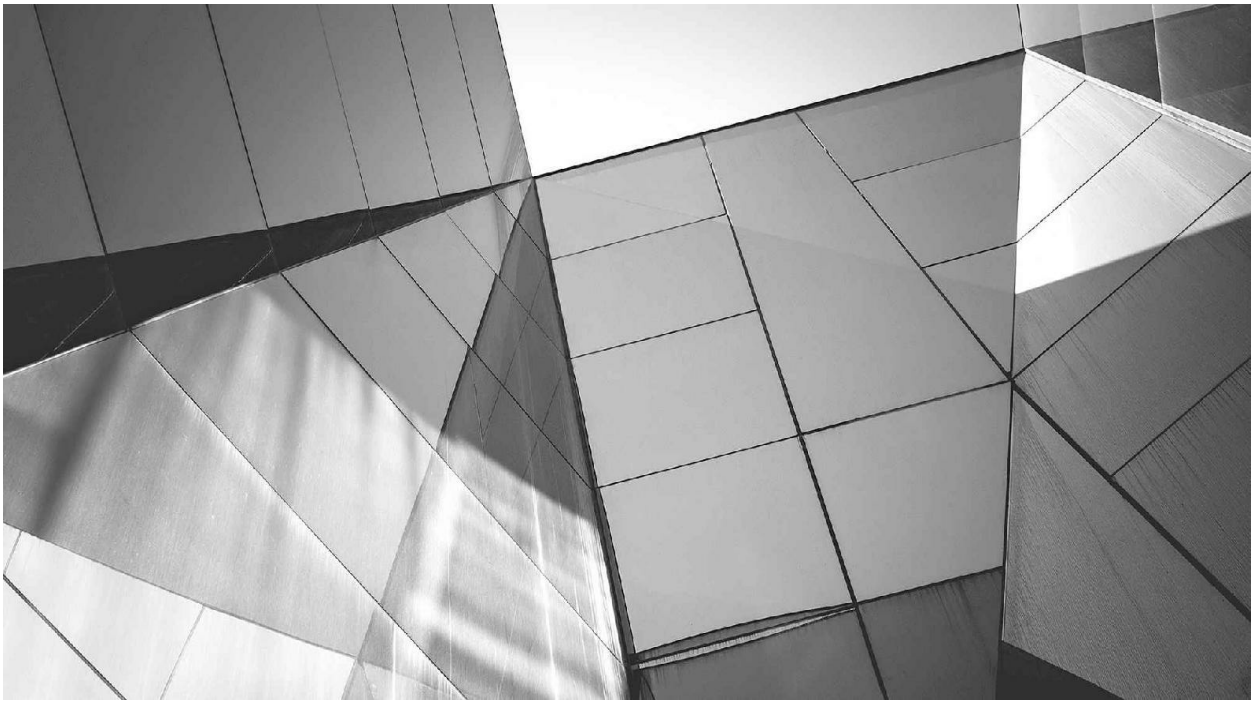

4

Dates, Times, Locales, and Resource Bundles

CERTIFICATION OBJECTIVES

- Create and Manage Date-Based and Time-Based Events Including a Combination of Date and Time into a Single Object Using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration
- Work with Dates and Times Across Timezones and Manage Changes Resulting from Daylight Savings Including Format Date and Times Values
- Define and Create and Manage Date-Based and Time-Based Events Using Instant, Period, Duration, and TemporalUnit
- Read and Set the Locale by Using the Locale Object
- Create and Read a Properties File
- Build a Resource Bundle for Each Locale and Load a Resource Bundle in an Application

✓ Two-Minute Drill

Q&A Self Test

This chapter focuses on the exam objectives related to working with date- and time-related events, formatting dates and times, and using resource bundles for localization and internationalization tasks. Many of these topics could fill an entire book. Fortunately, you won't have to become a guru to do well on the exam. The intention of the exam team was to include just the basic aspects of these technologies, and in this chapter, we cover

more than you'll need to get through the related objectives on the exam.

CERTIFICATION OBJECTIVE

Dates, Times, and Locales (OCP Objectives 7.1, 7.2, 7.3, and 12.1)

7.1 Create and manage date-based and time-based events including a combination of date and time into a single object using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration.
7.2 Work with dates and times across timezones and manage changes resulting from daylight savings including Format date and times values.
7.3 Define, create and manage date-based and time-based events using Instant, Period, Duration, and TemporalUnit.
12. 1 Read and set the locale by using the Locale object.

The Java API provides an extensive (perhaps a little *too* extensive) set of classes to help you work with dates and times. The exam will test your knowledge of the basic classes and methods you'll use to work with dates and such. When you've finished this section, you should have a solid foundation in tasks such as creating date and time objects, and creating, manipulating, and formatting dates and times, and doing all of this for locations around the globe. In fact, a large part of why this section was added to the exam was to test whether you can do some basic internationalization (often shortened to "i18n").

Note: In this section, we'll introduce the `Locale` class. Later in the chapter, we'll be discussing resource bundles, and you'll learn more about `Locale` then.

Working with Dates and Times

If you want to work with dates and times from around the world (and who doesn't?), you'll need to be familiar with several classes from the `java.time` package. `java.time` is new in Java 8, so if you're looking for the old familiar `java.util.Date` and `java.util.Calendar`, you won't find them on the exam, although we'll briefly mention them here for comparison purposes.

Here's an overview of how the classes in `java.time` are organized:

- **Local dates and times** These dates and times are local to your time zone and so don't have time-zone information associated with them. These are represented by classes `java.time.LocalDate`, `java.time.LocalTime`, and `java.time.LocalDateTime`.
- **Zoned dates and times** These dates and times include time-zone information. They are represented by classes `java.time.ZonedDateTime` and `java.time.OffsetDateTime`.
- **Formatters for dates and times** With `java.time.format.DateTimeFormatter`, you can parse and print dates and times with patterns and in a variety of styles.
- **Adjustments to dates and times** With `java.time.temporal.TemporalAdjusters` and `java.time.temporal.ChronoUnit`, you can adjust and manipulate dates and times by handy increments.
- **Periods, Durations, and Instants** `java.time.Periods` and `java.time.Durations` represent an amount of time, periods for days or longer and durations for shorter periods like minutes or seconds. `java.time.Instants` represent a specific instant in time, so you can, say, compute the number of minutes between two instants.

If you're used to working with `java.util.Date`, `java.util.Calendar`, and `java.text.DateFormat`, you're going to have to forget most of what you've learned and start over (although the concepts are similar, so you have a bit of a head start). All those classes are still around (although some methods are marked as deprecated), but they are considered "old," and the classes in `java.time` are designed to replace them completely. Hopefully this design will stick!

The `Date` Class

The API design of the `java.util.Date` class didn't do a good job of handling internationalization and localization situations, so it's been largely replaced by the classes in `java.time`, like `java.time.localDateTime` and `java.time.ZonedDateTime`. You might find `Date` used in legacy code, but, for the most part, it's time to leave it behind.

The `Calendar` Class

Likewise, the `java.util.Calendar` class has largely been replaced with the classes in `java.time`. You'll still find plenty of legacy `Calendar` code around, but if you've worked with `Calendar` before, you'll likely find the `java.time` classes easier and less convoluted to work with.

With that, we'll dive right into the `java.time` classes.

The java.time.* Classes for Dates and Times

To make learning about dates and times more fun, let's imagine you are a solar eclipse hunter. You love chasing solar eclipses around the country and around the world. This example will be U.S.-centric, but you can easily apply all these ideas to dates and times in other countries too (and doing so is great practice for the exam).

Let's begin by figuring out the current date and time where you are, right now:

```
import java.time.*;
public class Eclipse {
  public static void main(String[] args) {
    LocalDate nowDate = LocalDate.now();
    LocalTime nowTime = LocalTime.now();
    LocalDateTime nowDateTime = LocalDateTime.of(nowDate, nowTime);
    System.out.println("It's currently " + nowDateTime + " where I am");
  }
}
```

Here we're using the static method `LocalDate.now()` to get the current date. It has no time zone, so think of it as a description of "the date," whatever that date is for you today, wherever you are, as you try this code. Similarly, we're using the static method `LocalTime.now()` to get the current time, so that will be whatever the time is for you right now, wherever you

are, as you try this code. We then use the date and time of “now” to create a `LocalDateTime` object using the `of()` static method and then display it.

When we run this code we see

```
It's currently 2017-10-11T14:51:19.982 where I am
```

The string `2017-10-11T14:51:19.982` represents the date, October 11, 2017, and time, 14:51:19.982, which is 2:51 `PM` and 19 seconds and 982 milliseconds. Notice that Java displays a “`T`” between the date and the time when converting the `LocalDateTime` to a string.

Of course, you’ll see a completely different date and time because you’re running this code in your own date and time, wherever and whenever that is.

We could also write:

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

to get the current date and time of now as a `LocalDateTime`.

What if you want to set a specific date and time rather than “now”?

Let’s say you went to Madras, Oregon, in the United States to see the solar eclipse on August 21, 2017. Here are a couple of ways you can set that specific date:

```
// The day of the eclipse in Madras, OR
LocalDate eclipseDate1 = LocalDate.of(2017, 8, 21);
LocalDate eclipseDate2 = LocalDate.parse("2017-08-21");
System.out.println("Eclipse date: " + eclipseDate1 + ", " +
eclipseDate2);
```

We’re creating the same date in two slightly different ways: first, by specifying a year, month, and day as arguments to the `LocalDate.of()` static method; and second, by using the `LocalDate.parse()` method to parse a string that matches the date. `LocalDate` represents a date in the ISO-8601 calendar system (which specifies a format of YYYY-MM-DD). You don’t need to know that, except to know the kinds of strings that the `parse()`

method can parse correctly. (For a full list of the formats of the dates and times that can be parsed and represented, check out the documentation for `java.time.format.DateTimeFormatter`, which we'll talk more about in a little bit. See https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatte

When we run this code, we see the default display (using that ISO format) for both dates:

```
Eclipse date: 2017-08-21, 2017-08-21
```

An eclipse happens at a specific time of day. The eclipse begins awhile before *totality* (when the moon almost completely obscures the sun), so let's create `LocalTime` objects to represent the time the eclipse begins and the time of totality:

```
// Eclipse begins in Madras, OR
LocalTime begins = LocalTime.of(9, 6, 43);          // 9:06:43
// Totality starts in Madras, OR
LocalTime totality = LocalTime.parse("10:19:36");   // 10:19:36
System.out.println("Eclipse begins at " + begins +
   " and totality is at " + totality);
```

As with `LocalDate`, a `LocalTime` has no time zone associated with it, so in this case, we need to know that we're creating times that are valid in Madras, OR (on U.S. Pacific time). For these times, we again use the static `of()` and `parse()` methods to demonstrate two different ways to create `LocalTime` objects.

