Standard Calendar

The core of the Date-Time API is the java.time package. The classes defined in java.time base their calendar system on the ISO calendar, which is the world standard for representing date and time. The ISO calendar follows the proleptic Gregorian rules. The Gregorian calendar was introduced in 1582; in the proleptic Gregorian calendar, dates are extended backwards from that time to create a consistent, unified timeline and to simplify date calculations.

Overview

There are two basic ways to represent time. One way represents time in human terms, referred to as human time, such as year, month, day, hour, minute and second. The other way, machine time, measures time continuously along a timeline from an origin, called the epoch, in nanosecond resolution. The Date-Time package provides a rich array of classes for representing date and time. Some classes in the Date-Time API are intended to represent machine time, and others are more suited to representing human time.

First determine what aspects of date and time you require, and then select the class, or classes, that fulfill those needs. When choosing a temporal-based class, you first decide whether you need to represent human time or machine time. You then identify what aspects of time you need to represent. Do you need a time zone? Date and time? Date only? If you need a date, do you need month, day, and year, or a subset?

Terminology

The classes in the Date-Time API that capture and work with date or time values, such as Instant, LocalDateTime, and ZonedDateTime, are referred to as temporal-based classes (or types) throughout this tutorial. Supporting types, such as the TemporalAdjuster interface or the DayOfWeek enum, are not included in this definition.

Example

You might use a LocalDate object to represent a birth date, because most people observe their birthday on the same day, whether they are in their birth city or across the globe on the other side of the international date line. If you are tracking astrological time, then you might want to use a LocalDateTime object to represent the date and time of birth, or a ZonedDateTime, which also includes the time zone. If you are creating a time-stamp, then you will most likely want to use an Instant, which allows you to compare one instantaneous point on the timeline to another.

Table summarizes the temporal-based classes in the java.time package that store date and/or time information, or that can be used to measure an amount of time. A check mark in a column indicates that the class uses that particular type of data and the toString Output column shows an instance printed using the toStringmethod. The Where Discussed column links you to the relevant page in the tutorial.

Class or Enum	Year	Month	Day	Hours	Minutes	Seconds*	Zone Offset	Zone ID	toString Output	Where Discussed
Instant						✓			2013-08-20T15:16:26.355Z	Instant Class
LocalDate	✓	✓	✓						2013-08-20	Date Classes
LocalDateTime	✓	✓	✓	✓	✓	✓			2013-08-20T08:16:26.937	Date and Time Classes
ZonedDateTime	✓	✓	✓	✓	✓	✓	✓	✓	2013-08-21T00:16:26.941+09:00[Asia/Tokyo]	Time Zone and Offset Classes
LocalTime				✓	✓	✓			08:16:26.943	Date and Time Classes
MonthDay		✓	✓						--08-20	Date Classes
Year	✓								2013	Date Classes
YearMonth	✓	✓							2013-08	Date Classes
Month		✓							AUGUST	DayOfWeek and Month Enums
OffsetDateTime	✓	✓	✓	✓	✓	✓	✓		2013-08-20T08:16:26.954-07:00	Time Zone and Offset Classes
OffsetTime				✓	✓	✓	✓		08:16:26.957-07:00	Time Zone and Offset Classes
Duration			**	**	**	✓			PT20H (20 hours)	Period and Duration
Period	✓	✓	✓				***	***	P10D (10 days)	Period and Duration

*Seconds are captured to nanosecond precision.
**This class does not store this information, but has methods to provide time in these units.
***When a Period is added to a ZonedDateTime, daylight saving time or other local time differences are observed.

Standard Calendar

DayOfWeek & Month Enums

The Date-Time API provides enums for specifying days of the week and months of the year.

DayOfWeek

The DayOfWeek enum consists of 7 constants that describe the days of the week: MONDAY through SUNDAY. The integer values of the DayOfWeek constants range from 1 (Monday) through 7 (Sunday). Using the defined constants (DayOfWeek.FRIDAY) makes your code more readable.

This enum also provides a number of methods, similar to the methods provided by the temporal-based classes Eg: the following code adds 3 days to "Monday" and prints the result. The output is "THURSDAY":

```
System.out.printf("%s%n", DayOfWeek.MONDAY.plus(3));
```

By using the getDisplayName(TextStyle, Locale) method, you can retrieve a string to identify the day of the week in the user's locale. The TextStyle enum enables you to specify what sort of string you want to display: FULL, NARROW (typically a single letter), or SHORT (an abbreviation). The STANDALONE TextStyle constants are used in some languages where the output is different when used as part of a date than when it is used by itself. The following example prints the three primary forms of the TextStyle for "Monday":

```
DayOfWeek dow = DayOfWeek.MONDAY;
Locale locale = Locale.getDefault();
System.out.println(dow.getDisplayName(TextStyle.FULL, locale));
System.out.println(dow.getDisplayName(TextStyle.NARROW, locale));
System.out.println(dow.getDisplayName(TextStyle.SHORT, locale));
```

This code has the following output for the en locale:

```
Monday
M
Mon
```

Month

The Month enum includes constants for the twelve months, JANUARY through DECEMBER. As with the DayOfWeek enum, the Month enum is strongly typed, and the integer value of each constant corresponds to the ISO range from 1 (January) through 12 (December). Using the defined constants (Month.SEPTEMBER) makes your code more readable.

The Month enum also includes a number of methods. The following line of code uses the maxLength method to print the maximum possible number of days in the month of February. The output is "29":

```
System.out.printf("%d%n", Month.FEBRUARY.maxLength());
```

The Month enum also implements the getDisplayName(TextStyle, Locale) method to retrieve a string to identify the month in the user's locale using the specified TextStyle. If a particular TextStyle is not defined, then a string representing the numeric value of the constant is returned. The following code prints the month of August using the three primary text styles:

```
Month month = Month.AUGUST;
Locale locale = Locale.getDefault();
System.out.println(month.getDisplayName(TextStyle.FULL, locale));
System.out.println(month.getDisplayName(TextStyle.NARROW,
locale));
System.out.println(month.getDisplayName(TextStyle.SHORT, locale));
```

This code has the following output for the en locale:

```
August
A
Aug
```

Standard Calendar

Date Classes

The Date-Time API provides four classes that deal exclusively with date information, without respect to time or time zone.
The use of these classes are suggested by the class names:`LocalDate, YearMonth, MonthDay, and Year.`

