Return values from functions

Many functions, by default, return the value of **undefined**.

An example is the **console.log()** function.

If I run:

1

console.log('Hello');





... here's the output in the console:

1

2

Hello

undefined





Because the **console.log()** function is built so as to not have the explicitly set return value, it gets the default return value of **undefined**.

I'll now code my own implementation of **console.log()**, which doesn't return the value of **undefined**:

1

2

3

4

function consoleLog(val) {

    console.log(val)

    return val

}





I'm using the **console.log()** function inside my custom **consoleLog** function declaration. And I'm specifying it to return the value of its argument.

Now when I run my custom **consoleLog()** function:

1

consoleLog('Hello')





I get the following output:

1

2

Hello

'Hello'





So, the value is output in the console, but it's also returned.

Why is this useful?

It's useful because I can use return values from one function inside another function.

Here's an example.

I'll first code a function that returns a double of a number that it received:

1

2

3

function doubleIt(num) {

    return num \* 2

}





Now I'll code another function that builds an object with a specific value:

1

2

3

4

5

function objectMaker(val) {

    return {

        prop: val

    }

}





I can call the **objectMaker()** function with any value I like, such as:

1

objectMaker(20);





The returned value will be an object with a single **prop** key set to **20**:

1

{prop:20}





Now consider this code:

1

doubleIt(10).toString()





The above code returns the number **20** as a string, that is: **"20"**.

I can even combine my custom function calls as follows:

1

objectMaker( doubleIt(100) );





This will now return the following value:

1

{prop: 200}





What does all of this mean?

It means that by JavaScript allowing me to use the **return** keyword as described above, I can have multiple function calls, returning data and manipulating values, based on whatever coding challenge I have in front of me.

Being able to return custom values is one of the foundations that makes functional programming possible.

# The functional programming paradigm

## Learning Objectives

* Be able to explain that there are several programming paradigms
* Be able to explain the basic difference between the two predominant programming paradigms: functional programming and object-oriented programming
* Understand, at a very high level, how the functional programming paradigm works

"There are actually several styles of coding, also known as **paradigms**. A common style is called **functional programming**, or FP for short.

In functional programming, we use a lot of functions and variables.

1

2

3

4

5

6

7

function getTotal(a,b) {

    return a + b

}

var num1 = 2;

var num2 = 3;

var total = getTotal(num1, num2);





When writing FP code, we keep data and functionality separate and pass data into functions only when we want something computed.

1

2

3

4

5

6

function getDistance(mph, h) {

    return mph \* h

}

var mph = 60;

var h = 2;

var distance = getDistance(mph, h);





In functional programming, functions return new values and then use those values somewhere else in the code.

1

2

3

4

5

6

7

8

function getDistance(mph, h) {

    return mph \* h

}

var mph = 60;

var h = 2;

var distance = getDistance(mph, h);

console.log(distance); // <====== THIS HERE!





Another style is **object-oriented programming (OOP)**. In this style, we group data and functionality as properties and methods inside objects.

For example, if I have a **virtualPet** object, I can give it a **sleepy** property and a **nap()** method:

1

2

3

4

var virtualPet = {

    sleepy: true,

    nap: function() {}

}





In OOP, methods **update properties** stored in the object instead of generating new return values.

For example, if I check the **sleepy** property on the **virtualPet** object, I can confirm that it's set to **true**.

However, once I've ran the **nap()** method on the **virtualPet** object, will the **sleepy** property's value change?

1

2

3

4

5

6

7

8

9

10

//creating an object

var virtualPet = {

    sleepy: true,

    nap: function() {

        this.sleepy = false

    }

}

console.log(virtualPet.sleepy) // true

virtualPet.nap()

console.log(virtualPet.sleepy) // false





OOP helps us model real-life objects. It works best when the grouping of properties and data in an object makes logical sense - meaning, the properties and methods "belong together".

Note that the goal here is not to discuss OOP in depth; instead, I just want to show you the simplest explanation of what it is and how it works, in order to make the single most important distinction between FP and OOP.

To summarize this point, we can say that the Functional Programming paradigm works by keeping the data and functionality separate. It's counterpart, OOP, works by keeping the data and functionality grouped in meaningful objects.

There are many more concepts and ideas in functional programming.

Here are some of the most important ones:

* First-class functions
* Higher-order function
* Pure functions and side-effects

There are many other concepts and priciples in functional programming, but for now, let's stick to these three.

### First-class functions

It is often said that functions in JavaScript are “first-class citizens”. What does that mean?

