OCA Java SE 8 Programmer I Study Guide (Exam 1Z0-808)
Chapter 10 - Programming with the Date and Time API
Oracle Press © 2016

Chapter 10: Programming with the Date and Time API

Oracle's exam topics include working with selected classes from the Java SE API. In this chapter and the next, we cover the objectives related to the new features of Java SE 8. The new Date and Time API is covered in this chapter, and new lambda expressions are covered in Chapter 11.

Because most applications rely heavily on calendar data, most people need to be familiar with this API. Whether calendar data is being presented on a web page, persisted in a database, or present in logging records or filenames, calendar data is everywhere when it comes to software applications. The rich, robust, and fluent Java SE 8 Date and Time API makes it easy for coders to work with calendar data.

Objectives related to APIs originating in Java versions previous to Java 8 are covered in prior chapters. This additional external coverage includes the following:

❑ Creating and manipulating strings (see Chapter 3)
❑ Manipulating data using the `StringBuilder` class and its methods (see Chapter 3)
❑ Declaring and using an `ArrayList` of a given type (see Chapter 6)

CERTIFICATION OBJECTIVE: Understand the Date and Time API

Exam Objective Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period

Date, time, and calendar calculations are supported by the Date and Time API (Java Special Request [JSR] 310), which is provided by the ThreeTen Project (www.threeten.org) as its reference implementation (RI). JSR 310 is available in Java 8. The Date and Time API includes five calendar-related packages: `java.time`, `java.time.chrono`, `java.time.format`, `java.time.temporal`, and `java.time.zone`. For the OCA 8 exam, you will need to be acquainted with only a few classes in the `java.time` package—`LocalTime`, `LocalDate`, `LocalDateTime`, `DateTimeFormatter`, and `Period`—all of which are covered in this chapter.

The International Organization for Standardization date and time data exchange mode (ISO 8601) is used by the Date and Time API. ISO 8601 is properly named, "Data elements and interchange formats – Information interchange – Representation of dates and times." The Gregorian calendar sets the basis for ISO 8601 and for the Date and Time API.

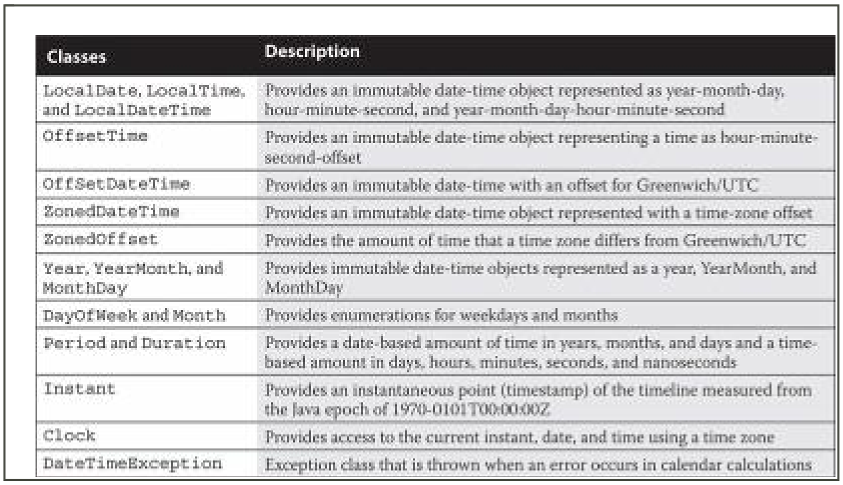
This chapter explores the Date and Time API through calendar data creation, calendar data manipulation, period support, and calendar data formatting support. Each area of coverage is provided in its own section.

Calendar Data Creation

Prior to Java 8, calendar data creation was supported with the `Date`, `Calendar`, and `GregorianCalendar` classes. Moving forward, we are leaving these classes in the past and aim to create our calendar data with a new set of classes. For the exam, you will need to master three calendar data creation classes: `LocalDate`, `LocalTime`, and `LocalDateTime`.

Before stepping through each one of these classes, let's take a look at the main classes in the API used in association with calendar data creation. These classes are shown in Table 10-1.

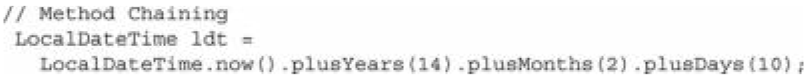
Table 10-1: Calendar Creation–Related Classes



Classes	Description
`LocalDate`, `LocalTime`, and `LocalDateTime`	Provides an immutable date-time object represented as year-month-day, hour-minute-second, and year-month-day-hour-minute-second
`OffsetTime`	Provides an immutable date-time object representing a time as hour-minute-second-offset
`OffSetDateTime`	Provides an immutable date-time with an offset for Greenwich/UTC
`ZonedDateTime`	Provides an immutable date-time object represented with a time-zone offset
`ZonedOffset`	Provides the amount of time that a time zone differs from Greenwich/UTC
`Year`, `YearMonth`, and `MonthDay`	Provides immutable date-time objects represented as a year, YearMonth, and MonthDay
`DayOfWeek` and `Month`	Provides enumerations for weekdays and months
`Period` and `Duration`	Provides a date-based amount of time in years, months, and days and a time-based amount in days, hours, minutes, seconds, and nanoseconds
`Instant`	Provides an instantaneous point (timestamp) of the timeline measured from the Java epoch of 1970-0101T00:00:00Z
`Clock`	Provides access to the current instant, date, and time using a time zone
`DateTimeException`	Exception class that is thrown when an error occurs in calendar calculations

Many say Java reads like a book, and so do we. The Date and Time API makes use of the fluent API design to influence the implementation of its API. A fluent API, also known as a fluent interface, makes code more readable and maintainable. The usability goals of fluent APIs are achieved using *method chaining*, which allows objects to be wired together. Here's an example:

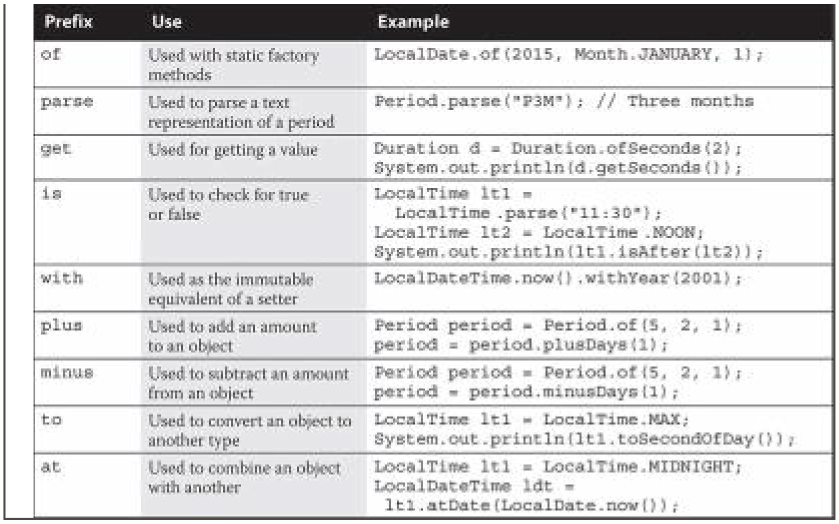
Larger View



```
// Method Chaining
 LocalDateTime ldt =
   LocalDateTime.now().plusYears(14).plusMonths(2).plusDays(10);
```

The method prefixes in Table 10-2 are seen throughout the API when creating, manipulating, and formatting calendar data and when working with the `Period` class.

