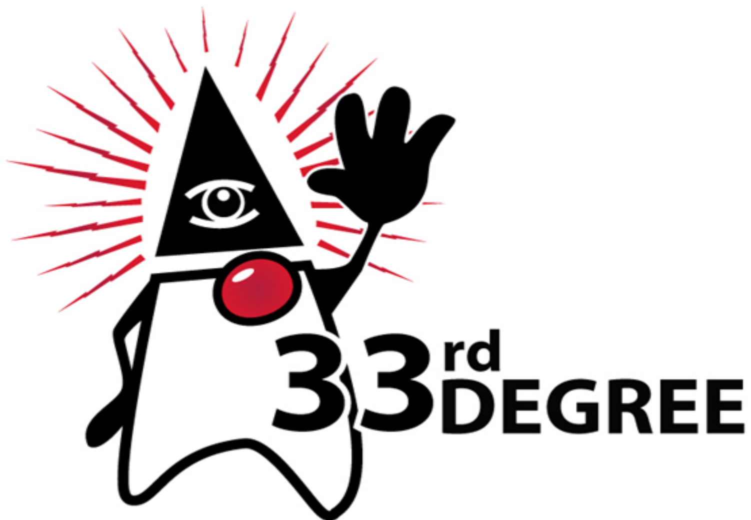
LUXOFT
SOFTWARE IMAGINEERING
PLATINIUM SPONSOR

