Java Globalization

JSR-310 International Calendar Support Use Cases

This wiki captures use cases required to support different calendars. This is a working document subject to change to reflect the 310 project updates. The purpose is to help ensure meeting i18n requirements. It is not intended to serve as the document that defines the i18n requirements or specify the functionalities. It is intended as a way to provide evaluation feedback on the API with anticipated usage scenarios.

References

JSR-310 API Javadoc:

http://spiro.us.oracle.com/~rriggs/threeten-pirate/ (The Pirate branch; holds the latest code base)

http://spiro.us.oracle.com/~rriggs/threeten-common-javadoc/index.html
http://spiro.us.oracle.com/%7Erriggs/radical/

Observation

As of 4/25/2012, the latest re-designed Javadoc API doesn't have several calendar neutral APIs available in previous Javadoc APIs mentioned in References section above. For more details, please see the evaluation of individual use cases.

Contents [hide]

Allow a single application code path to support different calendars

The API should allow applications to write calendar neutral code. The same code path should work with all calendars unless it is performing an operation that is only available with a specific calendar.

Evaluation: As of the snapshot on 16-April-2012 03:33 on the Pirate 0.7.0-alpha branch, it appears that this requirement will be met for the most part by the Chrono and ChronoDate classes.

Previously we saw many methods defined on specific Chronology or Date classes. As of this writing, we only see methods on a generic type such as ChronoDate. This is favorable from an i18n point of view. See resolved issues R1 section at the bottom.

Identify local calendars using CLDR calendar identifiers

Applications should be able to identify the calendars using the standard calendar names defined in Unicode CLDR.

Unicode CLDR:
http://cldr.unicode.org/

CLDR calendar name definitions. (source: calendar.xml in http://www.unicode.org/Public/cldr/)

<?xml version="1.0" ?>
<ldmlBCP47>
<version number="$Revision: 6003 $"/>
<generation date="$Date: 2011-07-11 19:00:53 -0500 (Mon, 11 Jul 2011)
<keyword>
<key name="ca" alias="calendar" description="Calendar algorithm key">
<type name="buddhist" description="Thai Buddhist calendar"/>
<type name="chinese" description="Traditional Chinese calendar"/>
<type name="coptic" description="Coptic calendar"/>
<type name="ethioaa" alias="ethiopic-amete-alem" description="Ethiopic calendar, Amete Alem (epoch approx. 5493 B.C.E)"/>
<type name="ethiopic" description="Ethiopic calendar, Amete Mihret (epoch approx, 8 C.E.)"/>
<type name="gregory" alias="gregorian" description="Gregorian calendar"/>
<type name="hebrew" description="Traditional Hebrew calendar"/>

<type name="indian" description="Indian calendar"/>
<type name="islamic" description="Astronomical Arabic calendar"/>
<type name="islamicc" alias="islamic-civil" description="Civil (algorithmic) Arabic calendar"/>
<type name="iso8601" description="ISO calendar (Gregorian calendar using the ISO 8601 calendar week rules)" since="2.0"/>
<type name="japanese" description="Japanese Imperial calendar"/>
<type name="persian" description="Persian calendar"/>
<type name="roc" description="Republic of China calendar"/>
</key>
</keyword>
</ldmlBCP47>

Evaluation: This requirement does not appear to be satisified as of Pirate 0.7.0-alpha 16-April-2012 03:33. This is because no factory method is found for Chrono and ChronoDate. When creating a calendar dependent type instance, the calendar must be identified. The standard calendar names should be used for the predefined calendars that ship with JDK. The mechanism to locate a user defined calendar should be defined specifically.

Background: Supporting CLDR as source of locale support definitions is a general JDK8 requirement. Because JSR 310 is introduced in JDK8, its calendar names should be based on CLDR. See the next two topics for a few applicable use cases on the API.

These standard names could be specified as part of the common locale tag through the BCP47 Unicode extension mechanism. (RFC 6067 http://www.ietf.org/rfc/rfc6067.txt) Java has been supporting BCP47 locales since JDK7. i.e. Java application would no longer need to override the regular locale codes for a specific calendar. This has caused problems, for example, to Thai locale users whose codes are th-TH, for which JRE creates a Thai calendar. Example bug 13044013 - NLS: THAI LOCALE: START DATE CONDITION IS WRONG WHEN IT IS CREATED IN TH_TH

List of available local calendars

Applications should be able to discover what calendars are available in the system. This is to provide a choice of local calendar. It could be as simple as what's shown in the pseudo code fragments below. Having this method is consistent with similar i18n support classes from the historical JDK releases. For instance, Locale provides availableLocales() method to list supported locales. Once supported calendars are listed, the Chrono.of() method may then be used with a specific calendar passed in to the parameter.

