```
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

didn't write "behaviours." If we are making a website or program that will be used in multiple countries, we want to use the correct language and formatting.

Internationalization is the process of designing your program so it can be adapted. This involves placing strings in a properties file and ensuring that the proper data formatters are used. *Localization* means supporting multiple locales or geographic regions. You can think of a locale as being like a language and country pairing. Localization includes translating strings to different languages. It also includes outputting dates and numbers in the correct format for that locale.

Initially, your program does not need to support multiple locales. The key is to future-proof your application by using these techniques. This way, when your product becomes successful, you can add support for new languages or regions without rewriting everything.

In this section, we look at how to define a locale and use it to format dates, numbers, and strings.

Picking a Locale

While Oracle defines a locale as "a specific geographical, political, or cultural region," you'll only see languages and countries on the exam. Oracle certainly isn't going to delve into political regions that are not countries. That's too controversial for an exam!

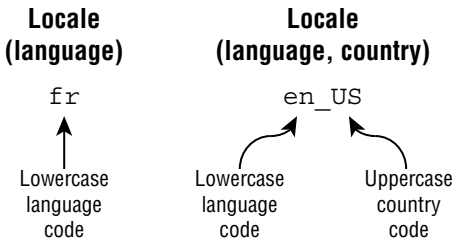
The `Locale` class is in the `java.util` package. The first useful `Locale` to find is the user's current locale. Try running the following code on your computer:

```
Locale locale = Locale.getDefault();
System.out.println(locale);
```

When we run it, it prints `en_US`. It might be different for you. This default output tells us that our computers are using English and are sitting in the United States.

Notice the format. First comes the lowercase language code. The language is always required. Then comes an underscore followed by the uppercase country code. The country is optional. Figure 11.6 shows the two formats for `Locale` objects that you are expected to remember.

FIGURE 11.6 `Locale` formats



As practice, make sure that you understand why each of these `Locale` identifiers is invalid:

```
US     // Cannot have country without language
enUS   // Missing underscore
US_en  // The country and language are reversed
EN     // Language must be lowercase
```

The corrected versions are `en` and `en_US`.

You do not need to memorize language or country codes. The exam will let you know about any that are being used. You do need to recognize valid and invalid formats. Pay attention to uppercase/lowercase and the underscore. For example, if you see a locale expressed as `es_CO`, then you should know that the language is `es` and the country is `CO`, even if you didn't know that they represent Spanish and Colombia, respectively.

As a developer, you often need to write code that selects a locale other than the default one. There are three common ways of doing this. The first is to use the built-in constants in the `Locale` class, available for some common locales.

```
System.out.println(Locale.GERMAN);   // de
System.out.println(Locale.GERMANY);  // de_DE
```

The first example selects the German language, which is spoken in many countries, including Austria (`de_AT`) and Liechtenstein (`de_LI`). The second example selects both German the language and Germany the country. While these examples may look similar, they are not the same. Only one includes a country code.

The second way of selecting a `Locale` is to use the constructors to create a new object. You can pass just a language, or both a language and country:

```
System.out.println(new Locale("fr"));        // fr
System.out.println(new Locale("hi", "IN"));  // hi_IN
```

The first is the language French, and the second is Hindi in India. Again, you don't need to memorize the codes. There is another constructor that lets you be even more specific about the locale. Luckily, providing a variant value is not on the exam.

Java will let you create a `Locale` with an invalid language or country, such as `xx_XX`. However, it will not match the `Locale` that you want to use, and your program will not behave as expected.

There's a third way to create a `Locale` that is more flexible. The builder design pattern lets you set all of the properties that you care about and then build the `Locale` at the end. This means that you can specify the properties in any order. The following two `Locale` values both represent `en_US`:

```
Locale l1 = new Locale.Builder()
   .setLanguage("en")
```

```
    .setRegion("US")
    .build();

Locale l2 = new Locale.Builder()
    .setRegion("US")
    .setLanguage("en")
    .build();
```

When testing a program, you might need to use a `Locale` other than your computer's default.

