Date and Time 17

Chapter Topics

- Overview of the new Date and Time API in the `java.time` package
- Understanding the temporal concepts represented by the `LocalTime`, `LocalDate`, `LocalDateTime`, `ZonedDateTime`, `Instant`, `Period`, and `Duration` classes
- Creating and using temporal objects
- Accessing temporal objects using temporal units and temporal fields
- Comparing temporal objects
- Creating modified instances of temporal objects
- Performing temporal arithmetic with temporal objects
- Using time zones and daylight savings with `ZonedDateTime` objects.
- Interoperability between date/time values and legacy dates

Java SE 17 Developer Exam Objectives	
[1.3] Manipulate date, time, duration, period, instant and time-zone objects using Date-Time API	§17.1, p. 1024 *to* §17.7, p. 1072

Java 8 introduced a new and comprehensive API for date and time. This chapter provides comprehensive coverage of essential topics regarding the Date and Time API. Coverage of formatting and parsing of date and time values is deferred to §18.6, p. 1127.

17.1 Date and Time API Overview

The `java.time` package provides the main support for dealing with dates and times. It contains the main classes that represent date and time values, including those that represent an amount of time.

- `LocalDate`: This class represents a date in terms of *date fields* (year, month, day). A *date* has no time fields or a time zone. (This class is not to be confused with the `java.util.Date` legacy class.)
- `LocalTime`: This class represents time in a 24-hour day in terms of *time fields* (hour, minute, second, nanosecond). A *time* has no date fields or a time zone.
- `LocalDateTime`: This class represents the concept of date and time combined, in terms of *both* date and time fields. A *date-time* has no time zone.

- `ZonedDateTime`: This class represents the concept of a date-time with a time zone— that is, a *zoned date-time*.
- `Instant`: This class represents a measurement of time as a point on a timeline starting from a specific origin (called the *epoch*). An *instant* is represented with nanosecond precision and can be a negative value.
- `Period`: This class represents an amount or quantity of time in terms of number of days, months, and years, which can be negative. A *period* is a *date-based* amount of time. It has no notion of a clock time, a date, or a time zone.
- `Duration`: This class represents an amount or quantity of time in terms of number of seconds and nanoseconds, which can be negative. A *duration* is a *time-based* amount of time. As with instants, durations have no notion of a clock time, a date, or a time zone.

We will use the term *temporal objects* to mean objects of classes that represent temporal concepts.

The temporal classes implement *immutable* and *thread-safe* temporal objects. The state of an immutable object cannot be changed. Any method that is supposed to modify such an object returns a modified copy of the temporal object. It is a common mistake to ignore the object returned, thinking that the current object has been modified. Thread-safety guarantees that the state of such an object is not compromised by concurrent access.

Table 17.1 summarizes the fields in selected classes from the Date and Time API. The table shows the relative size of the objects of these classes in terms of their fields; for example, a `LocalTime` has only time fields, whereas a `ZonedDateTime` has time-, date-, and zone-based fields. The three asterisks `***` indicate that this information can be derived by methods provided by the class, even though these fields do not exist in an object of the `Duration` class.

Table 17.1 *Fields in Selected Classes in the Date and Time API*

Classes	Year	Month	Day	Hours	Minutes	Seconds/Nanos	Zone offset	Zone ID
`LocalTime` (p. 1027)				`+`	`+`	`+`		
`LocalDate` (p. 1027)	`+`	`+`	`+`					
`LocalDateTime` (p. 1027)	`+`	`+`	`+`	`+`	`+`	`+`		
`ZonedDateTime` (p. 1072)	`+`	`+`	`+`	`+`	`+`	`+`	`+`	`+`

Classes	Year	Month	Day	Hours	Minutes	Seconds/Nanos	Zone off-set	Zone ID
`Instant` (p. 1049)						+		
`Period` (p. 1057)	+	+	+					
`Duration` (p. 1064)			***	***	***	+		

The information in **Table 17.1** is crucial to understanding how the objects of these classes can be used. A common mistake is to access, format, or parse a temporal object that does not have the required temporal fields. For example, a `LocalTime` object has only time fields, so trying to format it with a formatter for date fields will result in a `java.time.DateTimeException`. Many methods will also throw an exception if an invalid or an out-of-range argument is passed in the method call. It is important to keep in mind which temporal fields constitute the state of a temporal object.

Table 17.2 provides an overview of the method naming conventions used in the temporal classes `LocalTime`, `LocalDate`, `LocalDateTime`, `ZonedDateTime`, and `Instant`. This method naming convention makes it easy to use the API, as it ensures method naming is standardized across all temporal classes. Depending on the method, the suffix *`XXX`* in a method name can be a specific field (e.g., designate the `Year` field in the `getYear` method name), a specific unit (e.g., designate the unit `Days` for number of days in the `plusDays` method name), or a class name (e.g., designate the class type in the `toLocalDate` method name).

Table 17.2 *Selected Method Name Prefixes in the Temporal Classes*

Prefix (parameters not shown)	Usage
`at`*`XXX`*`()`	Create a new temporal object by combining this temporal object with another temporal object. Not provided by the `ZonedDateTime` class.
`of()` `of`*`XXX`*`()`	Static factory methods for constructing temporal objects from constituent temporal fields.
`get()` `get`*`XXX`*`()`	Access specific fields in this temporal object.

Prefix (parameters not shown)	Usage
`is`*XXX*`()`	Check specific properties of this temporal object.
`minus()` `minus`*XXX*`()`	Return a copy of this temporal object after subtracting an amount of time.
`plus()` `plus`*XXX*`()`	Return a copy of this temporal object after adding an amount of time.
`to`*XXX*`()`	Convert this temporal object to another type.
`with()` `with`*XXX*`()`	Create a copy of this temporal object with one field modified.

Apart from the methods shown in **Table 17.2**, the selected methods shown in **Table 17.3** are common to the temporal classes `LocalTime`, `LocalDate`, `LocalDateTime`, `ZonedDateTime`, and `Instant`.

Table 17.3 *Selected Common Methods in the Temporal Classes*

Method (parameters not shown)	Usage
`now()`	Static method that obtains the current time from the system or specified clock in the default or specified time zone.
`from()`	Static method to obtain an instance of this temporal class from another temporal.
`until()`	Calculate the amount of time from this temporal object to another temporal object.
`toString()`	Create a text representation of this temporal object.
`equals()`	Compare two temporal objects for equality.
`hashCode()`	Returns a hash code for this temporal object.
`compareTo()`	Compare two temporal objects. (The class `ZonedDateTime` does not implement the `Comparable<E>` interface.)

Method (parameters not shown)	Usage
`parse()`	Static method to obtain a temporal instance from a specified text string (§18.6, p. 1127).
`format()`	Create a text representation of this temporal object using a specified formatter (§18.6, p. 1127). (`Instant` class does not provide this method.)

Subsequent sections in this chapter provide ample examples of how to create, combine, convert, access, and compare temporal objects, including the use of temporal arithmetic and dealing with time zones and daylight savings. For formatting and parsing temporal objects, see §18.6, p. 1127.

17.2 Working with Dates and Times

The classes `LocalTime`, `LocalDate`, and `LocalDateTime` in the `java.time` package represent time-based, date-based, and combined date-based and time-based temporal objects, respectively, that are all time zone agnostic. These classes represent *human time* that is calendar-based, meaning it is defined in terms of concepts like year, month, day, hour, minute, and second, that humans use. The `Instant` class can be used to represent *machine time*, which is defined as a point measured with nanosecond precision on a continuous timeline starting from a specific origin (p. 1049).

Time zones and daylight savings are discussed in §17.7, p. 1072.

Creating Dates and Times

The temporal classes in the `java.time` package do not provide any `public` constructors to create temporal objects. Instead, they provide overloaded static factory methods named `of` which create temporal objects from constituent temporal fields. We use the term *temporal fields* to mean both time fields (hours, minutes, seconds, nanoseconds) and date fields (year, month, day). The `of()` methods check that the values of the arguments are in range. Any invalid argument results in a `java.time.DateTimeException`.

Click here to view code image

```
// LocalTime
static LocalTime of(int hour, int minute)
static LocalTime of(int hour, int minute, int second)
static LocalTime of(int hour, int minute, int second, int nanoOfSecond)
static LocalTime ofSecondOfDay(long secondOfDay)
```

These static factory methods in the `LocalTime` class return an instance of `Local-Time` based on the specified values for the specified time fields. The second and nanosecond fields are set to zero, if not specified.

The last method accepts a value for the `secondOfDay` parameter in the range [0, 24 * 60 * 60 – 1] to create a `LocalTime`.

Click here to view code image

```
// LocalDate
static LocalDate of(int year, int month, int dayOfMonth)
static LocalDate of(int year, Month month, int dayOfMonth)
static LocalDate ofYearDay(int year, int dayOfYear)
```

These static factory methods in the `LocalDate` class return an instance of `LocalDate` based on the specified values for the date fields. The `java.time.Month` enum type allows months to be referred to by name—for example, `Month.MARCH`. Note that month numbering starts with 1 (`Month.JANUARY`).

The last method creates a date from the specified year and the day of the year.

Click here to view code image

```
// LocalDateTime
static LocalDateTime of(int year, int month, int dayOfMonth,
                        int hour, int minute)
static LocalDateTime of(int year, int month, int dayOfMonth,
                        int hour, int minute, int second)
static LocalDateTime of(int year, int month, int dayOfMonth, int hour,
                        int minute, int second, int nanoOfSecond)
static LocalDateTime of(int year, Month month, int dayOfMonth,
                        int hour, int minute, int second)
static LocalDateTime of(int year, Month month, int dayOfMonth,
                        int hour, int minute)
static LocalDateTime of(int year, Month month, int dayOfMonth,
                        int hour, int minute, int second, int nanoOfSecond)
```

These static factory methods in the `LocalDateTime` class return an instance of `LocalDateTime` based on the specified values for the time and date fields. The second and nanosecond fields are set to zero, if not specified. The `java.time.Month` enum type allows months to be referred to by name—for example, `Month.MARCH` (i.e., month 3 in the year).

Click here to view code image

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

Combines a `LocalDate` and a `LocalTime` into a `LocalDateTime`.

All code snippets in this subsection can be found in Example 17.1, p. 1031, ready for running and experimenting. An appropriate `import` statement with the `java.time` package should be included in the source file to access any of the temporal classes by their simple name.

The `LocalTime` *Class*

The declaration statements below show examples of creating instances of the `LocalTime` class to represent time on a 24-hour clock in terms of hours, minutes, seconds, and nanoseconds.

Click here to view code image

```
LocalTime time1 = LocalTime.of(8, 15, 35, 900);   // 08:15:35.000000900
LocalTime time2 = LocalTime.of(16, 45);           // 16:45
// LocalTime time3 = LocalTime.of(25, 13, 30);    // DateTimeException
```

The ranges of values for time fields hour (0–23), minute (0–59), second (0–59), and nanosecond (0–999,999,999) are defined by the ISO standard. The `toString()` method of the class will format the time fields according to the ISO standard as follows:

```
HH:mm:ss.SSSSSSSSS
```

Omitting the seconds (`ss`) and fractions of seconds (`SSSSSSSSS`) in the call to the `of()` method implies that their value is zero. (More on formatting in §18.6, p. 1134.) In the second declaration statement above, the seconds and the nanoseconds are not specified in the method call, resulting in their values being set to zero. In the third statement, the value of the hour field (25) is out of range, and if the statement is uncommented, it will result in a `DateTimeException`.

The `LocalDate` *Class*

Creating instances of the `LocalDate` class is analogous to creating instances of the `LocalTime` class. The `of()` method of the `LocalDate` class is passed values for date fields: the year, month of the year, and day of the month.

Click here to view code image

```
LocalDate date1 = LocalDate.of(1969, 7, 20);            // 1969-07-20
LocalDate date2 = LocalDate.of(-3113, Month.AUGUST, 11);// -3113-08-11
//  LocalDate date3 = LocalDate.of(2021, 13, 11);       // DateTimeException
//  LocalDate date4 = LocalDate.of(2021, 2, 29);        // DateTimeException
```

The ranges of the values for date fields year, month, and day are (–999,999,999 to +999,999,999), (1–12), and (1–31), respectively. The month can also be specified using the enum constants of the `java.time.Month` class, as in the second declaration statement above. A `DateTimeException` is thrown if the value of any parameter is out of range, or if the day is invalid for the specified month of the year. In the third declaration, the value of the month field 13 is out of range. In the last declaration, the month of February cannot have 29 days, since the year 2021 is not a leap year.

The `toString()` method of the `LocalDate` class will format the date fields according to the ISO standard (§18.6, p. 1134):

```
uuuu-MM-dd
```

The year is represented as a *proleptic year* in the ISO standard, which can be negative. A year in CE (Current Era, or AD) has the same value as a proleptic year; for example, 2021 CE is the same as the proleptic year 2021. However, for a year in BCE (Before Current Era, or BC), the proleptic year 0 corresponds to 1 BCE, the proleptic year –1 corresponds to 2 BCE, and so on. In the second declaration in the preceding set of examples, the date `-3113-08-11` corresponds to 11 August 3114 BCE.

The `LocalDateTime` *Class*

The class `LocalDateTime` allows the date and the time to be combined into one entity, which is useful for representing such concepts as appointments that require both a time and a date. The `of()` methods in the `LocalDateTime` class are combinations of the `of()` methods from the `LocalTime` and `LocalDate` classes, taking values of both time and date fields as arguments. The `toString()` method of this class will format the temporal fields according to the ISO standard (§18.6, p. 1134):

```
uuuu-MM-ddTHH:mm:ss.SSSSSSSSS
```

The letter `T` separates the values of the date fields from those of the time fields.

Click here to view code image

```
// 2021-04-28T12:15
LocalDateTime dt1 = LocalDateTime.of(2021, 4, 28, 12, 15);
// 2021-08-19T14:00
LocalDateTime dt2 = LocalDateTime.of(2021, Month.AUGUST, 19, 14, 0);
```

The `LocalDateTime` class also provides an `of()` method that combines a `LocalDate` object and a `LocalTime` object. The first declaration in the next code snippet combines a date and a time. The static field `LocalTime.NOON` defines the time at noon. In addition, the `LocalTime` class provides the instance method `atDate()`, which takes a date as an argument and returns a `LocalDateTime` object. The second declaration combines the time at noon with the date referred to by the reference `date1`. Conversely, the `LocalDate` class provides the overloaded instance method `atTime()` to combine a date with a specified time. In the last two declarations, the `atTime()` method is passed a `LocalTime` object and values for specific time fields, respectively.

Click here to view code image

```
// LocalDate date1 is 1969-07-20.
LocalDateTime dt3 = LocalDateTime.of(date1, LocalTime.NOON); // 1969-07-20T12:00
LocalDateTime dt4 = LocalTime.of(12, 0).atDate(date1);       // 1969-07-20T12:00
LocalDateTime dt5 = date1.atTime(LocalTime.NOON);            // 1969-07-20T12:00
LocalDateTime dt6 = date1.atTime(12, 0);                     // 1969-07-20T12:00
```

As a convenience, each temporal class provides a static method `now()` that reads the system clock and returns the values for the relevant temporal fields in an instance of the target class.

Click here to view code image

```
LocalTime currentTime = LocalTime.now();
LocalDate currentDate = LocalDate.now();
LocalDateTime currentDateTime = LocalDateTime.now();
```

Example 17.1 includes the different ways to create temporal objects that we have discussed so far.

Click here to view code image

```
// LocalTime
LocalDateTime atDate(LocalDate date)
```

Returns a `LocalDateTime` that combines this time with the specified date.

Click here to view code image

```
// LocalDate
LocalDateTime atTime(LocalTime time)
LocalDateTime atTime(int hour, int minute)
LocalDateTime atTime(int hour, int minute, int second)
LocalDateTime atTime(int hour, int minute, int second, int nanoOfSecond)
LocalDateTime atStartOfDay()
```

Return a `LocalDateTime` that combines this date with the specified values for time fields. The second and nanosecond fields are set to zero, if their values are not specified. In the last method, this date is combined with the time at midnight.

Click here to view code image

```
// LocalDateTime
ZonedDateTime atZone(ZoneId zone)
```

Returns a `ZonedDateTime` by combining this date-time with the specified time zone (p. 1072).

Click here to view code image

```
// LocalTime, LocalDate, LocalDateTime, respectively.
static LocalTime now()
static LocalDate now()
static LocalDateTime now()
```

Each temporal class has this static factory method, which returns either the current time, date, or date-time from the system clock.

Example 17.1 *Creating Local Dates and Local Times*

Click here to view code image

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;

public class CreatingTemporals {

  public static void main(String[] args) {

    // Creating a specific time from time-based values:
    LocalTime time1 = LocalTime.of(8, 15, 35, 900);// 08:15:35.000000900
    LocalTime time2 = LocalTime.of(16, 45);        // 16:45
//  LocalTime time3 = LocalTime.of(25, 13, 30);    // DateTimeException
    System.out.println("Surveillance start time: " + time1);
    System.out.println("Closing time: " + time2);

    // Creating a specific date from date-based values:
    LocalDate date1 = LocalDate.of(1969, 7, 20);            // 1969-07-20
    LocalDate date2 = LocalDate.of(-3113, Month.AUGUST, 11);// -3113-08-11
//  LocalDate date3 = LocalDate.of(2021, 13, 11);           // DateTimeException
//  LocalDate date4 = LocalDate.of(2021, 2, 29);            // DateTimeException
    System.out.println("Date of lunar landing:        " + date1);
    System.out.println("Start Date of Mayan Calendar: " + date2);

    // Creating a specific date-time from date- and time-based values.
    // 2021-04-28T12:15
    LocalDateTime dt1 = LocalDateTime.of(2021, 4, 28, 12, 15);
    // 2021-08-17T14:00
    LocalDateTime dt2 = LocalDateTime.of(2021, Month.AUGUST, 17, 14, 0);
```

```
    System.out.println("Car service appointment: " + dt1);
    System.out.println("Hospital appointment:    " + dt2);

    // Combining date and time objects.
    // 1969-07-20T12:00
    LocalDateTime dt3 = LocalDateTime.of(date1, LocalTime.NOON);
    LocalDateTime dt4 = LocalTime.of(12, 0).atDate(date1);
    LocalDateTime dt5 = date1.atTime(LocalTime.NOON);
    LocalDateTime dt6 = date1.atTime(12, 0);
    System.out.println("Factory date-time combo: " + dt3);
    System.out.println("Time with date combo:    " + dt4);
    System.out.println("Date with time combo:    " + dt5);
    System.out.println("Date with explicit time combo: " + dt6);

    // Current time:
    LocalTime currentTime = LocalTime.now();
    System.out.println("Current time:      " + currentTime);

    // Current date:
    LocalDate currentDate = LocalDate.now();
    System.out.println("Current date:      " + currentDate);

    // Current date and time:
    LocalDateTime currentDateTime = LocalDateTime.now();
    System.out.println("Current date-time: " + currentDateTime);
  }
}
```

Possible output from the program:

Click here to view code image

```
Surveillance start time: 08:15:35.000000900
Closing time: 16:45
Date of lunar landing:        1969-07-20
Start Date of Mayan Calendar: -3113-08-11
Car service appointment: 2021-04-28T12:15
Hospital appointment:    2021-08-17T14:00
Factory date-time combo: 1969-07-20T12:00
Time with date combo:    1969-07-20T12:00
Date with time combo:    1969-07-20T12:00
Date with explicit time combo: 1969-07-20T12:00
Current time:      10:55:41.296744
Current date:      2021-03-05
Current date-time: 2021-03-05T10:55:41.299318
```

Accessing Fields in Dates and Times

A temporal object provides get methods that are tailored to access the values of specific temporal fields that constitute its state. The `LocalTime` and `LocalDate` classes provide get methods to access the values of time and date fields, respectively. Not surprisingly, the `LocalDateTime` class provides get methods for accessing the values of both time and date fields.

Click here to view code image

```
// LocalTime, LocalDateTime
int getHour()
int getMinute()
int getSecond()
int getNano()
```

Return the value of the appropriate time field from the current `LocalTime` or `LocalDateTime` object.