Pseudo code 1 --- creating a Chrono instance :

```
Set<String> calendars;

//list available Chronologies
calendars = Chrono.getAvailableChronologies();

/* User or admin makes a choice: see a separate use case: "Discovering local calendars for a locale" */

//create the calendar

c = Chrono.of(calendar);
```

Pseudo code 2 --- validating a calendar specified by an external input :

```
Set<String> calendars;

// list available Chronologies
calendars = Chrono.getAvailableChronologies();
```

```
// validate if a given calendar is supported
if ( !calendars.contains(calendar) )
throw new UnsupportedCalendarException();
```

Evaluation: As of Pirate 0.7.0-alpha 16-April-2012 03:33, the availableChronologies() method is not found. However, it was once seen in a javadoc snapshot we reviewed previously. We expect it to see it back.

List eras of a local calendar

This is to enable the applications to list the local calendar eras in a given duration. In Gregorian calendar, the era is virtually always CE. However, with non-Gregorian calendars it is often necessary to identify the era as well as date, month and year.

Evaluation: This requirement does not seem to be satisified as of Pirate 0.7.0-alpha 25-April-2012 10:10. To satisfy this requirement, please consider providing a way to list eras for a given period. As an example, pseudo code is provided below. Please notice that this assumes the Era class implements the Iterator interface to make it easy to visit each era in turn. It could be also an option to provide a utility method to return the list, given a specific period.

Pseudo code --- Listing eras for a given duration to construct the LOV to enter DOB :

A date picker may provide choices of era depending on the application context. For instance, when entering a date of birth with Japanese Imperial calendar, one of the eras need to be selected. In this example, we consider how the list of eras could be created.

```
Chrono chrono = Chrono.of ("japanese"); // a chronology. In practice this is rarely hardcoded

LocalDate startDate = LocalDate.now().minusYears(120); // find a date that is approximately 120 years ago

LocalDate endDate = LocalDate.now(); // find today's date

ChronoDate startChronoDate = chrono.from(startDate); // find the past date in the local calendar

ChronoDate endChronoDate = chrono.from(endDate); // find the current date in the local calendar

Era oldestEra = startChronoDate.getEra(); // get the era in the past

Era currentEra = endChronoDate.getEra(); // get the current era

SortedSet<Era> eras = new SortedSet<Era>();

Era era = oldestEra;

while (true) {

eras.add(era);

if ( !era.hasNext() || era.equals(currentEra) )

break;

era = era.next();

}

/* at this point eras has all the eras in the specified 120 year period */
```

Construct a specific local calendar using a calendar identifier

Applications should be able to create a calendar instance using a CLDR calendar name.

Evaluation: This requirement does not appear to be satisfied as of Pirate 0.7.0-alpha 16-April-2012 03:33. To satisfy this requirement, please consider providing the following way to create a new calendar or any equivalent. It should be also possible to create a ChronoDate from an identifier or an instance of Chrono.

Pseudo code --- creating a Chrono from a calendar identifier :

String calendar_id = "islamic"; //*defined by the application, coming from a user profile, etc.*

javax.time.chrono.Chrono c;

// create a Chrono instance from a CLDR calendar name

c = Chrono.of(calendar_id);

Conversion between java.util.Date and JSR-310 instant type

Applications that use java.util.Date should be able to convert it into JSR-310 equivalent which is Instant.

Pseudo code:

To convert java.util.Date into JSR-310 Instant type:

Date currentDate = new Date();
Instant currentInstant = Instant.ofEpochMilli(currentDate.getTime());

To convert JSR-310 Instant into java.util.Date:

Date currentDate = new Date(Instant.toEpochMilli());

Evaluation: This requirement is satisfied as shown in the above code, as of Pirate 0.7.0-alpha 16-April-2012 03:33.

Conversion between java.util.Calendar and JSR-310 calendar types

The purpose is to ease the migration in adopting 310, by facilitating the type conversion required at the boundary where the tradition JDK Calendar type is used on one side and new 310 type is used on the other. This allows an application to evolve from traditional calendar implementation into fully 310 based. With this type conversion, it will be easier to adopt 310 because 310 based components and traditional components can coexist and get along. Applications should be able to convert java.util.Calendar instance into JSR-310 equivalents and vise versa.

