Java Tutorials

Updated for Java SE 8

ORACLE

Java Tutorials

The Java Tutorials are a one stop shop for learning the Java language. The tutorials are practical guides for programmers who want to use the Java programming language to create applets and applications. They also serve as a reference for the experienced Java programmer.

Groups of related lessons are organized into "trails"; for example, the "Getting Started" trail, or the "Learning the Java Language" trail. This ebook contains a single trail. You can download other trails via the link on the Java Tutorials home page.

To read an ebook file that you have downloaded, transfer the document to your device.

We would love to hear what you think of this ebook, including any problems that you find. Please send us your feedback.

Legal Notices

This tutorial is a guide to developing applications for the Java Platform, Standard Edition and contains documentation (Tutorial) and sample code. The sample code made available with this Tutorial is licensed separately to you by Oracle under the Berkeley license. If you download any such sample code, you agree to the terms of the Berkeley license.

This Tutorial is provided to you by Oracle under the following license terms containing restrictions on use and disclosure and is protected by intellectual property laws. Oracle grants to you a limited, non-exclusive license to use this Tutorial for information purposes only, as an aid to learning about the Java SE platform. Except as expressly permitted in these license terms, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means this Tutorial. Reverse engineering, disassembly, or decompilation of this Tutorial is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If the Tutorial is licensed on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This Tutorial is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this Tutorial in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use.

THE TUTORIAL IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND. ORACLE FURTHER DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT.

IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOSS OF PROFITS, REVENUE, DATA OR DATA USE, INCURRED BY YOU OR ANY THIRD PARTY, WHETHER IN AN ACTION IN CONTRACT OR TORT, EVEN IF ORACLE HAS BEEN

ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. ORACLES ENTIRE LIABILITY FOR DAMAGES HEREUNDER SHALL IN NO EVENT EXCEED ONE THOUSAND DOLLARS (U.S. $1,000).

No Technical Support

Oracles technical support organization will not provide technical support, phone support, or updates to you.

The sample code and Tutorial may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Supported Platforms

The `mobi` file format works best on the following devices:

- Kindle Fire
- Kindle DX

The `ePub` file format works best on the following devices:

- iPad
- Nook
- Other eReaders that support the `ePub` format.

For best results when viewing preformatted code blocks, adjust the landscape/portrait orientation and font size of your device to enable the maximum possible viewing area.

Trail: Date Time

The Date-Time package, java.time, introduced in the Java SE 8 release, provides a comprehensive model for date and time and was developed under JSR 310: Date and Time API. Although `java.time` is based on the International Organization for Standardization (ISO) calendar system, commonly used global calendars are also supported.

This trail covers the fundamentals of using the ISO-based classes to represent date and time and to manipulate date and time values.

Trail: Date Time: Table of Contents

Lesson: Date-Time Overview

Time seems to be a simple subject; even an inexpensive watch can provide a reasonably accurate date and time. However, with closer examination, you realize the subtle complexities and many factors that affect your understanding of time. For example, the result of adding one month to January 31 is different for a leap year than for other years. Time zones also add complexity. For example, a country may go in and out of daylight saving time at short notice, or more than once a year or it may skip daylight saving time entirely for a given year.

The Date-Time API uses the calendar system defined in ISO-8601 as the default calendar. This calendar is based on the Gregorian calendar system and is used globally as the defacto standard for representing date and time. The core classes in the Date-Time API have names such as `LocalDateTime`, `ZonedDateTime`, and `OffsetDateTime`. All of these use the ISO calendar system. If you want to use an alternative calendar system, such as Hijrah or Thai Buddhist, the `java.time.chrono` package allows you to use one of the predefined calendar systems. Or you can create your own.

The Date-Time API uses the Unicode Common Locale Data Repository (CLDR). This repository supports the world's languages and contains the world's largest collection of locale data available. The information in this repository has been localized to hundreds of languages. The Date-Time API also uses the Time-Zone Database (TZDB). This database provides information about every time zone change globally since 1970, with history for primary time zones since the concept was introduced.

Note: See online version of topics in this ebook to download complete source code.

Date-Time Design Principles

The Date-Time API was developed using several design principles.

Clear

The methods in the API are well defined and their behavior is clear and expected. For example, invoking a Date-Time method with a `null` parameter value typically triggers a `NullPointerException`.