```
// LocalDate, LocalDateTime
int       getDayOfMonth()
DayOfWeek getDayOfWeek()
int       getDayOfYear()
Month     getMonth()
int       getMonthValue()
int       getYear()
```

Return the value of the appropriate date field from the current `LocalDate` or `LocalDateTime` object. The enum type `DayOfWeek` allows days of the week to be referred to by name; for example, `DayOfWeek.MONDAY` is day 1 of the week. The enum type `Month` allows months of the year to be referred to by name—for example, `Month.JANUARY`. The month value is from 1 (`Month.JANUARY`) to 12 (`Month.DECEMBER`).

Click here to view code image

```
// LocalTime, LocalDate, LocalDateTime
int get(TemporalField field)
long getLong(TemporalField field)
boolean isSupported(TemporalField field)
```

The first two methods return the value of the specified `TemporalField` (p. 1046) from this temporal object as an `int` value or as a `long` value, respectively. To specify fields whose value does not fit into an `int`, the `getLong()` method must be used.

The third method checks if the specified field is supported by this temporal object. It avoids an exception being thrown if it has been determined that the field is supported.

Using an invalid field in a get method will result in any one of these exceptions: `DateTimeException` (field value cannot be obtained), `UnsupportedTemporalType-Exception` (field is not supported), or `ArithmeticException` (numeric overflow occurred).

Here are some examples of using the get methods; more examples can be found in **Example 17.2**. Given that `time` and `date` refer to a `LocalTime` (`08:15`) and a `LocalDate` (`1945-08-06`), respectively, the code below shows how we access the values of the temporal fields using specifically named get methods and using specific temporal fields.

Click here to view code image

```
int minuteOfHour1 = time.getMinute();                        // 15
int minuteOfHour2 = time.get(ChronoField.MINUTE_OF_HOUR); // 15

int monthVal1 = date.getMonthValue();                        // 8
int monthVal2 = date.get(ChronoField.MONTH_OF_YEAR);         // 8
```

The temporal class `LocalDateTime` also provides two methods to obtain the date and the time as temporal objects, in contrast to accessing the values of individual date and time fields.

Click here to view code image

```
LocalDateTime doomsday = LocalDateTime.of(1945, 8, 6, 8, 15);
LocalDate date = doomsday.toLocalDate();                     // 1945-08-06
LocalTime time = doomsday.toLocalTime();                     // 08:15
```

```
// LocalDateTime
LocalDate toLocalDate()
LocalTime toLocalTime()
```

These methods can be used to get the `LocalDate` and the `LocalTime` components of this date-time object, respectively.

The following two methods return the number of days in the month and in the year represented by a `LocalDate` object.

Click here to view code image

```
LocalDate foolsday = LocalDate.of(2022, 4, 1);
int daysInMonth = foolsday.lengthOfMonth();    // 30
int daysInYear = foolsday.lengthOfYear();     // 365 (2022 is not a leap year.)
```

```
// LocalDate
int lengthOfMonth()
int lengthOfYear()
```

These two methods return the number of days in the month and in the year represented by this date, respectively.

Comparing Dates and Times

It is also possible to check whether a temporal object represents a point in time before or after another temporal object of the same type. In addition, the `LocalDate` and `LocalDateTime` classes provide an `isEqual()` method that determines whether a temporal object is equal to another temporal object of the *same* type. In contrast, the `equals()` method allows equality comparison with an *arbitrary* object.

Click here to view code image

```
LocalDate d1 = LocalDate.of(1948, 2, 28);                    // 1948-02-28
LocalDate d2 = LocalDate.of(1949, 3, 1);                     // 1949-03-01
boolean result1 = d1.isBefore(d2);                           // true
boolean result2 = d2.isAfter(d1);                            // true
boolean result3 = d1.isAfter(d1);                            // false
boolean result4 = d1.isEqual(d2);                            // false
boolean result5 = d1.isEqual(d1);                            // true
boolean result6 = d1.isLeapYear();                           // true
```

The temporal classes implement the `Comparable<E>` interface, providing the `compareTo()` method so that temporal objects can be compared in a meaningful way. The temporal classes also override the `equals()` and the `hashCode()` methods of the `Object` class. These methods make it possible to both search for and sort temporal objects.

Click here to view code image

```
// LocalTime
boolean isBefore(LocalTime other)
boolean isAfter(LocalTime other)
```

Determine whether this `LocalTime` represents a point on the timeline before or after the `other` time, respectively.

Click here to view code image

```
// LocalDate
boolean isBefore(ChronoLocalDate other)
boolean isAfter(ChronoLocalDate other)
boolean isEqual(ChronoLocalDate other)
boolean isLeapYear()
```

The first two methods determine whether this `LocalDate` represents a point on the timeline before or after the `other` date, respectively. The `LocalDate` class implements the `ChronoLocalDate` interface.

The third method determines whether this date is equal to the specified date.

The last method checks for a leap year according to the ISO proleptic calendar system rules.

Click here to view code image

```
// LocalDateTime
boolean isBefore(ChronoLocalDateTime<?> other)
boolean isAfter(ChronoLocalDateTime<?> other)
boolean isEqual(ChronoLocalDateTime<?> other)
```

The first two methods determine whether this `LocalDateTime` represents a point on the timeline before or after the specified date-time, respectively. The `Local-DateTime` class implements the `ChronoLocalDateTime<LocalDateTime>` interface.

The third method determines whether this date-time object represents the same point on the timeline as the other date-time.

Click here to view code image

```
int compareTo(LocalTime other)                  // LocalTime
int compareTo(ChronoLocalDate other)            // LocalDate
int compareTo(ChronoLocalDateTime<?> other)    // LocalDateTime
```

Compare this temporal object to another temporal object. The three temporal classes implement the `Comparable<E>` functional interface. The `compareTo()` method returns `0` if the two temporal objects are equal, a negative value if this temporal object is less than the other temporal object, and a positive value if this temporal object is greater than the other temporal object.

Creating Modified Copies of Dates and Times

An immutable object does not provide any set methods that can change its state. Instead, it usually provides what are known as `with` methods (or *withers*) that return a copy of the original object where exactly one field has been set to a new value. The `LocalTime` and

`LocalDate` classes provide `with` methods to set the value of a time or date field, respectively. Not surprisingly, the `LocalDateTime` class provides `with` methods to set the values of both time and date fields individually. A `with` method changes a specific property in an absolute way, which is reflected in the state of the new temporal object; the original object, however, is not affected. Such `with` methods are also called *absolute adjusters*, in contrast to the *relative adjusters* that we will meet later (p. 1040).

Click here to view code image

```
// LocalTime, LocalDateTime
LocalTime/LocalDateTime withHour(int hour)
LocalTime/LocalDateTime withMinute(int minute)
LocalTime/LocalDateTime withSecond(int second)
LocalTime/LocalDateTime withNano(int nanoOfSecond)
```

Return a copy of this `LocalTime` or `LocalDateTime` with the value of the appropriate time field changed to the specified value. A `DateTimeException` is thrown if the argument value is out of range.

Click here to view code image

```
// LocalDate, LocalDateTime
LocalDate/LocalDateTime withYear(int year)
LocalDate/LocalDateTime withMonth(int month)
LocalDate/LocalDateTime withDayOfMonth(int dayOfMonth)
LocalDate/LocalDateTime withDayOfYear(int dayOfYear)
```

Return a copy of this `LocalDate` or `LocalDateTime` with the value of the appropriate date field changed to the specified value. A `DateTimeException` is thrown if the specified value is out of range or is invalid in combination with the values of the other time or date fields in the temporal object.

The first and second methods will adjust the day of the month to the *last valid day* of the month, if the day of the month becomes invalid when the year or the month is changed (e.g., the month value 2 will change the date 2020-03-31 to 2020-02-29).

In contrast, the third method will throw a `DateTimeException` if the specified day of the month is invalid for the month-year combination (e.g., the day of month 29 is invalid for February 2021), as will the last method if the day of the year is invalid for the year (e.g., the day of year 366 is invalid for the year 2021).

Click here to view code image

```
// LocalTime, LocalDate, LocalDateTime
LocalTime/LocalDate/LocalDateTime with(TemporalField field, long newValue)
```

Returns a copy of this temporal object with the specified `TemporalField` (p. 1046) set to a new value. The `ChronoField` enum type implements the `TemporalField` interface, and its enum constants define specific temporal fields (p. 1046).

Using an invalid field in the `with()` method will result in any one of these exceptions: `DateTimeException` (field value cannot be set), `Unsupported-TemporalTypeException` (field is not supported), or `ArithmeticException` (numeric overflow occurred).

The code lines below are from **Example 17.2**. In the second assignment statement, the method calls are chained. Three instances of the `LocalDate` class are created consecutively, as each `with` method is called to set the value of a specific date field. The last assignment shows the use of temporal fields in the `with()` method for the same purpose.

Click here to view code image

```
LocalDate date2 = LocalDate.of(2021, 3, 1);                    // 2021-03-01
date2 = date2.withYear(2024).withMonth(2).withDayOfMonth(28); // 2024-02-28

LocalDate date3 = LocalDate.of(2021, 3, 1);                    // 2021-03-01
date3 = date3
    .with(ChronoField.YEAR, 2024L)
    .with(ChronoField.MONTH_OF_YEAR, 2L)
    .with(ChronoField.DAY_OF_MONTH, 28L);                      // 2024-02-28
```

The following code contains a logical error, such that the last two `LocalDate` instances returned by the `with` methods are ignored, and the reference `date2` never gets updated.

Click here to view code image

```
date2 = date2.withYear(2022);                     // 2022-03-01
date2.withMonth(2).withDayOfMonth(28);            // date2 is still 2022-03-01.
```

In the next code examples, each call to a `with` method throws a `DateTimeException`. The minute and hour values are out of range for a `LocalTime` object. Certainly the month value 13 is out of range for a `LocalDate` object. The day of month value 31 is not valid for April, which has 30 days. The day of year value 366 is out of range as well, since the year 2021 is not a leap year.

Click here to view code image

```
LocalTime time = LocalTime.of(14, 45);       // 14:45
time = time.withMinute(100);        // Out of range. DateTimeException.
time = time.withHour(25);           // Out of range. DateTimeException.

LocalDate date = LocalDate.of(2021, 4, 30);  // 2021-04-30
date = date.withMonth(13);          // Out of range. DateTimeException.
```

```
date = date.withDayOfMonth(31);    // Out of range for month. DateTimeException.
date = date.withDayOfYear(366);    // Out of range for year. DateTimeException.
```

The code snippets below illustrate how the `withYear()` and `withMonth()` methods adjust the day of the month, if necessary, when the year or the month is changed, respectively.

Click here to view code image

```
LocalDate date3 = LocalDate.of(2020, 2, 29);  // Original: 2020-02-29
date3 = date3.withYear(2021);                 // Expected: 2021-02-29
System.out.println("Date3: " + date3);        // Adjusted: 2021-02-28

LocalDate date4 = LocalDate.of(2021, 3, 31);  // Original: 2021-03-31
date4 = date4.withMonth(4);                   // Expected: 2021-04-31
System.out.println("Date4: " + date4);        // Adjusted: 2021-04-30
```

The year in the date 2020-02-29 is changed to 2021, resulting in the following date: 2021-02-29. Since the year 2021 is not a leap year, the month of February cannot have 29 days. The `withYear()` method adjusts the day of the month to the last valid day of the month (i.e., 28). Similarly, the month in the date 2021-03-31 is changed to 4 (i.e., April), resulting in the following date: 2021-04-31. Since the month April has 30 days, the `withMonth()` method adjusts the day of the month to the last valid day of the month (i.e., 30).

Example 17.2 *Using Local Dates and Local Times*

Click here to view code image

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
import java.time.temporal.ChronoField;

public class UsingTemporals {

  public static void main(String[] args) {
    // Date-Time: 1945-08-06T08:15
    LocalDateTime doomsday = LocalDateTime.of(1945, 8, 6, 8, 15);
    LocalDate date = doomsday.toLocalDate();                    // 1945-08-06
    LocalTime time = doomsday.toLocalTime();                    // 08:15
    System.out.println("Date-Time: " + doomsday);
    System.out.println();

    // Time: 08:15
    int hourOfDay      = time.getHour();                        // 8
    int minuteOfHour1  = time.getMinute();                      // 15
    int minuteOfHour2  = time.get(ChronoField.MINUTE_OF_HOUR); // 15
    int secondOfMinute = time.getSecond();                      // 0
```

```
    System.out.println("Time of day:      " + time);
    System.out.println("Hour-of-day:      " + hourOfDay);
    System.out.println("Minute-of-hour 1: " + minuteOfHour1);
    System.out.println("Minute-of-hour 2: " + minuteOfHour2);
    System.out.println("Second-of-minute: " + secondOfMinute);
    System.out.println();

    // Date: 1945-08-06
    int year       = date.getYear();                              // 1945
    int monthVal1  = date.getMonthValue();                        // 8
    int monthVal2  = date.get(ChronoField.MONTH_OF_YEAR);         // 8
    Month month    = date.getMonth();                             // AUGUST
    DayOfWeek dow  = date.getDayOfWeek();                         // MONDAY
    int day        = date.getDayOfMonth();                        // 6
    System.out.println("Date:  " + date);
    System.out.println("Year:  " + year);
    System.out.println("Month value 1: " + monthVal1);
    System.out.println("Month value 2: " + monthVal2);
    System.out.println("Month-of-year: " + month);
    System.out.println("Day-of-week:   " + dow);
    System.out.println("Day-of-month:  " + day);
    System.out.println();

    // Ordering
    LocalDate d1 = LocalDate.of(1948, 2, 28);                     // 1948-02-28
    LocalDate d2 = LocalDate.of(1949, 3, 1);                      // 1949-03-01
    boolean result1 = d1.isBefore(d2);                            // true
    boolean result2 = d2.isAfter(d1);                             // true
    boolean result3 = d1.isAfter(d1);                             // false
    boolean result4 = d1.isEqual(d2);                             // false
    boolean result5 = d1.isEqual(d1);                             // true
    boolean result6 = d1.isLeapYear();                            // true

    System.out.println("Ordering:");
    System.out.println(d1 + " is before "   + d2 + ": " + result1);
    System.out.println(d2 + " is after "    + d1 + ": " + result2);
    System.out.println(d1 + " is after "    + d1 + ": " + result3);
    System.out.println(d1 + " is equal to " + d2 + ": " + result4);
    System.out.println(d1 + " is equal to " + d1 + ": " + result5);
    System.out.println(d1.getYear() + " is a leap year: " + result6);
    System.out.println();

    System.out.println("Using absolute adjusters:");
    LocalDate date2 = LocalDate.of(2021, 3, 1);
    System.out.println("Date before adjusting: " + date2);     // 2021-03-01
    date2 = date2.withYear(2024).withMonth(2).withDayOfMonth(28);
    System.out.println("Date after adjusting:  " + date2);     // 2024-02-28
    System.out.println();

    System.out.println("Using temporal fields:");
    LocalDate date3 = LocalDate.of(2021, 3, 1);
    System.out.println("Date before adjusting: " + date3);     // 2021-03-01
    date3 = date3
```

```
        .with(ChronoField.YEAR, 2024L)
        .with(ChronoField.MONTH_OF_YEAR, 2L)
        .with(ChronoField.DAY_OF_MONTH, 28L);
    System.out.println("Date after adjusting:  " + date3);     // 2024-02-28
  }
}
```

Output from the program:

Click here to view code image

```
Date-Time: 1945-08-06T08:15

Time of day:      08:15
Hour-of-day:      8
Minute-of-hour 1: 15
Minute-of-hour 2: 15
Second-of-minute: 0

Date:  1945-08-06
Year:  1945
Month value 1: 8
Month value 2: 8
Month-of-year: AUGUST
Day-of-week:   MONDAY
Day-of-month:  6

Ordering:
-1004-03-01 is before 1004-03-01: true
1004-03-01 is after -1004-03-01: true
-1004-03-01 is after -1004-03-01: false
-1004-03-01 is equal to 1004-03-01: false
-1004-03-01 is equal to -1004-03-01: true
1004 is a leap year: true

Using absolute adjusters:
Date before adjusting: 2021-03-01
Date after adjusting:  2024-02-28

Using temporal fields:
Date before adjusting: 2021-03-01
Date after adjusting:  2024-02-28
```

Temporal Arithmetic with Dates and Times

The temporal classes provide *plus* and *minus* methods that return a copy of the original object that has been *incremented or decremented by a specific amount of time*— for example, by number of hours or by number of months.

The `LocalTime` and `LocalDate` classes provide `plus/minus` methods to increment/ decrement a time or a date by a specific amount in terms of a *time unit* (e.g., hours, minutes, and seconds) or a *date unit* (e.g., years, months, and days), respectively. The `LocalDateTime` class provides `plus/minus` methods to increment/decrement a date-time object by an amount that is specified in terms of either a time unit or a date unit. For example, the `plusMonths(m)` and `plus(m, ChronoUnit.MONTHS)` method calls to a `LocalDate` object will return a new `LocalDate` object after adding the specified number of months passed as an argument to the method. Similarly, the `minus-Minutes(mm)` and `minus(mm, ChronoUnit.MINUTES`) method calls on a `LocalTime` class will return a new `LocalTime` object after subtracting the specified number of minutes passed as an argument to the method. The change is relative, and is reflected in the new temporal object that is returned. Such `plus/minus` methods are also called *relative adjusters*, in contrast to *absolute adjusters* (p. 1035). The `ChronoUnit` enum type implements the `TemporalUnit` interface (p. 1044).

Click here to view code image

```
// LocalTime, LocalDateTime
LocalTime/LocalDateTime minusHours(long hours)
LocalTime/LocalDateTime plusHours(long hours)

LocalTime/LocalDateTime minusMinutes(long minutes)
LocalTime/LocalDateTime plusMinutes(long minutes)

LocalTime/LocalDateTime minusSeconds(long seconds)
LocalTime/LocalDateTime plusSeconds(long seconds)

LocalTime/LocalDateTime minusNanos(long nanos)
LocalTime/LocalDateTime plusNanos(long nanos)
```

Return a copy of this `LocalTime` or `LocalDateTime` object with the specified amount either subtracted or added to the value of a specific time field. The calculation always wraps around midnight.

For the methods of the `LocalDateTime` class, a `DateTimeException` is thrown if the result exceeds the date range.

Click here to view code image

```
// LocalDate, LocalDateTime
LocalDate/LocalDateTime minusYears(long years)
LocalDate/LocalDateTime plusYears(long years)

LocalDate/LocalDateTime minusMonths(long months)
LocalDate/LocalDateTime plusMonths(long months)

LocalDate/LocalDateTime minusWeeks(long weeks)
LocalDate/LocalDateTime plusWeeks(long weeks)
```

```
LocalDate/LocalDateTime minusDays(long days)
LocalDate/LocalDateTime plusDays(long days)
```

Return a copy of this `LocalDate` or `LocalDateTime` with the specified amount either subtracted or added to the value of a specific date field.

All methods throw a `DateTimeException` if the result exceeds the date range.

The first four methods will change the day of the month to the *last valid day* of the month if necessary, when the day of the month becomes invalid as a result of the operation.

The last four methods will adjust the month and year fields as necessary to ensure a valid result.

Click here to view code image

```
// LocalTime, LocalDate, LocalDateTime
LocalTime/LocalDate/LocalDateTime minus(long amountToSub,
                                        TemporalUnit unit)
LocalTime/LocalDate/LocalDateTime plus(long amountToAdd, TemporalUnit unit)
boolean isSupported(TemporalUnit unit)
```

The `minus()` and `plus()` methods return a copy of this temporal object with the specified amount subtracted or added, respectively, according to the `TemporalUnit` specified. The `ChronoUnit` enum type implements the `TemporalUnit` interface, and its enum constants define specific temporal units (**p. 1044**).

The `isSupported()` method checks if the specified `TemporalUnit` is supported by this temporal object. It avoids an exception being thrown if it has been determined that the unit is supported.

The `minus()` or the `plus()` method can result in any one of these exceptions: `DateTimeException` (the amount cannot be subtracted or added), `Unsupported-TemporalTypeException` (unit is not supported), or `ArithmeticException` (numeric overflow occurred).

Click here to view code image

```
// LocalTime, LocalDate, LocalDateTime
LocalTime/LocalDate/LocalDateTime minus(TemporalAmount amountToSub)
LocalTime/LocalDate/LocalDateTime plus(TemporalAmount amountToAdd)
```

Return a copy of this temporal object with the specified temporal amount subtracted or added, respectively. The classes `Period` (**p. 1057**) and `Duration` (**p. 1064**) implement the `TemporalAmount` interface.

The `minus()` or the `plus()` method can result in any one of these exceptions: `DateTimeException` (the temporal amount cannot be subtracted or added) or `ArithmeticException` (numeric overflow occurred).

Click here to view code image

```
// LocalTime, LocalDate, LocalDateTime
long until(Temporal endExclusive, TemporalUnit unit)
```

Calculates the amount of time between two temporal objects in terms of the specified `TemporalUnit` (p. 1044). The start and the end points are this temporal object and the specified temporal argument `endExclusive`, where the end point is excluded. The result will be negative if the other temporal is before this temporal.

The `until()` method can result in any one of these exceptions: `DateTime-Exception` (the temporal amount cannot be calculated or the end temporal cannot be converted to the appropriate temporal object), `UnsupportedTemporalType-Exception` (unit is not supported), or `ArithmeticException` (numeric overflow occurred).

Click here to view code image

```
// LocalDate
Period until(ChronoLocalDate endDateExclusive)
```

Calculates the amount of time between this date and another date as a `Period` (p. 1057). The calculation excludes the end date. The `LocalDate` class implements the `ChronoLocalDate` interface.

