What is Currying in Javascript? How to use Currying in JavaScript

The currying feature in JavaScript can help you keep your code tidy and give you a new way of seeing how functions work. Currying is ideal when you want to break complex logic into smaller, manageable, and self-contained pieces of code.

Here's what you need to know about currying in JavaScript.

What is Currying?

Currying is named after mathematician Haskell B. Curry, and the concept is derived from Lambda calculus. Currying takes a function that takes more than one parameter and splits it into a sequence of unique functions (one parameter). In other words, a curried function takes only one parameter at a time.

Basic example of currying

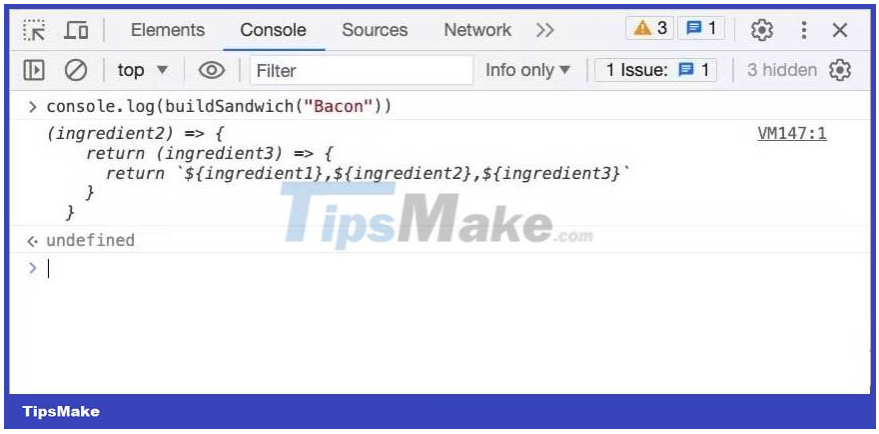
```
function buildSandwich(ingredient1) { return (ingredient2) => { return (ingredie
```

buildSandwich() returns another function - an anonymous function that takes an argument **ingredient2** . Then this function returns another anonymous function that takes **ingredient3** . The last function returns a pattern string, a way of formatting strings in JavaScript.

What you just created is a nested function where each function will call the function below it until we are finished. Now, when you call **buildSandwich()** and pass it a single parameter, it will return the part of the function whose arguments you haven't provided:

```
console.log(buildSandwich("Bacon"))
```

You can see that buildSandwich returns a function:



To complete the function call, you need to provide all 3 arguments:

```
buildSandwich("Bacon")("Lettuce")("Tomato")
```

This code passes 'Bacon' to the first function, 'Lettuce' to the second and 'Tomato' to the last function. In other words. **The buildSandwich()** function is actually divided into 3 functions, each of which takes only one parameter.

While it's perfectly legal to curry using traditional functions, all nested functions can get ugly if you go deeper. As a workaround, you can use arrow functions and take advantage of a cleaner syntax.

```
const buildMeal = ingred1 => ingred2 => ingred3 => `${ingred1}, ${ingred2}. ${ing
```

This refactored version is more accurate. This is the advantage of using arrow functions over regular functions. You can call this function the way you did with the previous function:

```
buildMeal("Bacon")("Lettuce")("Tomato")
```

Curry function is partially applied

Partial application of functions is a common use of curry. This technique requires providing only the necessary arguments at a time (instead of providing all arguments). Whenever you call a function by passing all required parameters, you say you have "applied" the function.

For example:

```
const multiply = (x, y) => x * y;
```

Below is the curryed version of the multiplication:

```
const curriedMultiply = x => y => x * y;
```

The curriedMultiply() function takes **x** arguments for the first function and **y** for the second function, and then it multiplies both values.

To create the first partially applied function, call **curriedMultiple()** with the first parameter and assign the return function to a variable:

```
const timesTen = curriedMultiply(10)
```

At this point, the code has partially applied the **curriedMultiply()** function . So whenever you want to call timesTen(), just convert it to a number and the number will automatically be multiplied by 10:

```
console.log(timesTen(8)) // 80
```

This allows you to build a single complex function by creating multiple custom functions from it, each with its own locked function.

JavaScript functions are incredibly versatile, and currying is just a small part of it . There are many other types of functions like arrow, constructor and anonymous. Familiarizing yourself with them and their associated components is the key to mastering JavaScript.