Joda-Time

時刻を省くことはできません。だったら手間を省いてください

J Steven Perry
Principal Consultant
Makoto Consulting Group, Inc.

2009年 10月 27日

どんなエンタープライズ・アプリケーションでも、時刻を省くことはできません。アプリケーションでは現在の時刻を把握したり、将来のある時点での時刻を把握したり、場合によっては 2 つの時刻からその期間を計算したりしなければなりません。JDK でこの計算を行うのは厄介で面倒なことです。そこで試してもらいたいのが、Java™ プラットフォームのための使いやすいオープンソースの日付/時刻ライブラリーである Joda-Time です。この記事を読むとわかるように、Joda-Time は日付と時刻を操作する面倒を軽減してくれます。

ビジネス・アプリケーションを作成するなかで、日付を扱わなければならないことは頻繁にあります。例えば、私が前回手掛けた保険業界の仕事では正しい日付を計算することが特に重要でした。けれども作業を進めるうちに、`java.util.Calendar` での処理にだんだん苛立ってきました。このクラスを使って日付と時刻の値を処理した経験があれば、これを理解するのがどんなに厄介かはご存知でしょう。そこで、Java の日時関連クラスの代わりとなる Joda-Time というライブラリーがあることを知った私は、このライブラリーを試してみることにしました。手短に言えば、その判断に満足しています。

Joda-Time では、時刻と日付の値を簡単に管理、操作、理解することができます。実際、使いやすさは、Joda を設計する上で第一の目標となっていました。使いやすさ以外には、拡張性、包括的な機能セット、そして複数の暦体系のサポートという目標もあります。しかも Joda は JDK との間に 100 パーセントのインターオペラビリティーがあるので、既存の Java コードのなかでも日付/時刻の計算を行う部分だけを置き換えれば、それ以外の部分は置き換える必要がありません。

Joda 傘下のプロジェクト

実際には、Joda は Java 言語の数々の代替 API プロジェクトを抱える包括的プロジェクトです。そのため厳密に言えば、Joda と Joda-Time が同じものを指しているかのように扱うのは正しくありません。しかし、この記事を執筆している時点で活発に開発が進められていると思われる Joda API は唯一、Joda-Time API だけです。Joda プロジェクトの現状を考えると、Joda-Time を単に Joda と呼んでも問題はないと思います。

この記事では、Joda の概要とその使い方を紹介します。取り上げるトピックは以下のとおりです。

- 日付/時刻の代替ライブラリーとしての有用性
- Joda の主要な概念
- Joda-Time オブジェクトの作成方法
- Joda による時刻の操作方法
- Joda による時刻の書式設定方法

以上の内容のデモンストレーションを行うサンプル・アプリケーションのソース・コードは、ダウンロードすることができます。

Joda の有用性

なぜわざわざ Joda を使うのでしょうか？その理由を明らかにするために、例えば 2000年 1 月 1日の午前 0時ちょうどという時点を作成する場合を考えてみてください。この特定の時点を表す JDK オブジェクトを作成するとしたら、どうすればよいのでしょう？`java.util.Date` を使用すればよいでしょうか？しかし、JDK 1.1 以降のすべての Java バージョンの Javadoc には、`java.util.Calendar` を使用すべきであると記載されているので、そういうわけにはいきません。推奨されていない `Date` コンストラクターが多いことから、このようなオブジェクトを作成する方法はかなり限られてきます。

ただし、`Date` には「今現在」以外の時点を表すオブジェクトを作成するために使用できるコンストラクターが 1 つあります。そのメソッドは引数として、1970年 1月 1日のグリニッジ標準時 (GMT) 午前 0時 (エポックとしても知られています) からのミリ秒数を取り、タイムゾーンを修正します。ソフトウェア開発ビジネスでの Y2K の重要性を考えると、皆さんはこの値をメモリーに保持しておくべきだと思われるかもしれませんが、私はそうしてはいません。`Date` についてはこれくらいにしておきます。

では、`Calendar` についてはどうでしょう。上記の例に必要なインスタンスを作成するとしたら、以下のコードになります。

```
Calendar calendar = Calendar.getInstance();
calendar.set(2000, Calendar.JANUARY, 1, 0, 0, 0);
```

一方、Joda を使うと、コードは以下のようになります。

```
DateTime dateTime = new DateTime(2000, 1, 1, 0, 0, 0, 0);
```

