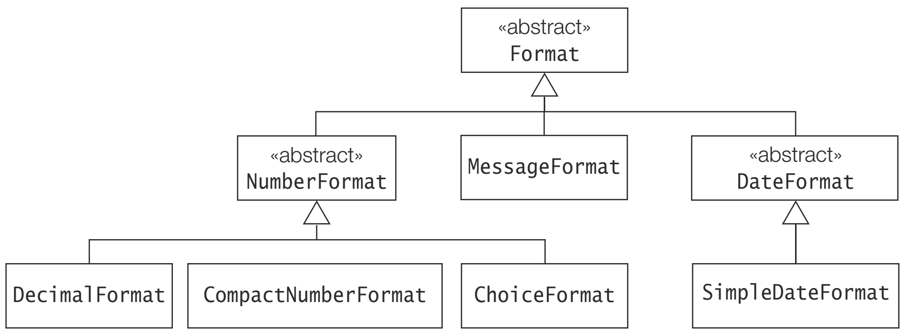
18.4 Core API for Formatting and Parsing of Values

The abstract class `Format` and its subclasses in the `java.text` package provide support for formatting and parsing of dates, times, numbers, currencies, and percentages (Figure 18.1).

Figure 18.1 *Core Classes for Formatting in the* `java.text` *Package*



A *formatter* has two primary functions. The first is to create a human-readable text representation of a value, called *formatting*. The second is to create a value from a character sequence, such as a string, containing a text representation of the value, called *parsing*.

In the rest of this chapter, we emphasize the *locale-sensitive* nature of formatting and parsing of different kinds of values.

- *Formatting numbers, currencies, and percentages (p. 1116)*

  The support for formatting numbers, currencies, and percentages is provided by the abstract class `java.text.NumberFormat` and its subclasses `java.text.DecimalFormat` and `java.text.CompactNumberFormat`.

- *Formatting dates and times (p. 1127)*

  The support provided for formatting dates and times by the abstract class `java.text.DataFormat` and its subclass `java.text.SimpleFormat` is aimed at objects of the `java.util.Date` legacy class, and is *not* covered in this book.

  Support for the new Date and Time API is provided by the `java.time` package, which is covered extensively in Chapter 17, p. 1023. The class `java.time.format.DateTimeFormatter` provides the support for formatting date and time values created using the new Date and Time API.

- *Formatting messages (p. 1139)*

  The support for formatting *messages* containing numbers, currencies, dates, and times is provided by the concrete class `java.text.MessageFormat`. The class `java.text.ChoiceFormat` allows a choice of formatting dependent on numerical values, and is typically used for *conditional formatting* of messages.

  The support provided by the `MessageFormat` class for date and time values is aimed at objects of the legacy class `Date`, but there are a number of ways that the

MessageFormat class can be utilized with formatting objects of the new Date and Time API, and they are discussed later (p. 1139).

18.5 Formatting and Parsing Number, Currency, and Percentage Values

The abstract class java.text.NumberFormat and its subclasses java.text.DecimalFormat and java.text.CompactNumberFormat provide methods for locale-sensitive formatting and parsing of *number*, *currency*, and *percentage* values.

Static Factory Methods to Create a Formatter

The abstract class NumberFormat provides static factory methods for creating locale-sensitive formatters for number, currency, and percentage values. However, the locale cannot be changed after the formatter is created. The factory methods return instances of the concrete classes DecimalFormat and CompactNumberFormat for formatting number, currency, and percentage values.

```
static NumberFormat getNumberInstance()
static NumberFormat getNumberInstance(Locale locale)
```

Return a general formatter for numbers—that is, a number formatter.

```
static NumberFormat getCurrencyInstance()
static NumberFormat getCurrencyInstance(Locale locale)
```

Return a formatter for currency values—that is, a currency formatter.

```
static NumberFormat getPercentInstance()
static NumberFormat getPercentInstance(Locale locale)
```

Return a formatter for percentages—that is, a percentage formatter.

```
static NumberFormat getCompactNumberInstance()
static NumberFormat getCompactNumberInstance(Locale locale,
                                             NumberFormat.Style formatStyle)
```

Return a compact number formatter with the default compact format style or a specific compact format style, respectively (p. 1120). For example, the value 2_345_678 can be formatted by a compact number formatter as "2M" with the SHORT compact form style and as "2 million" with the LONG compact form style, respectively, for the US locale.

In all cases, a locale can be specified to localize the formatter.

Formatting Number, Currency, and Percentage Values

A number formatter can be used to format a double, a long value, or an object. The abstract class NumberFormat provides the following concrete methods for this purpose. Depending on the number formatter, the formatting is locale-sensitive, determined by the default or a specific locale.

```
String format(double d)
String format(long l)
```

Formats the specified number and returns the resulting string.

```
String format(Object obj)              // inherited from the Format class.
```

Formats the specified object and returns the resulting string. For example, it can be used to format `BigInteger` and `BigDecimal` numbers.

The following code shows how we can create a *number formatter* for the Norwegian locale and one for the US locale, and use them to format numbers according to rules of the locale. The number formatted is a `double` (1a) or a `BigDecimal` (1b), giving the same results. Note that the grouping of the digits and the decimal separator used in formatting is according to the locale.

```
double num = 12345.6789;                                  // (1a)
// BigDecimal num = new BigDecimal("12345.6789");         // (1b)

Locale locNOR = new Locale("no", "NO");                   // Norway
NumberFormat nfNOR = NumberFormat.getNumberInstance(locNOR);
System.out.println(nfNOR.format(num));                    // 12 345,679

NumberFormat nfUS = NumberFormat.getNumberInstance(Locale.US);
System.out.println(nfUS.format(num));                     // 12,345.679
```

The following code shows how we can create a *currency* formatter for the Norwegian locale, and use it to format currency values according to this locale. Note the currency symbol and the grouping of the digits, with the amount being rounded to two decimal places. Also note that the delimiter between the currency symbol and the first digit is a *non-breaking space* (*nbsp*), having the unicode `\u00a0`; that is, it is not a normal space (`\u0020`). The grouping of digits is also with a nbsp. Such use of a nbsp is also locale-specific—formatting for the US locale has no nbsp, as can be seen below.

```
NumberFormat cfNOR = NumberFormat.getCurrencyInstance(locNOR);
String formattedCurrStr = cfNOR.format(num);
System.out.println(formattedCurrStr);              // kr 12 345,68 (with 2 nbsp)

NumberFormat cfUS = NumberFormat.getCurrencyInstance(Locale.US);
String formattedCurrStrUS = cfUS.format(num);
System.out.println(formattedCurrStrUS);            // $12,345.68
```

The value 1.0 is equivalent to 100%. A value is converted to a percentage by multiplying it by 100, and by default rounding it to an integer. By default, the result is rounded up only if the decimal part is *equal to or greater than* 0.5 after multiplying by 100. However, both the number of decimal places and the rounding behavior for the percentage value can be changed (p. 1122).

The following code shows how we can create a *percentage* formatter for the Norwegian locale, and use it to format percentage values according to this locale. Also note that the delimiter between the percentage number and the currency symbol is a *non-breaking space*. Again, such use of a nbsp is also locale-specific—formatting percentages for the US locale has no nbsp, as can be seen below. Note the rounding of the percentage values.

```
double rebate = 0.746;
NumberFormat pfNOR = NumberFormat.getPercentInstance(locNOR);
String formattedPStr = pfNOR.format(rebate);
System.out.println(formattedPStr);              // 75 %  (with nbsp)

NumberFormat pfUS = NumberFormat.getPercentInstance(Locale.US);
String formattedPStrUS = pfUS.format(rebate);
System.out.println(formattedPStrUS);            // 75%

System.out.println(pfUS.format(0.745));         // 74%
```

Accounting Currency Formatting

By default, the minus sign (-) is used as a prefix when formatting negative numbers using a currency formatter returned by the `NumberFormat.getCurrencyInstance()` method.

```
NumberFormat df0 = NumberFormat.getCurrencyInstance(Locale.US);
System.out.println(df0.format(-9.99));         // -$9.99
```

However, in accountancy, it is a common practice to enclose a negative currency value in parentheses, `()`. For example, `($9.99)` is interpreted as `-$9.99`. One way to format negative currency values with parentheses for a given locale is to create the locale so that it allows this style for currency formatting. The code below illustrates how this can be achieved for the US locale.

A US locale is created at (1) below that formats negative currency values with parentheses. The `Locale.forLanguageTag()` method creates a locale from the specified language tag `en-US-u-cf-account` that consists of several subtags separated by a hyphen:

- `en`: Represents the English language
- `US`: Represents the United States
- `u`: Indicates that a *Unicode locale/language extension* is specified as a *key–value* pair
- `cf-account`: The key–value pair, where the key `cf` stands for *currency format*, and the value `account` means to use parentheses for negative numbers

The language tag `en-US-u-cf-standard` will result in a US locale that is the same as `Locale.US`. For further details, kindly consult the Unicode Locale Data Markup Language (LDML).

The currency formatter created at (2) for the locale created at (1) will now use parentheses for negative currency values.

```
Locale loc = Locale.forLanguageTag("en-US-u-cf-account");   // (1)
NumberFormat df = NumberFormat.getCurrencyInstance(loc);    // (2)
System.out.println(df.format(-9.99));                       // ($9.99)
```

Parsing Strings to Number, Currency, and Percentage Values

A *number* formatter can be used to parse strings that contain text representations of *numeric* values. The abstract class `NumberFormat` provides the following concrete methods for this purpose.

> `Number parse(String str) throws ParseException`
>
> Parses the text in the specified string from the beginning of the string to produce a `Number`. No leading whitespace is allowed. Any trailing characters after the parsed text are ignored. This method throws the checked `java.text.ParseException` if unsuccessful.
>
> `void setParseIntegerOnly(boolean intOnly)`
> `boolean isParseIntegerOnly()`
>
> Set or get the status if this formatter should only parse integers.

The class `DecimalFormat` specifically provides the following concrete methods to customize the formatter to parse `BigDecimal` numbers.

> `void setParseBigDecimal(boolean newValue)`
> `boolean isParseBigDecimal()`
>
> Set or get the status if this formatter should parse `BigDecimal` numbers.

The code calling the `parse()` method must be prepared to handle a checked `ParseException`—for brevity, exception handling is omitted in the code below.

The following code shows the Norwegian number formatter from earlier being used to parse strings. At (1), the result is a `long` value because the dot (`.`) in the input string is a delimiter and not a legal character according to the number format used in the Norwegian locale. At (2), the result is a `double` value because the comma (`,`) in the input string is the decimal separator in the Norwegian locale. Note that the print statement prints the resulting number according to the *default* locale (U.S., in this case). For the US locale, the dot (.) and the comma (,) are interpreted as the decimal separator and the group separator, respectively.

```
System.out.println(nfNOR.parse("9876.598"));     // (1) 9876
System.out.println(nfNOR.parse("9876,598"));     // (2) 9876.598


System.out.println(nfUS.parse("9876.598"));      // (3) 9876.598
System.out.println(nfUs.parse("9876,598"));      // (4) 9876598
```

In order to parse a string to a `BigDecimal`, we can proceed via a `DecimalFormat` that is customized to parse a `BigDecimal`. The overloaded `parse()` method below returns a `Number`, which is actually a `BigDecimal` in this case.

```
DecimalFormat dfUS = (DecimalFormat) nfUS;
dfUS.setParseBigDecimal(true);
BigDecimal bd = (BigDecimal) dfUS.parse("9876,598");
```

The following code demonstrates using a *currency* formatter as a parser. Note that the currency symbol is interpreted according to the locale in the currency formatter.

