

# Java™ magazine

By and for the Java community 

```
package agenda.rest;

@Path("/agenda")
public class AgendaResource {
    private volatile Agenda agenda;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Conference> listConferences() {
        return agenda.listConferences();
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public void addConference(Conference conference) {
        agenda.addConference(conference);
    }

    @GET
    @Path("/agenda/{id}")
    public Conference getConference(@PathParam("id") String id) {
        return agenda.getConference(id);
    }
}
```



ORACLE CLOUD

## イノベーションのための プラットフォーム

Java ベースのサービスが提供する柔軟なツールによって、今までにないタイプのアプリケーションを開発、導入できます。















# JCPの選挙結果

## 2014年秋のJava Community Process (JCP) Executive Committeeの選挙

**結果:**この選挙はVotenet上で行われ、2014年11月10日午後11時59分(太平洋標準時)に投票が締め切られました。

この年の選挙では、8つの批准議席と5つの選出議席について投票を行いました。JCPは2015年にJCP 2.10へと移行するため、今回の議席獲得者は1年の任期を務めることになります。

批准議席を獲得(または再獲得)したJCPメンバーは、Freescale、Gemalto M2M GmbH、Goldman Sachs、MicroDoc、SAP、Software AG、TOTVS、V2COMです。

選出議席を獲得(または再獲得)したJCPメンバーは、ARM、Azul Systems、Hazelcast、Werner Keil氏、Geir Magnusson, Jr.氏です。

新たに選出されたメンバーは、2014年11月25日に議席につきました。

詳しい選挙結果については[こちら](#)をご覧ください。

写真提供: BEJUG

今注目のJAVA SPECIFICATION REQUEST

# JSR 364: BROADENING JCP MEMBERSHIP



## JSR 364: Broadening JCP Membership

は、JCP.nextによる取り組みの第4の要素で、2011年5月にJava Community Process (JCP) チームによって開始されました。JCP.nextは、JCP内部の透明性、参加状況、機敏性、ガバナンスを改善することを目指しています。JSR 364の具体的な目標は、「新しいメンバーシップ・クラスの定義、既存のメンバー

シップ・カテゴリの変更、コミュニティからの参加の実現、およびJCPメンバーからの、知的財産に関する適切な合意の取得によって、より広い範囲からJCPへの参加を促すこと」です。

**Heather VanCura氏** (写真) がJSR 364の仕様リードです。この専門家グループには、London Java Community (LJC) とSouJavaの代表者や、大小の企業の代表者が集まっています。参加者全員が、より幅広いJava開発者コミュニティとJCPの関わりを改善し、交流を深めることを目指しています。

JSR 364の進捗状況をご覧になる場合や、この取り組みへのご意見や新しいご提案がある場合は、[Java.net](#)のJSR 364プロジェクトにアクセスしてください。このページでは、JSR 364に皆さんの意見を反映するための各種の手段を取り揃えています。















報源から学んだものを自分の練習に取り入れる。

- 離：技能を完全に身に付ければ、新しいやり方で独自の手法を編み出すことができる。

アスリートや音楽家は、試合やコンサートに臨む前に激しいトレーニングを積む必要がありますが、一般にプログラミングの世界では、開発者は準備せずに本番に挑みます。開発者にとっては、Code Kata が練習の良い機会になります。

Code Kata は、テスト駆動型開発や、JUnit や TestNG などのテスト・フレームワークとともに使うと効果的です。これらのテスト手法は、反復的、段階的な問題解決に役立つからです。

**Java 8 のストリームとラムダ式の「形」を練習する (左側の記事を参照)。**

今すぐ Java 8 の新しいコードを使って練習したい皆さんに向けて、スキルを磨くための、Java 8 のストリームとラムダ式のシンプルな「形」をご用意しました (JUnit のテストも含まれています)。この Code Kata のソース・コードは [GitHub](#) からダウンロードできます。

ラムダ式をよく知らない場合は、まず必要な構文を使ったテストを通すことを目指してください。その後、少しずつラムダ式のコードをリファクタリングして、テストが通ることを必ず確認してください。楽しんでくださいね。

今、多くの開発者が Java 8 の新機能を学ぼうとしています。Code Kata を実践する上でまたとないチャンスです。スキル向上の最初のステップを踏み出しましょう。

