JavaプラットフォームとJavaエコシステムは、日々進化しています。プログラミングやハードウェア・アーキテクチャの最新動向とJava SE 7のアプローチをご覧ください。

ORACLE.COM/JAVAMAGAZINE

ORACLE

//目次 /

表紙画像：I-Hua Chen

Java Magazine創刊号をご覧いただきありがとうございます。今も拡大し続けている大規模なJavaエコシステムにおいて重要な役割を担う存在となるよう努めてまいりたいと思います。

*Java Magazine*のキャッチフレーズ「By and for the Java community（JavaコミュニティによるJavaコミュニティのための雑誌）」は、まさにこの雑誌の本質を表現した言葉といえます。「『For』すなわちJavaコミュニティのための」の部分で対象としているのは、Javaエコシステム全体にわたるさまざまな読者の皆様です。Java言語に息吹を与える現場のエンジニアから、テクノロジー・プラットフォーム戦略において重大な意思決定の役割を担う方々、そして、Javaが優れている理由を理解し始めた入門者や初心者まで、Javaにかかわるあらゆる人にとって有益な情報を発信します。

一方、「『By』Javaコミュニティによる」は、情報を発信する側を指しています。*Java Magazine*では、発信者を、読者同様重要な存在と認識しており、世界各国のエキスパートに協力を仰いでいます。本創刊号では、Java ChampionのAdam Bien氏、Michael Kölling氏、Kevin Nilson氏、Dick Wall氏の寄稿が掲載されています。今後は他のコミュニティの著名人にも参加していただく予定です。もしかすると、皆さまがその1人となるかもしれません。（ご興味があれば、ぜひお知らせください。）

*Java Magazine*は、双方向コミュニケーションが可能な1つのパッケージにすべてがまとめられており、デジタル形式であることの利点を最大限活用できるようデザインされています。（お気付きと思いますが、私たち制作者自身がこのプロジェクトをとても楽しんでいます。）

最後にお伝えしたいのは、*Java Magazine*は完成してしまったものではなく、進化の途上にあるものだということです。よりよい雑誌にしていくために、編集からデザインまであらゆる観点で皆さまのサポートを必要としています
。どうぞ*Java Magazine*をじっくりとお読みいただき、何かお気づきの点がありましたら、いつでもメッセージをお送りください。

私たちが*Java Magazine*を楽しんで制作したように、皆さまにも*Java Magazine*を楽しんでお読みいただければ幸いです。

編集長Justin Kestelyn BIO

写真：Richard Merchán

//フィードバックをお送りください /

今後の改善のために、いただきました提案はすべて検討いたします。分量によっては、直接の返信をいたしかねる場合がありますのでご了承ください。

Java 7、世界で提供開始

7月7日、オラクルは、間近に迫ったJava Platform, Standard Edition 7（Java SE 7またはJava 7）のリリースを記念して、世界規模のセレモニーを開催しました。
カリフォルニア州**レッドウッド・ショアズ**、英国**ロンドン**、ブラジル・**サンパウロ**の世界3か所のオラクル各本部で同時にイベントが開催され、コミュニティ・メンバーが登壇しました。Accenture、HP、IBM、London Java Community、Royal Bank of Scotland、Riot Games、SouJava、Travelexなどの代表者を招いて、組織の成功にJavaが不可欠である理由を全員で共有しました。
レッドウッド・ショアズのイベントでは、Oracle Fusion Middlewareグループ開発部門のVice President、Adam Messingerによるジェネラル・セッションに続いて、Java 7の主要改善点である言語機能拡張（JSR-334： Project Coin）、新規の軽量フォーク/ジョイン・フレームワーク、動的型付け言語のサポート（JSR-292：InvokeDynamic）など個別のテクニカル・セッションが実施されました。
また、59か国のJavaユーザー・グループに、ミーティング用に200以上のJava 7の「テクニカル・セッション」キットが配布されました。今、世界はJava 7を手にしようとしているのです。

写真：Enrique Aguirre

数々の受賞歴を有するJavaベースのゲームMinecraftはこちらでご覧頂けます。

Javaベースのゲームが快挙を達成

受賞を心よりお祝い申し上げます。スウェーデンのゲーム開発会社である**Mojang**が、Javaベースのゲーム**Minecraft**で5つの賞を獲得しました。**Game Developers Choice Awards**のInnovation Award（イノベーション賞）、Best Downloadable Game Award（ダウンロード可能ゲーム大賞）、Best Debut Game Award（新ゲーム大賞）、さらに**Independent Games Festival**のAudience Award（ユーザー賞）とSeumas McNally Grand Prize（Seumas McNallyグランプリ）に輝きました。このゲームについて詳しく知りたい方は、**ファンによるトレーラー**をご覧ください。

イベント

JavaOne 10月2～6日、カリフォルニア州サンフランシスコ

開催日やカンファレンス・トラックの内容が決定し、セッションの一般公募も締め切られました。さらにセッション・リストの最終版が選定され、エンターテイメント（Sting、Tom Petty and the Heartbreakers）についても発表されています。あとはあなたのJavaOne 2011への**参加登録**だけ。10月のサンフランシスコは、最新のJavaテクノロジーの話題で盛り上がります。詳細については、JavaOneカンファレンス・サイトをご覧ください。JavaOne 2011開催前のイベントについては、JavaOneカンファレンス・サイトのほか、JavaOneカンファレンス・**ブログ**、JavaOne **Twitterフィード**、**Facebook**および**LinkedIn**のJavaOneアカウント、**JavaOne Oracle Mixグループ**、**Java.net**でもご確認いただけます。

8月

PPPJ 2011
8月24日～26日、コンゲンス・リンビー（デンマーク）
Javaプログラミングの原理とプラクティスに関する国際カンファレンスで、今回で9回目となりました。Java言語およびJava仮想マシンを学んで利用する研究者、教師、実践者、プログラマーが一堂に会します。

Research Triangle Software Symposium
8月27日～29日、ノース・カロライナ州ローリー
エンタープライズ・ソフトウェア開発分野で最新のテクノロジーとベスト・プラクティスに関する講演がありました。

9月

JavaZone
9月7日～8日、オスロ（ノルウェー）
JavaZoneは、スカンジナビア半島のソフトウェア開発者が集う最大規模のイベント、ITプロフェッショナルたちが知識を交換するためのフォーラムです。

QCon
9月10日～11日、サンパウロ（ブラジル）
ソフトウェア開発に関する国際カンファレンス。Java言語から独立したさまざまなJavaプラットフォームの使用方法に関するトラックがあります。

10月

Silicon Valley Code Camp
10月8日～9日、カリフォルニア州ロス・アルトス・ヒルズ
開発者が互いに学び合うためのコミュニティ・イベント。どなたでもご参加いただけます。

GOTO
10月10日～12日、オーフス（デンマーク）
旧JAOOです。ソフトウェア開発者、ITアーキテクト、プロジェクト・マネージャー向けの教育や交流のためのフォーラムです。

BlackBerry DevCon Americas
10月18日～20日、カリフォルニア州サンフランシスコ
BlackBerry開発プラットフォームの最新のイノベーションを紹介する、開発者カンファレンスです。

写真：Hartmann Studios


Bruno Souza


Ben Evans


Martijn Verburg

JCP Executive CommitteeのJavaユーザー・グループ

先日、Java Community Process（JCP）Executive Committeeは、Standard/Enterprise Edition Executive Committeeのメンバーとして新たにSouJavaとLondon Java Community（LJC）を選出しました。JCPのさらなる発展が期待できる出来事といえます。**SouJava**はブラジルのJavaユーザー・コミュニティで、2011 年のExecutive Committee**特別選挙**においてStandard Edition/Enterprise Edition Executive Committeeの信任議席を獲得しました。代表はSouJavaのPresident、**Bruno Souza**氏です。

一方、**Ben Evan**氏が代表を務める**LJC**は同じくStandard Edition/Enterprise Edition Executive Committeeの空き議席を獲得しています。LJCの共同代表者である**Martijn Verburg**氏は、「ブラジルのJUG（Javaユーザー・グループ）の皆さんとともに、JCPメンバーから信頼をいただけたことを謙虚に受け止めたいと思います。世界中の何百万というJava開発者とユーザーの代表となることを楽しみにしています」と述べています。JCP Executive CommitteeへのJUGの加入はこれまでも広く評価されており、JCPでのオープン性と透明性のさらなる向上のための重要なステップとなっています。また、Standard/Enterprise Edition Executive Committeeのもう1つの信任議席はGoldman Sachsが獲得し、John Weir氏がその代表を務めます。Micro Edition Executive Committeeの空いた選出議席はAlex Terrazas氏が獲得しています。

Ben Evan氏の写真：Catherine Currie

写真：Matt Harnack

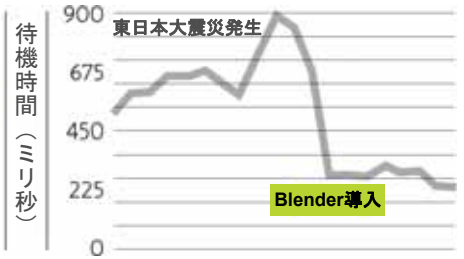
第1回Facebook Hacker Cupの世界チャンピオンがJavaを使用

ロシア人開発者の**Petr Mitrichev**氏が、第1回Facebook Hacker CupでJavaを使用して世界チャンピオンに輝きました。25人の決勝進出者に課せられたのは、Party Time、Safest Place、Alien Gameという3つのアルゴリズム問題をなるべく速く（2時間以内に）解くことです。

これら3つの問題すべてに解答できた決勝進出者はわずか3人、そのうち3問とも正解したのはMitrichev氏ただ1人でした。Mitrichev氏と3つの問題の詳細についてはこちらをご覧ください。

Javaへの移行でTwitter検索の速度が3倍に

Twitterは、「fail whale」（エラー画面に表示されるクジラ）の出現頻度を減らす試みとして、フロントエンド・サーバーをRuby on RailsからBlenderと呼ばれるJavaサーバーに移行しました。詳細については、Twitterによる発表をご覧ください。



Blender導入前後の検索API待機時間（95パーセンタイル値）

Javaの基本とトリックを学ぶ

Java 7リリースに伴い、アカデミックなイベントが多数開催されます。ここでは、8月の3日間のワークショップと、JavaOne 2011での約1週間に及ぶアクティビティについてご紹介します。

Java Summer Workshop（8月10日～12日）。カリフォルニア州サンフランシスコのベイ・エリアで高校生と教師を対象に3日間にわたって実施される、オラクル主催の無料ワークショップです。テーマは**Alice**と**Greenfoot**を使用したプログラミングで、参加者はAliceの3Dソフトウェアと Greenfootの2Dソフトウェアを使用し、アニメーションやゲームの開発方法とJavaプログラミング言語の概要について学びます。チュートリアルはイベント終了後、オンラインで利用できるようになっています。この**イベント**はカリフォルニア州レッドウッド・ショアズにあるOracle Conference Centerで行われます。

JavaOne（10月2日～6日）。教師や選考試験を通過した学生の皆さまは、ぜひ**Discoverパス**をご利用ください。お得な料金でこの業界を代表するJavaカンファレンスの数々のメリットを享受できます。プログラミングの初心者または未経験者向けのプログラミング・セッションに参加できます。プログラム未経験者の方にとっては、プログラミング関連のツールやプロジェクトを知るきっかけとなり、また、プログラミング経験者の方は、プログラミングの複雑なロジックに関するヒントやトリックについて学び、プログラミング・プロジェクトの知識を広げることができます。
このほかにも、Javaテクノロジーの最新情報に関するJavaOne基調講演や、JavaOne展示会場でのJavaソフトウェアの最新テクノロジーに関する実演デモやディスカッションなど、Javaを学ぶ機会がたくさん用意されています。さらに、学生や教師の皆さんも、Oracle User Groups PavilionやOTN Loungeで業界トップのプログラマーと交流できます。**ぜひ今すぐご登録ください**。

画像：I-Hua Chen

***Java Magazine*編集長のJustin Kestelynと対談するJCP議長のPatrick Curran**

Java Community Processの未来を定める

Java Community Process（JCP）の議長、**Patrick Curran**は、JCPの未来を決定付ける新たな2つのJava Specification Request（JSR）について**発表**しました。「JCP.next」と呼ばれるこの新しいJSRには、Java仕様開発プロセス使用時の公式手順を定めた**JCPプロセス文書**に対する変更点が記載されています。1つ目のJCP.next JSRである**JSR-348**は、JCPの透明性、参加の度合い、俊敏性、ガバナンスを高める目的で、Java Community Processのさまざまな変更点や修正点を提案するものです。2つ目のJSRでは、より複雑な問題に取り組む予定です。進捗状況については、**『JCP.next Resources』ページ**をご確認いただくか、Java.netの**JCP.nextプロジェクト**にご参加ください。

Java.netで拡大するプロジェクト・ホスト機能

今年は、Java.netのオープンソース・プロジェクト向けツールセットが急速に拡大しています。まず、2月の下旬に**Java.netのプロジェクト・インフラストラクチャ**がKenaiに移行しました。その後、Java.net/Kenaiインフラストラクチャの機能が継続的に拡張され、結果としてオープンソース・プロジェクトのリーダーや開発者が幅広い選択肢や強力な機能を得られるようになりました。
現在、Java.netのサービスには、SubversionおよびGitバージョン管理システム、JIRA課題/プロジェクト追跡、**SonatypeのNexus Maven Repository Managerサービス**などがあります。
また、2,000以上のアクティブ・プロジェクトで、アップグレード後のJava.netプロジェクト・インフラストラクチャが活用されています。参加ご希望の方は、Java.netの**『Create a Project』ページ**をご覧ください。

アフリカの開発者の組織的な活動

写真：Jcertif 2010

今、アフリカでは、大陸全土の開発者たちが国の枠を超えてグループを組織するという、新たな流れがおきています。2010年、Jean-François（Max）Bonbhel氏は、開発者同士が連携しやすくするために、アフリカのJavaユーザー・グループを統括するグループとしてJUG-AFRICAを設立しました。JUG-AFRICAは、今や14か国5,000人のメンバーが参加し、中央アフリカ最大のJavaコミュニティ・イベントであるJCertif（9月3日～4日）のスポンサーとなっています。

また、オラクルがスポンサーとなっているCoders4Africaという新しいグループが、初の大規模イベント「Coders4Africa 2011 Ghana」をガーナのアクラで開催しました（6月18日～19日）。共同設立者のAmadou Daffe氏によれば、グループの目的の1つは「アフリカ中のアフリカ人開発者が組織された環境のなかでコミュニティを構築すること」です。

JUG-AFRICAとCoders4Africaはどちらも、アフリカ人開発者が直面する固有の課題に組織的に対応しようとするもので、アフリカ全土に連携の精神が広まりつつあることを示しています。

OpenJDKがJava SE 7の公式リファレンス実装に

Java SEのProduct ManagerのHenrik Ståhlは、最近の**ブログ記事**で、オラクルはOpenJDKのみをベースとしたJava SE 7リファレンス実装（RI）バイナリを作成すると発表しました。

また、RIバイナリの利用について、商用実装者に対してはバイナリ・コード・ライセンスを適用し、オープンソース実装者に対してはGPLv2（クラスパス例外を含む）を適用すること、さらに**OpenJDK TCKライセンス契約**（OCTLA）がJava SE 7にも適用されるように更新することも発表しました。この変更により、オープンソース実装者は、Java SE RIのソース・コードを利用して自分たちの実装と直接比較し、互換性を検証できます。また、テクノロジー互換キット（TCK）も無料で利用可能です。

Java関連書籍

THE WELL-GROUNDED JAVA DEVELOPER-JAVA 7 AND POLYGLOT PROGRAMMING ON THE JVM

共著：Benjamin J. Evans、Martijn Verbur
出版社：Manning Publications

Javaの基本を十分に理解している開発者向けのユニークなガイド。Java 7の新機能を新鮮かつ実践的な視点でとらえており、開発者が次世代のビジネス・ソフトウェアを構築するために使用できる補助的なテクノロジーの数々も紹介しています。Java 7新機能を詳細にまとめているほか、Groovy、Scala、Clojureなどの新しいJava仮想マシン（JVM）ベース言語についても横断的に取り上げています。同時実行性とパフォーマンスに対する最新アプローチを採用した貴重な開発テクニックの数々を学べる1冊です。

JAVA FOR PROGRAMMERS, 2ND EDITION

共著：Paul Deitel、Harvey Deitel
出版社：Pearson/Prentice Hall Professional

高度な言語プログラミングの経験があるプログラマー向けに、Java言語とJava APIが詳細に解説されています。Deitelの特徴的なプログラミング教育法であるLive-Code（有効なコード）アプローチにより、Javaのコンセプトが、完全にテストされたプログラムのコンテキストに沿って説明されています。構文が網掛けされ、コードが強調表示されていて、コードを1行ずつ確認し、プログラムの出力も見ることができます。この書籍には、200以上の完全なJavaプログラム、実績のある18,000行以上のJavaコード、堅牢なアプリケーションを構築するための数百のヒントが含まれています。

サンプルの章を読む「Object-Oriented Programming:Polymorphism」

ブラジルのGlobo TVが提供する
新しいコンテンツと可能性

Javaによる双方向TVの実現

David Baum

自宅で、最近お気に入りのテレビドラマを見て過ごすある1日を想像してください。『Viver a Vida』は、ブラジルでとても人気のある双方向テレビ番組です。主人公のHelena（Taís Araújo）がリオデジャネイロ州の海辺の町ブジオスで海岸沿いを散歩しているシーンで、カメラがズーム・インしテレビ画面の下部にあまり目立たない小さなアラートが表示されます。ここから、主人公の生い立ちやドラマの設定などの詳細情報にアクセスできるようになっています。翌日このドラマが放送されるときまでには、ドラマティックなストーリー展開に関する数々の複雑な背景についてもすっかり熟知していることでしょう。

写真：Paulo Fridman

Globo TVエンジニアリング・マネージャー、Carlos Fini

調整室での操作を監督するGlobo TVのエンジニアリング・マネージャー、Carlos Fini氏。Globo TVはJavaを使用して、視聴者のテレビ体験を受動的なものから能動的なものへと変貌させた。

企業概要

Globo TV
globotvinternational.com

所在地：
ブラジル、リオデジャネイロ

業界：
メディア/エンターテイメント

従業員数：
8,500人

Javaを利用することで、ブラジルのテレビは受動的なメディアではなくなりました。ブラジル最大の放送局、Globo TVが提供する双方向デジタル・テレビ・サービスでは、視聴者が音声/映像を操作できます。たとえば、投票、広告への応答、スポーツの統計データの閲覧、ビデオ・クリップのダウンロード、クイズへの参加、ゲームのほか、気象情報や交通情報、ローカルニュース放送の受信方法のカスタマイズもできます。また、Javaアプリケーションを使って、電子メールの送信、スポーツ選手のプロフィールの確認、口座残高の確認、テレビ放映中の製品やサービスの購入可能です。

「Globo TVは、さらにモバイル機器への対応を拡大し、新しい双方向機能とリッチメディア体験を視聴者の皆さまに提供しています」と話すのは、Globo TVのエンジニアリング・マネージャーであるCarlos Fini氏です。「Javaはオープンソースでロイヤリティ・フリーである上に、大規模で積極的な開発者コミュニティでサポートされています。当社のニーズに合う最良の選択肢でした。」

Globo TVは、ブラジルで規模、影響力ともに最大の放送局です。また、171か国にコンテンツを輸出しています。ブラジル国民の大半が無料のテレビ放送を視聴しており、Globo TVはこの地上波放送の市場で70%のシェアを占めています。Fini氏はGlobo TVで15年にわたって技術チームを率いてきました。同チームのJavaプログラミング・スタッフは現在、新しいタイプの双方向テレビ・コンテンツを配信するためのソフトウェア・アプリケーションの構築に取り組んでいます。視聴者はリモコンのiボタンを押すだけで、そうした双方向機能にアクセスできます。この操作により、デジタル・メディア放送とともにセットトップ・ボックスに配信されているJavaアプレットが起動します。

「Javaは多くのエンターテイメント・デバイスでサポートされています。そのため、ゲーム・システム、携帯電話、セットトップ・ボックス、Blu-rayプレーヤなどで受信されるのと同じテクノロジーを使用して、当社の双方向テレビ番組を配信できるのです。さらに、JavaをGingaミドルウェア規格と併用することで、テレビの視聴者にコンテンツを配信し、また視聴者からフィードバックを受信する新たな手法を提供します。」

Java利用状況
セットトップ・ボックスから組込みの高画質チューナーまで、自宅用**Java対応デバイスの販売総数は30億台**

デジタル・テレビの展望

多くの国が、電波の有効利用のために、地上波アナログ・テレビ放送からデジタル・テレビ放送へと移行しています。

ブラジルのドラマ『Viver a Vida』の視聴者は、リモコンのiボタンでプログラムを双方向的に操作できる。ここでは、前回のエピソードの概要を表示させている（写真右）。

米国では、デジタル・テレビはおもに画質/音質が優れているという理由で一般的になりました。ブラジルのデジタル・テレビには、さらに双方向性という特長が加わります。視聴者は、サッカーのワールド・カップの試合中に良い選手を投票するようリアル番組の表示アングルを変更できるなど、自由に視聴方法をカスタマイズでき、企画に参加できます。

実際、この双方向環境を完成させるためにGlobo TVを動かしたのは、2010 FIFA ワールド・カップ・チャンピオンシップでした。トーナメントの統計データや選手のプロフィール、これまでの試合の概要などを好きなときに自由に見られる機能があれば、ブラジルの何百万もの熱狂的なサッカー・ファンは喜んで利用するだろうと、Globo TVのコンテンツ開発者やプログラミング・エキスパートは確信していました。さらに、Globo TVの経営陣も、それが非常に興味深いビジネス・モデルとなる可能性を理解していたのです。

デジタル信号を使用すると、必要な帯域幅が少なくてすむだけではなく、ショッピング、投票、ゲームといった新しいメディア機能のためのリッチ・コンテンツや双方向機能を実現できます。放送局は番組を放送する信号の中でGingaアプリケーションを配信して、音声/映像ストリームの強化、メディアの再生操作やディスプレイのハードウェアおよびチューニング機能の制御、他のアプリケーションの起動、標準の放送信号に対する補足的なメディア・コンテンツのオーバーレイなどを行うことができます。

Fini氏は次のように説明します。「他の新しいメディア・サービスと競争するために、柔軟性に優れた開発環境が必要でした。さらに、視聴者を魅了して視聴率を向上させるための圧倒的なユーザー・エクスペリエンスを構築したいと考えていました。Javaはオープンで、さまざまなプラットフォームで動作します。」

なんと言ってもJavaの魅力は、この「どこでもだれでも利用できる」、ユビキタスであることです。他の多くの国々と同様に、ブラジルにもJava開発者の大規模なコミュニティがあるため、Globo TVは開発プロジェクトで信頼できる幅広い専門知識を活用できました。Javaはコンピュータ業界に浸透しており、8億4,000万以上のJavaデスクトップが使用されているだけでなく、ホーム・エンターテイメント業界でも普及しており、セットトップ・ボックスから組込みの高画質チューナーまで、すでに30億台ものJava対応デバイスが販売されています。さらに、Javaはモバイル・コンピューティング市場にも急速に普及しつつあり、26億台のJava対応電話が流通しています。これは、世界中の全携帯電話の約85%を占めます。放送業界では、約180の事業者がJavaコンテンツを配信しています。

事業内容のポイント
Globo TVは171か国にコンテンツを輸出

放送局とメーカーにとってのミドルウェア

Sunは、2006年に創設されたブラジルのデジタル・テレビ・フォーラムであるSBTVD（Sistema Brasileiro de Televisão Digital）フォーラムと協力してJava DTV規格を策定しました。この規格は、ブラジルのデジタル・テレビの送受信に関する標準と技術仕様を示すものです。同グループの目的は、デジタル・テレビ規格を策定し実施することで、技術面や経済面の問題に対処するだけではなく、政府と市民との距離を縮める「情報社会」を推進することでした。これは、テレビを所有する世帯が96%に上る一方、コンピュータを所有する世帯は20%に満たない国にとっては重要です。

「Javaは双方向テレビ・サービスの新しいソリューションを生み出すために利用できる適切なツールだと考えました」と話すFini氏は、SBTVDフォーラム創設メンバーの1人です。