Table 10-2: Date and Time API Method Prefixes



Prefix	Use	Example
of	Used with static factory methods	`LocalDate.of(2015, Month.JANUARY, 1);`
parse	Used to parse a text representation of a period	`Period.parse("P3M"); // Three months`

get	Used for getting a value	`Duration d = Duration.ofSeconds(2);`
		`System.out.println(d.getSeconds());`

is	Used to check for true or false	`LocalTime lt1 =`
		`LocalTime.parse("11:30");`
		`LocalTime lt2 = LocalTime.NOON;`
		`System.out.println(lt1.isAfter(lt2));`

with	Used as the immutable equivalent of a setter	`LocalDateTime.now().withYear(2001);`

plus	Used to add an amount to an object	`Period period = Period.of(5, 2, 1);`
		`period = period.plusDays(1);`

minus	Used to subtract an amount from an object	`Period period = Period.of(5, 2, 1);`
		`period = period.minusDays(1);`

to	Used to convert an object to another type	`LocalTime lt1 = LocalTime.MAX;`
		`System.out.println(lt1.toSecondOfDay());`

at	Used to combine an object with another	`LocalTime lt1 = LocalTime.MIDNIGHT;`
		`LocalDateTime ldt =`
		`lt1.atDate(LocalDate.now());`

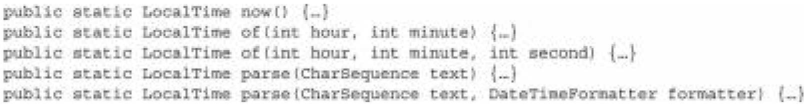
When creating dates, the `of`, `parse`, and `now` methods are commonly used for the `LocalTime`, `LocalDate`, and `LocalDateTime` classes.

LocalTime Class

The `LocalTime` class includes several method declarations in support of creating a time (without a date or time zone).

Here are some of the `LocalTime` class's method declarations:

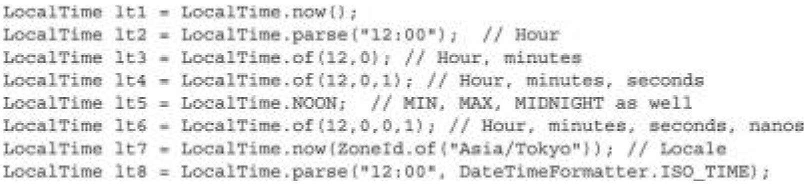
Larger View



```
public static LocalTime now() {…}
public static LocalTime of(int hour, int minute) {…}
public static LocalTime of(int hour, int minute, int second) {…}
public static LocalTime parse(CharSequence text) {…}
public static LocalTime parse(CharSequence text, DateTimeFormatter formatter) {…}
```

Here are some examples:

Larger View



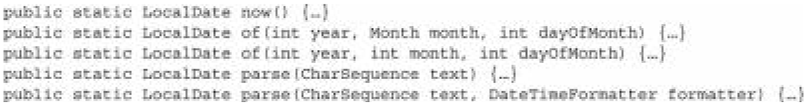
```
LocalTime lt1 = LocalTime.now();
LocalTime lt2 = LocalTime.parse("12:00");  // Hour
LocalTime lt3 = LocalTime.of(12,0); // Hour, minutes
LocalTime lt4 = LocalTime.of(12,0,1); // Hour, minutes, seconds
LocalTime lt5 = LocalTime.NOON;  // MIN, MAX, MIDNIGHT as well
LocalTime lt6 = LocalTime.of(12,0,0,1); // Hour, minutes, seconds, nanos
LocalTime lt7 = LocalTime.now(ZoneId.of("Asia/Tokyo")); // Locale
LocalTime lt8 = LocalTime.parse("12:00", DateTimeFormatter.ISO_TIME);
```

LocalDate Class

The `LocalDate` class includes several method declarations in support of creating a time without a time or time zone.

Here are some of the `LocalDate` class's method declarations:

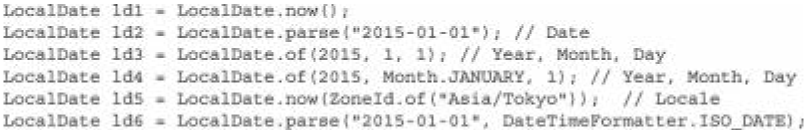
Larger View



```
public static LocalDate now() {…}
public static LocalDate of(int year, Month month, int dayOfMonth) {…}
public static LocalDate of(int year, int month, int dayOfMonth) {…}
public static LocalDate parse(CharSequence text) {…}
public static LocalDate parse(CharSequence text, DateTimeFormatter formatter) {…}
```

Here are some examples:

Larger View



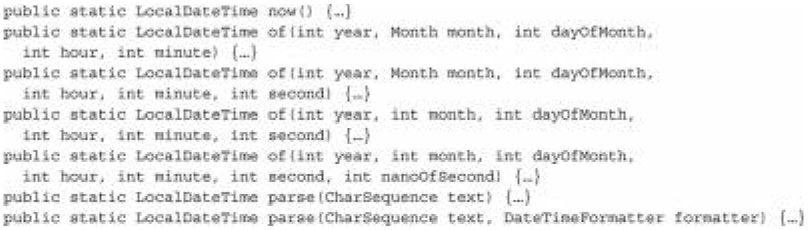
```
LocalDate ld1 = LocalDate.now();
LocalDate ld2 = LocalDate.parse("2015-01-01"); // Date
LocalDate ld3 = LocalDate.of(2015, 1, 1); // Year, Month, Day
LocalDate ld4 = LocalDate.of(2015, Month.JANUARY, 1); // Year, Month, Day
LocalDate ld5 = LocalDate.now(ZoneId.of("Asia/Tokyo"));  // Locale
LocalDate ld6 = LocalDate.parse("2015-01-01", DateTimeFormatter.ISO_DATE);
```

LocalDateTime Class

The `LocalDateTime` class includes several method declarations in support of creating a date-time without a time zone.

Here are some of the `LocalDateTime` class's method declarations:

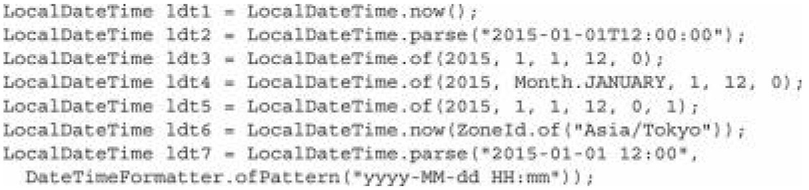
Larger View



```
public static LocalDateTime now() {…}
public static LocalDateTime of(int year, Month month, int dayOfMonth,
  int hour, int minute) {…}
public static LocalDateTime of(int year, Month month, int dayOfMonth,
  int hour, int minute, int second) {…}
public static LocalDateTime of(int year, int month, int dayOfMonth,
  int hour, int minute, int second) {…}
public static LocalDateTime of(int year, int month, int dayOfMonth,
  int hour, int minute, int second, int nanoOfSecond) {…}
public static LocalDateTime parse(CharSequence text) {…}
public static LocalDateTime parse(CharSequence text, DateTimeFormatter formatter) {…}
```