このたった 1 行のコードではそれほどの違いは出てきませんが、今度は問題をもう少し複雑にしてみます。例えば、この日付に 90 日を足した上で、結果を出力したいとします。JDK を使用した場合のコードはリスト 1 のとおりです。

リスト 1. ある時点に 90 日を足して結果を出力する (JDK の場合)

```
Calendar calendar = Calendar.getInstance();
calendar.set(2000, Calendar.JANUARY, 1, 0, 0, 0);
SimpleDateFormat sdf =
  new SimpleDateFormat("E MM/dd/yyyy HH:mm:ss.SSS");
calendar.add(Calendar.DAY_OF_MONTH, 90);
System.out.println(sdf.format(calendar.getTime()));
```

Joda を使用すると、コードはリスト 2 のようになります。

リスト 2. ある時点に 90 日を足して結果を出力する (Joda を使用した場合)

```
DateTime dateTime = new DateTime(2000, 1, 1, 0, 0, 0, 0);
System.out.println(dateTime.plusDays(90).toString("E MM/dd/yyyy HH:mm:ss.SSS");
```

JDK と Joda との違いが多少大きくなってきました (Joda のコードは 2 行で、JDK のコードは 5 行です)。

今度は、Y2K から 45 日後の週の最後の日の日付を出力するとします。正直なところ、このような問題には `Calendar` を使う気にもなりません。この単純な日付の計算でさえも、JDK を使用するのでは、あまりにも面倒です。私が Joda-Time の威力を初めて認識したのは、数年前、まさにこのような問題に取り組んでいるときでした。Joda を使用すれば、リスト 3 のようなコードで日付を計算することができます。

リスト 3. Joda の救いの手

```
DateTime dateTime = new DateTime(2000, 1, 1, 0, 0, 0, 0);
System.out.println(dateTime.plusDays(45).plusMonths(1).dayOfWeek()
  .withMaximumValue().toString("E MM/dd/yyyy HH:mm:ss.SSS");
```

リスト 3 の出力は以下のようになります。

```
Sun 03/19/2000 00:00:00.000
```

JDK の日付処理の代わりとなる使いやすい手段を探しているとしたら、是非とも Joda を検討してください。日付の計算にはどうしても今までどおり `Calendar` を使用したいというのであれば、それはいわば、ハサミを使って芝を刈ったり、使い古した歯ブラシを使って車を洗ったりすることと同じです。

Joda と JDK のインターオペラビリティー

JDK の `Calendar` クラスには使いやすさが欠けていること、そして Joda がこの欠点を補っていることがわかるまでに、そう時間はかかりません。Joda の設計者たちは使いやすさに加え、ある決定も行いました。私はこの決定が Joda の成功の鍵であると信じているのですが、それは JDK とのインターオペラビリティーです。Joda のクラスでは (この後説明するように、多少回りくどい方法になることもありますが) `java.util.Date` の (そして `Calendar` の) インスタンスを生成することができます。そのため、JDK に依存する既存のコードを維持しながらも、日付と時刻の計算に関しては Joda を使って困難な作業をこなせるというわけです。

例えば、リスト 3 の計算を行った後に再び JDK に戻るには、リスト 4 のような変更を加えればよいだけです。

リスト 4. Joda の計算結果を JDK オブジェクトに入力する

```
Calendar calendar = Calendar.getInstance();
DateTime dateTime = new DateTime(2000, 1, 1, 0, 0, 0, 0);
System.out.println(dateTime.plusDays(45).plusMonths(1).dayOfWeek()
  .withMaximumValue().toString("E MM/dd/yyyy HH:mm:ss.SSS");
calendar.setTime(dateTime.toDate());
```

たったこれだけの変更で、計算を行えるだけでなく、その結果を JDK オブジェクトで処理することができます。これは Joda ならではの極めて素晴らしい機能です。

Joda の主要な日付/時刻の概念

Joda が使用する以下の概念は、あらゆる日付/時刻ライブラリーに適用できるはずです。

- 不変性
- インスタント (Instant)
- パーシャル (Partial)
- 暦 (Chronology)
- タイムゾーン

Joda のなかでの上記の概念を順に説明していきます。

不変性

この記事で取り上げる Joda クラスは不変クラスです。つまり、これらのクラスのインスタンスを変更することはできません (不変クラスの 1 つの利点は、スレッドセーフであることです)。後で日付の計算に使用する API メソッドはいずれも対応する Joda クラスの新規インスタンスを返し、元のインスタンスを変更することはしません。API メソッドを使って Joda クラスを操作するときには、操作対象のインスタンスは変更できないため、メソッドの戻り値を取り込む必要があります。`java.lang.String` のさまざまな操作メソッドもこれと同じような動作をするので、おそらく皆さんにはお馴染みのパターンのはずです。

