あらゆるものが含まれています。また、ライブラリが動作する仕組みについても説明しています。たとえば、本号には、Java 9より前のライブラリをJavaモジュールに変換する方法を実践的に解説する記事 ([42ページ](#)) を掲載しています。これまで、JVMがライブラリを検索してロードする仕組み ([PDF](#)) について取り上げ、優れた[ライブラリの書き方](#)について詳しく説明してきました。

以降のページでは、PDFファイルの作成に使用できるもっとも人気のライブラリ ([5ページ](#))、テンプレートを使わずに実行時にHTMLを作成する方法 ([17ページ](#))、そしてJava/バイトコードを変換する便利な方法 ([28ページ](#)) について取り上げます。また、デザイン・パターンのうち、Stateパターン ([55ページ](#)) について詳しく見ていきます。もちろん、いつものクイズ ([65ページ](#)) も収録しています。



画像: WES ROWELL





BRUNO LOWAGIE



JORIS SCHELLEKENS

# 複雑なPDFを簡単に作成

# 人気のJavaライブラリiTextで、PDFファイルを作成して操作する方法

● iText は、Web などのアプリケーションで PDF ドキュメントの作成や処理を行う際に使われているオープンソース・ライブラリです。Java と C# の両方のバージョンが、AGPL ライセンスで公開されています。最新のメジャー・リリースは iText 7.1 で、新しい PDF 2.0 標準をサポートする最初のバージョンとなっています。

iText は、アーカイブ標準である PDF/A、視覚障害者が PDF ドキュメントにアクセスできるようにするための方法を定めた標準である PDF/UA など、さまざまな PDF 機能をサポートしています。iText を使うと、PDF 形式の請求書の作成、PDF フォームへの入力、さまざまな文書への PDF ドキュメントの組み込み、PDF ドキュメントからの個人データの削除など、ドキュメントの作成や管理に関連する多くの操作を行うことができます。

本記事では、以下に示すいくつかの iText の機能を紹介します。

- Paragraph、List、Image などの単純なオブジェクトを使って PDF ドキュメントをゼロから作成する。
- 次に、カンマ区切り値 (CSV) ファイルに格納されたデータセットを処理し、そのデータを PDF ファイルにレンダリングする。
- 最後に、HTML と CSS を使った、ブラウザにレンダリングするデータセットを読み取り、その Web ページを PDF ドキュメントに変換する。

## PDF ドキュメントの作成

図 1 に示す PDF ドキュメントは、iText を使って Java コードから作成したものです。「Hello World」というパラグラフの後に、それとは別のフォントが使われたパラグラフおよびリストからなるテキスト・ブロックが続いています。テキストの右側には、フードをかぶった開発者の画像が配置されています。

リスト 1 のコードは、このドキュメントをどのようにして作成したかを示しています。

## ■ リスト 1: Hello.java のコード

```
public void createPdf(String dest) throws IOException {
    PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
```

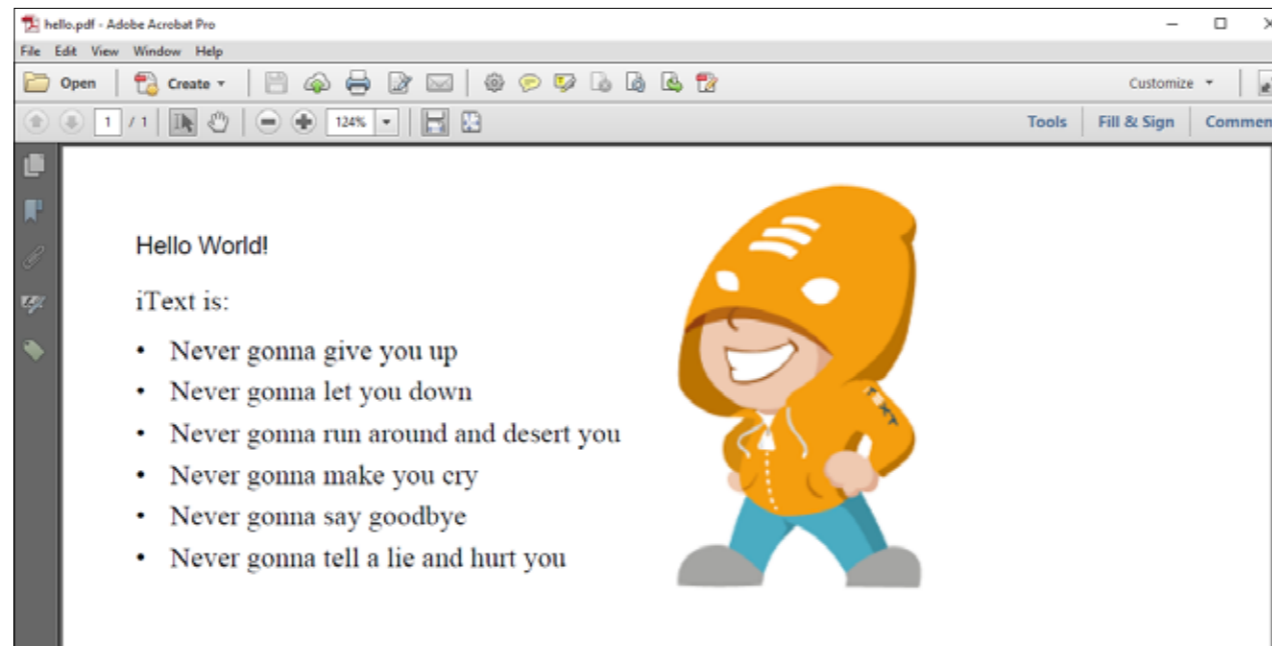


図1: iTextで作成した「Hello World」PDFドキュメント

```
Document document = new Document(pdf);
document.add(new Paragraph("Hello World!"));
PdfFont font = PdfFontFactory
    .createFont(StandardFonts.TIMES_ROMAN);
Div div = new Div().setFont(font).setFontSize(14);
div.add(new Paragraph("iText is:"));
List list = new List()
    .setSymbolIndent(12)
    .setListSymbol("\u2022");
list.add(new ListItem("Never gonna give you up"))
    .add(new ListItem("Never gonna let you down"))
    .add(new ListItem("Never gonna run around and desert you"))
    .add(new ListItem("Never gonna make you cry"))
    .add(new ListItem("Never gonna say goodbye"))
    .add(new ListItem("Never gonna tell a lie and hurt you"));
div.add(list);
```



```
document.add(div);
Image img = new Image(ImageDataFactory.create(IMG))
    .setFixedPosition(300, 625);
document.add(img);
document.close();
}
```

順を追ってこのコードを詳しく見てみます。

## 低レベル・ドキュメントとライター・オブジェクト

最初に、PdfWriter をパラメータとして、PdfDocument インスタンスを作成しました。これらは低レベル・オブジェクトです。PdfWriter は、PDF のデータを出力先に書き込む役割を担っています。この場合、出力先となるのはファイルで、そのパスが文字列によって定義されています。PdfWriter には、ファイル・オブジェクトまたは OutputStream を渡すこともできます。たとえば、メモリ内にのみ存在するドキュメントを作成したい場合は、ByteArrayOutputStream を渡して PdfWriter を作成できます。この方法は、Web アプリケーションなど、サーバー上でファイルを作成してクライアントに提供する場合によく使われます。

この時点で、PdfDocument インスタンスを使って PdfPage インスタンスを作成し、PDF ISO 32000 標準に記載されている低レベルの演算子やオペランドを使ってそのページにコンテンツを描画することができるようになっています。ただし、これはとても面倒な仕事になります。iText の高レベル・オプションを使用すると、煩雑な作業を行わずに前述のコンテンツを描画できます。

**iTextのAPIが提供するさまざまな機能を使用して、思いどおりの外観を備えたドキュメントをプログラムから作成できます。**  
ただし、便利な近道を利用することもできます。

**高レベル・オブジェクト:**次にPdfDocument インスタンスを使って Document オブジェクトを作成しました。Document には、Paragraph、List、Image などのオブジェクトを作成して追加することができます。

1. Hello World! という文字列を含む新しい **Paragraph** を追加しました。
2. **Div** オブジェクトを作成し、**PdfFontFactory** から取得した、デフォルトとは異なるフォントを設定して、フォント・サイズを 14 に設定しました。この **Div** オブジェクトには、**Paragraph** と **List** が含まれており、いずれ

の要素もフォントとフォント・サイズは親から継承されたものです。List は、インデントを 12 ポイント、箇条書き記号を黒丸に設定して作成しました。このリストには、いくつかの ListItems を追加しました。そのうえで、先ほどの Paragraph と List を Div に追加し、ドキュメントにその Div を追加しました。

**3.** `ImageDataFactory` から取得した画像データを使って `Image` オブジェクトを作成しました。ここでの `IMG` は、画像のパスを表しています。また、画像の x 座標と y 座標も定義しました。ドキュメントに画像を追加した際は、画像の左下隅がこの座標になりました。

Paragraph、Div、List、ListItem、Image は、iText で使用できる要素のほんの一部です。他にも、Text、Link、Tab、AreaBreak、LineSeparator、Table、Cell などの部品が存在します。

## ドキュメントのクローズ:

コンテンツの追加が終わったら、ドキュメントをクローズしました。ドキュメントをクローズすると、`PdfDocument` および `PdfWriter` と、`PdfWriter` が使用する `OutputStream` は自動的にクローズされます。

この例では、すべてのコンテンツがソース・コードにハードコードされていましたが、実際のアプリケーションでは、データベースなどの外部ソースからデータを取得できます。次の例では、`Table` クラスと `Cell` クラスを使って、CSV ファイルに格納されたデータが表形式でまとめられた PDF ファイルを作成しました。

## データから PDF ファイルを作成

米国のすべての州について、州名とその省略形、州都、もっとも人口の多い都市、人口、面積(平方マイル)、タイムゾーン、夏時間(DST)を使っているかどうかを一覧にまとめた CSV ファイルがあるとします。

次に、ファイルのエントリの例を示します。

CALIFORNIA;CA;Sacramento;Los Angeles;36,961,664;163,707;PT (UTC-8);;YES

この CSV ファイルを読み取って図 2 の PDF ファイルに変換する方法を見えます。

リスト2のコードでは、リスト1と同様に PdfDocument インスタンスを作成しました。次に、以下の処理を行いました。

- 最初の例で作成したページのサイズは A4 でした。米国以外ではこれがドキュメントの標準サイズになっています。この例で作成するドキュメントは、横向きの Letter サイズのページとしました。このページ・サイズ情報は、高レベルの **Document** オブジェクトを作成した際に、2 番目のパラメータとして渡しました。また、デフォルト・

name	abbr	capital	most populous city	population	square miles	time zone 1	time zone 2	dst
ALABAMA	AL	Montgomery	Birmingham	4,708,708	52,423	CST (UTC-6)	EST (UTC-5)	YES
ALASKA	AK	Juneau	Anchorage	698,473	656,425	AKST (UTC-09)	HST (UTC-10)	YES
ARIZONA	AZ	Phoenix	Phoenix	6,595,778	114,006	MT (UTC-07)		NO
ARKANSAS	AR	Little Rock	Little Rock	2,889,450	53,182	CST (UTC-6)		YES
CALIFORNIA	CA	Sacramento	Los Angeles	36,961,664	163,707	PT (UTC-8)		YES
COLORADO	CO	Denver	Denver	5,024,748	104,100	MT (UTC-07)		YES
CONNECTICUT	CT	Hartford	Bridgeport	3,518,288	5,544	EST (UTC-5)		YES
DELAWARE	DE	Dover	Wilmington	885,122	1,954	EST (UTC-5)		YES

## 図2：PDFの表にレンダリングしたデータセット

マージンを12ポイントに減らしました。

- すべてのコンテンツを、9つの列を持つ **Table** にまとめました。各列の幅は、相対幅を使用して定義しました。たとえば、第1列の幅は、第2列の幅の4倍としています。表全体の幅は、ページで利用できる幅の100%と定義しています。この使用可能幅を計算する際には、右側に12ポイント、左側に12ポイントのマージンが考慮されます。表にはヘッダーがあり、ヘッダーには通常の行とは異なるスタイルを使用したいと考えました。そこで、**Style** オブジェクトを作成し、太字のテキストがコンテナの中央ぞろえで出力されるようにしました。
- **BufferedReader** を使ってCSV ファイルを1行ずつ読み込み、最初の行(ヘッダー)は、ヘッダー用のスタイルを使って処理しました。残りの行は、ループしてスタイルを定義せずに処理しました。

以上の処理は、すべて `createPdf()` メソッドで行いました。

## ■ リスト 2: TableExample.java (パート 1)

```
public void createPdf(String dest) throws IOException {
    PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
    Document document = new Document(pdf, PageSize.LETTER.rotate());
    document.setMargins(12, 12, 12, 12);
    Table table = new Table(new float[]{4, 1, 3, 4, 3, 3, 3, 3, 1});
```

```
table.setWidth(UnitValue.createPercentValue(100));
Style header = new Style()
    .setBold().setTextAlignment(TextAlignment.CENTER);
BufferedReader br = new BufferedReader(new FileReader(DATA));
String line = br.readLine();
process(table, line, header);
while ((line = br.readLine()) != null) {
    process(table, line, null);
}
br.close();
document.add(table);
document.close();
}
```

