Joda-Time

You can't escape time. Why not make it easy?

J Steven Perry
Principal Consultant
Makoto Consulting Group, Inc.

27 October 2009

No enterprise application can escape time. Applications need to know what time it is and what time it's going to be, and sometimes they must calculate the path between the two. Using the JDK to do this job can be painful and tedious. Enter Joda-Time, an easy-to-use open source date/time library for the Java™ platform. As you'll see in this article, Joda-Time eases the pain and tedium of manipulating dates and time.

In writing business applications, I frequently need to manipulate dates. And in my most recent engagement — in the insurance industry — correct date calculations are especially important. I was growing a little restless with `java.util.Calendar`. If you've used this class to handle date/time values, you know how cumbersome it can be to grasp. So when I heard about Joda-Time — a replacement date/time library for Java applications — I decided to investigate. Long story short: I'm glad I did.

Joda-Time makes time and date values easy to manage, manipulate, and understand. Ease of use, in fact, is Joda's primary design goal. Other goals include extensibility, a comprehensive feature set, and support for multiple calendar systems. And Joda is 100 percent interoperable with the JDK, so you don't need to replace all of your Java code, only the parts that do date/time calculations.

Under the Joda umbrella

Joda is actually an umbrella project for a number of replacement APIs for the Java language, so technically it's a misnomer to use the names Joda and Joda-Time synonymously. But at the time of this writing, the Joda-Time API appears to be the only Joda API under active development. Given the current state of the Joda umbrella project, I think it's safe to refer to Joda-Time as simply Joda.

This article introduces Joda and shows you how to use it. I'll cover the following topics:

- The case for a date/time replacement library
- Joda's key concepts

- Creating Joda-Time objects
- Manipulating time, Joda style
- Formatting time, Joda style

You can download source code for a sample application that demonstrates these concepts.

The case for Joda

Why bother with Joda? Consider creating an arbitrary moment in time — say, the stroke of midnight on January 1, 2000. How might I create a JDK object that represents this instant in time? `java.util.Date`? Well, not really, because the Javadoc for every Java version since JDK 1.1 states that `java.util.Calendar` should be used. The number of deprecated constructors in `Date` severely restrict the number of ways you can create such an object.

`Date` does have one constructor, however, that you can use to create an object representing an instant in time other than "now." That method takes as an argument the number of milliseconds since January 1, 1970 at midnight Greenwich Mean Time (GMT) (also known as the *epoch*), correcting for time zone. Given Y2K's importance in the software-development business, you might think I'd have this value committed to memory — but I don't. So much for `Date`.

So what about `Calendar`? I'd create the necessary instance this way:

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

Using Joda, the code looks like this:

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

There's not much difference in this one line of code. But now I'll complicate the problem a bit. Suppose I want to add 90 days to that date and print the result. Using the JDK, I'd do this as shown in Listing 1:

Listing 1. Adding 90 days to an instant and printing the result, the JDK way

```
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()));
```

Using Joda, the code looks like Listing 2:

Listing 2. Adding 90 days to an instant and printing the result, using Joda

```
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");
```

The gap has widened a bit (two lines of code for Joda versus five for the JDK).

Now suppose I want to print out the date of the last day of the week of the month after the one that is 45 days from Y2K. Frankly, I don't even want to try tackling this using `Calendar`. It's just too painful to use the JDK to do even a simple date calculation like this one. It was at such a moment a few years ago when I first realized the power of Joda-Time. Using Joda, the code for the calculation looks like Listing 3:

Listing 3. Joda to the rescue

```
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");
```

Listing 3 prints:

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

If you are looking for an easy-to-use replacement for JDK date processing, then you really should consider Joda. If not, by all means continue to use `Calendar` to do all of your date calculations. While you're at it you could cut your grass with a pair of scissors and wash your car with an old toothbrush.

Joda and JDK interoperability

It doesn't take long to see that the JDK `Calendar` class is lacking in usability, and that Joda shores up this deficiency. The designers of Joda also made a decision that I believe is the key to its success: JDK interoperability. Joda's classes are capable of producing (though at times in a somewhat circuitous manner, as you'll see) instances of `java.util.Date` (and `Calendar`). This lets you keep your existing JDK-dependent code but use Joda to do the heavy lifting when it comes to date/time calculations.