Sample code fragments below:

From JDK java.util.Calendar to 310:

java.util.Calendar cal = Calendar.getInstance(TimeZone.getTimeZone("UTC"),Locale.US); // Returns a timezone neutral Gregorian Calendar instance
int year = cal.get(Calendar.YEAR); // Returns a field of the calendar (some operation)
int month= cal.get(Calendar.MONTH); // Returns a field of the calendar (some operation)
int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH); // Returns a field of the calendar (some operation)
cal.set(Calendar.YEAR, 1); // Sets a calendar field (some operation)

cal.set(Calendar.MONTH, 2); // Sets a calendar field (some operation)
cal.set(Calendar.DAY_OF_MONTH 3); // Sets a calendar field (some operation)
javax.time.LocalDate cal310 = LocalDate.of(year,month,dayOfMonth); // Returns a converted LocalDate instance for the JDK calendar
javax.time.chrono.ChronoDate cd = ChronoDate.from(cal310); // Returns an ISO ChronoDate that corresponds to the LocalDate instance

From JSR-310 to java.util.Calendar:

ChronoDate cal310 = ChronoDate.now(zoneid,"gregorian") // Returns a Gregorian calendar date instance
cal310 = cal310 . withYearOfEra (1); // Returns a copy with a specific field (some operation)
cal310 = cal310 .withMonthOfYear (2); // Returns a copy with a specific field (some operation)
cal310 = cal310 . DayOf Month (3); // Returns a copy with a specific field (some operation)
int year = cal310 . get YearOfEra (); // Returns a calendar field (some operation)
int month= cal310 . get MonthOfYear (); // Returns a calendar field (some operation)
int dayOfMonth = cal310 . getDayOf Month (); // Returns a calendar field (some operation)
java.util.Calendar cal = new java.util.GregorianCalendar(TimeZone.getTimeZone("UTC"),Locale.US); // Create a timezone neutral Gregorian Calendar instance, equivalent to Calendar.getInstance() above
cal.set(year,month,dayOfMonth); // now the value of cal is equivalent to that of cal310.

Evaluation:

As of Pirate 0.7.0-alpha 16-April-2012 03:33, it appears applications will be able to perform conversions between traditional JDK and JSR-310 calendar types for the most part, however, this cannot be confirmed thoroughly due to the project status. As the project stabilizes the API with well-defined types and their factory methods, it would be helpful to re-evaluate the relationships of old and new types.

Conversion between Chronology specific date and Local (ISO) date

Applications require to convert Chronology specific date ChronoDate to ISODate (local date).

Pseudo-code:

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
LocalDate isoDate1 = LocalDate.now(Clock.withZone(zone)); // returns the current ISO date
ChronoDate date = chrono.from(isoDate1); // returns the corresponding non-ISO date
LocalDate isoDate2 = date.toLocalDate(); // returns ISO representation of the Chrono Date, which should equal isoDate1.

Evaluation: As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc, this requirement is satisfied via toLocalDate() method in ChronoDate.

Determine last day of the month in a calendar neutral way

When date widgets are built, one of the key requirements is to determine the last day of the month. So the single code path using a single API should return the required last day of the month in a calendar neutral way.

Using existing JDK, the calendar (GregorianCalendar) specific last day of the month can be retrieved using Calendar's getActualMaximum() method.

Calendar cal = Calendar.getInstance();
int lastDay = cal.getActualMaximum(Calendar.DAY_OF_MONTH);

Pseudo-code :

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
LocalDate currentDate = LocalDate.now(Clock.withZone(zone)); // using the current date in this example
ChronoDate date = chrono.from(currentDate); // returns the corresponding non-ISO dat e
date = date.with(DateAdjusters.lastDayOfMonth()); // find the last day of the current month
int n = date.getDayOfMonth(); // returns the day of month for the last day in whichever calendar in use.

/ For instance, on ISO date July 4th 2012 it is in the Islamic month Shaban 1433. The last day of Shaban 1433 is 29, thus n would be 29. */*

/ Similarly, to get the last day of July 2012, at the end of the following code fragments the value of n would be 31 because July has 31 days. */*

LocalDate lastDayOfMonth = currentDate.with(DateAdjusters.lastDayOfMonth()); // exact same call used on the non-ISO date above now attempted on the ISO date
n = lastDayOfMonth .getDayOfMonth(); // returns the day of month

Evaluation:

As of Pirate 0.7.0-alpha 16-April-2012 03:33, the Chronology specific classes such as ThaiChronology doesn't exist or the API might be incomplete. It's good to see the introduction of the ChronoDate class. However ChronoDate class lacks some of the methods provided on LocalDate. Adjusters and other key calendaring functionalities are expected to match what's available in LocalDate. As shown in the pseudo-code above, it is needed to be able to discover range of values for the calendar fields.

Equality/Comparison

Comparing dates in a calendar neutral way. It should be possible for the application to do comparisons based on the chronological order, in order to tell if two dates are on the same day or which date comes before or after.