In the Norwegian locale, the decimal symbol is not a period (`.`) when parsing numbers; it is a *delimiter* in the input string, as can be seen at (1) below. For some locales, a *non-breaking space* (`\u00a0`) may be required between the currency symbol and the first digit, as can be seen at (2).

The grouping symbol in the Norwegian locale is not a space, and therefore, it is interpreted as a delimiter, as can be seen at (3). An explicit nbsp can be used for grouping digits, if necessary, in order to parse the input string, as at (4). For the US locale, no such considerations are necessary, as can be seen at (5).

```
System.out.println(cfNOR.parse("kr\u00a09876.59"));         // (1) 9876
System.out.println(cfNOR.parse("kr\u00a09876,59"));         // (2) 9876.59
System.out.println(cfNOR.parse("kr\u00a09 876,59"));        // (3) 9
System.out.println(cfNOR.parse("kr\u00a09\u00a0876,59"));   // (4) 9876.59

System.out.println(cfUS.parse("$9876.59"));                 // (5) 9876.59
```

When parsing percentages, a *nbsp* might be required between the percentage value and the percent sign (`%`) in the input string, as can be seen at (1) below for the Norwegian locale. For parsing a percentage value according to the US locale, no such consideration is necessary, as can be seen at (2). However, the format of the percentage value is according to the locale used by the formatter, as can be seen at (1) and (2).

```
System.out.println(pfNOR.parse("15,75\u00a0%"));            // (1) 0.1575
System.out.println(pfUS.parse("25.5%"));                    // (2) 0.255
```

Compact Number Formatting

A *compact number formatter* creates a textual representation that represents the *compact form* of a number. This formatter is an instance of the `CompactNumberFormat` that can be created by calling the `getCompactNumberInstance()` factory method of the `NumberFormat` class. For example, the value `1_000_000` can be formatted in a compact form as `"1M"` or `"1 million"`.

The compact form can be specified by the constants of the enum type `NumberFormat.Style`, shown in Table 18.6. The compact form has a suffix, depending on the value of the number and the locale. Suffixes for the US locale are shown in Table 18.6. The compact form of numbers below 1000 is without any suffix.

Table 18.6 *Compact Number Styles*

Styles for compact number form	Verbosity	Suffix for the US locale in compact form
`NumberFormat.Style.SHORT`	Short number format style (default)	`T` (Trillion), `B` (Billion), `M` (Million), `K` (Thousand) Examples: `2T`, `2.5M`, `1.5K`
`NumberFormat.Style.LONG`	Long number format style	`trillion, billion, million, thousand` Examples: `2 trillion`, `2.5 million`, `1.5 thousand`

The code below creates two compact number formatters for the US locale that use the SHORT and the LONG compact number styles, respectively, to format numbers to their compact form.

```
NumberFormat shortCompactFormat = NumberFormat.getCompactNumberInstance(
    Locale.US, NumberFormat.Style.SHORT)
NumberFormat longCompactFormat = NumberFormat.getCompactNumberInstance(
    Locale.US, NumberFormat.Style.LONG);
```

The compact number formatters are used on different numerical values to create their compact form, as shown in Table 18.7.

```
System.out.println(shortCompactFormat.format(9_400_000));    // 9M
System.out.println(longCompactFormat.format(9_400_000));     // 9 million
```

The second row in Table 18.7 shows the compact form generated after the number of maximum fraction digits is set to 2 (p. 1122):

```
shortCompactFormat.setMaximumFractionDigits(2);
longCompactFormat.setMaximumFractionDigits(2);
```

Note the rounding that takes place depending on the value of the number. By default, RoundingMode.HALF_EVEN is used (see Table 18.9).

Table 18.7 *Formatting Numbers to Compact Form*

Number n	Compact form returned by shortCompactFormatter.format(n) method	Compact form returned by longCompactFormatter.format(n) method
9_400_000 9_500_000 12_500 12_510 999	"9M" "10M" "12K" "13K" "999"	"9 million" "10 million" "12 thousand" "13 thousand" "999"
9_400_000 9_500_000 12_500 12_510 999	*(Max fraction digits = 2)* "9.4M" "9.5M" "12.5K" "12.51K" "999"	*(Max fraction digits = 2)* "9.4 million" "9.5 million" "12.5 thousand" "12.51 thousand" "999"

Compact Number Parsing

The parse() method of the compact number formatter can be used to parse a string that contains a compact form to a numerical value.

```
try {
  System.out.println(shortCompactFormat.parse("9M"));          // 9000000
  System.out.println(longCompactFormat.parse("9 million"));   // 9000000
} catch (ParseException pe) {
  System.out.println(pe);
}
```

The compact number formatters above are used to parse different compact forms to numerical values, as shown in Table 18.8. Note that parsing requires that the compact form ends in an appropriate suffix; otherwise, the suffix is ignored, as we can see in Table 18.8.

Table 18.8 *Parsing Compact Form to Numbers*

Compact form string s	Number returned by `shortCompactFormatter.parse(s)` method	Number returned by `longCompactFormatter.parse(s)` method
"9M" "9.5M" "2K" "1.5K" "999"	9000000 9500000 2000 1500 999	9 9.5 2 1.5 999
"9 million" "9.5 million" "2 thousand" "1.5 thousand"	9 9.5 2 1.5	9000000 9500000 2000 1500

Specifying the Number of Digits

The following methods of the `NumberFormat` abstract class and its subclass `DecimalFormat` allow formatting of numbers to be further refined by setting the number of digits to be allowed in the integral and the decimal part of a number. This also applies for `BigDecimal` numbers. However, a concrete number formatter can enforce certain limitations on these bounds. In addition, a rounding mode can be set as explained below.

```
void setMinimumIntegerDigits(int n)
int  getMinimumIntegerDigits()

void setMaximumIntegerDigits(int n)
int  getMaximumIntegerDigits()

void setMinimumFractionDigits(int n)
int  getMinimumFractionDigits()

void setMaximumFractionDigits(int n)
int  getMaximumFractionDigits()
```

Set or get the minimum or maximum number of digits to be allowed in the integral or decimal part of a number.

```
void setRoundingMode(RoundingMode roundingMode)
```

Sets the *rounding mode* used in this number formatter—that is, how the resulting value is rounded by the formatter. The enum type `java.math.RoundingMode` defines constants for such modes (see Table 18.9).

Table 18.9 *Selected Rounding Modes*

Enum type `java.math.RoundingMode` constants	**Description**
`CEILING`	Rounds toward *positive infinity*. ` 1.1 ->  2` (same as `UP`) `-1.8 -> -1` (same as `DOWN`)
`FLOOR`	Rounds toward *negative infinity*. ` 1.8 ->  1` (same as `DOWN`) `-1.1 -> -2` (same as `UP`)

`UP`	Rounds *away from zero*. Never *decreases* the *magnitude* of the calculated value.
	` 1.1 ->  2`
	`-1.1 -> -2`

`DOWN`	Rounds *toward zero*. Never *increases* the *magnitude* of the calculated value.
	` 1.8 ->  1`
	`-1.8 -> -1`

`HALF_UP`	Rounds toward the nearest neighboring value, unless both neighboring values are equidistant, in which case it rounds *up*. This is the same as *normal rounding*.
	` 1.5 ->  2 (same as UP, if discarded fraction is >= 0.5)`
	` 1.4 ->  1 (same as DOWN, if discarded fraction is < 0.5)`
	`-1.4 -> -1 (same as DOWN, if discarded fraction is < 0.5)`
	`-1.5 -> -2 (same as UP, if discarded fraction is >= 0.5)`

`HALF_DOWN`	Rounds toward the nearest neighboring value, unless both neighboring values are equidistant, in which case it rounds *down*.
	` 1.6 ->  2 (same as UP, if discarded fraction is > 0.5)`
	` 1.5 ->  1 (same as DOWN, if discarded fraction is <= 0.5)`
	`-1.5 -> -1 (same as DOWN, if discarded fraction is <= 0.5)`
	`-1.6 -> -2 (same as UP, if discarded fraction is > 0.5)`

`HALF_EVEN`	Rounds toward the nearest neighboring value, unless both neighboring values are equidistant, in which case it rounds toward the even neighbor. This rounding policy is used in floating-point arithmetic in Java.
	` 2.5 ->  2 (same as HALF_DOWN, if digit left of discarded fraction is even)`
	` 1.5 ->  2 (same as HALF_UP, if digit left of discarded fraction is odd)`
	`-1.5 -> -2 (same as HALF_UP, if digit left of discarded fraction is odd)`
	`-2.5 -> -2 (same as HALF_DOWN, if digit left of discarded fraction is even)`

Example 18.5 demonstrates the rounding modes defined by the enum type `java.math.RoundingMode`. The examples in Table 18.9 are computed in Example 18.5. A number format is created at (1) for the US locale. The maximum number of digits in the fraction part is set to 0 at (2); that is, the floating-point value will be rounded to an integer before formatting.

The method `roundIt()` at (3) does the rounding and the formatting when passed a formatter, the maximum number of digits (0) required in the fraction part, the rounding mode, and the two values to round and format. The method sets the maximum number of digits required in the fraction part and the rounding mode at (4) and (5), respectively, in the formatter. The formatting of the values is done by calling the `format()` method at (6) and (7). We encourage checking the output against the rules for rounding for each mode shown in Table 18.9.