`createPdf()` メソッドでは、`process()` メソッドを使って各行の処理を行いました。リスト 3 にこのメソッドを示します。

### ■ リスト 3: TableExample.java (ノート 2)

```
public void process(Table table, String line, Style style) {
    StringTokenizer tokenizer = new StringTokenizer(line, ";");
    while (tokenizer.hasMoreTokens()) {
        Cell cell = new Cell()
            .add(new Paragraph(tokenizer.nextToken()));
        if (style == null) {
            table.addCell(cell);
        }
        else {
            cell.addStyle(style);
            table.addHeaderCell(cell);
        }
    }
}
```

$$\left. \begin{array}{l} \{ \\ \} \end{array} \right\}$$

table.pdf - Adobe Acrobat Pro

File Edit View Window Help

name	abbr	capital	most populous city	population	square miles	time zone 1	time zone 2	dst
ALABAMA	AL	Montgomery	Birmingham	4,708,708	52,423	CST (UTC-6)	EST (UTC-5)	YES
ALASKA	AK	Juneau	Anchorage	698,473	656,425	AKST (UTC-09)	HST (UTC-10)	YES
ARIZONA	AZ	Phoenix	Phoenix	6,595,778	114,006	MT (UTC-07)		NO
ARKANSAS	AR	Little Rock	Little Rock	2,889,450	53,182	CST (UTC-6)		YES
CALIFORNIA	CA	Sacramento	Los Angeles	36,961,664	163,707	PT (UTC-8)		YES
COLORADO	CO	Denver	Denver	5,024,748	104,100	MT (UTC-07)		YES
CONNECTICUT	CT	Hartford	Bridgeport	3,518,288	5,544	EST (UTC-5)		YES
DELAWARE	DE	Dover	Wilmington	885,122	1,954	EST (UTC-5)		YES
FLORIDA	FL	Tallahassee	Jacksonville	18,537,969	65,758	EST (UTC-5)	CST (UTC-6)	YES
GEORGIA	GA	Atlanta	Atlanta	9,829,211	59,441	EST (UTC-5)		YES
HAWAII	HI	Honolulu	Honolulu	1,295,178	10,932	HST (UTC-10)		NO
IDAHO	ID	Boise	Boise	1,545,801	83,574	MT (UTC-07)	PT (UTC-8)	YES
ILLINOIS	IL	Springfield	Chicago	12,910,409	57,918	CST (UTC-6)		YES
INDIANA	IN	Indianapolis	Indianapolis	6,423,113	36,420	EST (UTC-5)	CST (UTC-6)	YES
IOWA	IA	Des Moines	Des Moines	3,007,856	56,276	CST (UTC-6)		YES
KANSAS	KS	Topeka	Wichita	2,818,747	82,282	CST (UTC-6)	MT (UTC-07)	YES
KENTUCKY	KY	Frankfort	Louisville	4,314,113	40,411	EST (UTC-5)	CST (UTC-6)	YES
LOUISIANA	LA	Baton Rouge	New Orleans	4,492,076	51,843	CST (UTC-6)		YES
MAINE	ME	Augusta	Portland	1,318,301	35,387	EST (UTC-5)		YES
MARYLAND	MD	Annapolis	Baltimore	5,699,478	12,407	EST (UTC-5)		YES
MASSACHUSETTS	MA	Boston	Boston	6,593,587	10,555	EST (UTC-5)		YES
MICHIGAN	MI	Lansing	Detroit	9,969,727	96,810	EST (UTC-5)	CST (UTC-6)	YES
MINNESOTA	MN	Saint Paul	Minneapolis	5,266,214	86,943	CST (UTC-6)		YES
MISSISSIPPI	MS	Jackson	Jackson	2,951,996	48,434	CST (UTC-6)		YES
MISSOURI	MO	Jefferson City	Kansas City	5,987,580	69,709	CST (UTC-6)		YES

name	abbr	capital	most populous city	population	square miles	time zone 1	time zone 2	dst
MONTANA	MT	Helena	Billings	974,989	147,046	MT (UTC-07)		YES
NEBRASKA	NE	Lincoln	Omaha	1,796,619	77,358	CST (UTC-6)	MT (UTC-07)	YES
NEVADA	NV	Carson City	Las Vegas	2,643,085	110,567	PT (UTC-8)	MT (UTC-07)	YES
NEW HAMPSHIRE	NH	Concord	Manchester	1,324,575	9,351	EST (UTC-5)		YES
NEW JERSEY	NJ	Trenton	Newark	8,707,739	8,722	EST (UTC-5)		YES
NEW MEXICO	NM	Santa Fe	Albuquerque	2,009,671	121,593	MT (UTC-07)		YES

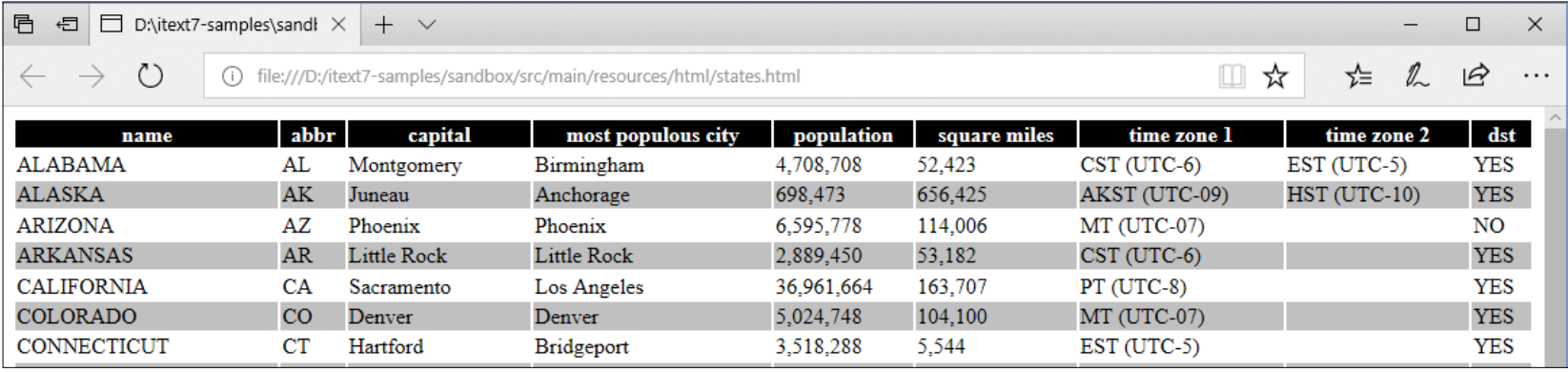
図3: 繰返しヘッダーとともに複数のページにわたって出力された行



```

<html>
<head>
  <meta http-equiv="Content-Type" content="text/html;
    charset=windows-1252">
  <style>
    th { background: black; color: white; }
    tr:nth-child(even) { background: #C0C0C0; }
    @page {
      size:A4 landscape;
      @bottom-right {
        content:"Page " counter(page) " of " counter(pages);
      }
    }
  </style>
</head>
<body>
<table width="100%">
  <thead>
    <tr>
    <th>name</th>
    <th>abbr</th>
    <th>capital</th>

```



name	abbr	capital	most populous city	population	square miles	time zone 1	time zone 2	dst
ALABAMA	AL	Montgomery	Birmingham	4,708,708	52,423	CST (UTC-6)	EST (UTC-5)	YES
ALASKA	AK	Juneau	Anchorage	698,473	656,425	AKST (UTC-09)	HST (UTC-10)	YES
ARIZONA	AZ	Phoenix	Phoenix	6,595,778	114,006	MT (UTC-07)		NO
ARKANSAS	AR	Little Rock	Little Rock	2,889,450	53,182	CST (UTC-6)		YES
CALIFORNIA	CA	Sacramento	Los Angeles	36,961,664	163,707	PT (UTC-8)		YES
COLORADO	CO	Denver	Denver	5,024,748	104,100	MT (UTC-07)		YES
CONNECTICUT	CT	Hartford	Bridgeport	3,518,288	5,544	EST (UTC-5)		YES

図4: ブラウザに表示された米国の州



```
<th>most populous city</th>
<th>population</th>
<th>square miles</th>
<th>time zone 1</th>
<th>time zone 2</th>
<th>dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALABAMA</td>
<td>AL</td>
<td>Montgomery</td>
<td>Birmingham</td>
<td>4,708,708</td>
<td>52,423</td>
<td>CST (UTC-6)</td>
<td>EST (UTC-5)</td>
<td>YES</td>
</tr>
<tr>
<td>
```

HTML コードをさらに詳しく見てみると、ヘッダーが `<thead>` セクションにあることに気づくでしょう。これによって、すべてのページでヘッダーが繰り返されるようになっています。`<tbody>` セクションには、50 のデータ行が存在します。

この HTML ファイルを PDF に変換するコードは非常に簡単で、リスト 5 に示す 1 行だけです。

## ■ リスト 5:

```
public void createPdf(String html, String dest) throws IOException {
    HtmlConverter.convertToPdf(new File(html), new File(dest));
}
```



**Joris Schellekens** (@Joris1989BE) : iText のリサーチ・エンジニア。機械学習や自然言語処理などの破壊的技術を重点的に扱う。新しいテクノロジーに情熱を抱き、PDF の課題を解決する方法を探し続けている。研究や iText のコーディングをしていないときは、自身のコーディング・プロジェクトや数学プロジェクトの作業、音楽鑑賞をしている。





Javaのタイプ保証とツール・エコシステムを活用し、  
動的に生成される小さなWebアプリケーションを簡単に作成する

j2htmlはテンプレート・エンジンではないことに注意してください。しかし、テンプレート・エンジンとは異なる方法で、タイプ保証しつつUIコードを動的に作成し、再利用することができるようになっています。このライブラリの作者はDavid Åse氏で、同氏が積極的なメンテナンスを行っています。本記事では、ライブラリを使い始めるにあたって知っておくべき点を説明し、その後、高度なユースケースを紹介したいと思います。以下に示すサンプルのソースは、[ダウンロード](#)することができます。

j2htmlの最新バージョンは、依存性を次のように定義して、Mavenのセントラル・リポジトリから入手することができます。本記事の執筆時点で、入手可能な最新バージョンは1.3.0です。

Gradleをお使いの場合は、次のようにして依存性を追加できます。

```
compile 'com.j2html:j2html:1.3.0'
```

j2htmlのバージョン1.2.2以降では、JDK 8が最小要件となっています。

それでは、最初のHTMLコードを生成してみます。中に見出しのあるbodyを作るのは簡単です。Listing 1をご覧ください。

#### ■ リスト1:

```
html(  
    body(  
        h1("Hello, Java Magazine Readers!")  
    )  
);
```

このコードにより、Listing 2に示すHTMLが出力されます。

#### ■ リスト2:

```
<html>  
  <body>  
    <h1>Hello, Java Magazine Readers!</h1>  
  </body>  
</html>
```

`body()` メソッドと `h1()` メソッドは、静的にインポートされています。この2つのメソッドは、このライブラリでもっとも重要なクラスの1つである、`TagCreator` クラスで定義されています。このクラスの静的インポートは、Listing 3に示す構文で行うことができます。

#### ■ リスト3:

```
import static j2html.TagCreator.*;
```

続いて、もう少し複雑な例を紹介します。Listing 4は、表を作成し、それぞれの行に従業員を表示するためのコードです。このページのheadセクションでは、スタイルシートのファイルを参照しています。また、従業員の表の上に、



ヘッダーとなるdivを設けています。

■ リスト4:

```
List<ContainerTag> rows = new ArrayList<>();
rows.add(tr().with(
    td().withText("Mert"),
    td().withText("Caliskan")
));
```

```
String output =
    html(
        head(
            title("Java Magazine Examples"),
            link().withRel("stylesheet").withHref("my.css")
        ),
        body(
            div().withId("header").with(h1("Employees")),
            table().withClass("tableClass").with(
                thead(
                    tr(
                        th("Name"), th("Last Name")
                    )
                )
            ).with(tbody().with(rows)),
            footer().withClass("footerClass")
        )
    ).render();
```

このコードでは、行のリストを`ContainerTag`（このクラス階層については、次のリストの後で詳しく説明します）のリストとして定義しています。このリストが、表の中にレンダリングされるコンテンツに対応します。次に、`html()`メ



ソッドでHTMLのコンテナとなるタグを定義し、`head()`メソッドでページのヘッダーを設定しています。また、リソースを組み込む方法を紹介するため、`link()`メソッドでヘッダー内部にスタイルシート・ファイルへの参照を作成しています。出力される結果をリスト5に示します。

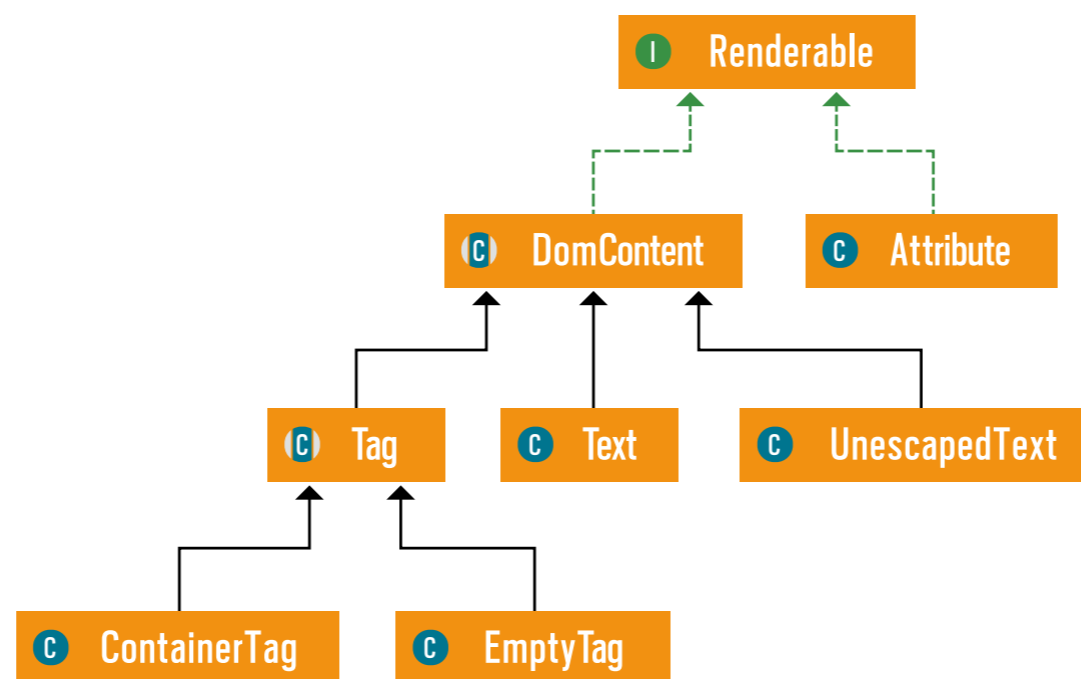
■ リスト5:

```
<html>
  <head>
    <title>Java Magazine</title>
    <link rel="stylesheet" href="my.css">
  </head>
  <body>
    <div id="header">
      <h1>Employees</h1>
    </div>
    <table class="tableClass">
      <thead>
        <tr>
          <th>Name</th>
          <th>Last Name</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Mert</td>
          <td>Caliskan</td>
        </tr>
      </tbody>
    </table>
    <footer class="footerClass"></footer>
  </body>
</html>
```

