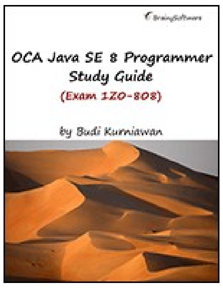

OCA Java SE 8 Programmer Study Guide (Exam 1Z0-808)

by Budi Kurniawan

Brainy Software Corp.. (c) 2015. Copying Prohibited.

Reprinted for Suresh Verma, Capgemini US LLC

suresh.verma@capgemini.com

Reprinted with permission as a subscription benefit of **Skillport**,

All rights reserved. Reproduction and/or distribution in whole or in part in electronic,paper or other forms without written permission is prohibited.

Chapter 9: Dates and Times

Overview

Support for dates and times has been available since Java version 1.0, mainly through the **java.util.Date** class. However, **Date** was poorly designed. For examples, months in **Date** start at 1 but days start at 0. A lot of its methods were deprecated in JDK 1.1 at the same time the **java.util.Calendar** was brought in to take over some of the functionality in **Date**. The duo were the main classes for dealing with dates and times, right up to JDK 1.7, even though they had been considered inadequate and not easy to work with, causing many to resort to third party alternatives such as Joda Time (http://joda.org). The new Date and Time API in JDK 1.8 resolves many of the issues in the old API and is similar to the Joda Time API.

9.1 Date and Time API Overview

The new Date and Time API makes it extremely easy to work with dates and times. The **java.time** package contains the core classes in the API. In addition, there are four other packages whose members are used less often: **java.time.chrono, java.time.format, java.time.temporal** and **java.time.zone**.

Within the **java.time** package, the **Instant** class represents a point on the time-line and is often used to time an operation. The **LocalDate** class models a date without the time component and time zone, suitable, for example, to represent a birthday.

If you need a date as well as a time, then **LocalDateTime** is for you. For instance, an order shipping date probably requires a time in addition to a date to make the order easier to track. If you need a time but do not care about the date, then you can use **LocalTime**.

On top of that, if a time zone is important, the Date and Time API provides the **ZonedDateTime** class. As the name implies, this class models a date-time with a time zone. For instance, you can use this class to calculate the flight time between two airports located in different time zones.

Then there are two classes for measuring an amount of time, **Duration** and **Period**. These two are similar except that **Duration** is time-based and **Period** is date-based. **Duration** provides a quantity of time to nanosecond precision. This class is good, for example, to model a flight time as it is often given in the number of hours and minutes. On the other hand, **Period** is suitable when you are only concerned with the number of days, months or years, such as when calculating your father's age.

The **java.time** package also comes with two enums, **DayOfWeek** and **Month. DayOfWeek** represents the day of the week, from **MONDAY** to **SUNDAY**. The **Month** enum represents the twelve months of the year, from **JANUARY** to **DECEMBER**.

Working with dates and times frequently involves parsing and formatting. The Date and Time API addresses these two issues by providing **parse** and **format** methods in all its major classes. In addition, the **java.time.format** contains a **DateTimeFormatter** class for formatting dates and times.

9.2 The Instant Class

An **Instant** object represents a point on the time-line. The reference point is the standard Java epoch, which is 1970-01-01T00:00:00Z (January 1, 1970 00:00 GMT). The **EPOCH** field of the **Instant** class returns an **Instant** representing the Java epoch. Instants after the epoch have positive values and instants prior to that have negative values.

The static method **now** of **Instant** returns an **Instant** object that represents the current time:

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

The **getEpochSecond** method returns the number of seconds that have elapsed since the epoch. The **getNano** method returns the number of nanoseconds since the beginning of the last second.

A popular use of the **Instant** class is to time an operation, as demonstrated in Listing 9.1.

Listing 9.1: Using Instant to time an operation

```
package app09;
import java.time.Duration;
import java.time.Instant;

public class InstantDemo1 {
    public static void main(String[] args) {
```

Reprinted for Q4OGY/1412663, Capgemini US LLC Brainy Software Corp., Brainy Software Inc. (c) 2015, Copying Prohibited

```
        Instant start = Instant.now();
        // do something here
        Instant end = Instant.now();
        System.out.println(Duration.between(start, end).toMillis());
    }
}
```

As shown in Listing 9.1, the **Duration** class is used to return the difference between two **Instant**s. You will learn more about **Duration** later in this chapter.

9.3 LocalDate

The **LocalDate** class models a date without a time component. It also has no time zone. Table 9.1 shows some of the more important methods in **LocalDate**.