Example 18.5 *Rounding Modes*

```
import java.math.RoundingMode;
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

public class Rounding {

  public static void main(String[] args) {
    System.out.println(" Rounding:   v1         v2");
    NumberFormat nfmtUS = NumberFormat.getNumberInstance(Locale.US);     // (1)
    int maxFractionDigits = 0;                                           // (2)
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.CEILING,   1.1, -1.8);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.FLOOR,     1.8, -1.1);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.UP,        1.1, -1.1);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.DOWN,      1.8, -1.8);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_UP,   1.5, 1.4);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_UP,   -1.4, -1.5);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_DOWN, 1.6, 1.5);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_DOWN, -1.5, -1.6);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_EVEN, 2.5, 1.5);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_EVEN, -1.5, -2.5);
  }

  static void roundIt(NumberFormat nf, int maxFractionDigits, RoundingMode rMode,
                      double v1, double v2) {                  // (3)
    nf.setMaximumFractionDigits(maxFractionDigits);            // (4)
    nf.setRoundingMode(rMode);                                 // (5)
    System.out.printf("%9s: ", rMode);
    System.out.printf("%5s -> %2s ", v1, nf.format(v1));       // (6)
    System.out.printf("%5s -> %2s%n", v2, nf.format(v2));      // (7)
  }
}
```

Output from the program:

```
 Rounding:   v1         v2
  CEILING:   1.1 ->  2  -1.8 -> -1
    FLOOR:   1.8 ->  1  -1.1 -> -2
       UP:   1.1 ->  2  -1.1 -> -2
     DOWN:   1.8 ->  1  -1.8 -> -1
  HALF_UP:   1.5 ->  2   1.4 ->  1
  HALF_UP:  -1.4 -> -1  -1.5 -> -2
HALF_DOWN:   1.6 ->  2   1.5 ->  1
HALF_DOWN:  -1.5 -> -1  -1.6 -> -2
HALF_EVEN:   2.5 ->  2   1.5 ->  2
HALF_EVEN:  -1.5 -> -2  -2.5 -> -2
```

Customizing Decimal Number Formatting

The `java.text.DecimalFormat` class formats and parses numbers (including `BigDecimal`) according to a *string pattern*, specified using special *pattern symbols* (Table 18.10) and taking into account any locale considerations. It can format numbers as integers (`1234`), fixed-point notation (`3.14`), scientific notation (`2.99792458E8`), percentages (`60%`), and currency amounts (`$64.00`).

Although the `DecimalFormat` class defines constructors, the Java API recommends using the factory methods of its abstract superclass `NumberFormat` to create an instance of this class.

```
NumberFormat df = NumberFormat.getNumberInstance(Locale.US);
if (nf intanceof DecimalFormat) {
  DecimalFormat df = (DecimalFormat) nf;
  df.applyPattern(pattern);                // Supply the pattern.
  String output = df.format(number);       // Format the number.
}
```

The idiom is to create a `DecimalFormat` object as shown above, localized if necessary, and customizing it with a pattern using the method shown below. The overloaded `format()` methods inherited by the `DecimalFormat` class from its superclass `NumberFormat` can now be called to format a number. The pattern can also be changed on the fly.

`void applyPattern(String pattern)`

Applies the given pattern to this `DecimalFormat` object.

Table 18.10 *Selected Number Format Pattern Symbols for the United States*

Pattern symbol	Purpose
`0` (Zero)	Placeholder for a digit. Can result in leading and/or trailing zeroes. Can result in rounding up of the fractional part of a number.
`#` (Hash sign)	Placeholder for a digit. Leading and/or trailing zeroes may be absent. Can result in rounding up of the fractional part of a number.
`.` (Dot)	Placeholder for *decimal separator*. If omitted, a floating-point number is rounded up to the nearest integer. Can be different for other countries. For example, in Germany, the decimal separator is the comma (`,`).
`,` (Comma)	Placeholder for *grouping separator* (a.k.a. *thousands separator*). Can be different for other countries. For example, in Germany, the grouping separator is the period (`.`).

Example 18.6 shows formatting of a number, either a `double` at (1a) or a `BigDecimal` at (1b), using different patterns at (2) and different locales at (3). Decimal formatters are created at (4), as explained above. A decimal formatter is customized with

a pattern at (5) and the number is formatted at (6). In particular, note how the decimal and group separators are localized for each locale. The currency amounts are also formatted according to the particulars of each locale, and not according to the currency symbol that is specified in the currency patterns.

Example 18.6 *Using the* `DecimalFormat` *class*

```
import java.math.BigDecimal;
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

public class FormattingDecimalNumbers {

  public static void main(String[] args) {
    // The number to format.
    double number = 1234.567;                                           // (1a)
//  BigDecimal number = new BigDecimal("1234.567");                     // (1b)

    // Formats to use:
    String[] patterns = {                                               // (2)
        "#",
        "###,###.##",
        "###.##",
        "00000.00",
        "BTC ###,###.##",                                        // BTC: Bitcoin
    };

    // Locales to consider:
    Locale[] locales = { Locale.US, Locale.GERMANY };                   // (3)

    // Create localized DecimalFormats:                                    (4)
    List<DecimalFormat> dfs = new ArrayList<>(locales.length);
    for (int i = 0; i < locales.length; i++) {
      NumberFormat nf = NumberFormat.getNumberInstance(locales[i]);
      if (nf instanceof DecimalFormat) {
        dfs.add((DecimalFormat) nf);
      }
    }

    // Write the header:
    System.out.printf("%15s", "Patterns");
    for (Locale locale : locales) {
      System.out.printf("%15s", locale);
    }
    System.out.println();

    // Do formatting and print results:
    for (String pattern : patterns) {
      System.out.printf("%15s", pattern);
      for (DecimalFormat df : dfs) {
        df.applyPattern(pattern);                                     // (5)
```

```
        String output = df.format(number);                              // (6)
        System.out.printf("%15s", output);
      }
      System.out.println();
    }
  }
}
```

Output from the program:

```
       Patterns          en_US          de_DE
              #           1235           1235
    ###,###.##       1,234.57       1.234,57
        ###.##        1234.57        1234,57
      00000.00       01234.57       01234,57
BTC ###,###.##   BTC 1,234.57   BTC 1.234,57
```

18.6 Formatting and Parsing Date and Time

The class `java.time.format.DateTimeFormatter` provides the support for locale-sensitive formatting and parsing of *date* and *time* values.

In this section we take a closer look at formatting temporal objects and parsing character sequences for temporal objects. In particular, we consider the following formatters, which provide increasing flexibility in customizing formatting and parsing:

- *Default formatters* are implicitly used by such methods as the `toString()` method of the temporal classes.
- *Predefined formatters* are ready-made formatters provided as constants by the `java.time.format.DateTimeFormatter` class, such as those that adhere to the ISO standard (Table 18.11, p. 1130).
- *Style-based formatters* are formatters that use the format styles defined by the constants of the `java.time.format.FormatStyle` enum type (Table 18.12, p. 1132). These formatters are created by the static factory methods `ofLocalized`*`Type`*`()` of the `DateTimeFormatter` class, where *`Type`* is either `Time`, `Date`, or `DateTime` (Table 18.13, p. 1132).
- *Pattern-based formatters* use customized format styles defined by *pattern letters* (Table 18.15, p. 1135). These formatters are created by the static factory method `ofPattern()` of the `DateTimeFormatter` class.

The `DateTimeFormatter` class provides static factory methods for obtaining a formatter. The idiom for using a formatter is to obtain a formatter first, localize it if necessary, and then pass it to the methods responsible for formatting and parsing temporal objects. Each of the temporal classes `LocalTime`, `LocalDate`, `LocalDateTime`, and `ZonedDateTime` provides the following methods: an instance method `format()` and a static method `parse()`. These two methods do the formatting and the parsing according to the rules of the formatter that is passed as an argument, respectively.

Analogous methods for formatting and parsing temporal objects passed as an arguments are also provided by the `DateTimeFormatter` class (see below). For example, the following code lines give the same result string, where `date` and `df` refer to a `LocalDate` object and a `DateTimeFormatter` object for formatting dates, respectively:

```
String resultStr1 = date.format(df);
String resultStr2 = df.format(date);
boolean eqStr = resultStr1.equals(resultStr2);    // true
```

From the method headers of the `format()` and the `parse()` methods of the temporal classes, we can see that these methods will readily compile with *any* `DateTimeFormatter`. The validity of the formatter for a given temporal object is resolved at runtime, resulting in a resounding exception if it is not valid.

```
// Defined by LocalTime, LocalDate, LocalDateTime, and ZonedDateTime
String format(DateTimeFormatter formatter)
```

Formats this temporal object using the specified formatter, and returns the resulting string. Each temporal class provides this method. The temporal object is formatted according to the rules of the formatter. The method throws a `java.time.DateTimeException` if formatting is not successful.

```
static TemporalType parse(CharSequence text)
static TemporalType parse(CharSequence text, DateTimeFormatter formatter)
```

Each temporal class provides these two static methods, where *`TemporalType`* can be any of the temporal classes `LocalTime`, `LocalDate`, `LocalDateTime`, or `ZonedDateTime`.

The first method returns an instance of the *`TemporalType`* from a character sequence, using the default parsing rules for the *`TemporalType`*.

The second method obtains an instance of the *`TemporalType`* from a character sequence using the specified formatter.

Both methods return an object of a specific temporal class, and both throw a `java.time.format.DateTimeParseException` if parsing is not successful.

Formatters supplied by the `DateTimeFormatter` class are immutable and thread-safe. All formatters created using static methods provided by the `DateTimeFormatter` class can be localized by the `localizedBy()` method in this class. The `DateTimeFormatter` class provides methods that can be used to format and parse temporal objects.

```
// Defined by DateTimeFormatter
DateTimeFormatter localizedBy(Locale locale)
```

Returns a copy of this formatter that will use the specified locale (§18.1, p. 1096). Although the formatters in the examples presented in this section use the default locale (the US locale in this case), we encourage the reader to experiment with changing the locale of a formatter with this method in the examples.

`String format(TemporalAccessor temporal)`

Formats a temporal object using this formatter. The main temporal classes implement the `TemporalAccessor` interface.

This method throws an unchecked `java.time.DateTimeException` if an error occurs during formatting.

`TemporalAccessor parse(CharSequence text)`

This method fully parses the text to produce a temporal object. It throws an unchecked `java.time.DateTimeException` if an error occurs during parsing.

`DateTimeFormatter withZone(ZoneId zone)`

Returns a copy of this formatter with a new override zone—that is, a formatter with similar state to this formatter but with the override zone set.

Default Formatters for Date and Time Values

Default formatters rely on the `toString()` method of the individual temporal classes for creating a text representation of a temporal object. The default formatter used by the `toString()` method applies the formatting rules defined by the ISO standard.

In the following code, the result of formatting a `LocalTime` object is shown at (1):

```
LocalTime time = LocalTime.of(12, 30, 15, 99);
String strTime = time.toString();                // (1) 12:30:15.000000099
LocalTime parsedTime = LocalTime.parse(strTime);  // (2)
System.out.println(time.toString().equals(parsedTime.toString())); // true
```

Each temporal class provides a static method `parse(CharSequence text)` that parses a character sequence using a default formatter that complies with the ISO standard. In the preceding code, the text representation created at (1) is parsed at (2) to obtain a new `LocalTime` object. Not surprisingly, the text representations of the two `LocalTime` objects referred to by the references `time` and `parsedTime` are equal.

The line of code below shows that the argument string passed to the `parse()` method is not in accordance with the ISO standard, resulting in a runtime exception:

```
LocalTime badTime = LocalTime.parse("12.30.15");  // DateTimeParseException
```

The examples in this section make heavy use of the `toString()` method to format temporal objects according to the ISO standard.

Predefined Formatters for Date and Time Values

The `DateTimeFormatter` class provides a myriad of predefined formatters for temporal objects, the majority of which comply with the ISO standard. Table 18.11 shows selected ISO-based predefined formatters from this class. We have also indicated in the last column which temporal classes a formatter can be used with for formatting and parsing.