Pseudo-code

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
LocalDate currentDate = LocalDate.now(Clock.withZone(zone)); // using the current date in this example
ChronoDate date1 = chrono.from(currentDate); // returns the corresponding non-ISO dat e
ChronoDate date2 = date1.minusWeeks(1); // returns a week ago
int compValue = date1.compareTo(date2); // returns a positive value because date 1 is later than date2

Evaluation:

As of Pirate 0.7.0-alpha 25-April-2012 10:10 Javadoc, with compareTo(ChronoDate) method, the date comparison requirement is satisfied. However, ChronoDate is implementing the Comparable interface only, which only supports the compareTo() method. It would seem more useful to implement the Comparator interface which also provide the equals() method.

Rolling operations on a specific date/time unit

Rolling operations such as adding/subtracting days, months & years in a calendar neutral way.

Pseudo-code :

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
LocalDate currentDate = LocalDate.now(Clock.withZone(zone)); // using the current date in this example
Chrono Date date1 = chrono.from(currentDate); // returns the corresponding non-ISO dat e
Chrono Date date2 = date1.plusDays(7) // returns newDate with 7 days added to date1

Chrono Date date3 = date2.plusWeeks(7) // returns newDate with 7 weeks added to date1
Chrono Date date4 = date3.plusMonths(7) // returns newDate with 7 months added to date1
Chrono Date date5 = date4.plusYears(7) // returns newDate with 7 years added to date1

Similarly there are minusDays(), minusWeeks(), minusMonths() & minusYears() are available to roll date/time units backwards.

Evaluation:

As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc, this requirement is satisfied.

Get a specific field value

Applications should be able to get a specific field such as day, month etc., on a given Date instance.

Pseudo-code:

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
ChronoDate date = chrono.date (year, month, day);
int year = date.getYearOfEra();
int month = date.getMonthOfYear();
int dayOfYear = date.getDayOfYear();
int dayOfWeek = date.getDayOfWeek() ;
DayOfWeek dayOfWeek = DayOfWeek.from(date);

Evaluation:

As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc, this requirement is satisfied.

Set a specific field value

Applications should be able to get a specific field such as day, month etc., on a given Date instance.

Pseudo-code:

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
ChronoDate date = chrono.date (year, month, day);
date = date .withDayOfYear(day);
date = date.withMonthOfYear (month);
date = date .withYearOfEra (year);

Evaluation:

As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc, this requirement is satisfied.

Identify a specific range of values for a specific date/time and unit

Applications should be able to determine a specific range of values such as minimum and maximum for a specific date/time in a calendar neutral way.

Pseudo-code #1:

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
DateTimeRule rule = chrono.dayOfMonthRule();
DateTimeRuleRange range = rule.getValueRange();
long lastDay = range.getMaximum(); // Gets the maximum value. For example, the ISO day-of-month runs to between 28 and 31 days. The maximum is therefore 31.

long firstDay = range.getMinimum(); // Gets the minimum value. For example, the ISO day-of-month always starts at 1. The minimum is therefore 1.
long largeMin = range.getLargestMinimum(); // Gets the largest possible minimum value. For example, the ISO day-of-month always starts at 1. The largest minimum is therefore 1.
long smallMax = range.getSmallestMaximum(); // Gets the smallest possible maximum value. For example, the ISO day-of-month runs to between 28 and 31 days. The smallest maximum is therefore 28.

Pseudo-code #2:

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
DateTimeRule rule = chrono.dayOfMonthRule();
DateTimeRuleRange range = rule.getValueRange();

Evaluation:

As of Pirate 0.7.0-alpha 16-April-2012 03:33 Javadoc, there are no APIs available to retrieve a range of values. Without getting an instance of DateTimeRuleRange, there is no way to determine a specific range of values.

Identify beginning of the week

Applications should be able to identify the beginning of the week in an internationalized way. The first day of the week is locale dependent which means for the same chronology it will vary based on the locale. For example, for Gregorian Calendar (ISOChrono) the first day of the week in US could be Sunday and in France it could be Monday.

Pseudo-code:

int firstDay = DayOfWeek.getFirstDayOfWeek(locale); // returns the first day of the week for the specified locale

Evaluation:

As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc, the DayOfWeek enum doesn't have an API that returns the first day of the week for a given locale.

Ordinal days, weeks of the year or month

Applications should be able to determine the ordinal days and weeks of a given year or month in a calendar neutral way.

Evaluation:

JDK Calendar API provides a constant DAY_OF_WEEK_IN_MONTH to determine the ordinal day of the month. As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc, there is no API available to determine the ordinal days or weeks of a given year or month.

Determine weekends for a given culture specific calendar

Several cultural calendars have weekends other than Saturday/Sunday. For example, in Islamic countries Friday/Saturday or Thursday/Friday are treated as weekends. When the date picker is presented in non-Gregorian calendar, it would be helpful to show/highlight weekends in respective calendars. Application developers usually don't have this culture/calendar specific information.