Example 17.3 demonstrates what we can call *temporal arithmetic*, where a `LocalDate` object is modified by adding or subtracting an amount specified as days, weeks, or months. Note how the value of the date fields is adjusted after each operation. In Example 17.3, the date `2021-10-23` is created at (1), and 10 months, 3 weeks, and 40 days are successively added to the new date object returned by each `plus` method call at (2), (3), and (4), respectively, resulting in the date `2022-10-23`. We then subtract 2 days, 4 weeks, and 11 months successively from the new date object returned by each `minus()` method call at (5), (6), and (7), respectively, resulting in the date `2021-10-23`. The method calls at (5), (6), and (7) are passed the temporal unit explicitly. In Example 17.3, several assignment statements are used to print the intermediate dates, but the code can be made more succinct by method chaining.

Click here to view code image

```
LocalDate date = LocalDate.of(2021, 10, 23);            // 2021-10-23
date = date.plusMonths(10).plusWeeks(3).plusDays(40);   // Method chaining
System.out.println(date);                               // 2022-10-23
date = date.minus(2, ChronoUnit.DAYS)
```

```
            .minus(4, ChronoUnit.WEEKS)
            .minus(11, ChronoUnit.MONTHS);                // Method chaining
System.out.println(date);                                 // 2021-10-23
```

The following code snippet illustrates the wrapping of time around midnight, as one would expect on a 24-hour clock. Each method call returns a new `LocalTime` object.

Click here to view code image

```
LocalTime witchingHour = LocalTime.MIDNIGHT               // 00:00
    .plusHours(14)                                        // 14:00
    .plusMinutes(45)                                      // 14:45
    .plusMinutes(30)                                      // 15:15
    .minusHours(15)                                       // 00:15
    .minusMinutes(15);                                    // 00:00
```

The next code snippet illustrates how the `plusYears()` method adjusts the day of the month, if necessary, when the year value is changed. The year in the date 2020-02-29 is changed to 2021 by adding 1 year, resulting in the following date: 2021-02-29. The `plusYears()` method adjusts the day of the month to the last valid day of the month, 28; as the year 2021 is not a leap year, the month of February cannot have 29 days.

Click here to view code image

```
LocalDate date5 = LocalDate.of(2020, 2, 29);  // Original: 2020-02-29
date5 = date5.plusYears(1);                   // Expected: 2021-02-29
System.out.println("Date5: " + date5);        // Adjusted: 2021-02-28
```

A temporal can also be adjusted by a *temporal amount*—for example, by a `Period` (p. 1057) or a `Duration` (p. 1064). The methods `plus()` and `minus()` accept the temporal amount as an argument, as shown by the code below.

Click here to view code image

```
LocalTime busDep = LocalTime.of(12, 15);                    // 12:15
Duration d1 = Duration.ofMinutes(30);                       // PT30M
LocalTime nextBusDep = busDep.plus(d1);                     // 12:45

LocalDate birthday = LocalDate.of(2020, 10, 23);            // 2020-10-23
Period p1 = Period.ofYears(1);                              // P1Y
LocalDate nextBirthday = birthday.plus(p1);                 // 2021-10-23
```

The `until()` method can be used to calculate the amount of time between two compatible temporal objects. The code below calculates the number of days to New Year's Day from the current date; the result, of course, will depend on the current date. In the call to the `until()` method at (1), the temporal unit specified is `ChronoUnit.DAYS`, as we want the difference between the dates to be calculated in days.

Click here to view code image

```
LocalDate currentDate = LocalDate.now();
LocalDate newYearDay = currentDate.plusYears(1).withMonth(1).withDayOfMonth(1);
long daysToNewYear = currentDate.until(newYearDay, ChronoUnit.DAYS); // (1)
System.out.println("Current Date: " + currentDate); // Current Date: 2021-03-08
System.out.println("New Year's Day: " + newYearDay);// New Year's Day: 2022-01-01
System.out.println("Days to New Year: " + daysToNewYear);// Days to New Year: 299
```

The statement at (1) below is meant to calculate the number of minutes until midnight from now, but throws a `DateTimeException` because it is not possible to obtain a `LocalDateTime` object from the end point, which is a `LocalTime` object.

Click here to view code image

```
long minsToMidnight = LocalDateTime.now()             // (1) DateTimeException!
         .until(LocalTime.MIDNIGHT.minusSeconds(1), ChronoUnit.MINUTES);
```

However, the statement at (2) executes normally, as both the start and end points are `LocalTime` objects.

Click here to view code image

```
long minsToMidnight = LocalTime.now()                 // (2)
         .until(LocalTime.MIDNIGHT.minusSeconds(1), ChronoUnit.MINUTES);
```

Example 17.3 *Temporal Arithmetic*

Click here to view code image

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class TemporalArithmetic {

  public static void main(String[] args) {

    LocalDate date = LocalDate.of(2021, 10, 23);           // (1)
    System.out.println("Date:              " + date);       // 2021-10-23
    date = date.plusMonths(10);                            // (2)
    System.out.println("10 months after:  " + date);       // 2022-08-23
    date = date.plusWeeks(3);                              // (3)
    System.out.println("3 weeks after:    " + date);       // 2022-09-13
    date = date.plusDays(40);                              // (4)
    System.out.println("40 days after:    " + date);       // 2022-10-23

    date = date.minus(2, ChronoUnit.DAYS);                 // (5)
    System.out.println("2 days before:    " + date);       // 2022-10-21
```

```
    date = date.minus(4, ChronoUnit.WEEKS);                // (6)
    System.out.println("4 weeks before:   " + date);       // 2022-09-23
    date = date.minus(11, ChronoUnit.MONTHS);              // (7)
    System.out.println("11 months before: " + date);       // 2021-10-23
  }
}
```

Output from the program:

```
Date:             2021-10-23
10 months after:  2022-08-23
3 weeks after:    2022-09-13
40 days after:    2022-10-23
2 days before:    2022-10-21
4 weeks before:   2022-09-23
11 months before: 2021-10-23
```

17.3 Using Temporal Units and Temporal Fields

Temporal units and temporal fields allow temporal objects to be accessed and manipulated in a human-readable way.

For temporal units and temporal fields supported by the `Period` and `Duration` classes, see §17.5, p. 1057, and §17.6, p. 1064, respectively.

Temporal Units

The `java.time.temporal.TemporalUnit` interface represents a *unit* of measurement, rather than an amount of such a unit—for example, the unit *years* to qualify that an amount of time should be interpreted as number of years. The `java.time.temporal.ChronoUnit` enum type implements this interface, defining the temporal units by constant names to provide convenient unit-based access to manipulate a temporal object. Constants defined by the `ChronoUnit` enum type include the following temporal units, among others: `SECONDS`, `MINUTES`, `HOURS`, `DAYS`, `MONTHS`, and `YEARS`.

The output from Example 17.4 shows a table with all the temporal units defined by the `ChronoUnit` enum type. It is not surprising that not all temporal units are valid for all types of temporal objects. The time units, such as `SECONDS`, `MINUTES`, and `HOURS`, are valid for temporal objects that are time-based, such as `LocalTime`, `LocalDateTime`, and `Instant`. Likewise, the date units, such as `DAYS`, `MONTHS`, and `YEARS`, are valid units for temporal objects that are date-based, such as `LocalDate`, `LocalDateTime`, and `ZonedDateTime`.

A `ChronoUnit` enum constant can be queried by the following selected methods:

```
Duration getDuration()
```

Gets the estimated duration of this unit in the ISO calendar system. For example, `ChronoUnit.DAYS.getDuration()` has the duration `PT24H` (i.e., 24 hours).

```
boolean isDateBased()
boolean isTimeBased()
```

Check whether this unit is a date unit or a time unit, respectively. For example, `ChronoUnit.HOURS.isDateBased()` is `false`, but `ChronoUnit.SECONDS.isTimeBased()` is `true`.

Click here to view code image

```
boolean isSupportedBy(Temporal temporal)
```

Checks whether this unit is supported by the specified temporal object. For example, `ChronoUnit.YEARS.isSupportedBy(LocalTime.MIDNIGHT)` is `false`.

Click here to view code image

```
static ChronoUnit[] values()
```

Returns an array containing the unit constants of this enum type, in the order they are declared. This method is called at (2) in **Example 17.4**.

The temporal classes provide the method `isSupported(unit)` to determine whether a temporal `unit` is valid for a temporal object. In **Example 17.4**, this method is used at (3), (4), and (5) to determine whether each temporal unit defined by the `ChronoUnit` enum type is a valid unit for the different temporal classes.

The following methods of the temporal classes all accept a temporal unit that qualifies how a numeric quantity should be interpreted:

- `minus(amount, unit)` and `plus(amount, unit)`, (p. 1040)
  Click here to view code image

  ```
  LocalDate date = LocalDate.of(2021, 10, 23);
  System.out.print("Date " + date);
  date = date.minus(10, ChronoUnit.MONTHS).minus(3, ChronoUnit.DAYS);
  System.out.println(" minus 10 months and 3 days: " + date);
  // Date 2021-10-23 minus 10 months and 3 days: 2020-12-20

  LocalTime time = LocalTime.of(14, 15);
  System.out.print("Time " + time);
  time = time.plus(9, ChronoUnit.HOURS).plus(70, ChronoUnit.MINUTES);
  ```

```
System.out.println(" plus 9 hours and 70 minutes is " + time);
// Time 14:15 plus 9 hours and 70 minutes is 00:25
```

- `until(temporalObj, unit)`, (p. 1040)

  Click here to view code image

  ```
  LocalDate fromDate = LocalDate.of(2021, 3, 1);
  LocalDate xmasDate = LocalDate.of(2021, 12, 25);
  long tilChristmas = fromDate.until(xmasDate, ChronoUnit.DAYS);
  System.out.println("From " + fromDate + ", days until Xmas: " + tilChristmas);
  // From 2021-03-01, days until Xmas: 299
  ```

Temporal Fields

The `java.time.temporal.TemporalField` interface represents a specific field of a temporal object. The `java.time.temporal.ChronoField` enum type implements this interface, defining the fields by constant names so that a specific field can be conveniently accessed. Selected constants from the `ChronoField` enum type include `SECOND_OF_MINUTE`, `MINUTE_OF_DAY`, `DAY_OF_MONTH`, `MONTH_OF_YEAR`, and `YEAR`.

The output from **Example 17.4** shows a table with all the temporal fields defined by the `ChronoField` enum type.

Analogous to a `ChronoUnit` enum constant, a `ChronoField` enum constant can be queried by the following selected methods:

```
TemporalUnit getBaseUnit()
```

Gets the unit that the field is measured in. For example, `ChronoField.DAY_OF_MONTH.getBaseUnit()` returns `ChronoUnit.DAYS`.

```
boolean isDateBased()
boolean isTimeBased()
```

Check whether this field represents a date or a time field, respectively. For example, `ChronoField.HOUR_OF_DAY.isDateBased()` is `false`, but `ChronoField.SECOND_OF_MINUTE.isTimeBased()` is `true`.

Click here to view code image

```
boolean isSupportedBy(TemporalAccessor temporal)
```

Checks whether this field is supported by the specified temporal object. For example, `ChronoField.YEAR.isSupportedBy(LocalTime.MIDNIGHT)` is `false`.

```
static ChronoField[] values()
```

Returns an array containing the field constants of this enum type, in the order they are declared. This method is called at (7) in **Example 17.4**.

The temporal classes provide the method `isSupported(field)` to determine whether a temporal `field` is valid for a temporal object. In **Example 17.4**, this method is used at (8), (9), and (10) to determine whether each temporal field defined by the `ChronoField` enum type is a valid field for the different temporal classes.

The following methods of the temporal classes all accept a temporal field that designates a specific field of the temporal object:

- `get(field)`, (**p. 1032**)
  Click here to view code image

  ```
  LocalDate date = LocalDate.of(2021, 8, 13);
  int monthValue = date.get(ChronoField.MONTH_OF_YEAR);
  System.out.print("Date " + date + " has month of the year: " + monthValue);
  // Date 2021-08-13 has month of the year: 8
  ```

- `with(field, amount)`, (**p. 1035**)
  Click here to view code image

  ```
  LocalDateTime dateTime = LocalDateTime.of(2021, 8, 13, 20, 20);
  System.out.print("Date-time " + dateTime);
  dateTime = dateTime.with(ChronoField.DAY_OF_MONTH, 11)
                     .with(ChronoField.MONTH_OF_YEAR, 1)
                     .with(ChronoField.YEAR, 2022);
  System.out.println(" changed to: " + dateTime);
  // Date-time 2021-08-13T20:20 changed to: 2022-01-11T20:20
  ```

In **Example 17.4**, the code at (1) and at (6) prints tables that show which `ChronoUnit` and `ChronoField` constants are valid in which temporal-based object. A `LocalTime` instance supports time-based units and fields, and a `LocalDate` instance supports date-based units and fields. A `LocalDateTime` or a `ZonedDateTime` supports both time-based and date-based units and fields. Using an invalid enum constant for a temporal object will invariably result in an `UnsupportedTemporalTypeException` being thrown.

Example 17.4 *Valid Temporal Units and Temporal Fields*

Click here to view code image

```
import java.time.Instant;
import java.time.LocalDate;
```

```
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZonedDateTime;
import java.time.temporal.ChronoField;
import java.time.temporal.ChronoUnit;

public class ValidTemporalUnitsAndFields {

  public static void main(String[] args) {

    // Temporals:
    LocalTime time = LocalTime.now();
    LocalDate date = LocalDate.now();
    LocalDateTime dateTime = LocalDateTime.now();
    ZonedDateTime zonedDateTime = ZonedDateTime.now();
    Instant instant = Instant.now();

    // Print supported units:                                          // (1)
    System.out.printf("%29s %s %s %s %s %s%n",
        "ChronoUnit", "LocalTime", "LocalDate", "LocalDateTime",
        " ZDT ", "Instant");
    ChronoUnit[] units =  ChronoUnit.values();                         // (2)
    for (ChronoUnit unit : units) {
      System.out.printf("%28S: %7b %9b %10b %9b %7b%n",
          unit.name(), time.isSupported(unit), date.isSupported(unit),   // (3)
          dateTime.isSupported(unit), zonedDateTime.isSupported(unit),   // (4)
          instant.isSupported(unit));                                    // (5)
      }
    System.out.println();

    // Print supported fields:                                         // (6)
    System.out.printf("%29s %s %s %s %s %s%n",
        "ChronoField", "LocalTime", "LocalDate", "LocalDateTime",
        " ZDT ", "Instant");
    ChronoField[] fields =  ChronoField.values();                      // (7)
    for (ChronoField field : fields) {
      System.out.printf("%28S: %7b %9b %10b %9b %7b%n",
          field.name(), time.isSupported(field), date.isSupported(field),// (8)
          dateTime.isSupported(field), zonedDateTime.isSupported(field), // (9)
          instant.isSupported(field));                                   // (10)
    }
    System.out.println();
  }
}
```

Output from the program (`ZDT` *stands for* `ZonedDateTime` *in the output*):

Click here to view code image

```
                   ChronoUnit LocalTime LocalDate LocalDateTime  ZDT  Instant
                       NANOS:    true     false         true     true    true
                      MICROS:    true     false         true     true    true
```

```
                    MILLIS:    true     false        true       true    true
                   SECONDS:    true     false        true       true    true
                   MINUTES:    true     false        true       true    true
                     HOURS:    true     false        true       true    true
                 HALF_DAYS:    true     false        true       true    true
                      DAYS:   false      true        true       true    true
                     WEEKS:   false      true        true       true   false
                    MONTHS:   false      true        true       true   false
                     YEARS:   false      true        true       true   false
                   DECADES:   false      true        true       true   false
                 CENTURIES:   false      true        true       true   false
                 MILLENNIA:   false      true        true       true   false
                      ERAS:   false      true        true       true   false
                   FOREVER:   false     false       false      false   false

               ChronoField LocalTime LocalDate LocalDateTime  ZDT  Instant
           NANO_OF_SECOND:    true     false        true       true    true
              NANO_OF_DAY:    true     false        true       true   false
          MICRO_OF_SECOND:    true     false        true       true    true
             MICRO_OF_DAY:    true     false        true       true   false
          MILLI_OF_SECOND:    true     false        true       true    true
             MILLI_OF_DAY:    true     false        true       true   false
         SECOND_OF_MINUTE:    true     false        true       true   false
            SECOND_OF_DAY:    true     false        true       true   false
           MINUTE_OF_HOUR:    true     false        true       true   false
            MINUTE_OF_DAY:    true     false        true       true   false
             HOUR_OF_AMPM:    true     false        true       true   false
       CLOCK_HOUR_OF_AMPM:    true     false        true       true   false
              HOUR_OF_DAY:    true     false        true       true   false
        CLOCK_HOUR_OF_DAY:    true     false        true       true   false
              AMPM_OF_DAY:    true     false        true       true   false
              DAY_OF_WEEK:   false      true        true       true   false
ALIGNED_DAY_OF_WEEK_IN_MONTH:   false      true        true       true   false
 ALIGNED_DAY_OF_WEEK_IN_YEAR:   false      true        true       true   false
             DAY_OF_MONTH:   false      true        true       true   false

              DAY_OF_YEAR:   false      true        true       true   false
                EPOCH_DAY:   false      true        true       true   false
     ALIGNED_WEEK_OF_MONTH:   false      true        true       true   false
      ALIGNED_WEEK_OF_YEAR:   false      true        true       true   false
            MONTH_OF_YEAR:   false      true        true       true   false
          PROLEPTIC_MONTH:   false      true        true       true   false
              YEAR_OF_ERA:   false      true        true       true   false
                     YEAR:   false      true        true       true   false
                      ERA:   false      true        true       true   false
          INSTANT_SECONDS:   false     false       false       true    true
           OFFSET_SECONDS:   false     false       false       true   false
```

17.4 Working with Instants

The temporal classes `LocalTime`, `LocalDate`, `LocalDateTime`, and `ZonedDateTime` are suitable for representing human time in terms of year, month, day, hour, minute, second, and time zone. The `Instant` class can be used for representing computer time, specially *time-stamps* that identify to a higher precision when an event occurred on the timeline. Instants are suitable for persistence purposes—for example, in a database.

An `Instant` represents *a point on the timeline*, measured with nanosecond precision from a starting point or origin which is defined to be at `1970-01-01T00:00:00Z`—that is, January 1, 1970, at midnight—and is called the *epoch*. Instants before the epoch have negative values, whereas instants after the epoch have positive values. The `Z` represents the time zone designator for the *zero UTC offset*, which is the time zone offset for all instants in the UTC standard (p. 1072). The text representation of the epoch shown above is in the ISO standard format used by the `toString()` method of the `Instant` class.

An `Instant` is modeled with two values:

- A `long` value to represent the *epoch-second*
- An `int` value to represent the *nano-of-second*

The nano-of-second must be a value in the range [0, 999999999]. This representation is reflected in the methods provided for dealing with instants. The `Instant` class shares many of the method name prefixes and the common method names in Table 17.2, p. 1026, and Table 17.3, p. 1026, with the other temporal classes, respectively. Although the `Instant` class has many methods analogous to the other temporal classes, as we shall see, there are also differences. `Instant` objects are, like objects of the other temporal classes, immutable and thread-safe.

Creating Instants

The `Instant` class provides the following predefined instants:

```
static Instant EPOCH
static Instant MAX
static Instant MIN
```

These static fields of the `Instant` class define constants for the epoch (`1970-01-01T00:00:00Z`), the maximum (`1000000000-12-31T23:59:59.999999999Z`), and the minimum instants (`-1000000000-01-01T00:00Z`), respectively.

Following are selected methods for creating and converting instances of the `Instant` class:

```
static Instant now()
```

Returns the current instant based on the system clock.

Click here to view code image

```
static Instant ofEpochMilli(long epochMilli)
static Instant ofEpochSecond(long epochSecond)
static Instant ofEpochSecond(long epochSecond, long nanoAdjustment)
```

These static factory methods return an `Instant` based on the millisecond, second, and nanosecond specified.

Nanoseconds are implicitly set to zero. The argument values can be negative. Note that the amount is specified as a `long` value.