Figure 1にクラス階層の重要な部分を示します。 `head()`、`body()`、`div()`などのメソッドで参照されるタグは、すべてライブラリの`ContainerTag`のインスタンスで表されています。`ContainerTag`は、タグの名前と、`List<DomContent>`で参照される子要素を一緒に格納します。`DomContent`は、HTMLドキュメント内で使われるテキストまたタグのいずれかを指す抽象クラスです。`ContainerTag`は、文字列として表現できるコンテンツを定義する`Renderable`インタフェースも実装しています。このインタフェースは、`render()`メソッドのデフォルト実装を提供します。`Attribute`クラスでは、HTML要素の属性を定義します。属性の値は`with*()`メソッド構文(`withRel()`、`withHref()`、`withClass()`など)で設定できます。Listing 4で使ったのもこの構文です。

## 掘り下げて調べる

j2htmlでは、HTMLを生成するためのさまざまなアプローチが提供されています。このセクションでは、その中からいくつかを紹介します。join()メソッドを使うと、テキストとタグを簡単につなぎ合わせることができます。このメソッドでは、各入力要素を指定された順番どおりに空白で区切って結合できます。その際、ピリオドおよびカンマの前にある空白の削除も行われます。Listing 6に例を示します。



### 図1: クラス階層の重要部分

```
body(
  p(join(
    i("Java Magazine"),
    "is a bimonthly deep dive into",
    b("Java"),
    "and the",
    b("JVM.")
  )
)
);
```

このスニペットにより、Listing 7に示すHTMLが出力されます。

## ■ リスト7:

```
<body>
  <p>
    <i>Java Magazine</i> is a bimonthly deep dive into
      <b>Java</b> and the <b>JVM.</b>
  </p>
</body>
```

`id`属性と`class`属性は、CSSセレクタで使用する文字列により記述できます。Listing 8の`attrs()`メソッドでは、`body`の`id`属性を`main`に、`class`属性を`content`に設定しています。

■ リスト8:

```
...
body(attrs("#main.content"),
      h1("Heading!"))
);
```

このコードでは、`attrs()`メソッドによって、`Attr.ShortForm`クラスのインスタンスを作成しています。このクラス



は、`id`と`classes`という2つの文字列プロパティを格納するラッパー・クラスです。`Attr`クラスには、タイプ保証されたHTML属性が列挙型として含まれています。そのため、`Attr.ID`、`Attr.HEIGHT`、`Attr.WIDTH`などとして属性を使用できます。

`iff()` メソッドや `iffElse()` メソッドを使うと、条件付き生成も可能です。Listing 9をご覧ください。

## ■ リスト9:

```
...
div().withClasses("menu",
    iff(isActive, "active")
);

div().withClasses("item",
    iffElse(isSelected, "selected", "not-selected")
);
```

isSelectedの値をtrueに設定すると、Listing 10のように出力されます。

## ■ リスト10:

```
<div class="menu active"></div>

<div class="item selected"></div>
```

コード・スニペットを作成しておき、それを再利用して、頻繁に使用するHTMLスニペットを生成することができます。Listing 11では、ユーザー名およびパスワードを入力するフィールドとフォームの送信ボタンを含むHTMLコンテンツを生成するコードを再利用し、ログイン・フォームを作成しています。genericInput()メソッドでは、指定された型、名前、プレースホルダ値を使って入力フィールドを作成する方法を定義しています。

## ■ リスト11:

```
form().withMethod("post").with(
    genericInput("text", "uname", "Enter Username"),
```

```
        genericInput("password", "psw", "Enter Password"),
        submitButton()
    );

    private static Tag genericInput(String type,
                                    String name,
                                    String placeholder) {
        return input()
            .withType(type)
            .withId(name)
            .withName(name)
            .withPlaceholder(placeholder)
            .isRequired();
    };

    private static Tag submitButton() {
        return button("Login").withType("submit");
    };
};
```

Listing 12に、出力されるログイン・フォームのHTMLを示します。

### ■ リスト12:

```
<form method="post">
  <input type="text" id="uname" name="uname"
    placeholder="Enter Username" required>
  <input type="password" id="psw" name="psw"
    placeholder="Enter Password" required>
  <button type="submit">
    Login
  </button>
</form>
```



フィルタリング・ルールを適用すると、j2htmlの`each()`メソッドや`filter()`メソッド、あるいはJava 8の`stream()`メソッドや`filter()`メソッドを使って条件付きHTMLを生成できます。Listing 13のコードでは、従業員idが偶数である従業員のリストをフィルタリングしています。最初にj2htmlのメソッドを使い、次にJava 8のストリームとフィルタリングのアプローチを使っています。

## ■ リスト13:

```
String j2htmlFilter = ul().with(
    each(filter(employees, e -> e.id % 2 == 0),
        employee -> li(
            h2(employee.name),
            p(employee.title)
        )
    )
).render();
```

```
String javaFilter = ul().with(
    rawHtml(employees.stream()
        .filter(e -> e.id % 2 == 0).map(
            employee -> li(
                h2(employee.name),
                p(employee.title)
            )
        )
    ).map(DomContent::render)
    .collect(Collectors.joining()))
).render();
```

`div()`メソッドで生成される`<div>`の内部にカスタムのHTMLスニペットを配置したい場合は、`rawHtml()`メソッドを使用できます。Listing 14をご覧ください。

## ■ リスト14:

```
...
div(
  rawHtml("<p>I like <em>HTML</em></p>")
);
```

このメソッドで作成されるのは、テキスト表現のラッパー・クラスである`UnescapedText`クラスのインスタンスです (Figure 1をご覧ください)。

## JavaScriptとCSSリソースの処理

TagCreatorには、HTMLの<style>タグを作成するstyle()メソッドと、<script>タグを作成するscript()メソッドがあります。Listing 15に、リソースを含むサンプルを示します。

## ■ リスト15:

```
script(  
  rawHtml("alert('ok')")  
);  
  
style(  
  rawHtml("body {background-color: blue}")  
);
```

TagCreatorには、ファイルから直接JavaScriptリソースやCSSリソースをロードする静的メソッドも定義されています。Listing 16に例を示します。`_min`接尾辞が付いたメソッドを使うと、コンテンツを文字列表現に変換しつつ、圧縮することができます。

## ■ リスト16:

```
scriptWithInlineFile("/test.js");
styleWithInlineFile("/test.css");
```

```
scriptWithInlineFile_min("/test.js");
styleWithInlineFile_min("/test.css");
```

JavaScript用のデフォルト圧縮ツールは、単純に空白と改行を取り除くものです。CSS用のデフォルト圧縮ツールは、[CSSMin](#)です。なお、これらのメソッドでロードしたコンテンツは、生成されるHTMLコードではインライン・コンテンツとなることに留意してください。リソースの実際のロードは、[InlineStaticResource](#)クラス内で実装されています。このクラスでは、[script\(\)](#)メソッドや[style\(\)](#)メソッドを使ってコンテナとなるタグを作成し、その入力として、指定されたファイルから読み取ったコンテンツを使います。

## まとめ

タイプ保証された方法を使って宣言的にサーバー・サイドでHTMLコンテンツを生成する必要がある場合、j2htmlはぜひともクラス・パスに入れておきたいソリューションです。ビルダー的な手法を用いて、HTMLコンポーネントと1対1で対応付けられた宣言的なJavaコードを書くことが、これほど容易なことはありませんでした。

j2htmlによるHTMLの生成は、既存のテンプレート・エンジンよりも高速です。そのため、パフォーマンス上の問題を抱えている場合、このライブラリを試してみる価値はあります。しかし、j2htmlはテンプレート・エンジンではなく、それに匹敵する機能もありません。静的なWebサイトを構築する場合、j2htmlは使用しない方がよいでしょう。コードに挿入するコンテンツをすべて生成しなければならない可能性が高いからです。BootstrapなどのCSSフレームワークを使用している場合、そこでj2htmlはうまく動作しないでしょう。そのため、j2htmlは自己責任で使用してください。

しかし、動的に作成される小さなWebアプリケーションを扱っている場合、j2htmlが、ポケットに入れておきたいスイス・アーミー・ナイフのようなツールになることは間違いありません。

**Mert Çalışkan** (@mertcal) : Java Champion。OpsGenie Academyディレクター。

『PrimeFaces Cookbook』(Packt Publishing、2013年) および『Beginning Spring』(Wiley Publications、2015年) の共著者。最新刊となる『Java EE 8 Microservices』を執筆しつつ、Payara FoundationでPayara Serverの開発に従事している。

写真: JOHN BLYTHE

ASMライブラリを使用して、実行時にバイトコードをスキャン、検査、生成、変換する

**A**SMライブラリは、本番環境で利用できる品質を備えたオープンソース・ライブラリで、JVMバイトコードの読取り、書込み、操作を行うことができます。このライブラリは、GradleやKotlinコンパイラを含む多くのプロジェクトの一部として使用され、JDK内部でもシェードされた形（つまり、ネームスペースの衝突を避けるために、コードをコピーしてパッケージの名前を変更した状態）で使用されています。実は、ラムダ式の実行時サポートを実現するコード生成エンジンとして使用されているのも、このASMライブラリです。なお、ASMを使用する場合、JDK内部にあるシェードされたものではなく、外部のバージョンを使用する必要があります。

本記事では、ASMを使ってさまざまな便利な操作を行う方法について説明します。以下の内容は、JVMバイトコードの基本とクラス・ファイルの構造についてすでに十分理解している方を対象にしています。本記事で紹介するコードは、[Java Magazineダウンロード・ページ](#)に掲載されています。

まずは、おなじみの例を取り上げることとし、「Hello World!」と出力するクラスを作成してみます。ここでは、ASM の `ClassWriter` API を使います。これはシンプルなAPIで、Visitor/ パターンを多用して目的を実現します。

今回の例では、まったく何もないところから新しいクラス・ファイルHelloWorld.classを作成します。このクラスには、元になるJavaソース・コードは存在しません。つまり、コンパイル済みのクラスの形でしか存在しないということです。

HelloWorld.classファイルを作成するのは、[MakeHelloWorld](#)という別のクラスです。このクラスは、ASMライブラリを使ってアセンブルを行い、HelloWorld.classを出力します。なお、生成された出力クラスは、完全にスタンドアロンで実行できます。実行時依存性としてASMなどのJARが必要になることはありません。

MakeHelloWorldで行っているクラス作成処理の全体像を説明します。このクラスは、`cw`という名前の

`ClassWriter`フィールドを使い、クラスに関する以下の情報を順番に訪問 (visit) しながら、クラスを構築していきます。

- すべてのメタデータ
- コンストラクタ本体
- `main` メソッドの定義とそのバイトコード

クラスに関する上記すべての情報を訪問したら、`visitEnd()`を呼び出してライター・オブジェクトをシリアライズする準備を整え、ディスクに書込み可能なバイト配列に変換できます。

次に示すのは、シリアルライズ処理を起動するメソッドのコードの全体像です。このメソッドは、出力するクラスの名前を渡すだけで呼び出すことができます。

```
public byte[] serializeToBytes(String outputClassName) {
    cw.visit(V1_8, ACC_PUBLIC + ACC_SUPER, outputClassName,
        null, "java/lang/Object", null);
    addStandardConstructor();
    addMainMethod();
    cw.visitEnd();
    return cw.toByteArray();
}
```

シリアライズを行うこのメソッドは、まずトップレベルのメタデータ（クラス・ファイルのバージョン、フラグ、クラス名、スーパークラス名）を訪問します。次にコンストラクタとメイン・メソッドを追加するメソッドを呼び出し、最後にクラスを固定化してバイト配列に変換します。

コンストラクタは、次のようにして作成します。

```
void addStandardConstructor() {
    MethodVisitor mv =
        cw.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
    mv.visitVarInsn(ALOAD, 0);
    mv.visitMethodInsn(
        INVOKESPECIAL, "java/lang/Object", "<init>", "()V", false);
}
```

```
mv.visitInsn(RETURN);
mv.visitMaxs(1, 1);
mv.visitEnd();
}
```

このコードでは、`ClassWriter`フィールドから`MethodVisitor`を作成し、それを使ってそれぞれの命令を訪問しています。その後、メソッドの処理を終える際に、コードが使用したスタック・スロットの数を明示する必要があります。これを行うために、`visitMaxs()`を呼び出しています。

メイン・メソッドは、別のMethodVisitorを使って追加します。

```
void addMainMethod() {
    MethodVisitor mv =
        cw.visitMethod(ACC_PUBLIC + ACC_STATIC,
            "main", "(Ljava/lang/String;)V", null, null);
    mv.visitCode();
    mv.visitFieldInsn(GETSTATIC, "java/lang/System",
        "out", "Ljava/io/PrintStream;");
    mv.visitLdcInsn("Hello World!");
    mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
        "println", "(Ljava/lang/String;)V", false);
    mv.visitInsn(RETURN);
    mv.visitMaxs(3, 3);
    mv.visitEnd();
}
```

このコードは今までよりも少しだけ複雑ですが、それは静的フィールドからオブジェクトを取得する必要があるためです (GETSTATIC 命令コードを使用してオブジェクトを取得してから、メソッドを呼び出さなければなりません)。

`MakeHelloWorld`を実行すると、ファイル・システム上にHelloWorld.classが現れます。生成されたクラスは、通常の方法で実行できます。java HelloWorldを実行すると、おなじみのメッセージが表示されます。

ASMのビジターAPIは、このライブラリが提供するAPIの中ではわかりやすいものです。一般的な原則は、クラ

ス・ファイルの各セクションを正しい順序で訪問（または、そのセクションに何も必要ない場合はスキップ）する必要があります。 [MethodVisitor](#) インタフェースはごく一般的なものです。