LocalDate

A LocalDate represents a year-month-day in the ISO calendar and is useful for representing a date without a time. You might use a LocalDate to track a significant event, such as a birth date or wedding date.
The following examples use the of and with methods to create instances of LocalDate:

```
LocalDate date = LocalDate.of(2000, Month.NOVEMBER, 20);
LocalDate nextWed = date.with(TemporalAdjusters.next(DayOfWeek.WEDNESDAY));
```

In addition to the usual methods, the LocalDate class offers getter methods for obtaining information about a given date. The getDayOfWeek method returns the day of the week that a particular date falls on. For example, the following line of code returns "MONDAY":

```
DayOfWeek dotw = LocalDate.of(2012, Month.JULY, 9).getDayOfWeek();
```

The following example uses a TemporalAdjuster to retrieve the first Wednesday after a specific date.

```
LocalDate date = LocalDate.of(2000, Month.NOVEMBER, 20);
TemporalAdjuster adj = TemporalAdjusters.next(DayOfWeek.WEDNESDAY);
LocalDate nextWed = date.with(adj);
System.out.printf("For the date of %s, the next Wednesday is %s.%n",
                  date, nextWed);
```

Running the code produces the following: `For the date of 2000-11-20, the next Wednesday is 2000-11-22.`

YearMonth

The YearMonth class represents the month of a specific year. The following example uses the YearMonth.lengthOfMonth() method to determine the number of days for several year and month combinations.

```
YearMonth date = YearMonth.now();
System.out.printf("%s: %d%n", date, date.lengthOfMonth());

YearMonth date2 = YearMonth.of(2010, Month.FEBRUARY);
System.out.printf("%s: %d%n", date2, date2.lengthOfMonth());

YearMonth date3 = YearMonth.of(2012, Month.FEBRUARY);
System.out.printf("%s: %d%n", date3, date3.lengthOfMonth());
```

The output from this code looks like the following:

```
2013-06: 30
2010-02: 28
2012-02: 29
```

Standard Calendar

MonthDay

The MonthDay class represents the day of a particular month, such as New Year's Day on January 1.

The following example uses the MonthDay.isValidYear method to determine if February 29 is valid for the year 2010. The call returns false, confirming that 2010 is not a leap year.

```
MonthDay date = MonthDay.of(Month.FEBRUARY, 29);
boolean validLeapYear = date.isValidYear(2010);
```

Year

The Year class represents a year. The following example uses the Year.isLeap method to determine if the given year is a leap year.
The call returns true, confirming that 2012 is a leap year.

```
boolean validLeapYear = Year.of(2012).isLeap();
```

Standard Calendar

Date and Time Classes

LocalTime

The LocalTime class is similar to the other classes whose names are prefixed with Local, but deals in time only. This class is useful for representing human-based time of day, such as movie times, or the opening and closing times of the local library. It could also be used to create a digital clock, as shown in the following example:

```
LocalTime thisSec;

for (;;) {
    thisSec = LocalTime.now();

    // implementation of display code is left to the reader
    display(thisSec.getHour(), thisSec.getMinute(), thisSec.getSecond());
}
```

The LocalTime class does not store time zone or daylight saving time information.

LocalDateTime

The class that handles both date and time, without a time zone, is LocalDateTime, one of the core classes of the Date-Time API. This class is used to represent date (month-day-year) together with time (hour-minute-second-nanosecond) and is, in effect, a combination of LocalDate with LocalTime. This class can be used to represent a specific event, such as the first race for the Louis Vuitton Cup Finals in the America's Cup Challenger Series, which began at 1:10 p.m. on August 17, 2013. Note that this means 1:10 p.m. in local time. To include a time zone, you must use a ZonedDateTime or an OffsetDateTime, as discussed in Time Zone and Offset Classes.

In addition to the *now* method that every temporal-based class provides, the *LocalDateTime* class has various of methods (or methods prefixed with of) that create an instance of *LocalDateTime*. There is a *from* method that converts an instance from another temporal format to a *LocalDateTime* instance. There are also methods for adding or subtracting hours, minutes, days, weeks, and months. The following example shows a few of these methods. The date-time expressions are in bold:

```
System.out.printf("now: %s%n", LocalDateTime.now());

System.out.printf("Apr 15, 1994 @ 11:30am: %s%n",
                  LocalDateTime.of(1994, Month.APRIL, 15, 11, 30));

System.out.printf("now (from Instant): %s%n",
                  LocalDateTime.ofInstant(Instant.now(), ZoneId.systemDefault()));

System.out.printf("6 months from now: %s%n",
                  LocalDateTime.now().plusMonths(6));

System.out.printf("6 months ago: %s%n",
                  LocalDateTime.now().minusMonths(6));
```

This code produces output that will look similar to the following:

```
now: 2013-07-24T17:13:59.985
Apr 15, 1994 @ 11:30am: 1994-04-15T11:30
now (from Instant): 2013-07-24T17:14:00.479
6 months from now: 2014-01-24T17:14:00.480
6 months ago: 2013-01-24T17:14:00.481
```

Standard Calendar

Time Zone & Offset Classes

A time zone is a region of the earth where the same standard time is used. Each time zone is described by an identifier and usually has the format region/city (Asia/Tokyo) and an offset from Greenwich/UTC time. For example, the offset for Tokyo is +09:00.

ZoneId and ZoneOffset

The Date-Time API provides two classes for specifying a time zone or an offset:

- ZoneId specifies a time zone identifier and provides rules for converting between an Instant and a LocalDateTime.
- ZoneOffset specifies a time zone offset from Greenwich/UTC time.

Offsets from Greenwich/UTC time are usually defined in whole hours, but there are exceptions. The following code, from the TimeZoneId example, prints a list of all time zones that use offsets from Greenwich/UTC that are not defined in whole hours.

This example prints the following list to standard out:

```
America/Caracas      -04:30
America/St_Johns     -02:30
   Asia/Calcutta     +05:30
    Asia/Colombo     +05:30
      Asia/Kabul     +04:30
  Asia/Kathmandu     +05:45
   Asia/Katmandu     +05:45
    Asia/Kolkata     +05:30
    Asia/Rangoon     +06:30
     Asia/Tehran     +04:30
Australia/Adelaide   +09:30
Australia/Broken_Hill +09:30
  Australia/Darwin   +09:30
   Australia/Eucla   +08:45
     Australia/LHI   +10:30
Australia/Lord_Howe  +10:30
   Australia/North   +09:30
   Australia/South   +09:30
Australia/Yancowinna +09:30
Canada/Newfoundland  -02:30
     Indian/Cocos    +06:30
             Iran    +04:30
           NZ-CHAT   +12:45
  Pacific/Chatham    +12:45
Pacific/Marquesas    -09:30
  Pacific/Norfolk    +11:30
```

```
Set<String> allZones = ZoneId.getAvailableZoneIds();
LocalDateTime dt = LocalDateTime.now();

// Create a List using the set of zones and sort it.
List<String> zoneList = new ArrayList<String>(allZones);
Collections.sort(zoneList);

...

for (String s : zoneList) {
    ZoneId zone = ZoneId.of(s);
    ZonedDateTime zdt = dt.atZone(zone);
    ZoneOffset offset = zdt.getOffset();
    int secondsOfHour = offset.getTotalSeconds() % (60 * 60);
    String out = String.format("%35s %10s%n", zone, offset);

    // Write only time zones that do not have a whole hour offset
    // to standard out.
    if (secondsOfHour != 0) {
        System.out.printf(out);
    }
    ...
}
```

The TimeZoneId example also prints a list of all time zone IDs to a file called timeZones.

