

# String Interpolation with 'f'

In the following lesson, you will be introduced to the 'f' string interpolator.

## We'll cover the following ^

- The f String Interpolator
  - Syntax
- printf-Style Formatting
  - Syntax
  - Flag
  - Width
  - Precision
  - Conversion-Characters
- Learn by Example

Remember when we discussed the `printf` method when learning how to `print`? `printf` is actually a Java method which is recognized by Scala. However, it is unconventional to use `printf` in Scala as it has its own, arguably better, `printf`.

## The `f` String Interpolator #

The `f` string interpolator is Scala's `printf`. For string interpolation with `f`, we prepend an `f` to any string literal. This allows us to create formatted strings. When using the `f` interpolator, all references to variables and expressions should be followed by a `printf`-style format string, like `%f`.

Let's look at an example:

This code requires the following environment variables to execute: ^

LANG C.UTF-8

```
val pi = 3.14159F
println(f"the value of pi is $pi%.2f")
```



In the code above, `f"the value of pi is $pi%.2f"` is a processed string literal. The `f` prepended before the string is letting the compiler know that the string should be processed using the `f` interpolator. `pi` is our variable identifier and `%.2f` is our format specifier and is formatting the string by telling the compiler to only print the floating-point number `pi` up to two decimal places.

Try replacing the 2 in `%.2f` with 3 or 4 and see how the output changes.

### Syntax #

`f"String $VariableIdentifier%FormatSpecifier String"`

### printf-Style Formatting #

Before we look at more examples of string interpolation in Scala, let's take a step back and look into Java's `printf` method. While we need not concern ourselves with the details of how `printf` works, it is important for us to look at format specifiers and how to use them.

First, we look at the syntax of the format specifier in detail.

### Syntax #

`%FlagWidth.PrecisionConversion-Character`

### Flag #

Flags are general modifiers and are mostly used to format integers and floating-point numbers. Below is a list of flags and what they are used for.

Flag	Use
-	left justify
+	Add a + sign
0	Add padded zeros

,	Add a local-specific grouping separator
---	---

## Width #





The width specifies the minimum length of the output. For example, if we say `%10d`, we mean that the output of the integer should be a minimum of 10 characters. To fulfill our requirement, the compiler will insert spaces before our argument until it is 10 characters long, pushing the argument further right.

This code requires the following environment variables to execute: ^

LANGC.UTF-8

```
val testWidth = 123

println(f"Without specifying the width, we get $testWidth")
println(f"With specifying the width, we get $testWidth%10d")
```



## Precision #

Precision can only be used on floating-point numbers and simply tells you how many places after the decimal point do you want to output. Just like in the first example of this lesson, by mentioning the precision `%.2f`, we told the compiler to only print up to two decimal places of our floating-point number.

## Conversion-Characters #

Finally, we come to the most important part of a format specifier and the only requirement (flag, width, and precision are optional); conversion characters.

Conversion-characters do the actual formatting and determine how the argument is to be formatted. While the list is long, for the scope of this course, we will look at some common ones.

Character	Use
s	formats strings

d	formats decimal integers
f	formats floating-point numbers
t	formats date/time values

Now that we have finally gone over the syntax, let's combine what we have learned and look at some examples.

At this point, if you are still not clear how to use the f `String` interpolator, the coming examples will go in great detail to clarify any confusion.

## Learn by Example #

1. In the first example, we will use the flag `,` which will automatically insert separators where ever required in our integer. Since we will be formatting an integer, the conversion-character of choice would be `d`.

This code requires the following environment variables to execute:

LANG C.UTF-8

```
val insertSeparator = 1000000

println(f"Without Formatting: $insertSeparator")
println(f"With Formatting: $insertSeparator%,d")
```

2. In our second example, we will use the *flag* `0` to insert zeros before the argument. We will also use *precision* to let the compiler know to which decimal place we would like our output to be printed; in this case: three decimal places. The `0` *flag* works simultaneously with *width*. As discussed above, when we specify a *width*, the compiler fills the required length with empty characters. When we combine the `0` flag with *width*, the compiler now fills the required length with zeros.

Let's choose a *width* of ten and since we are working with floating-points (due to *precision*), the conversion-character of choice would be `f`.

This code requires the following environment variables to execute: ^

LANG C.UTF-8

```
val insertZeros = 1.23456F

println(f"Without Formatting: $insertZeros")
println(f"With Formatting: $insertZeros%010.3f")
```







3. Our third and final example will be looking at the `-` *flag*. Like `0`, `-` also works simultaneously with *width*. While *width* on its own, pushes the argument to the right, with `-`, *width* pushes the argument to the left. Let's see it in action to get a better idea about how it works.

This code requires the following environment variables to execute: ^

LANG C.UTF-8

```
val leftJustify = 12345

println(f"Without Formatting: $leftJustify is the output")
println(f"With Width: $leftJustify%10d is the output")
println(f"With Flag: $leftJustify%-10d is the output")
```



And with the last example, our discussion on the `f` interpolator comes to an end. Feel free to challenge yourself and play around with the code snippets.

---

In the next lesson, you will be challenged to format a string.