MakeHelloWorldでは、ClassWriterからビジターを取得しました。実際にこのインタフェースを実装しているのは、MethodWriterです。このオブジェクトは、作成元となったClassWriterへの参照を保持しており、さまざまなvisitメソッドが呼び出されるたびに、メソッドに関するメタデータを蓄積できます。

`MethodWriter`が表しているメソッドは、訪問が完了した後にクローズする必要があるため、HelloWorld内のメソッドを作成するメソッドの最後のアクションとして、`mv.visitEnd()`を呼び出しています。

それでは、生成されたクラスをjavap -c HelloWorld.classで逆コンパイルして、ASMが生成したクラスのバイトコードを見てみます。

```
public class HelloWorld {
    public HelloWorld();
    Code:
        0: aload_0
        1: invokespecial #8 // Method java/lang/Object."<init>":()V
        4: return

    public static void main(java.lang.String[]);
    Code:
        0: getstatic    #16 // Field
                    // java/lang/System.out:Ljava/io/PrintStream;
        3: ldc         #18 // String Hello World!
        5: invokevirtual #24 // Method
                    // java/io/PrintStream.println:(Ljava/lang/String;)V
        8: return
}
```

`main()` メソッドでは特に、Java バイトコード 命令と ビジター API の呼出しとの対応が明確です。ASM を全体的に見渡せば、バイトコード形式に非常に近い形を維持しつつ、生産性を高めるために十分な高水準 API を提供していることがわかります。

それでは、もう少し複雑な、さらに2つの例について考えてみます。

- カウンタを安全にインクリメントしようとする際に生じる「ロスト・アップデート」問題
- ユーザー・コードでネイティブ・メソッドを実行できないようにする 「安全なクラス・ローダー」のプロトタイプ

## ロスト・アップデート保護の無効化

まずは、簡単なコードを使って、ロスト・アップデートを実際に起こしてみます。ロスト・アップデートの詳細については、後ほど説明します。この影響を説明する際によく用いられるクラシックな例の1つが、インクリメントを行うクラスです。

```
public class Counter {  
    private int i = 0;  
  
    public int increment() {  
        return i = i + 1;  
    }  
}
```

このクラスを起動する、別のクラスも必要です。

```
int MAX_INC = 10_000_000;  
Counter c = new Counter();  
Runnable r = () -> {  
    for (int i = 0; i < MAX_INC; i++) {  
        c.increment();  
    }  
};  
Thread t1 = new Thread(r);  
Thread t2 = new Thread(r);  
t1.start();  
t2.start();  
t1.join();  
t2.join();
```



```
int disc = 2 * MAX_INC - c.increment() + 1;
System.out.println("Discrepancy: " + disc);
```

このコードでは、各スレッドで1000万回インクリメントが行われることから、`increment()`は合計で2000万回呼ばれることになります。しかし、このコードを実行すると、食い違いが生じていることがはっきりとわかります。`increment()`のすべての呼出しがcで記録されているわけではないように見えます。

これがロスト・アップデートと呼ばれるアンチパターンで、`increment()`のような単純なコードでも発生する可能性があります。このパターンは、現代の環境における並行プログラミングでよく起きる典型的な落とし穴の1つです。

ロスト・アップデートは、CPUコア上で両方のスレッドを同時に実行する、オペレーティング・システムのスケジューラによって引き起こされます。各スレッドは、ローカルCPUキャッシュで見たiの値をインクリメントしますが、その結果をメイン・メモリにフラッシュすることはありません。そのため、CPUがキャッシュ・ラインをメイン・メモリにフラッシュする前に、両方のスレッドから不定回の更新が行われることになります。キャッシュのみへのこのような書込みは、メイン・メモリに記録される、全体の合計には反映されません。

言うまでもありませんが、この問題を解決するためには、`synchronized`キーワードを`increment()`に追加します。すると、`i`に対するすべての更新が、再度読み取られる前にメイン・メモリにフラッシュされるようになり、先ほどのような食い違いはなくなります。

このことを確認するために、まず同期カウンタを作成し、ASMを使って、クラス内のすべての同期をオフにするツールを記述します。すると、元のコードが安全だった場合でも、変換済みクラスではロスト・アップデート問題が発生します。

同期をオフにする[OfflineUnsyncronizer](#)のコードは、次のような仕組みで動作します。

- `ClassReader`を使って、クラス・ファイルを読み取る。
- カスタムの`ClassVisitor`を使って、クラスのASM表現を確認する。
- `ClassWriter`を使って、Javaバイトコードを`byte[]`形式で書き戻す。
- バイトコードを変換済みクラス・ファイルとして保存する。

この変換を行うには、Java/バイトコードについて、ある程度の詳細を知っておく必要があります。

たとえば、Javaバイトコードでのsynchronizedメソッドは、ACC\_SYNCHRONIZEDというフラグを対象のメソッドに追加することで表現されます。したがって、訪問先のメソッドからこのフラグを削除する必要があります。しかし、確実にすべての同期をなくすためには、ブロック方式の同期が若干異なる形で表現されることも知ってお

く必要があります。たとえば、次のコードがあるとします。

```
Object o = ...
synchronized (o) {
    // ...
}
```

このコードは、次のようなバイトコード・シーケンスに変換されます。

[oをスタックの最上位に置くシーケンス]  
monitorenter

// ...

[oの再ロード]  
monitorexit

monitorenterおよびmonitorexitというバイトコードは、スタックの先頭を取り出し、そこにあるオブジェクトをロックまたはロック解除します。したがって、この2つの命令コードを基本命令であるpopに置き換えれば、見つかったどのメソッド本体からも同期機能が取り除かれることになります。

これを実現するコードは、[UnsynchronizingClassVisitor](#)と[UnsynchronizingMethodVisitor](#)という2つのシンプルなクラスです。いずれも、ASMフレームワークのクラスを拡張しています。

```
public class UnsynchronizingClassVisitor extends ClassVisitor {

    public UnsynchronizingClassVisitor(int api, ClassVisitor cv) {
        super(Opcodes.ASM5, cv);
    }

    @Override
    public MethodVisitor visitMethod(int flags, String name,
```

```
String desc, String signature, String[] exceptions) {
    int maskedFlags = flags & (~ACC_SYNCHRONIZED);

    MethodVisitor baseMethodVisitor =
        super.visitMethod(maskedFlags, name, desc,
            signature, exceptions);

    return new UnsynchronizingMethodVisitor(baseMethodVisitor);
}
```

`UnsynchronizingClassVisitor`クラスは、Decoratorパターンを使用します。すなわち、`baseMethodVisitor`を取得し、それをラップして、メソッド本体で引数のない命令コードを見つけた場合にのみ呼び出される機能を追加しています。次のコードをご覧ください。

```
public class UnsynchronizingMethodVisitor extends MethodVisitor {
    public UnsynchronizingMethodVisitor(MethodVisitor mv) {
        super(Opcodes.ASM5, mv);
    }

    @Override
    public void visitInsn(final int opcode) {
        switch (opcode) {
            case Opcodes.MONITORENTER:
            case Opcodes.MONITOREXIT:
                super.visitInsn(Opcodes.POP);
                return;
        }

        super.visitInsn(opcode);
    }
}
```



}

この変換を起動するために、次のコードを使用します。

```
try (InputStream in =
    Files.newInputStream(Paths.get(fName))) {
    ClassReader classReader = new ClassReader(in);
    ClassWriter writer =
        new ClassWriter(ClassWriter.COMPUTE_FRAMES);

    ClassVisitor unsynchronizer =
        new UnsynchronizingClassVisitor(writer);
    classReader.accept(unsynchronizer,
        ClassReader.SKIP_FRAMES | ClassReader.SKIP_DEBUG);

    Path newClazz = Paths.get(transformName(fName));
    Files.write(newClazz, writer.toByteArray());
} catch (Exception ex) {
    System.err.println(
        "Exception whilst reading class: " + fName);
    ex.printStackTrace(System.err);
}
```

同期バージョンのCounterクラスを、OfflineUnsynchronizerを通してから実行すると、その結果生成された変換済みクラスでは、元のコードが安全だった場合でも、ロスト・アップデート問題が発生します。

## ネイティブ・コードの排除

Javaバイトコードはプラットフォームに依存しないため、オペレーティング・システムのライブラリ (I/O操作の処理など) を直接呼び出すことはできません。その代わりに、Javaプログラム (JDKを含む) は、オペレーティング・システムの関連部分呼び出すネイティブ・メソッド (C言語で書かれています) を呼び出します。



ユーザーがフレームワークまたはコンテナの一部として未知のコードを実行することを許可したいユースケースがあるとしてします。このような機能には、明らかにセキュリティ上の問題があります。そのため、ユーザーのクラスで特定の操作（ネイティブ・メソッドの実行など）を禁止することで、リスクを軽減した方がよいでしょう。幸いにも、Javaセキュリティ・モデルはクラス・ローディングに依存しています。つまり、ローディング・プロセスにフックすることで、新しいコードのロード方法（およびロードするかどうか）をカスタマイズすることができます。[編集注：クラス・ローディングの詳しい仕組みについては、Oleg Selajev氏による記事「JVMによるライブラリの検索、ロード、実行」をご覧ください。PDF形式でダウンロードすることもできます。]

この処理の全体像は、次のようになります。

- カスタム・クラス・ローダーを作成する。
- クラスをロードする際に、メソッドを呼び出している場所である「コール・サイト」をすべて検査する。
- メソッドのメタデータから、メソッドがネイティブであるかどうかを確認する。
- ネイティブ・メソッドの場合、クラスを拒否し、クラスのロードを失敗させる。
- 失敗することなく最後まで到達した場合、クラスに問題はなく、ロードが許可される。

ロード要求に対し、純粋なJavaクラス以外は拒否するクラス・ローダーの書き方を次に示します。

```
public final class PureJavaClassLoader extends ClassLoader {
    private final List<String> auxClasspath = new ArrayList<>();
```

```
public PureJavaClassLoader(ClassLoader parent) {
    super(parent);
}
```

```
public void setupClasspath(final String auxiliaryClassPath) {
    for (String entry : auxiliaryClassPath.split(":")) {
        if (entry.startsWith("/")) {
            auxClasspath.add(entry);
        } else {
            System.err.println(
                "Bad classpath entry seen: " +
                entry + ", ignoring");
        }
    }
}
```

```

    }
}

Path findClassFile(String qualifiedClassName)throws IOException {
    final String fileName =
        qualifiedClassName.replaceAll("/", "\\."). + ".class";
    for (String s : auxClasspath) {
        Path trial = Paths.get(s, fileName);
        if (trial.toFile().exists())
            return trial;
    }

    throw new IOException("Class " + qualifiedClassName +
        " not found on classpath");
}
}

```

ここでは、わかりやすさを優先して、メイン・クラス・パスを使用するのではなく、ロードするクラスを検索するディレクトリを補助クラス・パスとして管理しています。findClassFileメソッドは、修飾されたクラス名に対応するファイルを検索するヘルパーです。実際のアクションは、クラス・ローダーがloadClass()から委譲されるfindClassメソッドにあります。ネイティブ・コードのチェック機能はそこで実装しています。

```
@Override
public Class<?> findClass(final String qualifiedClassName) throws
    ClassNotFoundException {

    Class<?> cls = null;

    try {
        return super.findClass(qualifiedClassName);
    } catch (ClassNotFoundException ignored) {
```

```
try (final InputStream in = Files.newInputStream(
    findClassFile(qualifiedClassName))) {
    final byte[] allClassBytes = in.readAllBytes();
    final ClassReader classReader =
        new ClassReader(allClassBytes);
    final PureJavaCheckingClassVisitor
        classVisitor =
            new PureJavaCheckingClassVisitor();

    // クラスにデバッグ情報がある場合、
    // それを無視する
    classReader.accept(
        classVisitor, ClassReader.SKIP_DEBUG);

    if (classVisitor.containsNative()) {
        throw new ClassNotFoundException(
            "Class cannot be loaded - contains native code");
    } else {
        return defineClass(null, allClassBytes, 0,
            allClassBytes.length);
    }
} catch (IOException e) {
    throw new ClassNotFoundException(
        "Error finding and opening class", e);
}
```

上記のコードでは、`in`を`ClassReader`のコンストラクタに渡すのではなく、`in.readAllBytes()`を直接呼び出しています。これは、ASMのクラス`ClassReader`は入力ストリームを消費してしまうため、クラス・リーダーを作成する際に`in`を使ってしまうと、それを再利用できなくなるからです。

次に、カスタムのクラス・ビジターのインスタンスである[PureJavaCheckingClassVisitor](#)を作成しています。このビジターは、対象クラス内の各メソッドのメタデータを訪問し、ネイティブ・メソッドがあるかどうかを記録します。このクラスは、次のように定義されています。

```
public class PureJavaCheckingClassVisitor extends ClassVisitor {
    private boolean containsNative = false;

    public PureJavaCheckingClassVisitor() {
        super(Opcodes.ASM5);
    }

    @Override
    public MethodVisitor visitMethod(int flags, String name,
        String desc, String signature, String[] exceptions) {
        if ((flags & ACC_NATIVE) > 0) {
            containsNative = true;
        }

        return new MethodVisitor(Opcodes.ASM5) {};
    }

    public boolean containsNative() {
        return containsNative;
    }
}
```

クラス・ビジターがネイティブ・メソッドを見つけた場合、フラグを設定します。PureJava-ClassLoaderは、このフラグの有無を読み取り、フラグが設定されている場合、ClassNotFoundExceptionをスローしてクラスのロードを拒否します。ここでの選択として自然に思える例外（SecurityException）ではなく、この例外を使っているのは、ClassLoader（このクラスのスーパータイプ）の取り決めで、チェック例外ClassNotFoundExceptionが使われているからです。この状況で実行時例外（Security-

Exceptionなど) を使用した場合、クラス・ローダーのクライアントにとって予期しない動作になる可能性があります。

例外がスローされなかった場合、クラス・ファイルのバイト列が`defineClass()`に渡されます。これは `ClassLoader` で定義されている `protected` メソッドであるため、サブクラスからしかアクセスできません。つまり、アクセスできるのは、実質的にカスタム・クラス・ローダーだけです。このメソッドは、`findClass()` から返される `Class<?>` オブジェクトを返します。その結果、クラスが正常にロードされることになります。

