also throws an exception, the previous exception from line 7 is lost, with the code printing the following:

```
Exception in thread "main" java.lang.RuntimeException:
   and we couldn't find them
   at JammedTurkeyCage.main(JammedTurkeyCage.java:9)
```

This has always been and continues to be bad programming practice. We don't want to lose exceptions! Although out of scope for the exam, the reason for this has to do with backward compatibility. This behavior existed before automatic resource management was added.

Formatting Values

We now shift gears a bit and talk about how to format data for users. In this section, we're going to be working with numbers, dates, and times. This is especially important in the next section when we expand customization to different languages and locales. You may want to review Chapter 4, "Core APIs," if you need a refresher on creating various date/time objects.

Formatting Numbers

In Chapter 4, you saw how to control the output of a number using the `String.format()` method. That's useful for simple stuff, but sometimes you need finer-grained control. With that, we introduce the `NumberFormat` interface, which has two commonly used methods:

```
public final String format(double number)
public final String format(long number)
```

Since `NumberFormat` is an interface, we need the concrete `DecimalFormat` class to use it. It includes a constructor that takes a pattern `String`:

```
public DecimalFormat(String pattern)
```

The patterns can get quite complex. But luckily, for the exam you only need to know about two formatting characters, shown in Table 11.5.

TABLE 11.5 `DecimalFormat` symbols

Symbol	Meaning	Examples
`#`	Omit position if no digit exists for it.	`$2.2`
`0`	Put `0` in position if no digit exists for it.	`$002.20`

These examples should help illuminate how these symbols work:

```
12: double d = 1234.567;
13: NumberFormat f1 = new DecimalFormat("###,###,###.0");
14: System.out.println(f1.format(d));  // 1,234.6
15:
16: NumberFormat f2 = new DecimalFormat("000,000,000.00000");
17: System.out.println(f2.format(d));  // 000,001,234.56700
18:
19: NumberFormat f3 = new DecimalFormat("Your Balance $#,###,###.##");
20: System.out.println(f3.format(d));  // Your Balance $1,234.57
```

Line 14 displays the digits in the number, rounding to the nearest 10th after the decimal. The extra positions to the left are omitted because we used #. Line 17 adds leading and trailing zeros to make the output the desired length. Line 20 shows prefixing a nonformatting character along with rounding because fewer digits are printed than available. Notice that the commas are automatically removed if they are used between # symbols.

As you see in the localization section, there's a second concrete class that inherits `NumberFormat` that you'll need to know for the exam.

Formatting Dates and Times

The date and time classes support many methods to get data out of them.

```
LocalDate date = LocalDate.of(2022, Month.OCTOBER, 20);
System.out.println(date.getDayOfWeek());  // THURSDAY
System.out.println(date.getMonth());      // OCTOBER
System.out.println(date.getYear());       // 2022
System.out.println(date.getDayOfYear());  // 293
```

Java provides a class called `DateTimeFormatter` to display standard formats.

```
LocalDate date = LocalDate.of(2022, Month.OCTOBER, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dt = LocalDateTime.of(date, time);

System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE));
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));
System.out.println(dt.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
```

The code snippet prints the following:

```
2022-10-20
11:12:34
2022-10-20T11:12:34
```

The `DateTimeFormatter` will throw an exception if it encounters an incompatible type. For example, each of the following will produce an exception at runtime since it attempts to format a date with a time value, and vice versa:

```
date.format(DateTimeFormatter.ISO_LOCAL_TIME);  // RuntimeException
time.format(DateTimeFormatter.ISO_LOCAL_DATE);  // RuntimeException
```

Customizing the Date/Time Format

If you don't want to use one of the predefined formats, `DateTimeFormatter` supports a custom format using a date format `String`.

```
var f = DateTimeFormatter.ofPattern("MMMM dd, yyyy 'at' hh:mm");
System.out.println(dt.format(f));  // October 20, 2022 at 11:12
```

Let's break this down a bit. Java assigns each letter or symbol a specific date/time part. For example, `M` is used for month, while `y` is used for year. And case matters! Using `m` instead of `M` means it will return the minute of the hour, not the month of the year.

What about the number of symbols? The number often dictates the format of the date/time part. Using `M` by itself outputs the minimum number of characters for a month, such as `1` for January, while using `MM` always outputs two digits, such as `01`. Furthermore, using `MMM` prints the three-letter abbreviation, such as `Jul` for July, while `MMMM` prints the full month name.

It's possible, albeit unlikely, to come across questions on the exam that use `SimpleDateFormat` rather than the more useful `DateTimeFormatter`. If you do see it on the exam used with an older `java.util.Date` object, just know that the custom formats that are likely to appear on the exam will be compatible with both.

Learning the Standard Date/Time Symbols

For the exam, you should be familiar enough with the various symbols that you can look at a date/time `String` and have a good idea of what the output will be. Table 11.6 includes the symbols you should be familiar with for the exam.