Standard Calendar

The Date-Time Classes

The Date-Time API provides three temporal-based classes that work with time zones:

- **ZonedDateTime** handles a date and time with a corresponding time zone with a time zone offset from Greenwich/UTC.
- **OffsetDateTime** handles a date and time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.
- **OffsetTime** handles time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.

When would you use OffsetDateTime instead of ZonedDateTime? If you are writing complex software that models its own rules for date and time calculations based on geographic locations, or if you are storing time-stamps in a database that track only absolute offsets from Greenwich/UTC time, then you might want to use OffsetDateTime. Also, XML and other network formats define date-time transfer as OffsetDateTime or OffsetTime.

Although all three classes maintain an offset from Greenwich/UTC time, only ZonedDateTime uses the ZoneRules, part of the java.time.zone package, to determine how an offset varies for a particular time zone. For example, most time zones experience a gap (typically of 1 hour) when moving the clock forward to daylight saving time, and a time overlap when moving the clock back to standard time and the last hour before the transition is repeated. The ZonedDateTime class accommodates this scenario, whereas the OffsetDateTime and OffsetTime classes, which do not have access to the ZoneRules, do not.

ZonedDateTime

The ZonedDateTime class, in effect, combines the LocalDateTime class with the ZoneId class. It is used to represent a full date (year, month, day) and time (hour, minute, second, nanosecond) with a time zone (region/city, such as Europe/Paris).

The following code, from the Flight example, defines the departure time for a flight from San Francisco to Tokyo as a ZonedDateTime in the America/Los Angeles time zone. The withZoneSameInstant and plusMinutes methods are used to create an instance of ZonedDateTime that represents the projected arrival time in Tokyo, after the 10 hour 50 minute flight. The ZoneRules.isDaylightSavings method determines whether it is daylight saving time when the flight arrives in Tokyo.

A DateTimeFormatter object is used to format the ZonedDateTime instances for printing:

```
DateTimeFormatter format = DateTimeFormatter.ofPattern("MMM d yyyy  hh:mm a");

// Leaving from San Francisco on July 20, 2013, at 7:30 p.m.
LocalDateTime leaving = LocalDateTime.of(2013, Month.JULY, 20, 19, 30);
ZoneId leavingZone = ZoneId.of("America/Los_Angeles");
ZonedDateTime departure = ZonedDateTime.of(leaving, leavingZone);

try {
    String out1 = departure.format(format);
    System.out.printf("LEAVING:  %s (%s)%n", out1, leavingZone);
} catch (DateTimeException exc) {
    System.out.printf("%s can't be formatted!%n", departure);
    throw exc;
}

// Flight is 10 hours and 50 minutes, or 650 minutes
ZoneId arrivingZone = ZoneId.of("Asia/Tokyo");
ZonedDateTime arrival = departure.withZoneSameInstant(arrivingZone)
                                 .plusMinutes(650);

try {
    String out2 = arrival.format(format);
    System.out.printf("ARRIVING: %s (%s)%n", out2, arrivingZone);
} catch (DateTimeException exc) {
    System.out.printf("%s can't be formatted!%n", arrival);
    throw exc;
}

if (arrivingZone.getRules().isDaylightSavings(arrival.toInstant()))
    System.out.printf("  (%s daylight saving time will be in effect.)%n",
                      arrivingZone);
else
    System.out.printf("  (%s standard time will be in effect.)%n",
                      arrivingZone);
```

Output:

```
LEAVING:  Jul 20 2013  07:30 PM
          (America/Los_Angeles)
ARRIVING: Jul 21 2013  10:20 PM (Asia/Tokyo)
          (Asia/Tokyo standard time will be in effect.)
```

Standard Calendar

OffsetDateTime

The OffsetDateTime class, in effect, combines the LocalDateTime class with the ZoneOffset class. It is used to represent a full date (year, month, day) and time (hour, minute, second, nanosecond) with an offset from Greenwich/UTC time (+/-hours:minutes, such as +06:00 or -08:00).

```
// Find the last Thursday in July 2013.
LocalDateTime localDate = LocalDateTime.of(2013, Month.JULY, 20, 19, 30);
ZoneOffset offset = ZoneOffset.of("-08:00");

OffsetDateTime offsetDate = OffsetDateTime.of(localDate, offset);
OffsetDateTime lastThursday =

 offsetDate.with(TemporalAdjusters.lastInMonth(DayOfWeek.THURSDAY));
System.out.printf("The last Thursday in July 2013 is the %sth.%n",
                  lastThursday.getDayOfMonth());
```

The following example uses OffsetDateTime with the TemporalAdjuster.lastDay method to find the last Thursday in July 2013.

The output from running this code is:

```
The last Thursday in July 2013 is the 25th.
```

OffsetTime

The OffsetTime class, in effect, combines the LocalTime class with the ZoneOffset class. It is used to represent time (hour, minute, second, nanosecond) with an offset from Greenwich/UTC time

```
(+/-hours:minutes, such as +06:00 or -08:00)
```

The OffsetTime class is used in the same situations as the OffsetDateTime class, but when tracking the date is not needed.