For example, after doing the calculation in Listing 3, all I need to do to get back to the JDK is make the changes shown in Listing 4:

Listing 4. Entering Joda calculation results into a JDK object

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

And just like that, I've done the calculation but can proceed with the results in a JDK object. This is a very cool feature of Joda.

Joda's key date/time concepts

Joda uses these concepts, which could apply to any date/time library:

- Immutability
- Instant

- Partial
- Chronology
- Time zone

I'll discuss each of them in turn with respect to Joda.

Immutability

The Joda classes I discuss in this article are immutable, so instances of them cannot be modified. (One advantage of immutable classes is that they are thread-safe.) The API methods I'll show you for doing date calculations all return a new instance of the corresponding Joda class, leaving the original instance unchanged. When you manipulate a Joda class through an API method, you must capture the method's return value because the instance you're manipulating cannot be modified. You're probably familiar with this pattern; it's how the various manipulation methods of `java.lang.String` work, for example.

Instant

The `Instant` represents an exact moment in time, expressed in milliseconds from the epoch. This definition is consistent with the JDK, which is why any Joda `Instant` subclass is compatible with the JDK `Date` and `Calendar` classes.

To define it generically: an *instant* is a point on the time line that is unique and occurs only once, and such a date construct can occur only once in a meaningful way.

Partial

A partial, or partial time snippet as I'll refer to it in this article, is just that: a partial snippet of time. Whereas an instant specifies an exact moment in time relative to the epoch, a partial time snippet refers to a moment in time that could "slide" around such that it could apply to multiple instants. For example, *June 2* could apply to any instant of any occurrence of the second day of June (using the Gregorian calendar) in any year. Likewise, *11:06 p.m.* could apply once per day on any day of any year. Even though they do not specify an instant of time, partial time snippets can be useful.

I like to think of partial time snippets as a point on a circle that repeats, so that if the date construct I'm thinking of can occur multiple times (that is, repeat) in a meaningful way, then it is a partial.

Chronology

The key to Joda's internals — and at the heart of its design — is the *chronology* (the notion of which is captured by an abstract class of the same name). Fundamentally, chronology is a calendar system — a particular way of reckoning time — and is the framework in which date arithmetic is done. Examples of chronologies supported by Joda are:

- ISO (the default)
- Coptic
- Julian
- Islamic

Joda-Time 1.6 supports eight chronologies, each of which serves as the calculation engine for that particular calendar system.

Time zone

A time zone is a geographic location relative to Greenwich, England that is used to calculate the time. To know precisely what time an event occurs, it is also important to know the event's location. Any rigorous time calculation must account for time zone (or be relative to GMT), unless relative time calculations occur within the same time zone (and even then it could be important if the event is of interest to parties in another time zone).

`DateTimeZone` is the class that the Joda library uses to encapsulate this notion of location. Many date and time calculations can be done without regard to time zone, but it's still important to understand how `DateTimeZone` affects what Joda is doing. The default time, which is retrieved from the system clock of the machine the code is running on, is used most of the time.

Creating Joda-Time objects

Now I'll introduce you to some of the Joda classes that you will commonly use if you adopt the library and show how to create instances of them.

Mutable Joda classes

I'm not a big fan of mutable utility classes; I just don't think the use cases are there to support their widespread use. But if you decide you really need to use the mutable Joda classes, the information in this section should still prove helpful in your projects. The only difference between the `Readable` and `ReadWritable` APIs is the ability of the `ReadWritable` classes to mutate the encapsulated date/time value, so I won't cover them here.

All of the implementations you'll look at in this section have several constructors that allow you to initialize the encapsulated date/time. They fall into four categories:

- Use system time.
- Specify an instant (or partial time snippet) using individual fields to the most granular precision supported by that particular implementation.
- Specify an instant (or partial time snippet) in milliseconds.
- Use another object (for example, `java.util.Date`, or perhaps another Joda object).

I will cover these constructors for the first class you'll look at: `DateTime`. The information there will transfer nicely when you use the corresponding constructors of the other Joda classes.

Overloaded methods

If you create an instance of `DateTime` without supplying a `Chronology` or `DateTimeZone`, Joda uses `ISOChronology` (the default) and `DateTimeZone` (from your system setup). However, all of the constructors for Joda `ReadableInstant` subclasses contain an overloaded method that takes either a `Chronology` or a `DateTimeZone`. The sample code for the application provided with this article shows you how to use these overloaded methods (see Download). I won't cover them here in detail because they are so simple to use. However, I do encourage you to play around with the sample application to get a feel for how easy it is to write your application code so that you can swap out Joda's `Chronology` and `DateTimeZone` on-the-fly without affecting the rest of the code.