**アクション:** Java 8 のストリームとラムダ式の「形」を練習する(左側の記事を参照)。

**コミュニティ:** Java ユーザー・グループに参加する

立ち止まらずに、もっと大きな開発者コミュニティに参加しましょう。他の開発者と知識を共有し、一緒に作業を進めることは、自分にとって居心地のよい世界を抜け出し、学習のスピードを上げ、幅を広げる良い手段です。

Java コミュニティは、オープン、多様性、一体性という考えに基づく魅力的な世界であり、複数のプラットフォームをカバーし、多くのベンダーや言語、世界中の人々が参加しています。Java コミュニティは、開発者として探究し成長するための優れた場です。

私たちは学び、スキルアップしていかなければなりません。しかし、自分を試すことがなければ、新しいアイデアに触れる機会がなければ、あるいは自分の選択に疑問を持つことがなければ、無為に過ぎて終わるだけです。同じ興味と情熱を持ち、難しい質問をしてくれる人々が密集した場に参加することは、自分のキャリアアップと私生活の向上の大きな原動力となります。活動中の [Java ユーザー・グループ \(JUG\)](#) に参加すれば、同じような問題を抱え、違う解決策を知っている友人、あるいは同じような解決策を違う問題に適用している友人が簡単に見つかります。経験やチャンスについて情報交換することで認識が変わり、

いつもとは違うことに挑戦できます。

開発者コミュニティでの交流は、難しく楽しい経験です。コミュニティでは、ハッカソンや Adopt-a-JSR、Dojo、ライトニング・トークなどの実践的活動によって、双方向学習が進められています。これらの JUG の活動は、参加者全員のコーディング能力、考察能力、説明能力を高められるものになっています。

**アクション:** JUG のメーリングリストに登録し、次回のミーティングに参加する。

**アクション:** 地域の JUG で、実践的活動に参加する。そのような催しがなければ、催しを1つ提案し、その運営を手伝う。小さな学習グループでもキャリアに大きなインパクトを与える可能性がある。

**いつもとは違う考え方:**

クラウド・コンピューティングを探究する偉大な開発者はどんな意見にも耳を傾けます。私たちの脳では、多くの考えに触れるほど、それぞれの考えがより多く関連付けられ、大きな問題の解決につながります。テクノロジーによっては、たとえ日常業務に関わりがなくても、開発者の視野を広げてくれる可能性があります。

プログラミングは、制約のある環境で、自分のスキルを使って解決策を作り上げるアート的一种です。開発者はそれぞれの環境で、限られたツールや技法から選択を行いますが、その過程で解決策に対する独自の見方が生み出されます。時が経てば世界は変わり、制約も変わります。新しい制約によって、新しいツールと技法

による解決策が生まれ、そのようにして良い状況が維持されます。

クラウド・コンピューティングの話題はあふれていて、もう聞き飽きたかもしれませんが、それで興味をなくしては駄目です。クラウドに関わる概念は新しい思考法を生み出しますし、開発者の能力を広げ、開発者のキャリアに新しいチャンスをもたらすことができます。

開発者は、コンピュータを駆使し、メモリのアドレスを処理し、基盤となるストレージ・システムを利用しています。そのコンピュータを捨て、メモリを取り外し、ストレージも処分してしましましょう。全部ネットワークに変えるのです。そうすれば、アプリケーションに何が起きるでしょう

か。クラウド・コンピューティング、ビッグ・データ、マイクロサービス、コンテナ、Platform as a Service (PaaS)。これらは今までにないアプローチであり、存在しないコンピュータや、どこかに無限にあるかのように振る舞うメモリやストレージを扱います。

もちろん、そのすべての基盤部分で実際のコンピュータがどこかに存在し、アプリケーション・サーバーやオペレーティング・システムが稼働していることは誰もがわかっています。しかし、ここでの目標は、一歩引いて新しい思考法を試すことです。

クラウドは、今までとは違う考え方をやるチャンスです。制限が一切ないかのように振る舞ってみましょう。そうすれば、絶対作ることができないと思っていたも

ひとりで飛び出さない  
「速く行きたければひと  
りで行きなさい。遠く  
に行きたければ、みんな  
で一緒に行きなさい」  
—アフリカのことわざ







MICHAEL KÖLLING

**Michael Kölling** : イギリスのカンタベリーにあるケント大学コンピューティング学部教授。オブジェクト指向システム、ソフトウェア・ツール、プログラミング言語、コンピューティング教育、HCI を研究分野としている。Java 教本 2 冊を出版。BlueJ および Greenfoot の主要開発者。