Here are some examples:

Larger View

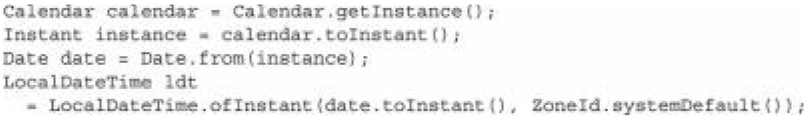


```
LocalDateTime ldt1 = LocalDateTime.now();
LocalDateTime ldt2 = LocalDateTime.parse("2015-01-01T12:00:00");
LocalDateTime ldt3 = LocalDateTime.of(2015, 1, 1, 12, 0);
LocalDateTime ldt4 = LocalDateTime.of(2015, Month.JANUARY, 1, 12, 0);
LocalDateTime ldt5 = LocalDateTime.of(2015, 1, 1, 12, 0, 1);
LocalDateTime ldt6 = LocalDateTime.now(ZoneId.of("Asia/Tokyo"));
LocalDateTime ldt7 = LocalDateTime.parse("2015-01-01 12:00",
  DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm"));
```

Legacy Date/Time Support

Legacy calendar classes are supported by new methods to allow integration with JSR 310. These changes include updates to `java.util.Calendar`, `java.util.DateFormat`, `java.util.GregorianCalendar`, `java.util.TimeZone`, and `java.util.Date`. The following code demonstrates the integration of the older classes (such as `Calendar` and `Date`) with the new classes of JSR 310 (such as `Instant` and `LocalDateTime`). This interoperability is not on the exam, but it is helpful to know.

Larger View



```
Calendar calendar = Calendar.getInstance();
Instant instance = calendar.toInstant();
Date date = Date.from(instance);
LocalDateTime ldt
  = LocalDateTime.ofInstant(date.toInstant(), ZoneId.systemDefault());
```

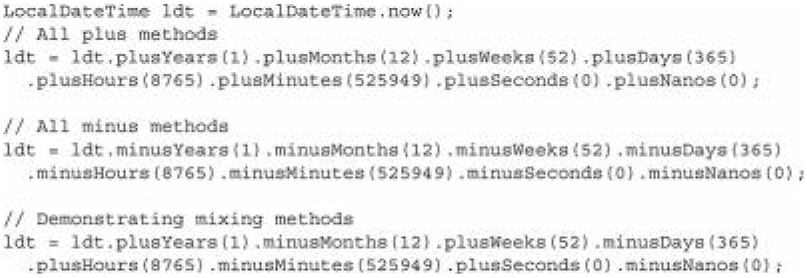
On the Job ***Four regional calendars are packaged with Java SE 8: Hijrah, Japanese imperial, Minguo, and Thai Buddhist. The API is***

***flexible enough to allow for the creation of additional calendars. For new calendars, the `Era, Chronology`, and `ChronoLocalDate` interfaces need to be implemented*.**

Calendar Data Manipulation

Questions on manipulating calendar data will perhaps be the easiest part of your exam. This section is concerned with adding or subtracting units of time to instances of `LocalTime`, `LocalDate`, and `LocalDateTime`. You should know 16 `plus/minus` methods that all apply to `LocalDateTime`, eight methods that apply to `LocalDate`, and eight methods that apply to `LocalTime`. Let's look at all of them starting with `LocalDateTime`.

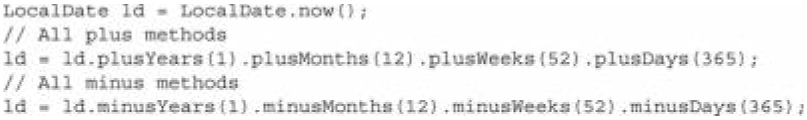
Larger View



```
LocalDateTime ldt = LocalDateTime.now();
// All plus methods
ldt = ldt.plusYears(1).plusMonths(12).plusWeeks(52).plusDays(365)
  .plusHours(8765).plusMinutes(525949).plusSeconds(0).plusNanos(0);

// All minus methods
ldt = ldt.minusYears(1).minusMonths(12).minusWeeks(52).minusDays(365)
  .minusHours(8765).minusMinutes(525949).minusSeconds(0).minusNanos(0);

// Demonstrating mixing methods
ldt = ldt.plusYears(1).minusMonths(12).plusWeeks(52).minusDays(365)
  .plusHours(8765).minusMinutes(525949).plusSeconds(0).minusNanos(0);
```

Working with the `LocalDate` class, you can add and subtract units of years, months, weeks, and days. In this context, you cannot add or subtract units of hours, minutes, seconds, or nanos.

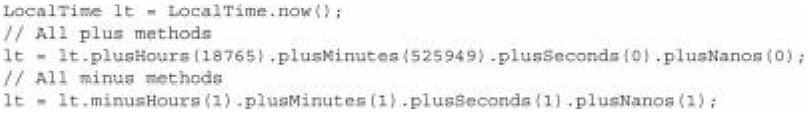
Larger View



```
LocalDate ld = LocalDate.now();
// All plus methods
ld = ld.plusYears(1).plusMonths(12).plusWeeks(52).plusDays(365);
// All minus methods
ld = ld.minusYears(1).minusMonths(12).minusWeeks(52).minusDays(365);
```

Working with the `LocalTime` class, you can add and subtract units of hours, minutes, seconds, or nanos. In this context, you cannot add or subtract units of years, months, weeks, or days.

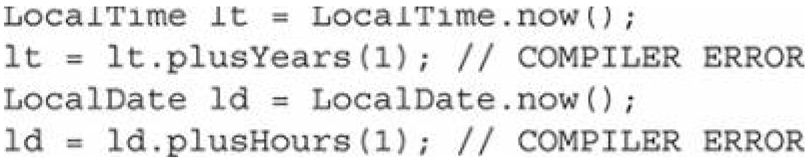
Larger View



```
LocalTime lt = LocalTime.now();
// All plus methods
lt = lt.plusHours(18765).plusMinutes(525949).plusSeconds(0).plusNanos(0);
// All minus methods
lt = lt.minusHours(1).plusMinutes(1).plusSeconds(1).plusNanos(1);
```

Look for the exam to try and trip you up in using methods where they do not belong. In the following code segment, `plusYears` is not a method of the `LocalTime` class and `plusHours` is not a method of the `LocalDate` class. The compiler will let you know accordingly—but you won't have a compiler at the exam.