Pseudo-code:

for (DayOfWeek d : DayOfWeek.values());
boolean weekend = !DayOfWeek.isWeekDay(locale); // this method doesn't exist

Evaluation:

If there is an API that identifies a particular day of the week as weekend in that culture specific Chronology/Calendar, it would help application developers eliminate custom code/logic to determine the weekend. As shown in the pseudo code above having isWeekDay() method in the DayOfWeek Enum would be helpful.

Retrieving translated display names for calendar elements

Applications should be able to retrieve translated text for names of eras, years, months & days.

Pseudo-code:

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
ChronoDate date = chrono.date (year, month, day);
String day = date.getDayOfWeek() .getText (TextStyle.FULL, new Locale("ar"));
String month = date.getMonthOfYear().getText (TextStyle.FULL, new Locale("ar"));

Evaluation:

1) The getText() method disappeared from DayOfWeek & MonthOfYear enums in Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc. From the DayOfWeek or MonthOfYear enum, we can call getText() method to retrieve translated name of the day of the week.

2) As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc, no API is available to retrieve translated display names.

Getting duration between two dates

Applications should be able to get the duration (years, months, day, hours, minutes, seconds) between two dates.

Pseudo-code:

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
ChronoDate date1 = chrono.of (year, month, day); // some date
ChronoDate date2 = chrono.of (year, month, day); // some date
Period period = Period.between (date1, date2); // returns a Period consisting of the number of days, months and years between two dates

Evaluation:

As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc, Period.between (date1, date2) disappeared and there is no equivalent API available.

Date/Time based on time zone

A Java equivalent of SQL's timestamp with time zone. This is a date time value associated with a timezone.

Evaluation: This requirement is satisfied with the ZonedDateTime type that is present as of Pirate 0.7.0-alpha 24-April-2012 02:25.

Pseudo-code:

N/A see notes below

Note 1: It is our assumption that a zoned non-Gregorian datetime is not required. However, this may be desirable for encapsulating the otherwise required conversion. e.g. when building a date & time picker with non-Gregorian calendar support.

Note 2: We will revisit this point when the 310 project has completed the expected modification to simplify the timezone support and see how this requirement may be satisfied.

Timezone independence

Timezone independence should be guaranteed where expected. Values of zoned and unzoned data types are on different timelines so evaluations and conversion between them should not confuse them. Unzoned values should be treated as timezone neutral/independent, while zoned values should be tied with a timezone.

Before JSR-310, one of the painful shortcomings was application developer's having to deal with timezones even if the datetime value is not timezone dependent. For instance, when printing a date on a business form typed as java.util.Date, the developer must apply the correct timezone when printing the value using java.text.SimpleDateFormat. If an incorrect timezone was applied, the printed date may be wrong --- one day ahead or behind due to the timezone conversion with an unexpected timezone used by the formatter in interpreting the Date value.

In JSR-310, application developers should not have to deal with timezones where timezone is logically irrelevant.

Evaluation:

This requirement appears to be satisfied through the stronger typing than the traditional JDK datetime classes, as in the strong contrast between LocalDateTime and ZonedDateTime. However, we notice some of the methods may lack the desired distinction between the zoned and unzoned timelines. There are three specific points of consideration discussed below:

Timezone dependency in LocalDate

For instance, LocalDate defines now() methods that return the current date.

now() ... Obtains the current date from the system clock in the default time-zone.

now(Clock clock) ... Obtains the current date from the specified clock.

Determining the current date is an act of taking the date value from the system clock, applying a specific timezone. Thus, it would be helpful to provide a way to create a LocalDate from the system clock and an arbitrary timezone. For example:

now(ZoneId zone) ... Obtains the current date from the specified timezone.

This is equivalent to now(Clock.withZone(zone)). So, the advantage of having this method may be limited, but its presence reminds the developer of the need to think about what timezone to use when determining the current date. For instance, let's consider a Web application that needs to determine the current date. The "current date" may be different for different users if they are from different timezones. Thus, the timezone independence of the resulting LocalDate value can be achieved more reliably if now(ZoneId) method was provided. Otherwise, an application is more likely to neglect applying the correct timezone, by simply using now() that applies the system default timezone, and incurrs unwanted timezone dependency in the LocalDate value. Failure to apply the correct timezone when creating a LocalDate could be considered as an application's design or programming bug if it goes wrong, however, the presence of LocalDate.now(ZoneId) would help avoid them from making the bug.

Comparison between zoned and unzoned types

Another point that could possibly exhibit a timezone dependence issue is comparison with values of different types. Let's look at LocalDateTime vs. ZonedDateTime and consider what should happen if the date and time portions have the same values. For instance, should the equals() return true or false.