写真 : JOHN BLYTHE

# パート2 Raspberry Pi での Java コーディング

## BlueJ による Raspberry Pi ハードウェアへの対話型アクセス

**本**シリーズのパート1では、Raspberry Pi で BlueJ を実行し、2 台目のマシンを使用せずに、ポケット・サイズのコンピュータで Java プログラミングを直接行う方法について説明しました。デスクトップ・マシンで Java プログラムを開発して JAR ファイルを Raspberry Pi に送信する必要はもうありません。Raspberry Pi をデスクトップの代わりに使用して、Java を研究し、学習できる明るい未来がここにあります。

ただ、パート1では、デスクトップでも可能なことを Raspberry Pi で行っただけです。標準的な Java のサンプルと BlueJ を実行し、デスクトップやラップトップで提供されてきた機能を利用しました。

誤解しないでほしいのですが、標準的な Java プログラムの記述方法を学習するためだけに Raspberry Pi で BlueJ を使用することに問題があるわけではありません。Raspberry Pi と BlueJ の組合せは教育に適した素晴らしいツールです。そして、

Raspberry Pi を低価格で購入できる点が世界を大きく変えています。しかし、そこで終わる必要はありません。

次の楽しいステップは、以前は不可能であったことに取り組むことです。たとえば、Raspberry Pi の物理的なコンポーネントに Java から直接アクセスすれば、Java コードだけでなく、ハードウェアでもさまざまなことに挑戦できるようになります。容易にアクセスできる Raspberry Pi のハードウェアと、BlueJ の対話型の実験機能があれば、魅力的なことを実現できます。

本記事では、Raspberry Pi のさまざまなハードウェア・コンポーネントに直接対応し、簡単に使用できる Java オブジェクトの作成方法について見ていきます。

### 前提

本記事では、Raspberry Pi がセットアップされていること、ログインして X Window System サーバーを起動できる状態にあるこ

と、および BlueJ をすでにインストールして起動していることを前提とします(これらの前提を満たしていない場合は、先に本シリーズのパート1をお読みください)。

さらに、本記事のサンプルに従って作業を進めるためには、追加のアイテムとして、LED1 個、ボタン1個、抵抗1個、ワイヤー4本が必要になります(図1)。また、必須ではありませんが、ブレッドボード(実験用基板)があれば便利です。ブレッドボードを使用すれば、はんだ付けをせずに部品同士を簡単に接続することができます。ワイヤーの端をねじって部品を接続するだけでは、すぐにはずれてしまいます。

**BlueJ** プロジェクトの取得  
まず、BlueJ プロジェクトを開きます。RasPi-IO プロジェクトをこちらからダウンロードし、解凍して、Raspberry Pi 上の BlueJ で開きます(図2)。

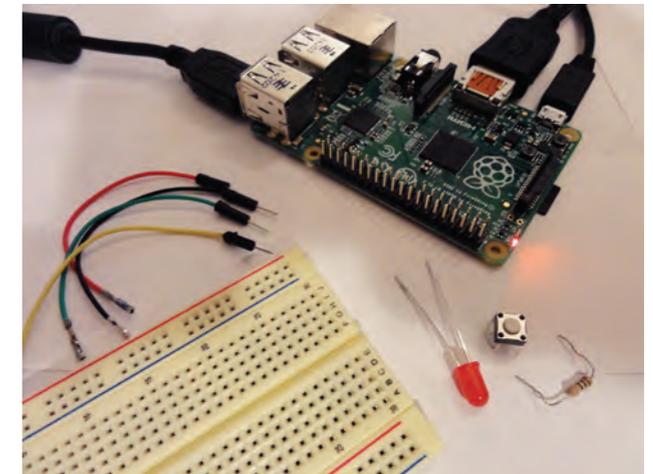


図1

このプロジェクトには、Raspberry Pi に接続可能な入出力部品を抽象化した2つのクラスが含まれています。1つは **GPIOOutput** クラスで、LED など、汎用入出力(GPIO)ピンに接続できる任意の部品を表します。もう1つは **Button** クラスで、GPIOピンに接続されたプッシュ式のボタンを表します。

