(Preprint) AAS 13-521

DATE AND TIME FOR NINE MILLION JAVA DEVELOPERS

Stephen Colebourne*

The Java programming platform is used by nine million developers worldwide. The next release, v1.8 due early next year, will include a new date and time API developed as the JSR-310 specification. As part of defining this specification, the handling of leap seconds was discussed, resulting in a number of options. The final choice was to define a "Java time-scale" that avoids exposing the concept of leap seconds to developers. The motivations behind the choice made will be discussed.

INTRODUCTION

The Java platform is the most widely used platform for developing business applications. It is particularly widely used in server-side development, where it is used to provide the software behind the many of the world's best known websites. Estimates suggest that there are currently nine million Java developers worldwide.

Developers interact with two parts of the platform day-to-day - the programming language itself, and the standard Application Programming Interface (API). Each new release of Java extends and enhances both the language and the API, enabling developers to work more effectively. Preparations are underway for the release of version 1.8 of the platform in February 2014 which will include a new date and time API. This paper discusses the overall the API design with a focus on leap-seconds.

DATE AND TIME APIS IN JAVA

The date and time APIs in the standard API have always been regarded as being very poor. The first API was added in version 1.0 and consists essentially of a single class called *Date*. Unfortunately, the *Date* class does not represent a date; instead it represents an instant along the timeline to millisecond precision. The class also contained methods that only worked in one time-zone. The class is so bad that most of its methods have now been deprecated, such that they should not be used.

The second API was added in version 1.1 by converting code from an existing library in the C programming language. This consists of two classes: *Calendar* and the subclass *GregorianCalendar*. Continuing with poor naming, neither represents a calendar system, instead they represent a date and time in a controllable time-zone, with *Calendar* being calendar-system neutral, and *GregorianCalendar* being specific to the Julian-Gregorian calendar system as used by the Vatican with cutover in 1582.

* Member of Technical Staff, OpenGamma Ltd, 185 Park Street, SE1 9BL, London.

While this second API has sufficient power for most basic needs, the conversion from C resulted in a design that does not fit well with Java. As a result, it is easy to make mistakes. For example, midnight is used to represent a date, which fails in those time-zones where daylight savings time leaps forward in the spring missing midnight altogether. Similarly, 1970-01-01 is used by developers as the date part when representing just a time, without considering that in London there were special daylight savings in place that year such that the 1st January was in "summer time", not "winter time" as developers might expect. Finally, the API uses the value zero for January through to 11 for December.

Given these issues, the author developed a library, Joda-Time, which developers could use as a quality alternative to the standard API.[*] Experience developing Joda-Time eventually provided enough knowledge to allow the author to start JSR-310, an official process to change the standard Java APIs.

REQUIREMENTS

The new date and time API for Java, developed under the name JSR-310,[†] set out to provide "a new and improved date and time API for Java. The main goal is to build upon the lessons learned from the first two APIs (*Date* and *Calendar*) in Java SE, providing a more advanced and comprehensive model for date and time manipulation."

The key goal was to create separate classes to represent concepts normally found in business applications. These include a date without time or time-zone which would be used for a birthdate, a time without date or time-zone which would be used for the local time that a shop opens, and an instant in time without time-zone which would be used for event timestamps.

Additional goals included a design that allows developer code to closely match business requirements where possible, avoid common errors seen in date and time code, plus support for complex formatting and parsing to and from text. Conversion to and from XML and database environments was also necessary.

Finally, the API had to be designed to support an element of extensibility, to allow developers to modify behavior. This is necessary, as new Java releases are so infrequent that it is not possible for developers to wait for the next release for an enhancement.

DESIGNING THE API

API design is part experience and part subjective. In this case, the basic classes were fairly obvious to define, although less so to name.[‡] These turned out to be as follows:

- *LocalDate*—represents a date, without time or time-zone, such as "2012-06-30".
- *LocalTime*—represents a time, without date or time-zone, such as "12:30".
- *LocalDateTime*—combines *LocalDate* and *LocalTime*, such as "2012-06-30T12:30".

---

[*] http://joda-time.sourceforge.net/
[†] http://jcp.org/en/jsr/detail?id=310
[‡] Each Java API consists of a number of classes and interfaces, each of which define a number of methods. Interfaces are used at a high level to specify functionality without defining the exact details of how the functionality is performed. Classes implement the interfaces to actually define those details. Each individual feature is exposed as a method.

- *ZonedDateTime*—a date and time in a time-zone, such as "2012-06-30T12:30+01:00 Europe/London".
- *Instant*—an instantaneous point on the time-line, without reference to time-zone.
- *Duration*—a time-based amount of time, measured in nanoseconds, such as "P123.456S".
- *Period*—a date-based amount of time, in years, months and days, such as "P2Y6M3D".