```
String toString()
```

Returns a text representation of this `Instant`, such as `"2021-01-11T14:18:30Z"`. Formatting is based on the ISO instant format for date-time:

Click here to view code image

```
uuuu-MM-ddTHH:mm:ss.SSSSSSSSSZ
```

where `Z` designates the UTC standard (also known as *Coordinated Universal Time*).

Click here to view code image

```
static Instant parse(CharSequence text)
```

Returns an `Instant` parsed from a character sequence, such as `"2021-04-28T14:18:30Z"`, based on the ISO instant format. A `DateTimeParse-Exception` is thrown if the text cannot be parsed to an instant.

Click here to view code image

```
ZonedDateTime atZone(ZoneId zone)
```

Returns a `ZonedDateTime` by combining this instant with the specified time zone (p. 1072).

Analogous to the other temporal classes, the `Instant` class also provides the `now()` method to obtain the current instant from the system clock.

Click here to view code image

```
Instant currentInstant = Instant.now();      // 2021-03-09T10:48:01.914826Z
```

The `Instant` class provides the static factory method `ofEpochUNIT()` to construct instants from seconds and nanoseconds. There is no method to construct an instant from just nanoseconds.

Click here to view code image

```
Instant inst1 = Instant.ofEpochMilli(-24L*60*60*1000);// Date 1 day before epoch.
Instant inst2 = Instant.ofEpochSecond(24L*60*60);     // Date 1 day after epoch.
Instant inst3 = Instant.ofEpochSecond(24L*60*60 - 1,  // Date 1 day after epoch.
                                      1_000_000_000L);
out.println("A day before: " + inst1); // Date 1 day before: 1969-12-31T00:00:00Z
out.println("A day after:  " + inst2); // Date 1 day after : 1970-01-02T00:00:00Z
out.println("A day after:  " + inst3); // Date 1 day after : 1970-01-02T00:00:00Z
```

Note that the amount specified is a `long` value. The last statement above also illustrates that the nanosecond is adjusted so that it is always between 0 and 999,999,999. The adjustment results in the nanosecond being set to 0 and the second being incremented by 1.

The `toString()` method of the `Instant` class returns a text representation of an `Instant` based on the ISO standard. The code shows the text representation of the instant 500 nanoseconds after the epoch.

Click here to view code image

```
Instant inst4 = Instant.ofEpochSecond(0, 500);
out.println("Default format:  " + inst4);       // 1970-01-01T00:00:00.000000500Z
```

The `Instant` class also provides the `parse()` static method to create an instant from a string that contains a text representation of an instant, based on the ISO standard. Apart from treating the value of the nanosecond as optional, the method is strict in parsing the string. If the format of the string is not correct, a `DateTimeParseException` is thrown.

Click here to view code image

```
Instant instA = Instant.parse("1970-01-01T00:00:00.000000500Z");
Instant instB = Instant.parse("1949-03-01T12:30:15Z");
Instant instC = Instant.parse("-1949-03-01T12:30:15Z");
Instant instD = Instant.parse("-1949-03-01T12:30:15"); // DateTimeParseException!
```

The code below illustrates creating an `Instant` by combining a `LocalDateTime` object with a time zone offset. Three different zone-time offsets are specified at (2), (3), and (4) to convert the date-time created at (1) to an `Instant` on the UTC timeline, which has offset zero. Note that an offset ahead of UTC is subtracted and an offset behind UTC is added to adjust the values of the date/time from the `LocalDateTime` object to the UTC timeline.

Click here to view code image

```
LocalDateTime ldt = LocalDate.of(2021, 12, 25).atStartOfDay();  //(1)
Instant i1 = ldt.toInstant(ZoneOffset.of("+02:00"));     // (2) Ahead of UTC
Instant i2 = ldt.toInstant(ZoneOffset.UTC);              // (3) At UTC
Instant i3 = ldt.toInstant(ZoneOffset.of("-02:00"));     // (4) Behind UTC
System.out.println("ldt: " + ldt);
System.out.println("i1:  " + i1);
System.out.println("i2:  " + i2);
System.out.println("i3:  " + i3);
```

Output from the code:

```
ldt: 2021-12-25T00:00
i1:  2021-12-24T22:00:00Z
i2:  2021-12-25T00:00:00Z
i3:  2021-12-25T02:00:00Z
```

Click here to view code image

```
// LocalDateTime
default Instant toInstant(ZoneOffset offset)
```

Converts a date-time to an instant by combining this `LocalDateTime` object with the specified time zone. The valid offset in Java is in the range from –18 to +18 hours. The absolute value of the offset is added to or subtracted from the date-time depending on whether it is specified as a negative or positive value, respectively, keeping in mind that an `Instant` represents a point in time on the UTC timeline.

This method is inherited by the `LocalDateTime` class from its superinterface `java.time.chrono.ChronoLocalDateTime`.

Accessing Temporal Fields in an Instant

The `Instant` class provides the following selected methods to access temporal fields in an instance of the class:

```
int getNano()
long getEpochSecond()
```

Return the number of nanoseconds and the number of seconds represented by this instant from the start of the epoch, respectively. Note that the method names are without the `s` at the end.

Click here to view code image

```
int get(TemporalField field)
long getLong(TemporalField field)
```

The `get(field)` method will return the value of the specified `field` in this `Instant` as an `int`. Only the following `ChronoField` constants are supported: `NANO_OF_SECOND`, `MICRO_OF_SECOND`, `MILLI_OF_SECOND`, and `INSTANT_SECONDS` (p. 1046). The first three fields will always return a valid value, but the `INSTANT_SECONDS` field will throw a `DateTimeException` if the value does not fit into an `int`. All other fields result in an `UnsupportedTemporalTypeException`.

As the `getLong(field)` method returns the value of the specified `field` in this `Instant` as a `long`, there is no problem with overflow in returning a value designated by any of the four fields mentioned earlier.

Click here to view code image

```
boolean isSupported(TemporalField field)
```

The `isSupported(field)` determines whether the specified `field` is supported by this instant.

```
long toEpochMilli()
```

Returns the number of milliseconds that represent this `Instant` from the start of the epoch. The method throws an `ArithmeticException` in case of number overflow.

The code below shows how the `getNano()` and `getEpochSecond()` methods of the `Instant` class read the value of the nanosecond and the epoch-second fields of an `Instant` object, respectively.

Click here to view code image

```
Instant inst = Instant.ofEpochSecond(24L*60*60,    // 1 day and
                                     555_555_555L);// 555555555 ns after epoch.
out.println(inst);                   // 1970-01-02T00:00:00.555555555Z
out.println(inst.getNano());         // 555555555 ns
out.println(inst.getEpochSecond());  // 86400 s
```

Reading the nanosecond and epoch-second fields of an `Instant` *in different units* can be done using the `get(field)` method. Note the value of the nanosecond field expressed in different units using `ChronoField` constants. To avoid a `DateTimeException` when number overflow occurs, the `getLong(field)` method is used instead of the `get(field)` method in accessing the epoch-second field.

[Click here to view code image]

```
out.println(inst.get(ChronoField.NANO_OF_SECOND));      // 555555555 ns
out.println(inst.get(ChronoField.MICRO_OF_SECOND));     // 555555 micros
out.println(inst.get(ChronoField.MILLI_OF_SECOND));     // 555 ms
out.println(inst.getLong(ChronoField.INSTANT_SECONDS)); // 86400 s
//out.println(inst.get(ChronoField.INSTANT_SECONDS));   // DateTimeException
//out.println(inst.get(ChronoField.HOUR_OF_DAY));       // UnsupportedTemporal-
                                                        // TypeException
```

The `Instant` class provides the `toEpochMilli()` method to derive the position of the instant measured in milliseconds from the epoch; that is, the second and nanosecond fields are converted to milliseconds. Converting 1 day (86400 s) and 555555555 ns results in 86400555 ms.

[Click here to view code image]

```
out.println(inst.toEpochMilli());                        // 86400555 ms
```

Comparing Instants

The methods `isBefore()` and `isAfter()` can be used to determine if one instant is before or after the other on the timeline, respectively.

[Click here to view code image]

```
// instA is  1970-01-01T00:00:00.000000500Z
// instB is  1949-03-01T12:30:15Z
// instC is -1949-03-01T12:30:15Z
out.println(instA.isBefore(instB));            // false
out.println(instA.isAfter(instC));             // true
```

The `Instant` class also overrides the `equals()` method and the `hashCode()` method of the `Object` class, and implements the `Comparable<Instant>` interface. Instants can readily be used in collections. The code below illustrates comparing instants.

Click here to view code image

```
out.println(instA.equals(instB));                 // false
out.println(instA.equals(instC));                 // false

List<Instant> list = Arrays.asList(instA, instB, instC);
Collections.sort(list);             // Natural order: position on the timeline.
// [-1949-03-01T12:30:15Z, 1949-03-01T12:30:15Z, 1970-01-01T00:00:00.000000500Z]
```

Click here to view code image

```
boolean isBefore(Instant other)
boolean isAfter(Instant other)
```

Determine whether this `Instant` is before or after the `other` instant on the timeline, respectively.

```
boolean equals(Object other)
```

Determines whether this `Instant` is equal to the `other` instant, based on the timeline position of the instants.

```
int hashCode()
```

Returns a hash code for this `Instant`.

```
int compareTo(Instant other)
```

Compares this `Instant` with the `other` instant, based on the timeline position of the instants.

Creating Modified Copies of Instants

The `Instant` class provides the `with(field, newValue)` method that returns a copy of this instant with either the epoch-second or the nano-of-second set to a new value, while the other one is unchanged.

Click here to view code image

```
Instant with(TemporalField field, long newValue)
```

Returns a copy of this instant where either the epoch-second or the nano-of-second is set to the specified value. The value of the other is retained.

This method only supports the following `ChronoField` constants: `NANO_OF_SECOND`, `MICRO_OF_SECOND`, `MILLI_OF_SECOND`, and `INSTANT_SECONDS` (p. 1046). For the first three fields, the nano-of-second is replaced by appropriately converting the specified value, and the epoch-second will be unchanged in the copy returned by the method. For the `INSTANT_SECONDS` field, the epoch-second will be replaced and the nanosecond will be unchanged in the copy returned by the method. Valid values that can be specified with these constants are [0–999999999], [0–999999], [0–999], and a `long`, respectively.

This method throws a `DateTimeException` if the field cannot be set, an `Unsupported-TemporalTypeException` if the field is not supported, and an `ArithmeticException` if number overflow occurs.

In the code below, the three instants `i1`, `i2`, and `i3` will have the nano-of-second set to 5,000,000,000 nanoseconds using the `with()` method, but the epoch-second will not be changed.

Click here to view code image

```
Instant i0, i1, i2, i3;
i0 = Instant.now();
out.println(i0);                                    // 2021-02-28T08:43:35.864Z
i1 = i0.with(ChronoField.NANO_OF_SECOND,  500_000_000);// 500000000 ns.
i2 = i0.with(ChronoField.MICRO_OF_SECOND, 500_000);    // 500000x1000 ns.
i3 = i0.with(ChronoField.MILLI_OF_SECOND, 500);        // 500x1000000 ns.
out.println(i1);                                    // 2021-02-28T08:43:35.500Z

out.println(i1.equals(i2));                         // true
out.println(i1.equals(i3));                         // true
```

In the code below, `oneInstant` has the nano-of-second set to 500,000,000 nanoseconds and the epoch-second set to 1 day after the epoch.

Click here to view code image

```
Instant oneInstant = Instant.now()
                            .with(ChronoField.MILLI_OF_SECOND, 500)
                            .with(ChronoField.INSTANT_SECONDS, 24L*60*60);
out.println(oneInstant);                            // 1970-01-02T00:00:00.500Z
```

Temporal Arithmetic with Instants

The `Instant` class provides *plus* and *minus* methods that return a copy of the original instant that has been *incremented or decremented by a specific amount* specified in terms of either seconds, milliseconds, or nanoseconds. Each amounts below is explicitly designated as a `long` to avoid problems if the amount does not fit into an `int`.

Click here to view code image

```
Instant event =
    Instant.EPOCH                     //            1970-01-01T00:00:00Z
           .plusSeconds(7L*24*60*60) // (+7days)   1970-01-08T00:00:00Z
           .plusSeconds(6L*60*60)    // (+6hrs)    1970-01-08T06:00:00Z
           .plusSeconds(5L*60)       // (+5mins)   1970-01-08T06:05:00Z
           .plusSeconds(4L)          // (+4s)      1970-01-08T06:05:04Z
           .plusMillis(3L*100)       // (+3ms)     1970-01-08T06:05:04.003Z
           .plusNanos(2L*1_000)      // (+2micros) 1970-01-08T06:05:04.003002Z
           .plusNanos(1L);           // (+1ns)     1970-01-08T06:05:04.003002001Z
```

However, it is more convenient to express the above calculation using the `plus(amount, unit)` method, which also allows the amount to be qualified by a unit. This is illustrated by the statement below, which is equivalent to the one above.

Click here to view code image

```
Instant ptInTime =
     Instant.EPOCH                           // 1970-01-01T00:00:00Z
            .plus(7L, ChronoUnit.DAYS)       // 1970-01-08T00:00:00Z
            .plus(6L, ChronoUnit.HOURS)      // 1970-01-08T06:00:00Z
            .plus(5L, ChronoUnit.MINUTES)    // 1970-01-08T06:05:00Z
            .plus(4L, ChronoUnit.SECONDS)    // 1970-01-08T06:05:04Z
            .plus(3L, ChronoUnit.MILLIS)     // 1970-01-08T06:05:04.003Z
            .plus(2L, ChronoUnit.MICROS)     // 1970-01-08T06:05:04.003002Z
            .plus(1L, ChronoUnit.NANOS);     // 1970-01-08T06:05:04.003002001Z
```

The code below shows the `plus()` method of the `Instant` class that takes a `Duration` (**p. 1064**) as the amount to add.

Click here to view code image

```
Instant start = Instant.EPOCH
                       .plus(20, ChronoUnit.MINUTES);// 1970-01-01T00:20:00Z
Duration length = Duration.ZERO.plusMinutes(90);     // PT1H30M (90 mins)
Instant end = start.plus(length);                    // 1970-01-01T01:50:00Z
```

The `until()` method calculates the amount of time between two instants in terms of the unit specified in the method.

Click here to view code image

```
long eventDuration1 = start.until(end, ChronoUnit.MINUTES);  // 90 minutes
```

```
long eventDuration2 = start.until(end, ChronoUnit.HOURS);    // 1 hour
```

As an `Instant` does not represent an amount of time, but a point on the timeline, it cannot be used in temporal arithmetic with other temporal objects. Although an `Instant` incorporates a date, it is not possible to access it in terms of year and month.

Click here to view code image

```
Instant plusSeconds/minusSeconds(long seconds)
Instant plusMillis/minusMillis(long millis)
Instant plusNanos/minusNanos(long nanos)
```

Return a copy of this instant, with the specified amount added or subtracted. Note that the argument type is `long`.

The methods throw a `DateTimeException` if the result is not a valid instant, and an `ArithmeticException` if numeric flow occurs during the operation.

Click here to view code image

```
Instant plus(long amountToAdd, TemporalUnit unit)
Instant minus(long amountToSub, TemporalUnit unit)
```

Return a copy of this instant with the specified amount added or subtracted, respectively, where the specified `TemporalUnit` qualifies the amount (p. 1044).

The following units, defined as constants by the `ChronoUnit` class, can be used to qualify the amount: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS` (p. 1044).

A method call can result in any one of these exceptions: `DateTimeException` (if the operation cannot be performed), `UnsupportedTemporalTypeException` (if the unit is not supported), or `ArithmeticException` (if numeric overflow occurs).

Click here to view code image

```
Instant isSupported(TemporalUnit unit)
```

Returns `true` if the specified unit is supported (p. 1044), in which case, the unit can be used in plus/minus operations on an instant. If the specified unit is not supported, the plus/minus methods that accept a unit will throw an exception.

Click here to view code image

```
Instant plus(TemporalAmount amountToAdd)
Instant minus(TemporalAmount amountToSubtract)
```

Return a copy of this instant, with the specified amount added or subtracted. The amount is typically defined as a `Duration`.

A method call can result in any one of these exceptions: `DateTimeException` (if the operation cannot be performed) or `ArithmeticException` (if numeric overflow occurs).

Click here to view code image

```
long until(Temporal endExclusive, TemporalUnit unit)
```

Calculates the amount of time between two temporal objects in terms of the specified `TemporalUnit` (p. 1044). The start and end points are this temporal object and the specified temporal argument, where the end point is excluded.

The start point is an `Instant`, and the end point temporal is converted to an `Instant`, if necessary.

The following units, defined as constants by the `ChronoUnit` class, can be used to indicate the unit in which the result should be returned: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS` (p. 1044).

The `until()` method can result in any one of these exceptions: `DateTimeException` (the temporal amount cannot be calculated or the end temporal cannot be converted to the appropriate temporal object), `UnsupportedTemporalTypeException` (the unit is not supported), or `ArithmeticException` (numeric overflow occurred).

Converting Instants

Each of the classes `LocalTime`, `LocalDate`, `LocalDateTime`, and `ZonedDateTime` provides the `ofInstant()` method to obtain a temporal object from an `Instant`. The code below shows how instants can be converted to other temporal objects for a given time zone. For date/time represented by this particular instant, the offset for the time zone `"America/New_York"` is -4 hours from UTC.

Click here to view code image

```
Instant instant = Instant.parse("2021-04-28T03:15:00Z");
ZoneId zid = ZoneId.of("America/New_York");
LocalTime lt = LocalTime.ofInstant(instant, zid);           // 10:18:30
LocalDate ld = LocalDate.ofInstant(instant, zid);           // 2021-04-27
LocalDateTime ldt = LocalDateTime.ofInstant(instant, zid);  // 2021-04-27T23:15
```

```
ZonedDateTime zdt = ZonedDateTime.ofInstant(instant, zid);
    // 2021-04-27T23:15-04:00[America/New_York]
```

Click here to view code image

```
static TemporalType ofInstant(Instant instant, ZoneId zone)
```

Creates a *`TemporalType`* object from the given `Instant` and `ZoneId` (p. 1072), where *`TemporalType`* can be `LocalTime`, `LocalDate`, `LocalDateTime`, or `ZonedDateTime`.

17.5 Working with Periods

For representing *an amount of time*, the Date and Time API provides the two classes `Period` and `Duration`. We will concentrate on the `Period` class in this section and discuss the `Duration` class in §17.6, p. 1064.

The `Period` class essentially represents a *date-based amount of time* in terms of years, months, and days, whereas, the `Duration` class represents a *time-based amount of time* in terms of seconds and nanoseconds.

The date-based `Period` class can be used with the `LocalDate` class, and not surprisingly, the time-based `Duration` class can be used with the `LocalTime` class. Of course, the `LocalDateTime` class can use both temporal amount classes.

The `Period` and `Duration` classes are in the same package (`java.time`) as the temporal classes, and the repertoire of methods they provide should look familiar, as they share many of the method prefixes with the temporal classes (Table 17.2, p. 1026).

The mantra of immutable and thread-safe objects also applies to both the `Period` and the `Duration` classes.

Creating Periods

Like the temporal classes, the `Period` class does not provide any `public` constructors, but rather provides an overloaded static factory method `of()` to construct periods of different lengths, based on date units.

Click here to view code image

```
Period p = Period.of(2, 4, 8);          // (1)
System.out.println(p);                  // (2) P2Y4M8D (2 Years, 4 Months, 8 Days)
Period p1 = Period.ofYears(10);         // P10Y, period of 10 years.
Period p2 = Period.ofMonths(14);        // P14M, period of 14 months.
Period p3 = Period.ofDays(40);          // P40D, period of 40 days.
Period p4 = Period.ofWeeks(2);          // P14D, period of 14 days (2 weeks).
```

The most versatile `of()` method requires the amount of time for all date units: years, months, and days, as at (1). Other `of()` methods create a period based on a particular date unit, as shown in the examples above.

The `toString()` method of the `Period` class returns a text representation of a `Period` according to the ISO standard: `P`*y*`Y`*m*`M`*d*`D`—that is, *y* Years, *m* Months, and *d* Days. The output from (2) above, `P2Y4M8D`, indicates a period of 2 years, 4 months, and 8 days.

The code snippet below does *not* create a period of 3 years, 4 months, and 5 days— it creates a period of only 5 days. The first method call is invoked with the class name, and the subsequent method calls are on the new `Period` object returned as a consequence of the previous call. The `of()` method creates a new `Period` object based on its argument.

Click here to view code image

```
Period period = Period.ofYears(3).ofMonths(4).ofDays(5); // P5D. Logical error.
```

As we would expect, we can create a period that represents the amount of time between two dates by calling the static method `between()` of the `Period` class.

Click here to view code image

```
LocalDate d1 = LocalDate.of(2021, 3, 1);  // 2021-03-01
LocalDate d2 = LocalDate.of(2022, 3, 1);  // 2022-03-01
Period period12 = Period.between(d1, d2); // P1Y
Period period21 = Period.between(d2, d1); // P-1Y
```

The `Period` class also provides the static method `parse()` to create a period from a string that contains a text representation of a period in the ISO standard. If the format of the string is not correct, a `java.time.format.DateTimeParseException` is thrown.

Click here to view code image

```
Period period2 = Period.parse("P1Y15M20D"); // 1 year, 15 months, 20 days
Period period3 = Period.parse("P20D");      // 20 days
Period period4 = Period.parse("P5W");       // 35 days (5 weeks)
//  Period pX = Period.parse("P24H"); // java.time.format.DateTimeParseException
```

```
static Period ZERO
```

This constant defines a `Period` of length zero (`P0D`).

Click here to view code image

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

These static factory methods return a `Period` representing an amount of time equal to the specified value of a date unit. Date units that are implicit are set to zero. A week is equal to 7 days. The argument value can be negative.

Click here to view code image

```
static Period between(LocalDate startDateInclusive,
                      LocalDate endDateExclusive)
```

This static method returns a `Period` consisting of the number of years, months, and days between the two dates. The calculation excludes the end date. The result of this method can be a negative period if the end date is before the start date.