When we print the times, we see:

```
Eclipse begins at 09:06:43 and totality is at 10:19:36
```

If you want to be precise about the format of the date and time you're parsing into a `LocalDate` or `LocalTime` or `LocalDateTime`, you can use `DateTimeFormatter`. You can either use one of several predefined formats or

create your own format for parsing using a sequence of letters and symbols. In the following example, we create a date and time in a string and then tell the `LocalDateTime` how to parse that using a `DateTimeFormatter`:

```
String eclipseDateTime = "2017-08-21 10:19";
DateTimeFormatter formatter =
   DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
LocalDateTime eclipseDay =
   LocalDateTime.parse(eclipseDateTime, formatter); // use formatter
System.out.println("Eclipse day: " + eclipseDay);
```

This creates a `LocalDateTime` object from the string, formatted using the pattern we specified, and when we print out the `LocalDateTime`, we see the correct date and time, printed in the standard ISO format we saw before:

```
Eclipse day: 2017-08-21T10:19
```

Of course, you can also use `DateTimeFormatter` to change the format of the output (again using letters and symbols):

```
System.out.println("Eclipse day, formatted: " +
  eclipseDay.format(DateTimeFormatter.ofPattern("dd, mm, yy hh,
mm")));
```

This code results in the output:

```
Eclipse day, formatted: 21, 19, 17 10, 19
```

We'll come back to `DateTimeFormatter` in a bit.

`LocalDateTime` has several methods that make it easy to add to and subtract from dates and times. For instance, let's say your Mom calls from Nashville, TN (on U.S. Central time) and asks, "What time will it be here when you're seeing the eclipse there?" You know that Central time is two hours ahead of Pacific time, so to answer that question you can write:

```
System.out.println("Mom time: " + eclipseDay.plusHours(2));
```

which reveals that it will be 12:19 PM where she is in Tennessee when you're watching the eclipse at 10:19 AM in Oregon:

```
Mom time: 2017-08-21T12:19:36
```

Then she asks, "When are you coming home?" You can tell her, "In three days" by writing the following code:

```
System.out.println("Going home: " + eclipseDay.plusDays(3));
```

which means you'll be going home on August 24:

```
Going home: 2017-08-24T10:19:36
```

Of course, there are loads of other handy methods too, like `getDayOfWeek()` to find out what day of the week the eclipse occurs:

```
System.out.println("What day of the week is eclipse? " +
 eclipseDay.getDayOfWeek());
```

which lets you know the eclipse was on a Monday:

```
What day of the week is eclipse? MONDAY
```

Zoned Dates and Times

Local dates and times are great when you don't need to worry about the time zone, but sometimes we need to share dates and times with people in other time zones, so knowing which time zone you're in or they're in becomes important.

You don't need to know too much about time zones for the exam,

except, of course, to understand what a time zone is and that different places in different parts of the world are on different time zones. For instance, in Madras, Oregon, you are in the U.S. Pacific time zone, whereas your Mom in Tennessee is in the U.S. Central time zone.

All time zones are based on *Greenwich Mean Time (GMT),* the time in Greenwich, England. GMT is a time zone. The name of the time standard that uses GMT as the basis for all other time zones is *Coordinated Universal Time (UTC).*

Your time zone will either be ahead of or behind GMT. For instance, in Madras, OR, for the eclipse, you are GMT-7, meaning you are seven hours behind GMT. That's for summer; in winter, you'll be GMT-8, or eight hours behind GMT because the United States has daylight savings time in summer and standard time in winter. Yes, zoned dates and times can get complicated fast, but rest assured, the exam is not about your depth of understanding of time zones. As long as you know the basics and you can create and use zoned dates and times, you'll be fine.

Let's create a zoned date and time for the date and time of the eclipse:

```
ZonedDateTime zTotalityDateTime =
   ZonedDateTime.of(eclipseDay, ZoneId.of("US/Pacific"));
System.out.println("Date and time totality begins with time zone: "
   + zTotalityDateTime);
```

Looking at the output, we see:

```
Date and time totality begins with time zone:
2017-08-21T10:19:36-07:00[US/Pacific]
```

A `ZonedDateTime` is a `LocalDateTime` plus a time zone, which is represented as a `ZoneId`. In this example, the `ZoneId` is "`US/Pacific`," which happens to be GMT-7 (which you may also see written as UTC-7). You can use either "US/Pacific" or "GMT-7" as the `ZoneId`.

You might be asking: How did you know the name of the `ZoneId`? Good

question. The names of the zones are not listed in the documentation page for `ZoneId`, so one good way to find them is to write some code to display them:

```
Set<String> zoneIds = ZoneId.getAvailableZoneIds();
List<String> zoneList = new ArrayList<String>(zoneIds);
Collections.sort(zoneList);
for (String zoneId : zoneList) {
   if (zoneId.contains("US")) {
      System.out.println(zoneId);
   }
}
```

With this code, we're displaying only the U.S. `zoneIds`. If you, say, want to display the `zoneIds` for Great Britain, then use `"GB"` in place of `"US"`. The list includes some `zoneIds` by country, some by city, and some by other names.

Let's get back to daylight savings time. Recall that the U.S. Pacific time zone is either GMT-7 (in winter, standard time) or GMT-8 (in summer, daylight savings time). That means when you're creating a `ZonedDateTime` for Madras, Oregon (or any other place that uses daylight savings), you're going to need to know if you're currently in daylight savings time.

You can find out if you're in daylight savings time by using `ZoneRules`. All you need to know about `ZoneRules` is that the class captures the current rules about daylight savings in various parts of the world. Unfortunately, this tends to change (as politicians tend to change their minds—what a shocker!) so the rules are, as the documentation states, only as accurate as the information provided. Let's assume the rules are up to date (as you should for the exam) and write some code to find out if the `"US/Pacific"` time zone is currently in daylight savings time:

```
ZoneId pacific = ZoneId.of("US/Pacific");
// pacific.getRules() returns a ZoneRules object that has all the rules
// about time zones, including daylight savings and standard time.
System.out.println("Is Daylight Savings in effect at time of totality: " +
   pacific.getRules().isDaylightSavings(zTotalityDateTime.toInstant()));
```

In this code, we first get the `ZoneId` for the `"US/Pacific"` time zone. We can then use that `ZoneId` to get the `ZoneRules` with the `getRules()` method. A `ZoneRules` object has a method `isDaylightSavings()`, which takes an `Instant` and determines whether that `Instant` is currently in daylight savings. We'll discuss `Instants` in more detail shortly; for now, just know that we can convert the `ZonedDateTime` representing the date and time of the eclipse into an `Instant` with the `toInstant()` method. The result is

```
Is Daylight Savings in effect at time of totality: true
```

The result is `true` because the date of the eclipse, `zTotalityDateTime` (that is, August 21, 2017, in the U.S. Pacific zone), was in daylight savings time (summer time).

Date and Time Adjustments

Once you've created a `LocalDateTime` or `ZonedDateTime`, you can't modify it. The documentation describes datetime objects as "immutable." However, you can create a new datetime object from an existing datetime object, and `java.time.*` provides plenty of adjusters that make it easy to do so. In other words, rather than modifying an existing datetime, you just make a new datetime from the existing one.

Let's say you want to find the date of the Thursday following the eclipse. We can take the `ZonedDateTime` for the eclipse we made above, `zTotalityDateTime`, and create a new `ZonedDateTime` from it to represent "the following Thursday" like this:

```
ZonedDateTime followingThursdayDateTime =
    zTotalityDateTime.with(TemporalAdjusters.next(   // adjust date time
        DayOfWeek.THURSDAY));                        // to next Thursday
System.out.println("Thursday following the totality: " +
    followingThursdayDateTime);
```

The output is:

```
Thursday following the totality: 2017-08-24T10:19:36-
07:00[US/Pacific]
```

We can see that the Thursday following the eclipse (which, remember, was on Monday, August 21, 2017) is August 24.

The class `TemporalAdjusters` has a whole slew of handy methods to make a `TemporalAdjuster` for a variety of scenarios, such as `firstDayOfNextYear()`, `lastDayOfMonth()`, and more.

You've already seen how you can add days and hours from a datetime; many other adjustments, including `plusMinutes()`, `plusYears()`, `minusWeeks()`, `minusSeconds()`, and so on, are available as methods in `LocalDateTime` and `ZonedDateTime`. You'll also find a variety of adjustments like `withHour()` and `withYear()`, which you can use to create a new datetime object from an existing one, but with a different hour or year. Each of these methods creates a new adjusted datetime from an existing one.

`ZonedDateTimes` are subject to `ZoneRules` when you adjust them. So, if you, say, add a month to an existing `ZonedDateTime`, the `ZoneRules` will be used to determine if the new `ZonedDateTime` is GMT-7 or GMT-8, for instance, (depending on daylight savings time). If you want to create a datetime with a zone offset from GMT that does not use the `ZoneRules`, then you can use an `OffsetDateTime`. An `OffsetDateTime` is a fixed datetime and offset that doesn't change even if the `ZoneRules` change.

Periods, Durations, and Instants

So far, we've looked at how to create specific dates and times. The `java.time.* package` also includes ways to represent a period of time,

whether that's a period of days, months, or years (a `Period`), a short period of minutes or hours (a `Duration`), or an instant in time (an `Instant`).

