```
        System.out.println( bd1.add( bd2 ) ); // 18446744073709551614

        BigDecimal numerator = new BigDecimal(1);
        BigDecimal denominator = new BigDecimal(3);
        BigDecimal fraction =
            numerator.divide( denominator, 100, BigDecimal.ROUND_UP );
        // 100 digit fraction = 0.333333 ... 3334
    }
    catch (NumberFormatException nfe) { }
    catch (ArithmeticException ae) { }
```

If you implement cryptographic or scientific algorithms for fun, `BigInteger` is crucial. `BigDecimal`, in turn, can be found in applications dealing with currency and financial data. Other than that, you're not likely to need these classes.

Dates and Times

Working with dates and times without the proper tools can be a chore. Prior to Java 8, you had access to three classes that handled most of the work for you. The `java.util.Date` class encapsulates a point in time. The `java.util.GregorianCalendar` class, which extends the abstract `java.util.Calendar`, translates between a point in time and calendar fields like month, day, and year. Finally, the `java.text.DateFormat` class knows how to generate and parse string representations of dates and times in many languages.

While the `Date` and `Calendar` classes covered many use cases, they lacked granularity and were missing other features. This caused the creation of several third-party libraries, all aimed at making it easier for developers to work with dates and times and time durations. Java 8 provided much needed improvements in this area with the addition of the `java.time` package. We will explore this new package, but you will still encounter many, many `Date` and `Calendar` examples in the wild, so it's useful to know they exist. As always, the online docs (*https://oreil.ly/Behlk*) are an invaluable source for reviewing parts of the Java API we don't tackle here.

Local Dates and Times

The `java.time.LocalDate` class represents a date without time information for your local region. Think of a holiday such as May 4, 2019. Similarly, `java.time.LocalTime` represents a time without any date information. Perhaps your alarm clock goes off at 7:15 every morning. The `java.time.LocalDateTime` stores both date and time values for things like appointments with your eye doctor so you can keep reading books on Java. All of these classes offer static methods for creating new instances using either appropriate numeric values with `of()` or by parsing strings with `parse()`. Let's pop into *jshell* and try creating a few examples.

```
jshell> import java.time.*

jshell> LocalDate.of(2019,5,4)
$2 ==> 2019-05-04

jshell> LocalDate.parse("2019-05-04")
$3 ==> 2019-05-04

jshell> LocalTime.of(7,15)
$4 ==> 07:15

jshell> LocalTime.parse("07:15")
$5 ==> 07:15

jshell> LocalDateTime.of(2019,5,4,7,0)
$6 ==> 2019-05-04T07:00

jshell> LocalDateTime.parse("2019-05-04T07:15")
$7 ==> 2019-05-04T07:15
```

Another great static method for creating these objects is `now()`, which provides the current date or time or date-and-time as you might expect:

```
jshell> LocalTime.now()
$8 ==> 15:57:24.052935

jshell> LocalDate.now()
$9 ==> 2019-12-12

jshell> LocalDateTime.now()
$10 ==> 2019-12-12T15:57:37.909038
```

Great! After importing the `java.time` package, we can create instances of each of the `Local...` classes for specific moments or for "right now." You may have noticed the objects created with `now()` include seconds and nanoseconds. You can supply those values to the `of()` and `parse()` methods if you want or need them. Not much exciting there, but once you have these objects, you can do a lot with them. Read on!

Comparing and Manipulating Dates and Times

One of the big advantages of using `java.time` classes is the consistent set of methods you have available for comparing and changing dates and times. For example, many chat applications will show you "how long ago" a message was sent. The `java.time.temporal` subpackage has just what we need: the `ChronoUnit` interface. It contains several date and time units such as `MONTHS`, `DAYS`, `HOURS`, `MINUTES`, etc. These units can be used to calculate differences. For example, we could calculate how long it takes us to create two example date-times in *jshell* using the `between()` method:

```
jshell> LocalDateTime first = LocalDateTime.now()
first ==> 2019-12-12T16:03:21.875196
```

```
jshell> LocalDateTime second = LocalDateTime.now()
second ==> 2019-12-12T16:03:33.175675

jshell> import java.time.temporal.*

jshell> ChronoUnit.SECONDS.between(first, second)
$12 ==> 11
```

A visual spot check shows that it did indeed take about 11 seconds to type in the line that created our `second` variable. You should check out the docs for `ChronoUnit` (*https://oreil.ly/BhCr2*) for a complete list of units available, but you get the full range from nanoseconds up to millennia.