TABLE 11.6 Common date/time symbols

Symbol	Meaning	Examples
y	Year	22, 2022
M	Month	1, 01, Jan, January
d	Day	5, 05

Symbol	Meaning	Examples
h	Hour	9, 09
m	Minute	45
S	Second	52
a	a.m./p.m.	AM, PM
z	Time zone name	Eastern Standard Time, EST
Z	Time zone offset	-0400

Let's try some examples. What do you think the following prints?

```
var dt = LocalDateTime.of(2022, Month.OCTOBER, 20, 6, 15, 30);

var formatter1 = DateTimeFormatter.ofPattern("MM/dd/yyyy hh:mm:ss");
System.out.println(dt.format(formatter1));  // 10/20/2022 06:15:30

var formatter2 = DateTimeFormatter.ofPattern("MM_yyyy_-_dd");
System.out.println(dt.format(formatter2));  // 10_2022_-_20

var formatter3 = DateTimeFormatter.ofPattern("h:mm z");
System.out.println(dt.format(formatter3));  // DateTimeException
```

The first example prints the date, with the month before the day, followed by the time. The second example prints the date in a weird format with extra characters that are just displayed as part of the output.

The third example throws an exception at runtime because the underlying `LocalDateTime` does not have a time zone specified. If `ZonedDateTime` were used instead, the code would complete successfully and print something like `06:15 EDT`, depending on the time zone.

As you saw in the previous example, you need to make sure the format `String` is compatible with the underlying date/time type. Table 11.7 shows which symbols you can use with each of the date/time objects.

Make sure you know which symbols are compatible with which date/time types. For example, trying to format a month for a `LocalTime` or an hour for a `LocalDate` will result in a runtime exception.

TABLE 11.7 Supported date/time symbols

Symbol	LocalDate	LocalTime	LocalDateTime	ZonedDateTime
y	√		√	√
M	√		√	√
d	√		√	√
h		√	√	√
m		√	√	√
s		√	√	√
a		√	√	√
z				√
Z				√

Selecting a *format()* Method

The date/time classes contain a `format()` method that will take a formatter, while the formatter classes contain a `format()` method that will take a date/time value. The result is that either of the following is acceptable:

```
var dateTime = LocalDateTime.of(2022, Month.OCTOBER, 20, 6, 15, 30);
var formatter = DateTimeFormatter.ofPattern("MM/dd/yyyy hh:mm:ss");

System.out.println(dateTime.format(formatter));  // 10/20/2022 06:15:30
System.out.println(formatter.format(dateTime));  // 10/20/2022 06:15:30
```

These statements print the same value at runtime. Which syntax you use is up to you.

Adding Custom Text Values

What if you want your format to include some custom text values? If you just type them as part of the format `String`, the formatter will interpret each character as a date/time symbol. In the best case, it will display weird data based on extra symbols you enter. In the worst case, it will throw an exception because the characters contain invalid symbols. Neither is desirable!

One way to address this would be to break the formatter into multiple smaller formatters and then concatenate the results.

```
var dt = LocalDateTime.of(2022, Month.OCTOBER, 20, 6, 15, 30);
```

```
var f1 = DateTimeFormatter.ofPattern("MMMM dd, yyyy ");
var f2 = DateTimeFormatter.ofPattern(" hh:mm");
System.out.println(dt.format(f1) + "at" + dt.format(f2));
```

This prints `October 20, 2022 at 06:15` at runtime.

While this works, it could become difficult if a lot of text values and date symbols are intermixed. Luckily, Java includes a much simpler solution. You can *escape* the text by surrounding it with a pair of single quotes (`'`). Escaping text instructs the formatter to ignore the values inside the single quotes and just insert them as part of the final value.

```
var f = DateTimeFormatter.ofPattern("MMMM dd, yyyy 'at' hh:mm");
System.out.println(dt.format(f));  // October 20, 2022 at 06:15
```

But what if you need to display a single quote in the output, too? Welcome to the fun of escaping characters! Java supports this by putting two single quotes next to each other.

We conclude our discussion of date formatting with some examples of formats and their output that rely on text values, shown here:

```
var g1 = DateTimeFormatter.ofPattern("MMMM dd', Party''s at' hh:mm");
System.out.println(dt.format(g1));  // October 20, Party's at 06:15

var g2 = DateTimeFormatter.ofPattern("'System format, hh:mm: 'hh:mm");
System.out.println(dt.format(g2));  // System format, hh:mm: 06:15

var g3 = DateTimeFormatter.ofPattern("'NEW! 'yyyy', yay!'");
System.out.println(dt.format(g3));  // NEW! 2022, yay!
```

If you don't escape the text values with single quotes, an exception will be thrown at runtime if the text cannot be interpreted as a date/time symbol.

```
DateTimeFormatter.ofPattern("The time is hh:mm");  // Exception thrown
```

This line throws an exception since `T` is an unknown symbol. The exam might also present you with an incomplete escape sequence.

```
DateTimeFormatter.ofPattern("'Time is: hh:mm: ");  // Exception thrown
```

Failure to terminate an escape sequence will trigger an exception at runtime.

Supporting Internationalization and Localization

Many applications need to work in different countries and with different languages. For example, consider the sentence "The zoo is holding a special event on 4/1/22 to look at animal behaviors." When is the event? In the United States, it is on April 1. However, a British reader would interpret this as January 4. A British reader might also wonder why we