```
System.out.println(Locale.getDefault());  // en_US
Locale locale = new Locale("fr");
Locale.setDefault(locale);
System.out.println(Locale.getDefault());  // fr
```

Try it, and don't worry—the `Locale` changes for only that one Java program. It does not change any settings on your computer. It does not even change future executions of the same program.

The exam may use `setDefault()` because it can't make assumptions about where you are located. In practice, we rarely write code to change a user's default locale.

Localizing Numbers

It might surprise you that formatting or parsing currency and number values can change depending on your locale. For example, in the United States, the dollar sign is prepended before the value along with a decimal point for values less than one dollar, such as `$2.15`. In Germany, though, the euro symbol is appended to the value along with a comma for values less than one euro, such as `2,15 €`.

Luckily, the `java.text` package includes classes to save the day. The following sections cover how to format numbers, currency, and dates based on the locale.

The first step to formatting or parsing data is the same: obtain an instance of a `NumberFormat`. Table 11.8 shows the available factory methods.

Once you have the `NumberFormat` instance, you can call `format()` to turn a number into a `String`, or you can use `parse()` to turn a `String` into a number.

The format classes are not thread-safe. Do not store them in instance variables or `static` variables. You learn more about thread safety in Chapter 13, "Concurrency."

TABLE 11.8 Factory methods to get a `NumberFormat`

Description	Using default `Locale` and a specified `Locale`
General-purpose formatter	`NumberFormat.getInstance()` `NumberFormat.getInstance(Locale locale)`
Same as `getInstance`	`NumberFormat.getNumberInstance()` `NumberFormat.getNumberInstance(Locale locale)`
For formatting monetary amounts	`NumberFormat.getCurrencyInstance()` `NumberFormat.getCurrencyInstance(Locale locale)`
For formatting percentages	`NumberFormat.getPercentInstance()` `NumberFormat.getPercentInstance(Locale locale)`
Rounds decimal values before displaying	`NumberFormat.getIntegerInstance()` `NumberFormat.getIntegerInstance(Locale locale)`

Returns compact number formatter	`NumberFormat.getCompactNumberInstance()`
	`NumberFormat.getCompactNumberInstance(`
	`Locale locale, NumberFormat.Style formatStyle)`

Formatting Numbers

When we format data, we convert it from a structured object or primitive value into a `String`. The `NumberFormat.format()` method formats the given number based on the locale associated with the `NumberFormat` object.

Let's go back to our zoo for a minute. For marketing literature, we want to share the average monthly number of visitors to the San Diego Zoo. The following shows printing out the same number in three different locales:

```
int attendeesPerYear = 3_200_000;
int attendeesPerMonth = attendeesPerYear / 12;

var us = NumberFormat.getInstance(Locale.US);
System.out.println(us.format(attendeesPerMonth));  // 266,666

var gr = NumberFormat.getInstance(Locale.GERMANY);
System.out.println(gr.format(attendeesPerMonth));  // 266.666

var ca = NumberFormat.getInstance(Locale.CANADA_FRENCH);
System.out.println(ca.format(attendeesPerMonth));  // 266 666
```

This shows how our U.S., German, and French Canadian guests can all see the same information in the number format they are accustomed to using. In practice, we would just call `NumberFormat.getInstance()` and rely on the user's default locale to format the output.

Formatting currency works the same way.

```
double price = 48;
var myLocale = NumberFormat.getCurrencyInstance();
System.out.println(myLocale.format(price));
```

When run with the default locale of `en_US` for the United States, this code outputs `$48.00`. On the other hand, when run with the default locale of `en_GB` for Great Britain, it outputs `£48.00`.

In the real world, use `int` or `BigDecimal` for money and not `double`. Doing math on amounts with `double` is dangerous because the values are stored as floating-point numbers. Your boss won't appreciate it if you lose pennies or fractions of pennies during transactions!

Finally, the exam may have examples that show formatting percentages:

```
double successRate = 0.802;
var us = NumberFormat.getPercentInstance(Locale.US);
System.out.println(us.format(successRate));  // 80%

var gr = NumberFormat.getPercentInstance(Locale.GERMANY);
System.out.println(gr.format(successRate));  // 80 %
```

Not much difference, we know, but you should at least be aware that the ability to print a percentage is locale-specific for the exam!

Parsing Numbers

When we parse data, we convert it from a `String` to a structured object or primitive value. The `NumberFormat.parse()` method accomplishes this and takes the locale into consideration.

For example, if the locale is the English/United States (`en_US`) and the number contains commas, the commas are treated as formatting symbols. If the locale relates to a country or language that uses commas as a decimal separator, the comma is treated as a decimal point.

The `parse()` method, found in various types, declares a checked exception `ParseException` that must be handled or declared in the method in which it is called.

Let's look at an example. The following code parses a discounted ticket price with different locales. The `parse()` method throws a checked `ParseException`, so make sure to handle or declare it in your own code.