### LED の準備

最初に取り上げる例は、LED を点灯または消灯するという非常に簡単なものです。

まず、お手元の LED を以下の手順で Raspberry Pi に接続































参照)。

## Java EE 7 によるコード削減

EJB ベースの `Executor` のカスタム実装には、大きな欠点があります。並列リクエスト数の制限や、基盤となるスレッド・プールの設定変更を行う標準的な方法がないことです。JSR 236 (Concurrency Utilities for Java EE) に対応した Java EE 7 には、Java SE の `java.util.concurrent.ExecutorService` のマネージド実装が付属しています。`ExecutorService` は `Executor` インタフェースを拡張しており、その結果、カスタム EJB 実装は過去のものとなりました。

### リスト 6 の

`ManagedExecutorService` が、`ExecutorService` のマネージド実装です。これはアプリケーション・サーバーのマネージド・リソースであり、`@Resource` アノテーションによってインジェクションされます。

**リスト 7** は、POJO の内部で `String` を加工する例で、保存の前に `Normalizer#normalize` メソッドで入力を処理しています。

Java SE 8 では、`java.util.function` パッケージに便利な SAM が数多く組み込まれています。**リスト 8** に示すように、`Consumer<String>`

## コードの削減

Java SE 8 の力を少し借りると、複数のコンポーネントを連結して、コードを 1 行に減らすことができます。

同時に、Java EE 7 が実用的な機能を提供しています。

に接続し、柔軟な実行パイプラインを実現します。**リスト 8** の例は、`CompletableFuture` によって大幅に簡略化できます。**リスト 9** では、`Storage` と `Normalizer` を `CompletableFuture` で接続しています。

`Function` インタフェースと `Consumer` インタフェースへの割り当ては、単なるデモの目的で行われています。なお、**リスト 9** のコードは、アプリケーション・サーバーのマネージド・スレッド・プールでは実行できません。このコードを実行できるのは、「共有プール」と呼ばれる `ForkJoinPool` のインスタンス `ForkJoinPool.commonPool()` です。名前が「Async」で終わるすべ

インタフェースは `Storage#store` メソッドの抽象化に使用でき、`Function<String,String>` は `Normalizer#normalize` メソッドのプレースホルダとして使用できます。

### Java SE 8 でさらにコードを削減

Java SE 8 には、協調動作をサポートする魅力的なユーティリティとして `java.util.concurrent.CompletableFuture` クラスが付属しています。`CompletableFuture` は `Supplier` インスタンスを `Function` や `Consumer`

リスト1

リスト2

リスト3

リスト4

リスト5

リスト6

```
@Stateless
public class StorageService {
    @Asynchronous
    public void save(String content) {
        //heavy lifting
        System.out.println("Storing " + content);
    }
}
```

すべてのリストのテキストをダウンロード

てのメソッドには、2 つ目のパラメータとして `Executor` インスタンスを渡せるようになっています。`ManagedExecutorService` は `Executor` であり、マネージド・アプリケーション・サーバーのスレッドでパイプラインを実行するために使用できます。**リスト 10** では、`Executor` として `ManagedExecutorService` を渡し、SAM としてメソッド・ハンドルを使用しています。

Java SE 8 の力を少し借りると、複数のコンポーネントを連結して、コードを 1 行に減らすことができます。それと同時に、Java EE 7 が、`ManagedExecutorService` の管理、監視、設定などの実用的な機能を提供しています。

### 発生する例外

**リスト 10** のパイプラインで発生し

た例外はすべて、通知されることなく飲み込まれてしまいます。この「便りが悪いのは悪い知らせ」的な振る舞いは、出発点としてはよくても、大半のユースケースでは許容できないでしょう。外側の `StorageService#save` メソッドは、パイプラインがタスクを処理する前に終了する可能性があります。`CompletableFuture` は、ブロッキング式の例外処理と、非同期式の例外処理の両方を提供しています。`Future#get` メソッドは呼出し元をブロックし、戻り値を返すか、または `ExecutionException` を返して発生したエラー原因を知らせます。**リスト 11** は、例外のブロッキングの例です。

**リスト 11** の `get()` メソッドによって例外は飲み込まれなくなりますが、呼び出すメソッドは `CompletableFuture` がすべてのタス

クを終えるまでブロックされてしま  
います。**リスト 12** は、非同期式の例  
外処理の例です。例外発生時に非  
同期で呼び出される `exceptionally()`  
メソッドには、パラメータとして  
`Function<Throwable,[RETURN_  
VALUE]>` を渡します。