`ReadableInstant`

Joda realizes the concept of an instant through the `ReadableInstant` class. Joda classes that represent immutable instants in time are subclasses of this one. (It might have been better named `ReadOnlyInstant`, which I believe is what the designers intended to convey. In other words, `ReadableInstant` represents an instant in time that cannot be modified.) Two of those subclasses are `DateTime` and `DateMidnight`:

- **`DateTime`**: This is the class you should use most often. It encapsulates an instant in time with millisecond precision. A `DateTime` is always associated with a `DateTimeZone`, which — if you don't specify it — defaults to the time zone of the machine the code is running on.
  There are a number of ways to construct a `DateTime` object. This constructor uses the system time:
  ```
  DateTime dateTime = new DateTime();
  ```

  In general, I try to avoid using the system clock to initialize the time for my application, preferring to externalize setting the system time that my application code uses. The sample application does this:
  ```
  DateTime dateTime = SystemFactory.getClock().getDateTime();
  ```

  This makes testing my code using different dates/times much simpler: I don't need to change the code to run different date scenarios through it, because the time is set inside the implementation of `SystemClock`, and away from my application. (I could change the time on my system, but what a pain!)
  This code uses individual field values to construct a `DateTime` object:
  ```
  DateTime dateTime = new DateTime(
    2000, //year
    1,    // month
    1,    // day
    0,    // hour (midnight is zero)
    0,    // minute
    0,    // second
    0     // milliseconds
  );
  ```

  As you can see, Joda gives you precise control over how to create a `DateTime` object that represents a particular instant in time. Every Joda class has a constructor, similar to this one, where you specify all of the fields that the Joda class can hold. You can use it as a quick guide to the level of granularity at which that particular class operates.
  This next constructor specifies the instant in milliseconds since the epoch. It creates a `DateTime` object from a JDK `Date` object's millisecond value, whose definition of time in milliseconds since the epoch is the same as Joda's:
  ```
  java.util.Date jdkDate = obtainDateSomehow();
  long timeInMillis = jdkDate.getTime();
  DateTime dateTime = new DateTime(timeInMillis);
  ```

  And this example is similar to the preceding one except that this time I pass the `Date` object directly to the constructor:
  ```
  java.util.Date jdkDate = obtainDateSomehow();
  dateTime = new DateTime(jdkDate);
  ```

Joda supports a number of other objects as the constructor parameter for creating a `DateTime`, as shown in Listing 5:

Listing 5. Passing different objects directly to `DateTime`'s constructor

```
// 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);
```

Note that if you plan to use a `String` (which must be parsed), you must format it precisely. See the Javadoc for Joda's `ISODateTimeFormat` class for more information (see Resources).

- **`DateMidnight`**: This class encapsulates the instant of a particular year/month/day at midnight in some time zone (usually the default time zone). It is basically just like `DateTime`, except that the time portion is always at midnight in the particular `DateTimeZone` associated with the object.

The rest of the classes you'll see in this article follow the same pattern as the `ReadableInstant` classes (as the most cursory glance at the Joda Javadoc will show), so to save space I won't cover them in the sections that follow.

`ReadablePartial`

Not all dates you work with in your applications are full instants in time, so instead you can deal with a partial instant. For example, sometimes all you care about is the year/month/day, or the time of day, or even just the day of the week. The Joda designers captured this partial-instant concept with the `ReadablePartial` interface, which is an immutable partial time snippet. Two useful classes for dealing with time snippets are `LocalDate` and `LocalTime`:

- **`LocalDate`**: This class encapsulates a year/month/day combination. It is handy for storing dates when location (that is, time zone) is unimportant. For example, *date of birth* for a particular object may have the value *April 16, 1999*, but from a technical standpoint all business value is retained without knowing anything further about that date (like what day of the week it was or the time zone the person was born in). In such cases, `LocalDate` should be used.
  The sample application uses the `SystemClock` to obtain an instance of `LocalDate` initialized to the system time:
  ```
  LocalDate localDate = SystemFactory.getClock().getLocalDate();
  ```
  You can also create a `LocalDate` by explicitly providing the value of each field it can hold:
  ```
  LocalDate localDate = new LocalDate(2009, 9, 6);// September 6, 2009
  ```
  `LocalDate` is a replacement for `YearMonthDay`, used in previous Joda versions.