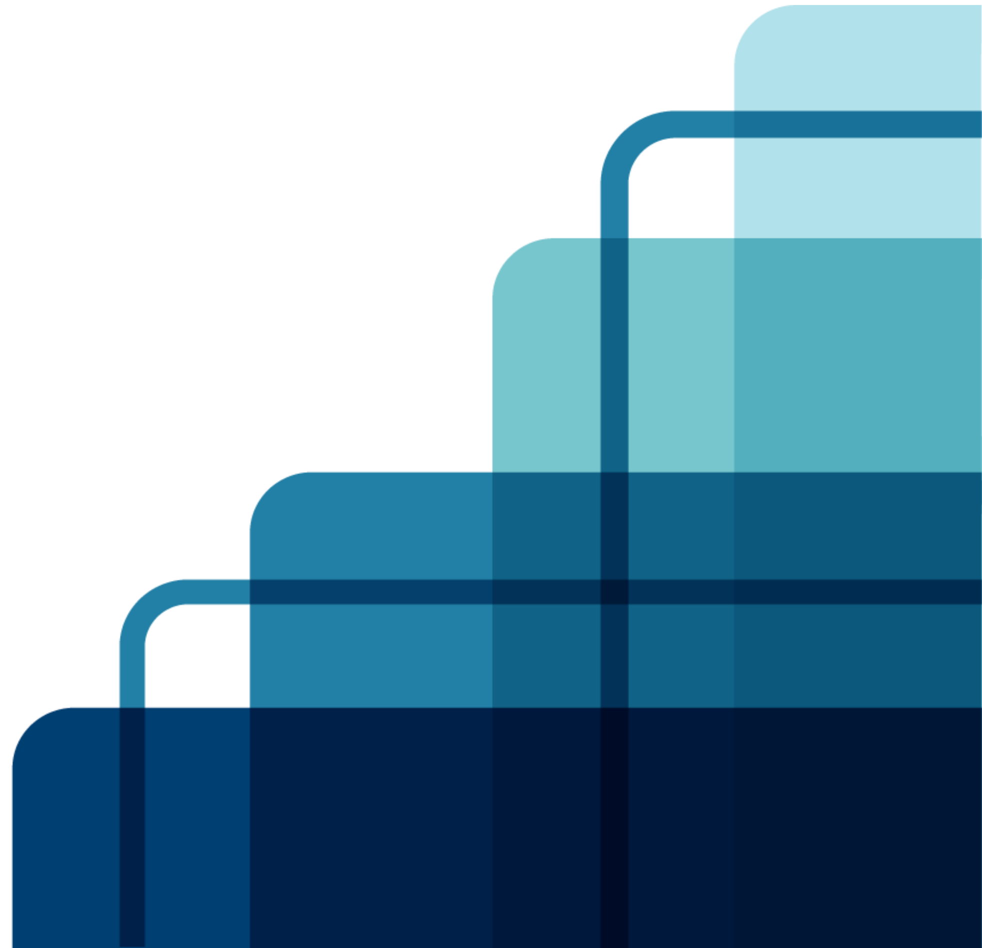
Grzegorz Borkowski

# THREETEN (JSR 310)



The Sector Specialists

# JSR-310 and ThreeTen



## The new Date and Time API in Java 8

Grzegorz Borkowski

# About me

- Grzegorz Borkowski
  - For 8 years a Java developer
  - For 2 years working in Rule Financial as Lead Consultant and 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, Łódź and Poznań. In Łódź and Poznań, we currently hire more than 200 developers, and we are still growing. See www.rulefinancial.com

- Contact: grzegorzbor@gmail.com or grzegorz.borkowski@rulefinancial.com

rule FINANCIAL

© Rule Financial 2012

# Agenda

- Current date/time support in JDK, and its limitations
- ISO-8601
- What JSR-310 provides (with examples)
  - Machine time
  - Local dates
  - Years and months
  - Timezones
  - Durations
- Q&A

rule FINANCIAL

© Rule Financial 2012

# The current set in JDK

- Basic set
  - java.util.Date
  - java.util.Calendar
- JDBC
  - java.sql.Date
  - java.sql.Time
  - java.sql.Timestamp
- Other
  - javax.xml.datatype.Duration/XmlGregorianCalendar
  - java.text.DateFormat

rule FINANCIAL

© Rule Financial 2012

# Problems: limited modelling capabilities

- How to model:
  - a date without a time component – e.g. 1 Mar 2012
  - time without a date – e.g. 11:00
  - time with vs without timezone – e.g. 11:00 vs 11:00 CEST
  - year-month – e.g. a payment list can be linked to "Oct 2012"
  - duration, e.g. marathon record "02:03:38"

rule FINANCIAL

© Rule Financial 2012

# Problems: poor API design and questionable implementation decisions

- Date/Calendar/sql.* classes are mutable
- Have methods which take no arguments but throw IllegalArgumentException
- Months in Calendar are counted from 0 through 11
- famous javadoc for Timestamp:

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

rule FINANCIAL

© Rule Financial 2012

# 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)

rule FINANCIAL

© Rule Financial 2012

# 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 the lead of JSR-310.
- Joda is mature and stable, recommended for production usage. JSR-310 is not yet stable enough.
- Joda has two basic concepts: Instant and Partial. JSR-310 uses slightly different approach, there is no such division.

rule FINANCIAL

© Rule Financial 2012

# ISO-8601

- An international standard covering the exchange of date and time-related data
- Examples of data formats:
  - 2012-10-22
  - 2012-10-22 T16:48Z // "Z" means UTC/GMT
  - 2012-10-22 T16:48:63.000+02:00
  - 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 webservice
  - 22/10/2012 , 4:48 PM BST // as displayed in GUI

rule FINANCIAL

© Rule Financial 2012

# What JSR-310 provides

- Well designed, consistent, modern API
- Five modules:
  - java.time – the main API
  - java.time.temporal – lower-level API for accessing the fields and units of date-time
  - java.time.zone – lower-level API to handle time-zones
  - java.time.format – printing and parsing date-time objects and strings
  - java.time.calendar – alternate calendar systems (Islamic, Japanese, Taiwan, Thai)


© Rule Financial 2012

# JSR-310 design principles

- No constructors, only factory methods

```
LocalDate firstMarch2012 = LocalDate.of(2012, 03, 01);
LocalDate firstMarch2013 = LocalDate.parse("2013-03-01");
```

- Immutable classes

```
firstMarch2012.plusYears(3); // wrong – lost assignment!
System.out.println(firstMarch2012); // still 2012-03-01
```

- Consistent method names

```
of(), parse(), plus(), minus(), with()
firstMarch2012.withYear(2015); // 2015-03-01
```

rule FINANCIAL

© Rule Financial 2012

# JSR-310 - two models of time

- 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).