Table 9.1: More important methods of LocalDate

Method	Description
now	A static method that returns today's date.
of	A static method that creates a **LocalDate** from the specified year, month and date.
getDayOfMonth, getMonthValue, getYear	Returns the day, month or year part of this **LocalDate** as an **int**.
getMonth	Returns the month of this **LocalDate** as a **Month** enum constant.
plusDays, minusDays	Adds or subtracts the given number of days to or from this **LocalDate**.
plusWeeks, minusWeeks	Adds or subtracts the given number of weeks to or from this **LocalDate**.
plusMonths, minusMonths	Adds or subtracts the given number of months to or from this **LocalDate**.
plusYears, minusYears	Adds or subtracts the given number of years to or from this **LocalDate**.
isLeapYear	Checks if the year specified by this **LocalDate** is a leap year.
isAfter, isBefore	Checks if this **LocalDate** is after or before the given date.
lengthOfMonth	Returns the number of days in the month in this **LocalDate**.
withDayOfMonth	Returns a copy of this **LocalDate** with the day of month set to the given value.
withMonth	Returns a copy of this **LocalDate** with the month set to the given value.
withYear	Returns a copy of this **LocalDate** with the year set to the given value.

LocalDate offers various methods to create a date. For example, to create a **LocalDate** that represents today's date, use the static method **now**.

```
LocalDate today = LocalDate.now();
```

To create a **LocalDate** that represents a specific year, month and day, use its **of** method, which is also static. For instance, the following snippet creates a **LocalDate** that represents December 31, 2015.

```
LocalDate endOfYear = LocalDate.of(2015, 12, 31);
```

The method **of** has another override that accepts a constant of the **java.time.Month** enum as the second argument. For example, here is the code to construct the same date using the second method override.

```
LocalDate endOfYear = LocalDate.of(2015, Month.DECEMBER, 31);
```

There are also methods for obtaining the day, month or year of a **LocalDate**, such as **getDayOfMonth, getMonth, getMonthValue** and **getYear**. They all do not take any argument and either return an **int** or a **Month** enum constant. In addition, there is a **get** method that takes a **TemporalField** and returns a part of this **LocalDate**. For example, passing **ChronoField.YEAR** to **get** returns the year component of a **LocalDate**.

```
int year = localDate.get(ChronoField.YEAR));
```

ChronoField is an enum that implements the **TemporalField** interface, therefore you can pass a **ChronoField** constant to **get.** Both **TemporalField** and **ChronoField** are part of the **java.time.temporal** package. However, not all constants in

Reprinted for Q4OGY/1412663, Capgemini US LLC
Brainy Software Corp., Brainy Software Inc. (c) 2015, Copying Prohibited

ChronoField can be passed to **get** as not all of them are supported. For example, passing **ChronoField.SECOND_OF_DAY** to **get** throws an exception. As such, instead of **get**, it is better to use **getMonth, getYear** or a similar method to obtain a component of a **LocalDate**.

In addition, there are methods for copying a **LocalDate**, such as **plusDays, plusYears, minusMonths**, and so on. For example, to get a **LocalDate** that represents tomorrow, you can create a **LocalDate** that represents today and then calls its **plusDays** method.

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
```

To get a **LocalDate** that represents yesterday, you can use the **minusDays** method.

```
LocalDate yesterday = LocalDate.now().minusDays(1);
```

In addition, there are **plus** and **minus** methods to get a copy of a **LocalDate** in a more generic way. Both accept an **int** and a **TemporalUnit**. The signatures of these methods are as follows.

```
public LocalDate plus(long amountToAdd,
        java.time.temporal.TemporalUnit unit)

public LocalDate minus(long amountToSubtract,
        java.time.temporal.TemporalUnit unit)
```

As an example, to get a **LocalDate** that represents a past date exactly two decades ago from today, you can use this code.

```
LocalDate pastDate = LocalDate.now().minus(2, ChronoUnit.DECADES);
```

ChronoUnit is an enum that implements **TemporalUnit**, so you can pass a **ChronoUnit** constant to the **plus** or **minus** method.

A **LocalDate** is immutable and therefore cannot be changed. Any method that returns a **LocalDate** returns a new instance of **LocalDate**.

Listing 9.2 shows an example of **LocalDate**.

Listing 9.2: LocalDate example