## まとめ

1つ注意点があります。先ほどの例では、nativeメソッドを持つクラスがロードされないようにしています。しかし、実際の環境では、次のようなケースを考慮する必要もあるでしょう。

- すでにロードされているクラスのネイティブ・メソッドを呼び出すコード（推移的なケース）
- リフレクションによるネイティブ・メソッドへのアクセス
- MethodHandlesインタフェースによるネイティブ・メソッドの呼出し

さらに、ネイティブ・メソッドの中には、ほぼすべてのJavaプログラムを正しく機能させるために不可欠なものもあります (`getClass()` や `Object::hashCode` など)。

ネイティブ・コードの実行を完全に制限するために必要なことについて論じ始めると、本題からかけ離れすぎてしまいますが、現実的には、JDKに含まれるコア・ネイティブ・メソッドのホワイトリストのようなものを使用する必要があります。しかしながら、本記事の例でASMを使って実現したことは記憶にとどめておいてください。Javaの特定リリースのバイトコードを読み取り、デバッグ・データをスキップして、特定のバイトコードを探しました。その前の例では、実行時にバイトコードの変換を行いました。

**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。先日発刊された『Optimizing Java』 (O'Reilly) を含め、プログラミングに関する4冊の書籍を執筆している。



移行に必要なのは、「早とちり」を避けつつ慎重な計画と丹念な実行

本記事では、Java 8 ライブラリをモジュールに移植するために必要なすべての手順を説明します。まずは、ツール、移行時に発生する可能性のある問題、複数の Java バージョンを対象とした並行ビルドの実行について説明します。次に、ライブラリを自動モジュールとして使用するように設定する方法を見ていきます。そして最後に、ライブラリをモジュール化する方法について説明します。本記事の情報を活用すれば、皆さん独自の移行とモジュール化の戦略を立てられます。モジュール・システムの基本、特にモジュール内部の強力なカプセル化についての知識があれば、本記事の内容を理解するうえで非常に役立ちます。しかし、そうでない方でもほとんどの内容は理解できるでしょう。

## 適切なツール

Java 9 のサポートは絶えず改善されています。そのため、ご使用の IDE で Java 9 の機能を最大限に活用するためには、その最新バージョンを使用する必要があります。最新バージョンは好みでないという方は、IntelliJ IDEA 2017.2 または Eclipse Oxygen.1a（これより前のバージョンの Eclipse では、Java 9 サポート・プラグインが必要ですが、現在はほとんど使われていません）以降を使用する必要があります。NetBeans をお使いの方は、バージョン 9.0 以降が必要です。



同様に、ビルド・ツールも最新バージョンを使用してください。Maven の場合は、Maven 自体をバージョン 3.5.0 以降に、コンパイラ・プラグインをバージョン 3.7.0 以降にする必要があります。Gradle の場合は、バージョン 4.2.1 以降を使用します。

## Java 9 への移行における移行の問題点

Java 9 のエコシステムにはモジュールが導入され、それに伴う多くの変更が行われました。変更の中には、モジュールが導入されたことによって生じたものもあります。このような変更によって、Java の動作は微妙に変わりました。互換性がないと見なすことができる変更もいくつか存在します。以下に、移行にあたって想定される問題と、可能な対応策について簡単にまとめます。ただし、これらすべての問題に遭遇する可能性は低いため、あまり心配することはありません。

**JDK の内部 API にアクセスできない:** もっとも遭遇しやすい問題としてはっきりしているのは、内部 API へのアクセスに関するものです。sun.\* パッケージや大半の com.sun.\* パッケージのクラスに依存するコードがある場合、次のようなコンパイル・エラーが発生します。

```
error: package com.sun.java.swing.plaf.nimbus is not visible
import com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel;
      ^
(package com.sun.java.swing.plaf.nimbus is declared
 in module java.desktop, which does not export it)
1 error
```

この問題を修正する適切な方法は、そのようなクラスを使わないことです。しかし、まだそれができないという方は、コマンドライン・フラグ `--add-exports` を使うと、コンパイル時および実行時に上記のクラスを利用できます。実行時にリフレクションを使用して JDK の内部 API にアクセスしている方は、フラグ `--add-opens` および `--illegal-access` を確認してください。

なお、コマンドライン・フラグについて注意しなければならない点があります。ここで説明した問題をフラグの使用により解決する場合、そのライブラリを含むアプリケーションを実行するとき、通常はユーザーが同じフラグを指定する必要があります。これは非常に不便になることがあるため、可能な限り避けた方がよいでしょう。どうしてもフラグを使用する必要がある場合は、ドキュメントにその点を必ず明記してください。

## Java EE モジュールへの依存性を解決できない：これまでの JDK には、JavaBeans Activation Framework (JAF)、JTA、JAXB、JAX-WS、CORBA など、いくつかの Java EE API が含まれていました。しかし、Java SE と Java EE をより明確に分離し、Java EE が Oracle の管理から離れることに備えるため、これらの API は Java 9 で非推奨となり、Java 11 で削除される予定です。

ライブラリが、現在 JDK に存在するこれらの API に依存している場合、Java 9 以降のリリースでコードをビルドした際に、次のようなエラーが発生します。

```
error: package javax.xml.bind is not visible
import javax.xml.bind.JAXBException;
    ^
(package javax.xml.bind is declared in module java.xml.bind,
 which is not in the module graph)
1 error
```

このエラーは、コマンドライン・フラグ `--add-modules` を使用して修正することができます。ただし、この方法を使えるのは Java 11 までです。長期的な解決策となるのは、必要になる API の実装を選び、通常の依存性として追加することです。

## URLClassLoader へのキャストが失敗する

ライブラリの中には、クラス・パスの参照や操作を行うものもあります。たとえば、クラス・パスの内容を調べるものや、JAR ファイルを追加するものです。多くの場合、このようなライブラリは、システム・クラス・ローダーを [URLClassLoader](#) にキャストすることでそれを実現しています。このクラスには、クラス・パスの参照や操作に必要なメソッドがあるからです。しかし、Java 9 ではクラスローディング戦略が変更されており、別のクラス・ローダーが使われるようになっています。そのため、先ほどのキャストは失敗し、次のようなエラーが発生します。

```
Exception in thread "main" java.lang.ClassCastException:
    java.base/jdk.internal.loader.ClassLoaders$AppClassLoader
cannot be cast to java.base/java.net.URLClassLoader
    at monitor.Main.logClassPathContent(Main.java:46)
    at monitor.Main.main(Main.java:28)
```

この解決策は、クラス・ローダーで何をしたいのかによって異なります。

- クラス・パスの内容を調べるだけの場合は、システム・プロパティ `java.class.path` を参照します。
- 実行中のアプリケーションに新しい JAR ファイルを追加する場合は、既存のシステム・クラス・ローダーに委譲する新しいクラス・ローダー（このローダーには [URLClassLoader](#) を使うことができます）を作成する必要があります。
- その他のユースケースについては、`ClassLoader` を確認してください。いくつかのメソッドが追加されているため、[Javadoc](#) を参照すると、必要なものが見つかるでしょう。

**Java のバージョン文字列の形式が新しくなっている**：Java 9 以降では、システム・プロパティ `java.version` と、その兄弟関係にあるプロパティは、`1.x` ではなく `x` で始まります。したがって、Java 9 では `9.9.0.1`、`9.0.4` などが返されます。Java 10 以降も同様です。この文字列を解析するのはうんざりだという方は、Java 9 で新たに導入された型である `java.lang.Runtime.Version` を確認してください。バージョン情報に簡単にアクセスすることができます。

他にも、JDK のフォルダ構造の変更など、Java 9 への移行の際に理論上発生する可能性がある細かな問題はいくつかありますが、ライブラリ開発者がそういった問題に遭遇することはほとんどないでしょう。

**ユーザーがプロジェクトをモジュール化する場合、requires**  
ディレクティブで依存性を宣言する必要があります。その際に、ライブラリも宣言に含めます。

## 複数のバージョンの Java を対象としたビルド

これまで見てきたように、Java 9 では、ライブラリの動作に影響を与えうるいくつかの変更が行われています。そのため、こういった変更に対応するためにコードを更新している間は、Java 9 以降でコンパイルとテストを行う継続的インテグレーション (CI) ビルドが必要になることも納得できます。通常、これを実現するためには、ビルド設定を追加し、既存のビルドと並行して実行されるようにします。

並行 CI ビルドを避けるために、プロジェクトのベースラインを Java 9 または Java 10 に引き上げたいと思う方もいるかもしれません。しかし、多くのサイトはまだ Java 9 以降に移行していないため、少なくとも現時点では、ユーザー・プールが大幅に減少するでしょう。さらに重要なのは、6 か月ごとに新しいリリースが登場するため、今後のリリースのライブラリで、結局は並行ビルドが必要になる可能性が高いことです。さまざまな Java バージョンに向けて複数の CI ビルドを設定しておくことは、十分見返りのある作業です。それでは、その具体的な方法に

ついて見ていきます。

## CI サーバーの構成

一般的にお勧めの方法は、(コンパイルとテストだけでなく) ビルド作業全体を目的のバージョンの Java で行うことです。その場合、最初に行うべき手順は、異なる Java バージョンを使ってビルドを実際に複数回実行するように CI サーバーを構成することです。Jenkins や Travis CI などを使うと、これをかなり簡単に実現できます。

ビルドに時間がかかり、リソースが不足する状況でも、この作業を先延ばしにしないでください。少なくとも毎晩、非ベースライン・バージョンでビルドすることが望ましいでしょう。この手法では、問題のある内容がコミットされ、丸1日発見されないことになる可能性もありますが、遅くとも翌朝には、何かがおかしいことがわかります。

**ビルド手順の構成:** ビルドの実行は簡単ですが、その構成は面倒かもしれません。ビルドを実行する Java のバージョンによっては、コンパイルやテストを行う際にコマンドライン・フラグを適用することが必要な場合もあります。依存性のバージョンの変更や、ビルド手順構成の編集も必要かもしれません。これができるようになるためには、ビルド・ツールで条件付き構成を行うことに慣れておく必要があります。

Maven の場合は、プロファイルを使用します。次のブロックに示すプロファイルは、Java 9 以降でビルドされると、自動的にアクティブ化されます。

```
<profiles>
  <profile>
    <!-- Java 9 での実行の場合、自動的にプロファイルをアクティブ化 -->
    <activation>
      <jdk>[9,)</jdk>
    </activation>
    <!-- ここにバージョン固有のビルド構成を指定 -->
  </profile>
</profiles>
```

上記の <profile> ブロック内には、バージョン固有の構成を指定できます。たとえば、Google Guice が使っているプラットフォーム・モジュールの内部パッケージをエクスポートするためには、次のようにします。

```
<profile>
  <id>java9+</id>
```

```
<activation>
  <jdk>[9,)</jdk>
</activation>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <configuration>
        <!-- 新たな問題を見つけるため、不正なアクセスを拒否 -->
        <!-- Google Guice 4.1 は、
              java.lang.ClassLoader::defineClass の内部メソッドを使用 -->
        <argLine>
          --illegal-access=deny
          --add-opens java.base/java.lang=ALL-UNNAMED
        </argLine>
      </configuration>
    </plugin>
  </plugins>
</build>
</profile>
```

**ビルド・プロセスの構成：**場合によっては、個々のビルド手順だけでなく、ビルド・プロセス全体を構成する必要があります。たとえば、ビルドの際に使用しているプラグインが、Java EE モジュールに依存している場合などです。この場合、ビルド・プロセス自体を Java 9 で動作させるために、コマンドライン・フラグが必要になります。

Maven を使っている場合、プロジェクトのルート・フォルダに .mvn/jvm.config ファイルを作成し、そこに必要なコマンドライン・オプションを含めることで、それが実現します。Maven コマンドが実行されると、コマンドはそのファイルを確認し、Maven を実行するために起動する Java プロセスにオプションを適用します。

問題は、Maven が Java 8 でも同じことを行い、--add-modules のような認識できないコマンドライン・オ



プシヨンのために処理が失敗することです。この問題を解決するためには、構成ファイルに `jvm9.config` という名前を付け、デフォルトでは無視されるようにします。そのうえで、ビルドを開始する前に、すべての構成ファイルの名前を `jvm.config` に変更するスクリプトを CI サーバーから呼び出します。あまり美しい方法ではありませんが、解決策として機能します。

筆者は、Java 9 でビルドを行うまで、ここで紹介した Maven の機能を使ったことはありませんでした。そもそも、Maven の機能を掘り起こすことに熱心だったわけではありません。しかし、少しばかり Maven の知識を深めるには絶好の機会だと考えました。そして、その努力は大いに報われています。

## ライブラリを自動モジュールとして使う

この時点で、プロジェクトは Java 9 でビルドされ、テストされています。ユーザーは喜んでライブラリを使うこと  
でしょう。最初はクラス・パス上での使用かもしれませんが、一歩進め、モジュールとして使いたいとすぐに思う  
ようになるでしょう。これは、どのような仕組みで実現できるのでしょうか。そもそも、まだモジュールを作成し  
ていません。

## 自動モジュールの概要

ユーザーがプロジェクトをモジュール化する場合、requires ディレクティブで依存性を宣言する必要があります。その際に、ライブラリも宣言に含める必要がありますが、そのためには、requires で使用するモジュール名が必要です。さらに、ライブラリをモジュール・パスに配置する必要があります。

Java ランタイムは、モジュール・パス上にモジュール記述子 (module-info.class) を含まない JAR ファイルを見つけた場合、そのファイルに対して、自動モジュールと呼ばれるものを作成します。自動モジュールは、通常のモジュールと似ていますが、すべてのパッケージをエクスポートし、他のすべてのモジュールを読み取ることができる特性を持つことなどのいくつかの前提が適用されます。

さらに興味深いのが、名前に関することです。自動モジュールの名前は、マニフェストの Automatic-Module-Name エントリで定義するのが理想的です。この定義が存在しない場合、モジュール・システムは JAR ファイルの名前からモジュール名を決めます。この代替方法が、どの開発環境でも安定的に使用できるものではないことは明かです。ライブラリのモジュール名が変わった場合（たとえば、モジュール化して適切な名前を付けた場合）には、さらなる問題が発生します。