```
String toString()
```

Returns a text representation of a `Period` according to the ISO standard. Typical formats are `P`*y*`Y`*m*`M`*d*`D` and `P`*n*`W`—that is, *y* Years, *m* Months, and *d* Days, or *n* Weeks.

Click here to view code image

```
static Period parse(CharSequence text)
```

This static method returns a `Period` parsed from a character sequence—for example, `"P3Y10M2D"` (3 years, 10 months, 2 days). A `java.time.format.Date-TimeParseException` is thrown if the text cannot be parsed to a period.

Accessing Date Units in a Period

The `Period` class provides the obvious `get`*`XXX`*`()` methods to read the values of date units of a `Period` object, where *`XXX`* can be `Years`, `Months`, or `Days`.

Click here to view code image

```
Period period5 = Period.of(2, 4, -10);
System.out.println("Period: " + period5);              // Period: P2Y4M-10D
System.out.println("Years:  " + period5.getYears());   // Years:  2
System.out.println("Months: " + period5.getMonths()); // Months: 4
System.out.println("Days:   " + period5.getDays());    // Days:   -10
```

Reading the value of date units of a `Project` object can also be achieved using the `get(unit)` method, where only the date units shown in the code below are allowed. A list of these valid temporal units can be obtained by calling the `getUnits()` method of the `Period` class.

The class also has methods to check if *any* date unit of a period has a negative value or if *all* date units of a period have the value zero.

Click here to view code image

```
System.out.println("Years:  " + period5.get(ChronoUnit.YEARS)); // Years:  2
System.out.println("Months: " + period5.get(ChronoUnit.MONTHS));// Months: 4
System.out.println("Days:   " + period5.get(ChronoUnit.DAYS));  // Days: -10
List<TemporalUnit> supportedUnits = period5.getUnits(); // [Years, Months, Days]

System.out.println("Total months: " + period5.toTotalMonths()); // 28 months
System.out.println(period5.isNegative());                       // true
System.out.println(period5.isZero());                           // false
```

The class `Period` provides the method `toTotalMonths()` to derive the *total* number of months in a period. However, this calculation is solely based on the number of years and months in the period; the number of days is *not* considered. A `Period` just represents an amount of time, so it has no notion of a date. Conversion between months and years is not a problem, as 1 year is 12 months. However, conversion between the number of days and the other date units is problematic. The number of days in a year and in a month are very much dependent on whether the year is a leap year and on a particular month in the year, respectively. A `Period` is oblivious to both the year and the month in the year, as it represents an *amount* of time and *not* a *point* on the timeline.

```
int getYears()
int getMonths()
int getDays()
```

Return the value of a specific date unit of this period, indicated by the name of the method.

```
long get(TemporalUnit unit)
```

Returns the value of the specified `unit` in this `Period`. The only supported date `ChronoUnit`s are `YEARS`, `MONTHS`, and `DAYS` (p. 1044). All other units throw an exception.

```
List<TemporalUnit> getUnits()
```

Returns the list of date units supported by this period: `YEARS`, `MONTHS`, and DAYS (p. 1044). These date units can be used in the `get(TemporalUnit)` method.

```
long toTotalMonths()
```

Returns the total number of months in this period, based on the values of the years and months units. The value of the days unit is not considered.

```
boolean isNegative()
```

Determines whether the value of any date units of this period is negative.

```
boolean isZero()
```

Determines whether the values of all date units of this period are zero.

Comparing Periods for Equality

The `Period` class overrides the `equals()` and `hashCode()` methods of the `Object` class, but the class does *not* implement the `Comparable<E>` interface. The value of each date unit is compared individually, and must have the same value to be considered equal. A period of 1 year and 14 months is not equal to a period of 2 years and 2 months, or to a period of 26 months. For this reason, `Period` objects do not implement the `Comparable<E>` interface.

Click here to view code image

```
Period px = Period.of(1, 14, 0);
Period py = Period.of(2, 2, 0);
Period pz = Period.ofMonths(26);
System.out.println(px.equals(py));          // false
System.out.println(px.equals(pz));          // false
System.out.println(px.equals(Period.ZERO)); // false
```

```
boolean equals(Object obj)
```

Determines whether this period is equal to another period, meaning that each corresponding date unit has the same value.

```
int hashCode()
```

Returns a hash code for this period.

Creating Modified Copies of Periods

The `Period` class provides `with` methods to set a new value for each date unit individually, while the values of the other date units remain unchanged. Note that each method call returns a new `Period` object, and chaining method calls work as expected.

Click here to view code image

```
Period p5 = Period.of(2, 1, 30) // P2Y1M30D
    .withYears(3)               // P3Y1M30D, sets the number of years
    .withMonths(16)             // P3Y16M30D, sets the number of months
    .withDays(1);               // P3Y16M1D, sets the number of days
```

```
Period withYears(int years)
Period withMonths(int months)
Period withDays(int days)
```

Return a copy of this period where a specific date unit is set to the value of the argument. The values of the other date units are not affected.

Temporal Arithmetic with Periods

The `Period` class provides *plus* and *minus* methods that return a copy of the original object that has been *incremented or decremented by a specific amount* specified in terms of a date unit—for example, as a number of years, months, or days. As the following code snippets show, only the value of a specific date unit is changed; the values of other date fields are unaffected. There is no implicit normalization performed, unless the `normalized()` method that normalizes only the months is called, adjusting the values of the months and years as necessary.

Click here to view code image

```
Period p6 = Period.of(2, 10, 30)  // P2Y10M30D
    .plusDays(10)                 // P2Y10M40D
    .plusMonths(8)                // P2Y18M40D
    .plusYears(1)                 // P3Y18M40D
    .normalized();                // P4Y6M40D
```

We can do simple arithmetic with periods. The code examples below use the `plus()` and the `minus()` methods of the `Period` class that take a `TemporalAmount` as an argument. Both the `Period` and the `Duration` classes implement the `TemporalAmount` interface. In the last assignment statement, we have shown the state of both new `Period` objects that are created.

Click here to view code image

```
Period p7 = Period.of(1, 1, 1);                  // P1Y1M1D
Period p8 = Period.of(2, 12, 30);                // P2Y12M30D
Period p9 = p8.minus(p7);                        // P1Y11M29D
p8 = p8.plus(p7).plus(p8);                       // P3Y13M31D, P5Y25M61D
```

Click here to view code image

```
Period plusYears/minusYears(long years)
Period plusMonths/minusMonths(long months)
Period plusDays/minusDays(long days)
```

Return a copy of this period, with the specified value for the date unit added or subtracted. The values of other date units are unaffected.

Click here to view code image

```
Period plus(TemporalAmount amount)
Period minus(TemporalAmount amount)
```

Return a copy of this period, with the specified temporal amount added or subtracted. The amount is of the interface type `TemporalAmount` that is implemented by the classes `Period` and `Duration`, but only `Period` is valid here. The operation is performed separately on each date unit. There is no normalization performed. A `DateTimeException` is thrown if the operation cannot be performed.

```
Period normalized()
```

Returns a copy of this period where the years and months are normalized. The number of days is not affected.

Click here to view code image

```
Period negated()
Period multipliedBy(int scalar)
```

Return a new instance of `Period` where the value of each date unit in this period is individually negated or multiplied by the specified `scalar`, respectively.

We can also do simple arithmetic with dates and periods. The following code uses the `plus()` and `minus()` methods of the `LocalDate` class that take a `TemporalAmount` as an argument

(p. 1040). Note the adjustments performed to the month and the day fields to return a valid date in the last assignment statement.

Click here to view code image

```
Period p10 = Period.of(1, 1, 1);                // P1Y1M1D
LocalDate date1 = LocalDate.of(2021, 3, 1);     // 2021-03-01
LocalDate date2 = date1.plus(p10);              // 2022-04-02
LocalDate date3 = date1.minus(p10);             // 2020-01-31
```

We can add and subtract periods from `LocalDate` and `LocalDateTime` objects, but not from `LocalTime` objects, as a `LocalTime` object has only time fields.

Click here to view code image

```
LocalTime time = LocalTime.NOON;
time = time.plus(p10);     // java.time.temporal.UnsupportedTemporalTypeException
```

Example 17.5 is a simple example to illustrate implementing period-based loops. The method `reserveDates()` at (1) is a stub for reserving certain dates, depending on the period passed as an argument. The `for(;;)` loop at (2) uses the `Local-Date.isBefore()` method to terminate the loop, and the `LocalDate.plus()` method to increment the current date with the specified period.

Example 17.5 *Period-Based Loop*

Click here to view code image

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

public class PeriodBasedLoop {
  public static void main(String[] args) {
    reserveDates(Period.ofDays(7),
                 LocalDate.of(2021, 10, 20), LocalDate.of(2021, 11, 20));
    System.out.println();
    reserveDates(Period.ofMonths(1),
                 LocalDate.of(2021, 10, 20), LocalDate.of(2022, 1, 20));
    System.out.println();
    reserveDates(Period.of(0, 1, 7),
                 LocalDate.of(2021, 10, 20), LocalDate.of(2022, 1, 21));
  }

  public static void reserveDates(Period period,                  // (1)
                                  LocalDate fromDate,
                                  LocalDate toDateExclusive) {
    System.out.println("Start date: " + fromDate);
    for (LocalDate date = fromDate.plus(period);                  // (2)
```

```
            date.isBefore(toDateExclusive);
            date = date.plus(period)) {
         System.out.println("Reserved (" + period + "): " + date);
       }
       System.out.println("End date: " + toDateExclusive);
     }
   }
```

Output from the program:

Click here to view code image

```
Start date: 2021-10-20
Reserved (P7D): 2021-10-27
Reserved (P7D): 2021-11-03
Reserved (P7D): 2021-11-10
Reserved (P7D): 2021-11-17
End date: 2021-11-20

Start date: 2021-10-20
Reserved (P1M): 2021-11-20
Reserved (P1M): 2021-12-20
End date: 2022-01-20

Start date: 2021-10-20
Reserved (P1M7D): 2021-11-27
Reserved (P1M7D): 2022-01-03
End date: 2022-01-21
```

We conclude this section with **Example 17.6**, which brings together some of the methods of the Date and Time API. Given a date of birth, the method `birthdayInfo()` at (1) calculates the age and the time until the next birthday. The age is calculated at (2) using the `Period.between()` method, which computes the period between two dates. The date for the next birthday is set at (3) as the birth date with the current year. The `if` statement at (4) adjusts the next birthday date by 1 year at (5), if the birthday has already passed. The statement at (6) calculates the time until the next birthday by calling the `LocalDate.until()` method. We could also have used the `Period.between()` method at (6). The choice between these methods really depends on which method makes the code more readable in a given context.

Example 17.6 *More Temporal Arithmetic*

Click here to view code image

```
import java.time.LocalDate;
import java.time.Month;
import java.time.Period;

public class ActYourAge {
```

```
  public static void main(String[] args) {
    birthdayInfo(LocalDate.of(1981, Month.AUGUST, 19));
    birthdayInfo(LocalDate.of(1935, Month.JANUARY, 8));
  }

  public static void birthdayInfo(LocalDate dateOfBirth) {           // (1)
    LocalDate today = LocalDate.now();
    System.out.println("Today:         " + today);
    System.out.println("Date of Birth: " + dateOfBirth);
    Period p1 = Period.between(dateOfBirth, today);                  // (2)
    System.out.println("Age:           " +
                             p1.getYears()  + " years, " +
                             p1.getMonths() + " months, and " +
                             p1.getDays()   + " days");

    LocalDate nextBirthday =  dateOfBirth.withYear(today.getYear()); // (3)
    if (nextBirthday.isBefore(today) ||                              // (4)
        nextBirthday.isEqual(today)) {
      nextBirthday = nextBirthday.plusYears(1);                      // (5)
    }
    Period p2 = today.until(nextBirthday);                           // (6)
    System.out.println("Birthday in " + p2.getMonths() + " months and " +
                                    p2.getDays()   + " days");
  }
}
```

Possible output from the program:

Click here to view code image

```
Today:         2021-03-05
Date of Birth: 1981-08-19
Age:           39 years, 6 months, and 14 days
Birthday in 5 months and 14 days
Today:         2021-03-05
Date of Birth: 1935-01-08
Age:           86 years, 1 months, and 25 days
Birthday in 10 months and 3 days
```

17.6 Working with Durations

The `java.time.Duration` class implements a *time-based amount of time* in terms of *seconds* and *nanoseconds*, using a `long` and an `int` value for these time units, respectively. Although the `Duration` class models an amount of time in terms of seconds and nanoseconds, a duration can represent an amount of time in terms of days, hours, and minutes. As these time units have fixed lengths, it makes interoperability between these units possible. The time-based `Duration` class can be used with the `LocalTime` and `LocalDateTime` classes, as these classes have time fields. In contrast, the `Period` class essentially represents a *date-based amount of time* in terms of years, months, and days (p. 1057).

The `Period` and `Duration` classes are in the same package (`java.time`) as the temporal classes. The `Period` and the `Duration` classes provide similar methods, as they share many of the method prefixes and common methods with the temporal classes (**Table 17.2**, **p. 1026**, and **Table 17.3**, **p. 1026**). Their objects are immutable and thread-safe. However, there are also differences between the two classes (**p. 1072**). Familiarity with one would go a long way toward understanding the other.

Creating Durations

Like the `Period` class, the `Duration` class provides the static factory methods `of`*UNIT*`()` to construct durations with different units.

Click here to view code image

```
Duration d1 = Duration.ofDays(1L);                               // PT24H
Duration d2 = Duration.ofHours(24L);                             // PT24H
Duration d3 = Duration.ofMinutes(24L*60);                        // PT24H
Duration d4 = Duration.ofSeconds(24L*60*60);                     // PT24H
Duration d5 = Duration.ofMillis(24L*60*60*1000);                 // PT24H
Duration d6 = Duration.ofSeconds(24L*60*60 - 1, 1_000_000_000L); // (1) PT24H
Duration d7 = Duration.ofNanos(24L*60*60*1_000_000_000); // (2) PT24H
Duration d8 = Duration.ofNanos(24*60*60*1_000_000_000);  // (3) PT-1.857093632S
```

The durations created above all have a length of 1 day, except for the one in the last declaration statement. Note that the amount specified should be a `long` value. It is a good defensive practice to always designate the amount as such in order to avoid inadvertent problems if the amount does not fit into an `int`. The designation `L` should be placed such that there is no danger of any previous operation in the expression causing a rollover. This problem is illustrated at (3), where the `int` value of the argument expression is rolled over, as it is greater than `Integer.MAX_VALUE`.

The statement at (1) above also illustrates that the value of the nanoseconds is adjusted so that it is always between 0 and 999,999,999. The adjustment at (1) results in the value 0 for nanoseconds and the number of seconds being incremented by 1.

Calling the `toString()` method on the first seven declarations above, the result is the string `"PT24H"` (a duration of 24 hours), whereas for the last duration at (3), the result string is `"PT-1.857093632S"`, which clearly indicates that the `int` amount was not interpreted as intended.

The previous declarations are equivalent to the ones below, where the amount is *qualified* with a specific unit in the call to the `of(value, unit)` method.

Click here to view code image

```
Duration d11 = Duration.of(1L, ChronoUnit.DAYS);              // P24H
Duration d22 = Duration.of(24L, ChronoUnit.HOURS);            // P24H
```

```
Duration d33 = Duration.of(24L*60, ChronoUnit.MINUTES);       // P24H
Duration d44 = Duration.of(24L*60*60, ChronoUnit.SECONDS);    // P24H
Duration d88 = Duration.of(24L*60*60*1000, ChronoUnit.MILLIS);// P24H
Duration d77 = Duration.of(24L*60*60*1_000_000_000,
                          ChronoUnit.NANOS);                 // P24H
```

The code snippet below does *not* create a duration of 8 days—it creates a duration of 24 hours. The first method call is invoked with the class name, and the subsequent method call is on the new `Duration` object returned as a consequence of the first call. The `of()` method creates a new `Duration` object based on its argument.

Click here to view code image

```
Duration duration = Duration.ofDays(7).ofHours(24);   // PT24H. Logical error.
```

Like the `Period` class, we can create a duration that represents the amount of time between two temporal objects by calling the static method `between()` of the `Duration` class.

Click here to view code image

```
LocalTime startTime = LocalTime.of(14, 30);                   // 14:30
LocalTime endTime   = LocalTime.of(17, 45, 15);               // 17:45:15
Duration interval1 = Duration.between(startTime, endTime);   // PT3H15M15S
Duration interval2 = Duration.between(endTime, startTime);   // PT-3H-15M-15S
```

Note the exception thrown in the last statement below because a `LocalDateTime` object *cannot* be derived from a `LocalTime` object, whereas the converse is true.

Click here to view code image

```
LocalDateTime dateTime = LocalDateTime.of(2021, 4, 28,
                                          17, 45, 15);    // 2021-04-28T17:45:15
Duration interval3 = Duration.between(startTime, dateTime);  // PT3H15M15S
Duration interval4 = Duration.between(dateTime, startTime);  // DateTimeException!
```

The `Duration` class also provides the `parse()` static method to create a duration from a text representation of a duration based on the ISO standard. If the format of the string is not correct, a `DateTimeParseException` is thrown. Formatting according to the `toString()` method is shown in parentheses.

Click here to view code image

```
Duration da = Duration.parse("PT3H15M10.1S");// 3hrs. 15mins. 10.1s.(PT3H15M10.1S)
Duration db = Duration.parse("PT0.999S");    // 999000000 nanos.    (PT0.999S)
Duration dc = Duration.parse("-PT30S");      // -30 seconds.        (PT-30S)
```

```
Duration dd = Duration.parse("P-24D");       // -24 days           (PT-576H)
Duration dd = Duration.parse("P24H");        // Missing T. DateTimeParseException!
```

```
static Duration ZERO
```

This constant defines a `Duration` of length zero (`PT0S`).

Click here to view code image

```
static Duration ofDays(long days)
static Duration ofHours(long hours)
static Duration ofMinutes(long minutes)
static Duration ofMillis(long millis)
static Duration ofSeconds(long seconds)
static Duration ofSeconds(long seconds, long nanoAdjustment)
static Duration ofNanos(long nanos)
```

These static factory methods return a `Duration` representing an amount of time in seconds and nanoseconds that is equivalent to the specified amount, depending on the method. Nanoseconds are implicitly set to zero. The argument value can be negative. Standard definitions of the units are used. Note that the amount is specified as a `long` value.

Click here to view code image

```
static Duration of(long amount, TemporalUnit unit)
```

This static factory method returns a `Duration` representing an amount of time in seconds and nanoseconds that is equivalent to the specified amount in the specified temporal unit. The amount is specified as a `long` value, which can be negative.

Valid `ChronoUnit` constants to qualify the amount specified in the method call are the following: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS` (p. 1044). These units have a standard or an estimated duration.

Click here to view code image

```
static Duration between(Temporal startInclusive, Temporal endExclusive)
```

This static method returns the duration between two temporal objects that must support the seconds unit and where it is possible to convert the second temporal argument to the first temporal argument type, if necessary. Otherwise, a `DateTimeException` is thrown. The result of this method can be a negative period if the end temporal is before the start temporal.

```
String toString()
```

Returns a text representation of a `Duration` according to the ISO standard: `PT`*h*`H`*m*`M`*d.d*`S`—that is, *h* Hours, *m* Minutes, and *d.d* Seconds, where the nanoseconds are formatted as a fraction of a second.

Click here to view code image

```
static Duration parse(CharSequence text)
```

This static method returns a `Duration` parsed from a character sequence. The formats accepted are based on the ISO duration format `PTnHnMn.nS`—for example, `"PT2H3M4.5S"` (2 hours, 3 minutes, and 4.5 seconds). A `java.time.format.Date-TimeParseException` is thrown if the text cannot be parsed to a duration.

Accessing Time Units in a Duration

The `Duration` class provides the `get`*UNIT*`()` methods to read the *individual* values of its time units. The class also has methods to check if the period has a negative value or if its value is zero.

Click here to view code image

```
Duration dx = Duration.ofSeconds(12L*60*60, 500_000_000L); // PT12H0.5S
out.println(dx.getNano());                                 // 500000000
out.println(dx.getSeconds());                              // 43200 (i.e. 12 hrs.)
```

Reading the *individual values* of time units of a `Duration` object can also be done using the `get(unit)` method, where only the `NANOS` and `SECONDS` units are allowed. A list of temporal units that are accepted by the `get(unit)` method can be obtained by calling the `getUnits()` of the `Duration` class.