Standard Calendar

Instant Class

One of the core classes of the Date-Time API is the Instant class, which represents the start of a nanosecond on the timeline. This class is useful for generating a time stamp to represent machine time.

```
import java.time.Instant;

Instant timestamp = Instant.now();
```

A value returned from the Instant class counts time beginning from the first second of January 1, 1970 (1970-01-01T00:00:00Z) also called the EPOCH. An instant that occurs before the epoch has a negative value, and an instant that occurs after the epoch has a positive value.
The other constants provided by the Instant class are MIN, representing the smallest possible (far past) instant, and MAX, representing the largest (far future) instant

Invoking toString on an Instant produces output like the following: `2013-05-30T23:38:23.085Z`

This format follows the ISO-8601 standard for representing date and time.
The Instant class provides a variety of methods for manipulating an Instant. There are plus and minus methods for adding or subtracting time. The following code adds 1 hour to the current time:

```
Instant oneHourLater = Instant.now().plus(1, ChronoUnit.HOURS);
```

There are methods for comparing instants, such as isAfter and isBefore. The until method returns how much time exists between two Instant objects. The following line of code reports how many seconds have occurred since the beginning of the Java epoch.

```
long secondsFromEpoch =
Instant.ofEpochSecond(0L).until(Instant.now(),
                               ChronoUnit.SECONDS);
```

The Instant class does not work with human units of time, such as years, months, or days. If you want to perform calculations in those units, you can convert an Instantto another class, such as LocalDateTime or ZonedDateTime, by binding the Instant with a time zone. You can then access the value in the desired units. The following code converts an Instant to a LocalDateTime object using the ofInstant method and the default time zone, and then prints out the date and time in a more readable form:

```
Instant timestamp;
...
LocalDateTime ldt = LocalDateTime.ofInstant(timestamp, ZoneId.systemDefault());
System.out.printf("%s %d %d at %d:%d%n", ldt.getMonth(), ldt.getDayOfMonth(),
                  ldt.getYear(), ldt.getHour(), ldt.getMinute());
```

The output will be similar to the following:

```
MAY 30 2013 at 18:21
```

Either a ZonedDateTime or an OffsetTimeZone object can be converted to an Instant object, as each maps to an exact moment on the timeline. However, the reverse is not true. To convert an Instant object to a ZonedDateTime or an OffsetDateTime object requires supplying time zone, or time zone offset, information.

Standard Calendar

Parsing & Formatting

The temporal-based classes in the Date-Time API provide parse methods for parsing a string that contains date and time information. These classes also provide formatmethods for formatting temporal-based objects for display. In both cases, the process is similar: you provide a pattern to the DateTimeFormatter to create a formatter object. This formatter is then passed to the parse or format method.

The DateTimeFormatter class provides numerous predefined formatters, or you can define your own.

The parse and the format methods throw an exception if a problem occurs during the conversion process. Therefore, your parse code should catch the DateTimeParseException error and your format code should catch the DateTimeException error. For more information on exception handing, see Catching and Handling Exceptions.

The DateTimeFormatter class is both immutable and thread-safe; it can (and should) be assigned to a static constant where appropriate.

Version Note: The java.time date-time objects can be used directly with java.util.Formatter and String.format by using the familiar pattern-based formatting that was used with the legacy java.util.Date and java.util.Calendar classes.

PARSING

The one-argument parse(CharSequence) method in the LocalDate class uses the ISO_LOCAL_DATE formatter. To specify a different formatter, you can use the two-argumentparse(CharSequence, DateTimeFormatter) method. The following example uses the predefined BASIC_ISO_DATE formatter, which uses the format 19590709 for July 9, 1959.

```
String in = ...;
LocalDate date =
LocalDate.parse(in,
DateTimeFormatter.BASIC_ISO_DATE);
```

You can also define a formatter using your own pattern. The following code, from the Parse example, creates a formatter that applies a format of "MMM d yyyy". This format specifies three characters to represent the month, one digit to represent day of the month, and four digits to represent the year. A formatter created using this pattern would recognize strings such as "Jan 3 2003" or "Mar 23 1994". However, to specify the format as "MMM dd yyyy", with two characters for day of the month, then you would have to always use two characters, padding with a zero for a one-digit date: "Jun 03 2003".

```
String input = ...;
try {
    DateTimeFormatter formatter =
                    DateTimeFormatter.ofPattern("MMM d yyyy");
    LocalDate date = LocalDate.parse(input, formatter);
    System.out.printf("%s%n", date);
}
catch (DateTimeParseException exc) {
    System.out.printf("%s is not parsable!%n", input);
    throw exc;      // Rethrow the exception.
}
// 'date' has been successfully parsed
```

The documentation for the DateTimeFormatter class specifies the full list of symbols that you can use to specify a pattern for formatting or parsing.

Standard Calendar

Formatting

The format(DateTimeFormatter) method converts a temporal-based object to a string representation using the specified format. The following code, from the Flightexample, converts an instance of ZonedDateTime using the format "MMM d yyy hh:mm a". The date is defined in the same manner as was used for the previous parsing example, but this pattern also includes the hour, minutes, and a.m. and p.m. components.

```
ZoneId leavingZone = ...;
ZonedDateTime departure = ...;

try {
    DateTimeFormatter format = DateTimeFormatter.ofPattern("MMM d yyyy  hh:mm a");
    String out = departure.format(format);
    System.out.printf("LEAVING:  %s (%s)%n", out, leavingZone);
}
catch (DateTimeException exc) {
    System.out.printf("%s can't be formatted!%n", departure);
    throw exc;
}
```

The output for this example, which prints both the arrival and departure time:

```
LEAVING:  Jul 20 2013  07:30 PM (America/Los_Angeles)
ARRIVING: Jul 21 2013  10:20 PM (Asia/Tokyo)
```

Standard Calendar

The Temporal Package

The java.time.temporal package provides a collection of interfaces, classes, and enums that support date and time code and, in particular, date and time calculations.

These interfaces are intended to be used at the lowest level. Typical application code should declare variables and parameters in terms of the concrete type, such as LocalDate or ZonedDateTime, and not in terms of the Temporal interface. This is exactly the same as declaring a variable of type String, and not of type CharSequence.

Temporal & TemporalAccessor

The Temporal interface provides a framework for accessing temporal-based objects, and is implemented by the temporal-based classes, such as Instant, LocalDateTime, and ZonedDateTime. This interface provides methods to add or subtract units of time, making time-based arithmetic easy and consistent across the various date and time classes. The TemporalAccessor interface provides a read-only version of the Temporal interface.

Both Temporal and TemporalAccessor objects are defined in terms of fields, as specified in the TemporalField interface. The ChronoField enum is a concrete implementation of the TemporalField interface and provides a rich set of defined constants, such as DAY_OF_WEEK, MINUTE_OF_HOUR, and MONTH_OF_YEAR.

The units for these fields are specified by the TemporalUnit interface. The ChronoUnit enum implements the TemporalUnit interface. The field ChronoField.DAY_OF_WEEK is a combination of ChronoUnit.DAYS and ChronoUnit.WEEKS. The ChronoField and ChronoUnit enums are discussed in the following sections.