For example, the row for the `ISO_LOCAL_DATE` formatter in Table 18.11 indicates that this formatter can be used for formatting a `LocalDate` and for formatting the date part of a `LocalDateTime` or a `ZonedDateTime`. It can parse the text representation of a `LocalDate`. In contrast, the row for the `ISO_LOCAL_DATE_TIME` formatter indicates that this formatter can be used for formatting a `LocalDateTime` and for formatting the date-time part of a `ZonedDateTime`. It can parse the text representation of a `LocalDateTime`. In addition, the text representation of a `LocalDateTime` can be parsed by this formatter to a `LocalTime` or a `LocalDate` by the `parse(text, formatter)` method of the appropriate class.

Table 18.11 *Selected ISO-Based Predefined Formatters for Date and Time*

DateTimeFormatter	**Examples**	**Formatting**	**Parsing**
ISO_LOCAL_TIME	12:30:15	LocalTime *Time part of:* LocalDateTime ZonedDateTime	LocalTime
BASIC_ISO_DATE	20210428	LocalDate *Date part of:* LocalDateTime ZonedDateTime	LocalDate
ISO_LOCAL_DATE	2021-04-28	LocalDate *Date part of:* LocalDateTime ZonedDateTime	LocalDate
ISO_LOCAL_DATE_TIME	2021-04- 28T12:30:15	LocalDateTime *Date-time part of:* ZonedDateTime	LocalTime LocalDate LocalDateTime
ISO_ZONED_DATE_TIME	2021-04- 28T12:30:15+01:00 [Europe/Paris]	ZonedDateTime	LocalTime LocalDate LocalDateTime ZonedDateTime
ISO_INSTANT	2021-04- 28T12:30:15.00000 0500Z	Instant	Instant

An example of using an ISO-based predefined formatter is given next. Note that the formatter obtained at (1) is a formatter for date fields. It can be used only with temporal objects that have date fields—in other words, the `LocalDate`, `LocalDateTime`, and `ZonedDateTime` classes. This formatter is passed at (2) to the `format()` method, to create a text representation of a date. The resulting string is parsed at (3) by the `parse()` method that uses the same formatter. The resulting date is also formatted using the same formatter at (4). It is hardly surprising that the text representations of both dates are equal.

```
DateTimeFormatter df = DateTimeFormatter.ISO_LOCAL_DATE;     // (1)
LocalDate date = LocalDate.of(1935, 1, 8);
String strDate = date.format(df);                            // (2) 1935-01-08
LocalDate parsedDate = LocalDate.parse(strDate, df);         // (3)
System.out.println(strDate + "|" +
                   parsedDate.format(df));          // (4) 1935-01-08|1935-01-08
```

As this code shows, a formatter can be reused, both for formatting and for parsing. The code at (4) in the code below applies the formatter from (1) in the preceding code snippet to format a `LocalDateTime` object. It should not come as a surprise that the resulting text representation of the `LocalDateTime` object pertains to only date fields in the temporal object; the time fields of the `LocalDateTime` object are ignored. Parsing this text representation back with the same formatter at (5) will yield only a `LocalDate` object.

```
LocalDateTime dateTime = LocalDateTime.of(1935, 1, 8, 12, 45);
String strDate2 = dateTime.format(df);                    // (4) 1935-01-08
LocalDate parsedDate2 = LocalDate.parse(strDate2, df);     // (5) LocalDate
```

To summarize, the `DateTimeFormatter.ISO_LOCAL_DATE` can be used to format and parse a `LocalDate`, but can only format the date part of a `LocalDateTime` object (or a `ZonedDateTime` object).

Using this date formatter with a `LocalTime` object is courting disaster, as shown by the following code. Formatting with this formatter results in an unchecked `java.time.temporal.UnsupportedTemporalTypeException`, and parsing results in an unchecked `java.time.format.DateTimeParseException`.

```
String timeStr2 = LocalTime.NOON.format(df);   // UnsupportedTemporalTypeException
LocalTime time2 = LocalTime.parse("12:00", df);// DateTimeParseException
```

The `DateTimeFormatter.ISO_INSTANT` predefined formatter shown in Table 18.11 is implicitly used by the `parse(text)` and `toString()` methods of the `Instant` class.

Style-Based Formatters for Date and Time Values

For more flexible formatters than the predefined ISO-based formatters, the `DateTimeFormatter` class provides the static factory methods `ofLocalized`*`Type`*`()`, where *`Type`* is either `Time`, `Date`, or `DateTime`. These methods create formatters that use a specific format style. However, the format style cannot be changed after the formatter is created. Format styles are defined by the enum type `java.time.format.FormatStyle`, and are shown in Table 18.12. The styles define format patterns that vary in their degree of verbosity.

The format style in a style-based formatter can be made locale-specific by setting the desired locale in the formatter using the `localizedBy()` method of the `DateTimeFormatter` class (p. 1128).

```
static DateTimeFormatter ofLocalizedTime(FormatStyle timeStyle)
static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)
static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle)
static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateStyle,
                                             FormatStylse timeStyle)
```

These static factory methods of the `DateTimeFormatter` class create a formatter that will format a time, a date, or a date-time, respectively, using the specified format style. The formatter can also be used to parse a character sequence for a time, a date, or a date-time, respectively.

Table 18.12 *Format Styles for Date and Time*

Styles for date/time	Verbosity	Example formatting a date (default locale: United States)
`FormatStyle.SHORT`	Short-style pattern	`1/11/14`
`FormatStyle.MEDIUM`	Medium-style pattern	`Jan 11, 2014`
`FormatStyle.LONG`	Long-style pattern	`January 11, 2014`
`FormatStyle.FULL`	Full-style pattern	`Saturday, January 11, 2014`

In the discussion below, we will make use of the following temporal objects.

```
LocalTime time = LocalTime.of(14, 15, 30);            // 14:15:30
LocalDate date = LocalDate.of(2021, 12, 1);           // 2021-12-01
LocalDateTime dateTime = LocalDateTime.of(date, time); // 2021-12-01T14:15:30
// 2021-12-01T14:15:30-06:00[US/Central]
ZonedDateTime zonedDateTime = ZonedDateTime.of(dateTime, ZoneId.of("US/Central"));
```

Table 18.13 shows which temporal classes can be formatted by a combination of a format style from Table 18.12 and an `ofLocalized`*`Type`*`()` method of the `DateTimeFormatter` class, where *`Type`* is either `Time`, `Date`, or `DateTime`. For example, in Table 18.13 we can see that the method call `DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)` will return a formatter that can be used to format instances of the `LocalDate` class, but will only format the date part of a `LocalDateTime` or a `ZonedDateTime` instance, as can be seen in the code below. This particular formatter requires date fields in the temporal object, and instances of these three temporal classes fit the bill.

```
DateTimeFormatter dfs = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
String str1 = date.format(dfs);           // 12/1/21
String str2 = dateTime.format(dfs);       // Date part: 12/1/21
String str3 = zonedDateTime.format(dfs);  // Date part: 12/1/21
```

Table 18.13 *Using Style-Based Formatters*

Format-Style	Using formatters created by factory methods of the `DateTimeFormatter` class: *`temporalReference`*`.format(formatter)`		
	`ofLocalizedTime(style)`	`ofLocalizedDate(style)`	`ofLocalizedDateTime(style)`

`SHORT`
`MEDIUM`
	`LocalTime`
	Time part of:
	`LocalDateTime`
	`ZonedDateTime`
		`LocalDate`
		Date part of:
		`LocalDateTime`
		`ZonedDateTime`
			`LocalDateTime`
			Date-time part of:
			`ZonedDateTime`

`LONG`
`FULL`
	Time part of:
	`ZonedDateTime`
		`LocalDate`
		Date part of:
		`LocalDateTime`
		`ZonedDateTime`
			`ZonedDateTime`

Similarly, the method call `DateTimeFormatter.ofLocalizedTime(FormatStyle.LONG)` will return a formatter that will only format temporal objects that have a time part *and* are compatible with the rules of `FormatStyle.LONG`. Only instances of the `ZonedDateTime` class fit the bill. The code below shows that we can format the time part of a `ZonedDateTime` instance with this formatter.

```
DateTimeFormatter tff = DateTimeFormatter.ofLocalizedTime(FormatStyle.FULL);
String str4 = zonedDateTime.format(tff);  // Time part:
                                          //   2:15:30 PM Central Standard Time
String str5 = time.format(tff);           // java.time.DateTimeException
String str6 = date.format(tff);           // java.time.temporal.
                                          //      UnsupportedTemporalTypeException
String str7 = dateTime.format(tff);       // java.time.DateTimeException
```

In summary, a style-based formatter will only format a temporal object (or its constituent parts) if the temporal object has the temporal parts required by the formatter *and* is compatible with the rules of the format style of the formatter.

Table 18.14 shows how the style-based formatters can be used as parsers. As in Table 18.13, this table also shows the combination of a format style from Table 18.12 and an `ofLocalized`*`Type`*`()` method of the `DateTimeFormatter` class, where *`Type`* is either `Time`, `Date`, or `DateTime`. From the table, we can read which temporal objects can be parsed from a character sequence that is compatible with the rules of a formatter for a given combination of the format style and a specific factory method.

Table 18.14 *Using Style-Based Parsers*

FormatStyle	**Using parsers created by factory methods of the `DateTimeFormatter` class: *`TemporalClass`*`.parse(characterSequence,formatter)`**		
	`ofLocalizedTime(style)`	`ofLocalizedDate(style)`	`ofLocalizedDateTime(style)`

`SHORT`
`MEDIUM`
	`LocalTime`	`LocalDate`	`LocalTime`
			`LocalDate`
			`LocalDateTime`

`LONG`
`FULL`
	`LocalTime`	`LocalDate`	`LocalTime`
			`LocalDate`
			`LocalDateTime`
			`ZonedDateTime`

For any format style, the two methods `ofLocalizedTime()` and `ofLocalizedDate()` return formatters that can be used to parse a compatible character sequence to a `LocalTime` or a `LocalDate` instance, respectively.

In the code that follows, the date formatter created at (1) is used at (3) to parse the input string at (2) to create a `LocalDate` object.

```
DateTimeFormatter df = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);// (1)
String inputStr = "2/29/21";                         // (2) en_US date, SHORT style
LocalDate parsedDate = LocalDate.parse(inputStr, df);    // (3)
System.out.println(parsedDate);                          // (4) 2021-02-28
```

In the above code, the input string `"2/29/21"` is specified in the short style of the default locale (which in our case is the US locale). The input string is parsed by the date formatter (using the `SHORT` format style) to create a new `LocalDate` object.

Although the value 29 is invalid for the number of days in February for the year 2021, the output shows that it was adjusted correctly. The contents of the input string (in this case, "2/29/21") must be compatible with the rules of the format style in the date formatter (in this case, FormatStyle.SHORT). If this is not the case, a DateTimeParseException is thrown. Note that in the print statement at (4), the LocalDate object from the parsing is converted to a string by the LocalDate.toString() method using the implicit ISO-based formatter.

