1. String Formatting

Contents

[1. Introduction 1](#_Toc28090)

[2. String Concatenation vs. Formatting 1](#_Toc2884)

[3. Parts of a Format Specifier 1](#_Toc7593)

[4. Displaying Radix 1](#_Toc6304)

[5. Managing Width and Padding 1](#_Toc30646)

[6. Grouping Values 2](#_Toc30513)

[7. Controlling Appearance of Positive and Negative Values 2](#_Toc15479)

[8. CalcEngine with String Formatting 2](#_Toc5041)

[9. Argument Index 2](#_Toc591)

[10. Summary 2](#_Toc23870)

# 1. Introduction

=>slides: Pg. 1-2

Welcome to our next module, String Formatting. This is part of the Pluralsight course Getting Started with Programming in Java. My name is Jim Wilson. In this module, we're going to build on our knowledge of strings to see how we can build well‑formatted strings. So the first thing we'll take a look at is what are the advantages of format specifiers versus some of the earlier techniques we've talked about in terms of building strings. We'll then see how to use format specifiers. We'll see how we can control the width and precision of numeric values that we place within our strings. We'll also see a number of flags that will give us very detailed control of the appearance of values that we place inside of our strings. And then we'll finish up with a look at argument indexes, which allow us to match up format specifiers with individual arguments.

slides: Pg. 3

Now you may be wondering at this point why do we need to talk about another way to create strings? We've already talked about things like string concatenation and StringBuilder, and those are really valuable tools. The thing we want to keep in mind though, when we work with string concatenation and StringBuilder, we very often find ourselves heavily focused on the details of how we construct the string. We often have to build that string very explicitly piece by piece. And then we often involve taking individual variables we want to work with, as well as mixing in a whole bunch of literal strings. In addition, when we work with string concatenation and StringBuilder, we don't really have a way to customize the appearance of numeric values. So any customization of those numeric values, we actually have to do manually. So when we're building strings with string concatenation and StringBuilder, we often find ourselves tied up in the details of the nuts and bolts of how we actually put these strings together.

slides: Pg. 4  
So that's where string formatting comes in because string formatting allows us to use what are known as format specifiers. And format specifiers allow us to describe our desired result. So instead of dealing with all the nuts and bolts of how we build that string, format specifiers in a sense allow us to paint a picture of what we want the desired result to look like. And format specifiers give us the ability to control many aspects of the appearance of the values we place into those strings. And this string formatting concept is very widely supported. There are a number of methods that support string formatting. For example, the one we'll work with most is String.format, which allows us to construct a string using these format specifiers. But we also have things like System.out.printf, which allow us to display formatted content. And there's even something known as the Formatter class, which will allow us to output formatted content to pretty much anything you can think of, a file, a network, just about anything. So in our next section, let's take a look at what it's like to start building strings using string formatting.

# 

# String Concatenation vs. Formatting

=>slides: Pg. 5

So let's take a look now what it's like to work with string formatting versus some of the other string construction techniques that we've already talked about. So let's start out with these four variables, and these variables each list the age of my nephews. And let's say I want to do a very simple task of construct a string that looks like this. My nephews are, I list each individual age with the appropriate separators, and then end in the text years old. So let's start out by creating a string using string concatenation. So I would start out with a string literal, My nephews are. I would then concatenate the variable for my oldest nephew, concatenate a string literal with a separator, concatenate another variable, another string literal, another variable, another string literal, another variable, and then finally, the last string literal with years old. Now building the string this way has a couple of issues. One, it's just kind of complicated. We're actually having to work with it kind of piece by piece. In addition, if someone opened up the source file and had no idea what we were trying to achieve, they would actually have to read the line pretty closely to figure out what we're doing. So now let's take a look at building that same string using string formatting. So we'll use the String.format method, and then all we have to do here is provide our format string. And you'll notice that we fully describe our output in this one single string. All of our string literals are inside this one string, but we've included these format specifiers. These format specifiers act as substitution points. So if I then list the variables that I want to use, each of those variables is substituted at the point of the format specifier. So, as you can see, using this technique was much easier for me to write than building the string piece by piece, and if someone else reads this later on, it's very clear what we're trying to achieve.