インスタント (Instant)

`Instant` はある特定の時点を表し、エポックからのミリ秒数で表現されます。この定義は JDK と一致します。これが、Joda の `Instant` サブクラスが JDK の `Date` および `Calendar` クラスと互換性を持つ理由です。

一般的に定義すると、インスタントとは時系列における固有の一点であり、一度しか発生しません。したがって、この日付構成体が意味のある形で起こり得るのは一度きりです。

パーシャル (Partial)

パーシャル (この記事ではパーシャル・タイム・スニペットと呼びます) とは、単なる時刻の断片のことです。インスタントがエポックを基準にした相対的なある特定の一時点を表すのに対し、パーシャル・タイム・スニペットは「スライドする」と複数のインスタントに適用できるようなある共通の意味を持つ時点を指しています。例えば 6月 2日は、(グレゴリオ暦の) の毎年 6月の 2 番目の日であれば、どの日にでも当てはまります。同様に、11:06 p.m. は年や日に関係なく、1 日のうちに必ず 1 回は該当する時点があります。このように、パーシャル・タイム・スニペットは特定の一時点を表すわけではありませんが、役に立つことはあります。

私としては、パーシャル・タイム・スニペットはある周期で繰り返されるなかの一時点だと考えています。したがって、日付構成体が意味のある形で何度も起こる (つまり繰り返される) と考えられる場合、それはパーシャルということになります。

暦 (Chronology)

Joda の内部構造の鍵、そしてその設計の中核にあるのは暦です (暦の概念は、Chronology という名前の抽象クラスに取り込まれます)。基本的には、Chronology とは暦の体系 (時間を計算する特

定の方法) であり、日付計算を行うためのフレームワークのことです。Joda でサポートしている Chronology には、例えば以下のものがあります。

- ISO (デフォルト)
- コプト暦
- ユリウス暦
- イスラム暦

Joda-Time 1.6 でサポートされている Chronology は全部で 8 つです。そのそれぞれが、特定の暦の体系の計算エンジンとして機能します。

タイムゾーン

タイムゾーンとは、時刻の計算に使用される、英国グリニッジを基準とした相対的な地理的位置です。イベントが発生した正確な時刻を知るには、イベントが発生した地点を知ることも重要になります。相対的な時刻の計算を同じタイムゾーン内で行うのでない限り、正確な時刻の計算にはタイムゾーンを考慮に入れるか、あるいは GMT を基準とした相対的な時刻で行う必要があります。また、同じタイムゾーン内で時刻を計算する場合でも、別のタイムゾーンの関係者にとって興味深いイベントであれば、タイムゾーンが計算に関係してきます。

`DateTimeZone` は、Joda ライブラリーがこの位置の概念をカプセル化するために使用するクラスです。多くの日付および時刻の計算は、タイムゾーンとは関係なく行うことができますが、`DateTimeZone` がJoda で行っている処理にどのように影響するかについては、理解しておかなければなりません。大体の場合は、コードを実行しているマシンのシステム・クロックから取得された時刻がデフォルトとして使用されます。

Joda-Time オブジェクトの作成方法

ここからは、Joda ライブラリーを導入する読者のために、よく使用することになる Joda クラスをいくつか紹介し、これらのクラスのインスタンスを作成する方法を説明します。

可変の Joda クラス

私は可変ユーティリティー・クラスの絶大なるファンというわけではありません。それは単に、可変ユーティリティー・クラスを普及させるだけの使用ケースは巷にないと思うからですが、どうしても可変の Joda クラスを使用しなければならないと判断した場合、このセクションの情報が皆さんそれぞれのプロジェクトに役立つはずです。`Readable` API と `ReadWritable` API との唯一の違いは、`ReadWritable` クラスではカプセル化された日付/時刻の値を変化させることができるというだけなので、`ReadWritable` クラスについての説明は省きます。

このセクションで取り上げる実装にはすべて、カプセル化された日付/時刻を初期化するために使用できるコンストラクターがいくつかあります。これらのコンストラクターは、以下のカテゴリーに分類することができます。