Fluent

The Date-Time API provides a fluent interface, making the code easy to read. Because most methods do not allow parameters with a `null` value and do not return a `null` value, method calls can be chained together and the resulting code can be quickly understood. For example:

```
LocalDate today = LocalDate.now();
LocalDate payday = today.with(TemporalAdjuster.lastDayOfMonth()).minusDays(2);
```

Immutable

Most of the classes in the Date-Time API create objects that are immutable, meaning that, after the object is created, it cannot be modified. To alter the value of an immutable object, a new object must be constructed as a modified copy of the original. This also means that the Date-Time API is, by definition, thread-safe. This affects the API in that most of the methods used to create date or time objects are prefixed with `of`, `from`, or `with`, rather than constructors, and there are no `set` methods. For example:

```
LocalDate dateOfBirth = LocalDate.of(2012, Month.MAY, 14);
LocalDate firstBirthday = dateOfBirth.plusYears(1);
```

Extensible

The Date-Time API is extensible wherever possible. For example, you can define your own time adjusters and queries, or build your own calendar system.

The Date-Time Packages

The Date-Time API consists of the primary package, `java.time`, and four subpackages:

`java.time`
: The core of the API for representing date and time. It includes classes for date, time, date and time combined, time zones, instants, duration, and clocks. These classes are based on the calendar system defined in ISO-8601, and are immutable and thread-safe.

`java.time.chrono`
: The API for representing calendar systems other than the default ISO-8601. You can also define your own calendar system. This tutorial does not cover this package in any detail.

`java.time.format`
: Classes for formatting and parsing dates and times.

`java.time.temporal`
: Extended API, primarily for framework and library writers, allowing interoperations between the date and time classes, querying, and adjustment. Fields (`TemporalField` and `ChronoField`) and units (`TemporalUnit` and `ChronoUnit`) are defined in this package.

`java.time.zone`
: Classes that support time zones, offsets from time zones, and time zone rules. If working with time zones, most developers will need to use only `ZonedDateTime`, and `ZoneId` or `ZoneOffset`.

Method Naming Conventions

The Date-Time API offers a rich set of methods within a rich set of classes. The method names are made consistent between classes wherever possible. For example, many of the classes offer a `now` method that captures the date or time values of the current moment that are relevant to that class. There are `from` methods that allow conversion from one class to another.

There is also standardization regarding the method name prefixes. Because most of the classes in the Date-Time API are immutable, the API does not include `set` methods. (After its creation, the value of an immutable object cannot be changed. The immutable equivalent of a `set` method is `with`.) The following table lists the commonly used prefixes:

Prefix	Method Type	Use
`of`	static factory	Creates an instance where the factory is primarily validating the input parameters, not converting them.
`from`	static factory	Converts the input parameters to an instance of the target class, which may involve losing information from the input.
`parse`	static factory	Parses the input string to produce an instance of the target class.
`format`	instance	Uses the specified formatter to format the values in the temporal object to produce a string.
`get`	instance	Returns a part of the state of the target object.
`is`	instance	Queries the state of the target object.
`with`	instance	Returns a copy of the target object with one element changed; this is the immutable equivalent to a `set` method on a JavaBean.
`plus`	instance	Returns a copy of the target object with an amount of time added.
`minus`	instance	Returns a copy of the target object with an amount of time subtracted.
`to`	instance	Converts this object to another type.
`at`	instance	Combines this object with another.

Lesson: 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.

This lesson covers the following topics:

Overview

This section compares the concepts of human time and machine time provides a table of the primary temporal-based classes in the `java.time` package.

DayOfWeek and Month Enums

This section discusses the enum that defines the days of the week (`DayOfWeek`) and the enum that defines months (`Month`).

Date Classes

This section shows the temporal-based classes that deal only with dates, without time or time zones. The four classes are `LocalDate`, `YearMonth`, `MonthDay` and `Year`.

Date and Time Classes

This section presents the `LocalTime` and `LocalDateTime` classes, which deal with time, and date and time, respectively, but without time zones.

Time Zone and Offset Classes

This section discusses the temporal-based classes that store time zone (or time zone offset) information, `ZonedDateTime`, `OffsetDateTime`, and `OffsetTime`. The supporting classes, `ZoneId`, `ZoneRules`, and `ZoneOffset`, are also discussed.

Instant Class

This section discusses the `Instant` class, which represents an instantaneous moment on the timeline.

Parsing and Formatting

This section provides an overview of how to use the predefined formatters to format and parse date and time values.