Click here to view code image

```
out.println(dx.get(ChronoUnit.NANOS));       // 500000000
out.println(dx.get(ChronoUnit.SECONDS));     // 43200
out.println(dx.get(ChronoUnit.MINUTES));     // UnsupportedTemporalTypeException
out.println(dx.getUnits());                  // [Seconds, Nanos]
```

The class `Duration` provides the method `to`*UNIT*`()` to derive the *total length* of the duration in the unit designated by the method name. The seconds and the nanoseconds are converted to this unit, if necessary.

Click here to view code image

```
out.println("Days:    " + dx.toDays());      // Days:    0
out.println("Hours:   " + dx.toHours());     // Hours:   12
out.println("Minutes: " + dx.toMinutes());   // Minutes: 720
out.println("Millis:  " + dx.toMillis());    // Millis:  43200500
out.println("Nanos:   " + dx.toNanos());     // Nanos:   43200500000000
```

```
int getNano()
long getSeconds()
```

Return the number of nanoseconds and seconds in this duration, respectively—*not* the total length of the duration. Note that the first method name is `getNano`, without the `s`.

```
long get(TemporalUnit unit)
```

Returns the value of the specified `unit` in this `Duration`—*not* the total length of the duration. The only supported `ChronoUnit` constants are `NANOS` and `SECONDS` (**p. 1044**). Other units result in an `UnsupportedTemporalTypeException`.

```
List<TemporalUnit> getUnits()
```

Returns the list of time units supported by this duration: `NANOS` and `SECONDS` (**p. 1044**). These time units can be used with the `get(unit)` method.

```
long toDays()
long toHours()
long toMinutes()
long toMillis()
long toNanos()
```

Return the *total* length of this duration, converted to the unit designated by the method, if necessary. Note that there is no `toSeconds()` method. Also, the method name is `toNanos`—note the `s` at the end.

The methods `toMillis()` and `toNanos()` throw an `ArithmeticException` in case of number overflow.

```
boolean isNegative()
```

Determines whether the total length of this duration is negative.

```
boolean isZero()
```

Determines whether the total length of this duration is zero.

Comparing Durations

The `Duration` class overrides the `equals()` method and the `hashCode()` method of the `Object` class, and implements the `Comparable<Duration>` interface. Durations can readily be used in collections. The code below illustrates comparing durations.

Click here to view code image

```
Duration eatBreakFast = Duration.ofMinutes(20L);                // PT20M
Duration eatLunch     = Duration.ofSeconds(30L*60);             // PT30M
Duration eatSupper    = Duration.of(45L, ChronoUnit.MINUTES);   // PT45M

out.println(eatBreakFast.equals(eatLunch));                     // false
out.println(Duration.ofSeconds(0).equals(Duration.ZERO));       // true

List<Duration> ld = Arrays.asList(eatSupper, eatBreakFast, eatLunch );
Collections.sort(ld);                                // Natural order.
out.println(ld);                                     // [PT20M, PT30M, PT45M]
```

Click here to view code image

```
boolean equals(Object otherDuration)
```

Determines whether the *total length* of this duration is equal to the total length of the other duration.

```
int hashCode()
```

Returns a hash code for this duration.

Click here to view code image

```
int compareTo(Duration otherDuration)
```

Compares the *total length* of this duration to the total length of the other duration.

Creating Modified Copies of Durations

The `Duration` class provides `with`*`UNIT`*`()` methods to set a new value for each time unit individually, while the value of the other time unit is retained. Note that each method call returns a new `Duration` object, and chaining method calls works as expected.

Click here to view code image

```
Duration oneDuration =  Duration.ZERO                    // PT0S
                                .withNanos(500_000_000)   // New copy: PT0.5S
                                .withSeconds(12L*60*60);  // New copy: PT12H0.5S
```

Click here to view code image

```
Duration withNanos(int nanoOfSecond)
Duration withSeconds(long seconds)
```

Return a copy of this duration where either the nanosecond or the seconds are set to the value of the argument, respectively. The value of the other time unit is retained.

Temporal Arithmetic with Durations

The `Duration` class provides *plus* and *minus* methods that return a copy of the original object that has been *incremented or decremented by a specific amount* specified in terms of a unit—for example, as a number of days, hours, minutes, or seconds.

Click here to view code image

```
Duration max20H = Duration.ZERO                          // PT0S
                          .plusHours(10)                 // PT10H
                          .plusMinutes(10*60 + 30)       // PT20H30M
                          .plusSeconds(6*60*60 + 15)     // PT26H30M15S
                          .minusMinutes(2*60 + 30)       // PT24H15S
                          .minusSeconds(15);             // PT24H
```

The `plus()` and the `minus()` methods also allow the amount to be qualified by a unit that has a standard or an estimated duration, as illustrated by the statement below, which is equivalent to the one above.

Click here to view code image

```
Duration max20H2 =
    Duration.ZERO                                        // PT0S
            .plus(10L,           ChronoUnit.HOURS)       // PT10H
            .plus(10*60 + 30,    ChronoUnit.MINUTES)     // PT20H30M
            .plus(6*60*60L + 15, ChronoUnit.SECONDS)     // PT26H3015S
            .minus(2*60 + 30,    ChronoUnit.MINUTES)     // PT24H15S
            .minus(15,           ChronoUnit.SECONDS);    // PT24H
```

The code below shows the `plus()` and the `minus()` methods of the `Duration` class that take a `Duration` as the amount to add or subtract.

Click here to view code image

```
Duration eatBreakFast = Duration.ofMinutes(20L);             // PT20M
Duration eatLunch     = Duration.ofSeconds(30L*60);          // PT30M
Duration eatSupper    = Duration.of(45L, ChronoUnit.MINUTES); // PT45M
Duration totalTimeForMeals = eatBreakFast                     // PT20M
    .plus(eatLunch)                                           // PT50M
    .plus(eatSupper);                                         // PT1H35M
```

The statement below shows other arithmetic operations on durations and how they are carried out, together with what would be printed if the intermediate results were also written out.

Click here to view code image

```
Duration result = Duration.ofSeconds(-100, -500_000_000) // -100.5 => PT-1M-40.5S
                          .abs()           // abs(-100.5) = 100.5  => PT1M40.5S
                          .multipliedBy(4) // 100.5*4 = 402 => PT6M42S
                          .dividedBy(2);   // 402 / 2 = 201 => PT3M21S
```

Click here to view code image

```
Duration plusDays/minusDays(long days)
Duration plusHours/minusHours(long hours)
Duration plusMinutes/minusMinutes(long minutes)
Duration plusSeconds/minusSeconds(long seconds)
Duration plusMillis/minusMillis(long millis)
Duration plusNanos/minusNanos(long nanos)
```

Return a copy of this duration, with the specified value of the unit designated by the method name added or subtracted, but converted first to seconds, if necessary. Note that the argument type is `long`.

Click here to view code image

```
Duration plus(long amountToAdd, TemporalUnit unit)
Duration minus(long amountToSub, TemporalUnit unit)
```

Return a copy of this duration with the specified amount added or subtracted, respectively, according to the `TemporalUnit` specified (p. 1044).

Valid `ChronoUnit` constants to qualify the amount specified in the method call are the following: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS` (p. 1044). These units have a standard or an estimated duration. Other units result in an `UnsupportedTemporalTypeException`.

Click here to view code image

```
Duration plus(Duration duration)
Duration minus(Duration duration)
```

Return a copy of this duration, with the specified duration added or subtracted.

```
Duration abs()
```

Returns a copy of this duration with a positive length.

```
Duration negated()
```

Returns a copy of this duration where the length has been negated.

Click here to view code image

```
Duration dividedBy(long divisor)
Duration multipliedBy(long multiplicand)
```

The first method returns a new instance with the result of dividing the length of this duration by the specified `divisor`. Division by zero would bring down untold calamities.

The second method returns a new instance with the result of multiplying the length of this duration by the specified `multiplicand`.

We can perform arithmetic operations on durations and temporal objects. The following code uses the `plus()` and `minus()` methods of the `LocalTime` and `LocalDateTime` classes that take a `TemporalAmount` as an argument (p. 1040). We can add and subtract durations from `LocalTime` and `LocalDateTime` objects, but not from `LocalDate` objects, as a `LocalDate` object only supports date units.

Click here to view code image

```
LocalTime timeA = LocalTime.of(14,45,30);                  // 14:45:30
LocalDate dateA = LocalDate.of(2021, 4, 28);               // 2021-04-28
LocalDateTime dateTimeA = LocalDateTime.of(dateA, timeA); // 2021-04-28T14:45:30

Duration amount = Duration.ofMinutes(20);                  // PT20M

timeA = timeA.plus(amount);                                // 15:05:30
dateTimeA = dateTimeA.minus(amount);                       // 2021-04-28T14:25:30

dateA = dateA.minus(amount);                  // UnsupportedTemporalTypeException
```

Example 17.7 illustrates implementing duration-based loops. The program prints the show-times, given when the first show starts, the duration of the show, and when the theatre closes. The `for(;;)` loop at (1) uses the `LocalTime.isBefore()` method and the `LocalTime.plus(duration)` method to calculate the showtimes.

Example 17.7 *Duration-Based Loop*

Click here to view code image

```
import java.time.LocalTime;
import java.time.Duration;

public class DurationBasedLoop {
  public static void main(String[] args) {
    Duration duration = Duration.ofHours(2).plusMinutes(15);      // PT2H15M
    LocalTime firstShowTime = LocalTime.of(10, 10);               // 10:10
    LocalTime endTimeExclusive = LocalTime.of(23, 0);             // 23:00
    for (LocalTime time = firstShowTime;                          // (1)
         time.plus(duration).isBefore(endTimeExclusive);
         time = time.plus(duration)) {
      System.out.println("Showtime (" + duration + "): " + time);
    }
    System.out.println("Closing time: " + endTimeExclusive);
  }
}
```

Output from the program:

```
Showtime (PT2H15M): 10:10
Showtime (PT2H15M): 12:25
Showtime (PT2H15M): 14:40
Showtime (PT2H15M): 16:55
Showtime (PT2H15M): 19:10
Closing time: 23:00
```

Differences between Periods and Durations

Table 17.4 summarizes the differences between selected methods of the `Period` and the `Duration` classes, mainly in regard to the temporal units supported, representation for parsing and formatting, and comparison. *N/A* stands for *Not Applicable*.

Table 17.4 *Some Differences between the* `Period` *Class and the* `Duration` *Class*

Methods	The `Period` class	The `Duration` class
`of(amount, unit)`	*N/A*	Valid `ChronoUnit`s: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`,

Methods	The `Period` class	The `Duration` class
		`MINUTES`, `HOURS`, `HALF_DAYS`, `DAYS` (p. 1065).
`parse(text)` `toString()`	Representation based on: `PnYnMnD` and `PnW` (p. 1057).	Representation based on: `PnDTnHnMn.nS` (p. 1065).
`get(unit)`	Supported `ChronoUnit`s: `YEARS`, `MONTHS`, `DAYS` (p. 1059).	Supported `ChronoUnit`s: `NANOS`, `SECONDS` (p. 1067).
`getUnits()`	Supported `ChronoUnit`s: `YEARS`, `MONTHS`, `DAYS` (p. 1059).	Supported `ChronoUnit`s: `NANOS`, `SECONDS` (p. 1067).
`equals(other)`	Based on values of individual units (p. 1059).	Based on total length (p. 1067).
`compareTo(other)`	*N/A*	Natural order: total length (p. 1067).
`minus(amount, unit)` `plus(amount, unit)`	*N/A*	Valid `ChronoUnit`s: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, `DAYS` (p. 1069).
`abs()`	*N/A*	Returns copy with positive length (p. 1069).
`dividedBy(divisor)`	*N/A*	Returns copy after dividing by divisor (p. 1069).
`normalized()`	Only years and months normalized (p. 1061).	*N/A*

17.7 Working with Time Zones and Daylight Savings

The following three classes in the `java.time` package are important when dealing with date and time in different time zones and daylight saving hours: `ZoneId`, `ZoneOffset`, and `ZonedDateTime`.

UTC (*Coordinated Universal Time*) is the primary *time standard* used for keeping time around the world. *UTC/Greenwich* is the time at Royal Observatory, Greenwich, England. It is the basis for defining time in different regions of the world.

Time Zones and Zone Offsets

A *time zone* defines a region that observes the same standard time. The time observed in a region is usually referred to as *the local time*. A time zone is described by a *zone offset from UTC/Greenwich* and any *rules* for applying daylight saving time (DST). Time zones that practice DST obviously have variable offsets during the year to account for DST.

In Java, each time zone has a zone ID that is represented by the class `java.time.ZoneId`. The class `java.time.ZoneOffset`, which extends the `ZoneId` class, represents a zone offset from UTC/Greenwich. For example, the time zone with `US/Eastern` as the zone ID has the offset `-04:00` during daylight saving hours—that is, it is 4 hours behind UTC/Greenwich when DST is in effect.

The time zone offset at UTC/Greenwich is represented by `ZoneOffset.UTC` and, by convention, is designated by the letter `Z`. GMT (*Greenwich Mean Time*) has zero offset from UTC/Greenwich (`UTC+0`), thus the two are often used as synonyms; for example, `GMT-4` is equivalent to `UTC-4`. However, GMT is a time zone, whereas UTC is a time standard.

Java uses the IANA Time Zone Database (TZDB) maintained by the Internet Assigned Numbers Authority (IANA) that updates the database regularly, in particular, regarding changes to the rules for DST practiced by a time zone (p. 1073).

A set with names of available time zones can readily be obtained by calling the `ZoneId.getAvailableZoneIds()` method. Time zones have unique names of the form *`Area/Location`*—for example, `US/Eastern`, `Europe/Oslo`. The following code prints a very long list of time zones that are available to a Java application.

Click here to view code image

```
ZoneId.getAvailableZoneIds()                 // Prints a long list of zone names.
      .stream()
      .sorted()
      .forEachOrdered(System.out::println);  // Output not shown intentionally.
```

The `ZoneId.of()` method creates an appropriate zone ID depending on the format of its argument:

- *UTC-equivalent ID*, if only `"Z"`, `"UTC"`, or `"GMT"` is specified. As these designations are equivalent, the result is `ZoneOffset.UTC`; that is, it represents the offset `UTC+0`.
- *Offset-based ID*, if the format is `"+hh:mm"` or `"-hh:mm"`—for example, `"+04:00"`, `"-11:30"`. The result is an instance of the `ZoneOffset` class with the parsed offset.
- *Prefix offset-based ID*, if the format is the prefix `UTC` or `GMT` followed by a numerical offset—for example, `"UTC+04:00"`, `"GMT-04:00"`. The result is a time zone represented by a `ZoneId` with the specified prefix and a parsed offset.

- *Region-based ID*, if the format is "*Area/Location*"—for example, `"US/Eastern"`, `"Europe/Oslo"`. The result is a `ZoneId` that can be used, among other things, to look up the underlying zone rules associated with the zone ID. In the examples in this section, a zone ID is specified in the format of a region-based ID.

The code below creates a region-based zone ID. The method `ZoneId.systemDefault()` returns the system default zone ID.

Click here to view code image

```
ZoneId estZID = ZoneId.of("US/Eastern");            // Create a time zone ID.
System.out.println(estZID);                         // US/Eastern
System.out.println(ZoneId.systemDefault());         // Europe/Oslo
```

Selected methods in the `ZoneId` abstract class are presented below. The concrete class `ZoneOffset` extends the `ZoneId` class.

Click here to view code image

```
static ZoneId of(String zoneId)
```

Returns an appropriate zone ID depending on the format of the zone ID string. See the previous discussion on zone ID.

Click here to view code image

```
String          toString()
abstract String getId()
```

Return a string with the zone ID, typically in one of the formats accepted by the `of()` method.

```
abstract ZoneRules getRules()
```

Retrieves the associated time zone rules for this zone ID. The rules determine the functionality associated with a time zone, such as daylight savings (p. 1082).

Click here to view code image

```
static Set<String> getAvailableZoneIds()
```

Returns a set with the available time zone IDs.

```
static ZoneId systemDefault()
```

Returns the zone ID of the default time zone of the system.

The `ZonedDateTime` Class

The `ZonedDateTime` class represents a date-time with time zone information, and an instance of the class is referred to as a *zoned date-time* (**Table 17.1**, **p. 1025**). The date and time fields are stored with nanosecond precision. The time zone information is represented by a time zone with a zone offset from UTC/Greenwich. In essence, a zoned date-time represents an instant on a specific timeline determined by its zone ID and its zone offset from UTC/Greenwich. This timeline is referred to as the *local timeline*, whereas the timeline at UTC/Greenwich is referred to as the *instant timeline*.

A majority of the methods in the `ZonedDateTime` class should be familiar from the temporal classes discussed earlier in this chapter (**p. 1027**). Both **Table 17.2**, **p. 1026**, and **Table 17.3**, **p. 1026**, showing the common method name prefixes and common methods, apply to the `ZonedDateTime` class as well. The `LocalDateTime` class is the closest equivalent of the `ZonedDateTime` class.

Issues relating to daylight savings are covered later in this section (**p. 1082**). In particular, the methods `of()`, `with()`, `minus()`, and `plus()` take into consideration daylight savings.

The methods `get()`, `with()`, `plus()`, `minus()`, and `until()` throw an `UnsupportedTemporalTypeException` when an unsupported field is accessed, and a `DateTimeException` when the operation cannot be performed for any other reason.

Objects of the `ZonedDateTime` class are also immutable and thread-safe. The class also overrides the `equals()` and the `hashCode()` methods of the `Object` class, but in contrast to the `LocalDateTime` class, its objects are *not* comparable.

With that overview, we turn our attention to dealing with zoned date-times as represented by the `ZonedDateTime` class.

Creating Zoned Date-Time

Analogous to the other temporal classes, the current zoned date-time can be obtained from the system clock in either the default time zone or a specific time zone by calling the `now()` method of the `ZonedDateTime` class. The zoned date-times created at (1a), (2a), and (3a) represent the same date-time locally in the default time zone `Europe/Oslo`, at UTC/Greenwich (`UTC`), and in the `US/Eastern` time zone, respectively.

The text representation of the zoned date-times from (1a), (2a), and (3a) is shown below at (1b), (2b), and (3b), respectively. The text representation of a zoned date-time returned by the `toString()` method shows the date and the time separated by the letter `T`, followed by the zone offset and the time zone.

Click here to view code image

```
ZonedDateTime defaultZDT = ZonedDateTime.now();                          // (1a)
ZonedDateTime utcZDT     = ZonedDateTime.now(ZoneId.of("UTC"));          // (2a)
ZonedDateTime edtZDT     = ZonedDateTime.now(ZoneId.of("US/Eastern"));   // (3a)
System.out.println("Default Zone Date-time:  " + defaultZDT);
System.out.println("UTC Date-time:           " + utcZDT);
System.out.println("EDT Zone Date-time:      " + edtZDT);
```

Output from the print statements:

Click here to view code image

```
Default Zone Date-time: 2021-07-11T11:35:20.008+02:00[Europe/Oslo]       // (1b)
UTC Date-time:          2021-07-11T09:35:20.023Z[UTC]                    // (2b)
EDT Zone Date-time:     2021-07-11T05:35:20.023-04:00[US/Eastern]        // (3b)
```

The local time in the `Europe/Oslo` time zone has the zone offset `+02:00`—meaning it is 2 hours *ahead* of UTC/Greenwich, as it is east of Greenwich. The date-time at UTC/Greenwich (`UTC`) has no zone offset, just the letter `Z` instead of an offset. The local time in the `US/Eastern` time zone has the zone offset `-04:00`—meaning it is 4 hours *behind* UTC/Greenwich, as it is west of Greenwich. On this particular date, the time difference between the `Europe/Oslo` time zone and the `US/Eastern` time zone is 6 hours, where daylight saving time is in effect in both time zones (p. 1082).

One convenient way to create a zoned date-time is to assemble it from its constituent parts. We will use the following declarations to create zoned date-times.

Click here to view code image

```
LocalTime concertTime = LocalTime.of(00, 10);                     // 00:10
LocalDate concertDate = LocalDate.of(1973, Month.JANUARY, 14); // 1973-01-14
LocalDateTime concertDT = LocalDateTime.of(concertDate,
                                           concertTime);  // 1973-01-14T00:10
ZoneId hwZID = ZoneId.of("US/Hawaii");                    // US/Hawaii
Instant instantZ = Instant.ofEpochSecond(95854200);       // 1973-01-14T10:10:00Z
```

The code below creates three zoned date-times from constituent parts. The arguments in the method call comprise the parts of a zoned date-time in each case. We note that the zone offset of the `US/Hawaii` time zone is `-10:00`—that is, 10 hours behind UTC/Greenwich. Note how the `ofInstant()` method converts the time in the instant to the correct local time in the specified time zone.