`exceptionally` メソッドは、  
`RETURN_VALUE` の型が様々で  
あり、任意の段階で適用できま  
す。たとえば、`supplyAsync` や  
`thenApplyAsync` の直後では、2つ  
目の総称型パラメータは `String` と  
なります。このメソッドに渡すのは  
SAM であり、メソッド・ハンドラに  
よって簡単に実装できます。エラー  
処理コードを専用のハンドラに移す  
ことによって、さらにコードを効率  
化できます。**リスト 13** で  
は、専用ハンドラで例外  
処理を行っています。

例外は非同期的  
に発生するため、  
`StorageService` クラス内  
で意味のある処理を行う  
ことはできません。その  
ため、例外処理は完全  
に専用ハンドラにまとめ  
ることができます。`save`  
メソッドの実装は、処理  
を呼び出すだけで結果は  
考慮しない形になってい  
ます。エラー処理戦略は、  
メッセージ駆動型 Bean  
(MDB) や Java Message  
Service (JMS) と同様で  
す。

非同期コードのエラー

を呼び出し元に伝えることはできま  
せん。そのため、エラーはデッ  
ド・レター・キューとも呼ばれる  
専用の JMS キューに送られます。  
`CompletableFuture` にも、同じメ  
カニズムを JMS を使用せずに適用  
できます。非同期の例外処理には、  
単純な Contexts and Dependency  
Injection (CDI) イベントがもっとも  
適しています。**リスト 14** では、例外  
をイベントとして送信しています。

補足すると、`javax.enterprise.  
event.Event#fire` メソッドは、総称  
型の `Consumer<Object>` です。こ  
のメソッドを使用すると、`Storage`  
への直接参照を、インジェクション  
された別の `Event` インスタンスで完  
全に置き換えることができます。**リス**

## 2つのパワー

Java SE 8 がアプリ  
ケーション・アーキテ  
クチャに与えるインパ  
クトと比べれば、Java  
EE 7 によるインパク  
トはわずかなもので  
す。本当に面白くなる  
のは、この2つを組  
み合わせたときです。  
Java SE 8 のパワーが  
Java EE 7 の生産性と  
融合します。

**ト 15** では、CDI イベント  
を `Consumer<String>` と  
して使用しています。

`Storage` クラスは、ほ  
とんどそのまま使用で  
きます。`save` メソッド  
の `String` パラメータに  
`@Observes` アノテーショ  
ンを追加する必要がある  
だけです。**リスト 16** では、  
`Storage` クラスを CDI リ  
スナーにしています。

## 非同期 JAX-RS 2.0

現在のところ、JAX-RS リ  
ソースでは大したことは  
していません。**リスト 17**  
の `@POST` メソッドと `@  
GET` メソッドは、いずれ  
も同期的に実行されま

リスト7 / リスト8 / リスト9 / リスト10 / リスト11 / リスト12

```
public class Normalizer {
    public String normalize(String input) {
        return "#" + input;
    }
}
//...
@Inject
Normalizer normalizer;

public void save(String input) {
    String content = normalizer.normalize(input);
    executor.execute(() -> storage.store(content));
}
```

すべてのリストのテキストをダウンロード

す。

JAX-RS 2.0 では、非同期リクエ  
スト処理のために `AsyncResponse`  
パラメータが導入されました。  
`AsyncResponse` に `@Suspended` ア  
ノテーションを追加すると、リソー  
スのメソッドはリクエストが完了する  
前に終了できるようになります。こ  
のアプローチによって、実際のビジ  
ネス・ロジックの処理から接続処理  
スレッドを分離できるようになりま  
す。**リスト 18** は、そのような非同期  
リクエスト処理の例です。

待機中のリクエストは、  
`AsyncResponse#resume` メソッド  
の呼び出しによって再開できます。通  
信プロトコルはビジネス・ロジックの  
実装から分離した方が良いため、ビ  
ジネス・ロジックに `AsyncResponse`  
を公開するべきではありません。そ

のため、ビジネス・ロジックはこの  
外側で実装します。ビジネス・ロジッ  
クに `AsyncResponse` を送信する際  
は、その存在を隠ぺいしたまま送信  
する必要があります。

Java SE 8 より前では、  
`AsyncResponse` をカスタム・ラッ  
パー内でカプセル化することが一  
般的でした。しかし、Java SE 8 で  
は、`AsyncResponse#resume` メ  
ソッドを `Consumer<String>` と  
して送信するだけで大丈夫です。  
`AsyncResponse` をビジネス・ロジッ  
クと密接に統合する際に、抽象化や  
インタフェースの追加を行う必要あ  
りません。