A formatter returned by the ofLocalizedDateTime() method can parse a character sequence to instances of different temporal classes, as shown in the rightmost column of Table 18.14. This is illustrated by the code below. Such a formatter is created at (1) and localized to the locale for France. It is first used to format a zoned date-time object at (2) to obtain a character string that we can parse with the formatter. At (3), (4), (5), and (6), this sequence is parsed to obtain a LocalTime, a LocalDate, a LocalDateTime, and a ZonedDateTime, respectively, as these temporal objects can be constructed from the format of the character string representing a zoned date-time.

```
DateTimeFormatter dtff
    = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL)
                       .localizedBy(Locale.FRANCE);          // (1)

// "mercredi 1 décembre 2021 à 14:15:30 heure normale du centre nord-américain"
String charSeq = zonedDateTime.format(dtff);                 // (2)
LocalTime pTime = LocalTime.parse(charSeq, dtff);            // (3) 14:15:30
LocalDate pDate = LocalDate.parse(charSeq, dtff);            // (4) 2021-12-01

// 2021-12-01T14:15:30
LocalDateTime pDateTime = LocalDateTime.parse(charSeq, dtff);        // (5)

// 2021-12-01T14:15:30-06:00[America/Chicago]
ZonedDateTime pZonedDateTime = ZonedDateTime.parse(charSeq, dtff);   // (6)
```

In summary, a character sequence can be parsed to a temporal object by a style-based formatter if the character sequence is in the format required by the formatter to create the temporal object.

Pattern-Based Formatters for Date and Time Values

For more fine-grained formatting and parsing capabilities for temporal objects, we can use the ofPattern() method of the DateTimeFormatter class. This method creates immutable formatters that interpret temporal objects according to a string pattern that is defined using the *pattern letters* shown in Table 18.15.

```
static DateTimeFormatter ofPattern(String pattern)
```

This static method creates a formatter using the specified pattern. The set of temporal objects it can be used with depends on the pattern letters used in the specification of the pattern. The letter pattern defines the rules used by the formatter. The method throws an IllegalArgumentException if the pattern is invalid.

Table 18.15 provides an overview of selected pattern letters. All letters are reserved when used in a letter pattern. A sequence of characters can be escaped by enclosing it in single quotes (e.g., `"EEEE 'at' HH:mm"`). Non-letter characters in the string are interpreted verbatim and need not be escaped using single quotes (e.g., `"uuuu.MM.dd @ HH:mm:ss"`). The number of times a pattern letter is repeated can have a bearing on the interpretation of the value of the corresponding date or time field. The uppercase letter `M` (Month of the year) should not be confused with the lowercase letter `m` (minutes in the hour).

Table 18.15 *Selected Date/Time Pattern Letters*

Date or time component	Pattern letter	Examples

Year
(2: two rightmost digits)
Proleptic year: use `u`
Year of era (AD/BC): use `y`
	`u`
	`uu`
	`uuu`
	`uuuu`
	`uuuuu`
		`2021; -2000 (2001 BC)`
		`15;     0` (i.e., 1 BC); `-1` (i.e., 2 BC)
		`2021; -2021` (i.e., 2022 BC)
		`02021` (padding)

Month in year
(1–2: number)
(3: abbreviated text form)
(4: full text form)
	`M`
	`MM`
	`MMM`
	`MMMM`
		`8`
		`08`
		`Aug`
		`August`

Day in month	`d` `dd`	`6` `06`

Day name in week
(1–3: abbreviated text form)
 	
(4: full text form)
	`E`
	`EE`
	`EEE`
	`EEEE`
		`Tue`
		`Tue`
		`Tue`
		`Tuesday`

Hour in day (0–23)	`H` `HH`	`9` `09`
Hour in am/pm (1–12) (does not include the AM/PM marker, but required for parsing)	`h` `hh`	`7` `07`
Minute in hour (0–59)	`m` `mm`	`6` `06`
Second in minute (0–59)	`s` `ss`	`2` `02`

Fraction of a second (`S`)	`SSS`
	`SSSSSS`
	`SSSSSSSSS`
		`123`
		`123456`
		`123456789`

Era designator (AD/BC)	`G`	`AD`
Time zone name (1 to 4)	`z` `zzzz`	`CST` `Central Standard Time`

Time zone offset
(1 to 5)
	`Z`
	`ZZZZ`
	`ZZZZZ`
		`-0600`
		`GMT-06:00`
		`-06:00`

Time zone ID (must be two)	`VV`	`US/Central`

Table 18.15 *Selected Date/Time Pattern Letters (Continued)*

Date or time component	Pattern letter	Examples
AM/PM marker	`a`	`AM`

Period-of-day (used with temporal values having time units)	`B`	`at night`
		`in the morning`
		`noon`
		`in the afternoon`
		`in the evening`

Escape for text	`'`	`'T'` prints as `T`
Single quote	`''`	`'`

A letter pattern can be used to format a temporal object if the temporal object has the temporal fields required by the pattern. The pattern `"'Hour': HH"` can be used to format the hour part of any `LocalTime` object or a `LocalDateTime` object, but not a `LocalDate`.

A letter pattern can be used to parse a string if the string matches the pattern *and* the letter pattern specifies the mandatory parts needed to construct a temporal object. The pattern `"MM/dd/uuuu"` can be used to parse the string `"08/13/2009"` to obtain a `LocalDate` object, but not a `LocalDateTime` object. The latter requires the time part as well.

Example 18.7 demonstrates both formatting temporal objects and parsing character sequences for temporal objects using letter patterns. The `main()` method at (1) calls four methods to format and parse different temporal objects. The formatting of the temporal object can be localized by specifying the desired locale in the `main()` method.

- The method `usingTimePattern()` at (2) demonstrates using a letter pattern for the time part to both format a `LocalTime` and parse a text representation of a `LocalTime`, respectively. The same pattern is used to format only the time part of a `LocalDateTime` and a `ZonedDateTime`, respectively.
- The method `usingDatePattern()` at (3) demonstrates using a letter pattern for the date part to both format a `LocalDate` and parse a text representation of a `LocalDate`, respectively. The same pattern is used to format only the date part of a `LocalDateTime` and a `ZonedDateTime`, respectively.
- The method `usingDateTimePattern()` at (4) demonstrates using a letter pattern for the date and time parts to both format a `LocalDateTime` and parse a text representation of a `LocalDateTime`, respectively. The same pattern is also used to parse the text representation of a `LocalDateTime` to obtain a `LocalDate` and a `LocalTime`, respectively.
- The method `usingZonedDateTimePattern()` at (5) demonstrates using a letter pattern for the date, time, and zone parts to both format a `ZonedDateTime` and parse a text representation of a `ZonedDateTime`, respectively. The same pattern is also used to parse the text representation of a `ZonedDateTime` to obtain a `LocalDateTime`, a `LocalDate`, and a `LocalTime`, respectively.

The usage of letter patterns with the `ofPattern()` method in Example 18.7 is analogous to the usage of style-based formatters provided by the `ofLocalized`*`Type`*`()` methods (Table 18.13, p. 1132). The main difference is that letter patterns provide great flexibility in creating customized format styles.

Example 18.7 *Formatting and Parsing with Letter Patterns*

```
import java.time.*;
import java.time.format.*;
import java.util.Locale;

public class FormattingParsingWithPatterns {

  /** Temporals */
  private static LocalTime time = LocalTime.of(12, 30, 15, 99);
  private static LocalDate date = LocalDate.of(2021, 4, 28);
  private static LocalDateTime dateTime = LocalDateTime.of(date, time);
  private static ZoneId zID = ZoneId.of("US/Central");
  private static ZonedDateTime zonedDateTime = ZonedDateTime.of(dateTime, zID);

  public static void main(String[] args) {                                  // (1)
    Locale locale = Locale.US;
    usingTimePattern(locale);
    usingDatePattern(locale);
    usingDateTimePattern(locale);
    usingZonedDateTimePattern(locale);
  }

  /** Pattern with time part. */
  public static void usingTimePattern(Locale locale) {                      // (2)
    String timePattern = "HH::mm::ss:SSS";
    DateTimeFormatter timeFormatter = DateTimeFormatter.ofPattern(timePattern)
                                                       .localizedBy(locale);
    String strTime = time.format(timeFormatter);
    LocalTime parsedTime = LocalTime.parse(strTime, timeFormatter);
    String strTime2 = dateTime.format(timeFormatter);
    String strTime3 = zonedDateTime.format(timeFormatter);

    System.out.printf("Time pattern: %s%n", timePattern);
    System.out.printf("LocalTime (formatted): %s%n", strTime);
    System.out.printf("LocalTime (parsed):    %s%n", parsedTime);
    System.out.printf("LocalDateTime (formatted time part): %s%n", strTime2);
    System.out.printf("ZonedDateTime (formatted time part): %s%n%n", strTime3);
  }

  /** Pattern with date part. */
  public static void usingDatePattern(Locale locale) {                      // (3)
    String datePattern = "EEEE, uuuu/MMMM/dd";
    DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern(datePattern)
                                                       .localizedBy(locale);
    String strDate = date.format(dateFormatter);
    LocalDate parsedDate = LocalDate.parse(strDate, dateFormatter);
    String strDate2 = dateTime.format(dateFormatter);
    String strDate3 = zonedDateTime.format(dateFormatter);
```

```
    System.out.printf("Date pattern: %s%n", datePattern);
    System.out.printf("LocalDate (formatted): %s%n", strDate);
    System.out.printf("LocalDate (parsed)   : %s%n", parsedDate);
    System.out.printf("LocalDateTime (formatted date part): %s%n", strDate2);
    System.out.printf("ZonedDateTime (formatted date part): %s%n%n", strDate3);
  }

  /** Pattern with date and time parts. */
  public static void usingDateTimePattern(Locale locale) {                 // (4)
    String dtPattern = "EE, HH::mm::ss 'on' uuuu/MM/dd";
    DateTimeFormatter dtFormatter = DateTimeFormatter.ofPattern(dtPattern)
                                                     .localizedBy(locale);
    String strDateTime = dateTime.format(dtFormatter);
    LocalDateTime parsedDateTime = LocalDateTime.parse(strDateTime,
                                                       dtFormatter);
    LocalDate parsedDate3 = LocalDate.parse(strDateTime, dtFormatter);
    LocalTime parsedTime3 = LocalTime.parse(strDateTime, dtFormatter);

    System.out.printf("DateTime pattern: %s%n", dtPattern);
    System.out.printf("LocalDateTime (formatted):    %s%n", strDateTime);
    System.out.printf("LocalDateTime (parsed):       %s%n", parsedDateTime);
    System.out.printf("LocalDate (parsed date part): %s%n", parsedDate3);
    System.out.printf("LocalTime (parsed time part): %s%n%n", parsedTime3);
  }

  /** Pattern with time zone, date and time parts. */
  public static void usingZonedDateTimePattern(Locale locale) {            // (5)
    String zdtPattern = "EE, HH::mm::ss 'on' uuuu/MM/dd VV";
    DateTimeFormatter zdtFormatter = DateTimeFormatter.ofPattern(zdtPattern)
                                                      .localizedBy(locale);
    String strZonedDateTime = zonedDateTime.format(zdtFormatter);
    ZonedDateTime parsedZonedDateTime
        = ZonedDateTime.parse(strZonedDateTime, zdtFormatter);
    LocalDateTime parsedDateTime2
        = LocalDateTime.parse(strZonedDateTime, zdtFormatter);
    LocalDate parsedDate4 = LocalDate.parse(strZonedDateTime,
                                            zdtFormatter);
    LocalTime parsedTime4 = LocalTime.parse(strZonedDateTime,
                                            zdtFormatter);

    System.out.printf("ZonedDateTime pattern: %s%n", zdtPattern);
    System.out.printf("ZonedDateTime (formatted):    %s%n", strZonedDateTime);
    System.out.printf("ZonedDateTime (parsed):       %s%n", parsedZonedDateTime);
    System.out.printf("LocalDateTime (parsed):       %s%n", parsedDateTime2);
    System.out.printf("LocalDate (parsed date part): %s%n", parsedDate4);
    System.out.printf("LocalTime (parsed time part): %s%n", parsedTime4);
  }
}
```

