Formatting Output with printf()
in Java

Last updated: January 8, 2024

Written by:
baeldung

Reviewed by:
Eric Martin

Core Java

Java OutputStream

1. Overview

In this tutorial, we'll demonstrate different examples of formatting with the *printf()* method.

The method is part of the *java.io.PrintStream* class and provides String formatting similar to the *printf()* function in C.

Further reading:

Guide to java.util.Formatter

Introduction to formatting Strings in Java using the java.util.Formatter.

Read more →

Guide to DateTimeFormatter

Learn how to use the Java 8 DateTimeFormatter class to format and parse dates and times

Read more →

Pad a String with Zeros or Spaces in Java

Learn how to pad a String in Java with a specific character.

Read more →

2. Syntax

We can use one of these *PrintStream* methods to format the output:

```
System.out.printf(format, arguments);
System.out.printf(locale, format, arguments);
```

We specify the formatting rules using the *format* parameter. Rules start with the *%* character.

Let's look at a quick example before we dive into the details of the various formatting rules:

```
System.out.printf("Hello %s!%n", "World");
```

This produces the following output:

```
Hello World!
```

As shown above, the format string contains plain text and two formatting rules. The first rule is used to format the string argument. The second rule adds a newline character to the end of the string.

2.1. Format Rules

Let's have a look at format string more closely. It consists of literals and format specifiers. **Format specifiers include flags, width, precision, and conversion characters** in this sequence:

```
%[flags][width][.precision]conversion-character
```

Specifiers in the brackets are optional.

Internally, *printf()* uses the java.util.Formatter class to parse the format string and generate the output. Additional format string options can be found in the Formatter Javadoc.

2.2. Conversion Characters

The *conversion-character* is required and determines how the argument is formatted.

Conversion characters are only valid for certain data types. Here are some common ones:

- *s* formats strings.
- *d* formats decimal integers.
- *f* formats floating-point numbers.
- *t* formats date/time values.

We'll explore these and a few others later in the tutorial.

2.3. Optional Modifiers

The *[flags]* define standard ways to modify the output and are most common for formatting integers and floating-point numbers.

***The [width]* specifies the field width for outputting the argument.** It represents the minimum number of characters written to the output.

***The [.precision]* specifies the number of digits of precision** when outputting floating-point values. Additionally, we can use it to define the length of a substring to extract from a *String*.

3. Line Separator

To break the string into separate lines, we have a *%n* specifier:

```
System.out.printf("baeldung%nline%nterminator");
```

The code snippet above will produce the following output:

```
baeldung
line
terminator
```

The *%n* separator *printf()* will automatically insert the host system's native line separator.

4. Boolean Formatting

To format Boolean values, we use the *%b* format.

According to the docs, it works the following way: if the second argument is *null*, then the result is "false". If the argument is a *boolean* or *Boolean*, then the result is the string returned by *String.valueOf(arg)*. Otherwise, the result is "true".

So, if we do the following:

```
System.out.printf("%b%n", null);
System.out.printf("%B%n", false);
System.out.printf("%B%n", 5.3);
System.out.printf("%b%n", "random text");
```

then we'll see:

```
false
FALSE
TRUE
true
```

Notice that we can use *%B* for uppercase formatting.

5. String Formatting

To format a simple string, we'll use the *%s* combination. Additionally, we can make the string uppercase:

```
printf("'%s' %n", "baeldung");
printf("'%S' %n", "baeldung");
```

And this is the output:

```
'baeldung'
'BAELDUNG'
```

Also, to specify a minimum length, we can specify a *width*:

```
printf("'%15s' %n", "baeldung");
```

which gives us:

```
'       baeldung'
```

If we need to left-justify our string, we can use the – *flag*:

```
printf("'%-10s' %n", "baeldung");
```

This is the output:

```
'baeldung  '
```

Even more, we can limit the number of characters in our output by specifying a *precision*:

```
System.out.printf("%2.2s", "Hi there!");
```

The first *x* number in *%x.ys* syntax is the padding. *y* is the number of chars.

For our example here, the output is *Hi*.

6. Char Formatting

The result of *%c* is a Unicode character:

```
System.out.printf("%c%n", 's');
System.out.printf("%C%n", 's');
```

The capital letter *C* will uppercase the result:

```
s
S
```

But if we give it an invalid argument, then *Formatter* will throw *IllegalFormatConversionException*.