```
package app09;
import java.time.LocalDate;
import java.time.temporal.ChronoField;
import java.time.temporal.ChronoUnit;

public class LocalDateDemo1 {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();
        LocalDate tomorrow = today.plusDays(1);
        LocalDate oneDecadeAgo = today.minus(1,
                ChronoUnit.DECADES);
        System.out.println("Day of month: "
                + today.getDayOfMonth());
        System.out.println("Today is " + today);
        System.out.println("Tomorrow is " + tomorrow);
        System.out.println("A decade ago was " + oneDecadeAgo);
        System.out.println("Year : "
                + today.get(ChronoField.YEAR));
        System.out.println("Day of year:" + today.getDayOfYear());
    }
}
```

9.4 Period

The **Period** class models a date-based amount of time, such as five days, a week or three years. Some of its more important methods are presented in Table 9.2.

Table 9.2: More important methods of Period

Method	Description
between	Creates a **Period** between two **LocalDates**.
ofDays, ofWeeks,	Creates a **Period** representing the given number of days/weeks/months/years.

Table 9.2: More important methods of Period

Method	Description
ofMonths, ofYears	
of	Creates a **Period** from the given numbers of years, months and days.
getDays, getMonths, getYears	Returns the number of days/months/years of this period as an **int**.
isNegative	Returns **true** if any of the three components of this **Period** is negative. Returns **false** otherwise.
isZero	Returns **true** if all of the three components of this **Period** are zero. Otherwise, returns **false**.
plusDays, minusDays	Adds or subtracts the given number of days to or from this **Period**.
plusMonths, minusMonths	Adds or subtracts the given number of months to or from this **Period**.
plusYears, minusYears	Adds or subtracts the given number of years to or from this **Period**.
withDays	Returns a copy of this **Period** with the specified number of days.
withMonths	Returns a copy of this **Period** with the specified number of months.
withYears	Returns a copy of this **Period** with the specified number of years.

Creating a **Period** is easy, thanks to the static factory methods **between, of**, and **ofDays/ofWeeks/ofMonths/ofYears**. For example, here is how you create a **Period** representing two weeks.

```
Period twoWeeks = Period.ofWeeks(2);
```

To create a **Period** representing one year, two months and three days, use the **of** method.

```
Period p = Period.of(1, 2, 3);
```

To obtain the year/month/day component of a **Period**, call its **getYears/getMonths/getDays** method. For instance, the **howManyDays** variable in the following code snippet will have a value of 14.

```
Period twoWeeks = Period.ofWeeks(2);
int howManyDays = twoWeeks.getDays();
```

Finally, you can create a copy of a **Period** using the **plus*XXX*** or **minus*XXX*** methods as well as one of the **with*XXX*** methods. A **Period** is immutable so these methods return new **Period** instances.

As an example, the code in Listing 9.3 shows an age calculator that calculates a person's age. It creates a **Period** from two **LocalDate**s and calls its **getDays, getMonths**, and **getYears** methods.

Listing 9.3: Using Period

```
package app09;
import java.time.LocalDate;
import java.time.Period;

public class PeriodDemo1 {
    public static void main(String[] args) {
        LocalDate dateA = LocalDate.of(1978, 8, 26);
        LocalDate dateB = LocalDate.of(1988, 9, 28);
        Period period = Period.between(dateA, dateB);
        System.out.printf("Between %s and %s"
                + " there are %d years, %d months"
                + " and %d days%n", dateA, dateB,
                period.getYears(),
                period.getMonths(),
                period.getDays());
    }
}
```

When run, the **PeriodDemo1** class in Listing 9.3 will print this string.

```
Between 1978-08-26 and 1988-09-28 there are 10 years, 1 months and 2
days
```

9.5 LocalDateTime

Reprinted for Q4OGY/1412663, Capgemini US LLC
Brainy Software Corp., Brainy Software Inc. (c) 2015, Copying Prohibited

The **LocalDateTime** class models a date-time without a time zone. Table 9.3 shows some of the more important methods in **LocalDateTime**. The methods are similar to those of **LocalDate** plus some other methods for modifying the time component, such as **plusHours, plusMinutes** and **plusSeconds**, that are not available in **LocalDate**.

Table 9.3: More important methods of LocalDateTime

Method	Description
now	A static method that returns the current date and time.
of	A static method that creates a **LocalDateTime** from the specified year, month, date, hour, minute, second and millisecond.
getYear, getMonthValue, getDayOfMonth, getHour, getMinute, getSecond	Returns the year, month, day, hour, minute or second part of this **LocalDateTime** as an **int**.
plusDays, minusDays	Adds or subtracts the given number of days to or from the current **LocalDateTime**.
plusWeeks, minusWeeks	Adds or subtracts the given number of weeks to or from the current **LocalDateTime**.
plusMonths, minusMonths	Adds or subtracts the given number of months to or from the current **LocalDateTime**.
plusYears, minusYears	Adds or subtracts the given number of years to or from the current **LocalDateTime**.
plusHours, minusHours	Adds or subtracts the given number of hours to or from the current **LocalDateTime**.
plusMinutes, minusMinutes	Adds or subtracts the given number of minutes to or from the current **LocalDateTime**.
plusSeconds, minusSeconds	Add or subtracts the given number of seconds to or from the current **LocalDateTime**.
IsAfter, isBefore	Checks if this **LocalDateTime** is after or before the given date-time.
withDayOfMonth	Returns a copy of this **LocalDateTime** with the day of month set to the given value.
withMonth, withYear	Returns a copy of this **LocalDateTime** with the month or year set to the given value.
withHour, withMinute, withSecond	Returns a copy of this **LocalDateTime** with the hour/minute/second set to the given value.

LocalDateTime offers various static methods to create a date-time. The method **now** comes with three overrides and return the current date-time. The no-argument override is the easiest to use:

```
LocalDateTime now = LocalDateTime.now();
```

To create a **LocalDateTime** with a specific date and time, use the **of** method. This method has a number of overrides and allows you to pass the individual component of a date-time or a **LocalDate** and a **LocalTime**. Here are the signatures of some of the **of** methods.

```
public static LocalDateTime of(int year, int month, int dayOfMonth,
          int hour, int minute)

public static LocalDateTime of(int year, int month, int dayOfMonth,
          int hour, int minute)

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)

public static LocalDateTime of(LocalDate date, LocalTime time)
```

For instance, the following snippet creates a **LocalDateTime** that represents December 31, 2015 at eight o'clock in the morning.

```
LocalDateTime endOfYear = LocalDateTime.of(2015, 12, 31, 8, 0);
```

You can create a copy of a **LocalDateTime** using the **plus*XXX*** or **minus*XXX*** method. For example, this code creates a

Reprinted for Q4OGY/1412663, Capgemini US LLC
Brainy Software Corp., Brainy Software Inc. (c) 2015, Copying Prohibited

LocalDateTime that represents the same time tomorrow.

```
LocalDateTime now = LocalDateTime.now();
LocalDateTime sameTimeTomorrow = now.plusHours(24);
```

9.6 Time Zones

The Internet Assigned Numbers Authority (IANA) maintains a database of time zones that you can download from this web page:

```
http://www.iana.org/time-zones
```

For easy viewing, however, you can just visit this Wikipedia page:

```
http://en.wikipedia.org/wiki/List_of_tz_database_time_zones
```

The Java Date and Time API caters for time zones too. The abstract class **ZoneId** (in the **java.time** package) represents a zone identifier. It has a static method called **getAvailableZoneIds** that returns all zone identifiers. Listing 9.4 shows how you can print a sorted list of all time zones using this method.

Listing 9.4: Listing all zone identifiers

```
package app09;
import java.time.ZoneId;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Set;

public class TimeZoneDemo1 {
    public static void main(String[] args) {
        Set<String> allZoneIds = ZoneId.getAvailableZoneIds();
        List<String> zoneList = new ArrayList<>(allZoneIds);

        Collections.sort(zoneList);
        for (String zoneId : zoneList) {
            System.out.println(zoneId);
        }
        // alternatively, you can use this line of code to
        // print a sorted list of zone ids
        // ZoneId.getAvailableZoneIds().stream().sorted().
        //        forEach(System.out::println);
    }
}
```

getAvailableZoneIds returns a **Set** of **String**s. You can sort the **Set** using **Collections.sort()** or more elegantly by calling its **stream** method. You could have written this code to sort the zone identifiers.

```
ZoneId.getAvailableZoneIds().stream().sorted()
        .forEach(System.out::println);
```

Chapter 20, "Working with Streams" explains what streams are.

getAvailableZoneIds returns a **Set** of 586 zone identifiers. Here are some of the zone identifiers from the code above.

```
Africa/Cairo
Africa/Johannesburg
America/Chicago
America/Los_Angeles
America/Mexico_City
America/New_York
America/Toronto
Antarctica/South_Pole
Asia/Hong_Kong
Asia/Shanghai
Asia/Tokyo
Australia/Melbourne
Australia/Sydney
Canada/Atlantic
Europe/Amsterdam
Europe/London
Europe/Paris
```

Reprinted for Q4OGY/1412663, Capgemini US LLC Brainy Software Corp., Brainy Software Inc. (c) 2015, Copying Prohibited

```
US/Central
US/Eastern
US/Pacific
```

9.7 ZonedDateTime

The **ZonedDateTime** class models a date-time with a time zone. For example, the following is a zoned date-time:

```
2015-12-31T10:59:59+01:00 Europe/Paris
```

A **ZonedDateTime** is always immutable and the time component is stored to nanosecond precision.

Table 9.4 shows the more important methods in **ZonedDateTime**.

Table 9.4: More important methods of ZonedDateTime

Method	Description
now	A static method that returns the current date and time of the system's default zone.
of	A static method that creates a **ZonedDateTime** from the specified datetime and zone identifier.
getYear, getMonthValue, getDayOfMonth, getHour, getMinute, getSecond, getNano	Returns the year, month, day, hour, minute, second or nanosecond part of this **ZoneDateTime** as an **int**.
plusDays, minusDays	Adds or subtracts the given number of days to or from the current **ZonedDateTime**.
plusWeeks, minusWeeks	Adds or subtracts the given number of weeks to or from the current **ZonedDateTime**.
plusMonths, minusMonths	Adds or subtracts the given number of months to or from the current **ZonedDateTime**.
plusYears, minusYears	Adds or subtracts the given number of years to or from the current **ZonedDateTime**.
plusHours, minusHours	Adds or subtracts the given number of hours to or from the current **ZonedDateTime**.
plusMinutes, minusMinutes	Adds or subtracts the given number of minutes to or from the current **ZonedDateTime**.
plusSeconds, minusSeconds	Add or subtracts the given number of seconds to or from the current **ZonedDateTime**.
IsAfter, isBefore	Checks if this **ZonedDateTime** is after or before the given zoned datetime.
getZone	Returns the zone ID of this **ZonedDateTime**.
withYear, withMonth, withDayOfMonth	Returns a copy of this **ZonedDateTime** with the year/month/day of month set to the given value.
withHour, withMinute, withSecond	Returns a copy of this **ZonedDateTime** with the hour/minute/second set to the given value.
withNano	Returns a copy of this **ZonedDateTime** with the nanosecond set to the given value.

Like **LocalDateTime**, the **ZonedDateTime** class offers the static methods **now** and **of** to construct a **ZonedDateTime. now** creates a **ZonedDateTime** representing the date and time of execution. The no-argument override of **now** creates a **ZonedDateTime** with the computer's default time zone.

```
ZonedDateTime now = ZonedDateTime.now();
```

Another override of **now** lets you pass a zone identifier:

```
ZonedDateTime parisTime =
        ZonedDateTime.now(ZoneId.of("Europe/Paris"));
```

The method **of** also comes with several overrides. In all cases, you need to pass a zone identifier. The first override allows you to pass each component of a zoned date-time, from the year to the nanosecond.

```
public static ZonedDateTime of(int year, int month, int dayOfMonth,
```

Reprinted for Q4OGY/1412663, Capgemini US LLC
Brainy Software Corp., Brainy Software Inc. (c) 2015, Copying Prohibited

```
        int hour, int minute, int second, int nanosecond,
        ZoneId zone)
```

The second override of **of** takes a **LocalDate**, a **LocalTime** and a **ZoneId:**

```
public static ZonedDateTime of(LocalDate date, LocalTime time,
        ZoneId zone)
```

The last override of of takes a **LocalDateTime** and a **ZoneId**.

```
public static ZonedDateTime of(LocalDateTime datetime, ZoneId zone)
```

Like **LocalDate** and **LocalDateTime, ZonedDateTime** offers methods to create a copy of an instance using the **plus*XXX***, **minus*XXX*** and **with*XXX*** methods.

For instance, these lines of code creates a **ZonedDateTime** with the default time zone and calls its **minusDays** method to create the same **ZonedDateTime** three days earlier.

```
ZonedDateTime now = ZonedDateTime.now();
ZonedDateTime threeDaysEarlier = now.minusDays(3);
```

9.8 Duration

The **Duration** class models a time-based duration. It is similar to **Period** except that a **Duration** has a time component to nanosecond precision and takes into account the time zones between **ZonedDateTime**s. Table 9.5 shows the more important methods in **Duration**.

Table 9.5: More important methods in Duration

Method	Description
between	Creates a **Duration** between two temporal objects, such as between two **LocalDateTime**s or two **LocalZonedDateTime**s.
ofYears, ofMonths, ofWeeks, ofDays, ofHours, ofMinutes, ofSeconds, ofNano	Creates a **Duration** representing the given number of years/months/weeks/days/hours/minutes/seconds/nanoseconds.
of	Creates a **Duration** from the given number of temporal units.
toDays, toHours, toMinutes	Returns the number of days/hours/minutes of this **Duration** as an int.
isNegative	Returns **true** if this **Duration** is negative. Returns **false** otherwise.
isZero	Returns **true** if this **Duration** is zero length. Otherwise, returns **false**.
plusDays, minusDays	Adds or subtracts the given number of days to or from this **Duration**.
plusMonths, minusMonths	Adds or subtracts the given number of months to or from this **Duration**.
plusYears, minusYears	Adds or subtracts the given number of years to or from this **Duration**.
withSeconds	Returns a copy of this **Duration** with the specified number of seconds.

You can create a **Duration** by calling its **between** or **of** static method. The code in Listing 9.5 creates a **Duration** between two **LocalDateTime**s, between January 26, 2015 11:10 and January 26, 2015 12:40.

Listing 9.5: Creating a Duration between two LocalDateTimes

```
package app09;
import java.time.Duration;
import java.time.LocalDateTime;

public class DurationDemo1 {

    public static void main(String[] args) {
        LocalDateTime dateTimeA = LocalDateTime
                .of(2015, 1, 26, 8, 10, 0, 0);
        LocalDateTime dateTimeB = LocalDateTime
                .of(2015, 1, 26, 11, 40, 0, 0);
        Duration duration = Duration.between(
                dateTimeA, dateTimeB);
```

Reprinted for Q4OGY/1412663, Capgemini US LLC
Brainy Software Corp., Brainy Software Inc. (c) 2015, Copying Prohibited

```
        System.out.printf("There are %d hours and %d minutes.%n",
                duration.toHours(),
                duration.toMinutes() % 60);
    }
}
```

The result of running the **DurationDemo1** class is this.

```
There are 3 hours and 30 minutes.
```

The code in Listing 9.6 creates a **Duration** between two **ZoneDateTime**s, with the same date-time but different timezones.

Listing 9.6: Creating a Duration between two ZonedDateTimes

```
package app09;
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class DurationDemo2 {

    public static void main(String[] args) {
        ZonedDateTime zdt1 = ZonedDateTime.of(
                LocalDateTime.of(2015, Month.JANUARY, 1,
                        8, 0),
                ZoneId.of("America/Denver"));
        ZonedDateTime zdt2 = ZonedDateTime.of(
                LocalDateTime.of(2015, Month.JANUARY, 1,
                        8, 0),
                ZoneId.of("America/Toronto"));
        Duration duration = Duration.between(zdt1, zdt2);
        System.out.printf("There are %d hours and %d minutes.%n",
               duration.toHours(),
               duration.toMinutes() % 60);
    }
}
```

Running the **DurationDemo2** class prints this on the console.

```
There are -2 hours and 0 minutes.
```

This is expected, because there are two hours difference between the time zones America/Denver and America/Toronto.

As a more complex example, the code in Listing 9.7 shows a bus travel time calculator. It has one method, **calculateTravelTime**, which takes a departure **ZonedDateTime** and an arrival **ZonedDateTime**. The code calls the **calculateTravelTime** method twice. Both times the bus departs from Denver, Colorado at 8 in the morning Denver time and arrives in Toronto at 8 in the next morning Toronto time. The first time the bus leaves on March 8, 2014 and the second time it leaves on March 18, 2014.

What are the travel time in both occasions?

Listing 9.7: Travel time calculator

```
package app09;
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class TravelTimeCalculator {

    public Duration calculateTravelTime(
            ZonedDateTime departure, ZonedDateTime arrival) {
        return Duration.between(departure, arrival);
    }

    public static void main(String[] args) {
        TravelTimeCalculator calculator =
```

Reprinted for Q4OGY/1412663, Capgemini US LLC Brainy Software Corp., Brainy Software Inc. (c) 2015, Copying Prohibited

```
                new TravelTimeCalculator();
        ZonedDateTime departure1 = ZonedDateTime.of(
                LocalDateTime.of(2014, Month.MARCH, 8,
                        8, 0),
                ZoneId.of("America/Denver"));
        ZonedDateTime arrival1 = ZonedDateTime.of(
                LocalDateTime.of(2014, Month.MARCH, 9,
                        8, 0),
                ZoneId.of("America/Toronto"));
        Duration travelTime1 = calculator
                .calculateTravelTime(departure1, arrival1);
        System.out.println("Travel time 1: "
                + travelTime1.toHours() + " hours");

        ZonedDateTime departure2 = ZonedDateTime.of(
                LocalDateTime.of(2014, Month.MARCH, 18,
                        8, 0),
                ZoneId.of("America/Denver"));
        ZonedDateTime arrival2 = ZonedDateTime.of(
                LocalDateTime.of(2014, Month.MARCH, 19,
                        8, 0),
                ZoneId.of("America/Toronto"));
        Duration travelTime2 = calculator
                .calculateTravelTime(departure2, arrival2);
        System.out.println("Travel time 2: "
                + travelTime2.toHours() + " hours");
    }
}
```

The result is this.

```
Travel time 1: 21 hours
Travel time 2: 22 hours
```

Why the difference? Because in 2014 daylight saving time began on Sunday, March 9 at 2AM. As such, you 'lost' one hour between March 8, 2014 and March 9, 2014.

9.9 Formatting A Date-Time

You use a **java.time.format.DateTimeFormatter** to format a local or zoned date-time. The **LocalDate, LocalDateTime, LocalTime** and **ZoneDateTime** classes offer a **format** method that has the following signature.

```
public java.lang.String format(java.time.format.DateTimeFormatter
        formatter)
```

It is clear that to format a date or time, you must first create an instance of **DateTimeFormatter**.

The code in Listing 9.8 formats the current date using two formatters.

Listing 9.8: Formatting dates

```
package app09;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.format.FormatStyle;

public class DateTimeFormatterDemo1 {
    public static void main(String[] args) {
        DateTimeFormatter formatter1 = DateTimeFormatter
                .ofLocalizedDateTime(FormatStyle.MEDIUM);
        LocalDateTime example = LocalDateTime.of(
                2000, 3, 19, 10, 56, 59);
        System.out.println("Format 1: " + example
                .format(formatter1));
        DateTimeFormatter formatter2 = DateTimeFormatter
                .ofPattern("MMMM dd, yyyy HH:mm:ss");
        System.out.println("Format 2: " +
                example.format(formatter2));
    }
}
```

The results are as follows (the first result depends on your locale).

```
Format 1: 19-Mar-2000 10:56:59 AM
Format 2: March 19, 2000 10:56:59
```

9.10 Parsing A Date-Time

There are two **parse** methods in many of the classes in the Java Date and Time API. The first requires a formatter, the second does not. The one that does not will parse the date-time based on the default pattern. To use your own pattern, use a **DateTimeFormatter**. The **parse** methods will throw a **DateTimeParseException** if the string passed cannot be parsed.

Listing 9.9 contains an age calculator to demonstrate date parsing.

Listing 9.9: An age calculator

```java
package app09;
import java.time.LocalDate;
import java.time.Period;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.Scanner;

public class AgeCalculator {
    DateTimeFormatter formatter = DateTimeFormatter
            .ofPattern("yyyy-M-d");
    public Period calculateAge(LocalDate birthday) {
        LocalDate today = LocalDate.now();
        return Period.between(birthday, today);
    }

    public LocalDate getBirthday() {
        Scanner scanner = new Scanner(System.in);
        LocalDate birthday;
        while (true) {
            System.out.println("Please enter your birthday "
                    + "in yyyy-MM-dd format (e.g. 1980-9-28): ");
            String input = scanner.nextLine();
            try {
                birthday = LocalDate.parse(input, formatter);
                return birthday;
            } catch(DateTimeParseException e) {
                System.out.println("Error! Please try again");
            }
        }
    }

    public static void main(String[] args) {
        AgeCalculator ageCalculator = new AgeCalculator();
        LocalDate birthday = ageCalculator.getBirthday();
        Period age = ageCalculator.calculateAge(birthday);
        System.out.printf("Today you are %d years, %d months"
                + " and %d days old%n",
                age.getYears(), age.getMonths(), age.getDays());
    }
}
```

The **AgeCalculator** class has two methods, **getBirthday** and **calculateAge**. The **getBirthday** method employs a **Scanner** to read user input and parses the input into a **LocalDate** using the class level **DateTimeFormatter**. The **getBirthday** method keeps begging for a date until the user types in a date in the correct format, in which case the method returns. The **calculateAge** method takes a birthday and creates a **Period** between the birthday and today.

If you run this example, you will see this on your console.

```
Please enter your birthday in yyyy-MM-dd format (e.g. 1980-9-28):
```

If you enter a date in the correct format, the program will print the calculated age, such as the following.

```
Today you are 79 years, 0 months and 15 days old
```

Self Test

1 Question

Reprinted for Q4OGY/1412663, Capgemini US LLC
Brainy Software Corp., Brainy Software Inc. (c) 2015, Copying Prohibited

Which of the following is part of the new Date and Time API added in Java 8?

A. java.util.Date

B. java.util.Time

C. java.time.LocalDate

D. java.time.LocalDateTime

E. java.time.Period

2 Question ?

If today was January 1, 2015, what would the code below print?

```
LocalDate localDate = LocalDate.now();
localDate = localDate.withMonth(6).withDayOfMonth(30);
localDate = localDate.plusMonths(-1);
System.out.println(localDate.getMonth());
```

A. JANUARY

B. DECEMBER

C. JUNE

D. MAY

3 Question ?

What is the output of this code?

```
LocalDate localDate = LocalDate.of(2008, 3, 30);
localDate = localDate.minusMonths(1);
System.out.println(localDate);
```

A. 2008-02-31

B. 2008-02-30

C. 2008-02-29

D. 2008-02-28

E. 2008-02-27

4 Question ?

Which of the following expressions return 10 o'clock? (Choose all that apply)

A. LocalTime.NOON.minus(2, ChronoUnit.HOURS)

B. LocalTime.of(10, 00)

C. LocalTime.of(22, 00)

D. LocalTime.of(10, 0, 0)

5 Question ?

What can be inserted into line 2 so that the code always prints true?

```
1. LocalDateTime dt1 = LocalDateTime.now();
2. ...
3. System.out.println(dt2.equals(dt1));
```

A. LocalDateTime dt2 = LocalDateTime.of(LocalDate.now(), dt1.toLocalTime());

B. LocalDateTime dt2 = LocalDateTime.of(LocalDate.now(), LocalTime.now());

C. LocalDateTime dt2 = LocalDateTime.of(dt1.toLocalDate, dt1.toLocalTime());

D. LocalDateTime dt2 = LocalDateTime.now();

6 Question ?

Reprinted for Q4OGY/1412663, Capgemini US LLC Brainy Software Corp., Brainy Software Inc. (c) 2015, Copying Prohibited

What is the output of the following code?

```
Period period = Period.of(0, 0, 1);
LocalDate today = LocalDate.now();
for (int i = 0; i < 30; i++) {
    System.out.println(today.plus(period));
}
```

A. It prints 30 dates from the first of this month to the 30th

B. It prints 30 dates starting from today's date

C. It prints today's date 30 times

D. It prints 30 dates from the last day of this month

7 Question

?

What is the output of the following code?

```
DateTimeFormatter formatter1 = DateTimeFormatter
        .ofLocalizedDateTime(FormatStyle.SHORT);
LocalDateTime date1 = LocalDateTime.of(
        2000, 1, 1, 0, 0, 0);
System.out.println(date1.format(formatter1));
```

A. 2000-1-1 00:00 AM

B. 2000/1/1 00:00 AM

C. 2000/01/01 00:00

D. The output cannot be determined

8 Question

?

How do you parse a local date using the new Date and Time API?

A. By using DateTimeFormatter.parse()

B. By using DateTimeFormatter.format()

C. By using DateTimeParser.parse()

D. By using LocalDate.parse()

9 Question

?

Which pattern can be used to parse the string "2015 4 4" as April 4, 2015?

A. yyyy MM dd

B. yyyy M dd

C. yyyy MM d

D. yyyy M d

10 Question

?

What is the output of the following code?

```
    LocalDate date1 = LocalDate.of(2016, 12, 20);
    LocalDate date2 = LocalDate.of(2016, 11, 21);
    Period period = Period.between(date1, date2);
    System.out.println(period.getDays());
```

A. 0

B. 1

C. 30

D. -29

Reprinted for Q4OGY/1412663, Capgemini US LLC
Brainy Software Corp., Brainy Software Inc. (c) 2015, Copying Prohibited

Answers

1 **C, D, E**.

A is incorrect because **java.util.Date** has been around since Java 1.

B is incorrect because **java.util.Time** does not exist.

C, D and E are correct because **LocalDate, LocalDateTime** and **Period** are some of the classes in the new Date and Time API.

2 **D**.

The code will always print **MAY** no matter when it is run. This is because **withMonth(6)** sets the date to June and **plusMonths(-1)** changes it to May.

3 **C**.

Since 2008 was a leap year, moving March 30, 2008 back by one month returns February 29, 2008.

4 **A, B, C, D**.

A, B, D return a time representing 10 am. C returns a time representing 10pm.

5 **C**.

A LocalDateTime can be created from a LocalDate and a LocalTime.

Option A would be correct most of the time because calling LocalDate.now() to create dt1 and calling LocalDate.now() to create dt2 would almost always return the same date, except when the code is run close to midnight, in which case there is a slim chance the second LocalDate.now() might return a different date. Therefore, there is no guarantee dt2.equals(dt1) returns true.

B is incorrect because calling LocalTime.now() twice will return different times.

C. is correct because dt2 is constructed from components of dt1.

D is incorrect because calling LocalDateTime.now() twice returns different local date times.

6 **C**.

This is a trick question. Calling LocalDate.plus(period) does not change the LocalDate. To have a different date, the expression must be assigned to itself:

```
today = today.plus(period)
```

7 **D**.

The **ofLocalizedDateTime** method return a **DateTimeFormatter** that depends on the computer settings. Therefore, the output cannot be determined.

8 **D**.

A is incorrect because **DateTimeFormatter** does not have a parse method.

B is incorrect because **DateTimeFormatter.format()** formats a date.

C is incorrect because **DateTimeParser** is not part of the Date and Time API.

D is correct.

9 **D**.

A is incorrect because with this pattern the date would have to presented as 2015 04 04.

B is incorrect because with this pattern the date would have to be presented as 2015 4 04

C is incorrect because with this pattern the date would have to be presented as 2014 04 4.

D is the correct answer.

10 **D**.

The **between** method calculates the period between the first argument and the second argument. Since **date2** is earlier than **date2**, the result must be negative.

Reprinted for Q4OGY/1412663, Capgemini US LLC
Brainy Software Corp., Brainy Software Inc. (c) 2015, Copying Prohibited