SBTVDフォーラムでのもっとも重要な決定事項の1つが、ブラジルのミドルウェア規格としてGingaを採用したことです。Gingaには高レベルのオープン開発環境と標準機能ライブラリがあるため、デジタル・テレビ・アプリケーションをより容易に構築できます。

オープン仕様であるGingaは習得しやすい上にロイヤリティ・フリーのため、コンテンツ制作者に広く浸透しています。また、メーカーの間でも、テレビ、セットトップ・ボックス、周辺機器にGingaを採用するケースがますます増えています。Gingaを使用すると多様なeコマース・アプリケーションを構築できるため、Gingaアプリケーションの導入は最終的に、消

費者向けのテレビ受信機やセットトップ・ボックスのコスト削減につながります。

GingaにはGinga-NCLとGinga-Jという2つのプログラミング・パラダイムがあります。Ginga-NCLは、宣言的アプリケーション向けのマルチメディア・プレゼンテーション環境です。開発者はこのXMLベースの言語を使用して、メディア・オブジェクトの同期、メディア・コンテンツの制御、家庭用デバイスへの双方向プログラムの表示を行うことができます。Ginga-JはJavaアプリケーションの実行インフラストラクチャとなります。

「Ginga-Jは、ブラジルのシステムに合った固有のAPIを使用してJava DTVを拡張するもので、家庭用デバイスの双方向操作と非同期メッセージング環境の管理を実現します。」

Ginga-Jは、Java DTV APIとともに2009年4月にSBTVDフォーラムで承認されました。Java開発者は、各規格に基づいて、放送局からコンテンツを受信し（「伝送」規格）、そのコンテンツを読取り（「受信」規格）、コンテンツを受信契約者に送信する（「復元」規格）アプリケーションを作成できます。以降、オラクルのロイヤリティ・フリーの規格はブラジルやその他南米のほとんどの国において地上波デジタル・テレビの規格となりました。

Javaデータ

モバイル・コンピューティング市場に流通する**Java対応電話は26億台**（世界中の全携帯電話の約85%）

また、SBTVDフォーラムはISDB-T規格も採用しました。これは、元々は日本で普及した規格で、現在は南米のほとんどの地域に波及しています。ISDB-Tb（bはブラジル版の規格であることを示します）は、特にワンセグ機能の点で先進的でした。ワンセグ機能により、放送局は標準解像度と高解像度のテレビ放送だけではなく、どの事業者でも自由に参入できる携帯電話やタブレット向けの低解像度信号を埋め込むことができます。ワンセグは、開発途上国において市民がニュースや情報にアクセスできるという点で、非常に魅力的な機能であるといえます。

双方向テレビのポイント

現在、双方向テレビの広範なアプリケーションとコンテンツを使用したGlobo TVのラインアップは、Fini氏のチームに属する10人の開発者が支えています。「すべてがまったく新しいので、このチームは新モデルのパイオニアとなっています。」とFini氏は話します。開発者たちはオラクルのLightweight UI Toolkit（LWUIT）を使用しています。量販されているモバイル・デバイスをターゲットにしたUIライブラリであり、さまざまな画面サイズに合わせたデバイス固有のコーディングは不要です。Globo TVは大型テレビと小型モバイル・デバイスに対して同時にコンテンツを放送しているため、この点は非常に重要です。

Globo TVのデジタル・テレビ・アプリケーションは、PBP Xletという形式で作成されています。これは、署名付きJARファイルにパッケージ化され、放送ストリームの一部として送信される自己完結型のJavaアプリケーションです。Xletを組み込むことで、放送局はアプリケーション機能をテレビ番組と同期させることができます。これにより視聴者は信号の開始、停止、一時停止ができるほか、ショッピング、投票、コンテンツのダウンロードといった双方向テレビ機能を自由に操作できます。

「Javaテクノロジーの利用によって、番組関連情報、ゲーム、投票、ターゲットを絞った広告、電子政府などのリッチ・メディア・アプリケーションや双方向アプリケーションを、通常のテレビ放送と統合できます。プラットフォーム・アーキテクト、アプリケーション開発者、メディア・コンテンツ作成者はこのようなアプリケーションを積極的に開発しています。」

Java DTV規格では複数のアプリケーションを同時に、しかも自律的に、永続ファイル・システム内の他のアプリケーションから保護して実行できます。アプリケーション、音声、画像などの情報を、カルーセル方式で構成される伝送ストリームに埋め込み、セットトップ・ボックスに送信できます。Java Media Frameworkプレーヤは時間ベースのメディア・ストリームを処理し、通常はメディア・デコーダに関連付けられます。

メディア評論家は、ブラジルにおける双方向テレビの輝かしい可能性を認めています。ブラジルはロイヤリティ・フリーのGinga規格を採用し、さらにJavaで補強したことで、米国やその他の国のように双方向テレビが手詰まり状態になることを回避できます。

「2011年、ブラジル市場ではデジタル・テレビの利用者数が1,700万～2,000万人に達する見込みで、その一部は当社の双方向テレビ・サービスを利用しています。」Fini氏は最後にこう結びました。「標準言語を使用することで互換性を確保できます。しかも、Javaは機能が豊富で堅牢なのです。」●

Java開発者向けリソース

ダウンロード

Java DTV API 1.0 Specification

フォーラム

LWUITオープンソース・コミュニティ
Java TVフォーラム・ホーム
Java Embeddedフォーラム・ホーム

データ・シート/ホワイト・ペーパー

Java Technology for Digital Media
Java Technologies for Interactive Television
JavaTV API Technical Overview

David Baum。カリフォルニア州サンタバーバラを拠点とし、革新的なビジネス、最新テクノロジー、魅力的なライフスタイルについて執筆活動中。

Travelexエンタープライズ・ソリューション・アーキテクト、Colin Renouf

Java利用の成功例

外貨両替のトップ企業、**Travelex Group**、グローバルITプラットフォームにJavaを採用　**Philip J. Gill**

国際的な通貨取引、貿易、旅行において大きな影響力を持つのはドル、ユーロ、ポンド、円でしょう。しかし、今や企業や旅行者が必要とする通貨は決してこれらだけではありません。このグローバル化の時代においては、遠く離れた国々のあまり知られていない通貨の重要性が高くなってきています。

アンゴラの通貨であるクワンザを例にします。わずか数年前には、ドルやユーロをクワンザ（アンゴラ最大の河川名に由来）に両替する必要があると考える人はほとんどいませんでした。

写真：John Blythe、Getty Images

企業概要
Travelex Group

www.travelex.com
本社所在地：
英国ロンドン
業界：
外貨両替
売上：
11億米ドル以上
（2010会計年度）
従業員数：
7,000人以上
使用している Javaバージョン：
Java Platform, Enterprise Edition 6（Java EE 6）

「Travelexが金融業務にJavaを利用する主な理由は移植性、スケーラビリティ、国際化です。」Travelexエンタープライズ・ソリューション・アーキテクト、Colin Renouf氏

しかし今日ではクワンザの両替を行う人は多数います。このアフリカ南部の共和国が新興企業や旅行者の目的地となっているからです。

クワンザのみならず、80以上の通貨について日々の変動を追跡し、国際的な通貨取引に関する指針を示している数々の国内財政法に則った取引を実行する、これが、英国ロンドンを本拠地とする世界トップの外貨両替と国際決済の専門業者である、Travelex Group（以下、「Travelex」）のビジネスです。

Travelexのエンタープライズ・ソリューション・アーキテクトであるColin Renouf氏は次のように述べています。「当社の業務は外貨両替ですが、通貨に関する専門知識を活かしたデータ交換ビジネスでもある、と私は思っています。」

以前は、Travelexの世界各国での地域業務をサポートするITシステムは、WindowsとNETテクノロジーで構築されていました。しかし、業務が世界規模に拡大するのに伴い、Travelexの上層部はシステムの問題に気づきました。特に新興経済圏における成長に対応するには移植性とスケーラビリティに優れた標準ベースのプラットフォームが必要ですが、このシステムはそうではなかったのです。「Windowsの問題点は、非標準の独自仕様である上にソースも公開されておらず、統合が難しいということです。」

現在の状況への対応だけでなく、将来の成長を支えていくためには、古いシステムを新しいプラットフォームに交換する必要があると上層部は実感しました。そして現在、TravelexはJavaに大きな魅力を感じています。「現在開発中のグローバル・エンタープライズ・システムはすべてJavaで構築しています。将来的にも、開発するグローバル・エンタープライズ・システムの大半はJavaで構築する予定です。」

Javaデータ
Travelexは、**ノンバンクとしては世界最大**の国際通貨決済業者

1から1,000へ

Travelexは、1976年にロンドンで1軒の両替店として創業しました。現在は、同社が展開する消費者向け業務で、24か国に約1,000店舗、500台以上のATMというネットワークを通じて、年間3,000万人以上のリテール顧客に外貨両替サービスの提供やプリペイド・カードの発行を行っています。Travelexグループ企業のTravelex Global Business Payments（TGBP）は、企業向け業務を手がけており35,000社を超える企業に国際企業決済サービスを提供しています。2011年7月5日、TravelexはWestern Unionに、TGBPを6億600万ポンドで売却することに合意しました。これにより、同グループが既存の市場と新しい市場での成長を加速させ、顧客が世界中で革新的な外国為替製品/サービスを利用できるようにするために、さらなる投資の実行のための資本が確保されます。この取引は、2011年末に向けて締結される予定です。それまでは、TGBPはTravelexの重要な事業体として残ります。

Travelexでは、M&Aによって成長してきたことも一因となって、各地のITシステムがそれぞれ独立して稼働していました。買収時点で稼働していたシステムがそのまま使用され続ける傾向にあったのです。

しかし、2008年春にTravelexの現CIOであるSteve Grigg氏が就任してから、この傾向が変わり始めました。金融業界での経験が豊富なGrigg氏は、エンタープライズ・システムの構築と統合において金融機関が直面している問題や、すべての業務部門を同じITの土台で管理するメリットを理解していました。Renouf氏によると、Grigg氏の使命は、世界各地から開発、統合、配置を実行できる単一プラットフォーム上に新しいシステムを構築することで、世界中の多くの地域を1つにまとめることでした。

テクノロジーを超えて

Javaは、比類のないスケーラビリティ、プログラマビリティ、信頼性を備えるエンタープライズ・システムを開発するためのテクノロジー・プラットフォームですが、Travelex Groupのエンタープライズ・ソリューション・アーキテクトであるColin Renouf氏にとってはそれだけにとどまりません。Renouf氏にとって、もっとも重要なJavaの資産はJavaコミュニティです。

すべての活気のあるコミュニティがそうであるように、Javaコミュニティが健全性を維持しながら成長するには個人個人の参加が不可欠です。たとえば、Travelexは、Javaプラットフォームの国際化に関するセキュリティに携わりました。ダブルバイト文字セットなどの利用の結果、異なるレイヤー間で一部変換が生じることを確認し、Renouf氏の言う「Javaレイヤーに入るものはそのレイヤーに合う形式となっている」ことを保証する方法を調査しました。この作業の一部はオラクルや他のパートナーにフィードバックされ、コミュニティ全体にフィードバックされたものもありました。

また、Travelexは教育活動にも携わっています。「チームの多くのメンバーがJavaコミュニティで非常に積極的に活動しています。3人のメンバーは書籍や雑誌の記事を執筆し、標準の策定に取り組み、コミュニティ・イベントを開催したことがありますし、ときには一部の顧客のために仕事をし、業界の大規模なイベントを外部からサポートもしました。」さらに、Renouf氏やメンバーたちは他の人にも参加するよう呼びかけたということです。「コミュニティに積極的に参加することで、自分の要件への対応状況についてよく理解し、また協力を得ることができるほか、『他者に何らかの影響を与えた』という一定の満足も得られます。参加すること自体が力なのです。」

そのプラットフォームこそがJavaです。Javaは、移植性、スケーラビリティ、信頼性に優れ、完全に国際化され、機能が豊富な標準ベースのアプリケーション開発プラットフォームです。Grigg氏が同社に採用したRenouf氏は、次のように語ります。「Javaは非常に豊富な機能を備えた開発環境であり、どのようなアプリケーションにも使用できます。しかし、こうしたJavaの機能が有用なのは、エンタープライズ・システムとB2B（business-to-business、企業間取引）システムにおいてです。なぜならこうしたシステムでは、各種オープンソース・テクノロジーやEJB（Enterprise JavaBeans）テクノロジーに備わった統合のしやすさと、Javaの優れたWeb機能や国際化機能とが相乗的に作用するためです。このことにより、当社では組織ごとに発展してきたローカル・システムを統合して、世界中の業務で使用できているのです。」

Renouf氏はさらにこう述べます。「同時に、Travelexが利用するインタフェースとメカニズムでは、さまざまな地域にまたがるアーキテクチャを次第に整合させながら、資産全体を、より豊富な機能とより素早いサービスを提供できるものへと変革させることができました。」

Travelexが最初に導入した新Javaシステムの1つが、Global Payments Gateway（GPG）です。このシステムの開発、テスト、配置は9か月もかからずに完了しました。Renouf氏によれば、GPGはOracle WebLogic Server 11g、Oracle Database 11gなどのオラクル製品を含む「完全なJava/Oracleスタック」で構成されています。GPGでは、Travelexの多くの企業クライアントや金融機関クライアント向けの国際通貨取引を処理するほか、これらのクライアントを国際銀行間通信協会（SWIFT）のさまざまな金融メッセージング・ネットワークに接続します。SWIFTは200の国と地域で9,000以上の金融機関を結んでいます。

Javaデータ

TravelexのJavaベースのGlobal Payments Gatewayのスケーラビリティは、**数回/分から400～500回/秒までのトランザクションに対応可能**

Travelexのクライアントは GPGを使用して、どこからでも、どこへでも、どの通貨でも送金を行うことができます。「クアラルンプールのだれかに送金する場合も、国、通貨、送金金額を選択し、自分の銀行口座を入力するだけでよいのです。」さらに、Renouf氏によれば、Webベースの新機能を使用して個人顧客が家庭や職場のコンピュータから送金することもできます。

GPGがJavaでコーディングされているおもな理由は移植性、スケーラビリティ、国際化です。Renouf氏によると、GPGには、1秒間に最大400～500トランザクションを処理する地域から、1分間に数回のトランザクションを処理すれば済む地域まで対応可能なスケーラビリティが備わっています。

しかしながら、Javaの本当に重要なポイントは、開発から配置に至るまで簡単に統合できる共通アーキテクチャを、どこにいても同じように利用できる点だとRenouf氏は言います。「Javaが共通だからこそ、さまざまな地域で同じ処理が可能なのです。」

TravelexではJavaの国際化機能により、システムのフロントエンドだけでなくバックエンドで実行されるビジネス・ロジックも数十の地域言語をサポートできるようになりました。このため、現在では、米国市場向けに開発した機能を、中国や日本など公用語が異なる地域でも簡単に配置できます。

Renouf氏によると、Travelexが独自に多くの労力を注いだ分野は信頼性です。世界各国の法規制に対応するために、Travelexはシステムで発生する可能性がある障害について、発生する理由やポイント、リカバリ方法を理解する必要があります。Javaはこの点において、他のプログラミング環境よりもずいぶん簡単に対応できるとRenouf氏は語ります。「Javaには、最初から非常に多くのツールが用意されていて、今起きている状況を正確に理解することができます。ただ、これらのツールを活用している人はほとんどいないようです。当社は、これらのツールで把握できる情報に基づき、例外的な状況での動作を図式化および文書化して、より信頼性が高くセキュアなシステムの開発方法を編み出しています。」●

Philip J. Gill、フリーランス・ライター兼編集者。活動拠点はカリフォルニア州サン・ディエゴ。●

Michael Kölling

最初の一歩を踏み出す

Greenfootを使用したJavaクラス、オブジェクト、メソッドの作成

これまで私が執筆した『Wombat Object Basics』、『Wombat Classes Basics』などの記事では、オブジェクト指向の重要な基本概念について、おもに既存のソース・コードを精査し理論を検討するという方法で説明してきました。しかし、プログラミングは実践です。そのため、この記事では、すぐに独自のクラスを作成することから始めます。そして、実際にプログラミングを行いながら、オブジェクト、メソッド、パラメータについて学習していきましょう。

前提条件

この記事に従って作業を進めるには、コンピュータに次のソフトウェアをインストールしておく必要があります。

- Java Platform, Standard Edition（Java SE）
- Greenfoot version 2.1.0以降

Greenfootはできる限り最新のバージョンを使用してください。**「About Greenfoot」**メニュー項目でバージョンを確認できます。

プロジェクトの作成

これまでの記事では、既存のプロジェクト（wombats）を使用しましたが、今回は独自のプロジェクトを作成します。

Greenfootでは、プロジェクトを*シナリオ*と呼びます。それでは、新しいシナリオの作成から始めましょう。

1. Greenfootを起動します。
2. Greenfootでは通常、前回作業したシナリオが開かれます。**Scenario**メニューの**「Close」**を選択して、このシナリオを閉じます。
3. Scenarioメニューの**「New」**を選択します。
4. New Scenarioダイアログ・ボックスで、新しいシナリオに**turtle**という名前を付けて**「OK」**をクリックします。

新しいシナリオ・ウィンドウが表示されます。この時点では、ほとんどの項目はグレーになっています（**図1**）。

これで新しいシナリオの枠、つまり作業を開始できる場所ができました。しかし、まだ何かが起こる場所となる世界（World）もなく、何かを行うことができるアクター（Actor）もいません。

ウィンドウの右側にある2つのクラス**World**と**Actor**は*スーパークラス*です。これらは特定の世界や特定のアクターではなく、存在しうるすべての世界とアクターを表しています。特定の世界とアクターを作成するには、これらの*サブクラス*を作成します。サブクラスは、スーパークラスを具体化したものです。

では、まずは世界を作成しましょう。

Worldの新規作成

1. World クラス（ベージュ色のボックス）を右クリックし、**「New Subclass」**を選択します。

新しいクラスの名前とイメージを入力するためのダイアログ・ボックスが表示されます（**図2**）。

1. クラスに**TurtleWorld**という名前を付け、イメージ・ライブラリの**Backgrounds**（背景）カテゴリからイメージを選択します。

この記事のスクリーンショットでは、weave.jpgという背景イメージを選択しています。

2. **「OK」**をクリックし、Greenfootメイン・ウィンドウの**「Compile」**ボ

図1

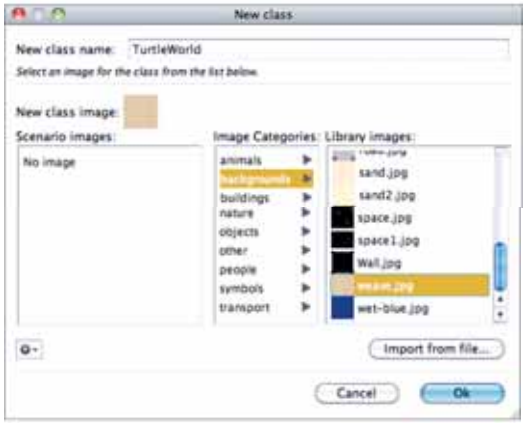


図2

図3

写真：John Blythe

タンをクリックします。
Greenfootウィンドウに、選択した背景が設定された世界が表示されます（**図3**）。
これで世界の作成が完了し、この世界に配置するアクターを作成できるようになりました。

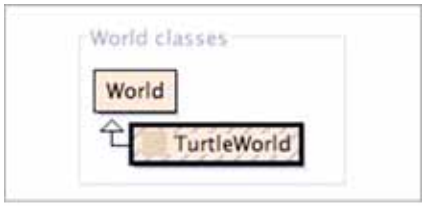


図4

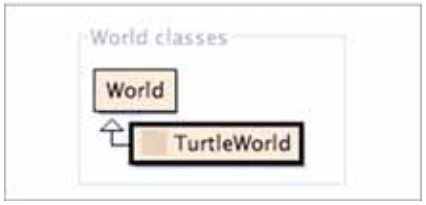


図5

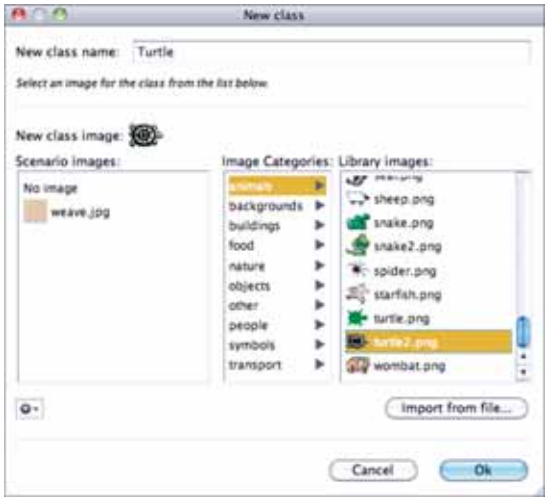


図6

コンパイルについて
プログラミングの際はJavaコードを記述します。しかしながら、コンピュータでJavaコードを直接実行することはできません。コンピュータには専用のマシン語での命令が必要です。
この問題を容易に解決できるツールが、コンパイラです。コンパイラは、Javaコードをマシン語に翻訳できるソフトウェアです。新しいクラスを作成するたびに、また、既存のクラスのソース・コードを変更するたびに、クラスを再度マシン語に翻訳する必要があります。この翻訳処理をコンパイルといい、**「Compile」**ボタンをクリックするだけで行うことができます。
Greenfootでは、変更後にコンパイルしていないクラスはストライプ表示されます（**図4**）。
「Compile」ボタンをクリックすると、クラスが翻訳されてストライプが消え（**図5**）、クラスを使用する準備が整います。

Actorの作成
アクターが居住できる世界はすでに作成しました。しかし、何かを行うアクターはまだいませんので、アクターを作成しましょう。

1. Actorクラスを右クリックし、**「New subclass」**を選択します。
2. 表示されるダイアログ・ボックスで、新しいクラスにTurtleという名前を付け、animalsカテゴリからturtle2.pngイメージを選択します（**図6**）。
3. **「OK」**をクリックします。
これで、シナリオに新しいTurtleクラスが作成されました。このクラスは作成時点ではストライプ表示されます（つまり、コンパイルされていません）。
4. **「Compile」**ボタンをクリックし、Turtleクラスをコンパイルします。
Javaの命名規則により、Javaクラス名は必ず大文字で始まります。

リスト1

```
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

/**
 * Write a description of class Turtle here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Turtle extends Actor
{
  /**
   * Act - do whatever the Turtle wants to do. This method is
   * called whenever
   * the 'Act' or 'Run' button gets pressed in the environment.
   */
  public void act()
  {
    // Add your action code here.
  }
}
```

リストをテキストで表示する

オブジェクトの作成
アクター・クラス（Turtle）は作成しましたが、まだアクター・オブジェクトはありません。
Turtleクラスを右クリックし、**「new Turtle()」**を選択すると、カメのオブジェクト、つまり、世界に配置できる実際のアクターが作成されます。
これで、1匹目のカメを作成できました。カメはいくつでも好きなだけ作成できますので、ぜひ試してみてください。

オブジェクトのプログラミング

1. **「Run」**ボタンをクリックします。
以前の記事では、**「Run」**をクリックするとウォンバットが世界中を駆け回りました。しかしここでは何も起きません。カメはただいるだけで、何もしないのです。
これは、カメが何かをするようにプログラミングしていないからです。ここからがプログラミングの面白いところです。Javaプログラムを記述して、カメを動かしてみましょう。
2. Turtleクラスのボックスをダブルクリックして、Turtleのエディタを開きます。Turtleクラスの一部のコードが表示されます（**リスト1**を参照）。
actメソッドについて考えてみましょう。actメソッドのコードは次のよ

うになっています。

```
public void act()
{
  // Add your action code here.
}
```

このコードでは、カメが行動する（act）ときに何をするかを指定します。波括弧（{ }）の間のコードは、カメが行動するたびに実行されます。今は、次の行が記載されているだけです。

```
// Add your action code here.
```

行の最初の二重スラッシュによって、この行はコメントとして認識されます。コメントは単にプログラマーに対する注意書きであり、Javaシステムでは無視されます。言いかえれば、このactメソッドにはコードがまったく含まれていません。カメが何もしないのはこの理由からです。
この行を変更します。