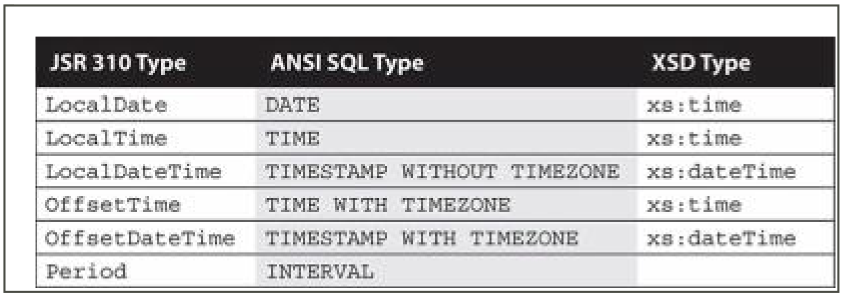
Larger View



```
LocalTime lt = LocalTime.now();
lt = lt.plusYears(1); // COMPILER ERROR
LocalDate ld = LocalDate.now();
ld = ld.plusHours(1); // COMPILER ERROR
```

On the Job *Interoperability between the calendar types within the `java.time` and `java.sql` packages exists in the Java API. Table 10-3 provides the relationships between the JSR 310 types and the SQL types, as well as the XML Schema (XSD) types. Note that there were no changes made to the JDBC API. Instead, you have to use the `setObject/getObject` to use this new API with JDBC.*

Table 10-3: JSR 310, SQL, and XSD Type Mapping in the Java SE API



JSR 310 Type	ANSI SQL Type	XSD Type
LocalDate	DATE	xs:time
LocalTime	TIME	xs:time
LocalDateTime	TIMESTAMP WITHOUT TIMEZONE	xs:dateTime
OffsetTime	TIME WITH TIMEZONE	xs:time
OffsetDateTime	TIMESTAMP WITH TIMEZONE	xs:dateTime
Period	INTERVAL	

Calendar Periods

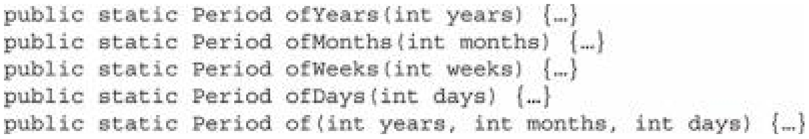
A calendar `Period` in Java is a date-based amount made up of years, months, and days. A calendar `Duration` is a time-based amount made up of days, hours, minutes, seconds, and nanoseconds. The `Period` class is on the exam, but the `Duration` class is not. Both classes implement the `ChronoPeriod` interface. Several methods of the `Period` class are commonly used, such as the following: `of[interval]`, `parse`, `get[interval]`, `with[interval]`, `plus[interval]`, `minus[interval]`, `is[`*`state`*`]`, and `between` methods. These methods and more are detailed in the following section with descriptions, declarations, and examples.

The of[*interval*] Method

The `Period` class `of`*`[interval]`* method returns a `Period` from an integer value representing years, months, weeks, or days.

There are five `of[`*`interval`*`]` method declarations:

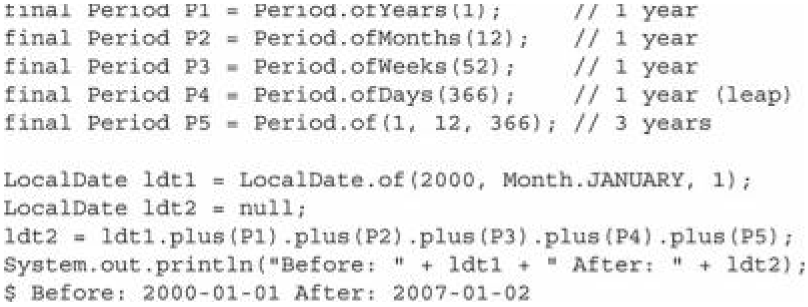
Larger View



```
public static Period ofYears(int years) {…}
public static Period ofMonths(int months) {…}
public static Period ofWeeks(int weeks) {…}
public static Period ofDays(int days) {…}
public static Period of(int years, int months, int days) {…}
```

Here are some examples:

Larger View



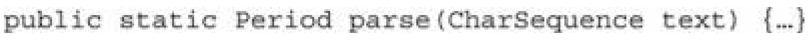
```
final Period P1 = Period.ofYears(1);     // 1 year
final Period P2 = Period.ofMonths(12);   // 1 year
final Period P3 = Period.ofWeeks(52);    // 1 year
final Period P4 = Period.ofDays(366);    // 1 year (leap)
final Period P5 = Period.of(1, 12, 366); // 3 years

LocalDate ldt1 = LocalDate.of(2000, Month.JANUARY, 1);
LocalDate ldt2 = null;
ldt2 = ldt1.plus(P1).plus(P2).plus(P3).plus(P4).plus(P5);
System.out.println("Before: " + ldt1 + " After: " + ldt2);
$ Before: 2000-01-01 After: 2007-01-02
```

The parse Method

The `Period` class static `parse` method returns a `Period` from a string `PnYnMnD`, where `P` is for period, `Y` is for years, `M` is for months, and `D` is for days. A `Period` is also returned from a string `PnW`, where `P` is for period and `W` is for weeks.

There is one `parse` method declaration:

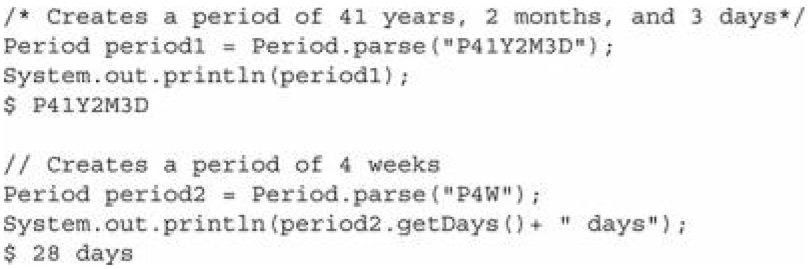
Larger View



```
public static Period parse(CharSequence text) {…}
```

Here is an example:

Larger View



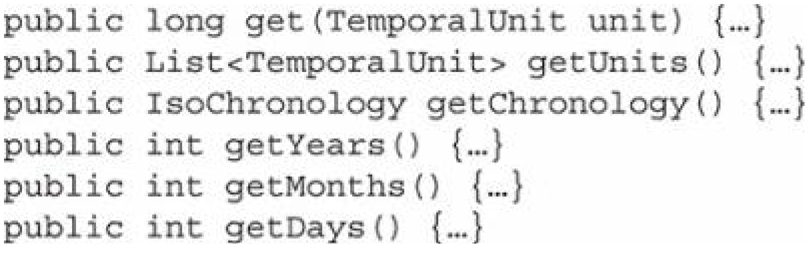
```
/* Creates a period of 41 years, 2 months, and 3 days*/
Period period1 = Period.parse("P41Y2M3D");
System.out.println(period1);
$ P41Y2M3D

// Creates a period of 4 weeks
Period period2 = Period.parse("P4W");
System.out.println(period2.getDays()+ " days");
$ 28 days
```

The get[*interval*] Method

The `Period` class `get[`*`interval`*`]` method returns a value relative to the type described in the method name.