The Temporal Package

This section presents an overview of the `java.time.temporal` package, which supports the temporal classes, fields (`TemporalField` and `ChronoField`) and units (`TemporalUnit` and `ChronoUnit`). This section also explains how to use a temporal adjuster to retrieve an adjusted time value, such as "the first Tuesday after April 11", and how to perform a temporal query.

Period and Duration

This section explains how to calculate an amount of time, using both the `Period` and `Duration` classes, as well as the `ChronoUnit.between` method.

Clock

This section provides a brief overview of the `Clock` class. You can use this class to provide an alternative clock to the system clock.

Non-ISO Date Conversion

This section explains how to convert from a date in the ISO calendar system to a date in a non-ISO calendar system, such as a `JapaneseDate` or a `ThaiBuddhistDate`.

Legacy Date-Time Code

This section offers some tips on how to convert older `java.util.Date` and `java.util.Calendar` code to the Date-Time API.

Summary

This section provides a summary of the Standard Calendar lesson.

Note: See online version of topics in this ebook to download complete source code.

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.

For 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.

The following 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 `toString` method. 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 O
`Instant`						✔			`2013-08-20T15:16:2`
`LocalDate`	✔	✔	✔						`2013-08-20`
`LocalDateTime`	✔	✔	✔	✔	✔	✔			`2013-08-20T08:16:2`

ZonedDateTime	✓	✓	✓	✓	✓	✓	✓	✓	2013-08-21T00:16:26.941+09
LocalTime				✓	✓	✓			08:16:26.943
MonthDay		✓	✓						--08-20
Year	✓								2013
YearMonth	✓	✓							2013-08
Month		✓							AUGUST
OffsetDateTime	✓	✓	✓	✓	✓	✓	✓		2013-08-20T08:16:2
OffsetTime				✓	✓	✓	✓		08:16:26.957-07:00
Duration			**	**	**	✓			PT20H (20 hours)
Period	✓	✓	✓				***	***	P10D (10 days)

* 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.

DayOfWeek and 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 seven 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. For example, 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
```

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));
```

For more information about the `TemporalAdjuster` interface, see Temporal Adjuster.

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.
```

The Period and Duration section also has examples using the `LocalDate` class.

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
```

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();
```

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
```

Time Zone and 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.

```
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);
    }
    ...
}
```

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
```

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

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 650 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);
```

This produces the following 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.)
```

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`).

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

```
// Find the last Thursday in July 2013.
```

```
LocalDateTime date = LocalDateTime.of(2013, Month.JULY, 20, 19, 30);
ZoneOffset offset = ZoneOffset.of("-08:00");
OffsetDateTime date = OffsetDateTime.of(date, offset);
OffsetDateTime lastThursday =
        date.with(TemporalAdjuster.lastInMonth(DayOfWeek.THURSDAY));
System.out.printf("The last Thursday in July 2013 is the %sth.%n",
                  lastThursday.getDayOfMonth());
```

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.

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().plusHours(1);
```

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 `Instant` to 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.

Parsing and 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 `format` methods 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-argument `parse(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.

The `StringConverter` example on the Non-ISO Date Conversion page provides another example of a date formatter.

Formatting

The `format(DateTimeFormatter)` method converts a temporal-based object to a string representation using the specified format. The following code, from the `Flight` example, 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, is as follows:

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

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 and 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 and 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`.

ChronoUnit

The `ChronoUnit` enum implements the `TemporalUnit` interface, and provides a set of standard units based on date and time, from milliseconds to millenia. Note that not all `ChronoUnit` objects are supported by all classes. For example, the `Instant` class does not support `ChronoUnit.MONTHS` or `ChronoUnit.YEARS`. The `TemporalAccessor.isSupported(TemporalUnit)` method 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`.

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

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
```

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. The `PaydayAdjuster` 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
```

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:

```
TemporalQueries 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 `TemporalQuery` interface. 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;
}
```

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 `CheckDate` example, 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);
```

Period and 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.between` method. 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);
}
...
```

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.

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.