The arithmetic-based methods in the Temporal interface require parameters defined in terms of TemporalAmount values. The Period and Duration classes (discussed in Period and Duration) implement the TemporalAmount interface.

ChronoField & IsoFields

The ChronoField enum, which implements the TemporalField interface, provides a rich set of constants for accessing date and time values. A few examples are CLOCK_HOUR_OF_DAY, NANO_OF_DAY, and DAY_OF_YEAR. This enum can be used to express conceptual aspects of time, such as the third week of the year, the 11th hour of the day, or the first Monday of the month. When you encounter a Temporal of unknown type, you can use the TemporalAccessor.isSupported(TemporalField) method to determine if the Temporal supports a particular field. The following line of code returns false, indicating that LocalDate does not support ChronoField.CLOCK_HOUR_OF_DAY:

```
boolean isSupported =
LocalDate.now().isSupported(ChronoField.CLOCK_HOUR_OF_DAY);
```

Additional fields, specific to the ISO-8601 calendar system, are defined in the IsoFields class. The following examples show how to obtain the value of a field using both ChronoField and IsoFields:

```
time.get(ChronoField.MILLI_OF_SECOND)
int qoy = date.get(IsoFields.QUARTER_OF_YEAR);
```

Two other classes define additional fields that may be useful, WeekFields and JulianFields.

Standard Calendar

ChronoUnit

The ChronoUnit enum implements the TemporalUnit interface, and provides a set of standard units based on date and time, from milliseconds to millennia. Note that not all ChronoUnit objects are supported by all classes. For example, the Instant class does not support ChronoUnit.MONTHS or ChronoUnit.YEARS. Classes in the Date-Time API contain the isSupported(TemporalUnit) method that can be used to verify whether a class supports a particular time unit. The following call to isSupported returns false, confirming that the Instant class does not support ChronoUnit.DAYS.

```
Instant instant = Instant.now();
boolean isSupported = instant.isSupported(ChronoUnit.DAYS);
```

Standard Calendar

Temporal Adjuster

The TemporalAdjuster interface, in the java.time.temporal package, provides methods that take a Temporal value and return an adjusted value. The adjusters can be used with any of the temporal-based types.
If an adjuster is used with a ZonedDateTime, then a new date is computed that preserves the original time and time zone values.

Predefined Adjusters

The TemporalAdjusters class (note the plural) provides a set of predefined adjusters for finding the first or last day of the month, the first or last day of the year, the last Wednesday of the month, or the first Tuesday after a specific date, to name a few examples. The predefined adjusters are defined as static methods and are designed to be used with the static import statement.

The following example uses several TemporalAdjusters methods, in conjunction with the with method defined in the temporal-based classes, to compute new dates based on the original date of 15 October 2000:

```
LocalDate date = LocalDate.of(2000, Month.OCTOBER, 15);
DayOfWeek dotw = date.getDayOfWeek();
System.out.printf("%s is on a %s%n", date, dotw);

System.out.printf("first day of Month: %s%n",
                  date.with(TemporalAdjusters.firstDayOfMonth()));
System.out.printf("first Monday of Month: %s%n",
                  date.with(TemporalAdjusters.firstInMonth(DayOfWeek.MONDAY)));
System.out.printf("last day of Month: %s%n",
                  date.with(TemporalAdjusters.lastDayOfMonth()));
System.out.printf("first day of next Month: %s%n",
                  date.with(TemporalAdjusters.firstDayOfNextMonth()));
System.out.printf("first day of next Year: %s%n",
                  date.with(TemporalAdjusters.firstDayOfNextYear()));
System.out.printf("first day of Year: %s%n",
                  date.with(TemporalAdjusters.firstDayOfYear()));
```

This produces the following output:

```
2000-10-15 is on a SUNDAY
first day of Month: 2000-10-01
first Monday of Month: 2000-10-02
last day of Month: 2000-10-31
first day of next Month: 2000-11-01
first day of next Year: 2001-01-01
first day of Year: 2000-01-01
```

Standard Calendar

Custom Adjusters

You can also create your own custom adjuster. To do this, you create a class that implements the TemporalAdjuster interface with a adjustInto(Temporal) method. ThePaydayAdjuster class from the NextPayday example is a custom adjuster. The PaydayAdjuster evaluates the passed-in date and returns the next payday, assuming that payday occurs twice a month: on the 15th, and again on the last day of the month. If the computed date occurs on a weekend, then the previous Friday is used. The current calendar year is assumed.

```
/**
 * The adjustInto method accepts a Temporal instance
 * and returns an adjusted LocalDate. If the passed in
 * parameter is not a LocalDate, then a DateTimeException is thrown.
 */
public Temporal adjustInto(Temporal input) {
    LocalDate date = LocalDate.from(input);
    int day;
    if (date.getDayOfMonth() < 15) {
        day = 15;
    } else {
        day = date.with(TemporalAdjusters.lastDayOfMonth()).getDayOfMonth();
    }
    date = date.withDayOfMonth(day);
    if (date.getDayOfWeek() == DayOfWeek.SATURDAY ||
        date.getDayOfWeek() == DayOfWeek.SUNDAY) {
        date = date.with(TemporalAdjusters.previous(DayOfWeek.FRIDAY));
    }

    return input.with(date);
}
```

The adjuster is invoked in the same manner as a predefined adjuster, using the with method. The following line of code is from the NextPayday example:

```
LocalDate nextPayday = date.with(new PaydayAdjuster());
```

In 2013, both June 15 and June 30 occur on the weekend. Running the NextPayday example with the respective dates of June 3 and June 18 (in 2013), gives the following results:

```
Given the date:  2013 Jun 3
the next payday: 2013 Jun 14

Given the date:  2013 Jun 18
the next payday: 2013 Jun 28
```

Standard Calendar

Temporal Query

A TemporalQuery can be used to retrieve information from a temporal-based object.

Predefined Queries

The TemporalQueries class (note the plural) provides several predefined queries, including methods that are useful when the application cannot identify the type of temporal-based object. As with the adjusters, the predefined queries are defined as static methods and are designed to be used with the static import statement.