Click here to view code image

```
ZonedDateTime concertZDT0 = ZonedDateTime.of(concertDate, concertTime, hwZID);
ZonedDateTime concertZDT1 = ZonedDateTime.of(concertDT, hwZID);
ZonedDateTime concertZDT2 = ZonedDateTime.ofInstant(instantZ, hwZID);
// 1973-01-14T00:10-10:00[US/Hawaii]
```

```
boolean areEqual = concertZDT0.equals(concertZDT1)
                && concertZDT0.equals(concertZDT2);      // true
```

The `LocalDateTime` and the `Instant` classes also provide the `atZone()` method that combines a date-time or an instant, respectively, with a time zone to create a zoned date-time. The two zoned date-times created below are equal to the three created earlier.

Click here to view code image

```
ZonedDateTime concertZDT3 = concertDT.atZone(hwZID);
ZonedDateTime concertZDT4 = instantZ.atZone(hwZID);
// 1973-01-14T00:10-10:00[US/Hawaii]
```

We can also use the `parse()` method to create a zoned date-time from a text string that is compatible with the ISO format (p. 1129).

Click here to view code image

```
ZonedDateTime concertZDT5
    = ZonedDateTime.parse("1973-01-14T00:10-10:00[US/Hawaii]");
```

Click here to view code image

```
static ZonedDateTime now()
static ZonedDateTime now(ZoneId zone)
```

Return a `ZonedDateTime` containing the current date-time from the system clock in the default time zone or in the specified time zone, respectively.

Click here to view code image

```
static ZonedDateTime of(int year, int month, int dayOfMonth, int hour,
                       int minute, int second, int nanoOfSecond, ZoneId zone)
static ZonedDateTime of(LocalDate date, LocalTime time, ZoneId zone)
static ZonedDateTime of(LocalDateTime localDateTime, ZoneId zone)
static ZonedDateTime ofInstant(Instant instant, ZoneId zone)
```

Return a `ZonedDateTime` formed from the specified arguments. Note the `zone` argument that is required. The methods throw a `DateTimeException` when the operation cannot be performed for some reason.

Note that these methods take into consideration any adjustments because of daylight savings (p. 1082).

Click here to view code image

```
static ZonedDateTime parse(CharSequence text)
```

Returns a `ZonedDateTime` by parsing the text string according to `DateTimeFormatter.ISO_ZONED_DATE_TIME` (p. 1129)—for example, `"2021-07-03T16:15:30+01:00 [Europe/Oslo]"`.

```
String toString()
```

Returns a string comprising the text representation of the constituent parts: the `LocalDateTime`, typically followed by the zone offset and the time zone—for example, `2021-07-11T05:35:20.023-04:00[US/Eastern]`.

Accessing Fields of Zoned Date-Time

The `ZonedDateTime` class provides many `get` methods to access the values of various fields of a zoned date-time. The most versatile is the `get(field)` method, which supports all the constants defined by the `ChronoField` enum type (p. 1046). In addition, there are get methods for specific fields, and methods for extracting the different constituent parts of a zoned date-time.

Click here to view code image

```
// Using ChronoField constants:
int theDay = concertZDT0.get(ChronoField.DAY_OF_MONTH);          // 14
int theMonthValue = concertZDT0.get(ChronoField.MONTH_OF_YEAR);  // 1
int theYear = concertZDT0.get(ChronoField.YEAR);                 // 1973

// Using specific get methods:
int theMonthValue2 = concertZDT0.getMonthValue();                // 1
Month theMonth     = concertZDT0.getMonth();                     // JANUARY

// Extracting constituent parts:
LocalTime theTime     = concertZDT0.toLocalTime();         // 00:10
LocalDate theDate     = concertZDT0.toLocalDate();         // 1973-01-14
LocalDateTime theDT   = concertZDT0.toLocalDateTime();     // 1973-01-14T00:10
ZoneId theZID         = concertZDT0.getZone();             // US/Hawaii
ZoneOffset theZoffset = concertZDT0.getOffset();           // -10:00
```

```
int getFIELD()
```

Gets the value of the field designated by the suffix *`FIELD`*, which can be `DayOf-Month`, `DayOfYear`, `MonthValue`, `Nano`, `Second`, `Minute`, `Hour`, or `Year`.

Click here to view code image

```
DayOfWeek getDayOfWeek()
Month     getMonth()
```

Get the value of the day-of-week and month-of-year field, respectively.

```
ZoneId     getZone()
ZoneOffset getOffset()
```

Gets the time zone ID (e.g., `US/Central`) and the time zone offset from UTC/ Greenwich (e.g., `-04:00`), respectively (p. 1073).

Click here to view code image

```
int  get(TemporalField field)
long getLong(TemporalField field)
boolean isSupported(TemporalField field)
```

The first two methods return the value of the specified `TemporalField` (p. 1046) as an `int` value or as a `long` value, respectively. The value of the `ChronoField` enum constants `NANO_OF_DAY`, `MICRO_OF_DAY`, `EPOCH_DAY`, `PROLEPTIC_MONTH`, and `INSTANT_SECONDS` will not fit into an `int`, and therefore, the `getLong()` method must be used.

The third method checks if the specified field is supported by this zoned date-time. All `ChronoField` enum constants are supported by the `ZonedDateTime` class (p. 1046).

Click here to view code image

```
LocalTime     toLocalTime()
LocalDate     toLocalDate()
LocalDateTime toLocalDateTime()
```

Return the respective part of this zoned date-time.

```
Instant toInstant()
```

Converts a zoned date-time to an instant representing the same point as this date-time. This method is inherited by the `ZonedDateTime` class from its super-interface `java.time.chrono.ChronoZonedDateTime`.

Conversion of a zoned date-time to an instance of the `java.util.Date` legacy class can be done via this method (p. 1088).

Creating Modified Copies of Zoned Date-Time

Individual fields of a zoned date-time can be set to a new value in a copy of a zoned date-time as illustrated by the following code.

Click here to view code image

```
ZonedDateTime theZDT = concertZDT0         // 1973-01-14T00:10-10:00[US/Hawaii]
    .withYear(1977)                        // 1977-01-14T00:10-10:00[US/Hawaii]
    .with(ChronoField.MONTH_OF_YEAR, 8)    // 1977-08-14T00:10-10:00[US/Hawaii]
    .withDayOfMonth(16)                    // 1977-08-16T00:10-10:00[US/Hawaii]
    .with(ChronoField.HOUR_OF_DAY, 9)      // 1977-08-16T09:10-10:00[US/Hawaii]
    .with(ChronoField.MINUTE_OF_HOUR, 30); // 1977-08-16T09:30-10:00[US/Hawaii]
```

The `with`*`Field`*`()` methods behave analogous to the `with()` method with a corresponding field argument, taking into consideration the time gap and the time overlap that occur when daylight saving time starts and ends (p. 1082).

The `withZoneSameLocal()` method can be used to change the time zone, while retaining the date-time. The code at (1) below changes the time zone from `US/Hawaii` to `US/Central`, while retaining the date-time. Note that the two zoned date-times, `theZDT` and `zdtSameLocal`, do *not* represent the same instant according to UTC/ Greenwich, as shown by the output from the print statements.

Click here to view code image

```
ZoneId cTZ = ZoneId.of("US/Central");
ZonedDateTime zdtSameLocal = theZDT.withZoneSameLocal(cTZ);            // (1)
System.out.printf("%23s %25s%n", "ZonedDateTime", "Instant");
System.out.printf("%-35s %s%n", theZDT, theZDT.toInstant());
System.out.printf("%-35s %s%n", zdtSameLocal, zdtSameLocal.toInstant());
```

Output from the print statements:

Click here to view code image

```
          ZonedDateTime                      Instant
1977-08-16T09:30-10:00[US/Hawaii]   1977-08-16T19:30:00Z
1977-08-16T09:30-05:00[US/Central]  1977-08-16T14:30:00Z
```

The local time `09:30` in the `US/Hawaii` time zone is 10 hours (offset `-10:00`) behind UTC/Greenwich, whereas the same local time in the `US/Central` time zone (offset `-05:00`) is 5 hours behind UTC/Greenwich on the date `1977-08-16`, resulting in different instants at UTC/Greenwich for the same local time.

The `withZoneSameInstant()` method can be used to change the time zone *and* adjust the date-time to the new time zone. We see in the code below at (2) that the adjusted local time

`14:30` in the `US/Central` time zone is 5 hours behind UTC/Greenwich, resulting in both zoned date-times representing the same instant at UTC/Greenwich.

Click here to view code image

```
ZonedDateTime zdtSameInstant = theZDT.withZoneSameInstant(cTZ);        // (2)
System.out.printf("%23s %25s%n", "ZonedDateTime", "Instant");
System.out.printf("%-35s %s%n", theZDT, theZDT.toInstant());
System.out.printf("%-35s %s%n", zdtSameInstant, zdtSameInstant.toInstant());
```

Output from the print statements:

Click here to view code image

```
          ZonedDateTime                    Instant
1977-08-16T09:30-10:00[US/Hawaii]   1977-08-16T19:30:00Z
1977-08-16T14:30-05:00[US/Central]  1977-08-16T19:30:00Z
```

Click here to view code image

```
ZonedDateTime withFIELD(int amount)
```

Returns a copy of this zoned date-time with the field designated by the suffix *`FIELD`* set to the specified value, where the suffix *`FIELD`* can be `DayOfMonth`, `DayOfYear`, `MonthValue`, `Nano`, `Second`, `Minute`, `Hour`, or `Year`.

Note that these methods take into consideration the time gap and the time overlap that can occur due to daylight savings (p. 1082).

Click here to view code image

```
ZonedDateTime with(TemporalField field, long newValue)
```

Returns a copy of this zoned date-time with the specified `TemporalField` set to the specified value. All constants of the `ChronoField` enum type (p. 1046) can be used to specify a particular field.

Click here to view code image

```
ZonedDateTime withZoneSameLocal(ZoneId zone)
ZonedDateTime withZoneSameInstant(ZoneId zone)
```

The first method returns a copy of this zoned date-time with the specified time zone, but normally retaining the date-time.

The second method returns a copy of this zoned date-time with the specified time zone, but retaining the instant; that is, it results in the date-time being adjusted according to the specified time zone.

Temporal Arithmetic with Zoned Date-Time

Temporal arithmetic with zoned date-times is analogous to temporal arithmetic with date-times (p. 1040). However, it is important to note the differences that are due to time zones and daylight savings (p. 1082).

Example 17.8 illustrates calculating flight times. The code at (1) creates a flight departure time for a flight to London from New York at 8:30 pm on July 4, 2021. The flight time is 7 hours and 30 minutes. The code at (2) calculates the local time of arrival at London, first by calling the `withZoneSameInstant()` method to find the local time in London at the time the flight departs from New York, and then adding the flight time to obtain the local arrival time in London.

In Example 17.8, the code at (3) calculates the flight duration by calling the `Duration.between()` method with the departure and arrival times as arguments. The code at (4) calls the `until()` method to calculate the flight time in minutes from the departure time to the arrival time.

Finally, the code at (5) and (6) calculates the local time at the departure airport at the time the flight arrives in London in two different ways. At (5), the `plusMinutes()` method adds the flight time to the departure time. At (6), the `withZoneSameInstant()` method converts the arrival time to its equivalent in the departure time zone.

Example 17.8 *Flight Time Information*

Click here to view code image

```
import java.time.DateTimeException;
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.temporal.ChronoUnit;

public class FlightTimeInfo {
  public static void main(String[] args) {
    try {
      // Departure from New York at 8:30pm on July 4, 2021.                (1)
      LocalDateTime departure = LocalDateTime.of(2021, Month.JULY, 4, 20, 30);
      ZoneId departureZone = ZoneId.of("America/New_York");
      ZonedDateTime departureZDT = ZonedDateTime.of(departure, departureZone);
```

```
    // Flight time is 7 hours and 30 minutes.
    // Calculate local arrival time at London:                      (2)
    ZoneId arrivalZone = ZoneId.of("Europe/London");
    ZonedDateTime arrivalZDT
        = departureZDT.withZoneSameInstant(arrivalZone)
                      .plusMinutes(7*60 + 30);
    System.out.printf("DEPARTURE:  %s%n", departureZDT);
    System.out.printf("ARRIVAL:    %s%n", arrivalZDT);

    // Flight time as a Duration:                                   (3)
    Duration flightduration = Duration.between(departureZDT, arrivalZDT);
    System.out.println("Flight duration:     " + flightduration);

    // Flight time in minutes:                                      (4)
    long flightTime = departureZDT.until(arrivalZDT, ChronoUnit.MINUTES);
    System.out.println("Flight time (mins.): " + flightTime);

    System.out.printf(                                           // (5)
        "Time at departure airport on arrival: %s%n",
        departureZDT.plusMinutes(7*60 + 30));

    System.out.printf(                                           // (6)
        "Time at departure airport on arrival: %s%n",
        arrivalZDT.withZoneSameInstant(departureZone));

  } catch (DateTimeException e) {
    e.printStackTrace();
  }
 }
}
```

Output from the program:

Click here to view code image

```
DEPARTURE:  2021-07-04T20:30-04:00[America/New_York]
ARRIVAL:    2021-07-05T09:00+01:00[Europe/London]
Flight duration:     PT7H30M
Flight time (mins.): 450
Time at departure airport on arrival: 2021-07-05T04:00-04:00[America/New_York]
Time at departure airport on arrival: 2021-07-05T04:00-04:00[America/New_York]
```

Selected methods from the `ZonedDateTime` class for temporal arithmetic are presented below.

Click here to view code image

```
ZonedDateTime plusUNIT(long amount)
ZonedDateTime minusUNIT(long amount)
```

Return a copy of this zoned date-time with the specified `amount` either added or subtracted, respectively, where the unit is designated by the suffix *`UNIT`*, which can be `Nanos`, `Seconds`, `Minutes`, `Hours`, `Days`, `Weeks`, `Months`, or `Years`.

With the time units (`Nanos`, `Seconds`, `Minutes`, `Hours`), the operation is on the instant timeline, where a `Duration` of the specified amount is added or subtracted.

With the date units (`Days`, `Weeks`, `Months`, `Years`), the operation is on the local timeline, adding and subtracting the amount from the date-time corresponding to this zoned date-time.

Click here to view code image

```
ZonedDateTime plus(long amountToAdd, TemporalUnit unit)
ZonedDateTime minus(long amountToSubtract, TemporalUnit unit)
boolean isSupported(TemporalUnit unit)
```

Return a copy of this zoned date-time with the specified amount added or subtracted, respectively, according to the `TemporalUnit` specified. The methods support all `ChronoUnit` enum constants, except `FORVER` (p. 1044).

Time units operate on the instant timeline, but date units operate on the local timeline, when performing these operations.

The `isSupported()` method checks if the specified `TemporalUnit` is supported by this zoned date-time (p. 1044).

Click here to view code image

```
ZonedDateTime plus(TemporalAmount amountToAdd)
ZonedDateTime minus(TemporalAmount amountToSubtract)
```

Return a copy of this zoned date-time with the specified temporal amount added or subtracted, respectively. The classes `Period` (p. 1057) and `Duration` (p. 1064) implement the `TemporalAmount` interface.

If the amount is a `Duration`, the time units operate on the instant timeline. However, if the amount is a `Period`, the date units operate on the local timeline.

Click here to view code image

```
long until(Temporal endExclusive, TemporalUnit unit)
```

Calculates the amount of time between this zoned date-time and the specified temporal object in terms of the specified `TemporalUnit` (p. 1044). The end point is excluded. An exception is thrown if the specified temporal object cannot be converted to a zoned date-time.

This method supports all `ChronoUnit` enum constants, except `FORVER` (p. 1044).

Date units operate on the local timeline, but time units operate on the instant timeline when performing this operation.

Daylight Savings

If a time zone practices daylight savings, the time zone offset of a `ZonedDateTime` can be different depending on whether daylight saving time (DST) or standard time (for the rest of the year) is in effect. Zone rules associated with the zone ID of a zoned date-time are used to determine the right zone offset for time zones that practice daylight savings.

A *time gap* occurs when the clock is moved *forward* at the start of DST, and a *time overlap* occurs when the clock is moved *backward* at the end of DST. Usually the gap and the overlap are 1 hour. In that case, when the time gap occurs, an hour is lost and that day has only 23 hours, whereas when a time overlap occurs, an hour is gained and that day has 25 hours. Care should be exercised when dealing with zoned date-times that can involve crossovers to and from DST. Luckily, the methods `of()`, `with()`, `plus()`, and `minus()` of the `ZonedDateTime` class take daylight savings into consideration.

As an example to illustrate daylight savings, we will use the `US/Central` time zone. For this time zone, DST starts at 02:00:00 on 14 March in 2021, when the clocks are moved forward by 1 hour, resulting in the hour between 02:00:00 and 03:00:00 being lost. The gap in this case is 1 hour. The day on 2021-03-14 will only be 23 hours long.

DST for the `US/Central` time zone ends at 02:00:00 on 7 November in 2021, when clocks are moved backward by 1 hour, resulting in the hour before 02:00:00 being repeated twice. The overlap in this case is 1 hour. The day on 2021-11-07 will be 25 hours long.

The following output from **Example 17.9** illustrates what happens when the `plus-Hours()` method increments a zoned date-time across the time gap.

Click here to view code image

```
Daylight Savings in US/Central starts at 2021-03-14T02:00 (spring forward 1 hour).
                      ___________ZonedDateTime__________
                         Date     Time  Offset    TZ        DST   UTC
(1) Before gap:       2021-03-14 01:30 -06:00 US/Central false 07:30
    + 1 hour
(2) After gap:        2021-03-14 03:30 -05:00 US/Central true  08:30
```

Line (1) above shows the following information about a zoned date-time that is before the gap: `Date` (`2021-03-14`), `Time` (`01:30`), `Offset` (`-06:00`), `TZ` (`US/Central`), `DST` that indicates whether it is in effect (`false`), and `UTC` equivalent of the time (`07:30`). Note that the time `01:30` is before the gap. Adding 1 hour to the zoned date-time in line (1) puts the resulting time (`02:30`) in the gap, which does not exist. The `plusH-ours()` method increments the expected time `02:30` by the gap length to `03:30` and increments the offset `-6:00` by the gap length to `-05:00` to comply with DST. Line (2) shows the final result of the operation. DST is now in effect. The UTC equivalent time is now `08:30`, an hour after `07:30`, as we would expect.

The following output from **Example 17.9** illustrates what happens when the `plus-Hours()` method increments a zoned date-time successively across the time overlap.

Click here to view code image

```
Daylight Savings in US/Central ends at 2021-11-07T02:00 (fall back 1 hour).
                                __________ZonedDateTime__________
                                   Date     Time  Offset    TZ        DST   UTC
(1) Before overlap: 2021-11-07 00:30 -05:00 US/Central true  05:30
    + 1 hour
(2) In overlap:     2021-11-07 01:30 -05:00 US/Central true  06:30
    + 1 hour
(3) In overlap:     2021-11-07 01:30 -06:00 US/Central false 07:30
    + 1 hour
(4) After overlap:  2021-11-07 02:30 -06:00 US/Central false 08:30
```

Line (1) shows a zoned date-time before the overlap with the following information: `Date` (`2021-11-07`), `Time` (`00:30`), `Offset` (`-05:00`), `TZ` (`US/Central`), `DST` that indicates whether it is in effect (`true`), and `UTC` equivalent of the time (`05:30`). Again note that the time `00:30` is before the time overlap. Lines (2), (3), and (4) show the result of successively adding 1 hour to the resulting zoned date-time from the previous operation, starting with the zoned date-time in line (1).

Adding 1 hour to the time `00:30` in line (1) with DST in effect changes the time to `01:30`, which is in the overlap, but before the DST crossover at `02:00`. The result is shown in line (2).

Adding 1 hour to the time `01:30` in line (2) with DST in effect does *not* change the time. The operation only decrements the offset `-05:00` by the overlap length (1 hour) to `-06:00`, as the time `01:30` is still in the overlap after the DST crossover because it is repeated, but now the standard time is in effect. The result is shown in line (3).

Adding 1 hour to the time `01:30` in line (3) with standard time in effect changes the time to `02:30`. The resulting time `02:30` is not in the overlap. The result is shown in line (4). The result would have been the same if we had added 3 hours to the zoned date-time in line (1): the time would be incremented by 2 hours (3 – overlap length) and the offset decremented by the overlap length (1 hour).

The last column, `UTC`, also shows that the time at UTC/Greenwich changed successively by 1 hour as a result of the plus operations.

In **Example 17.9**, the methods `adjustForGap()` at (1a) and `adjustForOverlap()` at (1b) create the scenarios for DST crossovers discussed above. The method `printInfo()` at (7) prints the result of each plus operation. The essential lines of code are (4a) and (4b) that perform the plus operations, with the rest of the code creating zoned date-times (2a, 3a, 2b, 3b) and printing formatted output (5a, 5b).

The method `isDST()` at (8) determines if DST is in effect for a zoned date-time. The method `localTimeAtUTC()` at (9) returns the UTC equivalent of the time in a zoned date-time. These auxiliary methods are simple but instructive examples of operations on zoned date-times.