7. Number Formatting

7.1. Integer Formatting

The *printf()* method accepts all the integers available in the language — *byte*, *short*, *int*, *long*, and *BigInteger* if we use *%d*:

```
System.out.printf("simple integer: %d%n", 10000L);
```

With the help of the *d* character, we'll have this result:

```
simple integer: 10000
```

In case we need **to format our number with the thousands separator, we can use the *, flag*.** And we can also format our results for different locales:

```
System.out.printf(Locale.US, "%,d %n", 10000);
System.out.printf(Locale.ITALY, "%,d %n", 10000);
```

As we can see, the formatting in the US is different than in Italy:

```
10,000
10.000
```

7.2. Float and Double Formatting

To format a float number, we'll need the *f* format:

```
System.out.printf("%f%n", 5.1473);
```

which will output:

```
5.147300
```

Of course, the first thing that comes to mind is to control the *precision*:

```
System.out.printf("'%5.2f'%n", 5.1473);
```

Here we define the *width* of our number as *5*, and the length of the decimal part is *2:*

```
' 5.15'
```

Here we have one-space padding from the beginning of the number to support the predefined width.

To have our output **in scientific notation, we just use the *e conversion-character***.

```
System.out.printf("'%5.2e'%n", 5.1473);
```

And this is our result:

```
'5.15e+00'
```

8. Date and Time Formatting

For date and time formatting, **the conversion string is a sequence of two characters: the *t* or *T* character and the conversion suffix.**

Let's explore the most common time and date formatting suffix characters with examples.

Definitely, for more advanced formatting, we can use *DateTimeFormatter*, which has been available since Java 8.

8.1. Time Formatting

First, let's see the list of some useful suffix characters for time formatting:

- ***H*, *M*, *S* characters are responsible for extracting the hours, minutes and seconds** from the input *Date*.
- *L*, *N* represent the time in milliseconds and nanoseconds accordingly.
- ***p* adds a.m./p.m. formatting.**
- ***z* prints out the time-zone offset.**

Now, let's say we want to print out the time part of a *Date*:

```
Date date = new Date();
System.out.printf("%tT%n", date);
```

The code above along with *%tT* combination produces the following output:

```
13:51:15
```

In case we need more detailed formatting, we can call for different time segments:

```
System.out.printf("hours %tH: minutes %tM: seconds %tS%n", date, date, date);
```

Having used *H*, *M* and *S*, we get this result:

```
hours 13: minutes 51: seconds 15
```

However, listing *date* multiple times is a pain.

Alternatively, **to get rid of multiple arguments, we can use the index reference of our input parameter, which is *1$* in our case**:

```
System.out.printf("%1$tH:%1$tM:%1$tS %1$tp %1$tL %1$tN %1$tz %n", date);
```

Here we want as an output the current time, a.m./p.m., the time in milliseconds and nanoseconds, and the time-zone offset:

```
13:51:15 pm 061 061000000 +0400
```

8.2. Date Formatting

Like time formatting, we have special formatting characters for date formatting:

- ***A* prints out the full day of the week.**
- *d* formats a two-digit day of the month.
- ***B* is for the full month name.**
- *m* formats a two-digit month.
- ***Y* outputs a year in four digits.**
- *y* outputs the last two digits of the year.

Suppose we want to show the day of the week, followed by the month:

```
System.out.printf("%1$tA, %1$tB %1$tY %n", date);
```

Then, using *A*, *B* and *Y*, we'd get this output:

```
Thursday, November 2018
```

To have our results all in numeric format, we can replace the *A*, *B*, *Y* letters with *d*, *m*, *y*:

```
System.out.printf("%1$td.%1$tm.%1$ty %n", date);
```

which will result in:

```
22.11.18
```

9. Conclusion

In this article, we discussed how to use the *PrintStream#printf* method to format output. We looked at the different format patterns used to control the output for common data types.

The code backing this article is available on GitHub. Once you're **logged in as a Baeldung Pro Member**, start learning and coding on the project.

Baeldung

COURSES
ALL COURSES
BAELDUNG ALL ACCESS
BAELDUNG ALL TEAM ACCESS
THE COURSES PLATFORM

SERIES
JAVA "BACK TO BASICS" TUTORIAL
LEARN SPRING BOOT SERIES
SPRING TUTORIAL
GET STARTED WITH JAVA
ALL ABOUT STRING IN JAVA
SECURITY WITH SPRING
JAVA COLLECTIONS

ABOUT
ABOUT BAELDUNG
THE FULL ARCHIVE
EDITORS
OUR PARTNERS
PARTNER WITH BAELDUNG
EBOOKS
FAQ
BAELDUNG PRO

TERMS OF SERVICE | PRIVACY POLICY | COMPANY INFO | CONTACT

PRIVACY MANAGER