3. 次のように、actメソッドのコメントを命令に書き換えます。

```
public void act()
{
  move(2);
}
```

4. コンパイルし、新しいカメを作成して、「Act」ボタンと「Run」ボタンをクリックします。

では、今行ったことを詳しく見てみましょう。
move(2);という命令によって、カメは2ピクセル前進します。カメはこれを、行動する（act）たびに実行します。「Act」ボタンをクリックすると、actメソッドが1回実行されます。このため、カメは少し（2ピクセル）右に動きます。「Run」ボタンをクリックすると、actメソッドが（プログラマーが一時停止するまでは）繰り返しコールされるので、カメは動き続けます。

5. 2以外の値を試します。

2の代わりに20を指定すればどうなるでしょうか。

6. 他の命令を試します。

move(2);をturn(2);に書き換えてください。

ここでも忘れずに、コードを変更したらコンパイルし、新しいカメを作成してから実行します。先ほどと同じように、2以外の値も試してください。

メソッドとパラメータ

では、今行ったことを詳しく見てみましょう。
moveというメソッドとturnというメソッドをコールし（*呼び出し*）、それぞれにパラメータ2を渡しました。
メソッドは、オブジェクトが実行すると知っている振る舞いのことで、すべてのアクターはmoveメソッドとturnメソッドを知っています（つまり、移動と回転の方法を知っています）。
moveメソッドとturnメソッドは、どちらもパラメータを受け取ります。パラメータは、移動距離や回転量を正確に伝える追加情報です。どちらのメソッドも、パラメータとして数値を受け取ります。プログラマーはその数値を、メソッド名に続く括弧の中に記述することで指定します。さらにJavaの各命令の末尾にはセミコロンを記述します。

エラー

ここまでの作業で、コードは正確に記述しなければならないことに気づいたかもしれません。1文字でも間違えば、プログラム全体が動作しなくなります。このような場合、コンパイラによってエラー・メッセージが出力され、プログラマーはコードを修正する必要があります。

ここまでまだエラー・メッセージを見たことがない場合は、今すぐ試してみましょう。たとえば、命令の後のセミコロンを削除してコンパイルしてみてください。今説明したことが何か、おわかりいただけるでしょう。

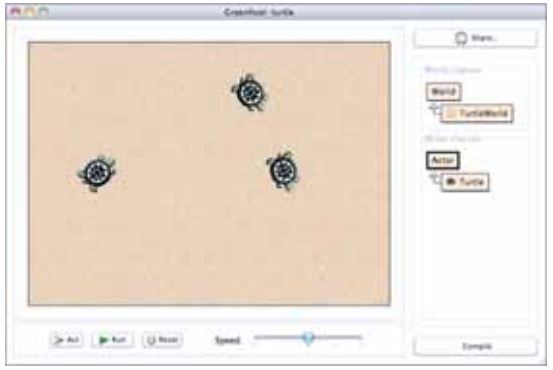


図7

連続した命令

actメソッドには、複数の命令をいくつでも含めることができます。つまり、次のように命令を連続して記述できます。

```
public void act()
{
 move(4);
 turn(2);
}
```

これを試してみてください。さらに、複数のカメを世界に配置して、moveメソッドとturnメソッドの両方にさまざまなパラメータ値を指定してみましょう（図7）。

まとめ

この記事では、独自のJavaコードを記述するための最初の手順について学び、次の原則について実際に確認しました。

- オブジェクトの振る舞いはオブジェクトのクラスに記述する。
- より正確には、オブジェクトの振る舞いはオブジェクトのクラスのメソッド定義に記述する。
- GreenfootのActorクラスには、おもな振る舞いを記述するためのactというメソッドが含まれている。
- actメソッドの本体（波括弧内の行）には、メソッドがコールされたときに何を実行するかを定義するコードが含まれる。
- 命令は、既存のメソッド（この例ではmoveやturnなど）をコールすることで指定できる。
- メソッドのコールは、コールするメソッドの名前と、それに続く括弧内のパラメータで構成される。
- すべての命令の末尾にはセミコロンを付加する。
- 命令を連続して記述することもできる。●

参考情報

- Greenfoot
- Java SE API
- Young Developersリソース
- Young Developersシリーズ
  『Wombat Object Basics (Part 1)』
  『Wombat Classes Basics (Part 2)』

Max Bonbhel

BIO

RESTful Webサービスの概要

RESTful Webサービスによるアプリケーションの機能の拡張や機能の追加

全3回シリーズの初回となるこの記事では、Representational State Transfer（REST）Webサービスを統合したJava Platform, Enterprise Edition（Java EE）アプリケーションの作成方法について説明します。アプリケーションの作成にはNetBeans 6.9.1 IDEとJava EE 6を使用し、アプリケーションのデプロイにはOracle GlassFish Server 3.1を使用します。

Java Magazine次号に掲載される第2回では、JavaScript Object Notation（JSON）フレームワークを使用してWebサービスからの応答を管理する方法について取り上げます。さらに、第3回ではJava API for XML Web Services（JAX-WS）の統合について説明します。これらの記事では、アプリケーションの俊敏性を高める方法が明らかになります。

Webサービスとは

Webサービスは、さまざまなプラットフォームやフレームワークで稼働するソフトウェア・アプリケーションを相互運用するための標準的な手法です。具体的に言えば、Webサービスは、Extensible Markup Language（XML）を使用してHyper Text Transfer Protocol（HTTP）経由で他のシステムと相互運用するようにアプリケーションを拡張するための最善で標準的な方法です。Webサービスにより、それぞれのアプリケーションの相互処理に伴う複雑性を意識することなく、複雑な運用を行うことができます。

Webサービスは単に、アプリケーションをデプロイするために使用するテクノロジーの集合体ではありません。ビジネス機能からサービスを切り出して、それをクライアント・アプリケーションが利用できるよう公開するための手法なのです。

なぜRESTful Webサービスか

Java EE 6では、RESTに関するすべてのメカニズムがJSR-311「Java API for RESTful Web Services」（JAX-RS）によってネイティブ・サポートされています。Webサービスをシステムに導入するためにRESTを選択すべき理由は多くあります。いくつかの例を紹介しましょう。

- もっとも重要な理由は、RESTful Webサービスが学びやすく、構築しやすく、デプロイしやすいことです。
- RESTでは、サービスのプロデューサとコンシューマとの間のインタフェースが統一されます。さらに、RESTful Webサービスはさまざまなメッセージ形式（XML、JSON、HTMLなど）をサポートしています。
- RESTful Webサービスは、機能拡張や新機能の追加のために、既存のアプリケーションに容易に統合できます。

RESTはアーキテクチャのスタイルであり、テクノロジーではありません。JavaでのREST実装方法を示すJSR- 311という仕様があるのはこのためです。JSR-311については、これまでにいくつかの実装が公開されています。Jerseyは公式のリファレンス実装であり、開発環境と本番環境においてもっとも広く利用されています。Jerseyはオープンソースで、オラクルによってサポートされています。
ただし、「大規模な」Webサービスの場合、RESTに制約が伴うことがあります。この制約については、シリーズの第3回で取り上げます。

前提条件

この記事で説明するアプリケーションの開発には、次のソフトウェアを使用しました。

- NetBeans IDE：ここからダウンロードできます。
- Jersey（NetBeansに含まれる）：RESTful Webサービスを構築するための、本番品質かつオープンソースのJava EE仕様（JAX-RS JSR-311）リファレンス実装です。

注：この記事のテストにはNetBeans IDE 6.9.1を使用しましたが、執筆時点での最新バージョンはNetBeans IDE 7.0です。

現実的なアプリケーション

この実践的なセクションでは、日常で目にする現実的なアプリケーションを段階的に構築しながら、RESTful Webサービスの基本を短時間で学ぶことができます。

取り上げるアプリケーションは、eBayのようなオンラインのオークション・サイトです。売主はリストに商品を出品し、買主はその商品に入札します。売主は1つ以上の商品を出品でき、買主は1つ以上の商品に入札できます。ここではシンプルに、Seller（売主）、Item（商品）、Bid（入札）という3つのエンティティについて考えます。

全体のソース・コードはここか

写真：
Allen Mcinnis / Getty Images

らもダウンロードできます。
では、NetBeansとJavaServer Facesを使用してこのアプリケーションを5分でコーディングしましょう。

1. 初期状態のNetBeansプロジェクトを生成します。
   a. NetBeansを起動し、新しいプロジェクトを作成します。
   b. Fileメニューで「New Project」を選択します。
   c. Categoriesで「Java Web」を選択します。
   d. Projectsで「Web Application」を選択します。
   e. 「Next」をクリックします。
   f. プロジェクト名としてAuctionAppと入力して、「Next」をクリックします。
   g. ServerがGlassFish Server（またはこれに類似した名称）となっていることを確認します。
   h. 「Finish」をクリックします。
   シンプルなindex.xhtmlを持つAuctionAppが作成されます。
2. プロジェクトを右クリックして、「Run」を選択します。
   デフォルト・ページ（図1）が開き、シンプルな「Hello from Facelets」というメッセージが表示されます。
3. エンティティを作成します。
   a. AuctionAppプロジェクトを右クリックし、「New」を選択して、「Entity class」を選択します。
   b. Class Nameフィールドに**Seller**と入力し、Packageフィールドに**com.bonbhel.oracle.auctionApp**と入力して、「Next」をクリックします。

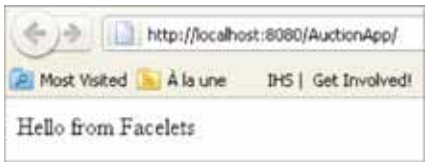


図1

   c. Provider and Databaseで、「EclipseLink (JPA 2.0)」（デフォルト）を選択します。
   d. NetBeansに用意されているデータソースの1つを選択します。「Finish」をクリックします。
   e. 手順3をそれぞれのエンティティに対して行います。NetBeansによって、**Seller.java**、**Item.java**、**Bid.java**の各ファイルが生成されます。

次に、NetBeansウィザードを使用してエンティティにプロパティを追加します。

1. Seller.javaファイルを開き、コード内の任意の場所を右クリックして、「Insert code」を選択します。
2. 「Add property」を選択して、売主のプロパティ（String firstName、String lastName、String email）を追加します。
3. **Item.java**ファイルを開き、商品のプロパティ（String title、String description、Double initialPrice、Seller seller）を追加します。
4. NetBeansの警告をクリックして、エンティティ関係（双方向のManyToOne）を定義します。
   この操作によって、Sellerエンティティにitemのリストが作成されます。
5. **Bid.java**ファイルを開き、商品のプロパティ（String bidderName、Double amount、item）を追加します。
6. NetBeansの警告をクリックして、エンティティ関係（双方向のManyToOne）を定義します。
   この操作によって、Itemエンティティにbidのリストが作成されます。
7. SellerエンティティとItemエンティティに作成されたitemのリストとbidのリストのそれぞれについて、GetterとSetterを生成します。
   この時点で、**Seller.java**ファイルは**リスト1**のようになります。
8. 初期状態のNetBeansプロジェクトにRESTful機能を追加します。

リスト1 **リスト2**

```
@Entity
public class Seller implements Serializable {
  @OneToMany(mappedBy = "seller")
  private List<Item> items;
  private static final long serialVersionUID = 1L;
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  protected String firstName;
  protected String lastName;
  protected String email;

  public List<Item> getItems() {
    return items;
  }

  public void setItems(List<Item> items) {
    this.items = items;
  }
```

リスト全体をテキストで表示する

   a. AuctionAppプロジェクトを右クリックし、「New」を選択して、Entity Classesで「RESTful Web Services」を選択します。
   b. Entity Classesで「Add all」をクリックし、「Next」をクリックします。
   c. 次の2つの名前を指定する必要があります。Resource Packageには**com.bonbhel.oracle.auctionApp.resource**などの名前を、Converter Packageには**com.bonbhel.oracle.auctionApp.converter**などの名前を付けます。
   d. 「Finish」をクリックします。

新しいリソース・クラス（JAX-RSアノテーションを使用してエンティティ表現を定義）とコンバータ・クラス（**@XmlElement**や**@XmlAttribute**などのJAXBアノテーションを使用してデータのマーシャリング/アンマーシャリングの方法を定義）がプロジェクトに追加されます。これでAuctionAppにRESTful機能が追加されました。
リスト2のBidConverter.javaコードについて見てみましょう。
NetBeansによって、GETとPUTのアノテーションを使用するメソッドが生成されています。これらのメソッドは、XML形式のIDで識別されるBidインスタンスの取得や更新を行います。
次の点に注目してください。

- **@Produces({"application/xml", "application/json"})**アノテーションはJAX-RSに、リソースで生成可能な表現タイプとしてXML形式とJSONを指定することを許可しています。
- **@Consumes({"application/xml", "application/json"})**アノテーションはJAX-RSに、リソースで利用可能な表現タイプとしてXML形式とJSONを指定することを許可しています。

リスト3のBidConverter.javaコード

について見てみましょう。
NetBeansによって、XmlElementアノテーションを使用するGetterメソッドが生成されています。このメソッドは、Bidエンティティのフィールドを XML形式にマップします。

- **@XmlElement** アノテーションはJAXBに、**id** をXMLドキュメントの **<id>** エレメントにマップするよう指示しています。
- **@XmlAttribute** アノテーションはJAXBに、**uri** をXMLドキュメントの **uri** 属性にマップするよう指示しています。

図2のように、ブラウザに

図2

図3

図4

http://localhost:8080/AuctionApp/resources/bidsというURLを入力すると、リソースのXML表現を確認できます。この結果が表示されれば、RESTfulサービスが正常に動作していることになります。

ただし、正確にテストするには、オラクルのJavaServer Facesを使用する必要があります。では、データを作成するためのJavaServer Facesフロントエンドをすばやく構築しましょう。

1. JavaServer Facesページを作成します。
   a. AuctionAppプロジェクトのWeb Pagesノードを右クリックし、**「New」** を選択して、**「Folder」** を選択します。
   b. Folder Nameにuiなどと入力します。**「Finish」** をクリックします。
   c. AuctionAppプロジェクトを右クリックし、**「New」** を選択して、Entity Classesで **「JSF Pages」** を選択します。
   d. Entity Classesで **「Add all」** をクリックし、**「Next」** をクリックします。
   e. Session Bean Packageには **com.bonbhel.oracle.auctionApp.facade** などの名前を、JSF Classes Packageには **com.bonbhel.oracle.auctionApp.presentation** などの名前を入力します。
   f. **「Browse」** をクリックし、JSFページ・フォルダとして **ui** フォルダを選択します。
   g. **「Finish」** をクリックします。
2. AuctionAppを実行しデータを追加します。
   a. AuctionAppプロジェクトを右クリックして、**「Run」** を選択します。

ホーム・ページが表示され、エンティティの作成、更新、削除のためのリンクが示されます（**図3**）。

   b. ホーム・ページで **「Show All Seller Items」** をクリックし、**「Create New Seller」** をクリックして、新しい売主を追加します（**図4**）。
3. RESTful Webサービスが利用可能かどうかをテストします。
   a. ブラウザを開き、リソースのURLである http://localhost:8080/AuctionApp/resources/sellers を入力します。

リスト4のように、リソース（Seller）のXML表現が表示されます。すでに作成したすべてのSellerをXML形式で参照できます。

リスト3 **リスト4**

```
@XmlElement
public Long getId() {
  return (expandLevel > 0) ? entity.getId() : null;
}
@XmlAttribute
public URI getUri() {
  return uri;
}
```

リスト全体をテキストで表示する

まとめ

RESTful Webサービスや、JavaServer Facesベースのインタフェースを扱うJava EEアプリケーションをNetBeansですばやく開発する方法について学びました。RESTとJerseyを統合することで、Webサービスで機能の再利用性を確保し、システム全体の柔軟性を高められることを確認しました。
第2回ではWebサービスとJSONフレームワークについて、第3回ではJAX-WS Webサービスについて説明します。●

参考情報

- Jerseyプロジェクトのホーム・ページ
- Roy T. Fielding氏のRESTアーキテクチャ・スタイル定義に関する論文
- JavaServer Faces
- この記事で使用されたソース・コードのダウンロード

お待たせしました! Java 7の登場

オラクルの**Mark Reinhold**がJava Magazineに語る、Java SE 7の最重要機能。

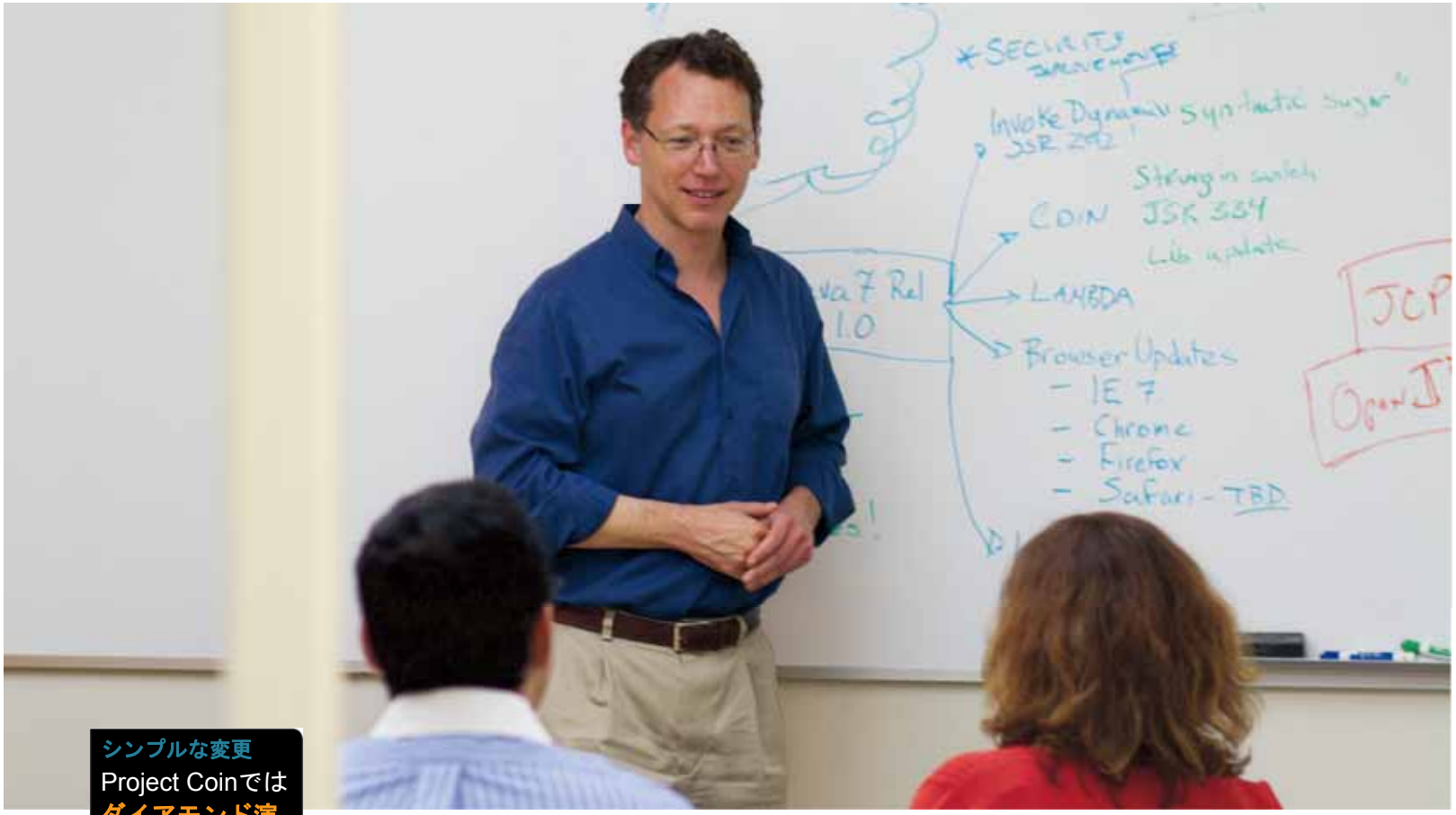
Michael Meloan

Java Platform, Standard Edition（Java SE）は、汎用コンピューティング向けの中核となるJavaプラットフォームです。Java SE 7リリースは数々の重要分野に対応しており、プログラミング・コミュニティのトレンドや、ハードウェア・アーキテクチャでの開発、Javaテクノロジーの成功に対する継続的なコミットメントが反映されています。Java SE 7（Java 7）により、広範囲のコンピューティング環境にわたる保守性とスケーラビリティに優れた高パフォーマンスのJavaアプリケーションを構築できます。オラクルのJava Platformグループでチーフ・アーキテクトを務めるMark Reinholdが、Java SE 7リリースのおもな機能と開発者にとってのメリットについて語ります。

Java Magazine：開発者、システム・アーキテクト、そしてJavaエンタープライズ全体にとってもっとも重要なJava SE 7の側面とは何でしょうか。

Reinhold：このリリースには4つの柱があります。Project Coin（JSR-334、小規模な拡張機能セット）、新しい**invokedynamic**バイトコード命令（JSR-292）、New I/O Part 2（JSR-203）、フォーク/ジョイン・フレームワークです。この4つのそれぞれで、開発者にとって新しく、また価値のあるオプションが用意されています。

写真：Bob Adler
画像：I-Hua Chen



Java SE 7の新機能についてチーム・メンバーと議論するMark Reinhold（オラクルJava Platformグループ、チーフ・アーキテクト）

シンプルな変更

Project Coinでは**ダイアモンド演算子**や**try-with-resources構造**などの機能によって日常的なプログラミング・タスクを簡素化する

Project Coinの使命は、日常的なプログラミング・タスクをより簡単にすることでした。この構想を指揮したのはオラクルのJoe Darcyです。彼は2年ほど前、この構想をOpenJDKプロジェクトとして開始しました。オラクル社外からのものも多く含まれる数々の要望に優先順位を付け、優れた改善策を考え出しました。1つは*ダイアモンド演算子*<>です。これは、ジェネリクス（総称）型のインスタンスを型に持つオブジェクトの生成時に使用するものです。Project Coinができる前は、代入文の左辺と右辺の両方で、山括弧の中に完全なジェネリクス型を記述する必要がありました。Java 7では左辺だけに完全な型を記述し、右辺には空の型パラメータ<>を使用するだけで済みます。右辺の型はコンパイラによって自動的に判定されます。

たとえば、以下のようなコードがあるとします。

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

これは、次のように簡潔に記述できます。

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

Project Coinのもう1つの価値ある機能が、try-with-resources構造です。これにより、当初からJava APIに潜んでいた正確性の問題が解決します。ソケット、フレーム・バッファ、ファイル記述子などの外部リソースをAPIで割り当てる際は、これらの限りあるリソースを再利用できるように、アプリケーションでリソースの解放やクローズを正しく行う必要があります。Project Coinができる前は、try-catchブロックを慎重に適用することで対応していました。しかし、複数のリソースを正しく処理するには、try-catchブロックを複数のレベルでネストする必要があります。これを問題なく行うのは容易ではなく、間違いも多く見られます。

try-with-resources機能は、この既存のtry構文を拡張したものです。たとえば、tryブロックの最初でリソースを作成して、その本文内では通常どおりリソースを使用できます。

Moving Java Forward:A Video Discussion About Java 7

catch句を記述する必要はありません。ブロックを抜けるときに、リソースを正しくクローズするために必要となるすべてのロジックがコンパイラによって生成されるためです。

リスト1では、try-with-resources構造を使用して、java.sql.Statementオブジェクトを自動的にクローズしています。

Project Coinでは、switch文で文字列定数を使用するオプションも用意されています。これも、コーディングを容易にするために役立ちます。Project Coinができる前は、switch文では整数値か列挙型（enum）のインスタンスしか使用できませんでした。

Java Magazine：非同期I/O処理のサポートが、New I/O Part 2（「NIO.2」）のおもな機能なのでしょうか。

Reinhold：そのとおりです。非同期I/O APIは、ある種のハイエンド・サーバー・アプリケーションや、大量のI/Oスループットを必要とするその他のソフトウェアにとって非常に便利なものになります。Java SE 7では、完全なファイル・システムAPIも用意されています。Javaプラットフォームにはこれまで、ファイル・システム操作のサポートにおいて何らかの制約がありました。たとえば、シンボリック・リンクの作成、ファイル権限のチェック、ファイル更新時のコールバックのリクエストといったシンプルな操作は、今日のオペレーティング・システムでは非常に一般的な機能です。これらすべてが、標準に含まれるAPIで利用できるようになりました。