- **`LocalTime`**: This class encapsulates a time of day and is useful for storing only the time of day when location is unimportant. For example, 11:52 p.m. may be an important time of day (say, when a cron job kicks off that backs up a part of my file system), but it's not specific to any particular day, so I don't need to know anything else about it.
  The sample application uses the `SystemClock` to obtain an instance of `LocalTime` initialized to the system time:
  ```
  LocalTime localTime = SystemFactory.getClock().getLocalTime();
  ```
  You can also create a `LocalTime` by explicitly providing the value of each field it can hold:
  ```
  LocalTime localTime = new LocalTime(13, 30, 26, 0);// 1:30:26PM
  ```

Spans of time

Knowing about a particular instant or partial time snippet can be useful, but it's also often useful to be able to express spans of time. Joda provides three classes to make this easy to do. You choose the class that expresses the type of span you want to express:

- **`Duration`**: This class represents an absolute mathematical span, expressed in milliseconds. Methods on this class can be used to convert to standard units such as seconds, minutes, and hours, using standard mathematical conversions (such as 60 seconds in a minute and 24 hours in a day).
  You use an instance of `Duration` only when you want to express a span of time wherein you don't care when that particular span of time occurs or it is convenient for you to handle that span in milliseconds.
- **`Period`**: This class represents the same concept as `Duration`, but in "human" terms such as years, months, and weeks.
  You use a `Period` when you don't care when the period of time necessarily occurs, or perhaps when it is more important to be able to retrieve the individual fields that describe the span of time encapsulated by the `Period`.
- **`Interval`**: This class represents a particular span of time with a definite instant marking its boundaries. The `Interval` is *half-open*, meaning that the span of time encapsulated by the `Interval` includes the beginning of the span but excludes the end.
  You use an `Interval` when it is important to express a span of time that begins and ends with definite points in the time continuum.

Manipulating time, Joda style

Now that you've seen how to create a few of the more useful Joda classes, I'll show you how to use them to perform date calculations. Then you'll see how Joda easily interoperates with the JDK.

Date calculations

If you only needed placeholders for date/time information, the JDK would be fine, but using it to do date/time calculations is painful. This is where Joda shines. I'll show you a few simple examples.

Suppose that I want to calculate the last day of the preceding month given the current system date. In this case, I don't care about the time of day because all I need is the year/month/day, as shown in Listing 6:

Listing 6. Calculating a date with Joda

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

You may be wondering about the `dayOfMonth()` call in Listing 6. It's something Joda calls a *property*. A property is an attribute of a Joda object. It is named according to the familiar construct it represents and is used to access that construct for the purpose of doing calculations. Properties are the key to Joda's calculation power. All four of the Joda classes you've looked at so far have such properties. Some examples are:

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

I'll walk through the example in Listing 6 to show you exactly what is going on. First, I subtract one month from the current one to get "last month." Next I ask the `dayOfMonth` property for its maximum value, which gets me to the end of the month. Notice that the calls are chained together (remember that Joda `ReadableInstant` subclasses are immutable) so that you need only capture the result of the last method call in the chain to get the result of the entire calculation.

This pattern is one I use a great deal with Joda when I don't need to know the intermediate results of a calculation. (I use JDK's `BigDecimal` the same way.) Suppose you want the date for the first Tuesday following the first Monday in November (of any particular year). Listing 7 shows how:

Listing 7. Calculating the first Tuesday following the first Monday in November

```
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
```

The comments in Listing 7 help you to see how the code arrives at the result. The `.setCopy("Monday")` line is the key to making it work. No matter what the intermediate `LocalDate` value is, setting its `dayOfWeek` property to Monday always rounds "down" such that adding six days to the beginning of the month gives you the first Monday. Add one day and you have the first Tuesday. Joda makes it easy to do these types of calculations.

Here are some other calculations that are super easy to do using Joda:

This code calculates two weeks from now:

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

You can calculate 90 days from tomorrow this way:

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

(Yes, I could just add 91 days to `now`, but what fun would that be?)