ただし、ユーザーがアプリケーションを開発しているのであれば、さほどひどいことは起きません。requires

## Java 9がもたらした未来に向かうための最後の手順は、プロジェクトのJARファイルをモジュールJARファイルに変換することです。

ディレクティブをすべて更新し、ライブラリを新しい名前で使用すればよいだけだからです。しかし、ライブラリも開発している場合、はるかにややこしい問題が発生します。それが起きるのは、ユーザーが JAR ファイルをファイル名で使用するプロジェクトをリリースした後、その JAR ファイルのモジュール名が変わった場合です。その場合、新しい JAR ファイルではユーザー側の依存性を満たせなくなります。さらに深刻なのは、推移的依存性によってライブラリへの依存性が 2 重になっており、それぞれが異なる名前で参照されている場合です。このような場合、ユーザーは重大な問題に陥ります。

## 自動モジュール名の定義

したがって、JAR マニフェストに Automatic-Module-Name エントリが設定されていない場合、ユーザーはモジュールとして扱うプレーンな JAR ファイルに依存することに慎重になるでしょう。そこで、ライブラリが Java 9 でうまく動作するようになり、（複数のパッケージを公開している場合に）JAR ファイル間でパッケージを移動させるような大規模なリファクタリングを行う予定がない場合は、モジュール名を決めて（決め方は次のセクションで説明します）マニフェストのエントリを設定します。

Maven JAR プラグインでこれを行うには、次のようにします。

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <configuration>
    <archive>
      <manifestEntries>
        <Automatic-Module-Name>$name</Automatic-Module-Name>
      </manifestEntries>
    </archive>
  </configuration>
</plugin>
```

\$name は、実際の名前で置き換えます。これを行うと、他のモジュールがこの JAR ファイルを自動モジュールとして安定した名前で使用できるようになります。このことと、選択したモジュール名を、ドキュメントに必ず明記してください。

## ライブラリのモジュール化

Java 9 がもたらした未来に向かうための最後の手順は、プロジェクトの JAR ファイルをモジュール JAR ファイルに変換することです。実は、これはとても簡単に行うことができます。プロジェクトのルート・ソース・フォルダにモジュール宣言 (module-info.java ファイル) を作成し、そこにライブラリのモジュール名、依存性、エクスポート、サービスなどを定義します。

```
module $name {
  requires $module;
  exports $package;
  uses $service;
  provides $service with $provider
}
```

各プロパティについては、後ほど見ることにします。その前に、他に必要なこととお話しようと思いますが、実は何も必要ありません。最新のツールの場合、現在作成しているのがモジュールであることを自動的に認識し、適切な対応を行います。IDE やビルド・ツールは、コンパイルするファイルのリストにモジュール宣言を追加し、モジュール・パス上に依存性を準備し、生成された `module-info.class` をビルドする JAR ファイルに含めます。そのため、通常は他に明示的な作業を行う必要はありません。

それでは、ライブラリをモジュール化する方法を詳しく説明します。本記事はモジュール・システムのチュートリアルではないため、モジュール・システムの動作を理解していることを前提とします。[編集注:そうでない方は、こちらの記事をお読みください。]

## モジュール名の決定

パッケージ名と同様に、モジュール名もグローバルで一意であることが必要です。パッケージ名と同じ手法を用いると、これをもっとも簡単に実現できます。すなわち、プロジェクトに関連するドメインを選び、それを逆順にします。モジュール名がパッケージ名の接頭辞と同じになっていれば、適切に処理したということです。

**依存性の宣言:** デフォルトでは、ビルド構成内のすべての非テスト依存性について、モジュール宣言に `requires ディレクティブ` を記述する必要があります。また、オプションの依存性 (`requires static` を使用) や、ライブラリを必要とするモジュールに公開される依存性 (`requires transitive` を使用) を追加することもできる点

に注意してください。

選択肢の1つは、自動モジュールに任せることです。自動モジュールについて説明したときに暗に示していたように、非モジュールJARファイルに依存することは技術的に可能であり、一般的にはそうしても問題はありません。そのため、すべての依存性をモジュールにしなくても、ライブラリをモジュールにすることができます。ただし、自動モジュール名が定義されていない非モジュールJARファイルを使用することは、非常に軽率なことである点を意識してください。前述のように、これにより、ユーザーに重大な問題が発生することになるため、行わないでください。

したがって、ライブラリのモジュール JAR ファイルを公開する場合は、すべての依存性について、モジュールにするか、または少なくともマニフェストのエントリで自動モジュール名を定義してください。実際、ファイル名でモジュールを要求している場合、Maven はそのアーティファクトを公開しないように明示的に警告します。しかし、さらに一歩進んで、推移的な依存性もすべて確認することを強くお勧めします。そのいずれも、ファイル名でモジュールを要求すべきではありません。

## Split Package の処理:

さらに考慮する点は、Split Package をどのように処理するかです。依存性をモジュールとして扱い始めると、依存性はモジュール・システムによる細かいチェックを受けることになります。そして、このチェックは失敗する可能性も

あります。その例の1つが、JAR ファイルの Split Package です。これは、異なる2つの JAR それぞれで、同じパッケージで定義される型がある状況を指します。たとえば `org.lib` というパッケージがあり `foo.jar` に `org.lib.Foo` が、`bar.jar` に `org.lib.Bar` が定義されている場合です。モジュール・システムは、これを許可していないため、この2つの JAR がモジュール・パス上に存在する場合、エラーが発生します。

このような構成で動作させる方法はいくつかあります（必要な場合は、`-patch-module` を参照してください）。しかし、どの方法をとるにせよ、ユーザーはビルド・ツールの広範囲にわたる構成を繰り返す必要があります。そのような作業をユーザーに強いることはお勧めできません。そのため、依存性で Split Package が発生している場合は、プロジェクトをモジュールとして公開する前に修正してください。

## パッケージのエクスポート

モジュールの命名と依存性の宣言を行った後は、パブリック API を定義します。これは技術的に見れば簡単なことで、アクセスを許可する `public` 型を含むすべてのパッケージ名を `exports` ディレクティブに列挙するだけです。しかし、悪魔は細部に宿るものです。

自動モジュール名が定義されていない  
非モジュールJARファイルを使用するこ  
とは、**非常に軽率なことである点を意  
識してください。**

モジュール・システムが導入される前は、公開する必要があるクラスを明示的に指定することはできませんでした。しかし、パブリック API について考慮しなくてよいということではなく、どのクラスがサポートされているかは、コメントを追加するか、ドキュメントに記述するという方法で示す必要がありました。つまり、パッケージの public クラスには、サポートされているものも、サポートされていないものも含まれている可能性がありました。しかし、新しくモジュール・システムが導入されたことで、その可能性はなくなりました。

パッケージをエクスポートするということは、その中のすべての public 型にアクセスできるだけでなく、そのすべてがサポートされているものと明示することでもあります。クラスをサポートしたくない場合は、そのクラスを public 以外にするか、エクスポートしないパッケージに含める必要があります。

つまり、パブリック API を定義する際には、サポートしないクラスについて再編成を行わなければならない場合があります。もっとも一般的な解決策は、サポートしないクラスを別のパッケージに移動し、そのパッケージをエクスポートしないようにすることです。

**サービスの使用と提供:** サービスの提供または使用を通して他の JAR ファイルと連携するライブラリを作る場合、次のようにしてモジュール宣言にその情報を含める必要があります。

- 使用するすべてのサービス（つまり、`ServiceLoader::load` を呼び出すすべての型）について、宣言に `uses` デイレクティブを追加する必要があります。
- `META-INF/services` にあるファイルのすべてのクラス名について、宣言に `provides $file-name with $class-name` という行を追加する必要があります。

なお、ユーザーは、モジュール JAR ファイルをモジュールとして使用しなくても構いません。モジュールとして使用しない場合、モジュール宣言は無視されます。つまり、META-INF/services にあるファイルは、モジュール宣言と同期させておく必要があります。

## 複数モジュールの提供:

ライブラリを複数の JAR ファイルとして提供する場合について、細かい点を 2 点お伝えしたいと思います。1 点目は、前述のように、Split Package は問題を引き起こすということです。したがって、パッケージが JAR ファイルによって分割されないようにしてください。現在そうになっている場合は、ライブラリをモジュール化する前にクラスの移動や、パッケージの名前変更を行う必要があります。

2 点目のポイントは、クラスの移動に関することです。クラス・パスを使う場合、クラスがどの JAR ファイルに存在するかは問題ではありませんでした。いずれかの JAR ファイルに必要なクラスが含まれていれば、アプリケーションは正常に動作しました。そのため、下流のコードを壊さずに JAR ファイル間でクラスを移動することもできました（慎重に行えば、ということです）。モジュールでは、それがはるかに難しくなります。あるモジュール

が、ライブラリ・モジュールのいずれかに格納されている型にアクセスするためには、そのモジュールを読み込む必要があります。そのためには、通常、宣言においてモジュールを要求する必要があります。アプリケーションが複数のライブラリ・モジュールのうち1つだけを必要とする場合、そのモジュールからクラスを別のライブラリ・モジュールに移動すると、アプリケーションはそのクラスにアクセスすることができなくなり、コンパイルや実行は失敗します。

したがって、モジュールJAR ファイル間で型を移動する場合、プレーンな JAR ファイル間での移動よりも互換性の問題を引き起こす危険性が高くなります。そのような移動を行う必要があるときは、十分注意してください。

## まとめ

Java 8 から、Java9 以降のリリースに対応した完全なモジュールに移行することは、かなり大がかりな作業になる場合があります。しかし、ほとんどのプロジェクトでは、そこまで大変なことではありません。一般的に、ライブラリが小さく、含まれる Java コードがプレーンであるほど、必要な労力は少なくなります。

</article>

**Nicolai Parlog** (@nipafx) : ソフトウェア開発をこよなく愛し、常にソフトウェア開発について読み、考え、書いている。コーディングは生計を立てる手段であるとともに趣味でもある。先日出版された本『The Java Module System』(Manning) の著者。codefx.org でソフトウェア開発についてのブログを書き、複数のオープン・ソース・プロジェクトに長期間にわたって貢献している。

## JEP 296: 単一リポジトリへのJDKソースの統合

Java 10 で実装されたこの提案から、JDK の特別な要件を備えたコードベースをメンテナンスするうえで生じているいくつかの困難がうかがえます。具体的に言えば、Java 拡張提案 (JEP) 296 で対処するのは、多くのコードのコミットがソース・ベース内の複数のリポジトリに及び、その結果、ソース・コード・マネージャに対してアトミックなコミットを行うことができないという問題です (コミットに関連する場合、アトミックという用語は、アクションが1回の操作だけですべて実行されるか、まったく実行されないかのいずれかであることを意味します)。完了するために、複数のリポジトリに別々にコミットしなければならない場合、明らかにアトミックではありません。これは、毎回のコミット作業を終えるまでにかなりの労力が必要となり、問題も起きやすいことを意味します。

JDK には 8 つの主要なリポジトリがあり、およそ 1,100 の欠陥やリクエストは、複数のリポジトリにまたがるものです。そこで、これらのリポジトリを単一のリポジトリにマージすることがこの JEP で提案されました。

大規模なコードベースを抱える企業は、一元化された単一リポジトリのメリットをますます強く実感しています。たとえば、Google や Facebook は、すべてのプロダクトのコード管理に単一のモノリシック・リポジトリを使用していることを頻繁に発信しています。20 億行超のコードを含む Google リポジトリは、バージョンの統一、コード共有の簡略化、再利用性の向上など、さまざまな利点を生んでいると言われています。

この JEP によって、内部開発手法や DevOps に変更が発生することは避けられません。これらの変更が Java 開発者にとって大きな価値があるものであれば、次号以降で紹介したいと思います。







## 巨大なswitch文や大量のif文を使わず、状態遷移をエレガントに管理

ステート・マシンの概念から生まれたのが、Stateパターンです。名前もステート・マシンに由来しています。Stateパターンでは、オブジェクトは一定数の状態のうち、厳密に1つの状態をとることができます。状態が遷移すると、オブジェクトの動作が変化します。

## アプリケーションの状態



```
StateName currentStateName;
```

次に、下記のコードのようなロジックを使用し、4つのメソッドをそれぞれ定義します。ほとんどの機能はいずれかの状態でしか動作しないため、オブジェクトの状態を確認する必要があります。そのため、それぞれのメソッドで、大量のif文や巨大なswitch文を使うことになります。

```
public void start() {
    if (currentStateName == StateName.STOPPED) {
        currentStateName = StateName.PLAYING;
        startPlay();
    } else if (currentStateName == StateName.PAUSED) {
        currentStateName = StateName.PLAYING;
        resumePlay();
    } else if (currentStateName == StateName.PLAYING) {
        System.out.println("Already playing!");
    } else if (currentStateName == StateName.REWINDING) {
        System.out.println("Wait a while, OK?");
    }
}
```

さらに、4つのメソッドそれぞれに同じ量の条件分岐コードが必要になります。この点は、機能の追加や変更が必要になった際に、保守上の重大な問題につながります。実際、数か月間プロジェクトに参加して機能追加が不要なことなどないでしょう。機能追加の要求がないということは、おそらくユーザーがいないということです。

Stateパターンを使えば、コードを美しく整理できます。先ほどのアプローチを、Stateパターンを使ってリファクタリングしてみます。まず、**State**というインタフェースを作成し、プレーヤがとることができる状態について1つずつ、合計4つのインスタンスを作成します。最初にインタフェースを定義します。

