Joda Time Java Library

Mark Volkmann
Object Computing, Inc.
mark@ociweb.com

Joda Time Overview

- Free, open-source Java library for working with dates and times
  - Apache V2 license; not viral
- Replacement for the JDK Date and Calendar classes
  - being standardized under JSR-310
  - likely to become a standard part of Java
- Small - version 1.6 JAR file is 524 KB
- No dependencies outside core Java classes
- http://joda-time.sourceforge.net/
  - API documentation is at a link on this page
    - http://joda-time.sourceforge.net/api-release/

Supported Concepts

- Instant
  - "an instant in the datetime continuum specified as a number of milliseconds from 1970-01-01T00:00Z"
- Partial
  - a partial date/time representation (subset of fields) with no time zone
- Interval
  - "an interval of time from one millisecond instant to another"
- Duration
  - "a duration of time measured in milliseconds"
- Period
  - "a period of time defined in terms of fields, for example, 3 years 5 months 2 days and 7 hours"
- Chronology
  - "a pluggable calendar system"

partial + missing fields + time zone = instant

Mutability

- Most classes that represent the concepts are immutable
- Using these is recommended for most cases to avoid issues with concurrent access
- The only mutable classes are
  - `MutableDateTime`
  - `MutableInterval`
  - `MutablePeriod`

org.joda.time Main Classes

Concept	Immutable Classes	Mutable Classes

Instant	`DateTime`
	`DateMidnight`
	`Instant`
		`MutableDateTime`

Partial	`LocalDate`
	`LocalDateTime`
	`LocalTime`
	`Partial`	

Interval	`Interval`	`MutableInterval`
Duration	`Duration`	

Period	`Period`
	`Minutes Hours Days`
	`Weeks Months Years`
		`MutablePeriod`

Chronology	`GregorianChronology` `ISOChronology` (default) many more	

Instants …

- "A number of milliseconds from 1970-01-01T00:00Z"

```
Instant instant = new Instant(1000); // 1 second after 1970
instant = instant.plus(500);
instant = instant.minus(30);
System.out.println("milliseconds = " + instant.getMillis());

// Creating an instant representing current date and time is easy.
DateTime dt = new DateTime();
System.out.println("now date and time = " + dt);
```

Output:
```
milliseconds = 1470
now date and time = 2008-12-22T10:07:26.468-06:00
```

… Instants

```
// An instant representing a specific date and time
// can be created by specifying all the field values.
dt = new DateTime(1961, 4, 16, 10, 19, 0, 0);
System.out.println("my birthday = " + dt);
System.out.println("year is " + dt.getYear());
System.out.println("month is " + dt.getMonthOfYear())
```

Output:
```
my birthday = 1961-04-16T10:19:00.000-06:00
year is 1961
month is 4
```

Partials …

- A partial date/time representation (subset of fields) with no time zone

```
// There are a large number of constructors for creating
// period objects.  We'll demonstrate a few of them.
LocalDate birthday = new LocalDate(1961, 4, 16);
long millis = birthday.toDateTimeAtCurrentTime().getMillis();
System.out.println("millis = " + millis);

birthday = new LocalDate(1970, 1, 1); // not 0ms in local timezone
millis = birthday.toDateTimeAtCurrentTime().getMillis();
System.out.println("millis = " + millis);
```

Output:
```
millis = -274866753528 (negative because it's before 1970)
millis = 58046472 (close to zero)
```

… Partials

```
LocalDate today = new LocalDate();
int year = today.getYear();

// Find the date of the next Christmas.
LocalDate christmas = new LocalDate(year, 12, 25);
if (today.isAfter(christmas)) christmas = christmas.plusYears(1);
System.out.println("The next Christmas is on " + christmas);
```

Output:
```
The next Christmas is on 2008-12-25
```

Intervals …

- "An interval of time from one millisecond instant to another"

```
// Create an Interval from now to one month from now.
DateTime startTime = new DateTime(); // now
DateTime endTime = startTime.plus(Months.months(1));
Interval interval = new Interval(startTime, endTime);
System.out.println("interval = " + interval);
System.out.println("start = " + interval.getStart());
System.out.println("end = " + interval.getEnd());
System.out.println("duration = " + interval.toDuration());
```

Output:
```
interval = 2008-12-22T10:07:26.466/2009-01-22T10:07:26.466
start = 2008-12-22T10:07:26.466-06:00
end = 2009-01-22T10:07:26.466-06:00
duration = PT2678400S (31*24*60*60)
```

… Intervals

```
// There are many Interval methods whose name begins with "with"
// that create a new interval relative to an existing one.
// For example, this creates an interval
// that lasts for one more hour.
interval = interval.withEnd(interval.getEnd().plusHours(1));
System.out.println("interval = " + interval);
```

Output:
```
interval = 2008-12-22T10:07:26.466/2009-01-22T11:07:26.466
```

Durations …

- "A duration of time measured in milliseconds"

```
// Can construct with a given number of milliseconds.
Duration duration = new Duration(1000); // 1 second
System.out.println("duration = " + duration);

// Can construct with two Instants
// to get the duration between them.
// Get the duration from midnight this morning to now.
DateTime now = new DateTime();
System.out.println("now = " + now);
// The method toDateMidnight returns a new DateTime
// where all the time fields are set to zero.
duration = new Duration(now.toDateMidnight(), now);
System.out.println("duration = " + duration);
```

Output:
```
duration = PT1S
```

Output:
```
now = 2008-12-22T10:07:26.333-06:00
duration = PT36446.333S
```

… Durations …

```
// A Duration can be obtained from an Interval.
Interval interval = new Interval(now.toDateMidnight(), now);
duration = interval.toDuration();
System.out.println("duration = " + duration);

// A Duration can be added to an Instant to get a new Instant.
duration = new Duration(1000); // 1 second
DateTime nowPlusOneSecond = now.plus(duration);
System.out.println("nowPlusOneSecond = " + nowPlusOneSecond);
```

Output:
```
duration = PT36446.333S
```

Output:
```
nowPlusOneSecond = 2008-12-22T10:07:27.333-06:00
```

… Durations

```
// There is no mutable Duration type,
// so a new Duration object must be created
// when time needs to be added or subtracted.
// Compare this to code in the partials examples.
duration = duration.plus(100);
duration = duration.minus(20);
System.out.println("duration = " + duration)
```

Output:
```
duration = PT1.080S
```

Periods …

- "A period of time defined in terms of fields"

```
Period period = new Period(1000); // 1 second
System.out.println("period = " + period);
```

Output:
```
period = PT1S
```

```
// A Period can be constructed with field values
// or a duration in milliseconds.  Create a period of
// 2 hours 57 minutes 15 seconds 0 milliseconds.
period = new Period(2, 57, 15, 0);
System.out.println("period = " + period);
```

Output:
```
period = PT2H57M15S
```

```
// A Period can be added to an instant to get a new instant.
// The date specified in the next line is March 4, 2007, 8 am.
// Unlike in the JDK, the value for January is 1 instead of 0.
DateTime startTime = new DateTime(2007, 3, 4, 8, 0, 0, 0);
DateTime endTime = startTime.plus(period);
System.out.println("endTime = " + endTime);
```

Output:
```
endTime =
2007-03-04T10:57:15.000-06:00
```

… Periods …

```
// Periods must be converted to durations in order to
// compare them.  This is because the exact length of period
// can vary based on context.  For example, a period of one day
// can be 23 hours instead of 24 when the context is
// a day in which daylight savings time is changing.

Hours hours = Hours.hoursBetween(startTime, endTime);
System.out.println("hours = " + hours);

Minutes minutes = Minutes.minutesBetween(startTime, endTime);
System.out.println("minutes = " + minutes);
```

Output:
```
hours = PT2H
minutes = PT177M
```

… Periods

```
// Using a MutablePeriod is the most efficient way
// to represent a duration that needs to be modified
// when concurrent access isn't an issue.
// Compare this to code in the durations examples.
MutablePeriod mp = new MutablePeriod(1000);
mp.add(100);
mp.add(-20);
System.out.println("mp = " + mp);
```

Output:
```
mp = PT1.080S
```

Chronologies

- Calendar system options
  - Gregorian - standard since 10/15/1582
    - defines every fourth year as leap, unless the year is divisible by 100 and not by 400
  - Julian - standard before 10/15/1582
    - defines every fourth year as leap
  - GJ
    - historically accurate with Julian followed by Gregorian starting on 10/15/1582
  - ISO - the default
    - based on the ISO8601 standard
    - same as Gregorian except for treatment of century
  - Buddhist, Coptic, Ethiopic - not commonly used
- For most applications, the default ISOCronology is appropriate

JSR-310

- From Stephen Colebourne,
  creator of Joda Time and JSR-310 spec. lead
  - "JSR-310 is inspired by Joda-Time,
    rather than a straight adoption of it.
    Thus, there are key differences in the APIs.
    JSR-310 isn't at the download and use stage yet."
- To keep up with the status
  of this standardization effort,
  browse https://jsr-310.dev.java.net/