- システム時刻を使用するコンストラクター。
- 個別のフィールドを使用して、インスタント (またはパーシャル・タイム・スニペット) をその特定の実装がサポートする最も粒度の細かい精度に指定するコンストラクター。
- インスタント (またはパーシャル・タイム・スニペット) をミリ秒単位で指定するコンストラクター。

- 別のオブジェクト (`java.util.Date`、または別の Joda オブジェクトなど) を使用するコンストラクター。

上記のコンストラクターについて、最初に取り上げる `DateTime` クラスの例で説明します。ここで説明する内容は、その他の Joda クラスの対応するコンストラクターを使用する場合にもぴったり当てはまります。

オーバーロードされたメソッド

Chronology または DateTimeZone を指定しないで DateTime のインスタンスを作成すると、Joda は ISOChronology (デフォルト) と、システムに設定されている DateTimeZone を使用します。けれども、Joda の ReadableInstant サブクラスのコンストラクターにはすべて、Chronology または DateTimeZone を引数に取るオーバーロードされたメソッドが含まれています。これらのオーバーロードされたメソッドの使用方法は、この記事に付属のアプリケーションのサンプル・コードに示されています (「ダウンロード」を参照)。オーバーロードされたメソッドの使い方は非常に単純なので、この記事で詳細を説明することはしませんが、皆さんには是非、サンプル・アプリケーションをいろいろと試してみることをお勧めします。そうすれば、アプリケーションの他のコードの部分に影響を与えることなく、即座に Joda の Chronology と DateTimeZone を交換できるようなコードを作成することが、いかに簡単であるかを実感できるはずです。

`ReadableInstant`

Joda ではインスタントの概念を `ReadableInstant` クラスによって実体化します。不変の一時点を表す一連の Joda クラスは、このクラスのサブクラスです (実際には、`ReadOnlyInstant` という名前にしたほうが設計者たちの意図が伝わっていたのではないかと思います。つまり、`ReadableInstant` は、変更不可能な一時点を表すということです)。これらのサブクラスのなかに、`DateTime` と `DateMidnight` の 2 つがあります。

- **`DateTime`**: 最も頻繁に使用することになるクラスです。このクラスは特定の時点をミリ秒の精度でカプセル化します。`DateTime` は常に `DateTimeZone` に関連付けられます。DateTimeZone を指定しなければ、コードが実行されているマシンのタイムゾーンにデフォルト設定されます。
  `DateTime` オブジェクトを構成するには、いくつかの方法があります。以下のコンストラクターは、システム時刻を使用します。

  ```
  DateTime dateTime = new DateTime();
  ```

  通常、私はシステム・クロックを使ってアプリケーションの時刻を初期化することは避け、アプリケーションのコードが使用するシステム時刻の設定を外部化するようにしています。サンプル・アプリケーションでは以下のようにしています。

  ```
  DateTime dateTime = SystemFactory.getClock().getDateTime();
  ```

  こうしておくと、さまざまな日付/時刻を使用するコードのテストが遥かに単純になります。この場合、異なる日付のシナリオを実行するためにコードを変更する必要はありません。時刻はアプリケーションとは別に、`SystemClock` の実装の内部で設定されるからです (システム上の時刻を変更することもできますが、あまりにも厄介です)。
  以下のコードは、個別のフィールド値を使用して `DateTime` オブジェクトを構成します。

```
DateTime dateTime = new DateTime(
  2000, //year
  1,    // month
  1,    // day
  0,    // hour (midnight is zero)
  0,    // minute
  0,    // second
  0     // milliseconds
);
```

このように、Joda では特定の一時点を表す `DateTime` オブジェクトをどのように作成するかを緻密に制御することができます。すべての Joda クラスには上記のようなコンストラクターが 1 つあり、Joda クラスが保持できるすべてのフィールドをこのコンストラクターで指定します。このコントラクターはいわば、特定のクラスがどのような粒度で動作するのかを知るためのクイック・ガイドとして利用することができます。
以下のコンストラクターは、エポックからのミリ秒数でインスタントを指定します。このコンストラクターは `DateTime` オブジェクトを、Joda と同じくエポックからのミリ秒数という定義を持つ JDK の `Date` オブジェクトのミリ秒の値から作成します。

```
java.util.Date jdkDate = obtainDateSomehow();
long timeInMillis = jdkDate.getTime();
DateTime dateTime = new DateTime(timeInMillis);
```