Javaが誕生して間もなくは、プラットフォームに依存しないことがもっとも重要な目標でした。これは多くの場合に妥当ですが、ときには問題となることもありました。従来のファイル・システムAPIに原始的な部分があったのはこういった理由によります。この新しいファイル・システムAPIでは、プラットフォーム固有の機能が公開されています。このAPIは玉ねぎのようです。最初の層には、どのような環境でも同じように動作するプラットフォーム非依存の処理が定義されています。しかし、その層の下には、プラットフォーム固有の機能があります。たとえば、POSIXアクセス制御リストを確認する必要がある場合、POSIX環境では、玉ねぎの皮を1枚むくようにしてその機能のサポートを利用できます。Windows環境に固有の機能についても、同じように対応できます。

Java Magazine：今日、マルチコア・プロセッサが標準となってきています。そのメリットを得るために、フォーク/ジョイン・フレームワークをどう活用できるでしょうか。

Reinhold：フォーク/ジョインAPIは、より小さな単位に再帰的に分割できる問題を取り上げ、その解決に必要な作業を任意数のプロセッサ・コア全体に分散するための、非常に簡単な方法です。分割統治戦略を適用できる問題には、フォーク/ジョインがぴったりと合うケースが多いのです。プログラマーに代わって同時実行に関するあらゆる事項に対応してくれます。

Java Magazine：フォーク/ジョインが特に役に立つアプリケーション環境について教えてください。

Reinhold：イメージ処理が良い例です。従来のイメージ処理アルゴリズムの多くは、非常に自然に、再帰的に分割されます。イメージはセグメントに分けられ、フォーク/ジョイン・タスクがこれらのセグメントの処理に割り当てられます。また、大きな配列の計算も良い例となります。問題が簡単に再帰的に分割されて、下位の計算が配列の各部分だけに対して同時に行われます。その後、結果が再び1つの値またはベクターに集約されます。

Java Magazine：invokedynamicバイトコード命令は、開発者にとってどのようなメリットがありますか。

Reinhold：invokedynamic命令は、現時点で定義されているJava言語では、有用とはいえません。invokedynamicのおもな目的は、動的言語をJavaバイトコードに容易にコン

リスト1

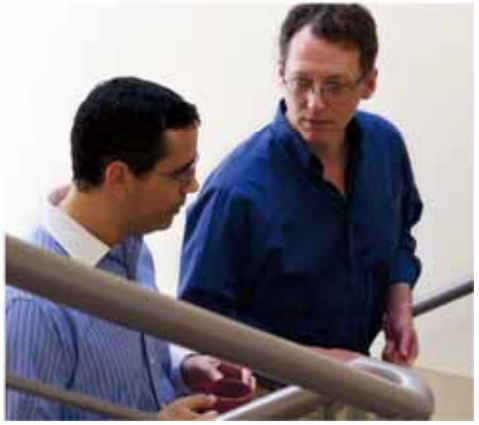
```
public static void viewTable(Connection con) throws SQLException {
  String query = "select COF_NAME, PRICE from COFFEES";
  try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
      String coffeeName = rs.getString("COF_NAME");

      float price = rs.getFloat("PRICE");

      System.out.println(coffeeName + ": " + price);
    }

  }
}
```

リストをテキストで表示する



Java SE 7の最終テストとリリース計画について話すMark ReinholdとJava SEプロダクト・マネージャー

パイルできるようにすることです。これによって、Rubyなどの言語をJava仮想マシン（JVM）上に実装できます。invokedynamicが解決する問題は、正確には動的型付けではなく、動的メソッド・ディスパッチです。Ruby、Smalltalk、JavaScriptなどの言語には、ランタイム情報に依存するメソッド・ディスパッチのパターンが存在します。Javaメソッド・ディスパッチにはないパターンです。invokedynamic命令を使用すると、こうしたディスパッチをバイトコード・レベルで表現できるようになります。しかもこれが、JVMで通常のJavaメソッド・ディスパッチ・パターンが最適化されるのと同じくらい効率的に最適化されます。パフォーマンスの観点では、動的言語の実行パフォーマンスが劇的に改善される見込みです。私たちはJRuby開発リーダーのCharles Nutter氏と協力してきました。Nutter氏はinvokedynamicの設計と実装に大きな影響を及ぼしています。現在ではinvokedynamicをJRubyで使用し、すばらしい成果を上げています。

Java Magazine：invokedynamicによって言語間の相互運用が容易になるとのことですが、開発者がJavaとRubyによるハイブリッド・アプリケーションを構築する典型的なシナリオにはどのようなものがありますか。

Reinhold：Rubyなどの動的言語を使用すると、ある種のアプリケーションで開発者の生産性を大幅に向上することができます。多くの開発者は、WebフロントエンドをRubyでコーディングする方がより生産的だと考えています。そのため、たとえばRubyフロントエンドを、複雑なビジネス・ロジックを処理するJavaバックエンドと連携させれば、これは非常に強力な組み合わせとなるでしょう。

他の例としては、IT部門によって、アプリケーション・サーバーへのデプロイ対象がJava Platform, Enterprise Edition（Java EE）コンポーネントに制限されている場合が考えられます。このような環境では、開発者には通常のRubyインタプリタを使用するという選択肢はありません。しかし、JRubyであればRuby WebアプリケーションとともにWARファイル内にバンドルできます。JRubyは、内部的にはRubyランタイムとRubyコードが含まれているにもかかわらず、1つのJavaアプリケーションのように動作します。このため、このような制約に直面する開発者がより柔軟に対処できます。

Java Magazine：invokedynamicは将来のJavaリリースで利用されますか。

Reinhold：その可能性はかなり高いです。Project Lambdaでは、Java SE 8リリースでJavaにクロージャを追加することを計画しています。現在のプロトタイプ実装では、ラムダ式を実装する効率的な方法としてinvokedynamicを使用しています。invokedynamicは元々、他の言語のコンパイルと実行を改善するという問題を対象にしていましたが、現在では長期的に見てJava自体にも役に立つだろうと見られています。

Java Magazine：社外コントリビュータにとってこのリリースはどれほど重要なのでしょうか。

Reinhold：とても重要です。Sunは2007年にJDKをオープンソース化しましたが、その後もオラクルは社外コントリビュータ（貢献者）の参加を重視し続けました。フォーク/ジョイン・フレームワークは、ニューヨーク州立大学機構（SUNY）オスウェゴ大学のDoug Lea教授によって開発されました。また、新しいサウンド・シンセサイザはオープンソース開発者のKarl Helgason氏が提供しています。新しいJava 2Dグラフィック・パイプラインは、数年前のOpenJDK Innovators' Challengeを受賞したClemens Eisserer氏が記述したものです。社外からこのような高度な技術を持つ方々に参加していただけるのはすばらしいことであり、Javaエコシステム全体により強力なイノベーションをもたらしています。

Java Magazine：Java SE 7リリースについて最後に伝えたいことはありますか。

Reinhold：はい。ぜひダウンロードして、楽しんでご利用ください。●

Michael Meloan：IBMメインフレームとDEC PDP-11のアセンブリ言語のコーディングに専門的に従事した後、PL/I、APL、C、Javaのコーディング分野でもプロフェッショナルとしての経験を持つ。また、小説がWIRED、BUZZ、Chic、L.A. Weeklyに掲載され、National Public Radioでも放送されている。Huffington Postブロガーでもある。

参考情報

- JDK 7 Features
- JDK 7プレビュー・リリースのダウンロード
- Project Coin
- JSR-334:「Small Enhancements to the Java Programming Language」
- JSR-292:「Supporting Dynamically Typed Languages on the Java Platform」
- JSR-203:「More New I/O APIs for the Java Platform ("NIO.2")」
- Fork/Join Framework（PDF）
- Mark Reinholdのブログ
- Joe DarcyのOracle Weblog

JDK 7ーコーディング方法の変革

JDK 7の新機能を利用して、エラーを削減し、難しく面倒なタスクを容易にする

Herb Schildt

最初のバージョンであるJava 1.0から、Java言語についてさまざまな記事を執筆してきましたが、その中で私はJavaがメジャー・リリースごとに進化し、成熟し、成長する姿を見てきました。新しいバージョンが発表されるたびに、言語を使用するプログラマーのニーズや要望に対応する機能が追加されました。この継続的な改良と適用のプロセスによって、Javaは、もっとも広く使用されているプログラミング言語という地位を不動のものとし、また、常にフレッシュで強力な生きた言語であり続けています。

Javaの進化についていつもおもしろいと感じるのは、新機能が定着する、つまり完全に主流になるまでにかなりの時間がかかっている点です。もっとも良い例は、JDK 5で追加されたジェネリクス（総称型）です。ジェネリクスによってJavaの能力とプログラムの信頼性は根本的に拡大されました。さらに、完全に新しい構文要素や、Javaのコーディングに関する新しい概念が導入されました。ジェネリクスの導入は非常に大きな変更だったため、この機能が一般的に使用されるようになるまでやや時間がかかりました。

JDK 7のリリースでも、Javaはさらに進化を遂げ、プログラマーのニーズに対応しようとしています。そして、Javaの能力とスコープを拡大する新機能がまたしても追加されました。しかし、なかなか定着しなかった過去の追加機能とは異なり、JDK 7の言語上の新機能はこれまでのコーディング方法を変えるものではありません。

ご存じかもしれませんが、JDK 7の言語上の新機能は*Project Coin*によって開発されました。Project Coinの目的は、JDK 7に組み込むJava言語の小さな変更点の数々を特定することでした。しかし、興味深いことに、これらの新機能は全体的に「*小さな*」ものとして扱われていますが、影響する*コードの観点*で言えば、これらが及ぼす影響は非常に大きいのです。簡単に言えば、多くのプログラマーにとって、Project Coinの変更点こそがJDK 7でもっとも重要な新機能となります。

小さな変更
多くのプログラマーにとって、Project Coinの変更点がJDK 7で**もっとも重要**な新機能となる

この理由を理解するために、Project Coinの次の新機能および変更点について考えてみましょう。

- **try**-*with-resources*と呼ばれる拡張**try**文で、ファイル・ストリームなどのリソースのクローズ処理が自動化された。
- ジェネリクス・インスタンスの生成時に、（*ダイアモンド演算子*<>を使用して）型推論が行われるようになった。
- 例外処理が拡張され、関連のない2つ以上の例外の型を1つの型で捕捉できるようになり、また、再スローされる例外の型チェックが改善された。
- **switch**文の制御に**文字列**を使用できるようになった。
- 新しい接頭辞**0b**または**0B**を使用したバイナリ整数リテラル（**0b1010**など）を利用できるようになった。
- 数値リテラルでアンダースコア（**59_234_412**など）利用できるようになった（アンダースコアはコンパイラでは無視されるが、長い数値をわかりやすく表記するために役立つ）。
- 具体化できないパラメータを持つ可変引数メソッドに関連するコンパイラ警告が改善され、より制御できるようになった。

これらはいずれも、プログラマーがずっと待ち望んできた機能です。これらすべての機能で、以前は難しかった、あるいは面倒であったタスクが効率化され、シンプルになります。すべての機能が、エラーのない優れたコーディングに役立ちます。誌面の都合上、これらの機能のそれぞれを検討することはできませんが、これらの広範な新機能のうち、両極端とも言える例を見ていくことにしましょう。

数値リテラルにおけるバイナリ・リテラルとアンダースコア

まずは、一見すると小さな変更点と呼ぶにふさわしい2つの機能について見ていきます。バイナリ・リテラルの指定機能、お

よび数値リテラルでのアンダースコアの利用です。最初の印象では、これらの新機能はほとんど重要ではなく、ここで取り上げる価値のないもののように思えますが、実際は非常に重要な変更です。これらの機能は便利なだけではなく、エラーの防止にも役立ちます。たとえば、おそらくはビット・マスクとして使用するために、ある特定のビット・パターンが必要となるような状況について検討しましょう。ビット・パターンについては、だれもがバイナリの観点から考えるのは明らかです。そのため、バイナリ・リテラルを使用してビット・パターンを指定したいところです。問題は、これまではバイナリ・リテラルがなかったということです。このため、別のアプローチが必要であり、複数の方法が使用されていました。

たとえば、ビット・パターン**0110 1101**が必要な場合に、**Byte.parseByte ("01101101", 2)**を使用してきたプログラマーもいるでしょうが、これにはメソッド・コールが伴います。実際のリテラルが必要ならどうなるでしょうか。ありがちなのは、とりあえず16進数リテラルを使用することでした。たとえば、次のような表現はだれもが見たことがあるでしょう。

```
byte myBits = 0x6D; // 0110 1101
```

ここでは、値を16進数リテラル**0x6D**としてエンコードし、コメントでビット・パターンを示しています。しかし、このアプローチは、これ自体では問題は起きませんが、バイナリから16進数に変換する際にミスを犯し、その結果誤ったビット・パターンとなる可能性を秘めています。（プログラマーが誤ったキーを押してしまい、それに気づかないような場合です。）残念ながら、このようなミスによって生じるバグは非常に発見しにくいものです。また、値自体は正しくても、コメントのビット・パターンが誤っており、コードの読み手が判断を誤るおそれもあります。

JDK 7では、ビット・パターンの指定にバイナリ・リテラルを使用できるため、このようなエラーが起きる可能性をなくすことができます。たとえば、次のように使用します。

```
byte myBits = 0b01101101;
```

ここでは、値はバイナリ・リテラルによって直接エンコードされます。このため、変換エラーの可能性はまったくなく、ビット・パターンは記述した値をそのまま示すため明確です。このようにして、プログラマーが求めていたとおり、ビット・パターンをイメージのまま表すことができるようになりました。これは、信頼性と透明性に優れたアプローチです。

また、次のようにアンダースコアを挿入して、バイナリ値の可読性をさらに高めることができます。

```
byte myBits = 0b0110_1101;
```

アンダースコアはこの例でも有効ですが、大きなバイナリ値に使用すると、その価値はさらに高まります。たとえば、次のどちらの値が読みやすいでしょうか。

```
0b0110110111000111
0b0110_1101_1100_0111
```

数値におけるバイナリ・リテラルとアンダースコアの使用は、「小さな」言語上の機能拡張の中でももっとも小さなものですが、これらはいずれもプログラマーのコーディングをわかりやすくし、エラーの頻度を減らす重要な改善点です。

リスト1

```
FileInputStream fIn = null;
try {
 fIn = new FileInputStream("somefilename");
 // Access the file ...
} catch(IOException e) {
 // ...
} finally {
 // Close file.
 try {
  if(fIn != null) fIn.close();
 } catch(IOException e) {
  // ...
 }
}
```

リスト全体をテキストで表示する

try-with-resources文

数値でのバイナリ・リテラルとアンダースコアがJDK 7の変更点の一端を担うとすれば、もう一端を担うのは**try**-with-resourcesです。私は**try**-with-resourcesは、JDK 7の言語上の新機能のうち**もっとも重要**な機能だと考えています。長年の問題を解決するだけではなく、クラス全体に及ぶエラーを防ぐものでもあるからです。ファイル・ストリームなどのリソースを扱う処理でもっとも悩ましいのは、不要となったリソースを確実にクローズすることです。リソースのクローズを忘れると、メモリ・リークなどの問題につながります。**try**-with-resources文を使用すれば、クローズ処理を自動化できます。しかもそれを非常に明快に行うことができるのです。

try-with-resourcesの重要性を理解するため、これが設計された理由となった、改善すべき状況の例について確認しましょう。ご存じのとおり、従来のファイル操作には3つのアクションが必要でした。ファイルのオープン、ファイルの利用、ファイルのクローズです。JDK 7より前のバージョンで、**リスト1**のようなコードを使用したことがあるでしょう。

ファイル・ストリームを**finally**ブロックでクローズしていることに注目してください。このブロックは、**try**ブロックを抜けた後、すべてのケースで自動的に実行されます。この例では、**fIn**には最初に**null**が代入されています。次に、**try**ブロックに入ります。**fIn**のオープンに成功した場合は、**fIn**に**null**以外の値が代入されます。ファイルのオープン時にエラーが発生した場合は、**fIn**は**null**のままです。**try**ブロックが（正常に、または例外により）終了すると、**finally**ブロックが実行されます。このとき、**fIn**が**null**でない場合は、ファイルがオープンされているためクローズする必要があります。**fIn**が**null**の場合は、それまでにエラーが発生しています。この場合、**close()**のコールは実行されません。**close()**でも例外が発生する可

能性があるため、ここでもまた、独自のtryブロックでラップしています。もちろん、このシーケンスには、コール元のルーチンに例外をスローするなどの多くのバリエーションがありますが、どのような実装であれ、ファイルをfinallyブロックで明示的にクローズしなければなりません。

正確に（かつ一貫性を保って）使用すれば、flnに関連付けられたファイルはこのシーケンスによって適切にクローズされます。もちろん、このアプローチには問題もあります。まず、ファイルのクローズを忘れる可能性がなくなったわけではありません。たとえば、同じtryブロック内で複数のファイルにアクセスする場合、プログラマーがfinallyブロックで1つのクローズをうっかり忘れるかもしれません。また、ファイルのクローズを妨げるようなコーディング・エラーを犯す可能性もあります。私の経験をもとに、簡単な例を紹介します。

先日、書籍で例として使用するコードを記述していたときのことです。前述の例に似たシーケンスを使用していたところ、finallyブロックでファイルをクローズする際に、次のような行を記述してしまいました。

```
if(fln == null) fln.close();
```

この問題がおわかりでしょうか。if文で!=を使用する代わりに、うっかり==とタイプしてしまったのです。このコードでは、ファイルがオープンしていないときのみ、ファイルのクローズが実行されることになります。

幸いにも私はこのミスに気づきましたが、気づいていなければ、発見しにくいバグにつながるものでした。このエラーは、ファイルのオープンに失敗したときにしか明らかにならないためです。（ファイルのオープンに失敗すると、flnがnullのままになるため、close()のコール時にNull Pointer Exceptionが発生します。）しかし、このとき作成していた例では、この状況が起きるのは非常にまれでした。ファイルのオープンは通常成功し、エラーの兆候は表れませんが、ファイルに関連付けられたリソースは解放されません。つまり、うっかり!=の代わりに==と記述すると、リソース・リークが発生するにもかかわらず、コード・シーケンス全体は「問題がないように見える」のです。しかし幸いなことに、JDK 7ではエラーの原因となるこのようなコードは過去のものとなります。

ぜひお試しください

try-with-resourcesは強力で重大な言語上の追加機能である。**本当に、それほど重要なもの**なのだ。今すぐお試しください。

try-with-resources文は2つの機能を実行します。1つ目に、ファイル・ストリームなどのリソースを宣言して初期化します。2つ目に、tryブロックの終了時に、リソースを自動的にクローズします。つまり、ファイル・ストリームの場合は、ファイルが自動的にクローズされます。明示的にclose()をコールする必要はもうありません。

リスト2では、try-with-resourcesを使用して前述のシーケンスを書き直しています。flnの宣言と初期化を、別のステップではなくtry文の中で行っている点に注目してください。

リスト2のコードは大幅に短縮化されているだけではなく、あらゆるケースでflnが確実にクローズされます。tryブロックを（正常に、または例外により）抜けると、flnが自動的にクローズされます。クローズし忘れることはありません。プログラミングのミスによってクローズされないという事態は起こりえないのです。

ここまでを読んで、こんな疑問が思い浮かんだかもしれません。I/O例外がtryブロックの内側で発生し、ファイル・ストリームが自動的にクローズされるときにさらにI/O例外が発生すれば、これら2つの例外はどうなるのでしょうか。このような場合は、tryブロック内で発生した例外がスローされ、クローズ処理での例外は、表示されない（suppressed）例外のリストに追加されます。このリストは、Throwableに定義されているgetSuppressed()メソッドをコールして取得できます。これが、try-with-resourcesのもう1つの利点です。

さらに他にも利点があります。try-with-resourcesは新しいAutoCloseableインタフェースを実装するあらゆるリソースを管理できます。つまり、単にファイル・ストリームのためだけの機能ではありません。try-with-resourcesはコードを整理し、リソース・リークを防ぎ、そのプロセスの中でコードの復元力を高めるので、この機能には批判の余地はほとんどありません。強力で重大な言語上の追加機能といえます。本当に、それほど重要なものなのです。私の考えでは、try-with-resourcesはすべてのJavaプログラマーが今すぐ使用したいと思う機能のはずです。

リスト2

```
try(FileInputStream fln = new FileInputStream("somefilename")) {
  // Access the file ...
} catch(IOException e) {
  // ...
}
```

リスト全体をテキストで表示する

まとめ

もちろん、私はJDK 7での他の言語上の新機能についても、その良さを確信しています。たとえば、ダイアモンド演算子による型推論では、ジェネリクス・インスタンスを作成するための構文がシンプルになります。また、1つのcatch文で複数の例外をキャッチできる機能では、コードを短くすることができます。switchで文字列を使用できる機能は、これまでの長年の要望に応えるものです。（いつの時代でも、文字列でswitchを制御できる機能はだれもが求めるものでしょう。）これらすべてが合わさったJDK 7の言語上の新機能はJavaに本当の価値をもたらし、皆さんのプログラマー生活を少し楽にしてくれることでしょう。簡潔に言えば、これらの機能はあまりにも便利なので、無視することなどできないのです。●

参考情報

- Project Coin
- Project Coinメーリングリスト・アーカイブ
- Project Coinに関するブログ
- Project Coin:JSR-334 in Public Review
- JSR-334のドキュメントとパブリック・レビュー

Raymond Gallardo

動的型付け言語とinvokedynamic命令

新しいinvokedynamic命令では、固定的なリンケージ処理を使用せずに、ランタイム・システムでコール・サイトとメソッド実装とのリンケージをカスタマイズできます。

SunがJavaプラットフォームを開発していたときは、開発者がJava仮想マシン（JVM）でJava以外のプログラミング言語を実行することは奨励されていませんでした。実際のところ、Sunは、開発者がエンタープライズ開発のおもなプログラミング言語としてJavaを使用することを望んでいました。それから10年が経過し、Java Platform, Standard Edition 7（Java SE 7）のリリースを間近に控えた今、この目標はすでに達成されています。

開発者はJavaプラットフォームや、開発媒体としてのJVMに特別の価値があることを認めています。しかし、Javaプラットフォームがいつも最良のツールとなるわけではないこともわかっています。新旧のさまざまな言語が、JVMで動作する言語として再実装されています。たとえばRubyはJRuby、PythonはJPython（現在はJython）として実装されました。

なぜ開発者は元のCの同等機能を使用せず、わざわざこれらの言語を再実装しているのでしょうか。古いコードをJavaに変換することもできたはずです。しかし、JRubyやJPythonなどの実装では、次のようなJVM固有の機能を活用できます。

- 高度な最適化
- クロスプラットフォーム互換性/移植性
- オープンソース
- 標準Javaライブラリへのアクセス
- マルチスレッド
- ガベージ・コレクション
- 複数のプロセッサを利用する大規模なエンタープライズ・システムでの優れたパフォーマンス
- 広範な利用者と大規模なインストール・ベース

JVM向けに言語を実装する難しさは、JVMがJava用に作成されたことにあります。つまり、再実装する言語にJavaとは異なるオブジェクト/メソッド解決メカニズムが含まれていると、JVMがうまく動作しないことがあるのです。特に、JVM上でのRubyやPythonの実装を難しくしているのは、これらが動的型付け言語であるのに対し、Javaは静的型付け言語であるという点です。これを解決するのが、新しいバイトコード命令である**invokedynamic**です。この命令を使用すれば、JVM上での動的型付け言語のコンパイラおよびランタイム・システムの実装がシンプルになり、改善されます。

静的型付け言語と動的型付け言語の違い

Javaは静的型付け言語です。これは、コンパイル時に型チェックを実行することを意味します。型チェックとは、プログラムの型保証（type-safe）を検証するプロセスです。すべての操作の引数が正しい型であれば、プログラムは型保証されています。

RubyとPythonは動的型付け言語です。これは、実行時に型チェックを実行することを意味します。これらの言語では通常、コンパイル時に型情報を取得できません。そのため、オブジェクトの型を判別できるのは実行時のみです。

静的型付け言語は必ずしも強い型付け言語ではない

強い型付けを特徴とするプログラミング言語では、演算に渡せる値の型に制約があります。Javaのように強い型付けを実装している言語では、演算の引数の型が誤っている場合は演算を実行できません。逆に、弱い型付けを特徴とする言語では、引数の型が誤っている場合や引数の型に互換性がない場合に、演算の引数が暗黙的に変換（キャスト）されます。