```
interface State {
    void stop();
    void start();
    void pause();
}
```

```

void rewind();
default enterState() {
    // これが必要になるのは、一部の状態のみ
}
}

```

図2に、今回の例の汎用クラス図を示します。process()メソッドは、stop()、start()など、さまざまな処理を行うメソッドに対応するものです。通常は、enterState()などのメソッドを作っておき、そのメソッド内で呼び出して、処理を適切に設定しますが、そうしなければならないわけではありません。

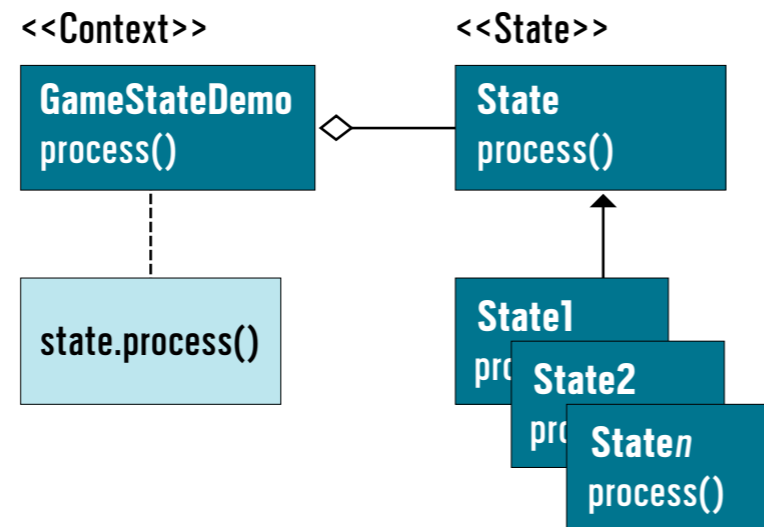


図2: State/パターンの典型的なクラス図

Stateクラスを使うクラスをContextクラスと呼びます。多くの場合、StateクラスはContextクラスのインナー・クラスとして記述されます（そうしなければならないわけではありません）。Contextクラスという用語は、以降の説明で重要な概念となりますので注意してください。

（なお、本記事のすべてのコードは、[筆者のGitHubリポジトリ](#)で公開されています。ここで説明しているのは、PlayerStateDemo.javaファイルのコードです）

Stateを定義したら、それをサブクラス化して4つの状態を定義します。アプリケーションで必要になるのは、1つのStateクラスにつき1つのインスタンスだけです。そのため、匿名クラスとしてインスタンスを作成し、そのクラス

をフィールドに保存します。

ただし、プログラムの状態がまれにしか変わらない場合は、`State`のサブクラスを名前付きクラスとして作成し、コンストラクタで設定を行っておき、実際に切り替えを行うときまで`State`サブクラスのインスタンス化を遅らせることもできます。

以下に、「停止」状態を表すStateサブクラスのコードを示します。このサブクラスでは、プレーヤが停止状態にあるときに各操作が呼び出された場合、どのように実行されるのかについて定義しています。この例のstop()は、低レベルのハードウェア制御メソッドであり、DVDプレーヤのモータを停止させるものかもしれません。setIcon()を呼び出している部分はコメント・アウトされていますが、GUIの更新を表しています。

```
State stoppedState = new State() {
    @Override
    public void enterState() {
        stop();
        // setIcon(icon.stopped);
    }

    @Override
    public void stop() {
        // すでに停止されているため、何もしない
    }

    @Override
    public void start() {
        currentState = playingState;
        currentState.enterState();
    }

    @Override
    public void pause() {
```

```
//すでに停止されているため、何もしない
}  
  
@Override  
public void rewind() {  
    currentState = rewindingState;  
    currentState.enterState();  
}  
};
```

`rewind()`の最後にある2つの文は、`Context`オブジェクトの「クラスが変化しているように見える」仕組みを実現する方法を示しています。`stop()`や`start()`など、メイン・プログラムのメソッドをみてみます。

```
PlayerStateDemo context = new PlayerStateDemo();
System.out.println("Initial state: " + context.getState());
// ユーザーが再生ボタンをクリック
context.start();
System.out.println("Current state: " + context.getState());
// ユーザーが停止ボタンをクリック
context.stop();
System.out.println("Current state: " + context.getState());
```

これらのメソッドは、現在の`State`オブジェクトのメソッドを呼び出しています（すなわち、「委譲」しています）。`currentState`変数に新しい値を代入するだけで、現在の`State`オブジェクトに基づいて、同じメソッド呼出しから異なるコードが呼び出されるため、プログラムがクラスを変化させているように見えます。たとえば、再生中に`stop()`を呼び出すと、プレーヤは停止しますが、同じメソッドをもう一度呼び出しても何も実行されません。最初の呼出しでは`playingState`の`stop()`メソッドが呼び出されますが、2回目の呼出しでは`stoppedState`の`stop()`メソッドが呼び出されるためです。

この例の場合、無効であっても害のない操作は、何もせず無視しています。実際のメディア・プレーヤでは、そのような動作になっているからです。もちろん、ログへの記録や例外のスローなどによって、無効な使い方であると

通知することが理にかなっている場合もあります。

状態を表す変数は、決してNULLにならないようにする必要があります。今回の例では、フィールドを初期化する際に確実に初期値を設定しています。

```
State currentState = stoppedState;
```

このようなやり方は、前述のif文の迷路よりもコードが増えているように見えるかもしれませんが（実際に行数で数えてみると、わずかに長くなっています）。しかし、保守の容易さという点では、それだけの価値があります。何がどこに向かうのかがより明確にわかるようになり、それと同時に重要なのは、特定の状態に必要なメソッドを書き忘れてしまっても、コンパイラがそのことを教えてくれるという点です。これは、すべてのメソッドを実装しなければならないというインタフェース要件があるためです。

このパターンの実装方法には、いくつかのバリエーションが考えられます。アプリケーションのスコープによっては、インタフェースを **Context** クラスの外に出し、**Context** クラス自体もインタフェースを実装するとよいかもしれません。そうすると、すべての委譲メソッドが存在すること、引数と戻りタイプが正しいことをコンパイラがチェックしてくれます。**State** クラス自体がインナー・クラスである必要はまったくありません。特に問題がなければ、パッケージ・レベルの可視性を持つ個別クラスにすることもできます。その場合でも、**Context** クラスは状態オブジェクトを作成できます。その際、**Context** はコンストラクタに自身への参照を渡すのが一般的です。たとえば、次のようにします。

**Stateパターンを使うと、状態に固有な動作が整理されるため、可読性可読性が高く保守の容易なコードが実現し、新しい状態の追加も簡単になります。**

```
State shuttingDownState = new ShuttingDownState(this);
```

このようにする理由は、`State`クラスが`Context`クラスのpublicでないメソッドやフィールドにアクセスする必要がある可能性が高いためです。

Contextクラスを使用するクライアント・コードは、実際のStateオブジェクトにアクセスする必要はありません。確実にカプセル化を行えるように、アクセス権を与えないようにするのが一般的です。必要に応じ



成しました。ただし、実際の部屋の描写は含まれていません。このプログラムは単なるデモです。ゲームは実際にプレイできますが、退屈するまでの平均時間は約4.2秒です。次に示すのは、部屋に関するアクションを処理する `State` クラスです。

```
abstract class State {
    public abstract void lookAround();
    public abstract void goInside();
    public abstract void goOutside();
    public void quitGame() {
        display("Goodbye!");
        System.exit(0);
    }
}
```

このシンプルなStateには、アクティブ化メソッドはありません。しかし、quitGame()が存在します。簡単なゲームでは、ユーザーがどの状態にあってもゲームを終了できるようにして問題ないでしょう。そのため、デフォルトのquitGame()メソッドでそれを行えるようにしています。状態によってこの動作をオーバーライドすることもできます。たとえば、ユーザーが重要なアイテムを持っているかどうかや、ゲームの状態を保存するかどうかをユーザーに確認するメッセージを表示するといったことも可能です。

ゲームの状態には、inRoom、inHallwayなどがあります。inHallway状態は、次のようになっています。

```
public State inHallwayState = new State() {
    public void lookAround() {
        display("You are in a hallway. There is a door here");
    }
    public void goInside() {
        display("You are in a room");
        state = inRoomState;
    }
    public void goOutside() {
```

他の状態を見たい方は、オンラインのコードベースをご覧ください。

Stateパターンは優れたパターンです。状態に固有な動作が整理されるため、メソッド内に長い条件文を書いた場合に比べて、可読性が高く保守の容易なコードが実現します。また、新しい状態の追加も簡単になります。さらに、状態遷移がContextクラスと各Stateクラスの内部で明示的に行われるようになり、外部からはまったく見えないようにすることができます。状態の遷移はアトミックでもあります。なぜなら、遷移時に変更されるのは、Contextクラスの1つの変数（通常はcurrentStateのような名前が付けられます）だけだからです。

複数のメソッド内に似たようなif文を書いていることに気づいたときや、状態によってオブジェクトの動作を大幅に変えなければならないときは、ぜひStateパターンを試してみてください。Stateパターンがもたらす可読性と保守性の向上を、きっと気に入っていただけると思います。





SIMON ROBERTS



MIKALAI ZAIKIN

# 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。

## 解答1

79ページ

**設問 1 (中級者向け)** . 次のコードについて:

```
public class Calculator {  
    public static void main(String[] args) {  
        int i = 0;  
        Calculator c = new Calculator();  
        System.out.print(i++ + c.operation(i));  
        System.out.println(i);  
    }  
  
    public int operation(int i) {  
        System.out.print(i++);  
        return i;  
    }  
}
```



**A.** 121

**B.** 123

**C.** 234

**D.** 345

**E.** 3桁の数字になるが、厳密な値はプラットフォームおよび実装に依存する

81ページ

```
public class BadCalculator {
    public static void main(String[] args) {
        try {
            new BadCalculator().divide();
        } catch (Error err) {
            System.out.println("main catch");
        }
    }
}
```

```
void divide() throws Error {
    try {
        int i = 1 / 0;
    } catch (RuntimeException re) {
        System.out.println("catch");
        throw new CustomException();
    } finally {
        System.out.println("finally");
    }
}
```

//fix this /

```
}  
}
```

どのような結果になりますか。1つ選んでください。

- A. catch  
finally  
main catch
- B. catch  
finally  
例外のスタック・トレースが続く
- C. catch  
finally
- D. コンパイルが失敗する

### 解答3

83ページ

設問 3 (上級者向け) . 正しい文はどれですか。2つ選んでください。

- A. 匿名クラスのベースタイプには、抽象クラスを指定できる。
- B. 匿名クラスのベースタイプには、インタフェースを指定できる。
- C. 匿名クラスのベースタイプには、抽象クラスとインタフェースの両方を指定できる。
- D. 匿名クラスは、常にラムダ式で置き換えることができる。
- E. 匿名クラスの親の型には、引数のないコンストラクタが必要である。



**解答4** **設問 4（上級者向け）** . 次のコードについて:

```
public class Car {
    int speed;
    public Car () { speed = 90; }
    abstract void accelerate(int deltaSpeed);
}
```

さらに、次のコードについて:

```
public class RacingCar extends Car {
    public RacingCar() { speed = 180; }
    public void accelerate(int deltaSpeed) {
        speed += deltaSpeed;
        System.out.println("The new speed is : " + speed);
    }
}
```

さらに、次のコードについて:

```
Car c = new RacingCar(); // line n2
c.accelerate(50);        // line n3
```

どのような結果になりますか。1つ選んでください。

- A.** Car クラスは、コンパイルに失敗する。
- B.** line n2 は、コンパイルに失敗する。
- C.** line n3 は、コンパイルに失敗する。
- D.** The new speed is :140 が出力される。
- E.** The new speed is :230 が出力される。

**設問1**

76ページ

**解答 1:** 選択肢 A が正解です。この設問は、式の評価順序と、メソッド呼出しの際に値がどのように渡されるかについて問うものです。

Java 言語仕様のセクション 15.7 には、「Java プログラミング言語は、演算子のオペランドが左から右という特定の評価順で評価されているように見えることを保証する」とあります。つまり、計算結果は予測可能でなければならないため、選択肢 E は誤りです。

評価順序が左から右と指定されているため、どのような動作になるのかを判断できます。最初に `print` を呼び出している部分では、引数の式は `i++ + c.operation(i)` となっています。

左から右というルールから、このコードではまず `i++` を評価し、次に `c.operation(i)` を呼び出さなければならないことがわかります。この行の前の時点での `i` の値は 0 (2 行前で明示的な初期化を行っているためです) であるため、`operation` メソッドを呼び出す際に渡される値は 1 になります。また、後置インクリメント式の値は変数をインクリメントする前の値であるため、その後の計算に使う `i++` の値は 0 になります。

最初に実行され、その結果最初に出力される `print` は、コードベースにおける実際の出現順序では最後、すなわち `operation()` メソッドの中にあるものです。先ほど、この引数の値が 1 になることを確認しました。`operation` メソッド内の `print` 文は、もう 1 つの後置インクリメント式である `i++` の値を出力します。したがって、最初に出力される数字は 1 になります。この点から、選択肢 C および D は誤りであることがわかります。

次に、`c.operation` はコール元に値 2 を返します。この時点で、`i` の値がメソッド呼出しの内部でインクリメントされているからです。そして、`main` メソッドでは、値 2 が `i` の元の値に加算されます。この元の値は 0 でした。`i++` では、インクリメントされた値ではなく、値 0 を使用する点に注意してください。したがって、次の数字は 2 になります。残念ながら、ここで他の選択肢を排除することはできません。正解が何であるかを判断するためには、最後の `println` に進む必要があります。



3桁目を出力するのは、`main` メソッドの最後の文である `println` です。これは、演算 `i++` を行った後の `i` の値、すなわち 1 となります。Java は、値渡しによるメソッド呼出しモデルを使っているため、`c.operation` の呼出しはこの値に影響しなかったということがその理由です。具体的に説明します。`c.operation` 呼出しの中の `i` は、コール元の同名の変数とは別物であるため、`operation` メソッド内で実行されたインクリメントは、現在注目している値には影響しません。

したがって、最終的に出力される値も1になります。これは、`i++ + c.operation(i)` の左側にある `i++` で0をインクリメントした結果です。

以上より、出力をすべて合わせると 121 となることがわかります。つまり、選択肢 B は誤りで、選択肢 A が正解となります。

## 設問2

77ページ

**解答 2:** 選択肢 D が正解です。通常、試験問題の制作者は、選択肢 D に示されているような、何の条件も付けずに「コンパイルに失敗する」と書く設問を作成しないようにしています。セミコロンを1つ付け忘れているかということまで受験者に徹底的に確認させるのは、限度を超えたものだと見なされているからです（可能性は非常に低いものの、そのようなささいな構文エラーが実際の試験問題に紛れ込み、「コンパイルに失敗する」という選択肢が意図せず正解になる可能性も絶対にないとは言いません）。実際の試験では、選択肢 D のような解答には特定の行が指定される可能性が高いでしょう。しかし、皆さんに頭の体操をしてもらうため、少しだけずるをして、このような設問をあえて用意しました。ただし、「コンパイル失敗」は、一般的に設問の正当な根拠となる点は変わりません。

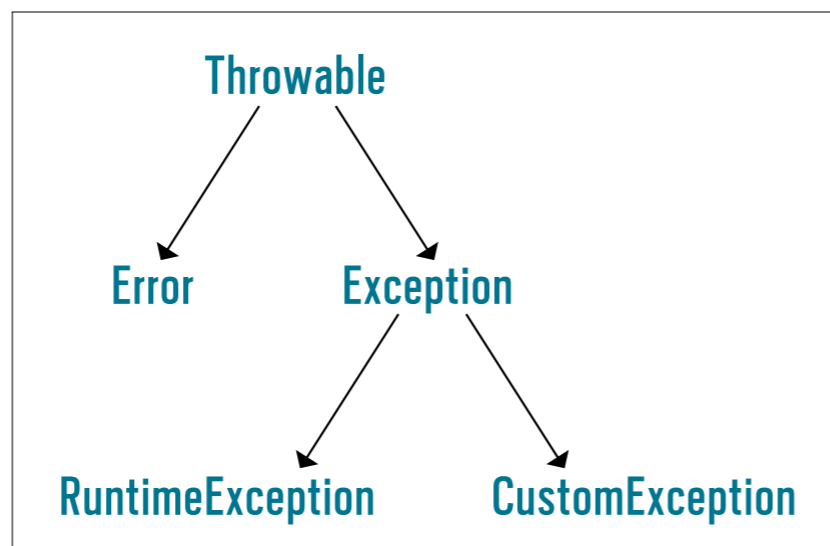
長年、プログラマーは「ハッピー・パス」だけ処理するコードを書き、失敗する可能性のある部分を無視してきたため、**困難に陥ってきました。**

この設問で問われているのは、チェック例外に関する Java のルールと、そのルールの中での実行時例外、通常のチェック例外、エラーの区別です。

長年、プログラマーは「ハッピー・パス」だけ処理するコードを書き、失敗する可能性のある部分を無視してきたため、**困難に陥って**きました。

この背景として、長年、プログラマーは「ハッピー・パス」だけ処理するコードを書き、失敗する可能性のある部分を無視してきたため、困難に陥ってきたことが挙げられます。そして後から、どういうわけかコードがクラッシュしたとユーザーに苦情を言われるのです。多くの場合、調査すると環境的な問題（ファイルが見つからない、ネットワーク・ケーブルが抜けているなど）が見つかります。そのような状況でどうするかということについて、プログラマーは何も考えていなかったのです。

Java には、「宣言または処理」というルールがあります。このルールがあることで、プログラマーが、隠れている不愉快な現実を避けて通るのはかなり難しくなっています。そして通常は、コードを最初に作成するときに、エラー・パスで何かをしなければならないと考えるようになります。なお、ここではその理念を過度に簡素化しています。その主な理由は、例外を使って優れた設計を行う方法について、何も説明していないからです（また、チェック例外は、悪い設計を改善するという面では、何の役割も果たさないという理由もあります）。それでも、チェック例外というルールはこれまで、開発者にとって非常に優れた考え方ではないかと思われています。



**図1: クラス階層**

理念や背景はさておき、Java の「宣言または処理」ルールの本質は、ソース・コードで `RuntimeException` でも `Error` でもない（当然ながら、この 2 つのサブクラスでもない）例外が発生する可能性がある場合、その問題について何かをしなければならないということです。「何かをしなければならない」の具体的な選択肢には、例外を捕捉することと、メソッドが例外をスローすると宣言することがあります。そこから、0 で除算されたときに発生する `ArithmeticException` の問題に対処するために、`catch` というメッセージを出力する `catch` ブロック（`ArithmeticException` は `RuntimeException` であるためです）が呼び出されることがわかります。

しかし、ややこしくなるのはここからです。この `catch` ブロックはその後、`CustomException` をスローしようとしています。`CustomException` を処理する `catch` ブロックがない（そもそも、`try` ブロックの中でスローされていません）ため、`CustomException` はメソッド外に伝播します。`CustomException` は `Exception` のサブクラスであり、`Error` のサブクラスではありません。この2つのクラスは、図 1 に示すように、クラス階層内で兄弟関係にあります。

divide メソッドは、CustomException ではなく Error をスローすると宣言しています。また、CustomException はチェック例外であるため、「宣言または処理」ルールに違反しており、コードはコンパイルできません。以上より、正解は選択肢 D となり、選択肢 A、B、C はすべて誤りです。コードがコンパイルできなければ、何も出力されることはないからです。

この問題を解消するためには、`Error` と記述されている 2 箇所 (`divide` メソッドの `throws` 句と、`main` メソッドの `catch` ブロック) を `CustomException` に変更することが考えられます。現在のコードでは、実際に `Error` がスローされることはありません。そもそも `Error` はチェック例外でないため、宣言も処理も必要ありません。以上の変更を行った場合、選択肢 A に示されている内容が出力されます。

```
catch
finally
main catch
```

### 設問3

78ページ

## 解答 3:

正解は選択肢 A と選択肢 B です。匿名クラスとは、プログラマーが名前を指定しないクラスです。つまり、「この親子関係とこの特定の機能を持つオブジェクト」と言えさえすれば、名前を付けなくてもよい状況だということです（「特定の機能」というのは、通常はメソッドのオーバーラ

匿名クラスには独自の名前がないため、工務店に大量の図面を渡して「これを作ってください」と頼む方法に似ています。完成品の名前を指定する必要はなく、「仕様に従って」作成されることになります。この場合、当然ながら、宣言とインスタンス化を同時に行わなければなりません。この、名前のないクラスは、クラスの仕様

**ラムダ式には匿名クラスには適用されない別の制限もあります。**匿名クラスは複数のメソッドをオーバーライド（または実装）できますが、ラムダ式で指定できるのは、1つのメソッドの動作だけです。

を記述したコードを使わない限り、いかなる方法でも参照できないのは明らかなです。

匿名クラスは Java 1.1 で導入されました。匿名クラスは、オブジェクトの作成を簡略化するための最初の構文サポートであり、いわゆる「構文の土台」ではなく、「重要」だと見なされるもの、すなわち、通常はオブジェクトが何をするかということを重点的に記述するコードを書けるようにするための仕組みです。その際に重要なポイントとなるのは、親クラスまたはインタフェース、それを一般化して記述する方法、そして独特な実装方法です。

匿名クラスを使用することによるもう1つの影響は、クラスの名前空間が簡潔になることです。結果的に、ドキュメント空間も簡潔になります。ある意味、「何と呼ばれるかは気にしません。書かれているとおりのことをするだけです」というスタンスです。

匿名クラスの構文では、クラス（抽象クラスまたは具象クラス）から派生するオブジェクト、またはインタフェースを実装するオブジェクトを作成できます。この特性から、選択肢 A と選択肢 B はいずれも正解です。

クラスを拡張しているか、インタフェースを実装しているかにかかわらず、親が `BaseType` という名前の場合、構文は次のようになります。

```
BaseType anon = new BaseType() {
    // ここに実装に固有のコードを記述。通常は
    // メソッドのオーバーライド
}
```

この構文では、通常のクラスを宣言する際に使われることがある、より一般的な implements 句とは異なり、1つのベースタイプしか指定できません。よって、選択肢 C は誤りです。

匿名クラスが有用な目的を果たすためには、その本体に何らかのコードが記述されている必要があります。`BaseType` がインタフェースである場合はおそらく、実装されるべき抽象メソッド（複数のこともあります）が宣言されています。`BaseType` がクラスの場合、何らかの動作を変更する必要があります。そうでなければ、`BaseType` を直接インスタンス化すればよいだけだからです。

Javaでは、初期化するクラスの名前を使ってコンストラクタを定義します。その名前が存在しない場合、明示的なコンストラクタを使えないのは明らかです。ただし、引数が `BaseType` の既存のコンストラクタと一致する場合は、匿名形式であっても `new BaseType(1,2)` のようにパラメータを指定してオブジェクトを作成できます（当然ですが、インタフェースはコンストラクタを持たないため、この方法は `BaseType` がクラスでなければ使用できません）。

前述のように、匿名クラスの形式では、あるクラス（具象クラスか抽象クラスかは問いません）またはインタフェースをベースとしたオブジェクトを作成できます。一方、ラムダ式はインタフェースを実装する場合にのみ使うことができます。そのため、匿名クラスにはでき、ラムダ式にはできないことがあります。この点から、選択肢Dは誤りであることがわかります。

実際、ラムダ式には、匿名クラスには適用されない別の制限もあります。匿名クラスは複数のメソッドをオーバーライド（または実装）できますが、ラムダ式で指定できるのは、1つのメソッドの動作だけです。さらに、そのメソッドは、実装されるインタフェースに存在する1つの抽象メソッドでなければなりません。以上のことを考えると、ラムダ構文が登場した後でも、匿名クラスには、依然として Java 言語における存在意義があることは明らかです。

先ほどの例の構文では、引数のないコンストラクタを使ってオブジェクトを作成しています。この場合、親クラスの `BaseClass` には引数のないコンストラクタが存在しなければなりません（他のコンストラクタがあっても構いません）。しかし、より一般的には、拡張される側の型のコンストラクタと一致すれば、引数を使うこともできます。したがって、選択肢 E は誤りです。

**設問4** **解答 4:** 選択肢 A が正解です。この設問は、抽象クラスとメソッドについて問うものです。class キーワードの前に abstract キーワードを追加すると、抽象クラスを宣言することができます。次に例を示します。

```
public abstract class MyClass { ...
```

Java 言語仕様のセクション 8.1.1.1 には、抽象クラスについて書かれています。クラスを抽象クラスとして宣言すると、主に 2 つのことが起こります。1 つ目として、そのクラスのインスタンスを作成することができなくなります。サブクラスを定義することはでき、そのサブクラスが抽象クラスでなければ、インスタンス化することができます。しかし、抽象クラスそのもののインスタンスを作成することはできません。

2つ目の変化は、抽象クラスでは抽象メソッドを宣言できるようになることです。具象クラス（非抽象クラス）では、抽象メソッドを宣言することはできません。この設問で問われているのは、まさにこの点です。Car クラスは抽象クラスではないため、抽象メソッドを含めることはできません。そのため、選択肢 A が正解で、選択肢 B、C、D、E は誤りです。

それでは、Car クラスが抽象クラスとして宣言されていた場合はどうなったでしょうか。その場合、Car クラス

は正常にコンパイルできます。この時点で、新しく考えなければならないことが出てきます。まず、`RacingCar` のインスタンスを作成することはできるでしょうか。当然、これは可能です。この時点で、`RacingCar` は問題のない、完全な具象クラスです。具象クラスに抽象メソッドを含めることはできませんが、`RacingCar` で定義されている `accelerate` メソッドは、`Car` で宣言されている同じシグネチャの抽象メソッドを正

しく実装しています。また、実装は public ですが、抽象メソッドはアクセス可能性がデフォルトになっています。しかし、これは問題ありません。オーバーライドまたは実装を行う側のメソッドは、オーバーライドされるメソッドよりもアクセス可能性を狭めてはなりません。アクセス可能性を広げる分には問題はありません。すなわち、`RacingCar` に抽象メソッドはないため、line n2 には何の問題もありません。

line n3 では、変数 `c` の型として、ベースとなる抽象クラスの `Car` が使われていますが、これもまったく問題ありません。抽象クラスをインスタンス化することはできませんが、抽象クラスまたはインタフェースを変数の型として使用することはでき、サブクラスまたは実装から作成したオブジェクトをその変数に割り当てることができます。抽象クラスが存在する理由の1つとして、このような一般化が挙げられます。一般化（抽象クラスまたはインタフェース）に着目してコードを記述すると、どのような特殊化（すなわち、具象サブクラスまたはインタフェースの実装から作成されたオブジェクトによるもの）が行われても、正しく動作させることができます。この時点で、line n2 および line n3 ではコンパイルの問題は起きないことがわかります。

それでは、抽象クラス `Car` を使ったコードが正しくコンパイルできた場合、どのような動作になるでしょうか。これに答えるためには、オブジェクトの作成プロセスがどのようなものであるかについて考える必要があります。ただし、その仕組みはかなり簡単なものです。コンストラクタは、クラス階層の上 (Object) から下に向かって実行されます。つまり `RacingCar` コンストラクタは `Car` コンストラクタの後に実行されることになります。そのため、`speed` の初期値は 90 ではなく 180 になります (おそらく、単位はキロメートル毎時でしょう)。 `speed` の初期値が 180 の場合、 `accelerate` メソッドの呼出し後に出力される `speed` は 230 になります。

興味深いと思っていただけるかもしれない余談として、試験での設問の一般的な表現についてお話しします。

**抽象クラスをインスタンス化することはできませんが、抽象クラスまたはインタフェースを変数の型として使用することはでき、サブクラスまたは実装から作成したオブジェクトをその変数に割り当てることができます。**

**Mikalai Zaikin** : ベラルーシのミンスクを拠点とする IBA IT Park のリード Java 開発者。Oracle による Java 認定試験の作成に携わるとともに、複数の Java 認定教科書のテクニカル・レビューを行っている。Kathy Sierra 氏と Bert Bates 氏による有名な学習ガイド『Sun Certified Programmer for Java』では、3 版にわたってテクニカル・レビューを務めた。

現在のところ、JEP 335は提案にすぎません。提案者らによると、この提案を行った理由の中には、Oracle以外の開発者がNashornの開発に興味があるかどうかを確認したいという点もあるということです。いずれにしても、現在Nashornを使って動作しているコードは、当面の間、そのまま動作することになるでしょう。JEP 335の状況については、コメントが集まってきた段階で改めてお知らせしたいと思います。