The precision query, for example, returns the smallest ChronoUnit that can be returned by a particular temporal-based object. The following example uses the precision query on several types of temporal-based objects:

```
TemporalQuery<TemporalUnit> query = TemporalQueries.precision();
System.out.printf("LocalDate precision is %s%n",
                  LocalDate.now().query(query));
System.out.printf("LocalDateTime precision is %s%n",
                  LocalDateTime.now().query(query));
System.out.printf("Year precision is %s%n",
                  Year.now().query(query));
System.out.printf("YearMonth precision is %s%n",
                  YearMonth.now().query(query));
System.out.printf("Instant precision is %s%n",
                  Instant.now().query(query));
```

The output looks like the following:

```
LocalDate precision is Days
LocalDateTime precision is Nanos
Year precision is Years
YearMonth precision is Months
Instant precision is Nanos
```

Custom Queries

You can also create your own custom queries. One way to do this is to create a class that implements the TemporalQuery interface with the queryFrom(TemporalAccessor)method. The CheckDate example implements two custom queries. The first custom query can be found in the FamilyVacations class, which implements the TemporalQueryinterface. The queryFrom method compares the passed-in date against scheduled vacation dates and returns TRUE if it falls within those date ranges.

```
// Returns true if the passed-in date occurs during one of the
// family vacations. Because the query compares the month and day only,
// the check succeeds even if the Temporal types are not the same.
public Boolean queryFrom(TemporalAccessor date) {
    int month = date.get(ChronoField.MONTH_OF_YEAR);
    int day   = date.get(ChronoField.DAY_OF_MONTH);

    // Disneyland over Spring Break
    if ((month == Month.APRIL.getValue()) && ((day >= 3) && (day <= 8)))
        return Boolean.TRUE;

    // Smith family reunion on Lake Saugatuck
    if ((month == Month.AUGUST.getValue()) && ((day >= 8) && (day <= 14)))
        return Boolean.TRUE;

    return Boolean.FALSE;
}
```

Standard Calendar

The second custom query is implemented in the FamilyBirthdays class. This class provides an isFamilyBirthday method that compares the passed-in date against several birthdays and returns TRUE if there is a match.

```
// Returns true if the passed-in date is the same as one of the
// family birthdays. Because the query compares the month and day only,
// the check succeeds even if the Temporal types are not the same.
public static Boolean isFamilyBirthday(TemporalAccessor date) {
    int month = date.get(ChronoField.MONTH_OF_YEAR);
    int day   = date.get(ChronoField.DAY_OF_MONTH);

    // Angie's birthday is on April 3.
    if ((month == Month.APRIL.getValue()) && (day == 3))
        return Boolean.TRUE;

    // Sue's birthday is on June 18.
    if ((month == Month.JUNE.getValue()) && (day == 18))
        return Boolean.TRUE;

    // Joe's birthday is on May 29.
    if ((month == Month.MAY.getValue()) && (day == 29))
        return Boolean.TRUE;

    return Boolean.FALSE;
}
```

The FamilyBirthday class does not implement the TemporalQuery interface and can be used as part of a lambda expression. The following code, from the CheckDateexample, shows how to invoke both custom queries.

```
// Invoking the query without using a lambda expression.
Boolean isFamilyVacation = date.query(new FamilyVacations());

// Invoking the query using a lambda expression.
Boolean isFamilyBirthday = date.query(FamilyBirthdays::isFamilyBirthday);

if (isFamilyVacation.booleanValue() || isFamilyBirthday.booleanValue())
    System.out.printf("%s is an important date!%n", date);
else
    System.out.printf("%s is not an important date.%n", date);
```

Standard Calendar

Period & Duration

When you write code to specify an amount of time, use the class or method that best meets your needs: the Duration class, Period class, or the ChronoUnit.betweenmethod. A Duration measures an amount of time using time-based values (seconds, nanoseconds). A Period uses date-based values (years, months, days).

Note: A Duration of one day is exactly 24 hours long. A Period of one day, when added to a ZonedDateTime, may vary according to the time zone. For example, if it occurs on the first or last day of daylight saving time.

Duration

A Duration is most suitable in situations that measure machine-based time, such as code that uses an Instant object. A Duration object is measured in seconds or nanoseconds and does not use date-based constructs such as years, months, and days, though the class provides methods that convert to days, hours, and minutes. A Duration can have a negative value, if it is created with an end point that occurs before the start point.

The following code calculates, in nanoseconds, the duration between two instants:

```
Instant t1, t2;
...
long ns = Duration.between(t1, t2).toNanos();
```

The following code adds 10 seconds to an Instant:

```
Instant start;
...
Duration gap = Duration.ofSeconds(10);
Instant later = start.plus(gap);
```

A Duration is not connected to the timeline, in that it does not track time zones or daylight saving time. Adding a Duration equivalent to 1 day to a ZonedDateTime results in exactly 24 hours being added, regardless of daylight saving time or other time differences that might result.

ChronoUnit

The ChronoUnit enum, discussed in the The Temporal Package, defines the units used to measure time. The ChronoUnit.between method is useful when you want to measure an amount of time in a single unit of time only, such as days or seconds. The between method works with all temporal-based objects, but it returns the amount in a single unit only. The following code calculates the gap, in milliseconds, between two time-stamps:

```
import java.time.Instant;
import java.time.temporal.Temporal;
import java.time.temporal.ChronoUnit;

Instant previous, current, gap;
...
current = Instant.now();
if (previous != null) {
    gap = ChronoUnit.MILLIS.between(previous,current);
}
...
```

Standard Calendar

Period

To define an amount of time with date-based values (years, months, days), use the Period class. The Period class provides various get methods, such as getMonths,getDays, and getYears, so that you can extract the amount of time from the period.
The total period of time is represented by all three units together: months, days, and years. To present the amount of time measured in a single unit of time, such as days, you can use the ChronoUnit.between method.

The following code reports how old you are, assuming that you were born on January 1, 1960. The Period class is used to determine the time in years, months, and days. The same period, in total days, is determined by using the ChronoUnit.between method and is displayed in parentheses:

```
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1960, Month.JANUARY, 1);

Period p = Period.between(birthday, today);
long p2 = ChronoUnit.DAYS.between(birthday, today);
System.out.println("You are " + p.getYears() + " years, " + p.getMonths() +
                   " months, and " + p.getDays() +
                   " days old. (" + p2 + " days total)");
```

The code produces output similar to the following:

```
You are 53 years, 4 months, and 29 days old. (19508 days total)
```

To calculate how long it is until your next birthday, you could use the following code from the Birthday example. The Period class is used to determine the value in months and days. The ChronoUnit.between method returns the value in total days and is displayed in parentheses.

```
LocalDate birthday = LocalDate.of(1960, Month.JANUARY, 1);

LocalDate nextBDay = birthday.withYear(today.getYear());

//If your birthday has occurred this year already, add 1 to the year.
if (nextBDay.isBefore(today) || nextBDay.isEqual(today)) {
    nextBDay = nextBDay.plusYears(1);
}

Period p = Period.between(today, nextBDay);
long p2 = ChronoUnit.DAYS.between(today, nextBDay);
System.out.println("There are " + p.getMonths() + " months, and " +
                   p.getDays() + " days until your next birthday. (" +
                   p2 + " total)");
```

The code produces output similar to the following:

```
There are 7 months, and 2 days until your next birthday. (216 total)
```

These calculations do not account for time zone differences. If you were, for example, born in Australia, but currently live in Bangalore, this slightly affects the calculation of your exact age. In this situation, use a Period in conjunction with the ZonedDateTime class. When you add a Period to a ZonedDateTime, the time differences are observed.