静的型付けプログラミング言語には、強い型付けでも弱い型付けでも適用できます。同様に動的型付け言語にも、強い型付けでも弱い型付けでも適用できます。たとえば、Rubyは動的型付けかつ強い型付けの言語です。Rubyでは、変数がいずれかの型の値で初期化された後は、その変数が他のデータ型に暗黙的に変換されることはありません。たとえば、Rubyでは次のような式は許可されません。

写真：
Geneviève Arboit

```
a = "40"
b = a + 2
```

Rubyでは、この例でFixnum型の数値2が文字列に暗黙的にキャストされることはありません。

動的型付け言語のコンパイルの問題

仮想のプログラミング言語による次のような動的型付けメソッドaddtwoについて考えてみます。このメソッドは、2つの数値（任意の数値型）を加算し、その合計を返します。

```
def addtwo(a, b)
  a + b;
end
```

今、このプログラミング言語のコンパイラおよびランタイム・システムを導入しており、プログラムには**addtwo**メソッドが記述されているとしましょう。強い型付け言語では、静的型付けでも動的型付けでも、**+**（加算演算子）の動作はオペランドの型によって変わります。

静的型付け言語のコンパイラでは、**a**および**b**の静的な型に基づいて、どのような**+**の実装が適しているかが選択されます。たとえば、Javaコンパイラでは、**a**および**b**の型が**int**の場合に、**+**はJVMの**iadd**命令を使用して実装されます。この加算演算子はメソッド・コールとしてコンパイルされます。JVMの**iadd**命令ではオペランド型を静的に判別しておく必要があるからです。

これとは対照的に、動的型付け言語のコンパイラではこの判断を実行時に行う必要があります。**a + b**という文は、メソッド・コール**+(a, b)**としてコンパイルされます。**+**はメソッド名です。（**+**という名前のメソッドはJVMでは許可されていますが、Javaプログラミング言語では許可されていません。）ここで、動的型付け言語のランタイム・システムで**a**と**b**が**int**型の変数だと識別できるとすれば、ランタイム・システムでコールする**+**の実装は、任意のオブジェクト型ではなく整数型に特化している方が望ましいでしょう。

動的型付け言語のコンパイルの問題は、ランタイム・システムをどう実装すれば、プログラムのコンパイル後、メソッドまたは関数の最適な実装を選択できるのかという点にあります。すべての変数を**Object**型のオブジェクトとして扱うのは非効率です。**Object**クラスには**+**という名前のメソッドは含まれないからです。

新バイトコード
Invokedynamicを使用すると、コンパイラおよびランタイム・システムの実装がシンプルになり、改善される

invokedynamic命令

Java SE 7では**invokedynamic**命令が導入されました。この命令を使用すると、ランタイム・システムでコール・サイトとメソッド実装とのリンケージをカスタマイズできます。これは、**invokevirtual**のように、Javaクラスおよびインタフェースに固有のリンケージ処理がJVMによって固定的に設定されるJVM命令とは対照的です。

前述の**addtwo**の例では、**invokedynamic**のコール・サイトは**+**です。ブートストラップ・メソッドによって、**invokedynamic**のコール・サイトがメソッドにリンクされます。*ブートストラップ・メソッド*とは、コンパイラが動的型付け言語向けに指定するメソッドであり、コール・サイトをリンクするためにJVMによって1度だけコールされます。ブートストラップ・メソッドから返されたオブジェクトにより、コール・サイトの動作が永続的に規定されます。

リスト1

```
invokedynamic  InvokeDynamic
 REF_invokeStatic:
  Example.mybsm:
   "(Ljava/lang/invoke/MethodHandles/Lookup;
    Ljava/lang/String;
    Ljava/lang/invoke/MethodType;)
   Ljava/lang/invoke/CallSite;":
  +:
   "(Ljava/lang/Integer;
    Ljava/lang/Integer;)
   Ljava/lang/Integer;";
```

リストをテキストで表示する

リスト1は、**invokedynamic**命令を使用した例です。この例では、ASM Javaバイトコード操作/分析フレームワークの構文が使用されており、わかりやすいように改行を加えています。

この例では、ランタイム・システムにより、**invokedynamic**命令で指定された動的コール・サイト（加算演算子の**+**）が、**IntegerOps.adder**メソッドにリンクされます。（**IntegerOps**クラスは、この例で実装している動的言語のランタイム・システムに付属するライブラリに属しています。）このリンケージは、ブートストラップ・メソッドの**Example.mybsm**で実行されます。このメソッドは、開発者側で記述する必要があります。

Java SE 7では**java.lang.invoke**パッケージが導入されました。このパッケージには、新規のデータ型**MethodHandle**を含む、ブートストラップ・メソッドの記述に不可欠となるAPIが含まれています。メソッド・ハンドルは、基盤となるメソッド、コンストラクタ、フィールドなどの低レベルの処理を指す、直接実行可能な型付け参照です。**java.lang.invoke**パッケージには、メソッド・ハンドルの作成と操作のための他のメソッドも含まれています。●

参考情報

- Java Virtual Machine Support for Non-Java Languages
- java.lang.invoke Package
- The Da Vinci Machine Project
- oracle.comのJohn Roseのブログ（John RoseはDa Vinci Machine Projectプロジェクト・リーダー兼invokedynamic仕様リーダー）

Oracle Technology Network：あなたのJavaの世界

Javaに関するあらゆるトピックで他のプロフェッショナルと交流できる場所でお待ちしています。

Oracle Technology NetworkはJavaをはじめ、オラクル製品による業界標準テクノロジーを利用する開発者、管理者、アーキテクトが集う世界最大のコミュニティです。メンバー登録（無料）すると、次の情報にアクセスできます。

・ディスカッション・フォーラムとハンズオン・ラボ
・ダウンロード可能なフリー・ソフトウェアとサンプル・コード
・製品ドキュメント
・ メンバーが投稿したコンテンツ

知のグローバル・ネットワークをご活用ください。

今すぐ参加 ▶ oracle.com/technetwork/javaにアクセス

ORACLE

JavaServer Faces 2.0でのAdobe FlexとJavaFXの利用

JSF 2.0の新機能を活用し、Adobe FlexとJavaFXをJSFアプリケーションに統合する

Re Lai

JavaServer Faces（JSF）2.0仕様は、過去6年間にわたりJSF 1.0仕様を利用してきた上での成功と教訓に基づいて策定されています。SeamなどのWebフレームワークから着想を得て、Convention over Configuration（設定より規約）、Annotation over XML（XMLよりアノテーション）などの一般的なアジャイル・プラクティスを取り入れています。この結果、より整備されたフレームワークになっています。特に注目すべき点としては、標準化されたAjaxのサポート、デフォルトのビュー・テクノロジーとしてのFacelets、カスタム複合コンポーネントなどがあります。これらによってようやく、コンポーネント・オーサリングが簡単で楽しいものになります。

この記事では、リッチ・クライアント・アプリケーションを容易に埋込むための新機能の利用方法について確認していきます。Adobe Flexは、人気の高いリッチ・インターネット・アプリケーション・フレームワークです。JavaFXは、Javaプラットフォームの最上位に位置する比較的新しい技術として、多くの注目を浴びてきました。リッチ・クライアントのJava Webアプリケーションへの統合は常に強い関心を集めています。JSF 2.0そのものはもちろん、JSF 2.0がシンプルな開発に力を入れてきたことからも、統合がかつてないほど簡単になってきています。

ここではまず、アイス・クリームのフレーバー人気度調査の結果を表示するサンプルのFlex円グラフ・アプリケーションを見ていきます。JSF複合コンポーネントを使用して、この埋込み部分をカプセル化します。次に、この調査結果はハードコーディングされずに、JSFマネージドBeanからFlexアプリケーションに渡されます。さらに、ユーザーのお気に入りのフレーバーを送信するサーバー・ラウンドトリップを追加することで、このサンプルを拡張します。最後に、このグラフをJavaFXで再実装し、それをJSFアプリケーションに埋込む方法を紹介します。

サンプルの実行

サンプル・アプリケーションのソース・コードは、こちらから入手できます。

このアプリケーションはFlex SDK 4.1、JSF実装のMojarra 2.0.2、およびJavaFX SDK 1.3.1を使用して開発しています。IDEにはNetBeans 6.9.1を使用しています。このIDEにはすでにJSFとJavaFXのライブラリがバンドルされています。付属する3つのプロジェクトは、SampleChartFlex、SampleChartFX、SampleWebです。

NetBeans内でこのWebアプリケーションを実行するには、これらのプロジェクトをNetBeansで開き、SampleWebプロジェクトを右クリックして実行します。

Flexプロジェクトを修正してコンパイルするには、Flex SDKをインストールし、そのインストール先を指すようにSampleChartFlex/build.xmlを編集する必要があります（**リスト1**）。

その後、NetBeansでBuildコマンドを実行してこれらのプロジェクトをビルドできます。SampleChartFlexプロジェクトとSampleChartFXプロジェクトのantビルド・ファイルは、パッケージ化実行後のswtファイルまたはjarファイルがビルド時にSampleWebプロジェクトに自動的にコピーされるようにカスタマイズされています。

アプリケーションの構築

まず、アイス・クリームのフレーバーの人気度を示すシンプルな円グラフ・アプリケーションをFlexで構築します（**図1**）。グラフのアイテムをクリックすると、選択したアイテムがメッセージ・ラベルに表示されます。

このFlexアプリケーションは円グラフとメッセージ・ラベルで構成されます。円グラフのデータは、getChartData()関数で設定されます（**リスト2**）。グラフのアイテムをクリックすると、onItemClickがそのイベントを処理してメッセージ・ラベルを更新します。ソース・ファイルは、mxmlcを使用してSampleChartFlex.swfにコンパイルされます。付属のSampleChartFlexサンプル・プロジェクトではプロジェクトのビルド時にmxmlcが呼び出されるように、antのbuild.xmlファイルがカスタマイズされています。

Flexアプリケーションの埋込み

Flashオブジェクトをこの JSF

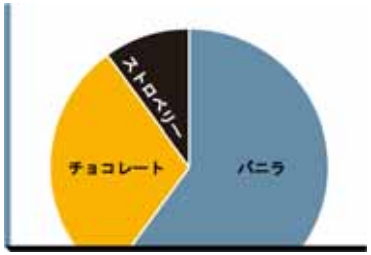


図1

Webアプリケーションに埋込むには、まずJSFアプリケーション・プロジェクトのSampleWebで、Webコンテンツの**resources/demochart**フォルダに**SampleChartFlex.swf**を追加します。さらに、埋込み部分をカプセル化するための複合コンポーネントを作成します。

複合コンポーネントは、JSF 2.0の新機能であり、カスタム・コンポーネントの開発が大幅に楽になります。エンコード、デコード、タグ・ライブラリ・ディスクリプタ（TLD）、レンダラなどについて難しく考える必要はもうありません。Facelet複合コンポーネント・ファイルを宣言して、そのファイルを使用するだけです。これは、Grailsで高く評価されているカスタム・タグのサポートに似ています。**リスト3**に、カスタム・コンポーネントの**demo:chart**について示します。

Adobe情報

LiveCycle Data Servicesは、クライアント側とサーバー側の両方にAdobeソリューション一式を導入できる場合は**特に魅力的**だが、BlazeDSおよびAMFのオープンソース化によって他のテクノロジーとの連携も可能になった。

オープンソースのSWFObjectを、Flashコンテンツを埋込むために使用します。swfobject.jsというJavaScriptファイルは、Flex 4 SDKインストールの**templates¥swfobject**フォルダにあります。このファイルをWebコンテンツの**resources¥demochart**フォルダにコピーします。

名前の衝突を防ぐために、この例では匿名関数の中でローカル変数を定義して、HTML要素のdivのIDに複合コンポーネントのクライアントIDをプリフィックスとして付加しています。

これでカスタム・コンポーネントの作成が完了し、他のJSFタグと同じように**demo:chart**タグを使用できるようになりました。Flexはこの実装で透過的に扱われます。**リスト4**に例を示します。

変数をFlexアプリケーションに渡す

たいていの場合、埋込みFlexアプリケーションは動的なデータを使用しています。flashVarsを活用すれば、変数をFlexアプリケーションに容易に渡すことができます。

ここでは、JSFマネージドBeanからアイス・クリームのフレーバーに関する調査結果を渡すことで、サンプルのグラフを拡張します（**リスト5**）。マネージドBeanの指定には、JSF 2.0アノテーションを使用します。

調査結果をマネージドBeanからFlexに渡すためには、まずJSF複合コンポーネントの**chart.xhtml**を修正して、調査結果を格納する**data**という名前の属性をinterfaceセクションに追加し、調査結果を**flashVars**としてFlexに渡すようにします（**リスト6**）。

利用側のJSFページで必要な処理は、アイス・クリーム・フレーバーの調査結果を**demo:chart**タグに渡すことだけです。次のJSFページのソース・コード（**index.xhtml**）を使用します。

```
<demo:chart
data="#{iceCreamSurvey.result}"/>
```

Flexアプリケーション側では、このパラメータを取得するように**getChartData**関数を修正する必要があります。**リスト7**のFlexソース・コードを使用します。

この例では、データ形式はシンプルです。そのため、単純に正規表現を使用して解析します。複雑なケースでは、JavaScript Object Notation（JSON）などのより体系的なエンコードを使用することを検討してください。

リスト1 リスト2 / リスト3 / リスト4 / リスト5 / リスト6

```
<!-- Change me to your Flex SDK installation location -->
<property name="FLEX_HOME" value="C:/Programs/Adobe/flex_sdk/4.1"/>
```

リスト全体をテキストで表示する

JSF Ajaxの利用

ここでは、より複雑なシナリオに進みます。FlexとJSFサーバー・セッションとの間のラウンドトリップ通信です。FlexとJSFを最高の状態で統合するためにJSF 2.0 Ajax機能を利用するという、斬新かつ実践的なアプローチに取り組みます。

Flexアプリケーションがサーバーと通信する方法はいくつかあります。LiveCycle Data Servicesは、Javaサーバーベース BlazeDSやAction Message Formatなどのさまざまなテクノロジーを包括したAdobeのデータ・ソリューションです。Flexには、HTTPやWebサービスなどの、サーバーと通信するための汎用データ・アクセス・コンポーネントも用意されています。さらに、FlexはJavaScriptとの統合の相性がよくブラウザ側での統合が可能です。この際、Ajaxアプリケーションに中継してサーバーと通信します。これらのアプローチではすべてJSFを利用できますが、それぞれにメリットとデメリットがあります。

JSF 2.0のリリース以降、Ajax APIが標準化されました。これにより、Flexアプリケーションと JSFの統合機能を活用できます。これは実質的にブラウザ側での統合です。ここでは、JSF Ajaxフレームワークを活用して、セッション・ステートおよびビュー・ステートの追跡を行います。JSF Ajax APIは2.0仕様の一部であることから、すべての実装でサポートされることが保証されます。サーバー側では、Flexクライアントはほぼ透過的に使用されます。そのため、このアプローチは既存のJSFアプリケーションでも簡単に利用できます。

そうはいっても、JSF Ajaxレイヤーを追加することでパフォーマンスのオーバーヘッドが生じるかもしれません。しかし、データ交換が小規模の場合は、Ajaxを使用するほとんどのケースで問題にならないでしょう。

ユーザーが円グラフのフレーバーをクリックすると選択されたアイテムが送信されるように、サンプルを修正します。JSFマネージドBeanでこの選択が処理されて、メッセージが返されます。すると、このメッセージがFlexアプリケーションに表示されます。Flexアプリケーション側では、ExternalInterfaceにより埋込みWebページ内部でJavaScript関数のdemo.ajax.submitが呼び出されるように、**onItemClick**関数を修正します。**demo.ajax.submit**関数については後ほど定義します。**リスト8**にFlexソース・コードを示します。

メッセージ・ラベルを更新するための**refresh**コールバック関数を追加します。この関数は、Flexアプリケーションの初期化時にExternalInterface.addCallbackによってJavaScriptに公開されます（**リスト9**）。

JSF複合コンポーネントについては、interfaceセクションにさらにresponse属性を追加します。この属性は、非同期送信に対するサーバー・レスポンスにマップされます。

JSF複合コンポーネントのソース・コード（**resources/demo/chart.xhtml**）は次のとおりです。

```
<cc:interface>
  <cc:attribute name="data" />
  <cc:attribute name="response" />
</cc:interface>
```

次に、implementationセクション内に、サーバーへの送信とサーバーからのレスポンス受信のためのhiddenフォームを定義します。

```
<h:form id="form"
 style="display:none">
 <h:outputText id="out"
 value="#{cc.attrs.response}"/>
</h:form>
```

リスト7 リスト8 リスト9 リスト10 リスト11

```
private function getChartData() : ArrayCollection {
    // Retrieve "data" from flashVars,
    // Formatted as Map.toString(), e.g.,
    // {Strawberry=10, Chocolate=30, Vanilla=60}
    var input : String = Application.application.parameters.data;
    var data : Array = input ? input.split(/\W+/) : [];
    var source = [];
    for (var index : int = 1; index < data.length - 1; index += 2) {
        source.push( {flavor: data[index], rank: parseInt(data[index+1])} );
    }
    return new ArrayCollection(source);
}
```

リスト全体をテキストで表示する

リスト10のJavaScriptを追加します。このJavaScriptは、非同期の送信と応答について処理します。

Flexの**onItemClick**関数では、リクエストをサーバーに送信するための**demo.ajax.submit**関数が呼び出されます。この関数では、JSF 2.0のJavaScript関数jsf.ajax.requestにより、次のようなオプションを持つhiddenフォームを使用して非同期リクエストが送信されます。

- ペイロードが、inputという名前のパススルー・リクエスト・パラメータとして送信されます。
- サーバーに対して、フォーム内で名前が付けられている**outputText**サブフォームをレンダリングするよう指定します。
- サーバーのレスポンスは**demo.ajax.onevent**イベント・ハンドラによって処理されます。

demo.ajax.oneventでは、Ajaxの送信イベントが処理されます。正常に処理されると、outputTextノードからレスポンスが取得され、Flashが公開するrefreshメソッドがコールされます。このメソッドでは、さまざまな方法でノード・テキストの取得を試行することで、ブラウザ間の違いによる問題が回避されます。

JSFサーバー側では、この送信に対応するJSFマネージドBeanのソース・コードを追加します（**リスト11**）。利用側のJSFページでは、まずページ・ヘッダに**jsf.js**を追加して、ページ内部で

JSF JavaScriptを有効にします。JSFページのソース・コード（index.xhtml）は次のとおりです。

```
<h:outputScript
  library="javax.faces"
  name="jsf.js" target="head"/>
```

さらに、カスタムのchartタグが公開するinputリクエスト・パラメータやresponse属性をマップする必要があります。これを行うための方法は複数あります。その1つは、ELアクション・バインディングで変数を扱うことができるJSF 2.0拡張機能を利用することです（**リスト12**）。

また、別の方法としてビュー・パラメータを活用できます。ビュー・パラメータによってリクエスト・パラメータをEL式にマップできます（**リスト13**）。

どちらの方法にも、それぞれ魅力的な点があります。JSF 2.0拡張機能を使用する場合は設定が少なくて済みます。一方、ビュー・パラメータは編集可能な値ホルダーであり、コンバータとバリデータを活用できます。複雑なエンコードが必要な場合は、ビュー・パラメータを使用する方法が適しています。

JavaFXとの統合

JavaFXでも同様の方法で、このグラフ・アプリケーションを実装できます（**図2**）。

リスト14にJavaFXソース・コードを示します（demo.piechart.Main.FX）。この円グラフのデータは、getChartData()関数によって設定されます（**リスト15**）。

このコードは意図的に先ほどのFlexアプリケーションに似せています。グラフのデータはdataという実行時引数によって設定されます。AppletStageExtensionを使用して、コンテナ・ページのJavaScript関数demo.ajax.submitを呼び出します。JavaFXの場合、refreshコールバック関数は簡単に公開できます。スクリプトレベルのパブリック関数はすべて、JavaFXのJavaScriptから見えるように可視性が自動的に確保されます。

JavaFXアプレットを埋込むには、SampleChartFX.jarとSampleChartFX_browser.jnlpを、Webコンテンツのresources/demochartフォルダにコピーします。NetBeansによって生成されたjnlpファイルは、デフォルトでローカル・コードベースを指していることに注意してください。いずれにせよWebページのjarファイルの場所を指定するので、jnlpファイルからcodebase属性を削除します。

その後に必要な作業は、JSF複合コンポーネントにJavaFXアプレットを埋込むための微修正だけです（**リスト16**）。

このJavaScriptコードのほとんどは、JavaFXアプレットに対しても動作します。変更されたのは、JavaScriptがJavaFXにコールバックする方法のみです。demo.ajax.onevent関数の内部では、chart.refresh(response)の代わりにchart.script.refresh(response)を使用する必要があります。両方の状況で動作させるには、次のコードを使用します。

```
chart.refresh?
   chart.refresh(response) :
   chart.script.refresh(response)
```

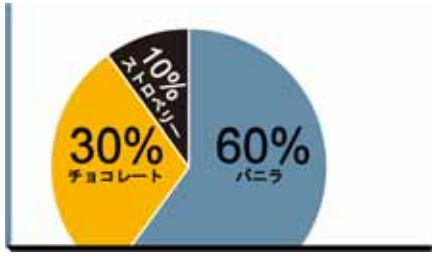


図2

リスト12 リスト13 リスト14 リスト15 リスト16

```
<demo:chart data="#{iceCreamSurvey.result}"
  response="#{iceCreamSurvey.reply(param.input)}" />
```

リスト全体をテキストで表示する

これで完成です。利用側のJSFページを変更する必要はありません。グラフの設定にJSFとJavaFXのどちらを使用するかは実装上の事項であり、利用側のページにとっては完全に透過的です。

まとめ

この記事では、JSF 2.0の新機能を活用して、Adobe FlexとJavaFXをJSFアプリケーションに統合しました。これらの新機能を利用すれば、エンコード、デコード、およびビュー・ステートの追跡について慎重に組み立てる必要がなくなります。ここでは特に、FlexおよびJavaFXの埋込み部分をカプセル化するためのカスタム・コンポーネントを開発しました。●

参考情報

- JavaFX
- Adobe Flex
- JavaServer Faces 2.0のダウンロード

なぜWebアプリケーションの自動テストなのか

Java ChampionのKevin Nilson氏が、言語にとらわれないテスト・ツールセットについて語る

Kevin Nilson

写真：
Margot Hartford

Kevin Nilson氏は、その経歴の大部分を、複雑なWebアプリケーションの構築に費やしてきました。そして今、高品質なWebアプリケーションを構築するには自動開発者テストが不可欠であると実感しています。Nilson氏は長い時間をかけて、優れたオープンソース・ツールをまとめ、カスタム・ツールを作成しました。この結果、テストがコーディング・プロセスに含まれることで、高品質アプリケーションの開発に必要となる時間が短縮されました。Nilson氏いわく、「アプリケーションの構築時にコードを修正する方が簡単です。」ここでは、Nilson氏自身が作成したツールセットについて話していただきました。このツールセットは言語にとらわれることなく、TestSwarm、QUnit、jQuery、Hudson、GlassFish Server Open Source Edition、Sun SPOT Java Development Kitなどのさまざまなツールと統合できます。

一般的なテストや自動テストを、特に開発者が実行することが重要な理由

開発環境でテストの優先順位を付けることは、高品質なソフトウェアの構築と維持に不可欠です。バグを修正するコストは、バグを早く発見できれば比例して安くすみます。ある行をコーディングした5分後にバグが見つかれば、そのバグはすぐに修正できます。一方、2週間後にバグが見つかれば、そのバグの修正ははるかに難しくなります。2週間経てば、最初にそのコードを追加した理由さえも忘れてしまうかもしれません。

私はいつも、すべての新機能について100%のテスト・カバレッジを設定することをお勧めしています。自動リグレッション・テストを行わなければ、バグはいつまでも付きまといます。修正されたバグが後で再発するということはよくあります。バグが見つかるたびに、まずはそのバグが原因で失敗するテストを記述すべきです。失敗するテストが完成したら、バグの修正に移ってください。バグを修正すれば、このテストは成功するはずです。この時点で、バグが再発した場合に失敗するテストが手元にあることになります。時間の経過とともに、特に労力を意識することもなく、大規模なリグレッション・テストを作成できるのです。実際、長期的には、テストを記述するために費やした労力よりも多くの効果がテストで得られます。