There are six `get[`*`interval`*`]` method declarations:

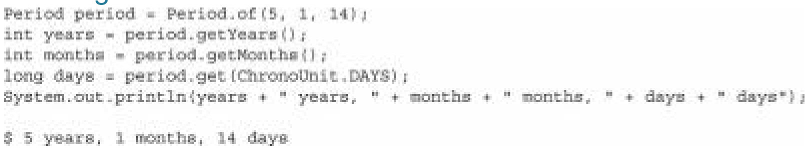
Larger View



```
public long get(TemporalUnit unit) {…}
public List<TemporalUnit> getUnits() {…}
public IsoChronology getChronology() {…}
public int getYears() {…}
public int getMonths() {…}
public int getDays() {…}
```

Here are some examples:

Larger View



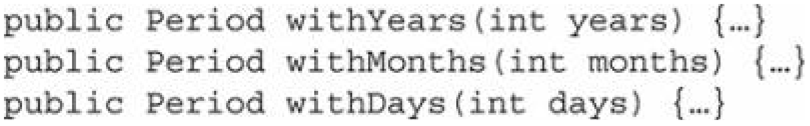
```
Period period = Period.of(5, 1, 14);
int years = period.getYears();
int months = period.getMonths();
long days = period.get(ChronoUnit.DAYS);
System.out.println(years + " years, " + months + " months, " + days + " days");

$ 5 years, 1 months, 14 days
```

The with[*interval*] Methods

The `Period` class `with[`*`interval`*`]` method returns a copy of the `Period` object from a specified `int` that identifies either the years, months, or days value to change.

There are three `with[`*`interval`*`]` method declarations:

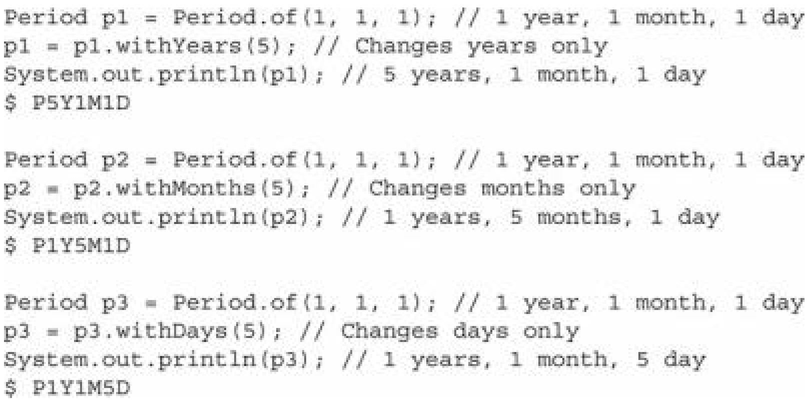
Larger View



```
public Period withYears(int years) {…}
public Period withMonths(int months) {…}
public Period withDays(int days) {…}
```

Here are some examples:

Larger View



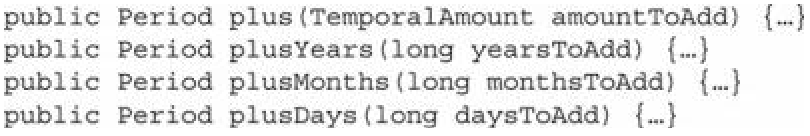
```
Period p1 = Period.of(1, 1, 1); // 1 year, 1 month, 1 day
p1 = p1.withYears(5); // Changes years only
System.out.println(p1); // 5 years, 1 month, 1 day
$ P5Y1M1D

Period p2 = Period.of(1, 1, 1); // 1 year, 1 month, 1 day
p2 = p2.withMonths(5); // Changes months only
System.out.println(p2); // 1 years, 5 months, 1 day
$ P1Y5M1D

Period p3 = Period.of(1, 1, 1); // 1 year, 1 month, 1 day
p3 = p3.withDays(5); // Changes days only
System.out.println(p3); // 1 years, 1 month, 5 day
$ P1Y1M5D
```

The plus[*interval*] Method

The `Period` class `plus[`*`interval`*`]` method returns a copy of the `Period` object from a specified `long` index value or `TemporalAmount` with the desired amount added.

There are four `plus[interval]` method declarations:

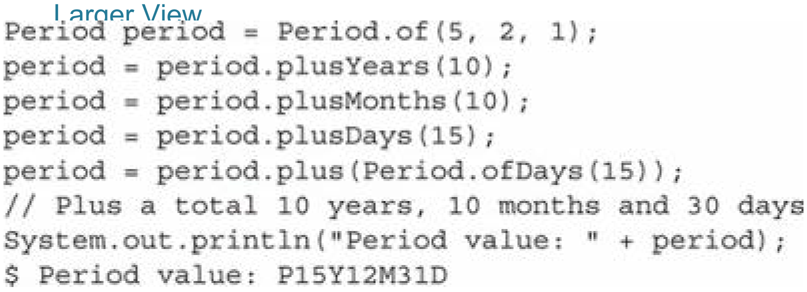
Larger View



```
public Period plus(TemporalAmount amountToAdd) {…}
public Period plusYears(long yearsToAdd) {…}
public Period plusMonths(long monthsToAdd) {…}
public Period plusDays(long daysToAdd) {…}
```

Here are some examples:

Larger View



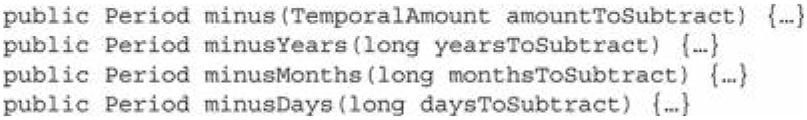
```
Period period = Period.of(5, 2, 1);
period = period.plusYears(10);
period = period.plusMonths(10);
period = period.plusDays(15);
period = period.plus(Period.ofDays(15));
// Plus a total 10 years, 10 months and 30 days
System.out.println("Period value: " + period);
$ Period value: P15Y12M31D
```

The minus[*interval*] Method

The `Period` class `minus[interval]` method returns a copy of the `Period` object from a specified `long` index value or `TemporalAmount` with the desired amount added.

There are four `minus[interval]` method declarations:

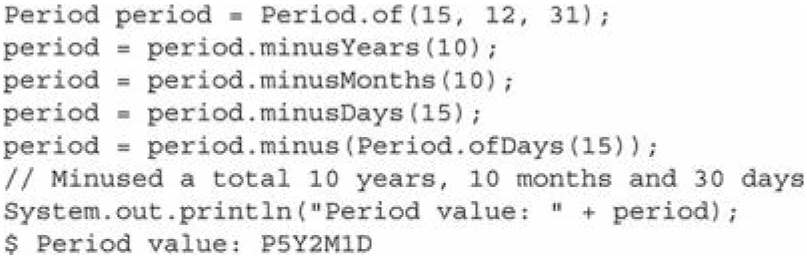
Larger View



```
public Period minus(TemporalAmount amountToSubtract) {…}
public Period minusYears(long yearsToSubtract) {…}
public Period minusMonths(long monthsToSubtract) {…}
public Period minusDays(long daysToSubtract) {…}
```

Here are some examples:

Larger View



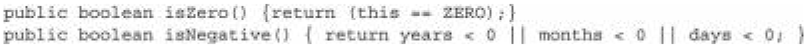
```
Period period = Period.of(15, 12, 31);
period = period.minusYears(10);
period = period.minusMonths(10);
period = period.minusDays(15);
period = period.minus(Period.ofDays(15));
// Minused a total 10 years, 10 months and 30 days
System.out.println("Period value: " + period);
$ Period value: P5Y2M1D
```

The is[*state*] Method

The `Period` class `is[state]` method returns a `boolean` from a string `PnYnMnD`, where `P` is for period, `Y` is for years, `M` is for months, and `D` is for days.