Here's how you can calculate 156 seconds from now:

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

And this code calculates the last day of February five years ago:

```
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
```

I could go on and on with these examples, but I think you get the idea. Play with the sample application to see for yourself how much fun calculating any date can be with Joda.

JDK interoperability

I have lots of code that uses JDK `Date` and `Calendar` classes. But thanks to Joda, I can do any necessary date arithmetic and then convert back to JDK classes. It's the best of both worlds! All of the Joda classes you've seen in this article can be created from a JDK `Calendar` or `Date`, as you saw in Creating Joda-Time objects. By the same token, it is possible to create a JDK `Calendar` or `Date` from any of the Joda classes you've seen.

Listing 8 shows how simple it is to go from Joda `ReadableInstant` subclasses to JDK classes:

Listing 8. Creating JDK classes from the Joda `DateTime` class

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

For the `ReadablePartial` subclasses, you need to jump through one extra hoop, as shown in Listing 9:

Listing 9. Creating a `Date` object representing `LocalDate`

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

To create a `Date` object that represents the `LocalDate` obtained from the `SystemClock` as shown in Listing 9, you must first convert it to a `DateMidnight` object, and then simply ask for the `DateMidnight` object as a `Date`. (The resulting `Date` object will have its time portion set to midnight, of course.)

JDK interoperability is built right into the Joda API, so you needn't replace your interfaces completely if they're tied to the JDK. You can, for example, use Joda to do the heavy lifting and the JDK for your interfaces.

Formatting time, Joda style

Using the JDK to format dates for printing is robust, but I've always thought it should be simpler. This is another feature the Joda designers got right. To format a Joda object, call its `toString()` method and, if you wish, pass either a standard ISO-8601 or a JDK-compatible control string to tell Joda how to format it. No more creating separate `SimpleDateFormat` objects (although Joda does provide a `DateTimeFormatter` class for the masochistic). One call to a Joda object's `toString()` method and you're done. I'll show you a few examples.

Listing 10 uses static methods of `ISODateTimeFormat`:

Listing 10. Using ISO-8601

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

The four `toString()` calls in Listing 10 create something like this, respectively:

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

You can also pass `SimpleDateFormat` JDK-compatible format strings, as shown in Listing 11:

Listing 11. Passing `SimpleDateFormat` strings

```
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
```

See the Javadoc for `joda.time.format.DateTimeFormat` for more information on the JDK `SimpleDateFormat`-compatible format strings you can pass to a Joda object's `toString()` method.

Conclusion

When it comes to processing dates, Joda is an amazingly effective tool. Whether you need to calculate dates, print them, or parse them, Joda is a handy addition to your toolbox. In this article, I made a case for Joda as a JDK date/time replacement library. Next you looked at some Joda concepts, and then how to use Joda to do date calculations and formatting.

Joda-Time has spawned some related projects that you might find helpful. A Joda-Time plug-in is now available for the Grails Web development framework. The joda-time-jpox project aims to add mappings necessary to persist Joda-Time objects using the DataNucleus persistence engine. And a Joda-Time implementation for Google Web Toolkit (called Goda-Time) has also seen some progress but at the time of this writing is on hold because of licensing issues. Explore the Resources for more information.

Downloads

Description	Name	Size
Source code	j-jodatime.zip	812KB

Resources

Learn

- Joda-Time: Visit the Joda project on SourceForge and check out the Javadoc for the Joda-Time library.
- Joda-Time Plugin for Grails: Read about the Joda-Time plug-in for Grails.
- joda-time-jpox: Learn more about the joda-time-jpox project.
- Goda-Time: Keep tabs on the Joda-Time porting project for Google Web Toolkit.
- Technology bookstore: Browse for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Joda-Time: Download the Joda-Time library.

Discuss

- Joda-Time mailing list: Subscribe to the Joda-Time mailing list or view the list archive.
- Get involved in the My developerWorks community.

About the author

J Steven Perry

J Steven Perry is a software developer, architect, and general Java nut who has been developing software professionally since 1991. His professional interests range from the inner workings of the JVM to UML modeling, and everything in between. Steve writes everything from technical documentation to Java code, and has a passion for teaching and mentoring. Steve is the author of *Java Management Extensions* (O'Reilly), co author of *Java Enterprise Best Practices* (O'Reilly), several magazine articles related to software development topics and the O'Reilly ShortCut: Log4J.