Periods You're looking forward to the next eclipse on April 8, 2024, which you're going to watch in Austin, Texas, and you want to set a reminder for yourself one month in advance, so you don't forget. You could just say, well, I'll remind myself on March 8, 2024, but what fun would that be? Let's compute one month before the eclipse in code:

```
// Totality begins in Austin, TX in 2024 at 1:35pm and 56 seconds;
// Specify year, month, dayOfMonth, hour, minute, second, nano, zone
ZonedDateTime totalityAustin =
   ZonedDateTime.of(2024, 4, 8, 13, 35, 56, 0, ZoneId.of("US/Central"));
System.out.println("Next total eclipse in the US, date/time in Austin, TX: " +
   totalityAustin);
```

To create the reminder, we first create a `ZonedDateTime` for when totality begins in Austin. We use the `ZonedDateTIme.of()` static method to create the datetime with the arguments year, month, day, hours, minutes, seconds, nanoseconds, and zone id. We have no idea what the nanoseconds are for the totality beginning, so we just put 0. Notice we used 13 to specify 1 `PM`. Austin is in the U.S. Central time zone, so we get the `zoneId` using that name (which we got earlier when we displayed all the U.S. time-zone names). When we print this `ZonedDateTime`, we see

```
Next total eclipse in the US, date/time in Austin, TX:
2024-04-08T13:35:56-05:00[US/Central]
```

Now let's create the reminder for one month before this date and time by creating a `Period` that represents one month and subtract it from the date and time for the eclipse:

```
// Reminder for a month before
Period period = Period.ofMonths(1);
System.out.println("Period is " + period);
ZonedDateTime reminder = totalityAustin.minus(period);
System.out.println("DateTime of 1 month reminder: " + reminder);
```

Here's the output:

```
Period is P1M
DateTime of 1 month reminder: 2024-03-08T13:35:56-
06:00[US/Central]
```

Notice how the period is displayed, with "`P`" meaning period and "`1M`" meaning month.

While we're here, let's see how to create a `LocalDateTime` from the `ZonedDateTime` for people who are in Austin:

```
System.out.println("Local DateTime (Austin, TX) of reminder: "
+
 reminder.toLocalDateTime());
```

Notice the difference in the `LocalDateTime`—there's no time zone:

```
Local DateTime (Austin, TX) of reminder: 2024-03-08T13:35:56
```

And finally, let's figure out when we'll see the reminder in Madras, Oregon:

```
System.out.println("Zoned DateTime (Madras, OR) of reminder: "
+
 reminder.withZoneSameInstant(ZoneId.of("US/Pacific")));
```

We'll see the reminder at 11 `AM`, two hours earlier than the reminder in Austin, because Madras is two hours behind.

```
Zoned DateTime (Madras, OR) of reminder: 2024-03-08T11:35:56-
```

```
08:00[US/Pacific]
```

One more thing to notice about this code. The eclipse is happening in April 2024. April happens to be in summer time, or daylight savings time, so notice that during daylight savings, U.S. Central time is five hours behind GMT:

```
Next total eclipse in the US, date/time in Austin, TX:
2024-04-08T13:35:56-05:00[US/Central]
```

When we subtracted the one-month period from this date and time to get the date and time for our reminder, we compute that the reminder is on March 8, which is in winter time, or standard time:

```
DateTime of 1 month reminder: 2024-03-08T13:35:56-
06:00[US/Central]
```

So, on March 8, 2024, Austin will be six hours behind GMT. Nice for us that Java correctly computed the time using `ZoneRules` behind the scenes.

Durations How many minutes from the time the eclipse begins to the time totality begins? We can compute the time in a couple of ways; we're going to do it using `ChronoUnit` and `Duration`. `ChronoUnit` is an `enum` in `java.time.temporal` that provides a set of predefined units of time periods. For instance, `ChronoUnit.MINUTES` represents the concept of a minute. `ChronoUnit` also supplies a method `between()` that we can use to compute a `ChronoUnit` time period between two times. Once we have the number of minutes between two times, we can use that to create a `Duration`. `Durations` have all kinds of handy methods for computing things, like adding and subtracting hours and minutes and seconds, or converting a `Duration` into a number of seconds or milliseconds, and so on. Think of `ChronoUnit` as a unit of time and `Duration` as specifying a period of time (like a `Period`, only for period lengths less than a day).

First, let's create two `LocalTimes` to represent the start of the eclipse (when the moon first starts to cross the sun) and the time of totality (when the moon completely obscures the sun):

```
// Eclipse begins in Austin, TX
LocalTime begins = LocalTime.of(12, 17, 32);    // 12:17:32
// Totality in Austin, TX
LocalTime totality = LocalTime.of(13, 35, 56);  // 13:35:56
System.out.println("Eclipse begins at " + begins +
   " and totality is at " + totality);
```

Notice we're just using `LocalTime` here, not `ZonedDateTime`, so we don't have to specify a time zone. The output looks like this:

```
Eclipse begins at 12:17:32 and totality is at 13:35:56
```

Now, let's use a `ChronoUnit` to compute the number of minutes between `begins` and `totality`:

```
// How many minutes between when the eclipse begins and totality?
long betweenMins = ChronoUnit.MINUTES.between(begins, totality);
System.out.println("Minutes between begin and totality: " + betweenMins);
```

The minutes returned by the `between()` method is a `long`. When we look at the output, we see we have 78 minutes between the beginning of the eclipse and the beginning of totality. Notice that we lost the number of seconds in this computation because we asked for the number of minutes between the two times.

Let's turn this into a `Duration`. As you might expect, we can turn the number of minutes into a `Duration` using `Duration.ofMinutes()`:

```
Duration betweenDuration = Duration.ofMinutes(betweenMins);
System.out.println("Duration: " + betweenDuration);
```

Looking at the output we see:

```
Duration: PT1H18M
```

`PT` means "period of time," meaning `Duration` (rather than `Period`), and then `1H18M` means "1 hour and 18 minutes" corresponding to our 78 minutes.

Just to double-check ourselves, let's take the `begin LocalTime` we created before and add back our `Duration` using the `LocalTime.plus()` method. We could also do this with our `betweenMins` value, using `LocalTime.plusMinutes()`. The `plus()` method takes a `TemporalAmount` (like a `Duration` or a `Period`), whereas `plusMinutes()` takes `minutes` as a `long`. Either way will work.

```
LocalTime totalityBegins = begins.plus(betweenDuration);
System.out.println("Totality begins, computed: " + totalityBegins);
```

The result is

```
Totality begins, computed: 13:35:32
```

This time is slightly different than our original `begins` time of 13:35:56 because we lost the seconds when we created the `Duration` from the minutes between `begins` and `totality`.

Instants An `Instant` represents an instant in time. Makes sense, right? But how is it different from a `DateTime`? If you're used to timestamps, then you'll probably recognize an `Instant` as the number of seconds (and nanoseconds) since January 1, 1970—the standard Java epoch. `Instants` can't be represented as just one `long`, like you might be used to, because an `Instant` includes nanoseconds, so the seconds plus the nanoseconds is too big for a `long`. However, once you've created an `Instant`, you can always get the number of seconds as a `long` value from the `Instant`.

`ZonedDateTimes` can be converted to `Instants` using the `toInstant()` method:

```
ZonedDateTime totalityAustin =
   ZonedDateTime.of(2024, 4, 8, 13, 35, 56, 0, ZoneId.of("US/Central"));
Instant totalityInstant = totalityAustin.toInstant();
System.out.println("Austin's eclipse instant is: " + totalityInstant);
```

Looking at the output we see

```
Austin's eclipse instant is: 2024-04-08T18:35:56Z
```

Even though we created a `ZonedDateTime` for Austin at 1:35 `PM`, in the `US/Central` time zone, the instant displays as 6:35 `PM` and shows a `Z` at the end. That datetime represents 6:35 `PM` GMT. The `Z` is how you know the time displayed is for the GMT zone, rather than the U.S. Central zone. This format is the `ISO_INSTANT` format of displaying a datetime.

Note that if you want to call the `toInstant()` method on a `LocalDateTime`, you'll need to supply a `ZoneOffset` as an argument. To create a unique instant in time that works globally, a time zone is required when the `Instant` is created. If we don't include a time zone, then *your* instant and *our* instant may mean two different things.

Let's once again compute the number of minutes between two times using `ChronoUnit.MINUTES`. This time we'll compute the minutes between now and the Austin eclipse as represented by `Instants` and then use that to create a `Duration`. We'll use the `totalityInstant` we created above for the instant of the totality, and we'll use the `Instant.now()` method to create an instant representing right now:

```
Instant nowInstant = Instant.now();  // represents now
Instant totalityInstant = totalityAustin.toInstant;  // same as above
long minsBetween =
   ChronoUnit.MINUTES.between(nowInstant, totalityInstant);
Duration durationBetweenInstants = Duration.ofMinutes(minsBetween);
System.out.println("Minutes between " + minsBetween +
   ", is duration " + durationBetweenInstants);
```

The output is

```
Minutes between 3405250, is duration PT56754H10M
```

As you can see (reading the `Duration`), between now and the next eclipse in Austin, we have only 56,754 hours and 10 minutes to wait. That eclipse will be here in no time.