These seven classes form the basis of the date and time classes that developers will use. These concepts and names would also be applicable to those writing APIs in other programming languages.

While the API classes above are the principle elements that developers first see, their day-to-day experience is driven by the methods each class exposes. This is where detailed design comes into play.

An early decision was to support nanosecond precision for times, such as in *LocalTime*, *LocalDateTime*, *ZonedDateTime* and *Instant*. This was primarily driven by the ability of some databases to store time information at this level. It was not driven by the ability, or otherwise, of obtaining a clock that could provide nanosecond resolution.

A further decision was to focus effort on a single calendar system. While there are many different calendar systems in the world, and some are in active daily use, there is only one *de facto* world civil calendar system. In the API this is referred to as the ISO calendar system as it is based on the description in the ISO-8601 date-time standard. As such, the meaning of concepts such as year, month and day in the main classes outlined above is very clear.

Another key decision was to support "nice" methods for easy access to the various fields of date and time. Thus, the *LocalDate* class has methods such as *getYear()*, *getMonth()*, *getDayOfMonth()* and *getDayOfWeek()*. For *LocalTime*, methods include *getHour()*, *getMinute()*, *getSecond()* and *getNano()*. Each of the time methods refers to that unit within the next larger, thus *getSecond()* is the second-of-minute field, and *getNano()* is the nano-of-second field. The end result of the design was an API that allows business logic to be clearly expressed:

```
LocalDate customerBirthday = customer.loadBirthdayFromDatabase();
LocalDate today = LocalDate.now();
if (customerBirthday.equals(today)) {
  LocalDate offerExpiryDate = today.plusWeeks(2).with(next(FRIDAY));
  customer.sendBirthdaySpecialOffer(offerExpiryDate);
}
```

IMPORTANCE OF LEAP SECONDS

During the API design it was clear that the issue of leap-seconds and UTC would have to be addressed, one way or another. However, it should be clear from the above that leap seconds were just one design challenge in an API for general business use.

The reality is that leap seconds are not an important problem for most developers, who work in terms of high level concepts like birth dates and event timestamps. There are four main types of complication within date and time that developers have to cope with and manage:

- months with different lengths
- leap years
- daylight saving time
- leap seconds

Through experience, the author can say that consideration is given to these complications in the order specified. In addition, experience tells us that many developers fail to get daylight saving time right, and a few fail to properly consider leap years and even variable length months. In the author's experience, leap seconds are never actively considered by developers.

Possible approaches to the perceived lack of importance of leap seconds would be education or an API design that forced exposure of leap seconds. The opinion of the author is that neither is practical across a group of nine million developers.

LEAP SECOND DESIGN OPTIONS

A number of options were considered for leap second support. The first option was full support in the main classes. This option would mean that the "nice" method *getSecond()* would occasionally return the value 60, rather than its normal range of 0 to 59, when a leap second occurs. This design would expose leap seconds to every developer. They would not expect a value of 60, nor would they test for it, resulting in applications that go wrong whenever a leap second occurs. This option was thus rejected.

(The situation is in fact more complex than not expecting the value of 60. The API supports second-based time-zone offsets from UTC/Greenwich. Thus, +01:15:34 is a valid offset from UTC/Greenwich. Given these offsets are valid, a leap second could occur in the *middle* of a minute, not just the end).

The second option was time-scale abstraction. This option involved changing the design so that an abstraction of time-scales would be at the center. Developers would be exposed to time-scales, and be expected to learn the differences between UTC, TAI and so on. This approach had some promise and was fully prototyped. Most developers however, have no experience with time-scales and have enough difficulty with the other complications of date and time outlined above. Forcing them to learn and understand time-scales was not ultimately a viable option and was thus rejected.

The third option was to have special classes and methods to support leap seconds, features which most users could ignore. While initially attractive, the extra classes and methods do still impact on the overall API. Developers would see them and wonder if they should be using them or not. The leap second time could not be entirely hidden and would "leak out" into other parts of the API where it would not be expected. In addition, certain websites for developer questions would have answers by gurus suggesting that the "right" way was to use the leap second feature without explanation of the associated costs. The inability to successfully hide leap seconds caused this option to be rejected.

The fourth option was no support for leap seconds. This simply involved ignoring the problem and hoping it would go away. This is essentially what the existing APIs in Java do, as they specify no leap second behavior. Had UTC been changed in 2012, this would have been a tempting option. Because future leap seconds were not abandoned, this option was rejected. What was needed therefore was an option which did not expose leap seconds to developers, yet still provided an element of support.

JAVA TIME-SCALE

The chosen option to support leap seconds in Java is the creation of a time-scale that effectively removes leap-seconds. Despite the unusual nature of this concept, it turns out to be a simple and very effective solution to the problem. The time-scale is defined as follows:[*]

> The Java Time-Scale divides each calendar day into exactly 86400 subdivisions, known as seconds. These seconds may differ from the SI second. It closely matches the *de facto* international civil time scale, the definition of which changes from time to time.
>
> The Java Time-Scale has slightly different definitions for different segments of the time-line, each based on the consensus international time scale that is used as the basis for civil time. Whenever the internationally-agreed time scale is modified or replaced, a new segment of the Java Time-Scale must be defined for it. Each segment must meet these requirements:
>
> - the Java Time-Scale shall closely match the underlying international civil time scale;
> - the Java Time-Scale shall exactly match the international civil time scale at noon each day;
> - the Java Time-Scale shall have a precisely-defined relationship to the international civil time scale.
>
> There are currently, as of 2013, two segments in the Java time-scale.
>
> For the segment from 1972-11-03 (exact boundary discussed below) until further notice, the consensus international time scale is UTC (with leap seconds). In this segment, the Java Time-Scale is identical to UTC-SLS. This is identical to UTC on days that do not have a leap second. On days that do have a leap second, the leap second is spread equally over the last 1000 seconds of the day, maintaining the appearance of exactly 86400 seconds per day.
>
> For the segment prior to 1972-11-03, extending back arbitrarily far, the consensus international time scale is defined to be UT1, applied proleptically, which is equivalent to the (mean) solar time on the prime meridian (Greenwich). In this segment, the Java Time-Scale is identical to the consensus international time scale. The exact boundary between the two segments is the instant where UT1 = UTC between 1972-11-03T00:00 and 1972-11-04T12:00.

This definition, refined thanks to Andrew Main, is effectively based on UTC-SLS - smoothed leap seconds.[†] By taking each leap second and spreading it equally across the last 1000 seconds of the day of the leap second, most developers have no experience of the leap. Time continues to flow normally, and the second-of-minute field continues to run from 0 to 59, never reaching 60.

---

[*] http://hg.openjdk.java.net/threeten/threeten/jdk/file/ba50227182b7/src/share/classes/java/time/Instant.java
[†] http://www.cl.cam.ac.uk/~mgk25/time/utc-sls/

By defining what is supposed to happen around a leap second, the API "supports" leap seconds. Yet by redefining the second to be an 86400^{th} subdivision of the day most developers do not have to experience them.

PRACTICAL EFFECTS

In practical terms, the Java time-scale approach involved no changes to the code, only changes to documentation. Except in the one place where the approach is documented, API methods ignore leap seconds entirely. Thus the *getSecond()* method is defined as returning a value from 0 to 59 and does not mention leap seconds. This achieves the goal of hiding leap seconds from most developers.

Note that this also affects durations of time. The duration from a time of 23:59 to 00:01 on the next day is always 2 minutes, or 120 seconds, irrespective of whether a leap second actually occurs.

There is a wrinkle in the approach however. The clock in Java (current time) is obtained from the operating system, and not all operating systems provide access to a suitable accurate leap second aware clock. The underlying current clock in Java, *System.currentTimeMillis()*, has not been changed as part of this work and will still return a value that make no guarantees about how leap seconds are handed. As such, despite the definition of a Java time-scale, the next release of Java will not actually implement the time-scale! It is effectively an idealized version of reality.

The API does, however, allow developers to implement their own clock. Were a developer to implement a clock in Java that directly communicated with an accurate clock, such as NTP or GPS, then the Java time-scale would be used to determine how the accurate clock should be mapped to fit the API.

BEYOND JAVA

The Java time-scale approach is applicable beyond Java. It allows clocks used by humans to tick in a different way to clocks used by time-geeks and low-level systems. To be a widely applicable approach, two things would have to happen. Firstly, UTC-SLS, or similar, would have to be an accepted international standard, rather than a draft. Secondly, leap seconds would need to be announced sufficiently far in advance, so as to be always handled correctly by servers and applications.

This approach appears to be an effective solution. It ensures that the day continues to be regulated by the rising and setting of the sun, as expected by most humans, while atomic clocks can still be used to effectively manage intra-day time.

CONCLUSION

When designing a date and time API for business application developers, there are many elements to consider. Leap seconds are the least important of these.

The JSR-310 API, after trying a number of alternate approaches, decided to create a Java time-scale that smoothes away the leap-second based on the UTC-SLS proto-standard. This allows the specification to "support" leap seconds without exposing developers to them.

ACKNOWLEDGMENTS

The JSR-310 specification has been defined by an expert group working publicly, whose input is gratefully acknowledged.[*] As required by the JSR process, I note that the specification is still officially in draft form and may be subject to change or even rejection.

[*] http://threeten.github.io/