Probable output from the program:

```
Time pattern: HH::mm::ss:SSS
LocalTime (formatted): 12::30::15:000
LocalTime (parsed):    12:30:15
LocalDateTime (formatted time part): 12::30::15:000
ZonedDateTime (formatted time part): 12::30::15:000
```

```
Date pattern: EEEE, uuuu/MMMM/dd
LocalDate (formatted): Wednesday, 2021/April/28
LocalDate (parsed)   : 2021-04-28
LocalDateTime (formatted date part): Wednesday, 2021/April/28
ZonedDateTime (formatted date part): Wednesday, 2021/April/28

DateTime pattern: EE, HH::mm::ss 'on' uuuu/MM/dd
LocalDateTime (formatted):    Wed, 12::30::15 on 2021/04/28
LocalDateTime (parsed):       2021-04-28T12:30:15
LocalDate (parsed date part): 2021-04-28
LocalTime (parsed time part): 12:30:15

ZonedDateTime pattern: EE, HH::mm::ss 'on' uuuu/MM/dd VV
ZonedDateTime (formatted):    Wed, 12::30::15 on 2021/04/28 US/Central
ZonedDateTime (parsed):       2021-04-28T12:30:15-05:00[US/Central]
LocalDateTime (parsed):       2021-04-28T12:30:15
LocalDate (parsed date part): 2021-04-28
LocalTime (parsed time part): 12:30:15
```

18.7 Formatting and Parsing Messages

A *compound message* may contain various kinds of data: strings, numbers, currencies, percentages, dates, and times. The locale-sensitive values in such messages should be formatted according to an appropriate locale. In this section we first consider a solution provided by the `java.text.MessageFormat` class for formatting compound messages. Later we consider the `java.text.ChoiceFormat` class that supports conditional formatting (p. 1145).

An overview of various other support in the Java SE APIs for formatting compound messages can be found at the end of this section (p. 1152).

Format Patterns

The modus operandi of formatters provided by the `MessageFormat` class is a classic one, as exemplified by the `printf()` method called on the `System.out` static field (§1.9, p. 24). A *string pattern* designating *placeholders* for values is specified. `MessageFormat` takes a list of values and formats them, and then inserts their text representation into the respective placeholders in the pattern to create the final result. The simple example below illustrates the basic use of the `MessageFormat` class.

```
String pattern = "At {3} on {2} Elvis landed at {0} and was greeted by {1} fans.";
String output = MessageFormat.format(pattern, "Honolulu", 3000,
                              LocalDate.of(1961,3,25), LocalTime.of(12,15));
System.out.println(output);
```

Output from the code:

```
At 12:15 on 1961-03-25 Elvis landed at Honolulu and was greeted by 3,000 fans.
```

The pattern specifies placeholders (called *format elements*) using curly brackets `{}` to designate where text representations of values are to be inserted. The number specified in a format element is the *argument index* of a particular argument in the list of arguments submitted to the formatter. In the code above, the pattern has four format elements.

The variable arity static method `MessageFormat.format()` is passed the pattern and a list of arguments to format. Below, we can see which argument in the method call is indicated by the argument index in a format element that is specified in the pattern above.

```
Format element:     {0}       {1}          {2}                       {3}
Arguments:       "Honolulu", 3000, LocalDate.of(1961,3,25), LocalTime.of(12,15)
```

Care must be taken that an argument index in a format element designates the right argument. The same argument index in multiple format elements just means that the argument it designates is applied to all of those format elements.

Note that the text outside of the format elements is copied verbatim to the result string. A single quote (`'`) can be used to quote arbitrary characters, and two single quotes (`''`) can be used to escape a single quote in a pattern. For example, the pattern `"'{1}'"` represents the string `"{1}"`, and not a format element.

The flexibility of the `MessageFormat` class will become clear as we dig into its functionality.

The general syntax of a format element is shown below:

```
{ Argument_Index }
{ Argument_Index, Format_Type }
{ Argument_Index, Format_Type, Format_Style }
```

The format type and the format style allow further control over the formatting. The *format type* indicates what kind of argument (e.g., `number`, `date`, or `time`) is to be formatted, and the *format style* indicates how the argument will be formatted (e.g., `currency` format for a `number`, `short` format for a `date`). Legal combinations of format type and format style are shown in Table 18.16.

Table 18.16 *Format Type and Format Style Combinations for Format Elements*

Format type value	Format style value that can be specified for a format type
none	*none*
`number`	*none*, `integer`, `currency`, `percent`, *subformat pattern*
`date` *or* `time`	*none*, `short`, `medium`, `long`, `full`, *subformat pattern*
`choice` (p. 1145)	*subformat pattern*

The first variant of the format element (first row in Table 18.16) was used in the code earlier, where neither format type nor format style is specified. In this case, the text representation of the argument as defined by the `toString()` method is used in the pattern to create the final result. We will explore other combinations from Table 18.16 in this section.

When format type and format style values are used in a format element, an appropriate formatter is implicitly created to format the argument corresponding to that particular format element. The formatters created are instances of the formatter classes shown in Figure 18.1, p. 1115. For example, the format element `{3,time,short}` results in a locale-specific formatter being created implicitly by the following method call:

```
DateFormat.getTimeInstance(DateFormat.SHORT, getLocale())
```

This formatter will format a `java.util.Date` object (which represents both date and time) by extracting its time components and formatting them according to the format defined by `DateFormat.SHORT` and taking the locale into consideration. If a `MessageFormat` instance does not specify the locale, then the default locale is used.

Format elements that specify the format types `date` and `time` are compatible with the `java.util.Date` legacy class, with subsequent reliance on the `DateFormat` class to provide an appropriate formatter. This presents a slight problem when we want to format date and time values of the new Date and Time API. We will use the static methods of the utility class `ConvertToLegacyDate` (§17.8, p. 1088) to convert `LocalDate`, `LocalTime`, `LocalDateTime`, and `ZonedDateTime` objects to `Date` objects in order to leverage the format element handling functionality of the `MessageFormat` class.

The pattern that we saw earlier with just vanilla format elements (i.e., format elements with just the argument index) is now shown at (1) below to include the specification of the type and style of the format elements. A `MessageFormat` instance that is based on the pattern at (1) is created at (2) using a constructor of the `MessageFormat` class. The arguments that we want to format are included in the `Object` array at (3). Note that the element index in this array corresponds to an argument index in the format elements specified in the pattern. We use the methods from the utility class `ConvertToDate` to convert `LocalTime` and `LocalDate` values to `Date` values (§17.8, p. 1088). Finally, at (4), the argument array is passed for formatting to the `format()` method inherited by the `MessageFormat` class from its superclass `Format`. The code below also outlines the pertinent steps that are involved when using a `MessageFormat` instance to format compound messages.

```
// Specify the pattern:                                                    (1)
String pattern2 = "At {3,time,short} on {2,date,medium} Elvis landed at {0} "
               +  "and was greeted by {1,number,integer} fans.";

// Create a MessageFormat based on the given pattern:                      (2)
MessageFormat mf2 = new MessageFormat(pattern2);

// Create the array with the arguments to format:                          (3)
Object[] messageArguments = {
    "Honolulu",                                       // argument index 0
    3000,                                             // argument index 1
    ConvertToDate.ldToDate(LocalDate.of(1961,3,25)),  // argument index 2
    ConvertToDate.ltToDate(LocalTime.of(12,15))       // argument index 3
};
```

```
// Format the arguments:                                              (4)
String output2 = mf2.format(messageArguments);
System.out.println(output2);
```

Output from the code:

```
At 12:15 PM on Mar 25, 1961 Elvis landed at Honolulu and was greeted by 3,000 fans.
```

This output is different from the output we saw earlier when using vanilla format elements. In this case, appropriate formatters are implicitly created for each format element that apply the format style and take into consideration the locale (in this case, it is the default locale, which happens to be the US locale).

The following constructors defined by the `MessageFormat` class are used in the examples in this section:

```
MessageFormat(String pattern)
MessageFormat(String pattern, Locale locale)
```

Create a message formatter to apply the specified pattern. The created formatters will format or parse according to the default locale or the specified locale, respectively.

Selected methods are provided by the `MessageFormat` class, many of which are used in the examples in this section:

```
final String format(Object obj)       // Inherited from the Format class.
```

Returns a string that is the result of formatting the specified object. Passing an `Object[]` formats the elements of the array individually.

```
static String format(String pattern, Object... arguments)
```

This static method implicitly creates a one-time formatter with the given pattern, and returns the result of using it to format the variable arity arguments.

```
void setLocale(Locale locale)
```

Sets the locale to be used when creating subformats. Subformats already created are not affected.

```
Locale getLocale()
```

Returns the locale used by this formatter.

```
void applyPattern(String pattern)
```

Sets the pattern used by this formatter by parsing the pattern and creating the necessary subformats.

```
String toPattern()
```

Returns a pattern that represents the current state of this formatter.

```
void setFormat(int formatElementIndex, Format newFormat)
void setFormats(Format[] newFormats)
```

The first method sets the format to use for the format element indicated by the `formatElementIndex` in the previously set pattern string.

The second method sets the formats to use for the format elements in the previously set pattern string. Note that the order of formats in the `newFormats` array corresponds to *the order of format elements in the pattern string*. Contrast this method with the `setFormatsByArgumentIndex()` method.

More formats provided than needed by the pattern string are ignored. Fewer formats provided than needed by the pattern string results in only the provided formats being inserted.

```
void setFormatByArgumentIndex(int argumentIndex, Format newFormat)
void setFormatsByArgumentIndex(Format[] newFormats)
```

The first method sets the format to use for the format element indicated by the `argumentIndex` in the previously set pattern string.

The second method sets the formats to use for the format elements in the previously set pattern string. However, note that the order of formats in the `newFormats` array corresponds to *the order of elements in the arguments array* passed to the format methods or returned by the parse methods. Contrast this method with the `setFormats()` method.

Any argument index not used for any format element in the pattern string results in the corresponding new format being ignored.

Fewer formats being provided than needed results in only the formats for provided argument indices being replaced.

Formatting Compound Messages

If the application is intended for an international audience, we need to take the locale into consideration when formatting compound messages.

Example 18.8 illustrates formatting compound messages for different locales, where locale-sensitive data is contained in resource bundles (p. 1102). The program output shows stock information about an item according to the requested locale, showing how formatting of number, currency, date, and time is localized. A resource bundle file is created for each locale. The resource bundle for the US locale is shown in Example 18.8. The program output shows stock information for the US locale and the locale for Spain.

Example 18.8 *Formatting Compound Messages*

```
# File: StockInfoBundle.properties
pattern = Stock date: {3,time,short}, {4,date,long}\n\
          Item name: {0}\n\
          Item price: {1,number,currency}\n\
          Number of items: {2,number,integer}
item.name = Frozen pizza
item.price = 9.99
```

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

public class CompoundMessageFormatting {
  static void displayStockInfo(Locale requestedLocale) {                  // (1)
    System.out.println("Requested Locale: " + requestedLocale);

    // Fetch the relevant resource bundle:                                   (2)
    ResourceBundle bundle =
        ResourceBundle.getBundle("resources.StockInfoBundle", requestedLocale);

    // Create a formatter, given the pattern and the locale:                 (3)
    MessageFormat mf = new MessageFormat(bundle.getString("pattern"),
                                            requestedLocale);

    // Argument values:                                                      (4)
    String itemName = bundle.getString("item.name");
    double itemPrice = Double.parseDouble(bundle.getString("item.price"));
    int numOfItems = 1234;
    Date timeOnly = ConvertToLegacyDate.ltToDate(LocalTime.of(14,30));
    Date dateOnly = ConvertToLegacyDate.ldToDate(LocalDate.of(2021,3,1));

    // Create argument array:                                                (5)
    Object[] messageArguments = {
        itemName,       // {0}
        itemPrice,      // {1,number,currency}
        numOfItems,     // {2,number,integer}
        timeOnly,       // {3,time,short}
        dateOnly,       // {4,date,long}
    };

    // Apply the formatter to the arguments:                                 (6)
    String result = mf.format(messageArguments);
    System.out.println(result);
  }

  public static void main(String[] args) {
    displayStockInfo(Locale.US);
    System.out.println();
    displayStockInfo(new Locale("es", "ES"));
    System.out.println();
  }
}
```

Output from the program:

```
Requested Locale: en_US
Stock date: 2:30 PM, March 1, 2021
Item name: Frozen pizza
Item price: $9.99
Number of items: 1,234

Requested Locale: es_ES
Fecha de stock: 14:30, 1 de marzo de 2021
```

```
Nombre del árticulo: Pizza congelada
Precio del articulo: 8,99 ?
Número de artículos: 1.234
```

In Example 18.8, the method `displayStockInfo()` at (1) epitomizes the basic steps for using `MessageFormat` for formatting compound messages. The method is passed the requested locale for which the stock information should be displayed. It goes without saying that the stock information is printed only if the relevant resource bundle files can be found.

The `MessageFormat` instance created at (2) is locale-specific. The pattern is read from the resource bundle. It is composed of several lines, and appropriate format elements for each kind of value in the pattern are specified, as shown in the resource bundle for the US locale. The `MessageFormat` instance implicitly creates the necessary formatters for type and style specified in the format elements, and applies them to the corresponding arguments.

The arguments to format are set up at (4). We convert `LocalTime` and `LocalDate` instances to `Date` instances. Another solution is to convert these instances to their text representation beforehand using an appropriate `DateTimeFormatter`. It is also possible to set specific formatters for any argument that is to be formatted. We leave the curious reader to explore these solutions for formatting compound messages.

At (5), an `Object` array is initialized with the arguments. Finally, at (6), the `format()` method is invoked on the formatter, passing the argument array. Again note that the currency delimiter for `?` (euro) is a nbsp in the formatted output, as discussed earlier (p. 1116).

Keep in mind that a `MessageFormat` can be reused, with a new locale, a new pattern, and new arguments:

```
MessageFormat mf = new MessageFormat(previousPattern);
mf.setLocale(newLocale);
mf.applyPattern(newPattern);
String output = mf.format(newMessageArguments);
```

Conditional Formatting

Consider *contiguous half-open intervals* on the number line, as shown below. The (lower and upper) *limits* of these intervals can be any numbers, which will always be in ascending order as they are on the number line. Below we see two half-open intervals with limits 0.0, 1.0, and 2.0.

```
limits:       0.0        1.0         2.0
        <-----[----------)[----------)-----> number line
                         vi          vi+1
```

If a value v falls in the half-open interval $[v_i, v_{i+1})$, then we select the (lower) limit v_i. So, if v is any number in the half-open interval `[0.0, 1.0)`, the limit 0.0 is selected. Similarly, if v is any number in the half-open interval `[1.0, 2.0)`, the limit 1.0 will be selected, and so on.

From Table 18.16, we see that the format element having the format type `choice` requires a *subformat pattern*. This subformat pattern specifies a *choice format*. The general syntax of the choice format is shown below, where limit v_i and its associated subformat pattern f_i are separated by the hash sign (#). Each v_i#f_i constitutes *a choice*, and choices are separated by the vertical bar (`|`). Two consecutive limits constitute a half-open interval: $[v_i,\ v_{i+1})$. Note that the last choice uses `<`, meaning any value greater than or equal to v_n. Also, the limits $v_1, v_2, ..., v_n$ must be in ascending order for the choice format to work correctly.

v_1#f_1 | v_2#f_2 | ... | v_n<f_n

The choice format thus defines the limits and the corresponding choices. When a value is supplied for formatting, the choice format determines the limit, as explained above, and the formatter uses the choice associated with this limit—hence the term *conditional formatting*.

Below is a pattern that uses a choice format. The pattern's only format element has *argument index* 0 and *format type* `choice`, and its subformat pattern specifies a *choice format* (underlined below, but see also Table 18.16). The argument index `0` in the pattern designates the value that is used to determine the limit, and thereby, select the corresponding choice in the choice format.

```
"There {0,choice,0#are no bananas|1#is only one banana|\
                                          2<are {1,number,integer} bananas}."
```

The choice format above has three choices, separated by the vertical bar (`|`). Its limits are 0, 1, and 2. Each limit in a choice is associated with a subformat pattern. For limits 0 and 1, the subformat pattern is just text. The subformat pattern for limit 2 specifies a format element to format a number as an integer, as it expects an argument to be supplied for this format element. Note that the argument index in this embedded format element is 1.

Example 18.9 uses a pattern that includes the choice format above to illustrate handling of plurals in messages. Resource bundles for different locales contain the pattern with the choice format. An appropriate resource bundle for a locale is fetched at (1). The choice pattern is read from the resource bundle and a `MessageFormat` is created at (2), based on the choice pattern and the requested locale. For each locale, the formatter is tested for three cases, exemplified by the rows of two-dimensional array `messageArguments` at (3). Each row of arguments is passed to the `format()` method at (5) for formatting. The program output shows the arguments passed and the corresponding result.

In the argument matrix, the first row, `{0.5}`, specifies the argument `0.5`, which falls in the half-interval `[0, 1)`. Limit `0` is chosen, whose choice is selected. This results in the following pattern to be applied:

```
"There are no bananas."
```

Similarly for the second row, {1.5}, the limit is determined to be 1, resulting in the following pattern to be applied:

```
"There is only one banana."
```

For the third row, {2.5, 2}, the argument 2.5 determines the limit 2, resulting in the pattern below to be applied. The second element (value 2) in this row is the argument that is formatted according to the format element in this pattern.

```
"There are {1,number,integer} bananas."
```

We see from the program output that the correct singular or plural form is selected depending on the value of the argument passed to the formatter. It is instructive to experiment with passing different values to the formatter.

Example 18.9 *Using Choice Pattern*

```
# File: ChoiceBundle.properties
choice.pattern = There {0,choice,0#are no bananas|1#is only one banana|\
                                 2<are {1,number,integer} bananas}.
...
```

```
import java.text.*;
import java.util.*;

public class ChoicePatternUsage {

  static void displayMessages(Locale requestedLocale) {
    System.out.println("Requested Locale: " + requestedLocale);

    // Fetch the resource bundle:                                       (1)
    ResourceBundle bundle =
        ResourceBundle.getBundle("resources.ChoiceBundle", requestedLocale);

    // choice.pattern = There {0,choice,0#are no bananas|1#is only one banana
    //                                   |2#are {1,number,integer} bananas}.
    // Create formatter for specified choice pattern and locale:        (2)
    MessageFormat mf =
        new MessageFormat(bundle.getString("choice.pattern"), requestedLocale);

    // Create the message argument arrays:                              (3)
    Object[][] messageArguments = { {0.5}, {1.5}, {2.5, 2} };

    // Test the formatter with arguments:                               (4)
    for (int choiceNumber = 0;
         choiceNumber < messageArguments.length; choiceNumber++) {
      String result = mf.format(messageArguments[choiceNumber]);     // (5)
      System.out.printf("Arguments:%-10sResult: %s%n",
               Arrays.toString(messageArguments[choiceNumber]),result);
    }
  }

  public static void main(String[] args) {
    displayMessages(new Locale("en", "US"));
```

```
    System.out.println();
    displayMessages(new Locale("es", "ES"));
  }
}
```

Output from the program:

```
Requested Locale: en_US
Arguments:[0.5]     Result: There are no bananas.
Arguments:[1.5]     Result: There is only one banana.
Arguments:[2.5, 2]  Result: There are 2 bananas.

Requested Locale: es_ES
Arguments:[0.5]     Result: Ahí no hay plátanos.
Arguments:[1.5]     Result: Ahí es solo un plátano.
Arguments:[2.5, 2]  Result: Ahí son 2 plátanos.
```

Example 18.9 illustrates using a choice format for handling plurals. Example 18.10 illustrates an alternative approach where the choice format is created programmatically using the `ChoiceFormat` class.

The class `ChoiceFormat` provides the following constructors for creating a choice format either programmatically or using a pattern.

> `ChoiceFormat(double[] limits, String[] formats)`
>
> Creates a `ChoiceFormat` with the limits specified in ascending order and the corresponding formats.
>
> `ChoiceFormat(String newPattern)`
>
> Creates a `ChoiceFormat` with limits and corresponding formats based on the specified pattern string.

The `ChoiceFormat` class is locale-agnostic—that is, providing no locale-specific operations. However, the class `MessageFormat` implements locale-specific operations, and allows formats to be set for individual format elements in its pattern. By setting a `ChoiceFormat` as a format for a choice format element in a locale-sensitive `MessageFormat`, it is possible to achieve locale-specific formatting behavior in a `ChoiceFormat`. This approach is illustrated in Example 18.10. The numbers below correspond to the numbers on the lines in the source code for Example 18.10.

(1) The locale-specific resource bundle with resources for the pattern and for constructing the choice format is fetched. See the listing of the resource bundle file in Example 18.10.

(2) A locale-specific `MessageFormat` is created based on the pattern (`"There {0}."`) which is read from the resource bundle.

(3) To create a `ChoiceFormat` programmatically requires two equal-length arrays that contain the limits and the corresponding subformats, respectively. The `double` array `limits` is created with the limit values `0`, `1`, and `2`. A `String` array `messageFormats` is created by reading the values of the keys `none`, `singular`, and

plural from the resource bundle. These are the subformats corresponding to the limits that we saw in the choice format in Example 18.9.

(4) Given the arrays for the limits and the subformats, the ChoiceFormat constructor creates the choice format.

(5) and (6) One way to set customized formats for individual format elements in a MessageFormat is by first creating an array of Format (superclass of format classes in java.text package). The order of the formats in this array must be the same as either *the order of format elements in the pattern string* or *the order of elements in the argument array passed to the format methods.* There is only one format element in the pattern ("There {0}."). The array of Format thus has only one element which is the choice format for this format element, and therefore, both orders are valid in this case. We can use either the setFormats() or the setFormatsByArgumentIndex() method, as shown at (6a) and (6b), respectively. Either method will associate the choice format with the format element having the argument index 0 in the pattern.

However, the format for the format element at the argument index 0 in the pattern can easily be set by calling the setFormat() method, as shown at (6c).

(7) through (9) The behavior of the program is exactly the same as for Example 18.9. Experimenting with other limits and arguments is recommended.

Example 18.10 *Using the* ChoiceFormat *Class*

```
# File: ChoiceBundle.properties
...
pattern = There {0}.
none = are no bananas
singular = is only one banana
plural = are {1,number,integer} bananas
```

```
import java.text.*;
import java.util.*;

public class ChoiceFormatUsage {

  static void displayMessages(Locale requestedLocale)  {
    System.out.println("Requested locale: " + requestedLocale);

    // Fetch the resource bundle:                                      (1)
    ResourceBundle bundle =
        ResourceBundle.getBundle("resources.ChoiceBundle", requestedLocale);

    // Create formatter for specified pattern and locale:              (2)
    MessageFormat mf = new MessageFormat(
        bundle.getString("pattern"),      // pattern = There {0}.
        requestedLocale
        );
```

```
    // Create the limits and the formats arrays:                              (3)
    double[] limits = {0,1,2};                                             // (3a)
    String [] grammarFormats = {                                           // (3b)
        bundle.getString("none"),       // none = are no bananas
        bundle.getString("singular"),   // singular = is only one banana
        bundle.getString("plural")      // plural = are {1,number,integer} bananas
    };

    // Create the choice format:                                              (4)
    ChoiceFormat choiceForm = new ChoiceFormat(limits, grammarFormats);

    // Create the formats:                                                    (5)
    Format[] formats = {choiceForm};

    // Set the formats in the formatter:                                      (6)
    mf.setFormats(formats);                                                // (6a)
//  messageForm.setFormatsByArgumentIndex(formats);                        // (6b)
//  mf.setFormat(0, choiceForm);                                           // (6c)

    // Create the arguments arrays:                                           (7)
    Object[][] messageArguments = { {0.5}, {1.5}, {2.5, 2} };

    // Test the formatter with arguments:                                     (8)
    for (int choiceNumber = 0;
         choiceNumber < messageArguments.length; choiceNumber++) {
      String result = mf.format(messageArguments[choiceNumber]);          // (9)
      System.out.printf("Arguments:%-10sResult: %s%n",
          Arrays.toString(messageArguments[choiceNumber]),result);    }
  }

  public static void main(String[] args)   {
    displayMessages(new Locale("en", "US"));
    System.out.println();
    displayMessages(new Locale("es", "ES"));
  }
}
```

Output from the program:

```
Requested Locale: en_US
Arguments:[0.5]     Result: There are no bananas.
Arguments:[1.5]     Result: There is only one banana.
Arguments:[2.5, 2]  Result: There are 2 bananas.

Requested locale: es_ES
Arguments:[0.5]     Result: Ahí no hay plátanos.
Arguments:[1.5]     Result: Ahí es solo un plátano.
Arguments:[2.5, 2]  Result: Ahí son 2 plátanos.
```

Parsing Values Using Patterns

An instance of the `MessageFormat` class can be used both for formatting and for parsing of values. The class provides the parse() methods for parsing text. We will primarily use the one-argument method shown below to demonstrate parsing with the `MessageFormat` class.

```
Object[] parse(String source) throws ParseException
Object[] parse(String source, ParsePosition position)
```

The first method parses text from the beginning of the string `source` to return an `Object` array with the parsed values.

The second method parses from the `position` in the string `source` where the parsing should start.

Both methods may not use the entire text of the given string `source`.

The `parse()` methods return an `Object` array with the parsed values, which means that the parsed values must be extracted from the array and cast to the appropriate type in order to compute with them. The cast to the appropriate type must be done safely—typically by determining the type with the `instanceof` operator before applying the cast.

As the one-argument `parse()` method throws a checked `ParseException` when an error occurs under parsing, the code snippets in this subsection must be executed in a context that handles this exception.

The round trip of formatting values and then parsing the formatted result to obtain the same values back is illustrated below. It should not come as a surprise that this round trip always works, as demonstrated by the code below. We see that the argument of type `double` that was formatted and the value of type `double` that was parsed according to the pattern at (1) are equal.

```
String pattern = "foo {0,number,currency} bar";                    // (1)
MessageFormat mfp = new MessageFormat(pattern, Locale.US);
Object[] arguments = new Object[] {10.99};            // [10.99]
String formattedResult = mfp.format(arguments);       // "foo $10.99 bar"
Object[] parsedResult = mfp.parse(formattedResult);   // [10.99]
System.out.println((double)arguments[0] == (double)parsedResult[0]); // true
```

A vanilla format element, {*i*}, in a pattern matches a string in the source, as no format type or style is specified for such a format element. The source at (3) matches the format element in the pattern at (2), as is evident from the parse result at (4).

```
String patternA = "{0}";                                           // (2)
MessageFormat mfpA = new MessageFormat(patternA, Locale.US);
Object[] parsedResultA = mfpA.parse("One Two Three");              // (3)
System.out.println(parsedResultA[0] instanceof String);   // true
System.out.println(Arrays.toString(parsedResultA));       // (4) [One Two Three]
```

Any text in a pattern must be matched verbatim in the source. Note how the text in the pattern at (5) is matched in the source at (6), with the string "2" being returned as the result of parsing.

```
String patternB = "One {0} Three";                                 // (5)
MessageFormat mfpB = new MessageFormat(patternB, Locale.US);
Object[] parsedResultB = mfpB.parse("One 2 Three");                // (6)
System.out.println(parsedResultB[0].equals("2"));                  // true
System.out.println(Arrays.toString(parsedResultB));                // [2]
```

The code below illustrates that the `parse()` method may not use the entire text in the source, only what is necessary to declare a match. Parsing stops after the word `Three` in the source at (7) when a match for the pattern has been found in the source.

```
Object[] parsedResultC = mfpB.parse("One 2 Three whatever");          // (7)
System.out.println(Arrays.toString(parsedResultC));                   // [2]
```

In order for the parse to succeed, the source must contain text that is compatible with the pattern so that it can be parsed according to any type or style specified in the format elements defined in the pattern. At (8) below, the pattern contains two format elements that require an integer and a currency value specified according to the US locale. The source at (9) meets the requirements to match the pattern. The parse result shows that an `int` value and a `double` value were parsed.

```
String patternD = "foo {0,number,integer} {1,number,currency} bar";  // (8)
MessageFormat mfpD = new MessageFormat(patternD, Locale.US);
Object[] parsedResultD = mfpD.parse("foo 2021 $64.99 bar");           // (9)
System.out.println(Arrays.toString(parsedResultD));          // [2021, 64.99]
```

We have seen earlier in this chapter that a *nbsp* is necessary in the source in order to parse certain currencies (p. 1119). That is also the case when parsing such currency values with the `MessageFormat` class, and is illustrated below for the Norwegian locale. Unless a *nbsp* is the delimiter between the currency symbol and the first digit of the amount, as shown at (10b), parsing will result in a checked `ParseException` being thrown.

```
Locale locale = new Locale("no", "NO");
String pattern1 = "foo {0,number,currency} bar";
MessageFormat mfp1 = new MessageFormat(pattern1, locale);
// String parseSource = "foo kr 10,99 bar";              // (10a) ParseException
String parseSource = "foo kr\u00a010,99 bar";            // (10b) nbsp. OK.
Object[] parsedResults = mfp1.parse(parseSource);
if (parsedResults[0] instanceof Double dValue) {
  System.out.println(dValue);                            // 10.99
}
```

Additional Support for Formatting in the Java SE Platform APIs

In this subsection we mention additional support for formatting values that is provided by various classes in the Java SE Platform APIs.

The class `java.util.Formatter` provides the core support for *formatted text representation* of primitive values and objects through its overloaded `format()` methods. See the Java SE Platform API documentation for the nitty-gritty details on how to specify the format.

```
format(String format, Object... args)
format(Locale loc, String format, Object... args)
```

Write a string that is a result of applying the specified `format` string to the values in the variable arity parameter `args`. The resulting string is written to the *destination object* that is associated with the formatter.

The destination object of a formatter is specified when the formatter is created. The destination object can, for example, be a `String`, a `StringBuilder`, a file, or any `OutputStream`.

Return the current formatter.

The classes `java.io.PrintStream` and `java.io.PrintWriter` also provide an overloaded `format()` method with the same signature for formatted output. These streams use an associated `Formatter` that sends the output to the `PrintStream` or the `PrintWriter`, respectively. We have used the `printf()` method of the `PrintStream` class for sending output to the terminal every time this method is invoked on the `System.out` field (§1.9, p. 24).

The `String` class also provides an analogous `format()` method, but it is static. This method also uses an associated `Formatter` for formatting purposes. Unlike the `format()` method of the classes mentioned earlier, this static method returns the resulting string after formatting the values (§8.4, p. 457).

The `java.io.Console` only provides the first form of the `format()` and the `printf()` methods (without the locale specification). These methods too use an associated `Formatter`. They write the resulting string to the console's output stream, and return the current console (§20.4, p. 1256).

Review Questions

18.5 Given the following code:

```
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.Locale;

public class DecimalNumberPatternsRQ {
  public static void main(String[] args) {
    // (1) INSERT CODE HERE
    double value = 0.456;
    DecimalFormat df = (DecimalFormat) NumberFormat.getNumberInstance(Locale.US);
    df.applyPattern(pattern);
    String output = df.format(value);
    System.out.printf("|%s|", output);
  }
}
```

Which code, when inserted independently at (1), will result in the following output: `|.46|`?

Select the five correct answers.

(a) `String pattern = ".00";`
(b) `String pattern = ".##";`
(c) `String pattern = ".0#";`
(d) `String pattern = "#.00";`