Lastly, if you want to get the number of seconds since January 1, 1970, from an `Instant`, use the method `getEpochSecond()`:

```
Instant now = Instant.now();
System.out.println("Seconds since epoch: " + now.getEpochSecond());
```

The number of seconds is

```
Seconds since epoch: 1508286832
```

A Few Other Handy Methods and Examples

Let’s add another reminder before the next eclipse, say, for three days before the eclipse, and then let’s figure out what day of the week this reminder will occur:

```
// Another reminder 3 days before
System.out.println("DateTime of 3 day reminder: " +
   totalityAustin.minus(Period.ofDays(3)));
// What day of the week is that?
System.out.println("Day of week for 3 day reminder: " +
   totalityAustin.minus(Period.ofDays(3)).getDayOfWeek());
```

We see that the three-day reminder is on April 5, which is a Friday:

```
DateTime of 3 day reminder: 2024-04-05T13:35:56-05:00[US/Central]
Day of week for 3 day reminder: FRIDAY
```

And we really should call our sister in Paris a couple of hours after the next eclipse to tell her how it was:

```
ZonedDateTime localParis =
   totalityAustin.withZoneSameInstant(ZoneId.of("Europe/Paris"));
System.out.println("Eclipse happens at " + localParis + " Paris time");
System.out.println("Phone sister at 2 hours after totality: " +
   totalityAustin.plusHours(2) + ", " +
   localParis.plusHours(2) + " Paris time");
```

From the output, we can see that the eclipse happens at 8:35 Paris time, and when we call our sister two hours after the eclipse, it will be 3:35 PM Austin time and 10:35 PM Paris time:

```
Eclipse happens at 2024-04-08T20:35:56+02:00[Europe/Paris] Paris time
Phone sister at 2 hours after totality:
2024-04-08T15:35:56-05:00[US/Central],
2024-04-08T22:35:56+02:00[Europe/Paris] Paris time
```

If you tend to lose track of time, but you really, really don't want to miss the eclipse, you can check to make sure the eclipse is still in the future with this code:

```
// compare two ZonedDateTimes (must be the same type!)
System.out.println("Is the 2024 eclipse still in the future? " +
   ZonedDateTime.now().isBefore(totalityAustin));
```

Since we're writing this in 2017, the 2024 eclipse is still far in the future, so we see

```
Is the 2024 eclipse still in the future? true
```

And finally, we'd better do one more check about 2024. How about checking to see if 2024 is a leap year? You definitely don't want to miss the eclipse by a day:

```
System.out.println("Is 2024 a leap year? " +
totalityAustin.isLeapYear());
```

Try this code and you'll get a compile-time error:

```
The method isLeapYear() is undefined for the type
ZonedDateTime
```

Hmm. It turns out `isLeapYear()` is defined only for `LocalDate`, not for `LocalDateTime` or `ZonedDateTime`. We can fix the code by converting `totalityAustin` to a `LocalDate`:

```
System.out.println("Is 2024 a leap year? " +
 totalityAustin.toLocalDate().isLeapYear());
```

Another way to check for a leap year is

```
System.out.println("Is 2024 a leap year? " +
Year.of(2024).isLeap());
```

The output from both lines of code shows that 2024 is, indeed, a leap year.

Formatting Output with DateTimeFormatter

Earlier we used `DateTimeFormatter` to specify a pattern when parsing a date string. We can also use `DateTimeFormatter` when we display a datetime as a string.

You might know that in the United States, we tend to write month/day/year, and in the European Union, they tend to write day/month/year. (Yes, that can get a bit confusing at times!)

Let's format and display the datetime of the eclipse in Austin using the European-preferred format. There are a couple of different ways we can do that. First, we can specify exactly the format we want using letters and symbols, as we described earlier:

```
System.out.println("Totality date/time written for sister in Europe: " +
   totalityAustin.format(
     DateTimeFormatter.ofPattern("dd/MM/yyyy hh:mm")));
```

Here, we're using the `format()` method of the `ZonedDateTime`, `totalityAustin`, and passing in a formatter. The formatter specifies a format to use for formatting the datetime, using allowed letters and symbols (see the `DateTimeFormatter` documentation for all the options). When we look at the output, we see

```
Totality date/time written for sister in Europe: 08/04/2024 01:35
```

Alternatively, we could specify a format style and a locale:

```
System.out.println("Totality date/time in UK Locale: " +
   totalityAustin.format(
     DateTimeFormatter.ofLocalizedDateTime(
       FormatStyle.SHORT)
     .withLocale(Locale.UK)));
```

Now when we look at the output, we see

```
Totality date/time in UK Locale: 08/04/24 13:35
```

You'll learn more about `Locales` shortly; essentially, `Locales` are designed to tailor data for a specific region. Here, we're creating a `DateTimeFormatter` by specifying a built-in style and then using that to create a new formatter for a specific locale, the UK locale. Creating a formatter with a specific locale means the formatter is adjusted appropriately for that locale. We see that the UK locale uses the day/month/year format for the date and the 24-hour format for the time. As you can see, there is a lot to the `java.time.*` package. There's no way you can memorize everything in the package for the exam, so we recommend you focus on the classes, properties, and methods in Tables 4-1 and 4-2 (later in the chapter) and familiarize yourself with the rest of the package by looking over 222the documentation to get a sense of what's there (see https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html).

You've probably noticed a pattern in the method names used in the `java.time.*` package. For instance, `of()` methods create a new date from, typically, a sequence of numbers specifying the year, month, day, and so on. `parse()` methods create a new date by parsing a string that's either in a standard ISO format already or by using a formatter. `with()` methods allow you to adjust a date with a `TemporalAdjuster` to make a new date. `plusX()` and `minusX()` methods create a new datetime object from an existing one by adding and subtracting `TemporalUnits` or longs representing weeks, minutes, and so on. Study the `LocalDateTime` and `ZonedDateTime` methods enough to get the hang of this pattern so you can recognize the methods on the exam (without having to memorize them all).

Using Dates and Times with Locales

The `Locale` class is your ticket to understanding how to internationalize your code. Both the `DateTimeFormatter` class and the `NumberFormat` class can use an instance of `Locale` to customize formatted output for a specific locale (and you just got a taste of this with `DateTimeFormatter` in the previous section).

You might ask how Java defines a locale. The API says a locale is "a specific geographical, political, or cultural region." The two `Locale` constructors you'll need to understand for the exam are

```
Locale(String language)
Locale(String language, String country)
```

The language argument represents an ISO 639 Language code, so, for instance, if you want to format your dates or numbers in Walloon (the language sometimes used in southern Belgium), you'd use `"wa"` as your language string. There are over 500 ISO Language codes, including one for Klingon (`"tlh"`), although, unfortunately, Java doesn't yet support the Klingon locale. We thought about telling you that you'd have to memorize all these codes for the exam…but we didn't want to cause any heart attacks. So rest assured, you will *not* have to memorize any ISO Language codes or ISO Country codes (of which there are about 240) for the exam.

Let's get back to how you might use these codes. If you want to represent basic Italian in your application, all you need is the Language code. If, on the other hand, you want to represent the Italian used in Switzerland, you'd want to indicate that the country is Switzerland (yes, the Country code for Switzerland is `"CH"`), but that the language is Italian:

```
Locale locIT = new Locale("it");       // Italian
Locale locCH = new Locale("it", "CH"); // Switzerland
```

Using these two locales on a date could give us output like this:

```
sabato 1 ottobre 2005
sabato, 1. ottobre 2005
```

Now let's put this all together in some code that creates a `ZonedDateTime` object and sets its date. We'll then take that datetime object and print it using locales from around the world:

```
Locale myLocale = Locale.getDefault();
System.out.println("My locale: " + myLocale);
LocalDateTime aDateTime = LocalDateTime.of(2024, 4, 8, 13, 35, 56);
System.out.println("The date and time: " +
    aDateTime.format(DateTimeFormatter.ofLocalizedDateTime(
                      FormatStyle.MEDIUM)));
ZonedDateTime zDateTime = ZonedDateTime.of(
                          aDateTime, ZoneId.of(
                          "US/Pacific"));
```

```
Locale locIT = new Locale("it", "IT");      // Italy
Locale locPT = new Locale("pt");            // Portugal
Locale locBR = new Locale("pt", "BR");      // Brazil
Locale locIN = new Locale("hi", "IN");      // India
Locale locJA = new Locale("ja");            // Japan
Locale locDK = new Locale("da", "DK");      // Denmark
System.out.println("Italy (Long) " +
    zDateTime.format(
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG)
    .withLocale(Locale.ITALY)));
System.out.println("Italy (Short) " +
    aDateTime.format(
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT)
    .withLocale(locIT)));

System.out.println("Japan (Long) " +
    zDateTime.format(
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG)
    .withLocale(Locale.JAPAN)));

System.out.println("Portugal (Long) " +
    zDateTime.format(
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG)
    .withLocale(locPT)));

System.out.println("India (Long) " +
    zDateTime.format(
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG)
    .withLocale(locIN)));

System.out.println("Denmark (Medium) " +
    zDateTime.format(
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM)
    .withLocale(locDK)));
```

This code, on our JVM, produces the output:

```
My locale: en_US
The date and time: Apr 8, 2024 1:35:56 PM
Italy (Long) 8 aprile 2024 13.35.56 PDT
Italy (Short) 08/04/24 13.35
Japan (Long) 2024/04/08 13:35:56 PDT
Portugal (Long) 8 de Abril de 2024 13:35:56 PDT
India (Long) 8 अप्रैल, 2024 1:35:56 अपराह्न PDT
Denmark (Medium) 08-04-2024 13:35:56
```

So you can see how a single `ZonedDateTime` object can be formatted to work for many locales and varying amounts of detail. (Note that you'll need Eclipse in UTF-8 format to see the Indian output properly.)

There are a couple more methods in `Locale` (`getDisplayCountry()` and `getDisplayLanguage()`) that you need to know for the exam. These methods let you create strings that represent a given locale's country and language in terms of both the default locale and any other locale:

```
Locale locBR = new Locale("pt", "BR");  // Brazil
Locale locDK = new Locale("da", "DK");  // Denmark
Locale locIT = new Locale("it", "IT");  // Italy

System.out.println("Denmark, country: " + locDK.getDisplayCountry());
System.out.println("Denmark, country, local: " +
   locDK.getDisplayCountry(locDK));
System.out.println("Denmark, language: " + locDK.getDisplayLanguage());
System.out.println("Denmark, language, local: " +
   locDK.getDisplayLanguage(locDK));

System.out.println("Brazil, country: " + locBR.getDisplayCountry());
System.out.println("Brazil, country, local: " +
   locBR.getDisplayCountry(locBR));
System.out.println("Brazil, language: " + locBR.getDisplayLanguage());
System.out.println("Brazil, language, local: " +
   locBR.getDisplayLanguage(locBR));
System.out.println("Italy, Danish language is: " +
   locDK.getDisplayLanguage(locIT));
```

This code, on our JVM, produces the output:

```
Denmark, country: Denmark
Denmark, country, local: Danmark
Denmark, language: Danish
Denmark, language, local: Dansk
Brazil, country: Brazil
Brazil, country, local: Brasil
Brazil, language: Portuguese
Brazil, language, local: português
Italy, Danish language is: danese
```

Our JVM's locale (the default for us, which we saw displayed earlier) is `en_US`, and when we display the country name for Brazil, in our locale, we get `Brazil`. In Brazil, however, the country is `Brasil`. Same with the language; for us the language of Brazil is `Portuguese`; for people in Brasil, it's `português`. Likewise, for Denmark, you can see how we have different names for the country and the language than the Danish do.

Finally, just for fun, we discovered that in Italy, the Danish language is called `danese`.

Orchestrating Date- and Time-Related Classes

When you work with dates and times, you'll often use several classes together. It's important to understand how the classes described earlier relate to each other and when to use which classes in combination. For instance, you need to know that if you're creating a new `ZonedDateTime` from an existing `LocalDate` and `LocalTime`, you need a `ZoneId` too; if you want to do date formatting for a specific locale, you need to create your `Locale` object before your `DateTimeFormatter` object because you'll need your `Locale` object as an argument to your `DateTimeFormatter` method; and so on. Tables 4-1 and 4-2 provide a quick overview and summary of common date- and time-related use cases: how to create datetime objects and how to adjust them. We are by no means including all of the many available methods to work with dates and times, however, so make sure you peruse the

documentation too (see https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html).

TABLE 4-1 Instance Creation for `java.time` Classes

java.time Class	Key Instance Creation Options

LocalDate	`LocalDate.now();`
	`LocalDate.of(2017, 8, 21);`
	`LocalDate.parse("2017-08-21");`

LocalTime	`LocalTime.now();`
	`LocalTime.of(10, 19, 36);`
	`LocalTime.parse("10:19:36");`

LocalDateTime	`LocalDateTime.now();`
	`LocalDateTime.of(aDate, aTime);`
	`LocalDateTime.parse("2017-04-08T10:19:36");`
	`LocalDateTime.parse(aDateTime, aFormatter);`
	`LocalDateTime.parse("2017-08-21T10:19",`
	`aformatter);`

ZonedDateTime	`ZonedDateTime.now();`
	`ZonedDateTime.of(aDateTime, ZoneId.of(aZoneString));`
	`ZonedDateTime.parse("2017-04-08T10:19:36-05:00");`

OffsetDateTime	`OffsetDateTime.now();`
	`OffsetDateTime.of(aDateTime,`
	`ZoneOffset.of("-05:00"));`
	`OffsetDateTime.parse("2017-04-08T10:19:36-05:00");`

format.
DateTimeFormatter
	`DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");`
	`DateTimeFormatter.ofLocalizedDateTime(`
	`FormatStyle.SHORT).withLocale(aLocale);`

Instant	`Instant.now();`
	`zonedDateTime.toInstant();`
	`aDateTime.toInstant(ZoneOffset.of("+5"));`

Duration	`Duration.between(aTime1, aTime2);` `Duration.ofMinutes(5);`
Period	`Period.between(aDate1, aDate2);` `Period.ofDays(3);`

util.Locale	`Locale.getDefault();`
	`new Locale(String language);`
	`new Locale(String language, String country);`

TABLE 4-2 Adjustment Options for `java.time` Classes

Class	Key Adjustment Options and Examples (all methods create a new datetime object)

LocalDate	`aDate.minusDays(3);`
	`aDate.plusWeeks(1);`
	`aDate.withYear(2018);`

LocalTime	`aTime.minus(3, ChronoUnit.MINUTES);`
	`aTime.plusMinutes(3);`
	`aTime.withHour(12);`

LocalDateTime	`aDateTime.minusDays(3);`
	`aDateTime.plusMinutes(10);`
	`aDateTime.plus(Duration.ofMinutes(5));`
	`aDateTime.withMonth(2);`

ZonedDateTime	`zonedDateTime.withZoneSameInstant(ZoneId.of("US/Pacific"));`

CERTIFICATION OBJECTIVE

Properties Files (OCP Objective 12.2)

12.2 Create and read a Properties file.

Property files are typically used to externally store configuration settings and operating parameters for your applications. In the Java world, there are at

least three variations on property files:

1. There is a system-level properties file that holds system information like hardware info, software versions, classpaths, and so on. The `java.lang.System` class has methods that allow you to update this file and view its contents. This property file is not on the exam.
2. There is a class called `java.util.Properties` that makes it easy for a programmer to create and maintain property files for whatever applications the programmer chooses. We'll talk about the `java.util.Properties` class in this section. In this section, when we say "property" files, we're referring to files that are compliant with the `java.util.Properties` class.
3. There is a class called `java.util.ResourceBundle` that *can*—optionally—use `java.util.Properties` files to make it easier for a programmer to add localization and/or internationalization features to applications. After we discuss `java.util.Properties`, we'll discuss `java.util.ResourceBundle`.

The `java.util.Properties` class is used to create and/or maintain human-readable text files. It's also possible to create well-formed, `Properties`-compliant text files using a text editor. If you're using a `Properties` file for applications other than to support resource bundles, you can give them whatever legal filenames you want. Typically their names end in "`.properties`," e.g., "`MyApp.properties`." Other suffixes like "`.props`" are also common. The basic structure of a `Properties` file is a set of comments (usually comment lines begin with "`#`") at the top of the file, followed by a number of rows of text data, each row representing a key/value pair, with the key and value usually separated with an "`=`".

Almost everyone uses `#` for comments and `=` to separate key/value pairs. There are alternative syntax choices, though, which you should understand if you come across them.

Property files can use two styles of commenting:

```
! comment
```

or

```
# comment
```

Property files can define key/value pairs in any of the following formats:

```
key=value
key:value
key value
```

Let's refresh what we've learned about property files and take a closer look. Aside from comments, a property file contains key/value pairs:

```
# this file contains a single key/value
hello=Hello Java
```

A *key* is the first string on a line. Keys and values are usually separated by an equal sign. If you want to break up a single line into multiple lines, you use a backslash. Given an entry in a property file:

```
hello1 = Hello \
World!
```

the code and output would be

```
System.out.println(rb.getString("hello1"));
Hello World!
```

If you actually want a line break, you use the standard Java `\n` escape sequence. Given an entry in a property file:

```
hello2 = Hello \nWorld  !
```

The code and output would be

```
System.out.println(rb.getString("hello2"));
Hello
World  !
```

You can mix and match these to your heart's content. Java helpfully

ignores any whitespace before subsequent lines of a multiline property, so you can use indentation for clarity:

```
hello3 =    123\
45
```

Given the above entry in a properties file, the code and output would be

```
System.out.println(rb.getString("hello3"));
12345
```

As we mentioned earlier, `java.lang.System` provides access to a property file. Although this file isn't on the exam, the following code

```
import java.util.*;
public class SysProps {
  public static void main(String[] args) {
    Properties p = System.getProperties();   // open system properties file
    p.setProperty("myProp", "myValue");      // add an entry
    p.list(System.out);                      // list the file's contents
  }
}
```

will produce output that contains entries like these:

```
myProp=myValue
java.version=1.8.0_45
os.name=Mac OS X
os.version=10.12.6
..
..
```

Again, what we're seeing here is a list of key/value pairs.

Let's move on to creating and working with our own property file. Here's some code that creates a new `Properties` object, adds a few properties, and then stores the contents of the `Properties` object to a file on disk:

```
import java.util.*;
import java.io.*;
class Props1 {
  public static void main(String[] args) {
    Properties p = new Properties();
    p.setProperty("k1", "v1");
    p.setProperty("k2", "v2");
    p.list(System.out);                    // what's in the object
    try {
      // creates or replaces file
      FileOutputStream out = new FileOutputStream("myProps1.props");
      p.store(out, "test-comment");        // adds header comment
      out.close();
    } catch (IOException e) {
      System.out.println("exc 1");
    }
  }
}
```

which produces the following output:

```
-- listing properties --
k2=v2
k1=v1
```

and a file named `myProps1.props`, which contains

```
#test-comment
#Fri Feb 02 14:53:30 PST 2018
k2=v2
k1=v1
```

Note that there are a couple of comments at the top of the file. Now let's run a second program that opens up the file we just created, adds a new key/value pair, then saves the result to a second file on disk:

```
import java.util.*;
import java.io.*;
class Props2 {
  public static void main(String[] args) {
    Properties p2 = new Properties();
    try {
      FileInputStream in = new FileInputStream("myProps1.props");
      p2.load(in);
      p2.list(System.out);
```

```
        p2.setProperty("newProp", "newData");
        p2.list(System.out);
        FileOutputStream out = new FileOutputStream("myProps2.props");
        p2.store(out, "myUpdate");
        in.close();
        out.close();
      } catch (IOException e) {
        System.out.println("exc 2");
      }
    }
  }
```

which produces

```
-- listing properties --
newProp=newData
k2=v2
k1=v1
```

and a file named `myProps2.props`, which contains

```
#myUpdate
#Fri Feb 02 14:53:58 PST 2018
newProp=newData
k2=v2
k1=v1
```

It’s important to know that `java.util.Properties` inherits from `java`

`.util.Hashtable`. Technically, when mucking around with property files, you could use methods from Hashtable like `put()` and `get()`, but Oracle encourages you to stick with the methods provided in the `Properties` class since those methods will force you to use arguments of type `String`. For the exam you should know the following methods from the `Properties` class:

```
String getProperty(String key)

void list(PrintStream out)

void load(InputStream inStream)

Object setProperty(String key, String value)

void store(OutputStream out, String headerComment)
```

Next, let's move on to the resource bundles to see how they work and how you can (if you want to) use property files to support your resource bundles.

CERTIFICATION OBJECTIVE

Resource Bundles (OCP Objectives 12.1, 12.2, and 12.3)

12.1 Read and set the locale by using the Locale object.
12.2 Create and read a Properties file.
12.3 Build a resource bundle for each locale and load a resource bundle in an application.

Earlier, we used the `Locale` class to display dates for basic localization. For full-fledged localization, we also need to provide language- and country-specific strings for display. There are only two parts to building an application with resource bundles:

- `Locale` You can use the same `Locale` we used for `DateFormat` and `NumberFormat` to identify which resource bundle to choose.
- `ResourceBundle` Think of a `ResourceBundle` as a map. You can use property files or Java classes to specify the mappings.

Let's build a simple application to be used in Canada. Since Canada has two official languages, we want to let the user choose her favorite language. Designing our application, we decided to have it just output "Hello Java" to show off how cool it is. We can always add more text later.

We are going to externalize everything language specific to special property files. They're just property files that contain keys and string values to display, but they follow very specific, `ResourceBundle`-required naming conventions. Here are two simple resource bundle files:

A file named `Labels_en.properties` that contains a single line of data:

```
hello=Hello Java!
```

A second file named `Labels_fr.properties` that contains a single line of data:

```
hello=Bonjour Java!
```

It's critical to understand that when you use `Properties` files to support `ResourceBundle` objects, the naming of the files MUST follow two rules:

1. These files must end in "`.properties`."
2. The end of the name before the `.properties` suffix must be a string that starts with an underscore and then declares the Locale the file represents (e.g., `MyApp_en.properties` or `MyApp_fr.properties` or `MyApp_fr_CA.properties`). `ResourceBundle` only knows how to find the appropriate file via the filename. There is no requirement for the data in the file to contain locale information.

Using a resource bundle requires three steps: obtaining the `Locale`, getting the `ResourceBundle`, and looking up a value from the resource bundle. First, we create a `Locale` object. To review, this means one of the following:

```
new Locale("en")          // language - English
new Locale("en", "CA")    // language and country - Canadian English
Locale.CANADA             // constant for common locales - Canadian English
```

Next, we need to create the resource bundle. We need to know the bundle name of the resource bundle and the locale. The bundle name of the resource bundle is that part of the filename up to (but not including) the underscore that is the start of the locale info. For example, if a `Properties` file is named `MyApp_en.properties`, then the bundle name is "`MyApp`." Then we pass those values to a factory, which creates the resource bundle. The `getBundle()` method looks in the classpath for bundles that match the bundle name (in the code below, the bundle name is `"Labels"`) and the provided `locale`.

```
ResourceBundle rb = ResourceBundle.getBundle("Labels", locale);
```

Finally, we use the resource bundle like a map and get a value based on the key:

```
rb.getString("hello");
```

So, back to our example, we have two files: `Labels_en.properties` and `Labels_fr.properties`. The following code takes a `locale` argument and builds a `ResourceBundle` object that's tied to the `Properties` file containing data for that `Locale` and read from the "resource bundle":

```
import java.util.Locale;
import java.util.ResourceBundle;

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

```
        Locale locale = new Locale(args[0]);
        ResourceBundle rb = ResourceBundle.getBundle("Labels", locale);
        System.out.println(rb.getString("hello"));
    }
}
```

Running the code twice, we get

```
> java WhichLanguage en
Hello Java!
> java WhichLanguage fr
Bonjour Java!
```

The Java API for `java.util.ResourceBundle` lists three good reasons to use resource bundles. Using resource bundles "allows you to write programs that can

- ***Be easily localized, or translated, into different languages***
- ***Handle multiple locales at once***
- ***Be easily modified later to support even more locales"***

If you encounter any questions on the exam that ask about the advantages of using resource bundles, this quote from the API will serve you well.

The most common use of localization in Java is web applications. You can get the user's locale from information passed in the request rather than hard-coding it.

Java Resource Bundles

When we need to move beyond simple property file key to string value mappings, we can use resource bundles that are Java classes. We write Java classes that extend `ListResourceBundle`. The class name is similar to the one for property files. Only the extension is different.

```
import java.util.ListResourceBundle;
public class Labels_en_CA extends ListResourceBundle {
  protected Object[][] getContents() {
    return new Object[][] {
       { "hello", new StringBuilder("from Java") }
     };
  }
}
```

We implement `ListResourceBundle`'s one required method that returns an array of arrays. The inner array is key/value pairs. The outer array accumulates such pairs. Notice that now we aren't limited to `String` values. We can call `getObject()` to get a non-`String` value:

```
Locale locale = new Locale("en", "CA");
ResourceBundle rb = ResourceBundle.getBundle("Labels", locale);
System.out.println(rb.getObject("hello"));
```

which prints `"from Java"`.

Default Locale

What do you think happens if we call `ResourceBundle.getBundle("Labels")` without any locale? It depends. Java will pick the resource bundle that matches the locale the JVM is using. Typically, this matches the locale of the machine running the program, but it doesn't have to. You can even change the default locale at runtime, which might be useful if you are working with people in different locales so you can get the same behavior on all machines.

Let's explore the API to get and set the default locale:

```
// store locale so can put it back at end
Locale initial  = Locale.getDefault();
System.out.println(initial);

// set locale to Germany
Locale.setDefault(Locale.GERMANY);
System.out.println(Locale.getDefault());

// put original locale back
Locale.setDefault(initial);
System.out.println(Locale.getDefault());
```

which on our computer prints:

```
en_US
de_DE
en_US
```

For the first and last line, you may get different output depending on where you live. The key is that the middle of the program executes as if it were in Germany, regardless of where it is actually being run. It is good practice to

restore the default unless your program is ending right away. That way, the rest of your code works normally—it probably doesn't expect to be in Germany.

Choosing the Right Resource Bundle

There are two main ways to get a resource bundle:

```
ResourceBundle.getBundle(baseName)
ResourceBundle.getBundle(baseName, locale)
```

Luckily, `ResourceBundle.getBundle(baseName)`is just shorthand for `ResourceBundle.getBundle(baseName, Locale.getDefault())`, and you only have to remember one set of rules. There are a few other overloaded signatures for `getBundle()`, such as taking a `ClassLoader`. But don't worry—these aren't on the exam.

Now on to the rules. How does Java choose the right resource bundle to use? In a nutshell, Java chooses the most specific resource bundle it can while giving preference to Java `ListResourceBundle`.

Going back to our Canadian application, we decide to request the Canadian French resource bundle:

```
Locale locale = new Locale("fr", "CA");
ResourceBundle rb = ResourceBundle.getBundle("RB", locale);
```

Java will look for the following files in the classpath in this order:

```
RB_fr_CA.java          // exactly what we asked for
RB_fr_CA.properties

RB_fr.java             // couldn't find exactly what we asked for
RB_fr.properties       // now trying just requested language
RB_en_US.java          // couldn't find French
RB_en_US.properties    // now trying default Locale
RB_en.java             // couldn't find full default Locale country
RB_en.properties       // now trying default Locale language
RB.java                // couldn't find anything any matching Locale,
RB.properties          // now trying default bundle
```

If none of these files exist, Java gives up and throws a `MissingResourceException`. Although this is a lot of things for Java to try, it is pretty easy to remember. Start with the full `Locale` requested. Then fall back to just language. Then fall back to the default `Locale`. Then fall back to the default bundle. Then cry.

Make sure you understand this because it is about to get more complicated.

You don’t have to specify all the keys in all the property files. They can inherit from each other. This is a good thing, as it reduces duplication.

```
RB_en.properties
  ride.in=Take a ride in the

RB_en_US.properties
  elevator=elevator
```

```
RB_en_UK.properties
  elevator=lift

Locale locale = new Locale("en", "UK");
ResourceBundle rb = ResourceBundle.getBundle("RB", locale);
System.out.println(rb.getString("ride.in") + " " +
                   rb.getString("elevator"));
```

Outputs:

```
Take a ride in the lift
```

The common `"ride.in"` property comes from the parent noncountry-specific bundle "`RB_en.properties`." The `"elevator"` property is different by country and comes from the UK version that we specifically requested.

The parent hierarchy is more specific than the search order. A bundle's parent always has a shorter name than the child bundle. If a parent is missing, Java just skips along that hierarchy. `ListResourceBundles` and `PropertyResourcesBundles` do not share a hierarchy. Similarly, the default locale's resource bundles do not share a hierarchy with the requested locale's resource bundles. Table 4-3 shows examples of bundles that do share a hierarchy.

TABLE 4-3 Resource Bundle Lookups

Name of Resource Bundle	Hierarchy
RB_fr_CA.java	RB.java RB_fr.java RB_fr_CA.java
RB_fr_CA.properties	RB.properties RB_fr.properties RB_fr_CA.properties
RB_en_US.java	RB.java RB_en.java RB_en_US.java
RB_en_US.properties	RB.properties RB_en.properties RB_en_US.properties

Remember that searching for a property file uses a linear list. However, once a matching resource bundle is found, keys can only come from that resource bundle's hierarchy.

One more example to make this clear. Think about which resource bundles will be used from the previous code if we use the following code to request a resource bundle:

```
Locale locale = new Locale("fr", "FR");
ResourceBundle  rb = ResourceBundle.getBundle("RB", locale);
```

First, Java looks for `RB_fr_FR.java` and `RB_fr_FR.properties`. Because

neither is found, Java falls back to using `RB_fr.java`. Then as we request keys from `rb`, Java starts looking in `RB_fr.java` and additionally looks in `RB.java`. Java started out looking for a matching file and then switched to searching the hierarchy of that file.

CERTIFICATION SUMMARY

Dates and Times The `Date` and `Calendar` classes, as well as `DateFormat`, have all been replaced by classes in the `java.time` package, so pay close attention if you're transitioning from the old classes to the new. The key datetime classes to know from `java.time` are `LocalDate`, `LocalTime`, `LocalDateTime`, and `ZonedDateTime`. Each has a variety of methods to create and adjust datetime objects. You also need to know about `TemporalAdjusters` (like `TemporalAdjuster.firstDayOfMonth()`) and `TemporalUnits` (like `ChronoUnit.DAYS`), both from the `java.time.temporal` package, and `Instants`, `Periods`, and `Durations`, in the `java.time` package. `DataFormat` has been replaced with `DateTimeFormatter` in the `java.time.format` package, which is used to parse, format, and print datetime objects. The `Locale` class is used with `DateTimeFormatter` to generate a variety of output styles that are language and/or country specific.

Make sure you are clear on how to work with time zones and daylight savings time. Fortunately, the `ZonedDateTime` and related classes handle most of the hard work for you, but pay close attention to the format of the datetimes when they are represented as strings so you can recognize a local datetime from a zoned datetime and so you know how to create a `ZonedDateTime` using a `ZoneId`.

Locales, Properties Files, and Resource Bundles Resource bundles allow you to move locale-specific information (usually strings) out of your code and into external files where they can easily be amended. This provides an easy way for you to localize your applications across many locales. Properties files allow you to create text files formatted as key/value pairs to store application customization parameters and such external to your application. The `ResourceBundle` class provides convenient ways to use files that are `Properties` class–compatible to store internationalization and localization values.

TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

Dates and Times (OCP Objectives 7.1, 7.2, and 7.3)

- The classes you need to understand are those in `java.time`, `java.time.temporal`, and `java.time.format`, as well as `java.util.Locale`.
- `Date` and `Calendar` are no longer used, and most of the `Date` class's methods have been deprecated.
- A `LocalDate` is a date, and a `LocalTime` is a time. Combine the two to make a `LocalDateTime`. None of these types have a time zone associated with them.
- A `ZonedDateTime` is a datetime object with a time zone. All zoned datetimes are relative to Greenwich Mean Time (GMT). You may sometimes see GMT written as UTC.
- A `ZoneId` can be created from a string representing a time zone (e.g. `"US/Pacific"`).
- When you adjust `ZonedDateTimes`, daylight savings time will be automatically handled using the `ZoneRules`.
- If you want a datetime object with a time zone that is independent of zone rules, use an `OffsetDateTime`.
- A `Period` is a period of time that is a day or longer.
- A `Duration` is a period of time that is shorter than a day.
- An `Instant` is an instant in time and represents the number of seconds and nanoseconds since January 1, 1970. You can get the number of seconds as a `long` value from an `Instant` and convert any `ZonedDateTime` object into an `Instant`.
- There are several format "styles" available in the `java.format` class. You can use format styles such as `FormatStyle.SHORT` with `DateTimeFormatter` to format datetime objects.

- The `DateTimeFormatter` class is used to parse and create strings containing properly formatted dates.
- The `Locale` class is used in conjunction with `DateFormat` and `NumberFormat`.
- A `DateTimeFormatter` object can be constructed with a specific, immutable `Locale`.
- For the exam, you should understand creating `Locale`s using either language or a combination of language and country.

Locales, Properties Files, and Resource Bundles (OCP 12.1, 12.2, and 12.3)

- The `java.util.Properties` class gives you a convenient way to create and maintain text files that are external to your applications and can hold configuration values.
- A file that is `java.util.Properties`–compliant and has a name that ends with a locale and a suffix of `.properties` can be used by `ResourceBundle.getBundle()`.
- A `ListResourceBundle` comes from Java classes, and a `PropertyResourceBundle` comes from `.properties` files.
- `ResourceBundle.getBundle(name)`uses the default `Locale`.
- `Locale.getDefault()`returns the JVM's default `Locale`. `Locale.setDefault(locale)` can change the JVM's `locale`.
- Java searches for resource bundles in this order: requested language/country, requested language, default locale language/country, default locale language, default bundle. Within each item, Java `ListResourceBundle` is favored over `PropertyResourceBundle`.
- Once a `ResourceBundle` is found, only parents of that bundle can be used to look up keys.

SELF TEST

1. Given the code fragment:

```
       ZonedDateTime zd = ZonedDateTime.parse("2020-05-
04T08:05:00");
      System.out.println(zd.getMonth() + " " +
zd.getDayOfMonth());
```

What is the result? (Choose all that apply.)

A. `MAY 4`

B. `APRIL 5`

C. `MAY 4 2020`

D. `APRIL 5 2020`

E. Compilation fails

F. Runtime exception

2. Given the code fragment:

```
LocalTime t1 = LocalTime.of(9, 0);
LocalTime t2 = LocalTime.of(10, 5);
```

Which of the following code fragment(s) will produce a new `LocalTime t3` that represents the same time as `t2`? (Choose all that apply.)

A. `LocalTime t3 = t1.plus(65, ChronoUnit.MINUTES)`

B. `LocalTime t3 = t1.plusMinutes(65);`

C. `LocalTime t3 = t1.plusHours(1);`

D. `LocalTime t3 = t1.plusDays(1);`

E. `LocalTime t3 = t1.plus(Duration.ofMinutes(65));`

3. Given the code fragment:

```
1. LocalDate d1 = LocalDate.of(2018, 1, 1);
2. LocalDate d2 = LocalDate.of(2018, 6, 15);
3. __________ r = __________.between(d1, d2);
4. System.out.println("Months and days: " + r.getMonths() + ", " + r.getDays());
```

What are the correct types to fill in the blanks on line 3?

A. `Duration, Duration`

B. `Instant, Period`

C. `Period, Instant`

D. `Period, ChronoUnit`

E. `Period, Period`

F. `Duration, LocalDate`

4. How would you use `nowzdt` from the code fragment below to compute the equivalent time in Berlin, Germany? (Choose all that apply.)

```
ZonedDateTime nowzdt =
    ZonedDateTime.of(LocalDateTime.now(), ZoneId.of("US/Pacific"));
```

A.

```
ZonedDateTime berlinZdt = ZonedDateTime.from(nowzdt, ZoneId.of("Europe/Berlin"));
```

B.

```
ZonedDateTime berlinZdt = nowzdt.withZoneSameInstant(ZoneId.of("Europe/Berlin"));
```

C.

```
ZonedDateTime berlinZdt =
    ZonedDateTime.ofInstant(nowzdt.toInstant(), ZoneId.of("Europe/Berlin"));
```

D.

```
ZonedDateTime berlinZdt =
    nowzdt.withZoneId("Europe/Berlin"));
```

E.

```
ZonedDateTime berlinZdt = nowzdt.now(ZoneId.of("Europe/Berlin"));
```

5. The next total solar eclipse visible in South America is on July 2, 2019, at 16:55 UTC. Which code fragment will correctly compute and display the time in San Juan, Argentina, for this solar eclipse?

A.

```
ZonedDateTime totalityUTC = ZonedDateTime.of(
    LocalDateTime.of(2019, 7, 2, 16, 55));
ZonedDateTime totalitySanJuan =
    totalityUTC.withZoneSameInstant(ZoneId.of("America/Argentina/San_Juan"));
System.out.println(totalitySanJuan);
```

B.

```
ZonedDateTime totalityUTC = ZonedDateTime.of(
    LocalDateTime.of(2019, 7, 2, 4, 55, "PM") , ZoneId.of("Z"));
ZonedDateTime totalitySanJuan =
    totalityUTC.withZoneSameInstant(ZoneId.of("America/Argentina/San_Juan"));
System.out.println(totalitySanJuan);
```

C.

```
ZonedDateTime totalityUTC = ZonedDateTime.of(
    LocalDateTime.of(2019, 7, 2, 16, 55), ZoneId.of("Z"));
ZonedDateTime totalitySanJuan =
    totalityUTC.withZoneSameInstant(ZoneId.of("America/Argentina/San_Juan"));
System.out.println(totalitySanJuan);
```

D.

```
ZonedDateTime totalityUTC = ZonedDateTime.of(
    LocalDateTime.of(2019, 7, 2, 16, 55),
    ZoneId.of("America/Argentina/San_Juan"));
ZonedDateTime totalitySanJuan =
    totalityUTC.withZoneSameInstant(ZoneId.of("America/Argentina/San_Juan"));
System.out.println(totalitySanJuan);
```

E.

```
ZonedDateTime totalityUTC = ZonedDateTime.of(
    LocalDateTime.of(2019, 7, 2, 16, 55), ZoneId.of("Z "));
LocalDateTime totalitySanJuan =
    totalityUTC.withZoneSameInstant(ZoneId.of("America/Argentina/San_Juan"));
System.out.println(totalitySanJuan);
```

6. Given:

```
public class Canada {
  public static void main(String[] args) {
    ResourceBundle rb = ResourceBundle.getBundle("Flag",
      new Locale("en", "CA"));
    System.out.println(rb.getString("key"));
  }
}
```

Assume the default `Locale` is Italian. If each of the following is the only resource bundle on the classpath and contains `key=value`, which will be used? (Choose all that apply.)

A. `Flag.java`

B. `Flag_CA.properties`

C. `Flag_en.java`

D. `Flag_en.properties`

E. `Flag_en_CA.properties`

F. `Flag_fr_CA.properties`

7. Given three resource bundles and a Java class:

```
Train_en_US.properties: train=subway
Train_en_UK.properties: train=underground
Train_en.properties: ride = ride


1: public class ChooChoo {
2:   public static void main(String[] args) {
3:     Locale.setDefault(new Locale("en", "US"));
4:     ResourceBundle rb = ResourceBundle.getBundle("Train",
5:       new Locale("en", "US"));
6:     System.out.print(rb.getString("ride")
         + " " + rb.getString("train"));
7:   }
8: }
```

Which of the following, when made independently, will change the output to “ride underground”? (Choose all that apply.)

A. Add `train=underground` to `Train_en.properties`

B. Change line 3 to `Locale.setDefault(new Locale("en", "UK"));`

C. Change line 5 to `Locale.ENGLISH);`

D. Change line 5 to new `Locale("en", "UK"));`

E. Delete file `Train_en_US.properties`

8. Let's say you want to print the day of the week and the date of Halloween (October 31) 2018, at 5 `PM` in German, using the `LONG` style. Complete the code below using the following fragments. Note: You can use each fragment either zero or more times, and you might not need to fill all of the slots. You probably won't encounter a fill-in-the-blank question on the exam, but just in case, we put a few in the book, like this one.

Code:

```
import java._________
import java._________
import java._________

public class DateHalloween {
   public static void main(String[] args) {
      ZonedDateTime d = _____________________
      Locale locDE = new Locale("de");
      DayOfWeek day = _________________
      String df = ____________________________________________
      System.out.println(day + " " + df);
   }
}
```

Fragments:

```
io.*;
nio.*;
util.*;
time.*;
date.*;
time.format.*;
new ZonedDateTime(2018, 10, 31, 17, 0);
new LocalDate(2018, 10, 31, 17, 0);
ZonedDateTime.of(2018, 10, 31, 17, 0, 0, ZoneId.of("Europe/Berlin"));
ZonedDateTime.of(2018, 10, 31, 17, 0, 0, 0,
                 ZoneId.of("Europe/Berlin"));
d.getDayOfWeek();
d.getDay();
DateTimeFormatter.of(FormatStyle.LONG).withLocale(locDE);
d.format(DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG)
                          .withLocale(locDE));
```

9. Given two files:

```
package rb;
public class Bundle extends java.util.ListResourceBundle {
  protected Object[][] getContents() {
    return new Object[][] { { "123", 456 } };
  }
}
```

```
package rb;
import java.util.*;
public class KeyValue {
  public static void main(String[] args) {
    ResourceBundle rb = ResourceBundle.getBundle("rb.Bundle",
      Locale.getDefault());
    // insert code here
  }
}
```

Which, inserted independently, will compile? (Choose all that apply.)

A. `Object obj = rb.getInteger("123");`

B. `Object obj = rb.getInteger(123);`

C. `Object obj = rb.getObject("123");`

D. `Object obj = rb.getObject(123);`

E. `Object obj = rb.getString("123");`

F. `Object obj = rb.getString(123);`

10. Given the following code fragment:

```
LocalDateTime now = LocalDateTime.of(2017,10,27,14,22,54,0);
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("_______________");   // L1
String formattedDateTime = now.format(formatter);
System.out.println("Formatted DateTime: " + formattedDateTime);
```

Which `String` inserted as an argument to `DateTimeFormatter.ofPattern()` at `// L1` will produce the output? (Choose all that apply.)

Formatted DateTime: 2017-10-27 14:22:54

A. `"yyyy-MM-dd hh:mm:ss a"`

B. `"yyyy-MM-dd hh:mm:ss"`

C `"yyyy-mm-dd HH:MM:ss"`

D. `"yyyy-MM-dd HH:mm:ss"`

E. `"yyyy-MM-dd HH:mm:ss Z"`

11. Given the following code fragment:

```
LocalDate d1 = LocalDate.of(2017, Month.NOVEMBER, 28);
System.out.print(d1 + ", ");
LocalDate d2 = d1.with(TemporalAdjusters.lastDayOfYear());
System.out.print(d2 + ", ");
LocalDate d3 = d1.plusDays(3).with(TemporalAdjusters.firstDayOfNextMonth());
System.out.print(d3 + ", ");
LocalDate d4 = d1.minusMonths(11).with(TemporalAdjusters.firstDayOfNextYear());
System.out.print(d4 + ", ");
LocalDate d5 = LocalDate.ofEpochDay(d1.plusDays(27).toEpochDay());
System.out.print(d5 + ", ");
LocalDate d6 = d1.minus(Period.ofDays(5));
System.out.println(d6);
```

What output will you see?

A. `2017-11-28, 2017-12-31, 2017-12-01, 2017-01-01, 2017-12-25, 2017-11-23`

B. `2017-11-28T00:00, 2017-12-31T00:00, 2017-12-01T00:00, 2017-01-01T00:00, 2017-12-25T00:00, 2017-11-23T00:00`

C. `2017-11-28, 2017-12-31, 2018-01-01, 2017-01-01, 2017-12-25, 2017-11-23`

D. `2017-11-28T00:00, 2017-12-31T00:00, 2018-01-01T00:00, 2017-01-01T00:00, 2017-12-25T00:00, 2017-11-23T00:00`

E. `2017-11-28, 2017-12-31, 2018-01-01, 2018-01-01, 2017-12-25, 2017-11-23`

12. If it is 19:12:53 on October 27, 2017, in the US/Pacific Zone (which is GMT-8:00, summer time), then what does the following code fragment produce? (Choose all that apply.)

```
ZoneId zid = ZoneId.of("US/Eastern"); // GMT-5:00
Instant i = Instant.now();
ZonedDateTime zdt = i.atZone(zid);
System.out.println(zdt.format(
    DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM)));
```

A. `10:12:53 PM`

B. `20:12:53`

C. `19:12:53`

D. `7:12:53 PM`

E. `2017-10-27 10:12:53 PM`

A SELF TEST ANSWERS

1. ☑ **F** is correct. The string we are parsing has no time zone, so the parse will fail at runtime.

   ☒ **A, B, C, D,** and **E** are incorrect based on the above. (OCP Objective 7.2)

2. ☑ **A, B,** and **E** are correct. Each adds 1 hour and 5 minutes to `t1` to make a new `LocalTime` `t3`, which represents 10:05, the same time as `t2`. The `plus()` method takes an amount to add as a `long`, and a unit **(A)** or a `TemporalAmount` **(E)**.

   ☒ **C** and **D** are incorrect. **C** adds only 1 hour to make 10 `AM` instead of 10:05 `AM`. **D** generated a compile error because `LocalTime` does not have a `plusDays()` method. (OCP Objective 7.1 and 7.3)

3. ☑ **E** is correct. Period is the correct type to measure a period of time in days.

   ☒ **A, B, C, D,** and **F** are incorrect based on the above. (OCP Objective 7.1 and 7.3)

4. ☑ **B** and **C** are correct. In both cases, we're creating an `Instant` from

`nowzdt` and then creating a new `ZonedDateTime` from that `Instant`, representing the same time as `nowzdt`, in Berlin.

☒ **A, D,** and **E** are incorrect. **A** is incorrect because, although you can create a new `ZonedDateTime` from an existing `ZonedDateTime` with `from()`, you can't change the zone when you do. **D** is incorrect because `withZoneId()` is not a valid method. **E** is almost correct, except that it is not precisely the same time as `nowzdt` because you're calling the `now()` method again, though it may only be slightly different (perhaps only a few nanoseconds). (OCP Objective 7.2)

5. ☑ **C** is correct. We first create a `ZonedDateTime` for the UTC time with zone "`Z`" (corresponding to GMT zone) and then create the equivalent `ZonedDateTime` for the San Juan, Argentina, zone.

☒ **A, B, D,** and **E** are incorrect. **A** is missing the time zone on the UTC time. **B** includes incorrect arguments to the `LocalDateTime.of()` method. **D** has the incorrect time zone on the UTC time. **E** has the incorrect type for `totalitySanJuan`. (OCP Objective 7.2)

6. ☑ **A, C, D,** and **E** are correct. The default `Locale` is irrelevant here since none of the choices use Italian. **A** is the default resource bundle. **C** and **D** use the language but not the country from the requested locale. **E** uses the exact match of the requested locale.

☒ **B** is incorrect because the language code of `CA` does not match `en`. And `CA` isn't a valid language code. **F** is incorrect because the language code `"fr"` does not match `en`. Even though the country code of `CA` does match, the language code is more important. (OCP Objectives 12.2 and 12.3)

7. ☑ **D** is correct. As is, the code finds resource bundle `Train_en_US.properties`, which uses `Train_en.properties` as a parent. Choice **D** finds resource bundle `Train_en_UK.properties`, which uses `Train_en.properties` as a parent.

☒ **A, B, C, E,** and **F** are incorrect. **A** is incorrect because both the parent and child have the same property. In this scenario, the more specific one (child) gets used. **B** is incorrect because the default locale only gets used if the requested resource bundle can't be found. **C** is incorrect because it finds the resource bundle `Train_en.properties`,

which does not have any “train” key. **E** is incorrect because there is no “ride” key once we delete the parent. **F** is incorrect based on the above. (OCP Objectives 12.2 and 12.3)

8. Answer:

```
import java.util.*;
import java.time.*;
import java.time.format.*;
public class DateHalloween {
   public static void main(String[] args) {
      ZonedDateTime d = ZonedDateTime.of(2018, 10, 31, 17, 0, 0, 0,
                              ZoneId.of("Europe/Berlin"));
      Locale locDE = new Locale("de");
      DayOfWeek day = d.getDayOfWeek();
      String df = d.format(DateTimeFormatter
                  .ofLocalizedDateTime(FormatStyle.LONG).withLocale(locDE));
      System.out.println(day + " " + df);
   }
}
```

Reminders: To create a `ZonedDateTime` with the `of()` method, you must include all portions of the date and time (including nanoseconds) and a zone. `DateTimeFormatter .ofLocalizedDateTime()` returns a locale-specific date-time formatter, and `withLocale()` returns a copy of this formatter with a new locale. (OCP Objectives 7.2 and 12.1)

9. ☑ **C** and **E** are correct. When getting a key from a resource bundle, the key must be a string. The returned result must be a string or an object. While that object may happen to be an integer, the API is still

`getObject()`. **E** will throw a ClassCastException since 456 is not a string, but it will compile.

☒ **A, B, D,** and **F** are incorrect because of the above. (OCP Objectives 12.2 and 12.3)

10. ☑ **D** is correct; this string corresponds to the format shown in the output.

☒ **A, B, C,** and **E** are incorrect. **A** uses `hh` for the hour, which will show `02` instead `14` (that is, a 12-hour format instead of a 24-hour format), and displays the `AM/PM` at the end, which is great if we're using 12-hour format, but that's not what we're looking for. **B** results in 12-hour format instead of 24-hour format. **C** switches months and minutes. **E** requires a `ZonedDateTime` instead of a `LocalDateTime`, and using this `String` will throw a runtime exception when we try to format `now` with this formatter. (OCP Objectives 7.1 and 7.2)

11. ☑ **C** is correct because of the below.

☒ **A, B, D,** and **E** are incorrect. **B** and **D** show the time, and we are displaying `LocalDate` values that have no time associated with them. **A** has the incorrect value for `d3`, and **E** has the wrong value for `d4`. (OCP Objectives 7.1 and 7.3)

q

12. ☑ **A** is correct. We first get the `zoneId` for "`US/Eastern`" time, which is GMT-5:00, and the locale to US. We then create an `Instant` for "now," which is 19:12:53 on October 27, 2017 (7:12:53 PM PDT, which is 10:12:53 PM EDT). We then create a `ZonedDateTime` from the `Instant`, using the `zoneId` for `"US/Eastern"` and format it using `DateTimeFormatter.ofLocalizedTime()`, which turns the `ZonedDateTime` into a `LocalTime` (dropping the date and zone information) and display it in the `MEDIUM` format style for the US locale, resulting in `10:12:53 PM`. Format styles depend on local configuration, but we know this answer is correct because **B, C,** and **D** show the incorrect times, and **E** shows the date.

☒ **B, C, D,** and **E** are incorrect. **B, C,** and **D** show the incorrect times, and **E** shows the date, which we dropped when we formatted `zdt` to a localized time. (OCP Objectives 7.2)