=>slides: Pg. 6

Now string formatting also has the benefit of giving us control over how each value is displayed. So let's start out with these same four variables, but let's go ahead and add in a calculation. This calculation figures out the age difference between each of my nephews and determines the average. So then once we have the average calculated, let's go ahead and construct a string that says the average age between each is, provides that value, and then has that literal years. Now in this case, the string is pretty simple, so it's pretty easy to read this and know what we're trying to do. But now if we look at the resulting string, notice the value that gets included there, 3.666 followed by a whole bunch of 6s. And that is an extremely accurate answer, but it's not really a helpful answer for the person has to look at this string. We'd like to be much more concise than that.

=>slides: Pg. 7

So let's try doing the same thing using string formatting. So we've got the same variables, the same calculation, we'll again use the String.format method, and we'll specify our format string. So again, we have everything described here in this one string, but notice here where we have the format specifier, we actually indicate we want the value to have just one decimal place. So when we substitute in our average value, the resulting string now rounds the value down to that one decimal place. So now our string contains a much more clear and meaningful value than that 3 followed by all those 6s. So as you can see, format strings simplify the process of building strings and give us a lot of control over the appearance of the values we place within those strings. So in our next section, let's take a closer look at the parts of our format specifiers.

# Parts of a Format Specifier

=>slides: Pg. 8

Within the format strings, it's the format specifiers that indicate when we want value substitution. And there's actually a number of parts before my specifiers. Now each format specifier starts with a percent sign, and it ends with what we call the conversion. And the conversion serves as the foundation for how a given data value will be displayed. But there are actually a number of other capabilities available to us beyond the conversion.

=>slides: Pg. 9

For example, we have the precision. The precision indicates the number of decimal places. We saw that used in one of our earlier examples. We also have the width. The width indicates the minimum number of characters that should be allocated for this value. By default, the data value will be right‑justified within that width. But now beyond that, we also have a series of flags that will allow us to further customize the appearance of the value. And we have what's called the argument index, which gives us control over which format specifiers match up with which data values. So let's take a look at some of the most common conversions.

=>slides: Pg. 10

Now far and away, one of the most common conversions is the d conversion, and that stands for decimal conversion of an integer type. And basically, this just displays integers kind of the way you expect it to. If you give it a value of 32, it will display a value of 32. Then we have the X conversion character, and that can be either lower or uppercase. And this does a hex conversion of an integer value. So it'll display the value as hexidecimal. So I give it the value 32. It will display that value as 20 because 20 is how you express the decimal value 32 in hex. Then we have f, and that does a decimal conversion of a floating point value. So again, it displays a floating point value kind of as you would expect. You get 123.0, and we'll display 123., followed by a number of decimal places. And then we have the e conversion, which displays floating point values using scientific notation. And scientific notation simply means that we express the value in terms of its relationship to a power of 10. So if we give it 123.0 that displays as 1.230000e+02. What that means is it's 1.23 times 10 to the second power or 1.23 times 100, which means move the decimal place two places to the right. And then finally, we have the s conversion, which simply means a string conversion. So this allows us to substitute string values into our format string. But this s conversion is actually fairly intelligent. Many data types know how to convert themselves into a string, and the s conversion respects that. And some data types even have built‑in formatting capability, and the s conversion respects that as well. Alright, so that's some of the most commonly used conversions. In our next section, let's start taking a look at some of the flag capabilities.

# Displaying Radix

=>slides: Pg. 11

Let's take a look now at some of the format flags. Remember, the flags allow us to further control the appearance of the values we display. Now the first flag we'll look at here is the pound symbol (#), and this flag indicates that we want to include the radix. And what that means is we want to include information in the value that indicates what base that value is displayed in. Now the easiest way to see how this flag works, I think, is an example.

