The Sector Specialists

JSR-310 and ThreeTen

The New API for Date and Time in Java 8

Grzegorz Borkowski

About me and my company

- For 7 years a Java developer, for 4 years a team leader in different Java projects
- For 2 years I have been working in Rule Financial as a Lead Consultant
    - starting this year, also as a Java Focus Group Leader

- Rule Financial is a provider of IT and software services to the investment banking, including top-10 global investment banks. We have offices in London, New York, Toronto, Barcelona and Łódź. In the Łódź office, we currently hire almost 200 developers. See www.rulefinancial.com
- If you want to contact me (re JSR-310, or working at Rule Financial), drop me an email: grzegorzbor@gmail.com or grzegorz.borkowski@rulefinancial.com

Agenda

- ISO-8601
- Current date/time support in JDK, and its limitations
- General problems with date/time handling
- What JSR-310 provides
- Examples of application
- Miscellaneous (current status, JSR-310 vs Joda, etc)
- Q&A

ISO-8601

- An international standard covering the exchange of date and time-related data
- Examples of data formats:
  - 2012-10-22
  - 2012-10-22T16:48:63.000+02:00
  - 2012-10-22T16:48Z // "Z" means UTC/GMT
  - 05:00-04:30
- Time zone calculations:
  - 12:00Z = 14:00+02:00 = 7:30-04:30
- Note: human/locale representation vs machine/standard representation:
  - 2012-10-22 T16:48+01:00 // as sent from/to web service
  - 22/10/2012 , 4:48 PM BST // as displayed in GUI

The current set

- Standard JDK:
  - java.util.Date
  - java.util.Calendar
  - java.sql.Date/Time/Timestamp
  - javax.xml.datatype.Duration/XmlGregorianCalendar
  - java.text.DateFormat
- Libraries:
  - Joda Time – the most popular and advanced

Problems with standard classes

- java.util.Date/Calendar –
  - they are mutable
  - their API is far from perfect
    - e.g. months are counted from 0 through 11
- java.sql.Date/Time/Timestamp – it's even worse

– e.g. methods which take no arguments but throw IllegalArgumentException

– famous javadoc for Timestamp:

  - *Due to the differences between the Timestamp class and the java.util.Date class mentioned above, it is recommended that code not view Timestamp values generically as an instance of java.util.Date. The inheritance relationship between Timestamp and java.util.Date really denotes implementation inheritance, and not type inheritance.*

Missing in JDK

- How to model a date without a time component – e.g. 1 Mar 2012
- How to model time without a date – e.g. 11:00
- Time with vs without timezone – e.g. 11:00 vs 11:00 CEST
- How to model month without a day – e.g. a payment list can be linked to "Oct 2012"
- How to model duration, e.g. marathon record "02:03:38"
- No support for virtual clock pattern in JDK - how to test applications which use System.currentTimeMillis?

General problems with dates and time

- What will happen if we add one year to 29 Feb 2004?
- What will happen if we add one month to 31 Mar?
- Date comparison logic, eg. does 1:30+02:00 == 2:30+03:00 ?
- Time change problems (summer/winter time): time jumps forward or backward, the given time can happen twice or never happen; e.g. what will happen if we add one hour to 2012-10-28 02:15 (note – at 3:00 on that day we'll move our clocks back by 1 hour, so we'll be back at 2:00)

It's not easy!

```
GregorianCalendar greogorianCalendar = new GregorianCalendar();
```
vs:
```
Calendar rightNow = Calendar.getInstance();
```

- getInstance() will return GregorianCalendar, in most countries of the world... but if you haven't test it with Thailand locale, you could be out of luck in that country
- Also, if you trusted iPhone alarm clock in October 2010, you were out of luck... it went off one hour later after time change
- If you used XSLT functions for date time conversion, you could be out of luck too – it used to take current offset, instead of offset related to the processed date

Workarounds

- Use String, e.g. “2012-03-01”
- Use java.util.Date with "normalized" components – like java.sql.Date does
  - but if your timezone changes by one hour, your date can change by one day
  - also, in some timezones, during a time change, there can be no midnight
- Write your own class
- Use Joda Time, e.g. LocalDate

- Integration problems (JDBC, JPA, XML etc)

What JSR-310 provides

- Well designed, consistent, modern API, based on immutable classes
- Two models of time: “machine” and “human”
- Machine time
    - Computers treat time as a counter, based on some oscillator and some reference point. Such time can be continuous or not.
- Human time
    - Humans treat time as a set of predefined fields (year X, month Y, day Z, plus maybe hour, minute, second).

