

# First-Class Citizen

Higher-order functions (HOFs) let you use functions as data, allowing for FP's most powerful patterns. (5 min. read)

## Higher-Order Functions

Functions operate on data, right?

Strings are data

```
const result = sayHi('User'); // 'Hi, User!'
console.log({ result });
```



Numbers are data

```
const result = double(4); // 8
console.log({ result });
```



Booleans are data

```
const result1 = getClearance(true); // 'Access granted'
const result2 = getClearance(false); // 'Access denied'
console.log({ result1, result2 });
```



Objects are data

```
const result = getFirstName({
  firstName: 'Bobo'
}); // 'Bobo'

console.log({ result });
```



## Arrays are data

```
const result = len([1, 2, 3]); // 3

console.log({ result });
```



These 5 types are considered [first-class citizens](#) in every mainstream language.

What makes them first-class? You can pass them around, store them in variables and arrays, use them as inputs for calculations. You can use them like *any piece of data*.

## Functions Can Be Data Too

JavaScript took a page from the FP book and added a sixth first-class citizen: **functions**.

# 4 Ways Functions Are Data



## Pass them to other functions

A function that takes or returns another function has become "higher-order". Higher-order functions let us abstract common actions like map, filter, and reduce.



## Set them as object properties

That makes them methods!



## Store them in arrays

Useful if you're calling a list of functions in response to an event. The JavaScript Event Loop works like this!



## Set them as variables

Referencing a function makes it easy to reuse, especially curried functions that take some params now and others later!

```
const isEven = (num) => num % 2 === 0;

const result = [1, 2, 3, 4].filter(isEven);

console.log({ result });
```



See how `filter` uses `isEven` to decide what numbers to keep? `isEven`, a function, was a parameter *to another function*.

You can also *return a function*.

```
const addNumbers = (x) => (y) => x + y;
```



`addNumbers` needs two parameters, but doesn't require them both at once. You can supply them immediately:

```
const addNumbers = (x) => (y) => x + y;

const result = addNumbers(10)(20);

console.log({ result });
```



Or one by one:

```
const addNumbers = (x) => (y) => x + y;

const add10 = addNumbers(10);
const result = add10(20);

console.log({ result });
```



A function that takes and/or returns another function is called a **higher-order function**. It's "higher-order" because it *operates on functions*, in addition to strings, numbers, arrays, etc. Pretty meta.

strings, numbers, arrays, etc. Pretty meta.

This is only possible because JavaScript made functions first-class like strings, booleans, objects, arrays, etc.

With functions, you can

- Store them as variables
- Insert them into arrays
- Assign them as object properties (methods)
- Pass them as arguments
- Return them from other functions

*Like any other piece of data.* That's the key here.

## Summary

- Strings, numbers, booleans, arrays, and objects can be stored as variables, arrays, and properties or methods.
- JavaScript treats functions the same way.
- This allows for functions that operate on other functions—**higher-order functions**.