// get the zoned current date in the default timezone ;

ZonedDateTime zdt = ZonedDateTime.now();

// get the unzoned current date from the zoned value; this chops off the timezone information

LocalDateTime ldt = ZonedDateTime.toLocalDateTime();

// now, compare the two values. should it be equal or not

boolean b = LocalDateTime.equals(zdt);

The equals() method is defined as "Checks if this date is equal to another date. The comparison is based on the time-line position of the dates." Based on this definition, it seems false is expected. This is because ZonedDateTime and LocalDateTime are on different time-lines, thus they cannot be equal. Alternatively, the equals() method could throw an exception for type mismatch. By enforcing explicit datatype conversion between zoned and unzoned types with a timezone specifically handled, JSR 310 would become easier to use.

Offset Date/DateTime/Time

UTC offset is generally of limited use in identifying a datetime value. JSR 310 should discount the fact that the ISO 8601 model defines UTC offset as a way to represent local time, because the time of an event can be recorded without UTC offset in most use cases. For instance, OffsetDateTime may be used when Instant can be a more suitable type, introducing an unwanted timezone dependency. When a specific local timezone does need to be identified, a real timezone ID must be used instead because UTC offset cannot. i.e. ZonedDateTime may be a more suitable type.

To help keep the class hierarchy compact and focus on classes in the areas of higher application demand, for overall good of JSR-310 the offset classes could be dropped or simplified.

Deviation Options

As an instance of Chrono is created, it should be possible to specify some deviations as needed. Examples include:

- Islamic/Hijri calendar deviations
- Japanese imperial era

The Islamic calendar has different forms with minor but significant variance in the way dates are identified. The requirement is to recognize the variant forms and accommodate the adjustments needed to create the exact form of calendar. This may be an option to specify when creating an Islamic calendar instance, as a list of leap years in 30 year cycles or a predefined ID that represents a specific list of leap years. See IslamicCalendar in Joda-Time .

The Japanese calendar enters a new era as the reign of a new emperor begins. Business forms can be dated using the Japanese calendar system and they need to adapt to the new era as soon as it starts. This is similar to the case when the daylight saving rule changes and the applications and the operating systems need updated timezone definitions. To allow applications to adapt to the new era instantly, it would be desirable to have them specify the new era through optional parameters on API and system properties or a similar configurable setting.

Pseudo-code:

The Japanese calendar could be extended to the possible new era in the following way.

Properties options = new Properties(); // java.util.Properties or any generic form to describe optional details of a calendar

options.setProperty(...); // specifies the new era: when the current(Heisei) era starts, what it is named, etc.

Chrono cal = Chrono.of("japanese",options); // creates the Japanese calendar extended with the new era

In addition to this optional parameter, a preferred approach is to enable the regular JapaneseChrono instance to respond to System properties, so applications will need no code change or specific engineering for prompt adaption to new era.

Evaluation:

As of Pirate 0.7.0-alpha 19-April-2012 12:07, most non-Gregorian calendars are not yet defined and this requirement cannot be confirmed.

It might be necessary to define how deviations are identified and managed.

Identify Bidi directionality attribute of the calendar

Applications should be able to identify the Bidi directionality attribute of the calendar to present it in a cultural specific way.

Evaluation:

As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc there is no API to determine the directionality of the Calendar. It would be appropriate to have isRTL() method or an equivalent in the Chrono class. For example, when applications construct the IslamicChrono instance they can get the directionality attribute and present it to the developer transparently. This attribute could be essential in drawing a calendar correctly in the expected directionality.

Max/Min dates and possible range information for the calendars

Applications may need to set the max & min dates per calendar. This needs to be done using a generic API so that calendar/chronology specific API is not used in the code.

Pseudo-code:

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
ChronoDate minDate = chrono.of (1970, 01, 01);
ChronoDate maxDate = chrono.of (2040, 12, 31);
ChronoDateTime minDate = chrono.Min_Supported_DateTime; // this contant doesn't exist & also there is no ChronoDateTime class
ChronoDateTime maxDate = chrono.Max_Supported_DateTime; // this constant doesn't exist & also there is no ChronoDateTime class

Evaluation:

As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc, it is possible to create a date instance using "of" factory method, but there are no constants as shown in the pseudo code above. The following link shows how it's used in .NET.

http://msdn.microsoft.com/en-us/library/7y5k6h07.aspx

Two digit year cutoff handling

JSR 310 should provide a way for the application to control how an abbreviated 2 digit year is interpreted for parsing the year in a date.