プロジェクトが複雑になればなるほど、成果物を手動で検証するために多くの時間がかかります。また、複雑なシステムでは、プロジェクト中に要件が大幅に変更されることも珍しくありません。大規模なシステムの変更や機能拡張を行うと、多くの場合、不用意にバグを埋込むことになります。

一方、開発者が開発プロセスの間に自動テストを実行すれば、開発の速度を大幅に上げることが可能です。開発者は自動テストをコーディング中に実行して、正しく動作していない機能やバグを発見できます。さらに、同じテストを後で、リリース前の検証として実行できます。

私はほとんどの時間を、JavaおよびJavaScriptでのフレームワークの構築に費やしてきました。これは複数のプロジェクトで、多数の開発チームが使用するようなフレームワークです。フレームワークの構築では、フレームワークが今後どのように使用されるのかを正確に把握していないこともよくあります。しっかりした自動テストがあれば、現在の成果物には含まれていないが今後は含まれる可能性のあるシナリオをフレームワークでテストできます。

Webアプリケーションのテストで直面する特有の問題

ここ数年でHTML5とWeb 2.0がWebアプリケーションの主流となり、ロジックがサーバーからブラウザに移行されました。Webアプリケーション・コーディングでもっとも大きな問題となるのは、アプリケーションを数多くのプラットフォーム、数多くのブラウザで実行しなければならないことです。すべてのブラウザはほぼ同じように動作しますが、多くのブラウザにバグがあり、それが非互換性の原因となっています。

Webアプリケーションの構築時には、ターゲットとするブラウザを決めることが非常に重要です。その上で、それらのブラウザ上でアプリケーションを徹底的にテストする必要があります。Webアプリケーションの最大のメリットは、1度アプリケー

ションを構築すると、インストールや設定なしで幅広いユーザーに利用してもらえるという点にあります。ユーザーが使用する可能性があるシステムには、新旧のMicrosoft Windows、Mac、Linux、Oracle Solarisの各システムのほか、iPhone、Android、BlackBerryといったモバイル・デバイスも含まれます。

多くのJava開発者にとってのもう1つの課題は、JavaScriptなどのブラウザで動作する動的言語を使用することです。動的言語は柔軟性と効率に優れているものの、実行時にさまざまな種類のバグが発生する可能性があります。Java開発者は一般的に、Java、C、C++などの静的言語には精通しています。JavaScriptでは、Javaであればコンパイル時に検出されるようなバグの多くが、実行時に検出されます。この種のバグの簡単な例として、Boolean値と整数の加算や、変数のスペルミスなどが挙げられます。これらのバグはJavaであればコンパイラによって検出されてアプリケーションのビルドが失敗しますが、JavaScriptでは検出されません。

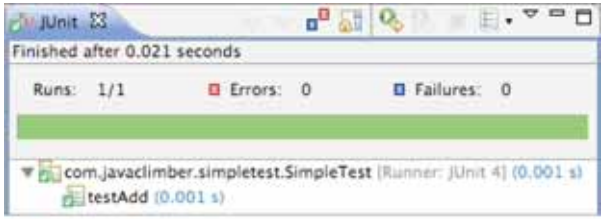


図1

図2

Webアプリケーションのテストのコーディングに適したツール

静的コード解析を実行してから、単体テストとインテグレーション・テストを記述することをお勧めします。JSLintは長年親しまれている静的コード解析ツールであり、JavaScriptコードの問題の検出に役立ちます。

Closure Compilerは比較的新しいツールであり、作成したJavaScriptを、より高速でダウンロード/実行が可能なJavaScriptにコンパイルできます。Closure CompilerではJavaScriptが解析され、デッド・コード（到達されないコード）の削除、コードの修正、構文チェック、変数参照チェック、型チェックが実行されて、JavaScriptのよくある問題についての警告が表示されます。

Webアプリケーションの単体テストと機能テストには、QUnitおよびTestSwarmを使用することをお勧めします。QUnitは、jQueryプロジェクトでのコードとプラグインのテストに使用するテスト・スイートです。TestSwarmでは、テストするコードに接続可能なすべてのブラウザでQUnitテストを実行できます。

QUnitとJUnitの類似点

QUnitとJUnitは、コードのテストのためのよく似たツールです。JUnitは非常によく利用されているJavaコード・テスト・フレームワークです。**リスト1**は、Javaの**add**メソッドをテストする基本的なJUnitテストです。

リスト1 **リスト2**

```
@Test
public void testAdd() {
  int a=1;
  int b=1;

  assertEquals("one plus one is two", 2, add(a,b));
}
```

リストをテキストで表示する

結果は**図1**のようになります。

JUnitでのテストは、**@Test**アノテーションでマークします。**org.junit.Assert**クラスには、**assertTrue**、**assertFalse**、**assertEquals**、**assertNull**、**assertNotNull**など、テストに役立つさまざまな静的アサーション・メソッドが用意されています。これらのアサーション・メソッドを使用して、期待される結果と実際の結果を比較します。JUnitテストはIDEで開発者が実行するほか、Maven、Gradle、Antなどのビルド・ツールで実行することもできます。

リスト2は、JavaScriptの**add**関数をテストするために、この同じテストをQUnitで記述した例です。

結果は**図2**のようになります。

テストは、論理的に区別するために、モジュールにグループ化できます。それぞれのテストには複数のアサーションを含めることができます。QUnitには、**ok**、**equals**、**notEqual**などのアサーション関数が用意されています。QUnitテストはブラウザで実行し、その結果はWebページに表示されます。このため、テストのURLをブラウザに入力するだけで、どのブラウザでもQUnitテストを実行できます。

JavaScriptにはスレッドがないため、QUnitテストのコーディングは難しい場合があります。たとえば、ユーザーがボタンのクリックなどの操作を実行した後の動作をテストしたいということはよくあります。ユーザーがボタンをクリックすると、長時間実行されるタスクや、Ajaxなどの非同期イベントが発生する場合もあるでしょう。問題となるのはこのようなケースです。ボタンに関連付けられている操作のコードが実行される前に、テスト・コードが実行されるためです。そのようなテスト・コードではスレッドを解放し、後で再実行する必要があります。

この問題の回避策の1つは、**setTimeout**を使用し、特定の時間が経過した後にアサーションを行うようスケジュールすることです。しかし、これでも、**setTimeout**の実行時にコードの実行が完了していない可能性があるため、うまく動作しません。そのため、ポーリングを使用して、テスト対象のコードの完了を確認できます。このポーリングにも、一定時間後にタイムアウトするように設定する必要があります。

appendToという企業がオープンソースのjQueryプラグインであるwhenAvailableを開発しました。これは、jQueryの利用者が使用できます。

whenAvailableプラグインは、処理の前にDocument Object Model（DOM）要素のポーリングを継続的に実行します。また、機能テストの問題点への対処には、FuncUnitを使用できます。FuncUnitは機能テストを行うためのQUnitアドオンです。FuncUnitにより、テストを続行するには要素が出現するのを待つ必要があるという問題が解決します。

厳しい見解

自動リグレッション・テストを行わなければ、**バグがいつまでも付きまとう**。修正されたバグが後で再発するということは非常によくある。

テスト実行をすべてのブラウザで自動化する方法

TestSwarmは、JavaScriptの分散型の継続的なインテグレーション・テストを行うためのMozilla Labsプロジェクトです。ブラウザはTestSwarmサーバーに接続することで、テスト実行対象のブラウザ群（Swarmとは「群れ」の意味）に参加できます。

テストを実行するには、シンプルなフォームを送信して、実行するテストとテスト対象ブラウザについてTestSwarmに通知します。テストは、テスト対象の各ブラウザのiFrame内で実行されます。それぞれのテストが完了すると、テストの結果がTestSwarmサーバーに送信されます。TestSwarmは、テスト対象となる接続されたブラウザにテストを実行するよう通知するプロセスを管理するPHPアプリケーションです。TestSwarmサーバーは、実行されたテストの結果を記録します。

ほとんどのJavaScript向け継続的インテグレーション・ツールではブラウザを起動しようとしますが、TestSwarmは、ネットワーク上のすべてのブラウザをサーバーに接続させるという点で異なります。これは、複数のブラウザとプラットフォームでテストする際には大きなメリットとなります。iPhoneやAndroidでテストに参加する場合でも、必要な作業はそのブラウザからTestSwarmサーバーのWebページにアクセスすることだけです。TestSwarmでは、QUnit（jQuery）、UnitTestJS（Prototype）、JSSpec（MooTools）、JSUnit、Selenium、Dojo Objective Harnessで記述されたテストを実行できます。

継続的インテグレーションに使用するツール

私は、TestSwarmでのテスト実行をトリガーするために、GlassFish Server Open Source EditionでHudson/Jenkinsを使用しています。チームのメンバーと協力して、HudsonをTestSwarmに統合するためのHudsonプラグインを作成しました。Hudsonはコード・チェックイン用のリポジトリを監視します。コードがチェックインされると、Hudsonは私が開発したプラグインを使用して新しいジョブをTestSwarmに送信します。このプラグインは次に、結果を取得するためにTestSwarmをポーリングします。テストでエラーが発生した場合は、Hudsonから電子メール通知が送信されます。

TestSwarmに接続するブラウザの管理には、Oracle VM VirtualBoxを使用しています。4つのオペレーティング・システムをOracle VM VirtualBoxで稼働させ、そこで10種類のブラウザを実行しています。Oracle VM VirtualBoxには、仮想環境の起動と停止に使用できるWebサービスAPIがあります。当チームではさらに、1日に1回仮想環境を再起動するHudsonプラグインも作成しました。

Hudsonにはビルドの状態を表示するXML APIを含む、RESTスタイルのAPIが用意されています。私はSun SPOT Java Development Kitsを使用して、HudsonのXML APIをポーリングしています。各Sun SPOT Java Development Kitには8個のLEDライトが備わっているため、同時に8つのビルドを監視できるのです。

これまでに構築した環境に似たテスト環境のセットアップの難易度

これに類似した環境をセットアップするのはかなり簡単です。当チームでは、2～3日間で基本的なセットアップを完了できました。その後時間をかけて、「なくてもいいがあれば便利な」機能を少しずつ追加しました。今使用しているツールはすべて、無料でオープンソースのものです。

Webアプリケーションのテストに使用できるその他のツール

QUnitとTestSwarmを使い始める前は、ほとんどのケースでCanoo WebTestを使用していました。Canoo WebTestでは、XMLまたはGroovyでテストを記述できます。また、Rhinoを使用してコマンドラインから実行します。

Seleniumも優れたツールです。これは、Webアプリケーションのテストで非常に人気があります。Seleniumにはクリック操作を記録する機能があるため、シンプルなテストを簡単に記述できます。私はシンプルかつ強力なツールであるという点で、QUnitを好んで利用しています。QUnitはJavaScript開発者にとってすばらしいツールです。しかしQA開発者にとっては、おそらくSeleniumの方が便利ではないでしょうか。

Javaおよびオラクルのテスト用テクノロジーの利用経験

現在利用しているのは、Hudson、GlassFish Server Open Source Edition、Oracle VM VirtualBox、Sun SPOT Java Development Kitsです。これらのツールはそれぞれオープンソースであり、非常に柔軟性に優れています。これらのツールを他のオープンソース・テスト・ツールと統合して、Webアプリケーション用のエンド・ツー・エンドの自動テスト環境を構築できています。●

参考情報

- TestSwarm
- QUnit
- jQuery
- GlassFish Server Open Source Edition
- Sun SPOT Java Development Kit
- JSLint
- Closure Compiler
- JUnit
- Maven
- Gradle
- Ant
- appendTo
- FuncUnit
- Hudson
- Jenkins
- Oracle VM VirtualBox
- Canoo WebTest
- Rhino
- Selenium

Adam Bien

Java EE 6におけるリソース・インジェクション

Java EE 6でリソース・インジェクションに利用できるさまざまなアノテーションや、これらが必要となる理由および利用可能な状況について学習する

@Resource、@Inject、@PersistenceContextのどれを使用すべきでしょうか。または、従来のJava Naming and Directory Interface（JNDI）ルックアップをそのまま使用すべきでしょうか。Java Platform, Enterprise Edition 6（Java EE 6）では、構成済みリソースをインジェクション（注入）するための複数の方法が用意されています。インジェクションできるリソースは、データソース、宛先、Java Persistence API（JPA）のEntityManager、Java Transaction API（JTA）のUserTransaction、URL、J2EE Connector Architecture（JCA）のコネクタ、メール・セッション、LDAPコネクション、さらにはJNDIに登録されたカスタム・リソースとさまざまです。では、リソース・インジェクションに複数のアノテーションが必要となる理由は何でしょうか。

この記事では、Java EE 6でリソース・インジェクションに利用できる各種アノテーションや、これらが必要な理由と利用できる状況について説明します。

写真：
Thomas Einberger / Getty Images

@Resource-汎用JNDIリソース・インジェクタ

@Resourceアノテーションは、JSR-250「Common Annotations for the Java Platform（Javaプラットフォーム向け共通アノテーション）」で定義されています。この仕様には、@PreDestroy、@PostConstruct、@RolesAllowedなどの、他のよく知られているアノテーションも含まれています。JSR-250はJava EE 5とJava EE 6で必須ですが、独立したJava EE仕様として定義されているため、Java EE以外のフレームワークやライブラリでも使用できます。JSR-250アノテーションの一部はJava Platform, Standard Edition（Java SE）でもパッケージ化されています。これには、@Resourceも含まれます。

@Resourceアノテーションは、JNDIネームスペースに登録されているすべてのリソースのインジェクションを目的としています。JNDI名は、構成済みリソースのエイリアスとして使用されます。通常、この特定のAPIはインタフェースとしてインジェクションされます。JNDIによって、リソースを使用する側は、その実際の実装や構成から切り離されます。

```
public class
ControlWithDataSourceDI {
 @Resource(name="jdbc/sample
 DataSource ds;
}
```

name要素には実際のJNDI名を指定します。上記のデータソースは、JNDI名jdbc/sampleで構成され、この名前を使用してインジェクションされています。Java EE 6には「Convention over Configuration（設定より規約）」の原則が全面的に採用されているため、name要素の指定は必須ではありません。指定しない場合は、JNDI名はフィールド名から直接導き出されます。しかしこの方法は、ここで扱う例には適しません。コードが@Resource DataSource jdbc/sampleのようになり、コンパイルに失敗していまいます。また、アプリケーション・サーバー構成を変更するとこのコードが無効になり、フィールドの名前を変更しなければならなくなります。この特定のケースでは、JNDI名を明確に指定する方が適しています。JNDIリソースの名前を変更した場合、影響を受けるのはname要素だけです。フィールド名は影響を受けません。

mappedNameの利用は避けてください。この属性はベンダーによってはサポートされておらず、アプリケーション・サーバー実装に依存するためです。インジェクションされたリソースは共有でき、shareable要素（デフォルトではtrueに設定）を使用して構成できます。ほとんどのリソースは不変なリソース（インジェクションされたプリミティブ型、インジェクションされたURLなど）とリソース・ファクトリ（DataSourceなど）のいずれかであるため、共有が可能です。

@Resourceアノテーションは、フィールド・インジェクションとsetterインジェクションをサポートしています。フィールド・インジェクションの方が余分なsetterを実装する必要がないため無駄がありません。インジェクションされたクラスのモックを単体テストで使用するためには、そのフィールドの可視性をpackageに弱める必要があることに注意してください。

@DataSourceDefinition-ちょっとしたDevOps

ほとんどのリソースは、管理コ

ンソール、Java Management Extensions（JMX）、コマンドライン・インタフェース、さらにはRepresentational State Transfer（REST）などを使用してアプリケーションにインストールされ、インストール方法は特定されません。しかし、データソースは標準的な方法で構成できます。JSR-250仕様のバージョン1.1で導入された**@DataSourceDefinition**アノテーションでは、データソースの構成、インストール、およびJNDI公開を、移植性のある方法で行うことができます（**リスト1**）。

1つ以上の**@DataSourceDefinition**アノテーションをクラスで宣言し（このクラスは**@DataSourceDefinition**アノテーションに含まれることになります）、アプリケーションとともにデプロイして、自動インストールに必要な情報を指定できます。データソースは**@Resource**アノテーションのname要素に指定されたJNDI名を使用して直接インジェクションできます。また、このデータソースは手動ルックアップでもアクセスできます。一般的なアプリケーションでデータソースへの直接アクセスが必要になるのは、ストアド・プロシージャを呼び出すときや特定の最適化機能を利用するときだけあり、このようなケースはまれです。永続化して使用するケースのほとんどはJPAで対処できます。ただし、JPA **EntityManager**では、persistence.xml構成ファイルで構成された既知のJNDI名でデータソースを登録しておく必要があります。

大多数のJava EEアプリケーションは、**@DataSourceDefinition**を使用して管理タスクなしでインストールできます。インストール成功のための前提となるのは、サーバーにJDBCドライバを適切にインストールしておくことだけです。ここでの明らかな欠点は柔軟性に欠けることです。構成の変更のために、アプリケーションを再デプロイする必要があります。

一見すると、再デプロイが必要な点はデメリットのように感じられますが、自己完結型アプリケーションのデプロイはDevOpsの中心的な考え方です。この考えでは、アプリケーションの運用（operation）と開発（development）が1つの一貫した単位として考えられています。アプリケーションは、手動で実行しなくても、ビルド、構成、インストールが続けて行われます。このシナリオでは、アプリケーション・サーバー、オペレーティング・システム構成、アプリケーション・コードの間に違いは生じません。アプリケーションのインストールまたは実行に必要なすべての情報が同じように扱われます。

Javaデータ
永続化ユースケースのほとんどはJPAで対処できる。

@PersistenceContext-特殊なケース

@PersistenceContextは、Java EE 5のJPA仕様で導入されたものであり、JSR-250には含まれていません。**EntityManager**は共有不可能なリソースと（さらに、**EntityManagerFactory**は共有可能リソースと）見なすことができますが、これを**@Resource**アノテーションでインジェクションするには、面倒な回避策が必要になります。**@Resource**アノテーションで**EntityManager**をインジェクションするためには、まず**EntityManager**をJNDIネームスペースに登録する必要があります。JNDIネームスペースへの**EntityManager**の登録は、クラス・レベルで**@PersistenceContext**アノテーションを適用することで行うことができます。**EntityManager**は要素名によってJNDI名にバインドされます。JNDIスコープは、事前定義済みの命名規則に従うことで割り当てられます。たとえば、**java:comp/env**の場合、EntityManagerはコンポーネント・ネームスペースに公開されます。ローカル・コンポーネント・ネームスペースからリソースを取得するには、**@Resource**アノテーションのlookup要素を使用する必要があります（**リスト2**）。また、JDNIグローバル・ネームスペース（**java:global**）に公開することも可能です。このためには、JNDI名に**java:global**というプリフィックスを付ける必要があります。インジェクションには、**@Resource**アノテーションの**name**要素を使用できます（**リスト3**）。

JNDIネームスペースにEntityManagerを登録した後は、**SessionContext**で直接ルックアップすると期待どおりに動作します（**リスト4**）。

手動ルックアップの場合は、まず**@Resource**を使用して**SessionContext**をインジェクションする必要があります。また、デフォルト・コンストラクタで**InitialContext**を作成することも可能です。これにより、登録したJNDI名を使用して**EntityManager**を取得できるようになります。手動ルックアップは、**@PostConstruct**アノテーションを付加したメソッドで初期化時に実行すべきです。EntityManagerのJNDIルックアップがJava EE 6アプリケーションで必要となることはほとんどありません。EntityManagerはEnterprise JavaBeans（EJB）3.1のBeanや、Contexts and Dependency Injection（CDI）のマネージドBeanに直接インジェクションできます。手動ルックアップは、マネージドではないコンポーネントで有用となる場合があります。

EntityManagerのインジェクションはCDIマネージドBeanでも動作しますが、このBeanをUIレイヤーに対して直接公開することはできません。ステートレス環境で**EntityManager**を構成できるのは、**@PersistenceContext(type=PersistenceContextType.TRANSACTION)**アノテーション（デフォルト値）を使用する場合だけです。そのため、**EntityManager**とのやり取りごとにア

リスト1 リスト2 リスト3 リスト4

```
@DataSourceDefinition(
  className="org.apache.derby.jdbc.ClientDataSource",
  serverName="localhost",
  name="java:global/jdbc/InjectionSample",
  databaseName="InjectionSample;create=true",
  portNumber=1527,
  user="sample",
  password="sample"
)
@Stateless
public class JDBCDataSourceConfiguration {
  @Resource(lookup="java:global/jdbc/InjectionSample")
  private DataSource dataSource;
}
```

リスト全体をテキストで表示する

クティブなトランザクションが必要となります。アクティブなトランザクションがない場合は、**javax.persistence.TransactionRequiredException** がスローされます。CDIマネージドBeanでは、そのままの状態ではトランザクションを開始できません。EJB 3.1ステートレス・セッションBeanではこの問題をもっともシンプルな方法で解決しています。手動によるトランザクション管理もCDI拡張の利用もEJBステートレス・セッションBeanでは不要です。インタフェースなしの単一のビューBean（ファサードなど）によってトランザクションが管理され、それ以外の構成、フレームワーク、手作業によるコーディングは必要ないのです。

@Injectが不十分な理由

@InjectアノテーションはJSR-330「Dependency Injection for Java（Javaの依存性注入）」で導入されたもので、Java EE 6には不可欠な機能です（そのため、Web Profileにも含まれています）。JSR-299「Contexts and Dependency Injection（CDI：コンテキストと依存性の注入）」は、Java EE 6プラットフォームの依存性注入（DI）機能を大幅に拡張するもので、最小限の仕様であるJSR-330に基づいています。残念ながら、**@Inject**アノテーションそのものは、どのような直接リソース・インジェクションにも適していません。それは、要素が不足しているからです。フィールド名はJNDI名として活用できるものの、Java言語の命名上の制約があるので、**jdbc/sample**、**queue/Orders**などの特定のパターンをフィールド名で表現できません。さらに、**EntityManager**のインジェクションでは、タイプ（TRANSACTIONまたはEXTENDED）や永続性ユニット名など、その他のパラメータも必要になります。

それでも、**@Inject**はプロデューサや限定子と合わせて使用すれば、クリーンで柔軟性に優れたリソース処理を行う1つの方法として検討できます。リソースの登録を単純なクラスに一元化して、これらのリソースのインジェクションを要求に応じて明確かつそれぞれ分離させて行うことができるのです。リソースのタイプが1つであれば、限定子を設定しなくてもインジェクションを行うことが可能です。より洗練されたプロジェクトでは、**DataSource**インスタンスが複数あったり、さまざまな永続性ユニットに対応する**EntityManager**があったりします。この場合、カスタムの**@Qualifier**を使用することで、プロデューサとインジェクション・ポイント（インジェクションするポイント）との適切な組み合わせを判断できます。**@Qualifier**アノテーションにわかりやすい名前を指定すれば、可読性と分離性が高まります。次のコンシューマはJNDI名にもリソースが実際に取得された方法にも依存していません。

Java 7や1日のスケジュールなどについて*Java Magazine*のJustin Kestelynに語るAdam Bien氏

リスト5 **リスト6**

```
public class LegacyDataSourceProducer {
  @Produces @Legacy @Resource(name="jdbc/sample")
  private DataSource ds;
}
```