=>slides: Pg. 12

So let's take a local variable here, iVal, that we've set to 32. Now, normally, when we print out an integer value, we're probably going to use %d. That indicates I want to print it out in decimal format, in other words, the formatting that we most commonly expect. But there are some cases where we want to see values in hexadecimal, in other words, base 16. We often use hex when displaying color values. We often tend to use it also when showing certain configuration information because hexadecimal is useful in cases where you want to have a sense of the underlying bit structure of the value. Now the way we display values in hexadecimal is use the hex conversion. So if I take the decimal value, 32, and I print it out as hex, I'll get the characters 2‑0. Now if a user sees this value, 2‑0, most likely they're going to think of it as the decimal value 20 when we actually mean the hexadecimal value. So in general, what we want to do is include information that indicates what base that displayed value is in. That's where our # flag comes in. By including the # flag, it says, Display the value using the traditional indicators to tell us what base this number is. So by printing out iVal using that flag, we now get 0x20, which is the traditional format for displaying hexadecimal numbers. Now in this case, we've displayed it using the lowercase x. We can also display the values using an uppercase X. The only difference is how the x appears in the output. If you display with a lowercase x, the output will use a lowercase x. If you display with an uppercase X, the output will have an uppercase X. There's no real difference in the meaning. It's really a matter of preference. In my experience, I use the lowercase x the overwhelming majority of the time. So now in our next section, let's take a look at how we can control the width of values and the flags to help us control the appearance of our values within that width.

# Managing Width and Padding

=>slides: Pg. 13

Let's take a look at some flags to give us control of the appearance of values within a specified width. So there are two of these flags. One flag is the zero flag, which indicates we want the value of zero padded. The other flag is what we call the dash flag or sometimes the minus flag, which indicates we want the value left justified. So let's look at an example.

=>slides: Pg. 14

So here I have four integers, w, x, y, and z. So if I print out W and X using kind of a standard format string. We've got literals that label each value, so W: and X:. And our format specifiers are simply %d, indicating decimal conversions with no further modification. So in this case, we would get W:5, because W has a value of 5, space X:235 because X has a value of 235. So you notice that each value is simply taking up whatever space it needs. So if I now display Y and Z using a similar format string, their format is similarly, with each value, just taking up the value that it needs. Now if you're displaying these strings using a fixed width font, this might be a bit confusing for the user because if these values relate to one another, the user's general going to expect some kind of alignment. And we can give them that alignment by specifying width values. So if I now format W and X, specifying a width of 4, that means that each value will take up four characters and will be right justified within those characters. So you'll notice the value of W becomes space space space 5. The value of X becomes space 235. So each value takes up four characters and is right justified within those four characters. So now if I format Y and Z using a similar format string, now the value for Y will align under the value for W. The value for Z will align under the value for X. But now we don't always want values to simply align.

=>slides: Pg. 15

Sometimes we want more control than that. So we can use the flags we've talked about. For example here, if I format to the four values using the zero flag, so in each case I've allocated a width of four, but I've specified zero flag on all of them, that indicates that I want to put leading zeros in the values. So that gives us an appearance like this. The values all take up four characters, and any non‑significant leading digits appear as a zero. And you'll often see values displayed this way in things like coordinate systems because it makes it very clear to understand the relationship of one value to another. Now when specifying the width, we also have the option to use the dash flag or what we call the minus flag. Again, that indicates that the value should be left justified. So if we look at our four values again, notice what happens here. Each value is allocated four characters, but the value is left justified within those four characters. So what that allows us to do in this case is by giving each value a consistent number of characters, the labels associated with those characters remain aligned, but the values are close to their label. So as you can see, by using the width in these flags, we have a great deal of control about how our values appear and how they align. So now in our next section, let's see how can enable grouping of digits within larger values.

# Grouping Values

=>slides: Pg. 16

In many situations when we display larger values, users expect the digits within those values to be grouped, and that's where the comma flag comes in. The comma flag indicates that we want our value to include grouping separators. So this one's pretty straightforward.