This is an equivalent of Apache Trinidad's attribute getTwoDigitYearStart() provided in the RequestContext class. It is defined as follows:

Returns the year offset for parsing years with only two digits. If not set this is defaulted to 1950 This is used by @link{org.apache.myfaces.trinidad.faces.view.converter.DateTimeConverter} while converting strings to Date object.

This implies the value is a request context wide attribute, however, it is known that a finer granularity is sometimes required, so each instance of the date parser can have different values. In an insurance application, there can be different dates to enter, such

as policy terms and age of the insured person/object and it cannot apply the same base year. For JSR 310, each instance of a date parser should be able to have its own value.

Pseudo-code:

N/A. set a PeriodRange for the year to the DateTimeRule used by DateTimeParser? how??

Evaluation:

N/A as of this writing.

Number of days from the epoch

Pseudo-code:

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
ChronoDate date = chrono.of (year, month, day); // some date
LocalDate isoDate = date.toLocalDate(); // returns ISO representation of the Chrono Date
long epochDays = localDate.toEpochDay() // returns the number of days from the Epoch (1970-01-01)

Evaluation:

As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc, this requirement is satisfied with toEpochDay() of LocalDate class.

General calendrical formatting

Ability to print a calendrical value in a calendar neutral way

Evaluation: As of JSR-310 Pirate 0.7.0-alpha 24-April-2012 02:25, this requirement appears to be satisfied. DateTimeFormatter.print(CalendricalObject calendrical) performs the printing. The formatter instance may be created in a calendar and locale neutral way.

Pseudo-code:

Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
ChronoDate date = chrono.of (year, month, day); // some date
DateTimeFormatter formatter = DateTimeFormatters.date(FormatStyle.SHORT, new Locale("ar"));
String result = formatter.print(date);

Calendrical formatting with timezone adjustment

Ability to apply an arbitrary timezone when printing a calendrical value. For example, printing date only, or date and time, when the value to be printed is a timestamp based on UTC or whichever different timezone.

Pseudo-code:

Adjusting timezone when presenting a datetime to the user

DateTimeFormatter formatter = DateTimeFormatters.dateTime(FormatStyle.SHORT, FormatStyle.SHORT, locale);
formatter = formatter.withTimezone(ZoneId);
String result = formatter.print(date);

Evaluation: As per Pirate 0.7.0-alpha 24-April-2012 02:25 javadoc, there is no method withTimezone(ZoneId) available in DateTimeFormatter class.

General calendrical parsing

Ability to parse a calendrical value in a calendar neutral way. Parsing behavior should be lenient and allow for locale dependent expectations. Details may be found here, LDML Lenient Parsing .

Evaluation: As of JSR-310 Pirate 0.7.0-alpha 24-April-2012 02:25,this requirement appears to be satisfied.

- DateTimeFormatter.parse(text,type) to perform the parsing.
- The formatter instance may be created in a calendar and locale neutral way.

Pseudo-code:

Parsing a date string to a ChronoDate

```
ChronoDate cd = /* ChronoDate<T> for the desired chronology, e.g. ChronoDate<IslamicChrono> */
C lass c = cd.getClass() ;
String text = /* user entry, e.g. "01-02-03" */ ;
DateTimeFormatter f = DateTimeFormatters.date(FormatStyle.SHORT, locale);
ChronoDate date = f.parse(text,c);
```

LDML format pattern syntax

This is to retain the datetime format pattern compliance with Unicode LDML. Java has been implementing the LDML syntax in SimpleDateFormat. JSR 310 should keep the syntax and any extension should also comply with LDML. For instance, the standalone month name is to be represented with an 'L' in the pattern string, which is being introduced in JDK8 in the LDML syntax. (See independent case "Context dependent month names")

Evaluation: As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc, this requirement seems to be satisfied for the most part. However, a few nonconforming patterns are observed. Desirable to look closer for impact analysis and seeking resolutions.

Context dependent month names

Context dependent month names are the names used in formatting. They need to be supported when the dates are formatted as per a given locale.

Pseudo-code:

```
Chrono chrono = Chrono.of ("Islamic"); // a chronology. In practice this is rarely hardcoded
ChronoDate date = chrono.date (year, month, day);
// this is assuming getMonthOfYear() would return an instance of MonthOfYear
String month = date.getMonthOfYear().getText (TextStyle.LONG_FORMAT, new Locale("ar")); // Note: the constant
LONG_FORMAT doesn't exist
```

Evaluation:

As per Pirate 0.7.0-alpha 16-April-2012 03:33 javadoc, there is no API available to support context dependent month names. As per bug 7079560 , this is a requirement for JDK8.

Resolved Items

This section holds the items that have been resolved. We are experiencing loss of a once-resolved point from merges between code branches. To help restore them promptly, we keep the resolved points here rather than just deleting and having to dig up from the history or start over.

R1: Calendar Neutral API

Previously, different calendars had separate interfaces and it would have been hard to write the application with multiple calendar support in an internationalized way. Equivalent methods on different calendar classes. Solution was providing a generic calendar/date class.

Here is a list of classes that had this issue.

- CopticChronology
- CopticDate
- CopticStandardChronology
- HijrahChronology
- HijrahDate
- HistoricChronology
- HistoricDate
- JapaneseChronology
- JapaneseDate
- JulianChronology
- MinguoChronology
- MinguoDate
- ThaiBuddhistChronology
- ThaiBuddhistDate

For instance, HijrahChronology has the following methods defined. However, none of these operations are unique to Hijrah calendar. These methods should be defined in a parent class or an interface instead so an application will not have to write the code to the specific calendars individually.

- dayOfMonthRule()
- dayOfWeekRule()
- dayOfYearRule()
- eraRule()
- getName()
- monthOfYearRule()
- periodDays()
- periodEras()
- periodMonths()
- periodWeeks()
- periodYears()
- yearOfEraRule()

R2: From JDK/310 calendar conversion

This has been resolved, not outstanding on Pirate 0.7.0-alpha 16-April-2012 03:33.

LocalDate date = isoCalendar.of (year, month, day); // there is no way to query year, month and day as in Calendar.DAY_OF_YEAR etc.,

Evaluation:

1) There is no way to create an instance in a calendar neutral way. The class LocalDate in the above code is essentially referring to ISODate. So renaming it to ISODate would be appropriate.

2) There is no API available to convert a java.util.Calendar instance into JSR-310 Chronologies. It would be better to have something like this:

Chronology myChronology = Chronology.of (java.util.Calendar calendar, java.lang.String myChronology);
LocalDate date = myChronology.of (myChronology.getYear(), myChronology.getMonth(); myChronology.getDay());

or

LocalDate date = Chronology.with(Chronology myChronology); // this will provide direct access to the LocalDate instance

Items on Hold

Discovering local calendars for a locale

This is a helper utility to help applications discover what calendars may be used for a given locale. In the application contexts, there can be a variety of ways to create select the desired calendar system. There is always a known set of common calendars in each locale. For instance, in Thai, either Thai or Gregorian calendars are common but the others are rarely used. Thus, the application may be able to achieve better usability if it gave the two common calendars, Gregorian and Thai, to a Thai user as the choice of preferred calendar. To enable applications to tap in the knowledge of what calendars are common for each locale, it could have the following API signature in the pseudo code below on availableChronologies(). The Chrono.of() method is to look up the common local calendars and create an instance.

Having this method is consistent with similar i18n support classes from the historical JDK releases and preferred in terms of ease of i18n. For instance, Collator provides getInstance() method to get a local collator for the given locale. Application developers' defining their own locale to calendar mappings is unfavorable in terms of the implementation and maintenance cost. Alternatively, the mapping information could be provided through a separate API, however, there is no such API, and i18n support is easier to use when the support is transparent to developers.

Pseudo code --- creating a local Chrono instance from a given locale:

Locale locale = Locale.US; // defined by the application, coming from a user profile, etc.

// Get available chronologies for the locale

List<String> list = Chrono.availableChronologies(locale);

//create a Chronology instance for the most common local calendar for the given locale, if available

if (list.size() > 1) {
c = Chrono.of(list.get(1));

} else {

c = null; // no local calendar available for the locale

}

Notice the specified index is 1 in this case. The Gregorian/ISO calendar may be always at index 0. i.e. the returned list always has one or more elements, and the list is never null, even if the locale is not supported. The application may decide to show the list of available calendars only if a non-Gregorian calendar was available for the locale.

Note 1: The behavior should be consistent with the definitions in CLDR supported by the Java release. For this particular item, the expected chronologies to be returned are described in LDML Appendix C.15 Calendar Preference Data.

Note 2: The given locale may have a specific calendar specified. (See "Special cases" in JDK Locale javadoc for how this may be done.) If a specific calendar is indicated by the locale object, then it may have

to override the default local calendar derived from the locale. To accommodate this requirement, the application should check the presence of the calendar parameter in the locale and not use this method if the locale parameter has an explicitly specified calendar. The availableChronologies() method is insensitive to the calendar parameter in the locale.

For those locales that have no common non-Gregorian calendar and for unsupported locales, the ISO/Gregorian calendar is the only item in the returned list.Some locales have multiple non-Gregorian calendars that may be in use.

Evaluation: As of Pirate 0.7.0-alpha 16-April-2012 03:33, the availableChronologies() method is not found. However, it was once seen in a javadoc snapshot we reviewed previously. We expect it to see the method merged to the Pirate branch.