What JSR-310 provides

- Machine time
  - **javax.time.Instant** – a point on the time-line with nanosecond precision; a reference point is 1 Jan 1970 ("unix epoch").
  - **javax.time.Duration** – difference (in nanosec) between two Instants, can be positive or negative
- What can you do with these classes? Not that much.
  - You can compare two instances – which one was first
  - Can be used in logs, audits, etc

What JSR-310 provides

- Human time
  - **javax.time.LocalDate** – date w/o time and offset, e.g. 2007-12-03
  - **javax.time.LocalTime** – time w/o date and offset, e.g. 10:15:30
  - **javax.time.LocalDateTime** – date and time w/o offset,
    e.g. 2007-12-03T10:15:30
  - **javax.time.ZoneOffset** – offset against UTC (positive or negative),
    e.g. +05:00, +01:00, -02:00, +04:30, Z, CEST, UTC, GMT
  - **javax.time.OffsetDate** – date w/o time but with offset, e.g. 2007-12-03+02:00
  - **javax.time.OffsetTime** – time w/o date but with offset, e.g. 10:15:30+02:00
  - **javax.time.OffsetDateTime** – date with time and offset,
    e.g. 2007-12-03T10:15:30+02:00

What JSR-310 provides

- Human time - cont.
  - **javax.time.ZoneId** – time zone identifier, e.g "Europe/Warsaw"
  - **javax.time.ZonedDateTime** - date with time with offset with time zone,
  - e.g. 2007-12-03T10:15:30+02:00[Europe/Warsaw]
  - **javax.time.Period** – time unit with mulitplier, e.g. "1 hour", "5years"

What JSR-310 provides

- Other classes
  - **javax.time.Clock** – virtual clock, can be bound to system clock, or to fixed time, can tick with more granular precision, e.g. tick by one second
  - **javax.time.chrono.ISOChronology** – standard Gregorian calendar (other can be available too)
  - **javax.time.Year**, **YearMonth**, **MonthDay**, **QuarterOfYear** – representations of a year (e.g. 2012), a month in a year (e.g. 2012-10), day of month (e.g. 15 października), quarter (e.g Q3 2012)
  - **javax.time.format.DateTimeFormatter** – date/time parser and formatter
  - ...**plus** a set of more advanced classes (low- and high-level)

Examples

- Date without time component - e.g. 1 Mar 2012

```
LocalDate firstMarch2012 = LocalDate.of(2012, 03, 01); //or:
LocalDate firstMarch2012 = LocalDate.parse("2012-03-01");
```

- Time without date component - e.g. 11:00

```
LocalTime elevenAm= LocalTime.of(11, 00); //or:
LocalTime elevenAm = LocalTime.parse("11:00");
```

Examples

- Time with vs time without offset, e.g. 11:00 vs 11:00 CEST

```
LocalTime elevenAmNoZone = LocalTime.of(11, 0);
OffsetTime elevenAmCest = OffsetTime.of(11, 0, ZoneOffset.ofHours(2));
//or:
OffsetTime elevenAmCest = OffsetTime.of(11, 0, ZoneOffset.of("+02:00"));
```

Examples

- A payment list for "October 2012" can be linked to

```
YearMonth october = YearMonth.of(2012, Month.OCTOBER);
```

- Duration – e.g. marathon record "02:03:38"

```
//a bit problematic:
Duration marathonTime =
    Duration
    .ofHours(2)
    .plus(3, LocalPeriodUnit.MINUTES)
    .plusSeconds(38);
```