=>slides: Pg. 17

So let's say we have an integer value here that we initialize to a fairly large value. Now if we format this value using simply %d, what we'll get out is just that value. Now for the end user, this value may be hard to read because in many cases, the end user is going to expect the digits to be grouped. So if we now format the value, but include that comma flag, the resulting value now has the groups of digits separated by the group separator. So in this case, it becomes very easy to see that the value is 1,234,567. Now something to keep in mind, not all languages and cultures use the comma character as the grouping separator. Well not to worry because many of the methods that interpret format strings also support localization. Localization simply means that we format values that are appropriate for the language or culture that we're using. So in cases where the culture uses a different grouping separator, we're still going to use the comma flag in the format specifier. But the method that interprets the string will use the appropriate grouping character for that culture. Now also, the comma flag is not limited to being used with integer values. Say I have this double value here. Well I can use a format string that has the f conversion, specifies two decimal places, and also includes the comma flag, and then that will then format that value appropriately. So we get a value that has the digits grouped and shows the appropriate number of decimal places. All right, so now in our next section, we'll look at our last group of flags, and those are flags related to controlling the appearance of positive and negative values.

# Controlling Appearance of Positive and Negative Values

=>slides: Pg. 18

Our last set of flags relates to how we handle positive and negative values. Now by default, when we format a negative value, we have a leading minus sign (‑). When we format a positive value, there'll be no sign at all. So these flags allow us to control that behavior. So the first flag we'll look at is the space flag. That indicates that in the case of a positive number, I don't want to see a sign. But I would like to leave a space where the sign would be. Then we have the + flag, which says I always want to see the sign. So in case of a negative number, I'll have a minus sign. In the case of a positive number, I'll have a plus sign. Then our last one is a ( flag, which indicates that we want negative numbers enclosed within parentheses. So let's look at some examples.

=>slides: Pg. 19

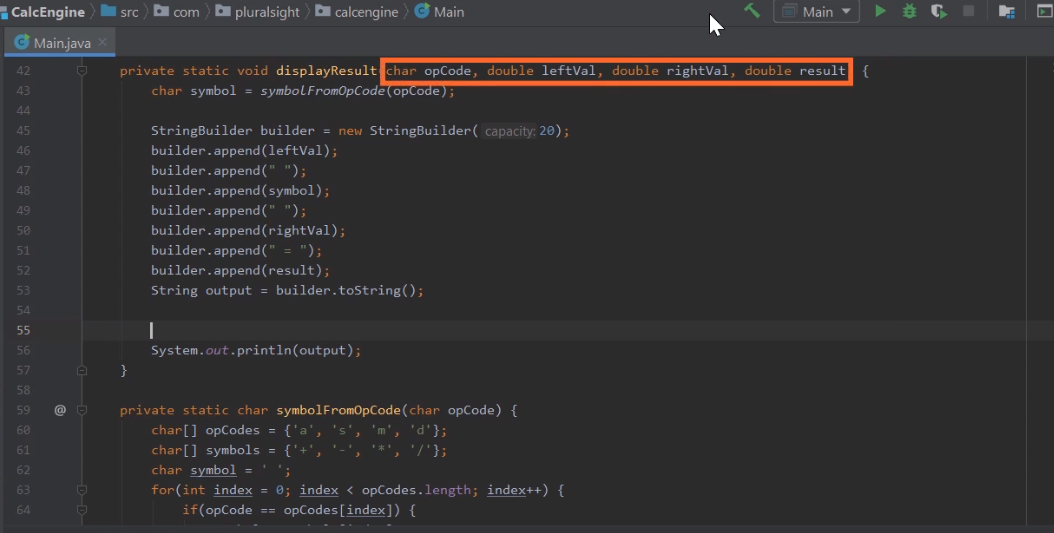
So I have two values here, iPosVal, which is initialized to 123, and iNegVal, which is initialized to ‑456. So if we go ahead and format both these values using a simple %d format specifier, in the case of our positive value, we'll get 123. In the case of our negative value, we'll get ‑456. Now, oftentimes, this is just fine, but not always, particularly if we're going to have a column containing multiple values displayed in a fixed font because one of the issues that comes up is how the left edge of the number is aligned. Notice that the first digit of the positive number is sitting right above the minus sign on the negative number. Oftentimes, we want the digits to start in the same place. So one way we can deal with that is by using the space flag. Notice the format specifier here has a space between the percent and the conversion. Now this space flag won't really have any effect on the negative value. It'll still be ‑456. But notice that the positive value is now moved over. Where previously the one started directly above the minus sign, there's now a space there to help the digits themselves line up.