以下の例は前の例と同様ですが、ここでは `Date` オブジェクトをコンストラクターに直接渡しているという点が異なります。

```
java.util.Date jdkDate = obtainDateSomehow();
dateTime = new DateTime(jdkDate);
```

リスト 5 に示すように、Joda は `DateTime` を作成する際に他の多数のオブジェクトをコンストラクターのパラメーターとしてサポートします。

リスト 5. `DateTime` のコンストラクターに直接さまざまなオブジェクトを渡す

```
// Use a Calendar
java.util.Calendar calendar = obtainCalendarSomehow();
dateTime = new DateTime(calendar);
// Use another Joda DateTime
DateTime anotherDateTime = obtainDateTimeSomehow();
dateTime = new DateTime(anotherDateTime);
// Use a String (must be formatted properly)
String timeString = "2006-01-26T13:30:00-06:00";
dateTime = new DateTime(timeString);
timeString = "2006-01-26";
dateTime = new DateTime(timeString);
```

解析しなければならない `String` を使用する場合には、正確な書式設定が必要となることに注意してください。詳細は、Joda の `ISODateTimeFormat` クラスに関する Javadoc を参照してください (「参考文献」を参照)。

- **`DateMidnight`**: このクラスは、あるタイムゾーン (通常はデフォルトのタイムゾーン) の特定の年/月/日の午前 0 時というインスタントをカプセル化します。基本的には `DateTime` と同じですが、時刻の部分は常に、オブジェクトに関連付けられた特定の `DateTimeZone` での午前 0 時であるという点が異なります。

記事でこの後使用する残りのクラスは、`ReadableInstant` クラスと同じパターンに従います (Joda の Javadoc にざっと目を通すと見えてくるパターンです)。以降のセクションではスペースを節約するため、それぞれのクラスについては説明しません。

`ReadablePartial`

アプリケーションで処理するすべての日付が、完全なインスタントであるわけではありません。そこでインスタントの代わりの手段として使用できるのが、パーシャル・インスタントです。例えば、年/月/日や 1 日のうちのある時刻、あるいは 1 週間の中のある曜日にしか関心がないという場合があります。Joda の設計者たちはこのパーシャル・インスタントの概念を、不変のパーシャル・タイム・スニペットである `ReadablePartial` インターフェースに取り込みました。タイム・スニペットを処理する上で役立つクラスには、`LocalDate` と `LocalTime` の 2 つがあります。

- **`LocalDate`**: 年/月/日の組み合わせをカプセル化するこのクラスは、位置 (つまり、タイムゾーン) が重要でない日付を保管する上で重宝します。例えば、特定オブジェクトの誕生日が 1999年 4月 16日という値だとします。技術的観点からすると、その日付に関する他の詳細 (その日付が何曜日であるかとか、その個人が生まれたタイムゾーンなど) がわからなくても、ビジネス価値はまったく損なわれません。そのような場合には、`LocalDate` を使用してください。
  サンプル・アプリケーションでは以下のように、システム時刻に初期化された `LocalDate` のインスタンスを `SystemClock` を使用して取得します。
  ```
  LocalDate localDate = SystemFactory.getClock().getLocalDate();
  ```
  `LocalDate` を作成する際には、このクラスが保持できる各フィールドの値を明示的に指定することもできます。
  ```
  LocalDate localDate = new LocalDate(2009, 9, 6);// September 6, 2009
  ```
  `LocalDate` は、Joda の以前のバージョンで使われていた `YearMonthDay` の代わりとなるクラスです。
- **`LocalTime`**: 1 日のうちのある時刻をカプセル化するこのクラスは、位置は重要でなく、時刻だけを保管したい場合に役立ちます。一例として、午後 11時 52分は重要な時刻ではあるものの、特定の曜日に固有の時刻ではないとします (例えばファイルシステムの一部をバックアップする cron ジョブが開始される時刻など)。その場合には、時刻さえわかっていればよいことになります。
  サンプル・アプリケーションでは以下のように、システム時刻に初期化された `LocalTime` のインスタンスを `SystemClock` を使用して取得します。
  ```
  LocalTime localTime = SystemFactory.getClock().getLocalTime();
  ```
  `LocalTime` も同じく、このクラスが保持できる各フィールドの値を明示的に指定して作成することができます。
  ```
  LocalTime localTime = new LocalTime(13, 30, 26, 0);// 1:30:26PM
  ```