Those units can also help you manipulate dates and times with the `plus()` and `minus()` methods. To set a reminder for one week from today, for example:

```
jshell> LocalDate today = LocalDate.now()
today ==> 2019-12-12

jshell> LocalDate reminder = today.plus(1, ChronoUnit.WEEKS)
reminder ==> 2019-12-19
```

Neat! But this `reminder` example brings up another bit of manipulation you may need to perform from time to time. You might want the reminder at a particular time on the 19th. You can convert between dates or times and date-times easily enough with the `atDate()` or `atTime()` methods:

```
jshell> LocalDateTime betterReminder = reminder.atTime(LocalTime.of(9,0))
betterReminder ==> 2019-12-19T09:00
```

Now we'll get that reminder at 9 A.M. Except, what if we set that reminder in Atlanta and then flew to San Francisco? When would the alarm go off? `LocalDateTime` is, well, local! So the `T09:00` portion is still 9 A.M. wherever we are when we run the program. But if we are handling something like a shared calendar and scheduling a meeting, we cannot ignore the different time zones involved. Fortunately the `java.time` package has thought of that, too.

Time Zones

The authors of the new `java.time` package certainly encourage you to use the local variations of the time and date classes where possible. Adding support for time zones means adding complexity to your app—they want you to avoid that complexity if possible. But there are many scenarios where support for time zones is unavoidable. You can work with "zoned" dates and times using the `ZonedDateTime` and `OffsetDateTime` classes. The zoned variant understands named time zones and things like daylight saving adjustments. The offset variant is a constant, simple numeric offset from UTC/Greenwich.

Most user-facing uses of dates and times will use the named zone approach, so let's look at creating a zoned date-time. To attach a zone, we use the `ZoneId` class, which has the common `of()` static method for creating new instances. You can supply a region zone as a `String` to get your zoned value:

```
jshell> LocalDateTime piLocal = LocalDateTime.parse("2019-03-14T01:59")
piLocal ==> 2019-03-14T01:59

jshell> ZonedDateTime piCentral = piLocal.atZone(ZoneId.of("America/Chicago"))
piCentral ==> 2019-03-14T01:59-05:00[America/Chicago]
```

And now you can do things like make sure your friends in Paris are able to join you at the correct moment using the verbose but aptly named `withZoneSameInstant()` method:

```
jshell> ZonedDateTime piAlaMode =
piCentral.withZoneSameInstant(ZoneId.of("Europe/Paris"))
piAlaMode ==> 2019-03-14T07:59+01:00[Europe/Paris]
```

If you have other friends who aren't conveniently located in a major metropolitan region but you want them to join as well, you can use the `systemDefault()` method of `ZoneId` to pick up their time zone programmatically:

```
jshell> ZonedDateTime piOther =
piCentral.withZoneSameInstant(ZoneId.systemDefault())
piOther ==> 2019-03-14T02:59-04:00[America/New_York]
```

In our case, *jshell* was running on a laptop in the standard Eastern time zone (not during the daylight saving period) of the United States, and `piOther` comes out exactly as hoped. The `systemDefault()` zone ID is a very handy way to quickly tailor date-times from some other zone to match what your user's clock and calendar are most likely to say. In commercial applications you may want to let the user tell you their preferred zone, but `systemDefault()` is usually a good guess.

Parsing and Formatting Dates and Times

For creating and showing our local and zoned date-times using strings, we've been relying on the default formats that follow ISO values and generally work wherever we need to accept or display dates and times. But as every programmer knows, "generally" is not "always." Fortunately, you can use the utility class `java.time.format.DateTimeFormatter` to help with both parsing input and formatting output.

The core of `DateTimeFormatter` centers on building a format string that governs both parsing and formatting. You build up your format with the pieces listed in Table 8-4. We are only listing a portion of the options available here, but these should get you through the bulk of the dates and times you will encounter. Note that case matters when using the characters mentioned!

Table 8-4. Popular `DateTimeFormatter` elements

Character	Description	Example
y	year-of-era	2004; 04
M	month-of-year	7; 07
L	month-of-year	Jul; July; J
d	day-of-month	10
E	day-of-week	Tue; Tuesday; T
a	am-pm-of-day	PM
h	clock-hour-of-am-pm (1-12)	12
K	hour-of-am-pm (0-11)	0
k	clock-hour-of-day (1-24)	24
H	hour-of-day (0-23)	0
m	minute-of-hour	30
s	second-of-minute	55
S	fraction-of-second	033954
z	time-zone name	Pacific Standard Time; PST
Z	zone-offset	+0000; -0800; -08:00

To put together a common US short format, for example, you could use the `M`, `d`, and `y` characters. You build the formatter using the static `ofPattern()` method. Now the formatter can be used (and reused) with the `parse()` method of any of the date or time classes:

```
jshell> import java.time.format.DateTimeFormatter

jshell> DateTimeFormatter shortUS = DateTimeFormatter.ofPattern("MM/dd/yy")
shortUS ==> Value(MonthOfYe ... (YearOfEra,2,2,2000-01-01)

jshell> LocalDate valentines = LocalDate.parse("02/14/19", shortUS)
valentines ==> 2019-02-14

jshell> LocalDate piDay = LocalDate.parse("03/14/19", shortUS)
piDay ==> 2019-03-14
```