rule FINANCIAL

© Rule Financial 2012

# Machine time

- **java.time.Instant** – a point on the time-line with nanosecond precision; a reference point is 1 Jan 1970 ("unix epoch").
  - Can be used in logs, audits, etc.

rule FINANCIAL

© Rule Financial 2012

# Machine time

- **java.time.Instant** – a point on the time-line with nanosecond precision; a reference point is 1 Jan 1970 ("unix epoch").
  - Can be used in logs, audits, etc.

```
Instant now = Instant.now();
Instant twoSecondsLater = now.plusSeconds(2);
```


© Rule Financial 2012

# Local date and time

- **java.time.LocalDate** – date without time and zone, e.g. 2007-10-31
- **java.time.LocalTime** – time withouth date and zone, e.g. 10:15:30
- **java.time.LocalDateTime** – date and time without zone, e.g. 2007-10-31T10:15:30

rule FINANCIAL

© Rule Financial 2012

# Local date and time

- **java.time.LocalDate** – date without time and zone, e.g. 2007-10-31
- **java.time.LocalTime** – time withouth date and zone, e.g. 10:15:30
- **java.time.LocalDateTime** – date and time without zone, e.g. 2007-10-31T10:15:30

```
LocalDate date = LocalDate.of(2007, 10, 31);
LocalTime time = LocalTime.of(10, 15);
LocalDateTime dateTime = LocalDateTime.of(date, time);
assertThat(dateTime.getHour(), equalTo(10));
//what if there is no such day?
LocalDate oneMonthLater = date.plusMonths(1);
assertThat(oneMonthLater.getDayOfMonth(), equalTo(30));
```

rule FINANCIAL

© Rule Financial 2012

# Years, months

- **java.time.Year** – pure year, e.g. 2007
- **java.time.Month** – enum
- **java.time.YearMonth** – e.g. 2007-10
- **java.time.MonthDay** – October 31st


© Rule Financial 2012

# Years, months

- **java.time.Year** – pure year, e.g. 2007
- **java.time.Month** – enum
- **java.time.YearMonth** – e.g. 2007-10
- **java.time.MonthDay** – October 31st

```
Year year = Year.of(2000);
YearMonth february = year.atMonth(Month.FEBRUARY);
if (year.isLeap()) {
    assertThat(february.lengthOfMonth(), equalTo(29));
} else {
    assertThat(february.lengthOfMonth(), equalTo(28));
}
```

rule FINANCIAL

© Rule Financial 2012

# Timezones

- **javax.time.ZoneId** – time zone identifier, e.g “Europe/Warsaw”
- **javax.time.ZoneOffset** – subclass of ZoneId, simple offset against UTC (positive or negative), e.g. +05:00, +01:00, -02:00, +04:30, Z, CEST, UTC, GMT
- **javax.time.ZonedDateTime** - date with time with offset with time zone, e.g. 2007-12-03T10:15:30+01:00[Europe/Warsaw]

rule FINANCIAL

© Rule Financial 2012

# Timezones

- **javax.time.ZoneId** – time zone identifier, e.g "Europe/Warsaw"
- **javax.time.ZoneOffset** – subclass of ZoneId, simple offset against UTC (positive or negative), e.g. +05:00, +01:00, -02:00, +04:30, Z, CEST, UTC, GMT
- **javax.time.ZonedDateTime** - date with time with offset with time zone, e.g. 2007-12-03T10:15:30+01:00[Europe/Warsaw]

```
ZoneId zonePL = ZoneId.of("Europe/Warsaw");
LocalDateTime localDateTime = LocalDateTime.of(2013, 3, 1, 12, 0);
ZonedDateTime firstDecemberMidday = ZonedDateTime.of(localDateTime, zonePL);
assertThat(firstDecemberMidday.getOffset().getId(), equalTo("+01:00"));
ZonedDateTime firstJuneMidday = firstDecemberMidday.withMonth(6);
assertThat(firstJuneMidday.getOffset().getId(), equalTo("+02:00"));
```