期間

特定のインスタントやパーシャル・タイム・スニペットについて知っておくことが有益である一方、期間を表現できると役立つ場合もよくあります。Joda には期間を簡単に表現できるように

する 3 つのクラスが用意されています。どのクラスを使用するかは、表現したい期間のタイプによって決まります。

- **`Duration`**: このクラスは期間をミリ秒単位で数学的に表現します。このクラスのメソッドは、ミリ秒単位での表現を標準的な数学変換 (1 分は 60 秒、1 日は 24 時間など) によって秒、分、時などの標準単位での表現に変換するために使用することができます。
  `Duration` のインスタンスを使用して期間を表現するのは、その特定の期間がいつから始まるかは重要でない場合、あるいは期間をミリ秒で処理すると都合がよい場合に限られます。
- **`Period`**: このクラスが表現する概念は `Duration` と同じですが、その表現は、例えば年、月、週などといった「人間」にとっての観点からです。
  `Period` を使用するのは、その期間がいつから始まるかが必ずしも重要でない場合や、あるいは `Period` によってカプセル化された期間を表す個々のフィールドを取得できるということが重要な場合です。
- **`Interval`**: このクラスは、明確なインスタントによって範囲が決められた特定の期間を表現します。`Interval` は半開の期間です。つまり、`Interval` によってカプセル化された期間には、期間の開始は含まれますが、終了は除外されることを意味します。
  `Interval` を使用するのは、時系列における特定の時点で始まり、特定の時点で終了する期間を表現しなければならない場合です。

Joda による時刻の操作方法

いくつかの便利な Joda クラスを作成する方法を学んだところで、今度はこれらのクラスを使って日付を計算する方法を説明します。その後、Joda を JDK と一緒に使用するのがいかに容易であるかを説明します。

日付の計算

日付/時刻情報のプレースホルダーだけが必要な場合には JDK でも十分に用は足りますが、JDK を使って日付/時刻の計算を行うとなると、かなり厄介なことになります。そこで力を発揮するのが、Joda です。これから、いくつかの単純な例を紹介します。

例えば、現在のシステム日付を前提として、前の月の最後の日を計算したいとします。この場合に必要なのは年/月/日だけなので、時刻は重要ではありません (リスト 6 を参照)。

リスト 6. Joda を使用して日付を計算する

```
LocalDate now = SystemFactory.getClock().getLocalDate();
LocalDate lastDayOfPreviousMonth =\
  now.minusMonths(1).dayOfMonth().withMaximumValue();
```

リスト 6 で `dayOfMonth()` が呼び出されていることについて疑問に思われるかもしれませんが、これは Joda がプロパティーと呼ぶもので、プロパティーは Joda オブジェクトの属性です。プロパティーは、プロパティーが表現するお馴染みの構成体にちなんで名付けられており、計算を行う目的でその構成体にアクセスするために使用されます。プロパティーは Joda の計算能力の鍵となります。これまで紹介した Joda の 4 つのクラスはすべて、このようなプロパティーを持っています。以下に、その例をいくつか挙げます。

- `yearOfCentury`

- `dayOfYear`
- `monthOfYear`
- `dayOfMonth`
- `dayOfWeek`

リスト 6 の例で何が行われているのかを順に説明すると、まず初めに現在の月から 1 ヶ月を引いて「前の月」を取得します。次に、`dayOfMonth` プロパティーにその最大値を要求することによって、その月の最後の日を取得します。注目すべき点は、これらの呼び出しは連鎖しているため (Joda の `ReadableInstant` サブクラスは不変であることを思い出してください)、この連鎖での最後のメソッド呼び出しの結果を取得するだけで、計算全体の結果を得られることです。

このパターンは、計算の中間結果を知る必要がない場合に、私は Joda でかなり頻繁に使用しているものです (JDK の `BigDecimal` は、これと同じ方法で使用しています)。一方、(特定の年には関係なく) 11月の最初の月曜日の翌日である最初の火曜日の日付を知りたいとしたら、リスト 7 の方法を使用します。

リスト 7. 11月最初の月曜日の翌日となる最初の火曜日を計算する

```
LocalDate now = SystemFactory.getClock().getLocalDate();
LocalDate electionDate = now.monthOfYear()
 .setCopy(11)        // November
 .dayOfMonth()       // Access Day Of Month Property
 .withMinimumValue() // Get its minimum value
 .plusDays(6)        // Add 6 days
 .dayOfWeek()        // Access Day Of Week Property
 .setCopy("Monday")  // Set to Monday (it will round down)
 .plusDays(1);       // Gives us Tuesday
```

リスト 7 のコードで結果に辿り着く方法を理解するには、リスト内のコメントを参考にしてください。このコードの成功の鍵となっているのは、`.setCopy("Monday")` の行です。中間の `LocalDate` の値とは関係なく、`dayOfWeek` プロパティーを月曜日に設定することで、月の初めに 6 日を足すと必ず最初の月曜日が返されるように「切り捨て」が行われます。それに 1 日を足せば、最初の火曜日を取得できるというわけです。このように、Joda ではこのタイプの計算を簡単に行えるようにしています。

以下に、Joda を使用すると極めて簡単に行える他の計算例をいくつか紹介します。

以下のコードは、現在から 2 週間後を計算します。

```
DateTime now = SystemFactory.getClock().getDateTime();
DateTime then = now.plusWeeks(2);
```

明日から 90 日後を計算するには、以下の方法を使います。

```
DateTime now = SystemFactory.getClock().getDateTime();
DateTime tomorrow = now.plusDays(1);
DateTime then = tomorrow.plusDays(90);
```

(もちろん `now` に 91 日を足すだけでも計算できますが、それでは面白味がありません。)

現在から 156 秒後を計算する方法は以下のとおりです。

```
DateTime now = SystemFactory.getClock().getDateTime();
DateTime then = now.plusSeconds(156);
```

そして以下のコードは、5 年前の 2月最後の日を計算します。

```
DateTime now = SystemFactory.getClock().getDateTime();
DateTime then = now.minusYears(5) // five years ago
               .monthOfYear()     // get monthOfYear property
               .setCopy(2)        // set it to February
               .dayOfMonth()      // get dayOfMonth property
               .withMaximumValue();// the last day of the month
```

この他にもまだまだ例を挙げられますが、要点はつかんでもらえたことでしょう。サンプル・サプリケーションをいろいろといじって、Joda を使用すればどんな日付の計算でも楽にこなせることをご自分で確かめてください。

JDK とのインターオペラビリティー

私の既存のコードには、JDK の `Date` クラスと `Calendar` クラスを使用しているものがかなりあります。それでも Joda のおかげで、Joda で必要な日付の計算を行ってから JDK クラスに変換することができます。まさに、Joda と JDK 両方の長所を生かせるというわけです。「Joda-Time オブジェクトの作成方法」で説明したとおり、この記事で紹介した Joda クラスはすべて、JDK の `Calendar` または `Date` から作成することができます。同様に、今まで紹介した Joda クラスのどれからでも JDK の `Calendar` または `Date` を作成することが可能です。

リスト 8 に、Joda の `ReadableInstant` サブクラスから JDK クラスに移行するのはいかに単純であるかを示します。

リスト 8. Joda の `DateTime` クラスから JDK クラスを作成する

```
DateTime dateTime = SystemFactory.getClock().getDateTime();
Calendar calendar = dateTime.toCalendar(Locale.getDefault());
Date date = dateTime.toDate();
DateMidnight dateMidnight = SystemFactory.getClock()
  .getDateMidnight();
date = dateMidnight.toDate();
```

`ReadablePartial` サブクラスの場合には、もうひと手間必要になります (リスト 9 を参照)。

リスト 9. `LocalDate` を表す `Date` オブジェクトを作成する

```
LocalDate localDate = SystemFactory.getClock().getLocalDate();
Date date = localDate.toDateMidnight().toDate();
```

リスト 9 に示すように、`SystemClock` から取得した `LocalDate` を表す `Date` オブジェクトを作成するには、まず `DateMidnight` オブジェクトに変換する必要があります。後は、`DateMidnight` オブジェクトを `Date` として要求すればよいだけです (したがって、当然のことながら `Date` オブジェクトの時刻の部分は午前 0時に設定されます)。

JDK とのインターオペラビリティーは Joda API 自体に組み込まれているため、インターフェースが JDK に結合されているとしても、インターフェースをまるごと置き換える必要はありません。例えば、Joda には困難な部分を任せ、インターフェースには JDK を使用するといったことも可能です。

Joda による時刻の書式設定方法

出力用の日付の書式設定に JDK を使用するのは、着実な方法ではあるものの、私は常々この方法をもっと単純にできないものかと思っていました。これが、Joda の設計者たちが核心を突いたもう 1 つの機能です。Joda オブジェクトの書式を設定するには、そのオブジェクトの `toString()` メソッドを呼び出し、必要に応じて標準の ISO-8601 または JDK 対応の制御文字列のいずれかを渡して Joda に書式設定の方法を指示します。別個の `SimpleDateFormat` オブジェクトを作成する必要はもうありません (ただし Joda は、敢えて苦労したいという人々のために `DateTimeFormatter` クラスを用意しています)。Joda オブジェクトの `toString()` メソッドを 1 回呼び出すだけでよいのです。以下に例をいくつか紹介します。

リスト 10 は、`ISODateTimeFormat` の静的メソッドを使用した例です。

リスト 10. ISO-8601 を使用する

```
DateTime dateTime = SystemFactory.getClock().getDateTime();
dateTime.toString(ISODateTimeFormat.basicDateTime());
dateTime.toString(ISODateTimeFormat.basicDateTimeNoMillis());
dateTime.toString(ISODateTimeFormat.basicOrdinalDateTime());
dateTime.toString(ISODateTimeFormat.basicWeekDateTime());
```

リスト 10 の 4 つの `toString()` 呼び出しは、それぞれ以下のような書式の出力を生成します。

```
20090906T080000.000-0500
20090906T080000-0500
2009249T080000.000-0500
2009W367T080000.000-0500
```

`SimpleDateFormat` という JDK に対応した書式文字列を渡すこともできます (リスト 11 を参照)。

リスト 11. `SimpleDateFormat` 文字列を渡す

```
DateTime dateTime = SystemFactory.getClock().getDateTime();
dateTime.toString("MM/dd/yyyy hh:mm:ss.SSSa");
dateTime.toString("dd-MM-yyyy HH:mm:ss");
dateTime.toString("EEEE dd MMMM, yyyy HH:mm:ssa");
dateTime.toString("MM/dd/yyyy HH:mm ZZZZ");
dateTime.toString("MM/dd/yyyy HH:mm Z");

09/06/2009 02:30:00.000PM
06-Sep-2009 14:30:00
Sunday 06 September, 2009 14:30:00PM
09/06/2009 14:30 America/Chicago
09/06/2009 14:30 -0500
```

Joda オブジェクトの `toString()` メソッドに渡すことのできる JDK の `SimpleDateFormat` 対応書式文字列についての詳細は、`joda.time.format.DateTimeFormat` に関する Javadoc を参照してください。

まとめ

日付の処理となると、Joda は驚くほどの効果を発揮するツールです。日付の計算、出力、あるいは解析のいずれにしても、Joda をツールボックスに加えておくと重宝します。この記事では、JDK

の日時関連クラスの代替ライブラリーとしての Joda の有用性を説明しました。続いて Joda が持つ概念をいくつか説明し、その後で実際に Joda を使用して日付の計算および書式設定を行う方法を説明しました。

Joda-Time から派生したいくつかの関連プロジェクトも、皆さんにとって役に立つ可能性があります。Joda-Time プラグインは現在、Grails Web 開発フレームワークで使用できるようになっています。joda-time-jpox プロジェクトは、Joda-Time オブジェクトを保持するために必要なマッピングを、 DataNucleus 永続化エンジンによって追加することを目的したプロジェクトです。Google Web Toolkit 対応の Joda-Time 実装 (Goda-Time) にも進展が見られますが、この記事を執筆している時点では、ライセンスの問題のために保留にされています。「参考文献」で、詳しい情報を調べてください。

ダウンロード

内容	ファイル名	サイズ
Source code	j-jodatime.zip	812KB

著者について

J Steven Perry

J Steven Perry は、ソフトウェア開発者兼アーキテクト、そして 1991年から専門的にソフトウェアを開発してきた一般の Java マニアです。彼の専門は、JVM の内部動作から UML モデリングに至るまで多岐にわたります。また技術文書の執筆から Java コードの作成までのあらゆる著作活動に携わり、教育と指導に情熱を注いでいます。彼は『Java Management Extensions』(O'Reilly) の著者、『Java Enterprise Best Practices』(O'Reilly) の共著者です。また、ソフトウェア開発の話題や O'Reilly ShortCut: Log4J に関する雑誌の記事も書いています。