また、`@POST` メソッドを手直  
して、`StorageService` への参  
照を完全に取り除くこともできま  
す。`StorageResource#store` メソッ



ドは入力パラメータをバックエンドに送信します。バックエンドは、`Supplier<String>` によって簡単に抽象化できます。パラメータはラムダ式を使用して即座に `Supplier` に変換され、通常の CDI イベントとして送信されます。すでに JAX-RS リソースに存在している Java SE 8 ネイティブのインタフェースを使用すると、`StorageService` のコードをさらに簡略化できます。**リスト 19** では、非同期に `String` を配信しています。

**リスト 19** の `getContent` メソッドは、`Storage#retrieve` メソッドを `Supplier<String>` として使用し、その出力を `Consumer`

プタは、個々のタスク (制御) や境界のモニタリングに非常に適しています。**リスト 20** は単純なパフォーマンス測定インターセプタです。

アノテーションは 1 つだけ (`@Interceptors(Monitor.class)`) でよく、アノテーションが付加されたクラスのすべてのパブリック・メソッドがインターセプトされてモニタリングされます。**リスト 20** の `Monitor` インターセプタは、ロギングに `System.out.println` を使用しています。実環境では、代わりに CDI イベントを使用してモニタリング・データをセンターに送信し、今後の分析に役立てることができます。

## Java SE 8 による JPA コードの削除

Java Persistence API (JPA) を使用すると、`Store` クラスを拡張して簡単に永続化機能を持たせることができます。**リスト 21** では、簡単な JPA 2.1 コードによって永続化を行っています。各ユーザーに対して、`String` の `content` がそれぞれ格納されています。

ここで統計をとってみます。例として、ユーザー 1 人当たりの平均 `String` サイズを計算します。このような計算は、CPU に負荷がかかる可能性があるため、並列処理を行います。JPA 問合せには、ストリーム可能な Java SE 8 コレクションが返されます。Java SE 8 コレクションによって、インメモリの処理や分析を Java SE 8 でシンプルに行う道が開かれます。**リスト 22** では、単純な `Store` の統計を作成しています。

`<Object>` の入力として接続します。また、この例では、`CompletableFuture` とインジェクションされた `ManagedExecutorService` をパイプラインとして使用しています。

## Java EE 7 によるモニタリング

非同期アプリケーションでは、同期アプリケーションよりもモニタリングやデバッグが難しくなります。Java EE 5 で導入されたインターセ

## モニタリング

非同期アプリケーションでは、同期アプリケーションよりもモニタリングやデバッグが難しくなります。インターセプタは、個々のタスクや境界のモニタリングに非常に適しています。

リスト13

リスト14

リスト15

リスト16

リスト17

リスト18

```
public void save(String input) {
    CompletableFuture.supplyAsync(() -> input, executor)
        .thenApplyAsync(normalizer::normalize, executor)
        .thenAcceptAsync(storage::store, executor)
        .exceptionally(this::handle);
}

public Void handle(Throwable error) {
    System.out.println("error = " + error);
    return null;
}
```

 [すべてのリストのテキストをダウンロード](#)

結果リスト全体が RAM (日々安くなっています) に収まるなら、**リスト 22** のように、JPA エンティティ上で Java SE 8 のラムダ式を使用し、簡単に実装やテスト、問合せの実行を行うことができます。Java SE 8 には、さまざまな `Collector` 統計ユーティリティが導入されており、コードを追加することなく必要な計算を行うことができます。`IntSummaryStatistics` ユーティリティ・クラスは、`JsonObject` として簡単に公開できます。たとえば、**リスト 23** では、

`IntSummaryStatistics` を `JsonObject` に変換しています。

並列ストリームにはスレッド管理のオーバーヘッドがかかるため、単純すぎるユースケースでは十分なパフォーマンスは得られません。また、多くのユースケースでは、**リスト 22** は単一スレッドの `stream` を使用する方が `parallelStream` を使用するよりパフォーマンスがおそらく良いでしょう。さらに、非同期 `parallelStream` は、アプリケーション・サーバーのマネージド・スレッドではなく、`ForkJoinPool` が提供す