Examples

- Virtual clock support – avoid System.currentTimeMillis()

```
//bad:
LocalDateTime now = LocalDateTime.now();
//better:
@Inject Clock clock;
LocalDateTime now = LocalDateTime.now(clock);

//possible injectors:
clock = Clock.systemDefaultZone();
clock = Clock.systemUTC();
clock = Clock.systemZoneId("Europe/Warsaw");
clock = Clock.fixedUTC(OffsetDateTime.parse("2012-10-
26T09:00Z").toInstant());
```

Examples

- What will happen if we add one year to 29 Feb 2004?

```
    LocalDate lastFebruary2004 = LocalDate.of(2004, 2, 29);
    LocalDate lastFebruary2005 = lastFebruary2004.plusYears(1);  // 2005-02-28
```

- What will happen if we add one month to 31 Mar?

```
    MonthDay lastMarch = MonthDay.of(3, 31);
    LocalDate lastMarchThisYear = lastMarch.atYear(2012);
    LocalDate lastMarchPlusMonth = lastMarchThisYear.plusMonths(1);  // 2012-
04-30
```

Examples

- Date comparison logic, eg. does 1:30+02:00 == 2:30+03:00?

```
OffsetTime oneThirty = OffsetTime.of(1, 30, ZoneOffset.ofHours(2));
OffsetTime twoThirty = OffsetTime.of(2, 30, ZoneOffset.ofHours(3));
oneThirty.equals(twoThirty); //false
oneThirty.equalInstant(twoThirty);  //true
```

Examples

- What will happen if we add one hour to 2012-10-28 02:15 (note – at 3:00 on that day we'll move our clocks back by 1 hour, so we'll be back at 2:00)

```
ZonedDateTime twoFifteen = ZonedDateTime.of(
   OffsetDateTime.of(2012, 10, 28, 2, 15, 0, 0, ZoneOffset.ofHours(2)),
   ZoneId.of("Europe/Warsaw")); //2012-10-28T02:15+02:00[Europe/Warsaw]

ZonedDateTime twoFifteenAfterOneHour = twoFifteen.plusHours(1);
                                //2012-10-28T03:15+01:00[Europe/Warsaw]

ZonedDateTime twoFifteenAfterOneHourFromInstant =
   ZonedDateTime.ofInstant(twoFifteen.toInstant().plusSeconds(3600),
   ZoneId.of("Europe/Warsaw")); //2012-10-28T02:15+01:00[Europe/Warsaw]
```

Why proper date/time modeling is important

Use case: a timetracking system. A user in Poland entered a work time:
1st Oct 2012, 9:00-17:00

- Now this user (or his/her manager) moves to a different time zone (say NY time, which is -04:00). What should they see?
  - if modeled as LocalDateTime – no change, it's still 9-17
  - if modeled as Offset/ZonedDateTime – possibly no change, 9-17, but with notice "this is in Poland timezone"
  - if modeled as Instants – it's displayed in NY timezone as 3AM – 11AM
- Also, be careful how it gets translated to the database
  - and what if you relocate DB to a different timezone?

JSR-310 vs Joda Time

- JSR-310 is a "better Joda Time"
  - Joda Time has some problems and design mistakes - see http://blog.joda.org/2009/11/why-jsr-310-isn-joda-time_4941.html
  - Stephen Colebourne, creator of Joda, is also the lead of JSR-310.
- Joda has two basic concepts: Instant and Partial. JSR-310 has machine and human time. The relationship is not 1:1, e.g. Joda's DateTime is an Instant in Joda but it's not the same as Instant in JSR-310.
- Joda is mature and stable, recommended for production usage. ThreeTen is not yet stable enough.

JSR-310 – module size

- For Java EE, library size does not matter – bigger API can be useful
- For Java ME, size is critical – all the helper methods like "plusMonths()" are problematic
- Different solutions are being currently considered

Summary

- JSR-310 – available in Java 8
- Project ThreeTen – reference implementation (can be also run on Java 7)
- Don't use java.util.Date/Calendar! Use Joda Time (and in the future, use JSR-310)
- Never use System.currentTimeMillis() inside business logic
- Use ISO standard to format dates and times when exchanging between systems