The zone rules associated with a time zone can be obtained by calling the `ZoneId.getRules()` method on a zone ID. The zone rules are represented by the `java.time.zone.ZoneRules` class that provides the `isDaylightSavings()` method to determine whether an instant is in daylight savings. A zoned date-time can be converted to an instant by the `toInstant()` method.

Click here to view code image

```
// java.time.zone.ZoneRules
boolean isDaylightSavings(Instant instant)
```

Determines whether the specified `instant` is in daylight savings.

Example 17.9 *Adjusting for DST Crossovers*

Click here to view code image

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZoneOffset;
import java.time.ZonedDateTime;

public class DSTAdjustment {
  public static void main(String[] args) {
    adjustForGap();
    adjustForOverlap();
  }

  /**
   * Adjustment due to the time gap at DST crossover.                    (1a)
   * DST starts in US/Central TZ: 2021-03-14T02:00:00,
   * clocks are moved forward 1 hour, resulting in a time gap of 1 hour.
   */
  static void adjustForGap() {
    // Start date and time for DST in US/Central in 2021.                (2a)
    ZoneId cTZ = ZoneId.of("US/Central");
```

```
    LocalDate dateStartDST = LocalDate.of(2021, 3, 14);
    LocalTime timeStartDST = LocalTime.of(2, 0);
    LocalDateTime ldtStartDST = LocalDateTime.of(dateStartDST, timeStartDST);
    ZonedDateTime zdtStartDST = ZonedDateTime.of(ldtStartDST, cTZ);

    // Time before the gap.                                                  (3a)
    LocalTime timeBeforeGap = LocalTime.of(1, 30);
    LocalDateTime ldtBeforeGap = LocalDateTime.of(dateStartDST, timeBeforeGap);
    ZonedDateTime zdtBeforeGap = ZonedDateTime.of(ldtBeforeGap, cTZ);

    // Add 1 hour.                                                           (4a)
    ZonedDateTime zdtAfterGap = zdtBeforeGap.plusHours(1);

    // Print a report.                                                       (5a)
    System.out.printf("Daylight Savings in %s starts at %s "
        + "(spring forward 1 hour).%n", cTZ, ldtStartDST);
    System.out.println("                    ___________ZonedDateTime__________");

    System.out.printf("%27s %7s %7s %5s %9s %5s%n",
                      "Date", "Time", "Offset", "TZ", "DST", "UTC");
    printInfo("(1) Before gap:     ", zdtBeforeGap);
    System.out.println("    + 1 hour");
    printInfo("(2) After gap:      ", zdtAfterGap);
    System.out.println();

    // Add 3 hours:                                                          (6a)
    ZonedDateTime zdtPlus3Hrs = zdtBeforeGap.plusHours(3);
    System.out.printf("%s + 3 hours = %s%n", zdtBeforeGap, zdtPlus3Hrs);
    System.out.println();
  }

  /**
   * Adjustment due to the time overlap at DST crossover.                    (1b)
   * DST ends in US/Central TZ: 2021-11-07T02:00:00,
   * clocks are moved backward 1 hour, resulting in a time overlap of 1 hour.
   */
  static void adjustForOverlap() {
    // End date and time for DST in US/Central in 2021.                      (2b)
    ZoneId cTZ = ZoneId.of("US/Central");
    LocalDate dateEndDST = LocalDate.of(2021, 11, 7);
    LocalTime timeEndDST = LocalTime.of(2, 0);
    LocalDateTime ldtEndDST = LocalDateTime.of(dateEndDST, timeEndDST);
    ZonedDateTime zdtEndDST = ZonedDateTime.of(ldtEndDST, cTZ);

    // Time before the overlap:                                              (3b)
    LocalTime timeBeforeOverlap = LocalTime.of(0, 30);
    LocalDateTime ldtBeforeOverlap = LocalDateTime.of(dateEndDST,
                                                      timeBeforeOverlap);
    ZonedDateTime zdtBeforeOverlap = ZonedDateTime.of(ldtBeforeOverlap, cTZ);

    // Add 1 hour:                                                           (4b)
    ZonedDateTime zdtInOverlap1 = zdtBeforeOverlap.plusHours(1);
    ZonedDateTime zdtInOverlap2 = zdtInOverlap1.plusHours(1);
```

```
    ZonedDateTime zdtAfterOverlap = zdtInOverlap2.plusHours(1);
    // Print a report.                                                     (5b)
    System.out.printf("Daylight Savings in %s ends at %s (fall back 1 hour).%n",
                      cTZ, ldtEndDST);
    System.out.println("                    ___________ZonedDateTime__________");

    System.out.printf("%27s %7s %7s %5s %9s %5s%n",
                      "Date", "Time", "Offset", "TZ", "DST", "UTC");
    printInfo("(1) Before overlap: ", zdtBeforeOverlap);
    System.out.println("    + 1 hour");
    printInfo("(2) In overlap:     ", zdtInOverlap1);
    System.out.println("    + 1 hour");
    printInfo("(3) In overlap:     ", zdtInOverlap2);
    System.out.println("    + 1 hour");
    printInfo("(4) After overlap:  ", zdtAfterOverlap);
    System.out.println();

    // Add 3 hours:                                                        (6b)
    ZonedDateTime zdtPlus3Hrs = zdtBeforeOverlap.plusHours(3);
    System.out.printf("%s + 3 hours = %s%n", zdtBeforeOverlap, zdtPlus3Hrs);
    System.out.println();
  }

  /**
   * Print info for a date-time.                                           (7)
   * @param leadTxt Text to lead the information.
   * @param zdt     Zoned date-time whose info is printed.
   */
  static void printInfo(String leadTxt, ZonedDateTime zdt) {
    System.out.printf(leadTxt + "%10s %5s %6s %5s %-5s %5s%n",
                      zdt.toLocalDate(), zdt.toLocalTime(),
                      zdt.getOffset(), zdt.getZone(),
                      isDST(zdt), localTimeAtUTC(zdt));
  }

  /**
   * Determine if DST is in effect for a zoned date-time.                  (8)
   * @param zdt  Zoned date-time whose DST status should be determined.
   * @return     true, if DST is in effect.
   */
  static boolean isDST(ZonedDateTime zdt) {
    return zdt.getZone().getRules().isDaylightSavings(zdt.toInstant());
  }

  /**
   *  Find local time at UTC/Greenwich equivalent to local time in          (9)
   *  the specified zoned date-time.
   * @param zdt   Zoned date-time to convert to UTC/Greenwich.
   * @return      Equivalent local time at UTC/Greenwich.
   */
  static LocalTime localTimeAtUTC(ZonedDateTime zdt) {
    return zdt.withZoneSameInstant(ZoneOffset.UTC).toLocalTime();
```

```
    }
  }
```

Analogous to the `plus()` methods, the `of()` methods also make adjustments at DST crossings. Sticking to the DST information from **Example 17.9**, if we try to create a zoned date-time with a time that is in the gap, the `of()` method typically moves the time by the length of the gap (1 hour) into DST.

Click here to view code image

```
ZonedDateTime zdt1 = ZonedDateTime.of(
    LocalDate.of(2021, 3, 14),
    LocalTime.of(2, 30),                // Time 02:30 is in the gap.
    ZoneId.of("US/Central"));
System.out.println(zdt1);               // 2021-03-14T03:30-05:00[US/Central]
```

For a time in the overlap, the offset is ambiguous—it can be -05:00 for DST or -06:00 for standard time. Typically, the `of()` methods return a zoned date-time with the DST offset, as shown in the following code:

Click here to view code image

```
ZonedDateTime zdt2 = ZonedDateTime.of(
    LocalDate.of(2021, 11, 7),
    LocalTime.of(1, 30),                // Time 01:30 in the overlap.
    ZoneId.of("US/Central"));
System.out.println(zdt2);               // 2021-11-07T01:30-05:00[US/Central]
```

The `withField()` methods behave analogously to the `to()` methods when it comes to DST crossings. If we try to create a zoned date-time with the time 02:00 that is in the gap, the `withHour()` method typically moves the time by the length of the gap (1 hour) into DST.

Click here to view code image

```
ZonedDateTime zdt3 = ZonedDateTime.of(
    LocalDate.of(2021, 3, 14), LocalTime.of(0, 0), ZoneId.of("US/Central")
  ).withHour(2);                     // Time 02:00 is in the gap.
System.out.println(zdt3);            // 2021-03-14T03:00-05:00[US/Central]
```

For a time in the overlap, the `withMinute()` method returns a zoned date-time with the DST offset, as shown in the following code:

Click here to view code image

```
ZonedDateTime zdt4 = ZonedDateTime.of(
    LocalDate.of(2021, 11, 7), LocalTime.of(1, 0), ZoneId.of("US/Central")
```

```
    ).withMinute(30);                  // Time 01:30 is in the overlap.
  System.out.println(zdt4);            // 2021-11-07T01:30-05:00[US/Central]
```

Finally, the result of temporal arithmetic with zoned date-times can depend on whether date units or time units are involved. We illustrate the behavior using the following zoned date-time that is before the crossover from DST to standard time.

Click here to view code image

```
ZonedDateTime zdtBeforeOverlap = ZonedDateTime.of(
    LocalDate.of(2021, 11, 7),
    LocalTime.of(0, 30),              // Time 00:30 is before the DST crossover.
    ZoneId.of("US/Central"));
```

We add 1 day with the `plusDays()` method and 24 hours with the `plusHours`() method to the zoned date-time using the code below. Number of days is measured by the date unit *days*, and number of hours by the time unit *hours*. From the output we see that the results are different. The day was added to the date-time and converted back to a zoned date-time with the offset adjusted, without affecting the time part; that is, date units operate on the *local* time-line. Adding 24 hours results in a zoned-based time that is exactly a duration of 24 hours later, taking the DST crossover into consideration; that is, time units operate on the *instant* timeline.

Click here to view code image

```
// Date units and time units.
System.out.printf("%s + 1 day    = %s%n",
    zdtBeforeOverlap, zdtBeforeOverlap.plusDays(1));        // (1) Add 1 day.
System.out.printf("%s + 24 hours = %s%n",
    zdtBeforeOverlap, zdtBeforeOverlap.plusHours(24));      // (2) Add 24 hours.
```

Output from the print statements:

Click here to view code image

```
2021-11-07T00:30-05:00[US/Central] + 1 day    = 2019-11-04T00:30-06:00[US/Central]
2021-11-07T00:30-05:00[US/Central] + 24 hours = 2021-11-07T23:30-06:00[US/Central]
```

By the same token, adding a `Period` (that has only date fields) and a `Duration` (that has only time fields) using the `plus(Period.ofDays(1))` and the `plus`(Duration .ofHours(24)) method calls instead at (1) and (2) above, respectively, will give the same results.

17.8 Converting Date and Time Values to Legacy Date

An object of the `java.util.Date` legacy class represents time, date, and time zone. The class provides the method `from()` to convert a `java.time.Instant` (p. 1049) to a `Date`. In order to convert dates and times created using the new Date and Time API, we need to go through an `Instant` to convert them to a `Date` object.

The `java.time.ZonedDateTime` class provides the `toInstant()` method to convert a `ZonedDateTime` object to an `Instant`, which is utilized by the `zdtToDate()` method of the `ConvertToLegacyDate` utility class in **Example 17.10** at (1).

A `java.time.LocalDateTime` object lacks a time zone in order to convert it to a `Date`. The `ldtToDate()` method at (2) adds the system default time zone to create a `Zoned-DateTime` object which is then converted to an `Instant`.

A `LocalDate` object lacks time and a time zone in order to convert it to a `Date`. The `ldTo-Date()` method at (3) adds a fictive time (start of the day), and the resulting `LocalDateTime` object is added the system default time zone to create a `ZonedDateTime` object which is then converted to an `Instant`.

A `LocalTime` object lacks a date and a time zone in order to convert it to a `Date`. The `ltTo-Date()` method at (4) adds a fictive date (2021-1-1), and the resulting `LocalDateTime` object is added to the system default time zone to create a `ZonedDateTime` object which is then converted to an `Instant.`

For an example, see **Example 18.8**, **p. 1143**.

Example 17.10 *Converting to Legacy Date*

Click here to view code image

```
import java.time.*;
import java.util.Date;

public class ConvertToLegacyDate {
  /** Convert a ZonedDateTime to Date. */
  public static Date zdtToDate(ZonedDateTime zdt) {                    // (1)
    return Date.from(zdt.toInstant());
  }

  /** Convert a LocalDateTime to Date. */
  public static Date ldtToDate(LocalDateTime ldt) {                    // (2)
    return Date.from(ldt.atZone(ZoneId.systemDefault()).toInstant());
  }

  /** Convert a LocalDate to Date. */
  public static Date ldToDate(LocalDate ld) {                          // (3)
    return Date.from(ld.atStartOfDay()
                       .atZone(ZoneId.systemDefault())
                       .toInstant());
  }

  /** Convert a LocalTime to Date. */
```

```
  public static Date ltToDate(LocalTime lt) {                              // (4)
    return Date.from(lt.atDate(LocalDate.of(2021, 1, 1))
                       .atZone(ZoneId.systemDefault())
                       .toInstant());
  }
}
```

Review Questions

17.1 Given the following code:

Click here to view code image

```
import java.time.LocalDate;
public class RQ1 {
  public static void main(String[] args) {
    LocalDate d1 = LocalDate.of(2021, 1, 31);
    LocalDate d2 = d1.plusMonths(1);
    LocalDate d3 = d2.minusMonths(1);
    System.out.println(d1.getDayOfYear() + " " + d2.getDayOfYear() + " " +
                       d3.getDayOfYear());
  }
}
```

What is the result?

Select the one correct answer.

a. `31 61 31`

b. `31 59 28`

c. `31 59 31`

d. The program will throw an exception at runtime.

17.2 Given the following code:

Click here to view code image

```
import java.time.LocalDate;
public class RQ2 {
  public static void main(String[] args) {
    LocalDate d1 = LocalDate.of(2021, 1, 1);
    d1 = d1.withDayOfMonth(d1.lengthOfMonth()).withMonth(2);
    System.out.println(d1);
```

```
  }
}
```

What is the result?

Select the one correct answer.

a. `2021-02-28`

b. `2021-02-31`

c. `2021-03-03`

d. The program will throw an exception at runtime.

17.3 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ3 {
  public static void main(String[] args) {
    LocalDateTime d1 = LocalDate.of(2021, 4, 1).atStartOfDay();
    Instant i1 = d1.toInstant(ZoneOffset.of("+18:00"));
    LocalDate d2 = LocalDate.ofInstant(i1, ZoneId.of("UTC"));
    System.out.println(d2);
  }
}
```

What is the result?

Select the one correct answer.

a. `2021-04-1`

b. `2021-04-2`

c. `2021-03-30`

d. `2021-03-31`

17.4 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ4 {
  public static void main(String[] args) {
```

```
    LocalDateTime dt = LocalDate.of(2021, 4, 1).atStartOfDay();
    ZonedDateTime zdt1 = dt.atZone(ZoneId.of("Europe/Paris"));
    ZonedDateTime zdt2 = dt.atZone(ZoneId.of("Europe/London"));
    Duration d = Duration.between(zdt1.minusMinutes(30), zdt2.plusMinutes(30));
    System.out.println(d);
  }
}
```

What is the result, given that the time difference between Paris and London is 1 hour?

Select the one correct answer.

a. `PT0H`

b. `PT1H`

c. `PT2H`

d. `PT-2H`

e. `PT-1H`

17.5 Which statement is false?

Select the one correct answer.

a. `Instant` objects can represent points in time with nanosecond precision.

b. `Instant` objects and `LocalTime` objects have same precision.

c. `Duration` objects can express the amount of time between two `LocalDate` objects.

d. `Period` objects can express the number of days between two `Instant` objects.

17.6 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ6 {
  public static void main(String[] args) {
    LocalTime t = LocalTime.of(8, 15);
    LocalDate d = LocalDate.of(2021, 4, 1);
    LocalDateTime dt = d.atTime(t);
    dt.minusMinutes(30).withDayOfMonth(12);
    System.out.println(dt);
  }
}
```

What is the result?

Select the one correct answer.

a. `2021-04-12T08:45`

b. `2021-04-12T07:45`

c. `2021-04-01T08:15`

d. `2021-04-01T07:15`

17.7 Which statement is true?

Select the one correct answer.

a. Adding a `Duration` and adding a `Period` to a `ZonedDateTime` object produces the same result.

b. A `Duration` of 36,000 seconds is the same as a `Period` of 1 hour.

c. Daylight savings is taken into consideration by `LocalTime` and `LocalDateTime` objects.

d. A `Period` of 1 day is always equivalent to a `Duration` of 24 hours.

e. `Period` and `Duration` objects can have positive and negative values.

17.8 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ8 {
  public static void main(String[] args) {
    Period d  = Period.parse("P2D");
    LocalDate ld = LocalDate.of(2021, 4, 1);
    LocalDateTime ldt = ld.plus(d);
    System.out.println(ldt);
  }
}
```

What is the result?

Select the one correct answer.

a. `2021-04-01T00:30`

b. `2021-04-01T23:30`

c. `2021-03-31T00:30`

d. `2021-03-31T23:30`

e. The program will fail to compile.

17.9 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ9 {
  public static void main(String[] args) {
    Duration d  = Duration.parse("PT-24H");
    LocalDate ld = LocalDate.of(2021, 4, 1).plus(d);
    System.out.println(ld);
  }
}
```

What is the result?

Select the one correct answer.

a. `2021-03-31`

b. `2021-04-01`

c. `2021-04-02`

d. The program will throw an exception at runtime.

e. The program will fail to compile.

17.10 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ10 {
  public static void main(String[] args) {
    LocalDate ld = LocalDate.of(2021, 4, 1);
    // (1) INSERT CODE HERE
    System.out.println(ldt);
  }
}
```

Which of the following statements, when inserted independently at (1), will print the following result: `2021-04-03T00:30`?

Select the two correct answers.

a.

Click here to view code image

```
LocalDateTime ldt = ld.atTime(LocalTime.of(0, 30))
                      .plus(Duration.ofHours(48));
```

b.

Click here to view code image

```
LocalDateTime ldt = ld.plus(Duration.ofHours(48))
                      .atTime(LocalTime.of(0, 30));
```

c.

Click here to view code image

```
LocalDateTime ldt = ld.atTime(LocalTime.of(48,30));
```

d.

Click here to view code image

```
LocalDateTime ldt = ld.plusDays(3).atTime(LocalTime.of(-23,30));
```

e.

Click here to view code image

```
LocalDateTime ldt = ld.atTime(0, 30).plus(Duration.ofHours(48));
```

17.11 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ11 {
  public static void main(String[] args) {
    LocalDate now = LocalDate.now();
    LocalDate foolsDay = LocalDate.of(2021, Month.APRIL, 1);
    LocalDateTime tomorrowAfternoon = now.plusDays(1)
                                         .atTime(LocalTime.of(12, 01));
```

```
      LocalDate anotherDay = foolsDay.withDayOfMonth(2).minusDays(1);
    }
  }
```

How many `LocalDate` objects are created in this example?

Select the one correct answer.

a. `3`

b. `4`

c. `5`

d. `6`

17.12 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ13 {
  public static void main(String[] args) {
    LocalTime lt = LocalTime.of(17,30);
    LocalDateTime ldt = LocalDateTime.of(2021, Month.APRIL, 2, 15, 15);
    Duration d = Duration.between(lt, ldt);
    System.out.println(d);
  }
}
```

What is the result?

Select the one correct answer.

a. `PT-1H-45M`

b. `PT1H-45M`

c. `PT2H-15M`

d. `PT-2H-15M`

e. The program will throw an exception at runtime.

17.13 Given the following code:

Click here to view code image

```
import java.time.*;
import static java.time.temporal.ChronoUnit.DAYS;
public class RQ14 {
  public static void main(String[] args) {
    LocalDateTime ldt = LocalDateTime.of(2021, Month.APRIL, 2, 15, 15);
    // (1) INSERT CODE HERE
    System.out.println(ldt);
  }
}
```

Which statement inserted at (1) will *not* give the following result: `2021-04-03T16:15`?

Select the one correct answer.

a.

Click here to view code image

```
ldt = ldt.plusHours(1).with(LocalDate.of(2021, Month.APRIL, 3));
```

b.

Click here to view code image

```
ldt = ldt.plusDays(1).with(LocalTime.of(16, 15));
```

c.

Click here to view code image

```
ldt = ldt.plus(Duration.of(2, DAYS)).minus(Duration.parse("PT23H"));
```

d.

Click here to view code image

```
ldt = ldt.plus(Duration.of(2, DAYS))
         .minus(Duration.ofMinutes(15).ofHours(16));
```

e.

Click here to view code image

```
ldt = ldt.plus(Duration.parse("PT25H"));
```

f.

[Click here to view code image]

```
ldt = ldt.plus(Duration.ofMinutes(25 * 60));
```