=>slides: Pg. 20

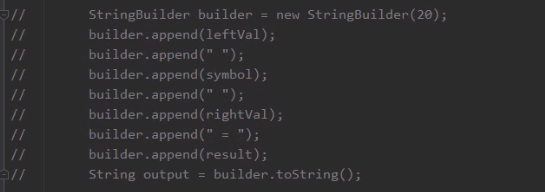
Now there are also cases where sometimes we always want to see a sign, and that's where the + flag comes in. The + flag says always show the sign. So, again, the negative value is unchanged. But now, in this case, the positive value will actually have the + sign. And one place where this is often used is when dealing with coordinate systems. When you're dealing with coordinates, you want to be very clear whether those values are positive or negative. And then our last flag is the ( flag.

This notation is sometimes what we call a counting notation. So in this case, negative values are enclosed in parentheses. So our iNegVal is formatted as 456 within parentheses. Now in the case of positive values, those are unchanged. We simply get the value itself. Now an issue that often comes up when using this parentheses notation to display negative values is, again, this issue of alignment. Notice that the first digit of the positive value is being aligned with the parentheses on a negative value. If you're using parentheses for negative numbers, you're often going to not want that to be a case. So to deal with that, what we can do is combine multiple flags. So we'll have our ( flag, but we'll combine that with the space flag. So now when we display the positive value, notice that the digits for the positive value are aligned with the digits for the negative value. A space has been placed in front of the positive value to occupy the position of that opening parentheses. All right, that's our formatting flags. To help us get a better understanding of all this, in our next section, we'll jump into our CalcEngine project, and we'll start using format strings.

# CalcEngine with String Formatting



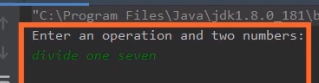
Here in STS, we're again looking at our CalcEngine project. Now, as you recall in our previous module, we created this method displayResult. DisplayResult receives our opCode, leftVal, rightVal, and result. We then convert the opCode into a symbol, and then using the StringBuilder class, construct the output of our application as if it's an equation. Now, the StringBuilder class is a really powerful class and has a lot of good uses, but this is a scenario where a format string could actually allow us to construct the output more easily.



So let me go ahead and comment out all the code related to our StringBuilder class, including where we convert the content of the StringBuilder to a string. Now once we do that, we're going to declare another variable named output of type string. Now we're going to use a format string, so we'll call it string.format. Then, of course, the first thing we'll need to do is specify that format string.



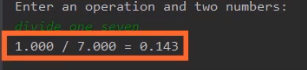
Well, the first thing we want for an output is the leftVal. That's a Double, so we'll use %f. Then we're going to want a space, the symbol, and another space. So now when using a format specifier with a character, we're going to use %c. Then we want our rightVal to be %f again. And then we can do a space = space, and then %f for the result. So now that takes care of our format string, so now we can simply provide the values we want to substitute into the format string. So that'll be leftVal, symbol, rightVal, and result. And believe it or not, that's all there is to it. All those lines of code using the StringBuilder are now replaced with this one line. So let's go ahead and run this code and see what it looks like.



So now down here in our run window we'll type in divide one seven. So now what that'll do is divide the number one by seven. So I'll go ahead and hit Enter. And now you'll notice we got our output, but maybe we're showing a few more decimal places than we'd like to. It might be kind of nice to cut these values down to just have three decimal places. So what we can do is head up here to our format string, and each place we use %f, we'll specify three decimal places.



So now for each place where we specified %f, we've limited the number of decimal places to three. So I'm going to go ahead and run it again.



We'll again say divide one seven, and then we'll go ahead and hit Enter. And as you can see, each of the values now have the number of decimal places limited to three. But I want to know one thing. Look at the value on the right of the equal sign. I don't know if you noticed this, but previously when we ran the code that value was 0.142857. Now it's 0.143, so that shows us when we limit the number of decimal places, we don't just throw away the extra decimal places. The formatting actually rounded the value for us. So as you can see, the format strings are giving us a lot of control over the way our output appears. Okay, so now in our next section, let's take a look at what's known as the argument index.