リスト全体をテキストで表示する

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, TYPE})
public @interface Legacy {}
```

EJB BeanまたはマネージドBeanのフィールドやリソースは、**@Produces**を指定することで、リソースとその「生成物」を同時に取得するために使用できます。**@Qualifier**は、プロデューサとインジェクション・ポイントを指定するものです。この両方が一致すると、リソースがインジェクションされます。また、インジェクションには、組込みの**@Named**限定子を使用することもできます。**@Named**アノテーションは、単純な**String**を使用して型保証（type-safe）されていないオブジェクトをマッチングするので、厄介なエラーの発生源となることがあります。大規模なコード・ベースでは**String**のスペルミスの検出は難しいからです。

生成されたリソースは、**@Inject**アノテーションとカスタムの**@Qualifier**アノテーションを使用して、簡単にコンシューマにインジェクションできます。リソースは直接、フィールドやメソッドで公開できます。メソッドの方がより優れた柔軟性を確保でき、たとえば、返されたリソースのロギング、デコレート、さらには再構成までもが可能です。メソッドをプロデューサとして使用する場合は、**@Produces**アノテーションと**@Legacy**アノテーション（**リスト5**）を、フィールドではなく任意のメソッドに付ける必要があります。

リソースのインジェクションに必須となるのは、**@Inject**アノテーションと**@Legacy**アノテーションのみです（**リスト6**）。それ以上の構成は必要ありません。また、アノテーションにより、「レガシーなデータソースをインジェクションする」ということが明示されるため、コードの可読性が非常に高まります。これは2つのアノテーションをフィールドから「getter」に移動するという問題のため、まずはフィールドを使用して、不要なコードの膨張を避けることもできます。メソッドのプロデューサは、コンシューマやインジェクション・ポイントに影響を与えずに、必要に応じて付加できます。

まとめ

JNDIに登録されている事前構成済みのリソースは、通常**@Resource**アノテーションでインジェクションします。**@Resource**でインジェクションできるリソースのタイプは**String**からCORBAサービスまで多岐にわたり、たとえばプリミティブ型（**String**、longなど）、**javax.xml.rpc.Service**、**javax.xml.ws.Service**、**javax.jws.WebService**、**javax.sql.DataSource**、**javax.jms.ConnectionFactory**、**javax.jms.QueueConnectionFactory**、**javax.jms.TopicConnectionFactory**、

javax.mail.Session、java.net.URL、javax.resource.cci.ConnectionFactory、org.omg.CORBA_2_3.ORB、javax.jms.Queue、javax.jms.Topic、javax.resource.cci.InteractionSpec、javax.transaction.UserTransactionなどがあります。これらのリソースは通常、特定のアプリケーションに限定されることなく、アプリケーション・サーバー上で管理されています。つまり、複数のアプリケーションで同じリソースが共有されることもあります。

EntityManagerはJNDIネームスペースに登録されないため、標準的なケースでは@Resourceインジェクションで利用できません。さらに、EntityManagerはアプリケーション内で構成され、他のアプリケーションとは共有されません。EntityManagerをインジェクションするには、タイプやunitNameなどの追加情報を使用します。@Resourceまたは@Injectで間接的にインジェクションすることもできますが、この目的ではほとんどのケースで、プレーンな@PersistenceContextアノテーションが使用されます。Java EEでは原則「Convention over Configuration」の全面的な適用により、適切なデフォルト値が設定されます。そのため、永続性ユニットが1つだけの場合は、それを明示する必要はありません。EntityManagerはトランザクション構成とともにインジェクションされ、それ以上のイベントは発生しません。

@Injectアノテーションは拡張や回避策を使用しない限り、JNDIネームスペースからリソースを直接インジェクションするのには適していません。@Injectをカスタムの@Qualifierとともに使用すれば、@Producesによって公開されるリソースのインジェクションを行う方法として検討できます。クライアント（インジェクション・ポイント）はJNDIから完全に分離されます。リソース登録ロジックやルックアップ・ロジックの場合にもこれは当てはまります。この追加のレイヤーが通常のプロジェクトで必要となることはまれですが、プラットフォーム開発、製品開発、API開発では非常に使い道があります。この場合、あるサービスを使用する側では、カスタムの限定子と@Injectアノテーションだけを使用して、必要なリソースをインジェクションできます。

@EJBアノテーションはリソースのインジェクションには使用できず、依存するEJB Beanのインジェクションのみに適しています。古いスタイルのInitialContext#lookupは、JNDI名がコンパイル時に不明であり実行時に指定する必要があるという例外的なケースでのみ必要となります。●

参考情報

- JSR -317：「Java Persistence 2.0」
- 「Contexts and Dependency Injection in Java EE 6」
- 「Enterprise JavaBeans 3.1 with Contexts and Dependency Injection:The Perfect Synergy」
- 「Simplicity by Design」

JSR-211：Content Handler APIの利用

非常に特化した問題をクリーンかつ明快に解決するCHAPIの使用方法について学ぶ

Vikram Goyal

写真：
Jonathan Wood /
Getty Images

「Content Handler API（CHAPI）」はJSR-211としても知られ、それほど重要な処理をしていないように見えるものの、知れば知るほど便利だとわかる小粋なAPIです。非常に特化した問題を、クリーンかつ明快に解決します。

この記事では、このAPIが対応する問題領域と、その解決のためのAPIの動作について理解を深めることができます。また、APIの構造の概要や、それぞれのMIDletでの最適な使用方法について説明します。最後に、このAPIの利用例を示すシンプルなユースケースを紹介します。

注：この記事で紹介するソース・コードはこちらからダウンロードできます。

問題領域

駆け出しの写真家が、カメラ付きのデバイスで撮影したイメージを、このデバイス自体に保存しているとします。このデバイス上でイメージ・ライブラリを閲覧するためのアプリケーションを作成するというケースについて、考えてみましょう。デバイスに搭載されたイメージ・ブラウザの代わりに、この小粋なアプリケーションを使用できるようにします。このアプリケーションには追加の機能（たとえば、オンラインでのイメージ共有ツール）を組み込み、ユーザーがイメージを閲覧する際には著作権情報を表示させたいと考えています。デバイスやJava仮想マシン（JVM）の標準のイメージ閲覧機能に頼らずに、イメージを閲覧する際には必ずこの著作権情報を表示する特別なイメージ・ビューアが開くようにします。

一言で言えば、イメージに常に著作権情報を表示できる新しいイメージ・コンテンツ・ハンドラを作成します。手の込んだコーディングを行ってもよいのですが、Content Handler APIを使用すると、この特別なイメージ・コンテンツ・ハンドラをデバイスのアプリケーション管理ソフトウェア（AMS）に登録することが可能です。

Content Handler API

CHAPIには、アプリケーションからJava Platform, Micro Edition（Java ME）アプリケーションおよびJava以外のアプリケーションを起動するための実行モデルが用意されています。これはつまり、このAPIを使用して既存のコンテンツ・ハンドラを登録すると、そのコンテンツ・ハンドラをAPIで起動できるということです。もちろん、コンテンツ・ハンドラを起動するにはある種の識別を行う必要があります。

たとえば、前述の「問題領域」のとおり、すべてのJPGイメージを特化したイメージ・コンテンツ・ハンドラで処理することを考えているとしましょう。この場合、このコンテンツ・ハンドラ・クラスを登録するには、JPG MIMEタイプを今後この特別なコンテンツ・ハンドラで処理するということをマニフェスト・ファイルに記述するか、またはプログラムによってAMSに通知します。（複数のハンドラが登録されている場合、CHAPIはランダムに1つを選択します。）MIMEタイプ（コンテンツ・タイプ）だけではなく、URLやコンテンツ・ハンドラID要素でも識別できます。同じタイプに対して複数のコンテンツ・ハンドラを登録して、アクションごとにどのハンドラを起動するのかを指定することもできます。

深い視点

JSR-211は、一見たいした処理を行わないように思える**小粋なAPI**である。

RegistryとInvocation

Registry（レジストリ）クラスはおわかりのとおり、Java ME環境内にある既知のすべてのコンテンツ・ハンドラを管理するセントラル・リポジトリです。このクラスにはすべてのコンテンツ・ハンドラのライフサイクル関連メソッド（つまり、登録と登録解除）があり、メタ情報の取得や、言うまでもなくコンテンツ・ハンドラの実際の起動を行うことができます。それぞれのコ

ンテンツ・ハンドラは0個以上のコンテンツ・タイプ（image/jpegなど）、拡張子（.jpgなど）、およびアクション（「開く」など）によってマークされます。もちろん、それぞれのハンドラには一意のIDが割り当てられます。このIDはアクセス制御に使用されます。

複数のコンテンツ・ハンドラをレジストリ全体でつなぐこと（「ハンドラ・チェーン」）が可能です。これにより、1つのコンテンツ・ハンドラでリクエストを完了できない場合に、コール元の（ハンドラを起動した側の）クラスに制御を戻す前に、他のハンドラを起動できます。このチェーン動作はレジストリによってシームレスに処理されます。

まず知っておくべき情報

CHAPIとその実行モデルでは、URL、コンテンツ・タイプ、またはコンテンツ・ハンドラIDによって、アプリケーションから登録済みのJava MEアプリケーションおよびJava以外のアプリケーションを起動できる。

セントラル・レジストリへのアクセスには、Registryクラスの**getRegistry(String classname)**静的メソッドを使用します。引数のclassnameは、レジストリのメソッドにアクセスするコンテンツ・ハンドラまたはアプリケーションの名前です。この名前は、Java Decompiler（JAD）ファイルに記述しておく必要があります。または、プログラムで**getRegistry**メソッドをコールする前に**register**メソッドを使用して登録することもできます。指定したレジストリが見つからない場合、**getRegistry()**メソッドはnullを返すのではなく、**IllegalArgumentException**をスローします。コンテンツ・ハンドラまたはアプリケーションのレジストリへの登録を解除するには、**unregister(String classname)**メソッドを使用します。

起動（invocation）時には、アプリケーションとコンテンツ・ハンドラとの間で渡す必要のあるパラメータをカプセル化します。URL、タイプ、アクション、ID、さらにはハンドラからのレスポンスの要否など、さまざまなパラメータがあります。このカプセル化した情報を識別するために、APIでは**Invocation**クラスを使用します。起動では、単にパラメータをコンテンツ・ハンドラに渡すだけではなく、ハンドラからのレスポンスも受け取るということを押さえておいてください。Invocationクラスには、コンテンツ・ハンドラ起動リクエストのある瞬間での状態を示す、いくつかの便利なステータス・フラグ（OK、ACTIVE、CANCELLEDなど）が定義されています。

レジストリを作成し、さらにInvocationインスタンスを使用してターゲット・パラメータを作成したら、レジストリの**invoke(Invocation invoke)**メソッドをコールして実際にコンテンツ・ハンドラを開始する必要があります。このメソッドは**mustExit**フラグを返します。これは、コンテンツ・ハンドラ自体が起動する前にコール元アプリケーションが終了する必要があるのかどうかを示すものです。

したがって、コンテンツ・ハンドラ起動イベントの流れは次のようになります。

1. JADマニフェスト・エントリを作成するか、プログラムで**register**メソッドを使用して、コンテンツ・ハンドラを登録します。
2. **getRegistry**メソッドでハンドラのレジストリにアクセスします。
3. **Invocation**クラスを使用して、起動リクエストを作成しパラメータを指定します。
4. レジストリの**invoke(Invocation invocation)**メソッドを使用して起動します。

ContentHandlerとContentHandlerServer

ContentHandlerインタフェースおよび**ContentHandlerServer**インタフェースは、前述のレジストリの反対側に位置するものです。つまり、これらのインタフェースは独自のコンテンツ・ハンドラをコーディングするための手段となります。ただし、これらのインタフェースを直接実装することはありません。これらのインタフェースの実装は、デバイス・メーカー独自のAPI実装として提供されます。この実装を利用して、ハンドラのコーディングを行います。

ContentHandlerインタフェースは、各コンテンツ・ハンドラ（すでに登録済みのものも含む）の登録に関する詳細をまとめたものです。たとえば、**getID()**や**getType()**などのメソッドが返すのは、基本的には登録時に指定した詳細のみです。

ContentHandlerServerインタフェースは**ContentHandler**インタフェースを継承しており、新しい起動リクエストの受信メソッド、これらのリクエスト処理の完了メソッド、その他のメタ情報の取得メソッドなどを利用できます。一言で言えば、この実装では、新しい起動リクエストを管理するためのライフサイクル関連メソッドを利用できます。そのため、独自のコンテンツ・ハンドラを作成する場合は、このライフサイクル（リクエストのキューイング、新しいリクエストへの応答、リクエストのチェーン処理、エラー処理など）を管理するメソッドを使用します。このようにして、プログラマーはコンテンツ・ハンドラのコーディングを行いますが、そのオーバーヘッドは（デバイス・メーカーのAPI実装による）インタフェースの実装に委ねられます。

RequestListenerとResponseListener

コンテンツ・ハンドラの実装では、新しいリクエストの実行時とレスポンスの要求時のためのリスナーを処理することをお勧めします。リクエスト実行時のリスナーは**RequestListener**、レスポンス要求時のリスナーは**ResponseListener**です。**RequestListener**インタフェースを実装することで、コンテンツ・ハンドラで**invocationRequestNotify(ContentHandlerServer handler)**メソッドの実装を利用できます。このメソッドは、コール元のクラス（またはその他のアプリケーション）から新しいリクエストが送信されたときに自動的にコールされます。また、**ResponseListener**インタフェースを実装することで、コンテンツ・ハンドラで**invocationResponseNotify(Registry registry)**メソッドの実装を利用できます。このメソッドは、コール元のクラスがコンテンツ・ハンドラにレスポンスを要求するたびにコールされます。

RequestListenerインタフェースは、**ContentHandlerServer**インタフェースの対応する**setListener()**メソッドを使用して設定します。一方、**ResponseListener**インタフェースは、**Registry**クラスの対応する**setListener()**メソッドを使用して設定します。

AdvancedImageContentHandler：コンテンツ・ハンドラ実装例

前述の問題領域に引き続き、ここではコンテンツ・ハンドラの例について説明します。この例では、表示する各イメージの端にテキスト文字列を追加します。このクラスのコンストラクタは**リスト1**のとおりです。

AdvancedImageContentHandlerコ

ンストラクタでは、レジストリを使用して、要求される可能性のあるレスポンスのリスナーとして自身のインスタンスを設定します。そのため、**invocationResponseNotify(Registry registry)**メソッドを実装しています。ただしこの例ではこのメソッドは空です。次に、レジストリの静的メソッド**getServer(String classname)**を使用して**ContentHandlerServer**を取得します。さらに、今後コンテンツ処理の新しいリクエストのすべてがこのクラスで**invocationRequestNotify (ContentHandlerServer handler)**メソッドにより実行されるということを、ハンドラに通知します。最後に、UIを設定し、イメージとその下部のテキストを表示します。

新しいリクエストが開始されると、**invocationRequestNotify(ContentHandlerServer handler)**メソッドがコールされます（**リスト2**）。

まず、既存のリクエストがあるかどうかをチェックし、コンテンツ・ハンドラ・サーバーがそのリクエストを完了するように指示します。次に、既存のリクエストがない場合は、新しいリクエストの詳細を取得し、その新しいリクエストを詳細とともに**displayImage()**メソッドに渡します。

リスト3に示されているとおり、**displayImage()**メソッドはかなり単純ですが、ここにはテキストを追加してイメージを表示するためのすべての処理が行われています。

リクエストされたファイルへの接続をオープンし（FileConnection APIを使用）、イメージが見つかった場合は、テキストを追加してそのイメージをフォームに表示します。エラー・チェックを行って、イメージがロードできることを確認します。ロードできない場合は、適宜メッセージを表示します。コール元では、このコンテンツ・ハンドラのコールに必要となるのは次の3つの手順だけです。**リスト4**は、コール元クラス**CHAPIExample**からこれらの手順に該当する部分を抜粋したものです。

手順1では、**getRegistry(String classname)**メソッドを使用してレジストリを作成します。引数としてコンテンツ・ハンドラのクラス名（**AdvancedImageContentHandler**）を渡します。手順2では、起動用のデータを作成します。最後の手順3では、手順2で作成した起動データを引数として渡し、レジストリの**invoke(Invocation inv)**メソッドでコンテンツ・ハンドラを起動します。これらすべてを行う前に、JADファイルで（またはプログラムから）コンテンツ・ハンドラを登録しておく必要があります。JADファイルの内容は**リスト5**のようになります。

最後の3行に注目してください。これらは、コンテンツ・タイプimage/jpgまたは拡張子.jpgの場合に、**AdvancedImageContentHandler**クラスをハンドラとして使用することをAMSに通知しています。その後、AMSがこのクラスをレジストリ内に登録すると、「開く」コマンドの処理でこのクラスを利用できるようになります。

注：このコード例を実行するときは、イメージをルート・フォルダに配置してください。私はエミュレータとして**DefaultCLDCPhone1**を使用しました。Windows Vista/SDK 3.0環境でのそのルートは、C:¥users¥username¥ javame-sdk¥3.0¥work¥devicenumber¥appdb¥filesystem¥root1です。●

参考情報

- Java Community Process（JCP）Webサイトで、JSR-211 API最終リリースを読む
- Java MEをダウンロードする

リスト1 リスト2 リスト3 リスト4 リスト5

```
public AdvancedImageContentHandler() {

 // notify the registry that this class is a listener
 registry = Registry.getRegistry(this.getClass().getName());
 registry.setListener(this);

 // now, get the handler which was registered by making entries in the JAD

 // file
 try {
  handler = Registry.getServer(CH_CLASSNAME);
 } catch (ContentHandlerException che) {
  System.err.println("Registration not done! Check JAD file");
 }

 // this class is the handler for all new requests
 handler.setListener(this);

 // setup the ui
 display = Display.getDisplay(this);
 form = new Form("Advanced Image");
 backCommand = new Command("Back", Command.BACK, 1);
 form.setCommandListener(this);
 imageItem = new ImageItem(null, null, Item.LAYOUT_CENTER, "--");

}
```

リスト全体をテキストで表示する

Dick Wall

Java仮想マシンで動作するScala

Scalaプログラミング言語が教えてくれるJVMの強みと限界とは

Scalaは、さまざまな点で興味深いオブジェクト指向言語です。.NETのF#と同じく、関数型とオブジェクト指向の特性を併せ持つハイブリッドな言語です。Javaよりも静的型付けの要素が強く、非常に強力な型システムがあります。関数リテラルとクロージャ（閉包）、ミックスイン（トレイト）、プロパティ、末尾呼出しの最適化、構造的型付け（静的なダック・タイピングの一種）、パターン・マッチング、並列処理に対処するための新イディオムなどの数多くの機能をサポートしています。また、ScalaはJVMを強く後押ししていると言っても差し支えないでしょう。特に型システムについて考えた場合、おそらくどのメインストリーム言語よりも強くJVMを後押ししています。そのためScalaを利用すると、JVMがその登場時には想像すらされなかったようなタスクにどう対処するのかがよくわかります。

この記事では、次の機能について検討します。

- 記号を使用したメソッド名（演算子のオーバーロードと間違われやすい）
- 末尾呼出しの最適化
- 関数リテラルとクロージャ
- トレイト
- 暗黙のマニフェスト
- パターン・マッチング

Java独自の機能セットを上回るScalaの機能を網羅しているわけではありませんが、これはJVMをある程度後押ししている機能のリストと言えます。問題なく動作する機能もあれば、制約に直面して理想どおりには働かない機能もありますが、何にせよすべて動作します。想定外の難しいジョブが次々と発生するような状況にJVMがどのように耐え、さらには対処してきたかということは、JVMの設計を証明するものとなります。

記号を使用したメソッド名

Scalaには演算子のオーバーロード機能があるというのは、よくある誤解です。確かにScalaには、演算子オーバーロードのように見える2つの機能がありますが、実際はもっとシンプルで**一貫性に優れた**機能です。

この2つの機能の1つ目が、中置演算子です。パラメータが1つのScalaメソッドは、2種類の方法で呼び出すことができます。**リスト1**に単純な例を示します。

注：これ以降の例では、太字はScalaのキーワードを、斜体はScalaシェルからの出力を示します。

この例では、整数値の**java.util.ArrayList**を作成します。（Scalaには独自の非常に優れたコレクションがありますが、それについては後で取り上げます。）

次に、**add**メソッドを使用して整数値1を**ArrayList**に追加します。この形式はJavaと同様です。

ここで、呼び出し方を少しだけ変えます。次の行では整数値2をリストに追加していますが、このときは中置演算子の形式を使用しています。**add**は1つのパラメータのみを取るため、Scalaではこのような形式を使用できます。結果は**a.add(2)**を呼び出した場合とまったく同じとなり、異なるのは構文のみです。これが、一見すると演算子をオーバーロードしているように見える1つ目の機能です。

写真：Geneviève Arboit

1990年代初頭にJavaおよびJava仮想マシン（JVM）が初めて世に出たときには、仮想マシンの概念は新しいものではありませんでした。仮想マシンとは、仮想的な機械語にコンパイルされたコードの実行環境です。仮想マシンを利用すれば、コンパイルされたプログラムを、多数の「現実の」ハードウェア・プラットフォームにまたがる共通ランタイムで実行できます。仮想マシンの概念は、遅くとも

90年代

1966年には（BCPL言語の中間コードである）O-codeによって実現されましたが、時間とともに仮想マシンはコードの実行速度が遅いという評価が広まっていきました。この速度に関する評価を覆したのが、JVMとJava2で導入された実行時（JIT）コンパイラです。悪評が完全になくなったとはいえないまでも、その大部分は過去のものとなりました。JavaとJVMの出現により、仮想マシンとその上で動作するマネージド言語は、特に企業システムの分野において、ソフトウェア開発の主流に躍り出ました。そして2001年前後に、Microsoftが共通言語ランタイム（CLR）と.NETを開発戦略として採用したことにより、仮想マシン技術の主流化は決定的となったのです。

1966

2つ目の機能は、記号を使用したメソッド名です。Scalaのメソッドや関数の**定義名**はアルファベット文字だけに限定されません。実のところ、禁止されている名前はほとんどありません（ただし、数字で始まる名前は使用できません。これは数値リテラルとして解析されます）。アンダースコアを単独ですることはできず、=と==は混乱を避けるために予約されています。しかし、その他のほとんどの記号は（_と=も2文字以上であれば）使用できます（**リスト2**）。

ArrayBufferは、Javaの**ArrayList**によく似たScalaのコレクションですが、**+=**などの名前のメソッドが含まれています。これらのメソッドには、メソッド名にアルファベットではなく記号が使用されている以外に特別なところはありません。実際、**ArrayBuffer**の**append()**メソッドは、**+=**メソッドとまったく同じように動作します。この点については、リスト2の後半の**ArrayBuffer**に7を追加する呼出しで確認できます。このメソッドは、他のメソッドとまったく同じように呼び出すことができます。**ab.+=(7)**という形式は、**ab.append(7)**と同じです。

中置演算子の表記法と記号を使用したメソッド名とを組み合わせると、演算子のオーバーロードによく似た表記になりますが、実際は異なります。

演算子のオーバーロードはエラーの原因になりやすく、C++などの言語で見られたあらゆる恐ろしい事態につながるという批判もあります。それが本当だとしても、そのような批判に私はこう問いかけます。コレクション内のすべてのデータを削除する**add**というメソッドと、まったく同じ動作をする**+**というメソッドがあるとして、その問題に大差があるでしょうか。

Scalaの末尾呼出しの最適化

現時点でのScalaの末尾呼出しの最適化機能は非常に限定的です。再帰的な呼出しを行う関数でしか動作せず、相互に呼び出される関数やより複雑な構成では使用できません。継承などの機能が原因で、最適化されるメソッドはfinalまたはprivateにする必要があり、その他の制約も存在します。JVM内部でも末尾呼出しの最適化がサポートされれば、非常に便利になるでしょう。末尾呼出しの最適化はJava言語で実行できるのだからそうすべきであり、JVMを巻き込む必要はないと主張するJVMエンジニアもいます。しかし、実際はもう少し複雑な問題だと私は考えています。JVMにはコンパイラに勝るメリットがいくつかあります。その最たるものは、実行時のプロファイル情報を利用できることです。この情報は、JVMでは他の最適化機能の多くですでに非常に効果的に利用されています。末尾呼出しがJVMでサポートされれば、最適化がそもそも必要かどうか、今あるクラスでそれが可能なのか、といった有用なコンテキスト情報を把握できるようになります。たとえば、実行中のVMに実際にロードされたクラスに基づいて、最適化すべき個所を特定できます。この場合、最適化する個所を判断するために、他のコードに記述されている可能性のある内容をコンパイラで推測する必要はありません。つまり、Scalaでは末尾呼出し機能のサポートにより、Scala自体で実現できることが増えますが、同時にJVM内部でも一定のサポートを行えば、将来はさらに便利で充実した機能となる可能性が高まります。実際、コンパイラとJVMとの間で何らかの連携処理を行うことがもっとも良い結果につながると思われます。

リスト1 **リスト2**

```
scala> val a = new java.util.ArrayList[Int]
a: java.util.ArrayList[Int] = []

scala> a.add(1)
res0: Boolean = true

scala> a add 2
res1: Boolean = true
scala> a
res2: java.util.ArrayList[Int] = [1, 2]
```

リスト全体をテキストで表示する

注目すべき点がもう1つあります。記号名はASCII文字だけに限定されません。数学者はギリシャ文字による表記法を非常に好んで使用します。次のコードは、Scalaでも、基盤となるJVMでもまったく問題なく動作します。

```
scala> def Σ(numbers:Seq[Int]) =
    numbers.sum
Σ:(numbers:Seq[Int])Int
scala> Σ(1 to 10)
res16:Int = 55
```

ここで作成した関数では、関数名に記号**Σ**（シグマ）を使用しています。シグマは、数学の表記法では合計を表します。このコードではまさに、並んだ数値を合計するという目的でシグマを使用しています（数値はこの場合は整数ですが、整数以外の数値も使用できます）。また、上記の**1 to 10**の式では中置演算子が使用されている点にも注目してください。こうした機能はプログラミング言語には不要だと判断する前に、数人の数学者にこのアイデアについてどう思うかを聞いてみるとよいでしょう。

末尾呼出しの最適化

次のScalaコードについて考えてみましょう。

```
def factorial(n:Int, acc:Long = 1):Long =
  if (n <= 1) acc else factorial(n - 1, acc * n)
```

これは、古くからある階乗計算関数であり、再帰的に実装されています。この実装について注目すべき点がいくつかあります。

- 現在の階乗値を再帰処理の間ずっと保持するためのアキュムレータを渡している。
- アキュムレータ（**acc**）パラメータを指定しない場合のアキュムレータ値のデフォルトは1（開始値）である。
- アキュムレータ（**acc**）には**n**が乗算され、その後**factorial**が再帰的に呼び出されている。
- **n**が1以下の場合は、単にその**acc**値を返す。

次のように、よりシンプルな表記で実装することもできます。

```
def factorial(n:Int):Long =
  if (n <= 1) n else n * factorial(n - 1)
```

確かに、このシンプルな表記法も階乗の数学的定義にはほぼ従っていますが、次のような理由から使用すべきではありません。

1つ目の表記法では、再帰的なメソッド呼出しのパラメータ・リストに乗算を渡しているため、アキュムレータとnとの乗算は、factorialが再帰的に呼び出される前に実行されます。2つ目の表記法では、再帰的な呼出しを最初に評価しなければならず、その後にnとの乗算が実行されます。

1つ目の表記法では、再帰的な呼出しが、関数型プログラミングで「末尾（tail）」と呼ばれる位置に置かれています。これはつまり、関数が最後に行う処理のことです。

このようなとき、Scalaでは再帰的な関数をループ関数に自動的に変換できます。関数が最後にその関数自身を呼び出すため、全体をループ化できるのです。これによって、ランタイムの処理が大幅に減少します。再帰は非常に賢明な手法ですが、かなり大きなオーバーヘッドを伴います。それぞれの呼出しでスタック・フレームを作成する必要があり、再帰が深すぎるとスタックがオーバーフローすることがあります。その上、JVMにはループ化されたコードを最適化するための数々の手法（インライン化、変数とレジスタの最適化など）がありますが、必ずしも再帰的なコードでこのような最適化が行われるとは限りません。

関数リテラルとクロージャ

Java言語でのクロージャ（閉包）の導入については多くの議論がなされており、賛否両論です。クロージャと関数リテラルがJava 8で導入されることはほぼ確実となっているようですが、それまでにまだ1～2年はかかります。

その一方で、（大部分とは言わないまでも）多くの代表的なJVM向けの代替言語では関数リテラルとクロージャが利用できます。

ここまでで、私が関数リテラルとクロージャとを別の概念として語っていることをお気づきの方もいるでしょう。まさにそのとおりです。クロージャとは、自身が含まれているスコープにある値や変数の一部を囲む関数リテラルの一種です。

Javaで関数リテラルまたはクロージャが必要とされる場合は、インナー・クラスまたは匿名インナー・クラスで同じ機能を実現できます。これらは問題なく機能しますが、大量の定型的なコードが必要となるため、実現される機能には見合わないほどの追加コーディングが発生する場合も多くあります。たとえば、次のコードは、Scalaの単純な関数リテラルです。

```
scala> val primes = List(2,3,5,7,11,13)
primes:List[Int] = List(2, 3, 5, 7, 11, 13)
scala> primes.map(n => n * 3)
res0:List[Int] = List(6, 9, 15, 21, 33, 39)
```

このコードをJavaの匿名インナー・クラスで表すと大量のコードが必要になるため、ほとんどの開発者は同じ処理を行うためにforループに頼らざるを得なくなります。言うまでもなく、その結果を保持する別のリストを作成して、結果を追加するようコーディングする（または、リスト内の値を適宜変更する）必要があります。これらの代替案にはそれぞれ、コード量が増加するか、ミュータブル（再代入可能）な状態になるというマイナス点があります。特にミュータブルな状態は、並列処理システムでは最終的に問題となる可能性があります。

関数リテラルまたはクロージャの機能の実現にJava匿名インナー・クラスが利用されるという点は非常に重要です。これがScalaでの実装方法となるからです。このため、Scalaファイルをコンパイルすると非常に多くのクラスが生成されることになります（クロージャごとに独自のクラスが生成されます）。JDK 7に追加されているメソッド・ハンドルが、余分に生成されるクラス・ファイルの数の削減に役立つ可能性があります。それによってコンパイルの速度が上がり、さらにはコンパイル後のバイナリのサイズが減少するかもしれません。

一方で、これはJVMが現在、比較的順調に機能している領域でもあります。代替言語でクロージャを利用することによる最大のデメリットは、現時点で標準的なアプローチがないことです。このため、Scalaのクロージャと、Groovyのクロージャ やJRubyのクロージャとの互換性は確保されないと思われます。Java 8でクロージャが利用できるようになれば、それが他のすべての言語が従うべき標準となるでしょう。

トレイト

Javaは多重継承の流行に強く抵抗しました。多重継承は非常に強力な機能ですが、いくつかの問題も伴います。たとえば、どのスーパークラスのどのメソッドが実際に参照され

CLRとJVMの比較

MicrosoftのCLRとJVMを比較すると面白いものです。CLRは当初からソース言語にとらわれないことを想定し、C#、J#（Javaの類似言語）、VB.NETなどのさまざまなソース・コード言語をサポートしてリリースされました。それ以降、Iron PythonやIron Rubyなどの数々のサード・パーティのソース言語オプションや、F#などのMicrosoftがサポートする新言語が加わりました。これと比較すると、JVMは、何よりもまずJavaプログラミング言語のサポートに集中していました。それでも、JVMをターゲットとする他の言語が出現し続けています。

実際、JVMのバイトコード実行をターゲットとする多種多様なソース言語が存在します。この要因は、JVMでは複数のプラットフォームの容易なサポートを約束していること、およびランタイムが非常に多くのマシンやデバイスにインストールされていることです。Robert Tolksdorf氏と彼のグループis-researchは、JVMをターゲットとする数百の言語を網羅するリストを長年にわたって提供してきました。

JVMで実行する言語は数多くありますが、そのすべてがJVMの制約の範囲でうまく動作しているわけではありません。JVMは、これらすべての言語の全体像が表す広範な言語機能をサポートすることを想定していなかったからです。

特に円滑に動作している言語は、JVMの機能とうまく連動する傾向があり、また、JVMの制約を回避するための賢明な方法を備えています。これらの言語の多くは、新世代のJVM向け言語の最先端へと変わり始めています。

このような言語には、Ruby（JRuby）、Python（Jython）、Mirah（Rubyを静的型付けにしたような言語）、Clojure、Groovy、Fantomなどがあります。

一方、この記事ではScalaという言語を取り上げています。この言語の考案者であるMartin Oderskyは、JVMとJava言語について非常に精通しています。実際、Java 1.3に使用されたjavacコンパイラはOdersky氏のGJコンパイラをベースとしたものでした。GJは総称型（ジェネリクス）を使用してJavaの型システムを拡張する実験的なコンパイラでした（ただし、Odersky氏は常々、ワイルドカードは彼のアイデアではなかったと言っています）。

るのかという問題（ひし形継承問題）があります。

Javaでは、スーパークラスはクラスごとに1つだけであるという制約が設定されました。しかし、新しいクラスを定義するごとに複数のインタフェースを実装できるため、多重継承に関連する問題なしに、ポリモフィズムの多くのメリットを享受できました。

Javaの考案以降（一部のケースではそれ以前にも）、ミックスインという別の戦略が、完全な多重継承よりも安全な代替策となりました。ミックスインはJavaのインタフェースに似ていますが、さらに振る舞いも定義できます。任意のクラスに実際のスーパークラスが1つしかない点は同じですが、他のより豊富な要素をミックスインすることができます。これは、単なるインタフェース定義と比べて有用といえます（インタフェース定義の場合、実装するためのコードを記述する必要があります）。

Scalaではこのような機能をトレイト（trait）としてサポートしています。驚くべきことに、JVMではトレイトのサポートを想定していなかったにもかかわらず、トレイトはJVMでも問題なく動作します。トレイトを定義すると、インタフェースと、必要な振る舞いと状態を持つクラスの両方が生成され、コンパイラによって巧妙に結び付けられ、全体が円滑に動作します。リスト3は、Scalaのトレイトの例です。

その使用例は次のとおりです。

```
scala> val kermit = new Frog
kermit:Frog = Frog@1dfe1a
scala> kermit.move
I move using 4 legs
scala> kermit.color
res2:java.lang.String = Green
scala> kermit.swims
res3:Boolean = true
```

Frogクラスには実際のスーパークラスは1つしかありませんが、GreenとHasLegsというトレイトがミックスインされています。HasLegsトレイトでは、クラスをインスタンス化する前に足の数（Legs）を指定する必要があります。このため、Frogクラスの定義時に足の数を代入しています。

トレイトはScalaの魅力的な機能であり、Javaで一般的に可能な規模よりもはるかに規模の大きい再利用につながります。また、Javaではアノテーションおよびアノテーション・プロセッサで実行することの多くをトレイトで実行できます。

トレイトは現時点でも、JVMと非常に相性よく動作しています。Java 7およびJava 8で導入されるメソッド・ハンドル機能を通じて、さらに改善される可能性があります。また、おそらくインタフェース・インジェクションのサポートによって、Scalaコンパイラのジョブがシンプルになるでしょう。しかしこれらがなくとも、JVMではトレイトとミックスインが適切に処理されています。

暗黙のマニフェスト

この記事で紹介する最後の機能は、表面的には関係のないように思えますが、実際は、JVMの欠点に関連しています。その欠点とは、型保証のイレイジャ（消去：erasure）の存在と、具体化された型（reified type）の欠如です。

Javaの登場時、Javaには豊富なコレクション・セットが備わっていました。これは、独自のコレクションを作成するか、C++のRogue Waveなどのライブラリを購入するのが普通だと考えられていた当時の言語としては画期的でした。問題は、Javaが強い型付け言語である一方、コレクションでは型が無視されたので、ArrayListを求めると、どんな型のデータでも保持できるArrayListが返ってくることでした。Java 5で総称型（ジェネリクス）が導入されるまでは、コレクションから取得したオブジェクトを特定の型で利用するには、オブジェクトの型をテストした上でキャストする必要がありました。

総称型が導入されてからは、コンパイラでコレクションの型保証がチェックおよび実行され、プログラマーが行わなくても、型のチェックとキャストが処理されるようになりました。これは、コードの信頼性（クラス・キャスト例外の削減）と可読性（ほとんどの場合）を高める大きな前進でした。しかし、コードで総称型が定義されていない、古いコレクションをもとに実行されるコードとの互換性を維持するために、総称型は単なるコンパイラの「作り話」とし、型情報はコレクション内に保存せずに消去することが決定されました。

Javaの総称型は非常に大きな前進なのですが、このような型情報のイレイジャのために、開発者は今でもさまざまないらだちを感じる状況に遭遇します。コレクション全体の総称型を実行時に判別できないため、個々のオブジェクトはコレクションから取得した後にチェックする必要があります。コンパイラに情報があればすべてが順調に行きますが、情報がない場合は、チェックとキャストにあふれた古い嫌な時代に無理やり戻されてしまいます。

Javaでよく見られる回避策は、ライブラリやコレクションの作成者が、今後保存されるクラスまたは現在保存されているクラス、あるいはメソッドでリクエストされたクラスに対するクラス参照を含むパラメータを渡すようにするというものです。

Scalaにはこのような問題を軽減する機能があります。それは、暗黙のマニフェスト（Manifest）という考え方です。**リスト4**に例を示します。

コンパイラでは、総称型の関数またはクラスの定義でこのような暗黙のマニフェストが検出されると、その2つ目のパラメータ・リストに問題となっている総称型のクラスが自動的に追加されます。このため、呼出し元では、関

リスト3 **リスト4**

```
abstract class Amphibian {
  def color: String
  def swims = true
  def breathes = true
}
trait Green {
  def color = "Green"
}
trait HasLegs {
  def legs: Int
  def move = println("I move using %d legs".format(legs))
}
class Frog extends Amphibian with Green with HasLegs {
  val legs = 4
}
```

リスト全体をテキストで表示する

数またはクラスで必要とする場合に、クラスを明示的に指定する必要はありません。

これは、型のイレイジャを部分的に回避するものであり、発生する問題のおそらく80パーセントで十分な対策となりますが、完全なソリューションではありません。この問題は、Scalaの別の機能で真に浮き彫りになります。

パターン・マッチング

Scalaのパターン・マッチング機能は非常に強力で、柔軟性にも優れています。Javaの**switch**文と少し似ていますが、実際は思いつくほぼすべてのことを行うことができます。これは驚くべきことではありません。パターン・マッチングはHaskellやErlangなどの多くの関数型言語の礎となっている機能だからです。

Scalaでは、パターン・マッチングはJavaと同様にリテラルでも、さらにはオブジェクトでも実行できます。文字列は問題なく機能し、クラスでも問題は起こりません。特に、パターン・マッチングでの利用を想定した**case**クラスが便利ですが、どのクラスでも少し作業するだけで、コンポーネントに逆アセンブルしてパターン・マッチングで利用できます。コレクションもマッチング可能なのですが、これこそが、型保証イレイジャが再び問題となる領域です。

例として、**リスト5**を見てみましょう。

この例にはややわざとらしく見えるかもしれませんが、実際にコーディングしているとこのような制限がひょっこりと現れるものです。さらに、実際には単にリストやコレクションだけではなく、どのような汎用クラスでもこのような問題が見られます。

リスト5には、**sumItUp**という関数が定義されており、この関数は**Any**の**List**を引数として取ります。（**Any**は、リストに任意の型を格納できることを表します。Javaの**Object**と少し似ていますが、**int**や**double**といったすべてのスカラー型も含まれる点が異なります。）

この例の実装は十分にシンプルです。現時点で気になるのは、リストの2つの型だけです。リストに整数が含まれる場合は、それらを合計し、文字列が含まれる場合は、整数に変換してから合計したいと考えています。その他の場合は、単純に**0**を返します。

この関数を定義すると、未チェックの型に関する警告が表示されます。ここでまず、何かが間違っていることを察することができます。

この関数を使用すると、最初は問題なく動作するように見えます。**int**の値のリストを指定すると、それらの合計が返されます。順調のように思えますが、文字列のリストを試すと、クラス・キャスト例外が発生してしまいます。なぜでしょうか。

問題は、型保証イレイジャが原因となり、JVMでコレクション内に保存された型の種類を判別できないことです。判別するための唯一の手段は、要素を個々に調べることです。Scalaでは、プログラマーが今何を行っているかを理解していることを前提としているため、**List**が最初のケースで整数のリストであることを示した上で、便宜上（プログラマーのために）、そのリストの型を整数に制限して扱うようになります。ここで問題となるのは、次に文字列のリストが渡された場合、Scalaではそれを整数のリストと区別できないため、そのリストを再度整数のリストとして扱うということです。そのため、リストの要素を利用したとたんに、クラス・キャスト例外が発生します。

Scalaの「unchecked」という警告は、実際に未チェックであるという状況そのものを伝えています。しかし、整数のリストしか発生しないとプログラマーがわかっている場合は型を制限できる方が便利であるため、Scalaはコンパイル・エラーではなく警告のみを発します。

実際、**int**のリストと**String**のリストをコレクション・レベルで区別できる方法はありません。JVMに具体化された型が追加され、コレクションが保存している内容をレポートできるようにならない限りは、今後も区別できないのです。

いくつかの回避策を利用できますが、通常はマッチング処理をネストする方法がとられます（**リスト6**）。

ここでは、任意の型のリストをマッチングし（アンダースコアは含まれている任意の型と一致します。これは、リストの要素について確信できない場合に、総称型のコンテナとマッチングするための適切な方法です）、次に各要素とマッチングして、その値を整数に変換できます。その後、値の合計を行います。この新しい実装には、整数と文字列が混合した状態で合計を算出できるというメリットがありますが、やはり回避策に過ぎません。均質なリストが必要な場合もあるでしょう。その場合は、安全性を確保するための作業がさらに必要となります。

型保証がScala開発者を苦しめるような領域は他にもありますが、パター

痛み止め

Javaの総称型は非常に**大きな前進**だが、このような型情報のイレイジャのために、開発者は今でもさまざまないらだちを感じる状況に遭遇する。コレクション全体の総称型を実行時に判別できない。Scalaにはこのような問題を**軽減する**機能がある。

リスト5　**リスト6**

```
scala> def sumItUp(list: List[Any]): Int = list match {
     |   case listOfInts: List[Int] => listOfInts.sum
     |   case listOfStrings: List[String] =>
     |         listOfStrings.map(_.toInt).sum
     |   case _ => 0
     | }
warning: there were unchecked warnings; re-run with -unchecked for details
scala> sumItUp(List(1,2,3))
res8: Int = 6
scala> sumItUp(List("1","2","3"))
java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
```

リスト全体をテキストで表示する

ン・マッチングではその例がもっとも簡単に示されます。

まとめ

JVMにはすでに、Java言語の要件を超えた柔軟性が備わっています。Java言語では許容されない豊富な記号を使用するメソッド名をサポートし、InvokeDynamicやメソッド・ハンドルを追加することで、現時点でも他の言語を歓迎しています。これは、1つの言語だけに対応するように作成された仮想マシンに期待する範囲を超越しています。しかし、JVMが代替言語にとってもっと優れたものとなるために望まれる機能がまだいくつか存在します。

理想的には、具体化された型、末尾呼出しの最適化、インタフェース・インジェクション、その他この記事では扱いきれなかった機能も備えたJVMが期待されます。これらの機能があれば、Scalaやその他のJVM言語は発展し、より高速に、より一貫性と簡潔性を保って実行できます。これらの機能は、可能性の限界をさらに遠くに押しやるものです。

現実には、これらの機能が追加されても、JVMの一般リリースに組み込まれるにはしばらくの時間がかかるでしょう。また、当面JVMの以前のバージョンをサポートする言語では、これらの機能だけを利用するよう切り替えるにはさらに時間がかかると思われます。メソッド・ハンドルがJava 7で利用できるようになったからといって、インナー・クラスが即座に置き換えられるわけではありません。これを行うと直ちに、Scalaやその他の言語が以前のバージョンのJVMとの互換性を保てなくなるからです。おそらくは、JDK 7にすぐに切り替えることを望まない人もいるでしょう。

しかし、JVMの改善の機運が高まっていることは、将来的にJVM上で代替言語がサポートされる十分な証です。当面の間は、少なくともJDK 8でJavaのクロージャと関数リテラルが導入されることを期待できます。幸運であれば、それらの機能がクロージャの標準となり、あらゆる言語が従うことでプラットフォーム上のすべての言語間での相互運用性が向上するでしょう。

また、より高度な機能が追加されるまでの当面の間は、JVM向け代替言語の開発者が、制約があるからとあきらめることなく、完全とはいかなくとも、問題に関する賢明な回避策を見つけることを私は確信しています。●

参考情報

- Scala Webサイト
- The Da Vinci Machine Project
- Project Lambda
- Scalaの型とイレイジャの詳細を見る
- Java Posseポッドキャストを聞く
- Scalaをダウンロードする
- Scalaのトレーニングを受ける
- 『Programming in Scala, Second Edition』(Artima、2010)

//クイズ-コード間違い探し /

第1回「コード間違い探し」のコーナーです。

ここでは、あなたのコーディング・スキルを試します。毎号、コードに関する難問を出題し、次の号で正解と解説を掲載します。また、投稿者の解答とその比率も掲載しますので、ご自分の実力を他の投稿者と比較できます。

第1回目は、オラクルのJavaエバンジェリストArun Guptaからの問題です。

1 問題

Contexts and Dependency Injection(CDI)は、Java EE 6プラットフォームで新たに導入された仕様です。この仕様では、Webアプリケーションへの型保証(type-safe)された標準ベースの依存性注入について定義されています。CDIでは、JSFとEJBのプログラミング・モデルを統合し、EJBをJSF バッキングBeanとして使用できるようにすることでWebとトランザクション層とのギャップを埋めています。

2 コード

JSF バッキングBean用の次のコードについて考察します。

```
@Named @Stateless
public class MyBean {
  public void save() {
    // データベースへの永続化のためのビジネス・ロジック
  }
}
```

WAR構造は次のようになります。

```
WEB-INF/classes
                /MyBean
index.xhtml
```

ヒント：これは、CDI対応アプリケーションの構築時にもっともよく見られる間違いです。

3 修正点はどこか

@Namedを使用すると、JSFページ用のxhtmlファイルで、式言語#{myBean.save}でこのBeanにアクセスできます。では、このJSFページにEJBをインジェクションできない理由は何でしょうか。

1) EJBでインジェクションを有効にするには、EJBをJARファイルまたはEARファイルにパッケージ化する必要がある。
2) インジェクションを有効にするには、「beans.xml」が必要である。
3) EJBのビジネス・メソッドを呼び出すには、そのパラメータとしてActionEventが必要となる。
4) CDIインジェクションは仕様で定義されたクラスからは利用できない。

わかりましたか?

解答は電子メールで
お送りください。
正解は次号に掲載されます。

編集
編集長
Justin Kestelyn
シニア編集長
Caroline Kvitka
コミュニティ編集員
Cassandra Clark、Sonya Barry、
Yolande Poirier
Javaイン・アクション編集員
Michelle Kovac
テクノロジー編集員
Janice Heiss、Tori Wieldt
補助執筆員
Kevin Farnham
補助編集員
Blair Campbell、Claire Breen、Karen Perkins

デザイン
シニア・クリエイティブ・ディレクター
Francisco G Delgadillo
デザイン・ディレクター
Richard Merch?n
補助デザイナー
Chris Strach、Jaime Ferrand
プロダクション・デザイナー
Sheila Brennan

発行
発行者
Jeff Spicer
プロダクション・ディレクター兼
共同発行者
Jennifer Hamilton　+1.650.506.3794
オーディエンス開発運用、
シニア・マネージャー
Karin Kinnear　+1.650.506.1985

広告/販売
共同発行者
Kyle Walkenhorst　＋1.323.340.8585
米国北西部および中央部
Tom Cometa　+1.510.339.2403
米国南西部、LAD
Shaun Mehr　+1.949.923.1660
米国北東部、EMEA/APAC
Mark Makinney　+1.805.709.4745
メーリング・リスト・サービス
営業担当にご連絡ください。

リソース
オラクル製品
+1.800.367.8674（米国/カナダ）
オラクル・サービス
+1.888.283.0591（米国）
オラクル出版物
oraclepressbooks.com

寄稿について
記事を寄稿してくださる方は、編集者まで電子メールでご連絡ください。

定期購読について
定期購読は無料です。定期購読フォームに必要事項を入力して
ご登録ください。

マガジン・カスタマー・サービス
java@halldata.com **電話**：+1.847.763.9635

プライバシについて
Oracle Publishingでは、選ばれたサード・パーティとのメーリング・リストの共有が許可されています。このメーリング・リストにお客様の住所または電子メール・アドレスを含めることを希望されない場合は、カスタマー・サービスまでご連絡ください。

Java Magazineはオラクル（500 Oracle Parkway, MS OPL-3C, Redwood City, CA 94065-1600）によって隔月で発行され、無料で購読いただけます。

デジタル出版：Texterity