るスレッドで実行されます。使用可能なフォーク/ジョイン・スレッドの数は使用可能な CPU の数によって異なりますが、システム・プロパティ [java.util.concurrent.ForkJoinPool.common.parallelism](#) で設定できます。アプリケーション・サーバーは、このような「勝手に」作られたフォーク/ジョイン・スレッドは認識しないため、大量のスレッドによってパフォーマンスのボトルネックや堅牢性の問題が発生する可能性があります。

## まとめ

Java EE 7 は多くの注目を集めています。「Java EE」という名前がついているだけで、会議室は一杯になり、ブログ記事は人気になります。しかし、Java SE 8 がアプリケーション・アーキテクチャに与えるインパクトと比べれば、Java EE 7 によるインパクトはわずかなものです。本当に面白くなるのは、この2つを組み合わせたときです。Java SE 8 のパワーが Java EE 7 の生産性と融合します。

本記事の目的は、無駄なプログラミングをなくし、きれいなコードを書くためのアイデアを提供することです。また、本記事に掲載したサンプル・コードは大幅に簡略化しています。String の永続化も、Java SE 8 と Java EE 7 の API を最大限

に使用して行いました。しかし実際環境では、実際のユーザーには意味のないクールなテクノロジーを追い求めるよりも、中核となるビジネス・ロジックに集中するべきでしょう。

</article>

**大変な注目**  
**Java EE 7 は多くの注目を集めています。**  
「Java EE」という名前がついているだけで、会議室は一杯になり、ブログ記事は人気になります。

## LEARN MORE

- [この記事で使用されたサンプル・プロジェクト](#)
- [JSR 346 : Contexts and Dependency Injection for Java EE 1.1](#)
- [「Going Asynchronous With CompletableFuture and Java EE 7」スクリーンキャスト](#)

リスト19

リスト20

リスト21

リスト22

リスト23

```
@Stateless
public class StorageService {
    @Resource
    ManagedExecutorService executor;
    @Inject
    Normalizer normalizer;
    @Inject
    Event<Throwable> exceptionHandler;
    @Inject
    Event<String> sink;
    @Inject
    Storage storage;

    public void save(@Observes Supplier<String> supplier) {
        CompletableFuture.supplyAsync(supplier, executor)
            .thenApplyAsync(normalizer::normalize, executor)
            .thenAcceptAsync(this.sink::fire, executor)
            .exceptionally(this::handle);
    }

    public void handle(Throwable error) {
        exceptionHandler.fire(error);
        return null;
    }

    public void getContent(
        @Observes Consumer<Object> listener) {
        CompletableFuture.supplyAsync(
            thenAcceptAsync(listener, executor)
            .exceptionally(this::handle);
        }
    }
}
```

 [すべてのリストのテキストをダウンロード](#)





まれているが、読みやすくするため、ここでは省略しています。

コードからわかるように、エンドポイントに HTTP で接続するために RestSource を作成し、パス、パラメータ、接続設定（タイムアウト値など）といった一般的な情報を指定しています。次に、RestSource を ObjectDataProvider に渡します。ObjectDataProvider はリクエストを作成し、結果を answer という変数名のプロパティに格納します。そして、ExecutorService インスタンスを DataFX の ObjectDataProvider に渡します。これは、大量のログ情報を送信しなければならない場合

**モニタリング  
Java EE 7 API** を使用すると、高い性能とスケーラビリティを両立し、クライアントとしてのニーズ（WebSocket のサポートなど）にも対応できる**クラウドベースのモニタリング・システム**の開発を行うことができます。

もあることを想定しているためです。すべての通信に1つのスレッドを使用すると転送が遅延する可能性があります。各メッセージに1つのスレッドを使用すると、大量のリソースが消費されてしまう可能性があります。そのため、ExecutorService には通常5つのスレッドが割り当てられます。

DataFX の詳細な説明は、本記事では割愛します。詳細は、

<http://datafx.io> を参照してください。

インターネット接続されたデバイスで多発する問題のひとつは、インターネットとの接続が切断されてしまうことです。当然ながら、接続が切断されたというログ・メッセージは送信できません。そのような場合は、MonitorHandler がすべてのログ・メッセージをファイル・システムに格納します。接続が回復すると、メッセージが IoT モニター・クラウドに送信されます。

### JavaFX モニターのクライアント・アプリケーション

JavaFX モニターのクライアント・アプリケーションは、IoT モニター・クラウドに接続し、オペレータが参照できるモニタリング情報を表示します。IoT モニター・クラウドでは、登録オペレータのリストが管理されています。また、オペレータは1つまたは複数のカード・システムに関連付けられています。カード・システムとは、ひとまとまりとなった CID のグループです。このグループは通常、ある都市のすべての CID です。