And as we mentioned earlier, the formatter works in both directions. Just use the `format()` method of your formatter to produce a string representation of your date or time:

```
jshell> LocalDate today = LocalDate.now()
today ==> 2019-12-14

jshell> shortUS.format(today)
$30 ==> "12/14/19"
```

```
jshell> shortUS.format(piDay)
$31 ==> "03/14/19"
```

Of course, formatters work for times and date-times as well!

```
jshell> DateTimeFormatter military = DateTimeFormatter.ofPattern("HHmm")
military ==> Value(HourOfDay,2)Value(MinuteOfHour,2)

jshell> LocalTime sunset = LocalTime.parse("2020", military)
sunset ==> 20:20

jshell> DateTimeFormatter basic = DateTimeFormatter.ofPattern("h:mm a")
basic ==> Value(ClockHourOfAmPm)':'Value(MinuteOfHour,2)' 'Text(AmPmOfDay,SHORT)

jshell> basic.format(sunset)
$42 ==> "8:20 PM"

jshell> DateTimeFormatter appointment =
DateTimeFormatter.ofPattern("h:mm a MM/dd/yy z")
appointment ==>
Value(ClockHourOfAmPm)':' ...
0-01-01)' 'ZoneText(SHORT)

jshell> ZonedDateTime dentist =
ZonedDateTime.parse("10:30 AM 11/01/19 EST", appointment)
dentist ==> 2019-11-01T10:30-04:00[America/New_York]

jshell> ZonedDateTime nowEST = ZonedDateTime.now()
nowEST ==> 2019-12-14T09:55:58.493006-05:00[America/New_York]

jshell> appointment.format(nowEST)
$47 ==> "9:55 AM 12/14/19 EST"
```

Notice in the `ZonedDateTime` portion above that we put the time zone identifier (the `z` character) at the end—probably not where you were expecting it! We wanted to illustrate the power of these formats. You can design a format to accommodate a very wide range of input or output styles. Legacy data and poorly designed web forms come to mind as direct examples of where `DateTimeFormatter` can help you retain your sanity.

Parsing Errors

Even with all this parsing power at your fingertips, things will sometimes go wrong. And regrettably, the exceptions you see are often too vague to be immediately useful. Consider the following attempt to parse a time with hours, minutes, and seconds:

```
jshell> DateTimeFormatter withSeconds = DateTimeFormatter.ofPattern("hh:mm:ss")
withSeconds ==>
Value(ClockHourOfAmPm,2)':' ...
Value(SecondOfMinute,2)
```

```
jshell> LocalTime.parse("03:14:15", withSeconds)
|  Exception java.time.format.DateTimeParseException:
|  Text '03:14:15' could not be parsed: Unable to obtain
|  LocalTime from TemporalAccessor: {MinuteOfHour=14, MilliOfSecond=0,
|  SecondOfMinute=15, NanoOfSecond=0, HourOfAmPm=3,
|  MicroOfSecond=0},ISO of type java.time.format.Parsed
|        at DateTimeFormatter.createError (DateTimeFormatter.java:2020)
|        at DateTimeFormatter.parse (DateTimeFormatter.java:1955)
|        at LocalTime.parse (LocalTime.java:463)
|        at (#33:1)
|  Caused by: java.time.DateTimeException:
|  Unable to obtain LocalTime from ...
|        at LocalTime.from (LocalTime.java:431)
|        at Parsed.query (Parsed.java:235)
|        at DateTimeFormatter.parse (DateTimeFormatter.java:1951)
|        ...
```

Yikes! A `DateTimeParseException` will be thrown any time the string input cannot be parsed. It will also be thrown in cases like our example above; the fields were correctly parsed from the string but they did not supply enough information to create a `LocalTime` object. It may not be obvious, but our time, "3:14:15," could be either mid-afternoon or very, very early in the morning. Our choice of the `hh` pattern for the hours turns out to be the culprit. We can either pick an hour pattern that uses an unambiguous 24-hour scale or we can add an explicit AM/PM element:

```
jshell> DateTimeFormatter valid1 = DateTimeFormatter.ofPattern("hh:mm:ss a")
valid1 ==> Value(ClockHourOfAmPm,
2)':'Value(MinuteOfHour,2)' ... 2)' 'Text(AmPmOfDay,SHORT)

jshell> DateTimeFormatter valid2 = DateTimeFormatter.ofPattern("HH:mm:ss")
valid2 ==> Value(HourOfDay,2)':'Value(MinuteOfHour,2)':'Value(SecondOfMinute,2)

jshell> LocalTime piDay1 = LocalTime.parse("03:14:15 PM", valid1)
piDay1 ==> 15:14:15

jshell> LocalTime piDay2 = LocalTime.parse("03:14:15", valid2)
piDay2 ==> 03:14:15
```

So if you ever get a `DateTimeParseException` but your input looks like a correct match for the format, double-check that your format itself includes everything necessary to create your date or time. One parting thought on these exceptions: you may need to use the nonmnemonic "u" character for parsing years.

There are many, *many* more details on `DateTimeFormatter`. More than most utility classes, it's worth a trip to read the docs online (*https://oreil.ly/rhosl*).