```
String s = "40.45";

var en = NumberFormat.getInstance(Locale.US);
System.out.println(en.parse(s));  // 40.45

var fr = NumberFormat.getInstance(Locale.FRANCE);
System.out.println(fr.parse(s));  // 40
```

In the United States, a dot (`.`) is part of a number, and the number is parsed as you might expect. France does not use a decimal point to separate numbers. Java parses it as a formatting character, and it stops looking at the rest of the number. The lesson is to make sure that you parse using the right locale!

The `parse()` method is also used for parsing currency. For example, we can read in the zoo's monthly income from ticket sales:

```
String income = "$92,807.99";
var cf = NumberFormat.getCurrencyInstance();
double value = (Double) cf.parse(income);
System.out.println(value);  // 92807.99
```

The currency string `"$92,807.99"` contains a dollar sign and a comma. The `parse` method strips out the characters and converts the value to a number. The return value of `parse` is a `Number` object. `Number` is the parent class of all the `java.lang` wrapper classes, so the return value can be cast to its appropriate data type. The `Number` is cast to a `Double` and then automatically unboxed into a `double`.

Formatting with *CompactNumberFormat*

The second class that inherits `NumberFormat` that you need to know for the exam is `CompactNumberFormat`. It is new to the Java 17 exam, so you're likely to see a question on it!

`CompactNumberFormat` is similar to `DecimalFormat`, but it is designed to be used in places where print space may be limited. It is opinionated in the sense that it picks a format for you, and locale-specific in that output can change depending on your location.

Consider the following sample code that applies a `CompactNumberFormat` five times to two locales, using a `static` import for `Style` (an enum with value `SHORT` or `LONG`):

```
var formatters = Stream.of(
   NumberFormat.getCompactNumberInstance(),
   NumberFormat.getCompactNumberInstance(Locale.getDefault(), Style.SHORT),
   NumberFormat.getCompactNumberInstance(Locale.getDefault(), Style.LONG),
```

```
    NumberFormat.getCompactNumberInstance(Locale.GERMAN, Style.SHORT),
    NumberFormat.getCompactNumberInstance(Locale.GERMAN, Style.LONG),
    NumberFormat.getNumberInstance());

formatters.map(s -> s.format(7_123_456)).forEach(System.out::println);
```

The following is printed by this code when run in the `en_US` locale (line breaks added for readability):

```
7M
7M
7 million

7 Mio.
7 Millionen

7,123,456
```

Notice that the first two lines are the same. If you don't specify a style, `SHORT` is used by default. Next, notice that the values except the last one (which doesn't use a compact number formatter) are truncated. There's a reason it's called a compact number formatter! Also, notice that the short form uses common labels for large values, such as `K` for thousand. Last but not least, the output may differ for you when you run this, as it was run in an `en_US` locale.

Using the same formatters, let's try another example:

```
formatters.map(s -> s.format(314_900_000)).forEach(System.out::println);
```

This prints the following when run in the `en_US` locale:

```
315M
315M
315 million

315 Mio.
315 Millionen

314,900,000
```

Notice that the third digit is automatically rounded up for the entries that use a `CompactNumberFormat`. The following summarizes the rules for `CompactNumberFormat`:

- First it determines the highest range for the number, such as thousand (`K`), million (`M`), billion (`B`), or trillion (`T`).
- It then returns up to the first three digits of that range, rounding the last digit as needed.
- Finally, it prints an identifier. If `SHORT` is used, a symbol is returned. If `LONG` is used, a space followed by a word is returned.

For the exam, make sure you understand the difference between the `SHORT` and `LONG` formats and common symbols like `M` for million.

Localizing Dates

Like numbers, date formats can vary by locale. Table 11.9 shows methods used to retrieve an instance of a `DateTimeFormatter` using the default locale.

TABLE 11.9 Factory methods to get a `DateTimeFormatter`

Description	Using default `Locale`
For formatting dates	`DateTimeFormatter.ofLocalizedDate(FormatStyle dateStyle)`
For formatting times	`DateTimeFormatter.ofLocalizedTime(FormatStyle timeStyle)`
For formatting dates and times	`DateTimeFormatter.ofLocalizedDateTime (FormatStyle dateStyle, FormatStyle timeStyle)` `DateTimeFormatter.ofLocalizedDateTime (FormatStyle dateTimeStyle)`

Each method in the table takes a `FormatStyle` parameter (or two) with possible values `SHORT`, `MEDIUM`, `LONG`, and `FULL`. For the exam, you are not required to know the format of each of these styles.

What if you need a formatter for a specific locale? Easy enough—just append `withLocale(locale)` to the method call.

Let's put it all together. Take a look at the following code snippet, which relies on a `static` import for the `java.time.format.FormatStyle.SHORT` value:

```
public static void print(DateTimeFormatter dtf,
      LocalDateTime dateTime, Locale locale) {
   System.out.println(dtf.format(dateTime) + " --- "
      + dtf.withLocale(locale).format(dateTime));
}
public static void main(String[] args) {
   Locale.setDefault(new Locale("en", "US"));
   var italy = new Locale("it", "IT");
   var dt = LocalDateTime.of(2022, Month.OCTOBER, 20, 15, 12, 34);
```

```
   // 10/20/22 --- 20/10/22
   print(DateTimeFormatter.ofLocalizedDate(SHORT),dt,italy);

   // 3:12 PM --- 15:12
   print(DateTimeFormatter.ofLocalizedTime(SHORT),dt,italy);

   // 10/20/22, 3:12 PM --- 20/10/22, 15:12
   print(DateTimeFormatter.ofLocalizedDateTime(SHORT,SHORT),dt,italy);
}
```

First we establish `en_US` as the default locale, with `it_IT` as the requested locale. We then output each value using the two locales. As you can see, applying a locale has a big impact on the built-in date and time formatters.

Specifying a Locale Category

When you call `Locale.setDefault()` with a locale, several display and formatting options are internally selected. If you require finer-grained control of the default locale, Java subdivides the underlying formatting options into distinct categories with the `Locale.Category` enum.

The `Locale.Category` enum is a nested element in `Locale` that supports distinct locales for displaying and formatting data. For the exam, you should be familiar with the two enum values in Table 11.10.

TABLE 11.10 `Locale.Category` values

Value	Description
`DISPLAY`	Category used for displaying data about locale
`FORMAT`	Category used for formatting dates, numbers, or currencies

When you call `Locale.setDefault()` with a locale, the `DISPLAY` and `FORMAT` are set together. Let's take a look at an example:

```
10: public static void printCurrency(Locale locale, double money) {
11:    System.out.println(
12:       NumberFormat.getCurrencyInstance().format(money)
13:       + ", " + locale.getDisplayLanguage());
14: }
15: public static void main(String[] args) {
16:    var spain = new Locale("es", "ES");
17:    var money = 1.23;
```

```
18:
19:    // Print with default locale
20:    Locale.setDefault(new Locale("en", "US"));
21:    printCurrency(spain, money);  // $1.23, Spanish
22:
23:    // Print with selected locale display
24:    Locale.setDefault(Category.DISPLAY, spain);
25:    printCurrency(spain, money);  // $1.23, español
26:
27:    // Print with selected locale format
28:    Locale.setDefault(Category.FORMAT, spain);
29:    printCurrency(spain, money);  // 1,23 €, español
30: }
```

The code prints the same data three times. First it prints the language of the `spain` and `money` variables using the locale `en_US`. Then it prints it using the `DISPLAY` category of `es_ES`, while the `FORMAT` category remains `en_US`. Finally, it prints the data using both categories set to `es_ES`.

For the exam, you do not need to memorize the various display and formatting options for each category. You just need to know that you can set parts of the locale independently. You should also know that calling `Locale.setDefault(us)` after the previous code snippet will change both locale categories to `en_US`.

Loading Properties with Resource Bundles

Up until now, we've kept all of the text strings displayed to our users as part of the program inside the classes that use them. Localization requires externalizing them to elsewhere.

A *resource bundle* contains the locale-specific objects to be used by a program. It is like a map with keys and values. The resource bundle is commonly stored in a properties file. A *properties file* is a text file in a specific format with key/value pairs.

Our zoo program has been successful. We are now getting requests to use it at three more zoos! We already have support for U.S.-based zoos. We now need to add Zoo de La Palmyre in France, the Greater Vancouver Zoo in English-speaking Canada, and Zoo de Granby in French-speaking Canada.

We immediately realize that we are going to need to internationalize our program. Resource bundles will be quite helpful. They will let us easily translate our application to multiple locales or even support multiple locales at once. It will also be easy to add more locales later if zoos in even more countries are interested. We thought about which locales we need to support, and we came up with four: