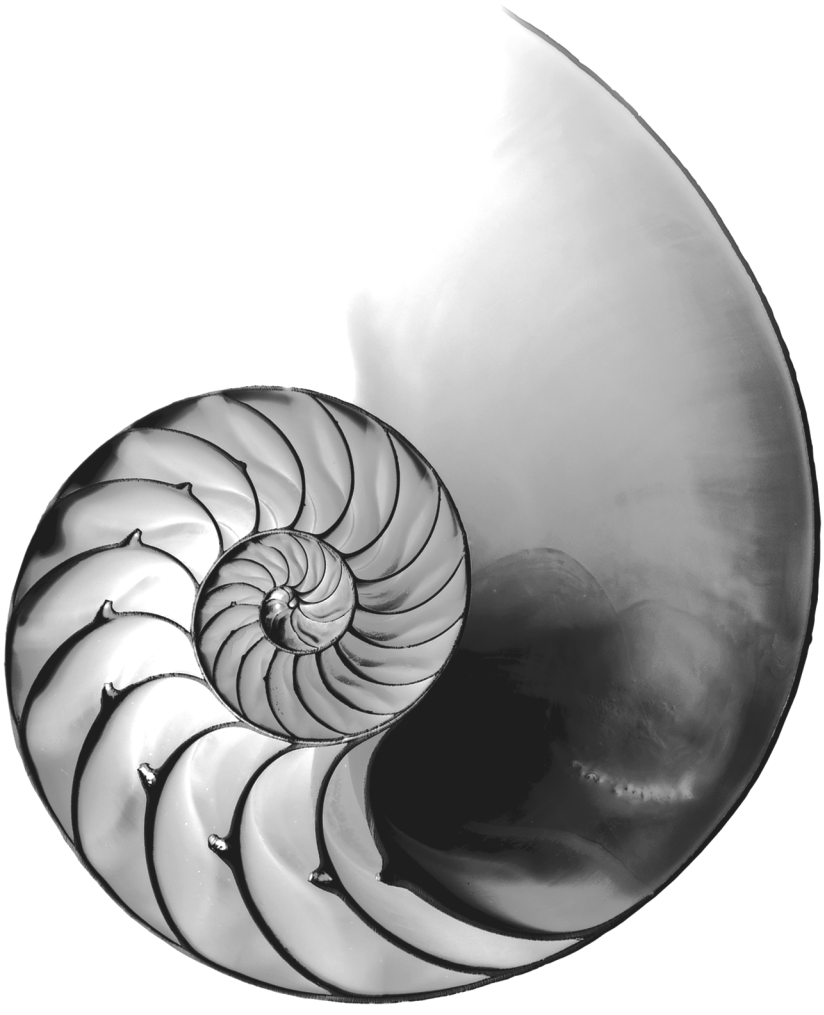
I

Formatted Output



Objectives

In this appendix you'll:

- Use `printf` formatting.
- Print with field widths and precisions.
- Use formatting flags in the `printf` format string.
- Print with an argument index.
- Output literals and escape sequences.
- Format output with class `Formatter`.

Outline

I.1 Introduction

In this appendix, we discuss the formatting features of method `printf` and class `Formatter` (package `java.util`). Class **`Formatter`** formats and outputs data to a specified destination, such as a string or a file output stream. Many features of `printf` were discussed earlier in the text. This appendix summarizes those features and introduces others, such as displaying date and time data in various formats, reordering output based on the index of the argument and displaying numbers and strings with various flags.

I.2 Streams

Input and output are usually performed with streams, which are sequences of bytes. In input operations, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection) to main memory. In output operations, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection).

When program execution begins, three streams are created. The standard input stream typically reads bytes from the keyboard, and the standard output stream typically outputs characters to a command window. A third stream, the **standard error stream** (`System.err`), typically outputs characters to a command window and is used to output error messages so they can be viewed immediately. Operating systems typically allow these streams to be redirected to other devices. Streams are discussed in detail in Chapter 15, Files, Input/Output Streams, NIO and XML Serialization.

I.3 Formatting Output with `printf`

Precise output formatting is accomplished with `printf`. Java borrowed (and enhanced) this feature from the C programming language. Method `printf` can perform the following formatting capabilities, each of which is discussed in this appendix:

1. Rounding floating-point values to an indicated number of decimal places.
2. Aligning a column of numbers with decimal points appearing one above the other.
3. Right justification and left justification of outputs.
4. Inserting literal characters at precise locations in a line of output.

5. Representing floating-point numbers in exponential format.
6. Representing integers in octal and hexadecimal format.
7. Displaying all types of data with fixed-size field widths and precisions.
8. Displaying dates and times in various formats.

Every call to `printf` supplies as the first argument a **format string** that describes the output format. The format string may consist of **fixed text** and **format specifiers**. Fixed text is output by `printf` just as it would be output by `System.out` methods `print` or `println`. Each format specifier is a placeholder for a value and specifies the type of data to output. Format specifiers also may include optional formatting information.

In the simplest form, each format specifier begins with a percent sign (%) and is followed by a **conversion character** that represents the data type of the value to output. For example, the format specifier `%s` is a placeholder for a `string`, and the format specifier `%d` is a placeholder for an `int` value. The optional formatting information, such as an argument index, flags, field width and precision, is specified between the percent sign and the conversion character. We demonstrate each of these capabilities.

I.4 Printing Integers

Figure I.1 describes the **integer conversion characters**. (See Appendix J for an overview of the binary, octal, decimal and hexadecimal number systems.) Figure I.2 uses each to print an integer. In lines 9–10, the plus sign is not displayed by default, but the minus sign is. Later in this appendix (Fig. I.14) we'll see how to force plus signs to print.

Conversion character	Description
d	Display a decimal (base 10) integer.
o	Display an octal (base 8) integer.
x or **X**	Display a hexadecimal (base 16) integer. `X` uses uppercase letters.

Fig. I.1 | Integer conversion characters.

```
1  // Fig. I.2: IntegerConversionTest.java
2  // Using the integer conversion characters.
3
4  public class IntegerConversionTest
5  {
6     public static void main(String[] args)
7     {
8        System.out.printf("%d\n", 26);
9        System.out.printf("%d\n", +26);
10       System.out.printf("%d\n", -26);
11       System.out.printf("%o\n", 26);
12       System.out.printf("%x\n", 26);
```

Fig. I.2 | Using the integer conversion characters. (Part 1 of 2.)

```
13          System.out.printf("%X\n", 26);
14       } // end main
15    } // end class IntegerConversionTest
```

```
26
26
-26
32
1a
1A
```

Fig. I.2 | Using the integer conversion characters. (Part 2 of 2.)

The `printf` method has the form

```
printf(format-string, argument-list);
```

where *format-string* describes the output format, and the optional *argument-list* contains the values that correspond to each format specifier in *format-string*. There can be many format specifiers in one format string.

Each format string in lines 8–10 specifies that `printf` should output a decimal integer (`%d`) followed by a newline character. At the format specifier's position, `printf` substitutes the value of the first argument after the format string. If the format string contains multiple format specifiers, at each subsequent format specifier's position `printf` substitutes the value of the next argument in the argument list. The `%o` format specifier in line 11 outputs the integer in octal format. The `%x` format specifier in line 12 outputs the integer in hexadecimal format. The `%X` format specifier in line 13 outputs the integer in hexadecimal format with capital letters.

I.5 Printing Floating-Point Numbers

Figure I.3 describes the floating-point conversions. The **conversion characters `e`** and **`E`** display floating-point values in **computerized scientific notation** (also called **exponential notation**). Exponential notation is the computer equivalent of the scientific notation used in mathematics. For example, the value `150.4582` is represented in scientific notation in mathematics as

$$1.504582 \times 10^2$$

and is represented in exponential notation as

```
1.504582e+02
```

in Java. This notation indicates that `1.504582` is multiplied by `10` raised to the second power (`e+02`). The `e` stands for "exponent."

Values printed with the conversion characters `e`, `E` and `f` are output with six digits of precision to the right of the decimal point by default (e.g., `1.045921`)—other precisions must be specified explicitly. For values printed with the conversion character `g`, the precision represents the total number of digits displayed, excluding the exponent. The default is six digits (e.g., `12345678.9` is displayed as `1.23457e+07`). **Conversion character `f`** always prints at least one digit to the left of the decimal point. Conversion characters `e` and `E` print

Conversion character	Description
`e` or `E`	Display a floating-point value in exponential notation. Conversion character `E` displays the output in uppercase letters.
`f`	Display a floating-point value in decimal format.
`g` or `G`	Display a floating-point value in either the floating-point format `f` or the exponential format `e` based on the magnitude of the value. If the magnitude is less than 10^{-3}, or greater than or equal to 10^{7}, the floating-point value is printed with `e` (or `E`). Otherwise, the value is printed in format `f`. When conversion character `G` is used, the output is displayed in uppercase letters.
`a` or `A`	Display a floating-point number in hexadecimal format. Conversion character `A` displays the output in uppercase letters.

Fig. I.3 | Floating-point conversion characters.

lowercase `e` and uppercase `E` preceding the exponent and always print exactly one digit to the left of the decimal point. Rounding occurs if the value being formatted has more significant digits than the precision.

Conversion character `g` (or `G`) prints in either `e` (`E`) or `f` format, depending on the floating-point value. For example, the values `0.0000875`, `87500000.0`, `8.75`, `87.50` and `875.0` are printed as `8.750000e-05`, `8.750000e+07`, `8.750000`, `87.500000` and `875.000000` with the conversion character `g`. The value `0.0000875` uses `e` notation because the magnitude is less than `10-3`. The value `87500000.0` uses `e` notation because the magnitude is greater than 10^{7}. Figure I.4 demonstrates the floating-point conversion characters.

```
1  // Fig. I.4: FloatingNumberTest.java
2  // Using floating-point conversion characters.
3
4  public class FloatingNumberTest
5  {
6     public static void main(String[] args)
7     {
8        System.out.printf("%e\n", 12345678.9);
9        System.out.printf("%e\n", +12345678.9 );
10       System.out.printf("%e\n", -12345678.9);
11       System.out.printf("%E\n", 12345678.9);
12       System.out.printf("%f\n", 12345678.9);
13       System.out.printf("%g\n", 12345678.9);
14       System.out.printf("%G\n", 12345678.9);
15    } // end main
16 } // end class FloatingNumberTest
```

```
1.234568e+07
1.234568e+07
-1.234568e+07
```

Fig. I.4 | Using floating-point conversion characters. (Part 1 of 2.)

```
1.234568E+07
12345678.900000
1.23457e+07
1.23457E+07
```

Fig. I.4 | Using floating-point conversion characters. (Part 2 of 2.)

I.6 Printing Strings and Characters

The `c` and `s` conversion characters print individual characters and strings, respectively. **Conversion characters `c`** and **`C`** require a `char` argument. **Conversion characters `s`** and **`S`** can take a `String` or any `Object` as an argument. When conversion characters `C` and `S` are used, the output is displayed in uppercase letters. Figure I.5 displays characters, strings and objects with conversion characters `c` and `s`. Autoboxing occurs at line 9 when an `int` constant is assigned to an `Integer` object. Line 15 outputs an `Integer` argument with the conversion character `s`, which implicitly invokes the `toString` method to get the integer value. You can also output an `Integer` object using the `%d` format specifier. In this case, the `int` value in the `Integer` object will be unboxed and output.

Common Programming Error I.1

Using `%c` to print a `String` causes an `IllegalFormatConversionException`—a `String` cannot be converted to a character.

```
1  // Fig. I.5: CharStringConversion.java
2  // Using character and string conversion characters.
3  public class CharStringConversion
4  {
5     public static void main(String[] args)
6     {
7        char character = 'A';  // initialize char
8        String string = "This is also a string";  // String object
9        Integer integer = 1234;  // initialize integer (autoboxing)
10
11       System.out.printf("%c\n", character);
12       System.out.printf("%s\n", "This is a string");
13       System.out.printf("%s\n", string);
14       System.out.printf("%S\n", string);
15       System.out.printf("%s\n", integer); // implicit call to toString
16    } // end main
17 } // end class CharStringConversion
```

```
A
This is a string
This is also a string
THIS IS ALSO A STRING
1234
```

Fig. I.5 | Using character and string conversion characters.

I.7 Printing Dates and Times

The **conversion character** `t` (or `T`) is used to print dates and times in various formats. It's always followed by a **conversion suffix character** that specifies the date and/or time format. When conversion character `T` is used, the output is displayed in uppercase letters. Figure I.6 lists the common conversion suffix characters for formatting **date and time compositions** that display both the date and the time. Figure I.7 lists the common conversion suffix characters for formatting dates. Figure I.8 lists the common conversion suffix characters for formatting times. For the complete list of conversion suffix characters, visit `http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html`.

Conversion suffix character	Description
`c`	Display date and time formatted as `day month date hour:minute:second time-zone year` with three characters for `day` and `month`, two digits for `date`, `hour`, `minute` and `second` and four digits for `year`—for example, `Wed Mar 03 16:30:25 GMT-05:00 2004`. The 24-hour clock is used. `GMT-05:00` is the time zone.
`F`	Display date formatted as `year-month-date` with four digits for the `year` and two digits each for the `month` and `date` (e.g., `2004-05-04`).
`D`	Display date formatted as `month/day/year` with two digits each for the `month`, `day` and `year` (e.g., `03/03/04`).
`r`	Display time in 12-hour format as `hour:minute:second AM|PM` with two digits each for the `hour`, `minute` and `second` (e.g., `04:30:25 PM`).
`R`	Display time formatted as `hour:minute` with two digits each for the `hour` and `minute` (e.g., `16:30`). The 24-hour clock is used.
`T`	Display time as `hour:minute:second` with two digits for the `hour`, `minute` and `second` (e.g., `16:30:25`). The 24-hour clock is used.

Fig. I.6 | Date and time composition conversion suffix characters.

Conversion suffix character	Description
`A`	Display full name of the day of the week (e.g., `Wednesday`).
`a`	Display the three-character name of the day of the week (e.g., `Wed`).
`B`	Display full name of the month (e.g., `March`).
`b`	Display the three-character short name of the month (e.g., `Mar`).
`d`	Display the day of the month with two digits, padding with leading zeros as necessary (e.g., `03`).
`m`	Display the month with two digits, padding with leading zeros as necessary (e.g., `07`).

Fig. I.7 | Date formatting conversion suffix characters. (Part 1 of 2.)

Conversion suffix character	Description
`e`	Display the day of month without leading zeros (e.g., 3).
`Y`	Display the year with four digits (e.g., 2004).
`y`	Display the last two digits of the year with leading zeros (e.g., 04).
`j`	Display the day of the year with three digits, padding with leading zeros as necessary (e.g., 016).

Fig. I.7 | Date formatting conversion suffix characters. (Part 2 of 2.)

Conversion suffix character	Description
`H`	Display hour in 24-hour clock with a leading zero as necessary (e.g., 16).
`I`	Display hour in 12-hour clock with a leading zero as necessary (e.g., 04).
`k`	Display hour in 24-hour clock without leading zeros (e.g., 16).
`l`	Display hour in 12-hour clock without leading zeros (e.g., 4).
`M`	Display minute with a leading zero as necessary (e.g., 06).
`S`	Display second with a leading zero as necessary (e.g., 05).
`Z`	Display the abbreviation for the time zone (e.g., EST, stands for Eastern Standard Time, which is 5 hours behind Greenwich Mean Time).
`p`	Display morning or afternoon marker in lowercase (e.g., pm).
`P`	Display morning or afternoon marker in uppercase (e.g., PM).

Fig. I.8 | Time formatting conversion suffix characters.

Figure I.9 uses the conversion characters `t` and `T` with the conversion suffix characters to display dates and times in various formats. Conversion character `t` requires the corresponding argument to be a date or time of type `long`, `Long`, **`Calendar`** (package `java.util`) or **`Date`** (package `java.util`)—objects of each of these classes can represent dates and times. Class `Calendar` is preferred for this purpose because some constructors and

```
1  // Fig. I.9: DateTimeTest.java
2  // Formatting dates and times with conversion characters t and T.
3  import java.util.Calendar;
4
5  public class DateTimeTest
6  {
7     public static void main(String[] args)
8     {
9        // get current date and time
10       Calendar dateTime = Calendar.getInstance();
```

Fig. I.9 | Formatting dates and times with conversion characters `t` and `T`. (Part 1 of 2.)

```
11
12         // printing with conversion characters for date/time compositions
13         System.out.printf("%tc\n", dateTime);
14         System.out.printf("%tF\n", dateTime);
15         System.out.printf("%tD\n", dateTime);
16         System.out.printf("%tr\n", dateTime);
17         System.out.printf("%tT\n", dateTime);
18
19         // printing with conversion characters for date
20         System.out.printf("%1$tA, %1$tB %1$td, %1$tY\n", dateTime);
21         System.out.printf("%1$TA, %1$TB %1$Td, %1$TY\n", dateTime);
22         System.out.printf("%1$ta, %1$tb %1$te, %1$ty\n", dateTime);
23
24         // printing with conversion characters for time
25         System.out.printf("%1$tH:%1$tM:%1$tS\n", dateTime);
26         System.out.printf("%1$tZ %1$tI:%1$tM:%1$tS %tP", dateTime);
27      } // end main
28   } // end class DateTimeTest
```

```
Wed Feb 25 15:00:22 EST 2009
2009-02-25
02/25/09
03:00:22 PM
15:00:22
Wednesday, February 25, 2009
WEDNESDAY, FEBRUARY 25, 2009
Wed, Feb 25, 09
15:00:22
EST 03:00:22 PM
```

Fig. I.9 | Formatting dates and times with conversion characters `t` and `T`. (Part 2 of 2.)

methods in class `Date` are replaced by those in class `Calendar`. Line 10 invokes `static` method **`getInstance`** of `Calendar` to obtain a calendar with the current date and time. Lines 13–17, 20–22 and 25–26 use this `Calendar` object in `printf` statements as the value to be formatted with conversion character `t`. Lines 20–22 and 25–26 use the optional **argument index** (`"1$"`) to indicate that all format specifiers in the format string use the first argument after the format string in the argument list. You'll learn more about argument indices in Section I.11. Using the argument index eliminates the need to repeatedly list the same argument.

I.8 Other Conversion Characters

The remaining conversion characters are **`b`**, **`B`**, **`h`**, **`H`**, **`%`** and **`n`**. These are described in Fig. I.10. Lines 9–10 of Fig. I.11 use `%b` to print the value of `boolean` (or `Boolean`) values `false` and `true`. Line 11 associates a `String` to `%b`, which returns `true` because it's not `null`. Line 12 associates a `null` object to `%B`, which displays `FALSE` because `test` is `null`. Lines 13–14 use `%h` to print the string representations of the hash-code values for strings `"hello"` and `"Hello"`. These values could be used to store or locate the strings in a `Hashtable` or `HashMap` (both discussed in Chapter 16, Generic Collections). The hash-code values for these two strings differ, because one string starts with a lowercase letter and the

other with an uppercase letter. Line 15 uses `%H` to print `null` in uppercase letters. The last two `printf` statements (lines 16–17) use `%%` to print the `%` character in a string and `%n` to print a platform-specific line separator.

Conversion character	Description
`b` or `B`	Print `"true"` or `"false"` for the value of a `boolean` or `Boolean`. These conversion characters can also format the value of any reference. If the reference is non-`null`, `"true"` is output; otherwise, `"false"`. When conversion character `B` is used, the output is displayed in uppercase letters.
`h` or `H`	Print the string representation of an object's hash-code value in hexadecimal format. If the corresponding argument is `null`, `"null"` is printed. When conversion character `H` is used, the output is displayed in uppercase letters.
`%`	Print the percent character.
`n`	Print the platform-specific line separator (e.g., `\r\n` on Windows or `\n` on UNIX/LINUX).

Fig. I.10 | Other conversion characters.

```
1  // Fig. I.11: OtherConversion.java
2  // Using the b, B, h, H, % and n conversion characters.
3
4  public class OtherConversion
5  {
6     public static void main(String[] args)
7     {
8        Object test = null;
9        System.out.printf("%b\n", false);
10       System.out.printf("%b\n", true);
11       System.out.printf("%b\n", "Test");
12       System.out.printf("%B\n", test);
13       System.out.printf("Hashcode of \"hello\" is %h\n", "hello");
14       System.out.printf("Hashcode of \"Hello\" is %h\n", "Hello");
15       System.out.printf("Hashcode of null is %H\n", test);
16       System.out.printf("Printing a %% in a format string\n");
17       System.out.printf("Printing a new line %nnext line starts here");
18    } // end main
19 } // end class OtherConversion
```

```
false
true
true
FALSE
Hashcode of "hello" is 5e918d2
```

Fig. I.11 | Using the `b`, `B`, `h`, `H`, `%` and `n` conversion characters. (Part 1 of 2.)

```
Hashcode of "Hello" is 42628b2
Hashcode of null is NULL
Printing a % in a format string
Printing a new line
next line starts here
```

Fig. I.11 | Using the `b`, `B`, `h`, `H`, `%` and `n` conversion characters. (Part 2 of 2.)

Common Programming Error I.2

Trying to print a literal percent character using `%` rather than `%%` in the format string might cause a difficult-to-detect logic error. When `%` appears in a format string, it must be followed by a conversion character in the string. The single percent could accidentally be followed by a legitimate conversion character, thus causing a logic error.

I.9 Printing with Field Widths and Precisions

The size of a field in which data is printed is specified by a **field width**. If the field width is larger than the data being printed, the data is right justified in that field by default. We discuss left justification in Section I.10. You insert an integer representing the field width between the `%` and the conversion character (e.g., `%4d`) in the format specifier. Figure I.12 prints two groups of five numbers each, right justifying those numbers that contain fewer digits than the field width. The field width is increased to print values wider than the field and that the minus sign for a negative value uses one character position in the field. Also, if no field width is specified, the data prints in exactly as many positions as it needs. Field widths can be used with all format specifiers except the line separator (`%n`).

```
1  // Fig. I.12: FieldWidthTest.java
2  // Right justifying integers in fields.
3
4  public class FieldWidthTest
5  {
6     public static void main(String[] args)
7     {
8        System.out.printf("%4d\n", 1);
9        System.out.printf("%4d\n", 12);
10       System.out.printf("%4d\n", 123);
11       System.out.printf("%4d\n", 1234);
12       System.out.printf("%4d\n\n", 12345); // data too large
13
14       System.out.printf("%4d\n", -1);
15       System.out.printf("%4d\n", -12);
16       System.out.printf("%4d\n", -123);
17       System.out.printf("%4d\n", -1234); // data too large
18       System.out.printf("%4d\n", -12345); // data too large
19    } // end main
20 } // end class RightJustifyTest
```

Fig. I.12 | Right justifying integers in fields. (Part 1 of 2.)

```
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
```

Fig. I.12 | Right justifying integers in fields. (Part 2 of 2.)

Common Programming Error I.3

Not providing a sufficiently large field width to handle a value to be printed can offset other data being printed and produce confusing outputs. Know your data!

Method `printf` also provides the ability to specify the precision with which data is printed. Precision has different meanings for different types. When used with floating-point conversion characters `e` and `f`, the precision is the number of digits that appear after the decimal point. When used with conversion characters `g`, `a` or `A`, the precision is the maximum number of significant digits to be printed. When used with conversion character `s`, the precision is the maximum number of characters to be written from the string. To use precision, place between the percent sign and the conversion specifier a decimal point (`.`) followed by an integer representing the precision. Figure I.13 demonstrates the use of precision in format strings. When a floating-point value is printed with a precision smaller than the original number of decimal places in the value, the value is rounded. Also, the format specifier `%.3g` indicates that the total number of digits used to display the floating-point value is 3. Because the value has three digits to the left of the decimal point, the value is rounded to the ones position.

The field width and the precision can be combined by placing the field width, followed by a decimal point, followed by a precision between the percent sign and the conversion character, as in the statement

```
printf("%9.3f", 123.456789);
```

which displays `123.457` with three digits to the right of the decimal point right justified in a nine-digit field—this number will be preceded in its field by two blanks.

```
1  // Fig. I.13: PrecisionTest.java
2  // Using precision for floating-point numbers and strings.
3  public class PrecisionTest
4  {
5     public static void main(String[] args)
6     {
7        double f = 123.94536;
8        String s = "Happy Birthday";
```

Fig. I.13 | Using precision for floating-point numbers and strings. (Part 1 of 2.)

```
9
10         System.out.printf("Using precision for floating-point numbers\n");
11         System.out.printf("\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f);
12
13         System.out.printf("Using precision for strings\n");
14         System.out.printf("\t%.11s\n", s);
15      } // end main
16   } // end class PrecisionTest
```

```
Using precision for floating-point numbers
        123.945
        1.239e+02
        124

Using precision for strings
        Happy Birth
```

Fig. I.13 | Using precision for floating-point numbers and strings. (Part 2 of 2.)

I.10 Using Flags in the printf Format String

Various flags may be used with method printf to supplement its output formatting capabilities. Seven flags are available for use in format strings (Fig. I.14).

Flag	Description
- (minus sign)	Left justify the output within the specified field.
+ (plus sign)	Display a plus sign preceding positive values and a minus sign preceding negative values.
space	Print a space before a positive value not printed with the + flag.
#	Prefix 0 to the output value when used with the octal conversion character o. Prefix 0x to the output value when used with the hexadecimal conversion character x.
0 (zero)	Pad a field with leading zeros.
, (comma)	Use the locale-specific thousands separator (i.e., ',' for U.S. locale) to display decimal and floating-point numbers.
(	Enclose negative numbers in parentheses.

Fig. I.14 | Format string flags.

To use a flag in a format string, place it immediately to the right of the percent sign. Several flags may be used in the same format specifier. Figure I.15 demonstrates right justification and left justification of a string, an integer, a character and a floating-point number. Line 9 serves as a counting mechanism for the screen output.

Figure I.16 prints a positive number and a negative number, each with and without the **+ flag**. The minus sign is displayed in both cases, the plus sign only when the + flag is used.

```
1  // Fig. I.15: MinusFlagTest.java
2  // Right justifying and left justifying values.
3
4  public class MinusFlagTest
5  {
6     public static void main(String[] args)
7     {
8        System.out.println("Columns:");
9        System.out.println("01234567890123456789012345678901234567890123456789\n");
10       System.out.printf("%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23);
11       System.out.printf(
12          "%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23);
13    } // end main
14 } // end class MinusFlagTest
```

```
Columns:
01234567890123456789012345678901234567890123456789

     hello         7         a  1.230000

hello     7         a         1.230000
```

Fig. I.15 | Right justifying and left justifying values.

```
1  // Fig. I.16: PlusFlagTest.java
2  // Printing numbers with and without the + flag.
3
4  public class PlusFlagTest
5  {
6     public static void main(String[] args)
7     {
8        System.out.printf("%d\t%d\n", 786, -786);
9        System.out.printf("%+d\t%+d\n", 786, -786);
10    } // end main
11 } // end class PlusFlagTest
```

```
786     -786
+786    -786
```

Fig. I.16 | Printing numbers with and without the + flag.

Figure I.17 prefixes a space to the positive number with the **space flag**. This is useful for aligning positive and negative numbers with the same number of digits. The value -547 is not preceded by a space in the output because of its minus sign. Figure I.18 uses the **# flag** to prefix 0 to the octal value and 0x to the hexadecimal value.

```
1  // Fig. I.17: SpaceFlagTest.java
2  // Printing a space before non-negative values.
3
```

Fig. I.17 | Printing a space before nonnegative values. (Part 1 of 2.)

```
4  public class SpaceFlagTest
5  {
6     public static void main(String[] args)
7     {
8        System.out.printf("% d\n% d\n", 547, -547);
9     } // end main
10 } // end class SpaceFlagTest
```

```
 547
-547
```

Fig. I.17 | Printing a space before nonnegative values. (Part 2 of 2.)

```
1  // Fig. I.18: PoundFlagTest.java
2  // Using the # flag with conversion characters o and x.
3
4  public class PoundFlagTest
5  {
6     public static void main(String[] args)
7     {
8        int c = 31;       // initialize c
9
10       System.out.printf("%#o\n", c);
11       System.out.printf("%#x\n", c);
12    } // end main
13 } // end class PoundFlagTest
```

```
037
0x1f
```

Fig. I.18 | Using the # flag with conversion characters o and x.

Figure I.19 combines the + flag the **0 flag** and the space flag to print 452 in a field of width 9 with a + sign and leading zeros, next prints 452 in a field of width 9 using only the 0 flag, then prints 452 in a field of width 9 using only the space flag.

```
1  // Fig. I.19: ZeroFlagTest.java
2  // Printing with the 0 (zero) flag fills in leading zeros.
3
4  public class ZeroFlagTest
5  {
6     public static void main(String[] args)
7     {
8        System.out.printf("%+09d\n", 452);
9        System.out.printf("%09d\n", 452);
10       System.out.printf("% 9d\n", 452);
11    } // end main
12 } // end class ZeroFlagTest
```

Fig. I.19 | Printing with the 0 (zero) flag fills in leading zeros. (Part 1 of 2.)

```
+00000452
000000452
      452
```

Fig. I.19 | Printing with the 0 (zero) flag fills in leading zeros. (Part 2 of 2.)

Figure I.20 uses the comma (,) flag to display a decimal and a floating-point number with the thousands separator. Figure I.21 encloses negative numbers in parentheses using the (flag. The value 50 is not enclosed in parentheses in the output because it's a positive number.

```
1  // Fig. I.20: CommaFlagTest.java
2  // Using the comma (,) flag to display numbers with thousands separator.
3
4  public class CommaFlagTest
5  {
6     public static void main(String[] args)
7     {
8        System.out.printf("%,d\n", 58625);
9        System.out.printf("%,.2f", 58625.21);
10       System.out.printf("%,.2f", 12345678.9);
11    } // end main
12 } // end class CommaFlagTest
```

```
58,625
58,625.21
12,345,678.90
```

Fig. I.20 | Using the comma (,) flag to display numbers with the thousands separator.

```
1  // Fig. I.21: ParenthesesFlagTest.java
2  // Using the (flag to place parentheses around negative numbers.
3
4  public class ParenthesesFlagTest
5  {
6     public static void main(String[] args)
7     {
8        System.out.printf("%(d\n", 50);
9        System.out.printf("%(d\n", -50);
10       System.out.printf("%(.1e\n", -50.0);
11    } // end main
12 } // end class ParenthesesFlagTest
```

```
50
(50)
(5.0e+01)
```

Fig. I.21 | Using the (flag to place parentheses around negative numbers.

I.11 Printing with Argument Indices

An **argument index** is an optional integer followed by a `$` sign that indicates the argument's position in the argument list. For example, lines 20–22 and 25–26 in Fig. I.9 use argument index `"1$"` to indicate that all format specifiers use the first argument in the argument list. Argument indices enable programmers to reorder the output so that the arguments in the argument list are not necessarily in the order of their corresponding format specifiers. Argument indices also help avoid duplicating arguments. Figure I.22 prints arguments in the argument list in reverse order using the argument index.

```
1  // Fig. I.22: ArgumentIndexTest
2  // Reordering output with argument indices.
3
4  public class ArgumentIndexTest
5  {
6     public static void main(String[] args)
7     {
8        System.out.printf(
9           "Parameter list without reordering: %s %s %s %s\n",
10          "first", "second", "third", "fourth");
11       System.out.printf(
12          "Parameter list after reordering: %4$s %3$s %2$s %1$s\n",
13          "first", "second", "third", "fourth");
14    } // end main
15 } // end class ArgumentIndexTest
```

```
Parameter list without reordering: first second third fourth
Parameter list after reordering: fourth third second first
```

Fig. I.22 | Reordering output with argument indices.

I.12 Printing Literals and Escape Sequences

Most literal characters to be printed in a `printf` statement can simply be included in the format string. However, there are several "problem" characters, such as the quotation mark (") that delimits the format string itself. Various control characters, such as newline and tab, must be represented by escape sequences. An escape sequence is represented by a backslash (\), followed by an escape character. Figure I.23 lists the escape sequences and the actions they cause.

Escape sequence	Description
`\'` (single quote)	Output the single quote (') character.
`\"` (double quote)	Output the double quote (") character.
`\\` (backslash)	Output the backslash (\) character.
`\b` (backspace)	Move the cursor back one position on the current line.

Fig. I.23 | Escape sequences. (Part 1 of 2.)

Escape sequence	Description
`\f` (new page or form feed)	Move the cursor to the start of the next logical page.
`\n` (newline)	Move the cursor to the beginning of the next line.
`\r` (carriage return)	Move the cursor to the beginning of the current line.
`\t` (horizontal tab)	Move the cursor to the next horizontal tab position.

Fig. I.23 | Escape sequences. (Part 2 of 2.)

Common Programming Error I.4

Attempting to print as literal data in a `printf` *statement a double quote or backslash character without preceding that character with a backslash to form a proper escape sequence might result in a syntax error.*

I.13 Formatting Output with Class `Formatter`

So far, we've discussed displaying formatted output to the standard output stream. What should we do if we want to send formatted outputs to other output streams or devices, such as a `JTextArea` or a file? The solution relies on class `Formatter` (in package `java.util`), which provides the same formatting capabilities as `printf`. `Formatter` is a utility class that enables programmers to output formatted data to a specified destination, such as a file on disk. By default, a `Formatter` creates a string in memory. Figure I.24 demonstrates how to use a `Formatter` to build a formatted string, which is then displayed in a message dialog.

Line 11 creates a `Formatter` object using the default constructor, so this object will build a string in memory. Other constructors are provided to allow you to specify the destination to which the formatted data should be output. For details, see `http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html`.

```
1  // Fig. I.24: FormatterTest.java
2  // Formatting output with class Formatter.
3  import java.util.Formatter;
4  import javax.swing.JOptionPane;
5
6  public class FormatterTest
7  {
8     public static void main(String[] args)
9     {
10       // create Formatter and format output
11       Formatter formatter = new Formatter();
12       formatter.format("%d = %#o = %#X", 10, 10, 10);
13
14       // display output in JOptionPane
15       JOptionPane.showMessageDialog(null, formatter.toString());
16    } // end main
17 } // end class FormatterTest
```

Fig. I.24 | Formatting output with class `Formatter`. (Part 1 of 2.)

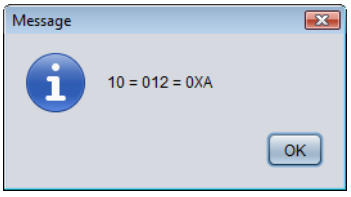


Fig. I.24 | Formatting output with class `Formatter`. (Part 2 of 2.)

Line 12 invokes method **`format`** to format the output. Like `printf`, method `format` takes a format string and an argument list. The difference is that `printf` sends the formatted output directly to the standard output stream, while `format` sends the formatted output to the destination specified by its constructor (a string in memory in this program). Line 15 invokes the `Formatter`'s `toString` method to get the formatted data as a string, which is then displayed in a message dialog.

Class `String` also provides a `static` convenience method named `format` that enables you to create a string in memory without the need to first create a `Formatter` object. Lines 11–12 and line 15 in Fig. I.24 could have been replaced by

```
String s = String.format("%d = %#o = %#x", 10, 10, 10);
JOptionPane.showMessageDialog(null, s);
```

I.14 Wrap-Up

This appendix summarized how to display formatted output with various format characters and flags. We displayed decimal numbers using format characters `d`, `o`, `x` and `X`; floating-point numbers using format characters `e`, `E`, `f`, `g` and `G`; and dates and times in various format using format characters `t` and `T` and their conversion suffix characters. You learned how to display output with field widths and precisions. We introduced the flags `+`, `-`, space, `#`, `0`, comma and `(` that are used together with the format characters to produce output. We also demonstrated how to format output with class `Formatter`.