There are two `is[state]` method declarations:

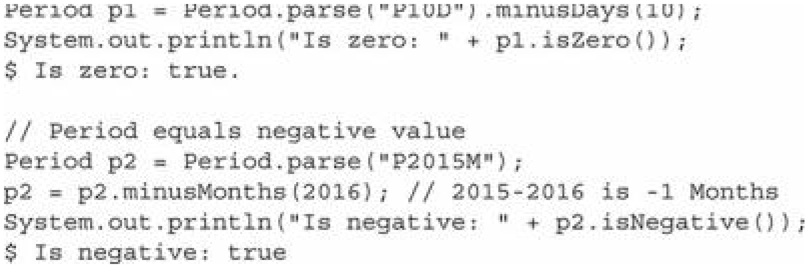
Larger View



```
public boolean isZero() {return (this == ZERO);}
public boolean isNegative() { return years < 0 || months < 0 || days < 0; }
```

Here is an example:

Larger View



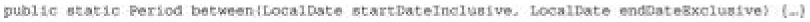
```
Period p1 = Period.parse("P10D").minusDays(10);
System.out.println("Is zero: " + p1.isZero());
$ Is zero: true.

// Period equals negative value
Period p2 = Period.parse("P2015M");
p2 = p2.minusMonths(2016); // 2015-2016 is -1 Months
System.out.println("Is negative: " + p2.isNegative());
$ Is negative: true
```

The between Method

The `Period` class `between` method returns a `Period` from two `LocalDate` arguments.

There is one `between` method declaration:

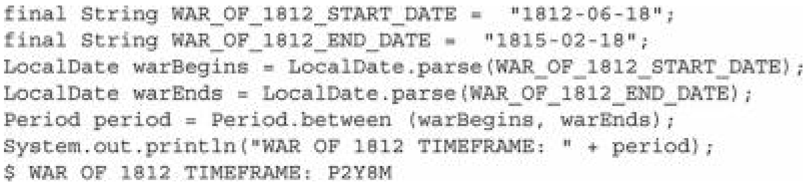
Larger View



```
public static Period between(LocalDate startDateInclusive, LocalDate endDateExclusive) {…}
```

Here is an example:

Larger View

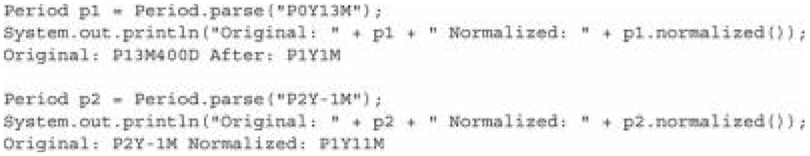


```
final String WAR_OF_1812_START_DATE =  "1812-06-18";
final String WAR_OF_1812_END_DATE =  "1815-02-18";
LocalDate warBegins = LocalDate.parse(WAR_OF_1812_START_DATE);
LocalDate warEnds = LocalDate.parse(WAR_OF_1812_END_DATE);
Period period = Period.between (warBegins, warEnds);
System.out.println("WAR OF 1812 TIMEFRAME: " + period);
$ WAR OF 1812 TIMEFRAME: P2Y8M
```

EXERCISE 10-1: Using the normalized Method of the Period Class

In this exercise, you will examine the `normalized` method of the `Period` class. This method is not on the exam, but this exercise will help get you more familiar with the `Period` class. The `normalized` method adjusts the months in tandem with the years so there is never less than zero or more than eleven months, as you see demonstrated here:

Larger View



```
Period p1 = Period.parse("P0Y13M");
System.out.println("Original: " + p1 + " Normalized: " + p1.normalized());
Original: P13M400D After: P1Y1M

Period p2 = Period.parse("P2Y-1M");
System.out.println("Original: " + p2 + " Normalized: " + p2.normalized());
Original: P2Y-1M Normalized: P1Y11M
```

Both of the `Period` and `Duration` classes implement the `TemporalAmount` interface.

1. Use an IDE (such as NetBeans) to view the contents of the src.zip file that is distributed in JDK 1.8 at C:\Program Files\Java\jdk1.8.0_40\src.zip.

2. Open the package node for `java.util` and double-click the `Period.java` class.

3. Examine the Javadoc header and body of the `normalized` method to get a better idea of exactly how the method operates.

4. Now answer these questions:

   a. As the `Period` class has a `normalized` method, does the `Duration` class have one as well?

   b. If the `Duration` class does have a `normalized` method, what does it normalize?

   c. If the `Duration` class does not have a `normalized` method, why not?

5. Visit the Javadoc for the `Duration` class to verify your hypothesis.

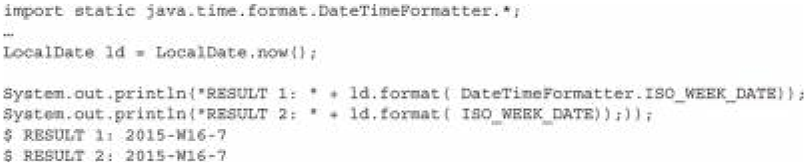
Calendar Data Formatting

Calendar data formatting is supported by the `DateTimeFormatter` class. The API supplies predefined formatters, localized formatting with support of the `FormatStyle` enumeration type (enum), and specialized formatting (which is your own customization). The following sections examine all three.

Predefined Formatters

Several predefined formatters are included in the `DateTimeFormatter` class. The constant static variables that associate each formatter can be used directly with the class name, or the static import can be used to remove the class name from inline use, as shown here. This means that `DateTimeFormatter.ISO_WEEK_DATE` and `ISO_WEEK_DATE` (with `import static java.time.format.DateTimeFormatter.*;`) are essentially the same.

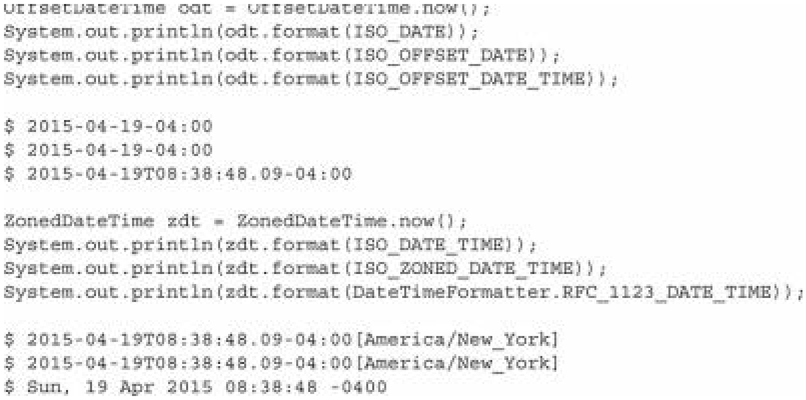
Larger View



```
import static java.time.format.DateTimeFormatter.*;
…
LocalDate ld = LocalDate.now();

System.out.println("RESULT 1: " + ld.format( DateTimeFormatter.ISO_WEEK_DATE));
System.out.println("RESULT 2: " + ld.format( ISO_WEEK_DATE));));
$ RESULT 1: 2015-W16-7
$ RESULT 2: 2015-W16-7
```

Several predefined formatters will work for different classes, such as the `OffsetDateTime` and the `ZonedDateTime` classes.

Larger View

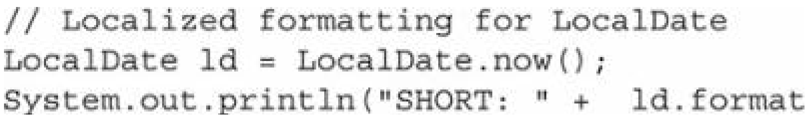


```
OffsetDateTime odt = OffsetDateTime.now();
System.out.println(odt.format(ISO_DATE));
System.out.println(odt.format(ISO_OFFSET_DATE));
System.out.println(odt.format(ISO_OFFSET_DATE_TIME));

$ 2015-04-19-04:00
$ 2015-04-19-04:00
$ 2015-04-19T08:38:48.09-04:00

ZonedDateTime zdt = ZonedDateTime.now();
System.out.println(zdt.format(ISO_DATE_TIME));
System.out.println(zdt.format(ISO_ZONED_DATE_TIME));
System.out.println(zdt.format(DateTimeFormatter.RFC_1123_DATE_TIME));

$ 2015-04-19T08:38:48.09-04:00[America/New_York]
$ 2015-04-19T08:38:48.09-04:00[America/New_York]
$ Sun, 19 Apr 2015 08:38:48 -0400
```

Localized Formatters

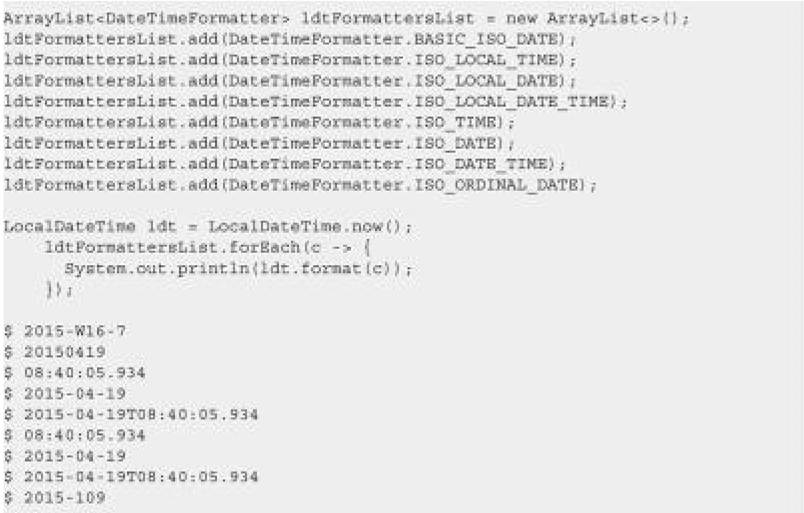
Localized `DateTimeFormatter` class formatters with static methods `ofLocalizedTime`, `ofLocalizedDate`, and `ofLocalizedDateTime` use the `FormatStyle` enum values `FormatStyle.SHORT`, `FormatStyle.MEDIUM`, `FormatStyle.LONG`, and `FormatStyle.FULL` to support localized formats. `FormatStyle.LONG` and `FormatStyle.FULL` are not on the exam.

Larger View



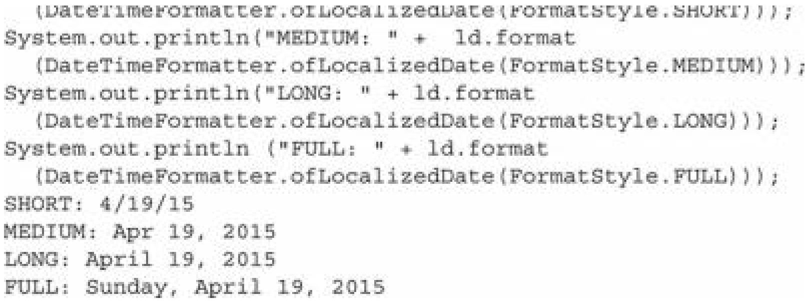
```
// Localized formatting for LocalDate
LocalDate ld = LocalDate.now();
System.out.println("SHORT: " +  ld.format
```

Exam Watch ***For the scope of the exam, you should be familiar with the formatters that are most commonly used with the `LocalDateTime` class and what the formatted results look like*.**



```
ArrayList<DateTimeFormatter> ldtFormattersList = new ArrayList<>();
ldtFormattersList.add(DateTimeFormatter.BASIC_ISO_DATE);
ldtFormattersList.add(DateTimeFormatter.ISO_LOCAL_TIME);
ldtFormattersList.add(DateTimeFormatter.ISO_LOCAL_DATE);
ldtFormattersList.add(DateTimeFormatter.ISO_LOCAL_DATE_TIME);
ldtFormattersList.add(DateTimeFormatter.ISO_TIME);
ldtFormattersList.add(DateTimeFormatter.ISO_DATE);
ldtFormattersList.add(DateTimeFormatter.ISO_DATE_TIME);
ldtFormattersList.add(DateTimeFormatter.ISO_ORDINAL_DATE);

LocalDateTime ldt = LocalDateTime.now();
    ldtFormattersList.forEach(c -> {
      System.out.println(ldt.format(c));
    });

$ 2015-W16-7
$ 20150419
$ 08:40:05.934
$ 2015-04-19
$ 2015-04-19T08:40:05.934
$ 08:40:05.934
$ 2015-04-19
$ 2015-04-19T08:40:05.934
$ 2015-109
```

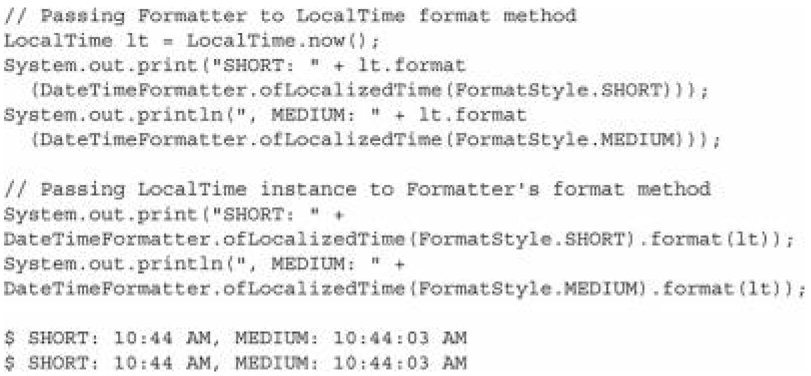
Larger View



```
  (DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)));
System.out.println("MEDIUM: " +  ld.format
  (DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM)));
System.out.println("LONG: " + ld.format
  (DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG)));
System.out.println ("FULL: " + ld.format
  (DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)));
SHORT: 4/19/15
MEDIUM: Apr 19, 2015
LONG: April 19, 2015
FULL: Sunday, April 19, 2015
```

In addition to getting the formatted value from passing the localized formatter into the format method of the calendar classes, the formatters have a format method that accepts the calendar instance to achieve the same formatted string results.

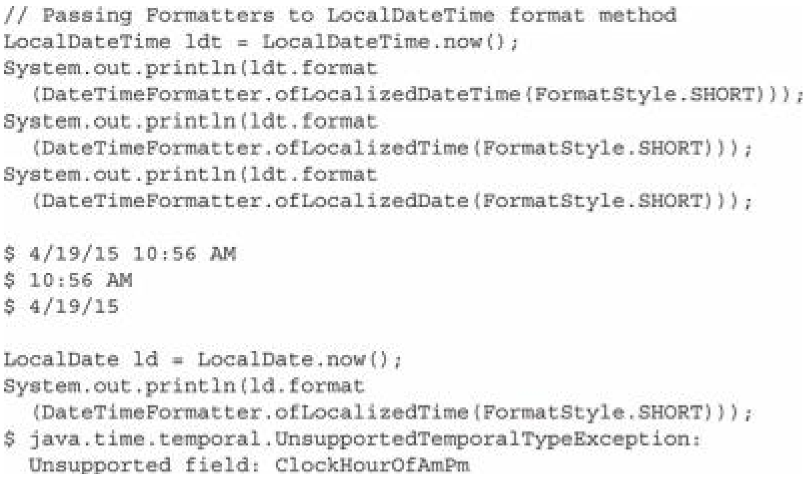
Larger View



```
// Passing Formatter to LocalTime format method
LocalTime lt = LocalTime.now();
System.out.print("SHORT: " + lt.format
  (DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)));
System.out.println(", MEDIUM: " + lt.format
  (DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM)));

// Passing LocalTime instance to Formatter's format method
System.out.print("SHORT: " +
DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT).format(lt));
System.out.println(", MEDIUM: " +
DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM).format(lt));

$ SHORT: 10:44 AM, MEDIUM: 10:44:03 AM
$ SHORT: 10:44 AM, MEDIUM: 10:44:03 AM
```

The three localized methods can be used only with the appropriate calendar classes; otherwise, an `UnsupportedTemporalTypeException` will be thrown. Also, using `FormatStyle.LONG` and `FormatStyle.FULL` where they are not accepted will result in `java.time.DateTimeException` exceptions being thrown.

Larger View

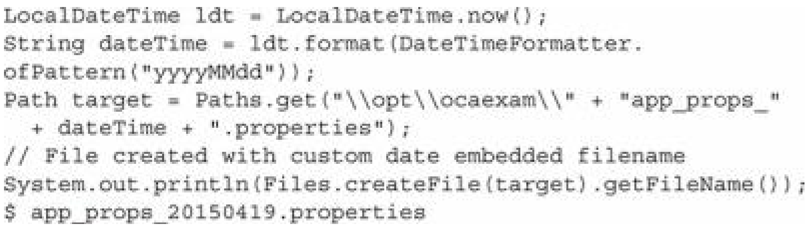


```
// Passing Formatters to LocalDateTime format method
LocalDateTime ldt = LocalDateTime.now();
System.out.println(ldt.format
  (DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT)));
System.out.println(ldt.format
  (DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)));
System.out.println(ldt.format
  (DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)));

$ 4/19/15 10:56 AM
$ 10:56 AM
$ 4/19/15

LocalDate ld = LocalDate.now();
System.out.println(ld.format
  (DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)));
$ java.time.temporal.UnsupportedTemporalTypeException:
  Unsupported field: ClockHourOfAmPm
```

Specialized Formatters

Specialized formatters allow the use of letter and symbol sequences to produce custom desired format output.

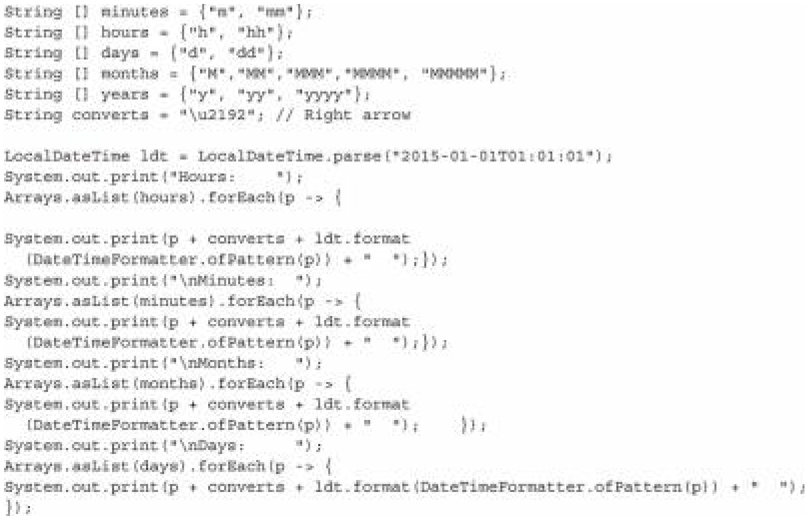
Larger View



```
LocalDateTime ldt = LocalDateTime.now();
String dateTime = ldt.format(DateTimeFormatter.
ofPattern("yyyyMMdd"));
Path target = Paths.get("\\opt\\ocaexam\\" + "app_props_"
  + dateTime + ".properties");
// File created with custom date embedded filename
System.out.println(Files.createFile(target).getFileName());
$ app_props_20150419.properties
```

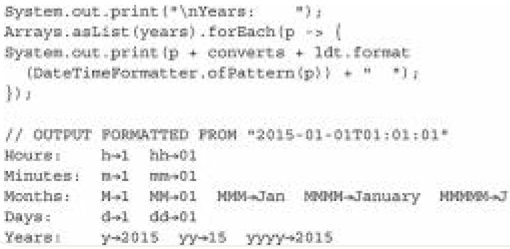
The syntax for the formatting can be found in the `DateTimeFormatter` documentation within the Java 8 Javadoc (https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html). However, for the exam, the following code example demonstrates the extent of what you will need to know, being `m`, `mm`, `h`, `hh`, `d`, `dd`, `M`, `MM`, `MMM`, `MMMM`, `MMMMM`, `y`, `yy`, `yyy`, and `yyyy`.

Larger View



```
String [] minutes = {"m", "mm"};
String [] hours = {"h", "hh"};
String [] days = {"d", "dd"};
String [] months = {"M","MM","MMM","MMMM", "MMMMM"};
String [] years = {"y", "yy", "yyyy"};
String converts = "\u2192"; // Right arrow

LocalDateTime ldt = LocalDateTime.parse("2015-01-01T01:01:01");
System.out.print("Hours:    ");
Arrays.asList(hours).forEach(p -> {

System.out.print(p + converts + ldt.format
  (DateTimeFormatter.ofPattern(p)) + "  ");});
System.out.print("\nMinutes:  ");
Arrays.asList(minutes).forEach(p -> {
System.out.print(p + converts + ldt.format
  (DateTimeFormatter.ofPattern(p)) + "  ");});
System.out.print("\nMonths:   ");
Arrays.asList(months).forEach(p -> {
System.out.print(p + converts + ldt.format
  (DateTimeFormatter.ofPattern(p)) + "  ");    });
System.out.print("\nDays:     ");
Arrays.asList(days).forEach(p -> {
System.out.print(p + converts + ldt.format(DateTimeFormatter.ofPattern(p)) + "  ");
});
```

Larger View



```
System.out.print("\nYears:    ");
Arrays.asList(years).forEach(p -> {
System.out.print(p + converts + ldt.format
  (DateTimeFormatter.ofPattern(p)) + "  ");
});

// OUTPUT FORMATTED FROM "2015-01-01T01:01:01"
Hours:    h→1  hh→01
Minutes:  m→1  mm→01
Months:   M→1  MM→01  MMM→Jan  MMMM→January  MMMMM→J
Days:     d→1  dd→01
Years:    y→2015  yy→15  yyyy→2015
```