そのため、IoT モニター・クラウドの設定において、認証は重要な要素です。図3は、JavaFX モニター・クライアントのログイン画面です。

オペレータが認証されると、監視コンソール（図4参照）が表示されます。

コンソールの左上には、オペレータがアクセスできるカード・システムのリストが表示され、その下には、アクセス可能なカード・システムに



図3

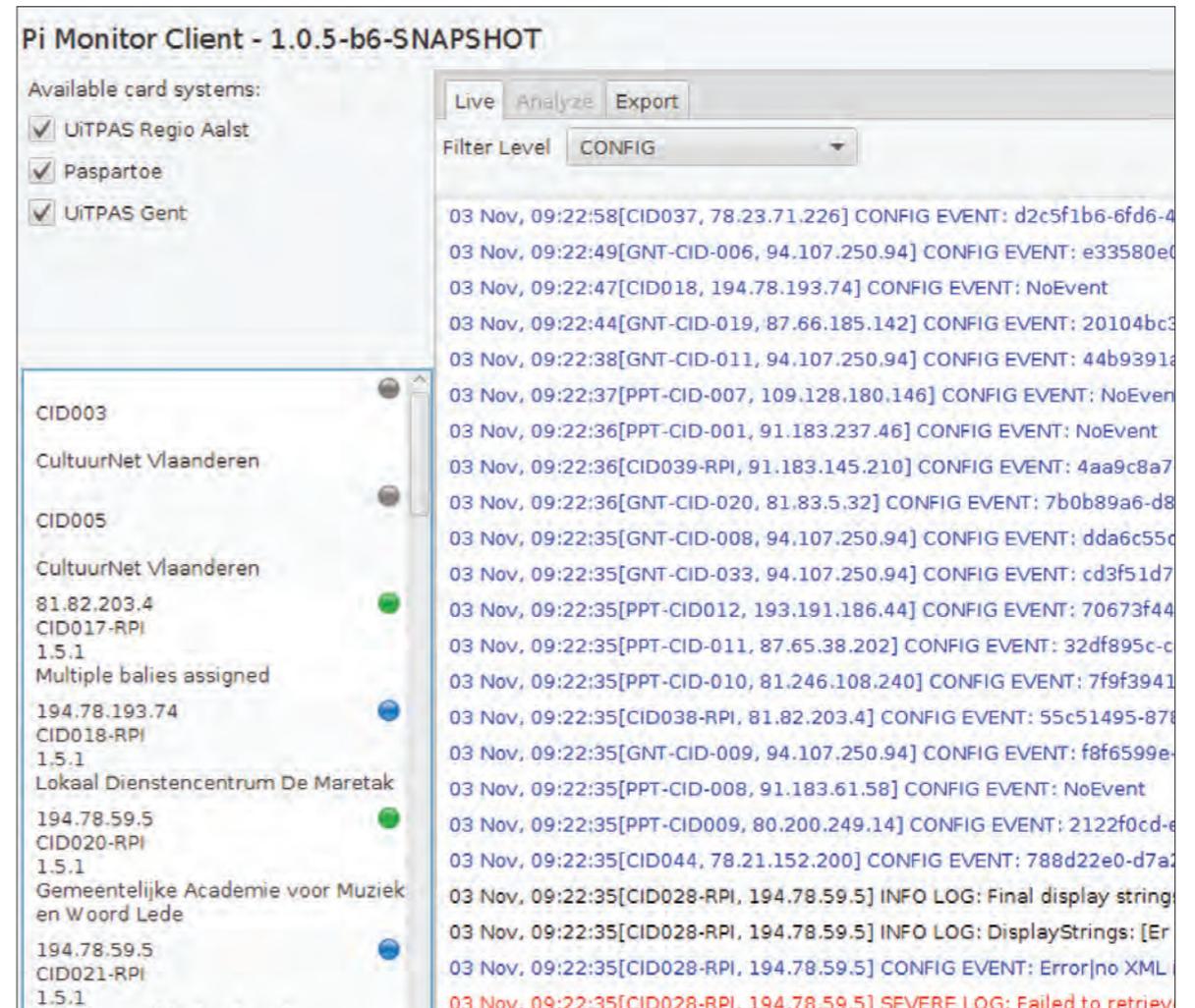


図4