Standard Calendar

Clock

Most temporal-based objects provide a no-argument now() method that provides the current date and time using the system clock and the default time zone. These temporal-based objects also provide a one-argument now(Clock) method that allows you to pass in an alternative *Clock*.

The current date and time depends on the time-zone and, for globalized applications, a Clock is necessary to ensure that the date/time is created with the correct time-zone. So, although the use of the Clock class is optional, this feature allows you to test your code for other time zones, or by using a fixed clock, where time does not change.

The Clock class is abstract, so you cannot create an instance of it. The following factory methods can be useful for testing.

- Clock.offset(Clock, Duration) returns a clock that is offset by the specified Duration.
- Clock.systemUTC() returns a clock representing the Greenwich/UTC time zone.
- Clock.fixed(Instant, ZoneId) always returns the same Instant. For this clock, time stands still.

Standard Calendar

non-ISO Date Conversion

This tutorial does not discuss the java.time.chrono package in any detail. However, it might be useful to know that this package provides several predefined chronologies that are not ISO-based, such as Japanese, Hijrah, Minguo, and Thai Buddhist. You can also use this package to create your own chronology.
This section shows you how to convert between an ISO-based date and a date in one of the other predefined chronologies.

Converting to a Non-ISO-Based Date

You can convert an ISO-based date to a date in another chronology by using the from(TemporalAccessor) method, such as JapaneseDate.from(TemporalAccessor). This method throws a DateTimeException if it is unable to convert the date to a valid instance. The following code converts a LocalDateTime instance to several predefined non-ISO calendar dates:

```
LocalDateTime date = LocalDateTime.of(2013, Month.JULY, 20, 19, 30);
JapaneseDate jdate     = JapaneseDate.from(date);
HijrahDate hdate       = HijrahDate.from(date);
MinguoDate mdate       = MinguoDate.from(date);
ThaiBuddhistDate tdate = ThaiBuddhistDate.from(date);
```

The StringConverter example converts from a LocalDate to a ChronoLocalDate to a String and back. The toString method takes an instance of LocalDate and a Chronologyand returns the converted string by using the provided Chronology. The DateTimeFormatterBuilder is used to build a string that can be used for printing the date:

```
/**
 * Converts a LocalDate (ISO) value to a ChronoLocalDate date
 * using the provided Chronology, and then formats the
 * ChronoLocalDate to a String using a DateTimeFormatter with a
 * SHORT pattern based on the Chronology and the current Locale.
 *
 * @param localDate - the ISO date to convert and format.
 * @param chrono - an optional Chronology. If null, then IsoChronology is used.
 */
public static String toString(LocalDate localDate, Chronology chrono) {
    if (localDate != null) {
        Locale locale = Locale.getDefault(Locale.Category.FORMAT);
        ChronoLocalDate cDate;
        if (chrono == null) {
            chrono = IsoChronology.INSTANCE;
        }
        try {
            cDate = chrono.date(localDate);
        } catch (DateTimeException ex) {
            System.err.println(ex);
            chrono = IsoChronology.INSTANCE;
            cDate = localDate;
        }
        DateTimeFormatter dateFormatter =
            DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)
                             .withLocale(locale)
                             .withChronology(chrono)
                             .withDecimalStyle(DecimalStyle.of(locale));
        String pattern = "M/d/yyyy GGGGG";
        return dateFormatter.format(cDate);
    } else {
        return "";
    }
}
```

Standard Calendar

When the method is invoked with the following date for the predefined chronologies:

```
LocalDate date = LocalDate.of(1996, Month.OCTOBER, 29);
System.out.printf("%s%n",
     StringConverter.toString(date, JapaneseChronology.INSTANCE));
System.out.printf("%s%n",
     StringConverter.toString(date, MinguoChronology.INSTANCE));
System.out.printf("%s%n",
     StringConverter.toString(date, ThaiBuddhistChronology.INSTANCE));
System.out.printf("%s%n",
     StringConverter.toString(date, HijrahChronology.INSTANCE));
```

Output:

```
10/29/0008 H
10/29/0085 1
10/29/2539 B.E.
6/16/1417 1
```

Converting to an ISO-Based Date

You can convert from a non-ISO date to a LocalDate instance using the static LocalDate.from method, as shown in the following example: `LocalDate date = LocalDate.from(JapaneseDate.now());`

Other temporal-based classes also provide this method, which throws a DateTimeException if the date cannot be converted.

The fromString method, from the StringConverter example, parses a String containing a non-ISO date and returns a LocalDate instance.

```
/**
 * Parses a String to a ChronoLocalDate using a DateTimeFormatter
 * with a short pattern based on the current Locale and the
 * provided Chronology, then converts this to a LocalDate (ISO)
 * value.
 *
 * @param text    - the input date text in the SHORT format expected
 *                  for the Chronology and the current Locale.
 *
 * @param chrono - an optional Chronology. If null, then IsoChronology
 *                  is used.
 */
public static LocalDate fromString(String text, Chronology chrono) {
    if (text != null && !text.isEmpty()) {
        Locale locale = Locale.getDefault(Locale.Category.FORMAT);
        if (chrono == null) {
           chrono = IsoChronology.INSTANCE;
        }
        String pattern = "M/d/yyyy GGGGG";
        DateTimeFormatter df = new DateTimeFormatterBuilder().parseLenient()
                               .appendPattern(pattern)
                               .toFormatter()
                               .withChronology(chrono)
                               .withDecimalStyle(DecimalStyle.of(locale));
        TemporalAccessor temporal = df.parse(text);
        ChronoLocalDate cDate = chrono.date(temporal);
        return LocalDate.from(cDate);
    }
return null;
}
```

The printed strings should all convert back to October 29th, 1996:

```
1996-10-29
1996-10-29
1996-10-29
1996-10-29
```