# Argument Index

=>slides: Pg. 21

Now the last aspect of format specifiers we'll look at is the argument index. And what the argument index does is it allows us to associate a particular format specifier with a particular argument. Now, up until now, we have not specified the argument index. And so, what that means is that format specifiers are matched up with the arguments in order. So the first specifier goes with the first argument, second specifier with the second argument, and so forth, but there are times where we want to be more explicit than that. So within our format specifier, we can provide an index for the argument followed by a $ sign. So in that case, the format specifier ties to the identified argument. Now note that this index is 1‑based, so unlike arrays which are 0‑based, argument indexes are 1‑based, so 1 dollar sign corresponds to the very first argument value. And then one last option we have is the less than index and I often call this one the I'll have what he's having argument index. What it basically says is, I want to use this format specifier for the same argument that the previous format specifier was applied to. So let's look at a couple of examples.

=>slides: Pg. 22

So I've got three variables declared here, valA with a value 100, valB with a value of 200, and valC with a value of 300. Now, up until now, we've been running our format strings like this where we simply have the format specifiers and no argument index. And in my experience, this is the most common way that we lay out format strings. So basically, the format specifiers are matched up with the values in order. So the first specifier goes with the first value, second specifier with the second value, third specifier with the third value. So the resulting string here would be 100 space 200 space 300. But as I mentioned in some cases, we want to take more control than this, so we can write format strings like this. Notice that each format specifier has a number followed by a dollar sign. So in this case, we're explicitly tying a format specifier to an argument. So in this first one, we have an argument index of three, so that says it's tied to the third argument. So then our next format specifier is explicitly tied to the first argument, and our third format specifier is explicitly tied to the second argument, so the result in this case would be 300 space 100 space 200. Now, when we use these explicit argument indexes, in general, we don't use them this way so we can put the arguments in just any old order. What often comes up is you have a very complex format string where it may not be easy to identify which format specifier goes with which argument or you have scenarios where one argument value needs to be used with multiple format specifiers. So being explicit this way just gives us that kind of control. Now in the scenario where we want to apply multiple format specifiers to the same value, we also have another option. So we look at this format string, notice that our first format specifier is explicitly tied to the second argument, and then we go to our second format specifier, notice that it's using that less than value. So what that says is I want to format the same value that the previous format specifier formatted. So, since that one applied to the second argument, the second format specifier is also applied to the second argument. Then finally, we have our third format specifier that's explicitly tied to our first argument. So the result, in this case, becomes 200 space 200 space 100. So, as you can see, the argument index gives us a lot of control over how format specifiers are associated with our argument values.

# Summary

=>slides: Pg. 23

To wrap up, here are some of the key things you'll want to remember from this module. Remember that when we deal with our format strings, we use format specifiers. And by using format specifiers, we get away from the nuts and bolts of how we build these string values. Instead, we focus on describing the desired result. In effect, we paint a picture of what we want the outcome to be. Now when we provide our format specifiers, there are a couple of basic things we always have to have. The format specifiers always start with a percent sign, and they always end with a conversion. So, for example, we have the d conversion to display integer values in decimal format, and the f conversion for displaying floating point values. Now as we build up our format specifiers, we can add more and more detail about the desired outcome.

=>slides: Pg. 24

For example, we have the precision. Precision allowed us to indicate how many decimal places we wanted to display. And we also had the width, which allows to indicate the minimum number of characters to display for a given value.

=>slides: Pg. 25

And then we had the flags, which allows us to specify even more detail. We could indicate what we want the alignment to be. Do we want any special padding in place? Do we want leading 0s on the value? Do we want to group the digits within a number? How do we want to handle positive and negative signs? So these flags are really the heart of creating this very detailed control over the behavior of the formatting of these values. And then we finished up with the argument index. So remember that by default, the format specifiers match up with our argument values in order, but we don't have to do it that way. We can tie a given specifier to a specific value. But remember, when we do that, the values are matched up on a one‑based system. So unlike things like arrays, which are zero‑based, argument indexes are one‑based. So if you want associate a format specifier with the first argument, we use 1. When you match up with the second argument, you use 2, and so forth. Alright, that wraps up this module. In our next module, we'll take a closer look at some of the other data types provided by Java.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*