© Rule Financial 2012

# Virtual clock abstraction

- **java.time.Clock** – virtual clock, **should be** injected and used by **all** date/time-related calculations; avoid using System.currentTimeMillis()
  - Different implementations available: can be bound to system clock, or to fixed time, can tick with more granular precision, e.g. tick by one second

rule FINANCIAL

© Rule Financial 2012

# Virtual clock abstraction

- **java.time.Clock** – virtual clock, **should be** injected and used by **all** date/time-related calculations; avoid using System.currentTimeMillis()
  - Different implementations available: can be bound to system clock, or to fixed time, can tick with more granular precision, e.g. tick by one second

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

rule FINANCIAL

© Rule Financial 2012

# Durations

- **java.time.Duration** – duration in time units (seconds, millis, nanos)
  - basically, the difference between two Instances
  - can be positive or negative
- **java.time.Period** – duration in date units (days, months, years)
  - useful for modelling calculations over months with different lenghts or over DST changes, e.g. a day can have 23 or 25 hours if there is a time change

rule FINANCIAL

© Rule Financial 2012

# Durations

- **java.time.Duration** – duration in time units (seconds, millis, nanos)
  - basically, the difference between two Instances
  - can be positive or negative
- **java.time.Period** – duration in date units (days, months, years)
  - useful for modelling calculations over months with different lenghts or over DST changes, e.g. a day can have 23 or 25 hours if there is a time change

```
Duration timeBetween = Duration.between(timeStart, timeEnd);
Duration marathonRecord = Duration.ofHours(2).plusMinutes(3).plusSeconds(38);
Period oneWeek = Period.ofDays(7);
```

rule FINANCIAL

© Rule Financial 2012

# Other classes

- **java.time.OffsetTime** – time w/o date but with offset, e.g. 10:15:30+02:00
- **java.time.OffsetDateTime** – date with time and offset, but no zone ID, e.g. 2007-12-03T10:15:30+02:00
  - useful for interaction with systems that do not support storing zone ID, e.g. databases or ISO-complient web services
- **java.time.format.DateTimeFormatter** – date/time parser and formatter
- **java.time.zone.ZoneRules** – keeps all the rules related to DST for given zone
- **java.time.zone.ZoneRulesProvider** – can be used to query for all available zone IDs


© Rule Financial 2012

# Why proper date/time modelling 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 different time zone (say NY time, which is -04:00). What should they see?
  - if modelled as LocalDateTime – no change, it's still 9-17
  - if modelled as Instants – it's displayed in NY timezone as 3AM – 11AM
  - if modelled as Offset/ZonedDateTime – possibly no change, 9-17,  but with notice "this is in Poland timezone"; or 3AM – 11M, with notice "converted to your local time"
- Also, be careful about conversions, e.g. how it gets translated to a database
  - and what if you relocate DB to a different timezone?


© Rule Financial 2012

# Design problems

- Date comparison logic
  - e.g. does 1:30+02:00 equals 2:30+03:00?
  - what about implementing Comparable – should it be consistent with equals?
    - example: BigDecimal
      1.0 equals 1.00 returns false, but 1.0 compareTo 1.00 returns 0
- Zones and offsets
  - offset is enough to identify the date precisely; but is not enough for adding/removing durations (because of DST)
  - zone ID is enough, in most cases, to identify the offset for given date; but not always, example:
    2012-10-28T**02**:15+**02**:00[Europe/Warsaw]
    2012-10-28T**02**:15+**01**:00[Europe/Warsaw] //one hour later

rule FINANCIAL

© Rule Financial 2012

# Summary

- JSR-310 – available in Java 8 this year
- Backport available for Java 7
- Dos and don'ts:
  - Don't use java.util.Date/Calendar! Use Joda Time (and in the future, use JSR-310)
  - Use proper classes to model concepts accurately – e.g. do not use date with timezone (like java.util.Date) for modelling date of birth
  - Never use System.currentTimeMillis() inside business logic
  - Use ISO standard to format dates and times when exchanging them between systems

rule FINANCIAL

© Rule Financial 2012