When the method is invoked with the following strings:

```
System.out.printf("%s%n", StringConverter.fromString("10/29/0008 H",
    JapaneseChronology.INSTANCE));
System.out.printf("%s%n", StringConverter.fromString("10/29/0085 1",
    MinguoChronology.INSTANCE));
System.out.printf("%s%n", StringConverter.fromString("10/29/2539 B.E.",
    ThaiBuddhistChronology.INSTANCE));
System.out.printf("%s%n", StringConverter.fromString("6/16/1417 1",
    HijrahChronology.INSTANCE));
```

Standard Calendar

Legacy Date-Time Code

Prior to the Java SE 8 release, the Java date and time mechanism was provided by the java.util.Date, java.util.Calendar, and java.util.TimeZoneclasses, as well as their subclasses, such as java.util.GregorianCalendar. These classes had several drawbacks, including:

- The Calendar class was not type safe.
- Because the classes were mutable, they could not be used in multithreaded applications.
- Bugs in application code were common due to the unusual numbering of months and the lack of type safety.

Interoperability with Legacy Code

Perhaps you have legacy code that uses the java.util date and time classes and you would like to take advantage of the java.time functionality with minimal changes to your code.

Added to the JDK 8 release are several methods that allow conversion between java.util and java.time objects:

- Calendar.toInstant() converts the Calendar object to an Instant.
- GregorianCalendar.toZonedDateTime() converts a GregorianCalendar instance to a ZonedDateTime.
- GregorianCalendar.from(ZonedDateTime) creates a GregorianCalendar object using the default locale from a ZonedDateTime instance.
- Date.from(Instant) creates a Date object from an Instant.
- Date.toInstant() converts a Date object to an Instant.
- TimeZone.toZoneId() converts a TimeZone object to a ZoneId.

The following example converts a Calendar instance to a ZonedDateTime instance.
Note that a time zone must be supplied to convert from an Instant to a ZonedDateTime:

```
Calendar now = Calendar.getInstance();
ZonedDateTime zdt = ZonedDateTime.ofInstant(now.toInstant(), ZoneId.systemDefault()));
```

The following example shows conversion between a Date and an Instant:

```
Instant inst = date.toInstant();

Date newDate = Date.from(inst);
```

The following example converts from a GregorianCalendar to a ZonedDateTime, and then from a ZonedDateTime to a GregorianCalendar. Other temporal-based classes are created using the ZonedDateTime instance:

```
GregorianCalendar cal = ...;

TimeZone tz = cal.getTimeZone();
int tzoffset = cal.get(Calendar.ZONE_OFFSET);

ZonedDateTime zdt = cal.toZonedDateTime();

GregorianCalendar newCal = GregorianCalendar.from(zdt);

LocalDateTime ldt = zdt.toLocalDateTime();
LocalDate date = zdt.toLocalDate();
LocalTime time = zdt.toLocalTime();
```

Standard Calendar

Mapping java.util Date & Time Functionality to java.time

Because the Java implementation of date and time has been completely redesigned in the Java SE 8 release, you cannot swap one method for another method. If you want to use the rich functionality offered by the java.time package, your easiest solution is to use the toInstant or toZonedDateTime methods listed in the previous section. However, if you do not want to use that approach or it is not sufficient for your needs, then you must rewrite your date-time code.

The table introduced on the Overview page is a good place to begin evaluating which java.time classes meet your needs.
There is no one-to-one mapping correspondence between the two APIs, but the following table gives you a general idea of which functionality in the java.util date and time classes maps to the java.time APIs.

java.util Functionality	java.time Functionality	Comments
`java.util.Date`	`java.time.Instant`	The `Instant` and `Date` classes are similar. Each class: - Represents an instantaneous point of time on the timeline (UTC) - Holds a time independent of a time zone - Is represented as epoch-seconds (since 1970-01-01T00:00:00Z) plus nanoseconds The `Date.from(Instant)` and `Date.toInstant()` methods allow conversion between these classes.
`java.util.GregorianCalendar`	`java.time.ZonedDateTime`	The `ZonedDateTime` class is the replacement for `GregorianCalendar`. It provides the following similar functionality. Human time representation is as follows: `LocalDate`: year, month, day `LocalTime`: hours, minutes, seconds, nanoseconds `ZoneId`: time zone `ZoneOffset`: current offset from GMT The `GregorianCalendar.from(ZonedDateTime)` and `GregorianCalendar.to(ZonedDateTime)` methods facilitate conversions between these classes.
`java.util.TimeZone`	`java.time.ZoneId` or `java.time.ZoneOffset`	The `ZoneId` class specifies a time zone identifier and has access to the rules used each time zone. The `ZoneOffset` class specifies only an offset from Greenwich/UTC. For more information, see Time Zone and Offset Classes.
`GregorianCalendar` with the date set to `1970-01-01`	`java.time.LocalTime`	Code that sets the date to 1970-01-01 in a `GregorianCalendar` instance in order to use the time components can be replaced with an instance of `LocalTime`.
`GregorianCalendar` with time set to `00:00`.	`java.time.LocalDate`	Code that sets the time to 00:00 in a `GregorianCalendar` instance in order to use the date components can be replaced with an instance of `LocalDate`. (This `GregorianCalendar` approach was flawed, as midnight does not occur in some countries once a year due to the transition to daylight saving time.)

Interoperability with Legacy Code

Although the java.time.format.DateTimeFormatter provides a powerful mechanism for formatting date and time values, you can also use the java.time temporal-based classes directly with java.util.Formatter and String.format, using the same pattern-based formatting that you use with the java.util date and time classes.

Standard Calendar

The java.time package contains many classes that your programs can use to represent time and date. This is a very rich API. The key entry points for ISO-based dates are as follows:

- The Instant class provides a machine view of the timeline.
- The LocalDate, LocalTime, and LocalDateTime classes provide a human view of date and time without any reference to time zone.
- The ZoneId, ZoneRules, and ZoneOffset classes describe time zones, time zone offsets, and time zone rules.
- The ZonedDateTime class represents date and time with a time zone. The OffsetDateTime and OffsetTime classes represent date and time, or time, respectively. These classes take a time zone offset into account.
- The Duration class measures an amount of time in seconds and nanoseconds.
- The Period class measures an amount of time using years, months, and days.

Other non-ISO calendar systems can be represented using the java.time.chrono package. This package is beyond the scope of this tutorial, though the Non-ISO Date Conversion page provides information about converting an ISO-based date to another calendar system.

The Date Time API was developed as part of the Java community process under the designation of JSR 310. For more information, see JSR 310: Date and Time API.