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 `Chronology` and 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 "";
    }
}
```

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));
```

The output looks like this:

```
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;
}
```

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));
```

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

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

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.TimeZone` classes, 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();
```

Mapping java.util Date and 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 s Each class: - Represents an instantaneous point the timeline (UTC) - Holds a time independent of a time - Is represented as epoch-seconds (s 1970-01-01T00:00:00Z) plus nanos The `Date.from(Instant)` and `Date.toInstant()` methods allow conversion between these classes.
		The `ZonedDateTime` class is the rep for `GregorianCalendar`. It provide following similar functionality. Human time representation is as foll `LocalDate`: year, month, day `LocalTime`: hours, minutes, seconds nanoseconds

`java.util.GregorianCalendar`	`java.time.ZonedDateTime`	`ZoneId`: time zone `ZoneOffset`: current offset from GM The `GregorianCalendar.from(ZonedI` and `GregorianCalendar.to(ZonedDat` methods faciliate conversions betwe classes.
`java.util.TimeZone`	`java.time.ZoneId` or `java.time.ZoneOffset`	The `ZoneId` class specifies a time zo identifier and has access to the rules time zone. The `ZoneOffset` class sp only an offset from Greenwich/UTC. information, see Time Zone and Offs Classes.
`GregorianCalendar` with the date set to `1970-01-01`	`java.time.LocalTime`	Code that sets the date to 1970-01-0 `GregorianCalendar` instance in orc the time components can be replaced 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 orc the date components can be replaced instance of `LocalDate`. (This `GregorianCalendar` approach was as midnight does not occur in some c once a year due to the transition to d saving time.)

Date and Time Formatting

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.

Summary

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.

Questions and Exercises: Date-Time API

Questions

1. Which class would you use to store your birthday in years, months, days, seconds, and nanoseconds?

2. Given a random date, how would you find the date of the previous Thursday?

3. What is the difference between a `ZoneId` and a `ZoneOffset`?

4. How would you convert an `Instant` to a `ZonedDateTime`? How would you convert a `ZonedDateTime` to an `Instant`?

Exercises

1. Write an example that, for a given year, reports the length of each month within that year.

2. Write an example that, for a given month of the current year, lists all of the Mondays in that month.

3. Write an example that tests whether a given date occurs on Friday the 13th.

Check your answers.

Questions

Question 1. Which class would you use to store your birthday in years, months, days, seconds, and nanoseconds?

Answer 1. Most likely you would use the `LocalDateTime` class. To take a particular time zone into account, you would use the `ZonedDateTime` class. Both classes track date and time to nanosecond precision and both classes, when used in conjunction with `Period`, give a result using a combination of human-based units, such as years, months, and days.

Question 2. Given a random date, how would you find the date of the previous Thursday?

Answer 2. You can use the `previous` method of a `TemporalAdjuster`:

```
LocalDate date = ...;
System.out.printf("The previous Thursday is: %s%n",
          date.with(TemporalAdjuster.previous(DayOfWeek.THURSDAY)));
```

Question 3. What is the difference between a `ZoneId` and a `ZoneOffset`?

Answer 3. Both `ZoneId` and `ZoneOffset` track an offset from Greenwich/UTC time, but the `ZoneOffset` class tracks only the absolute offset from Greenwich/UTC. The `ZoneId` class also uses the `ZoneRules` to determine how an offset varies for a particular time of year and region.

Question 4. How would you convert an `Instant` to a `ZonedDateTime`? How would you convert a `ZonedDateTime` to an `Instant`?

Answer 4. You can convert an `Instant` to a `ZonedDateTime` by using the `ZonedDateTime.ofInstant` method. You also need to supply a `ZoneId`:

```
ZonedDateTime zdt = ZonedDateTime.ofInstant(Instant.now(),
                                            ZoneId.systemDefault());
```

Alternatively, you could use the `Instant.atZone` method:

```
ZonedDateTime zdt = Instant.now().atZone(ZoneId.systemDefault());
```

You can use the `toInstant` method in the `ChronoZonedDateTime` interface, implemented by the `ZonedDateTime` class, to convert from a `ZonedDateTime` to an `Instant`:

```
Instant inst = ZonedDateTime.now().toInstant();
```

Exercises

Exercise 1. Write an example that, for a given year, reports the length of each month within that year.

Answer 1. See `MonthsInYear.java` for a solution.

Exercise 2. Write an example that, for a given month of the current year, lists all of the Mondays in that month.

Answer 2. See `ListMondays.java` for a solution.

Exercise 3. Write an example that tests whether a given date occurs on Friday the 13th.

Answer 3. See `Superstitious.java` and `FridayThirteenQuery.java` for a solution.

Date Time: End of Trail

You have reached the end of the "Date Time" trail.

If you have comments or suggestions about this trail, use our feedback page to tell us about it.

 Essential Classes: Contains information about strings and properties, both of which are used when internationalizing programs.