It means that a function in JavaScript is just another value that we can:

* pass to other functions
* save in a variable
* return from other functions

In other words, a function in JavaScript is just a value - from this vantage point, almost no different then a string or a number. For example, in JavaScript, it's perfectly normal to pass a function invocation to another function. To explain how this works, consider the following program.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

function addTwoNums(a, b) {

    console.log(a + b)

}

function randomNum() {

    return Math.floor((Math.random() \* 10) + 1);

}

function specificNum() { return 42 };

var useRandom = true;

var getNumber;

if(useRandom) {

    getNumber = randomNum

} else {

    getNumber = specificNum

}

addTwoNums(getNumber(), getNumber())





I start the program with the **addTwoNums()** function whose definition I've already used earlier in various variations. The reason why this function is a recurring example is because it's so simple that it helps explain concepts that otherwise might be a bit harder to grasp. Next, I code a function named **randomNum()** which returns a random number between 0 and 10. I then code another function named **specificNum()** which returns a specific number, the number 42. Next, I save a variable named **useRandom**, and I set it to the boolean value of **true**. I declare another variable, named **getNumber**. This is where things get interesting. On the next several lines, I have an if else statement. The if condition is executed when the value of **useRandom** is set to **true**. If that's the case, the entire **randomNum()** function's declaration is saved into the **getNumber** variable. Otherwise, I'm saving the entire **specificNum()** function's declaration into the **getNumber** variable. In other words, based on the **useRandom** being set to **true** or **false**, the **getNumber** variable will be assigned either the **randomNum()** function declaration or the **specificNum()** function declaration. With all this code set, I can then invoke the **addTwoNums()** function, passing it the invocation of the **getNumber()** variables as its first and second arguments. **This works because functions in JavaScript are truly first-class citizens, which can be assigned to variable names and passed around just like I would pass around a string, a number, an object, etc.** Note: most of the code inside the **randomNum()** function declaration comes from a previous lesson, namely the lesson that discussed the Math object in JavaScript. This brings me to the second foundational concept of functional programming, which is the concept of higher-order functions.

**Higher-order functions**

A higher-order function is a function that has either one or both of the following characteristics:

* It accepts other functions as arguments
* It returns functions when invoked

There's no "special way" of defining higher-order functions in JavaScript. It is simply a feature of the language. The language itself allows me to pass a function to another function, or to return a function from another function. Continuing from the previous section, consider the following code, in which I'm re-defining the **addTwoNums()** function so that it is a higher-order function:

1

2

3

function addTwoNums(getNumber1, getNumber2) {

    console.log(getNumber1() + getNumber2());

}





You can think of the above function declaration of **addTwoNums** as describing how it will deal with the **getNumber1** and **getNumber2** inputs: once it receives them as arguments, it will then attempt invoking them and concatenating the values returned from those invocations. For example:

1

2

addTwoNums(specificNum, specificNum); // returned number is 84

addTwoNums(specificNum, randomNum); // returned number is 42 + some random number





### Pure functions and side-effects

Another concept of functional programming are pure functions.

A pure function returns the exact same result as long as it's given the same values.

An example of a pure function is the **addTwoNums()** function from the previous section:

function addTwoNums(a, b) {

    console.log(a + b)

}





This function will always return the same output, based on the input. For example, as long as we give it a specific value, say, a **5**, and a **6**:

1

addTwoNums(5,6); // 11





... the output will always be the same.

Another rule for a function to be considered pure is that it should not have side-effects. A side-effect is any instance where a function makes a change outside of itself.

This includes:

* changing variable values outside of the function itself, or even relying on outside variables
* calling a Browser API (even the console itself!)
* calling **Math.random()** - since the value cannot be reliably repeated

The topic of pure and impure functions can get somewhat complex.

For now, it's sufficient to know that this concept exists and that it is related to functional programming.

# The functional programming paradigm

## Learning Objectives

* Be able to explain that there are several programming paradigms
* Be able to explain the basic difference between the two predominant programming paradigms: functional programming and object-oriented programming
* Understand, at a very high level, how the functional programming paradigm works

"There are actually several styles of coding, also known as **paradigms**. A common style is called **functional programming**, or FP for short.

In functional programming, we use a lot of functions and variables.

1

2

3

4

5

6

7

function getTotal(a,b) {

    return a + b

}

var num1 = 2;

var num2 = 3;

var total = getTotal(num1, num2);





When writing FP code, we keep data and functionality separate and pass data into functions only when we want something computed.

1

2

3

4

5

6

function getDistance(mph, h) {

    return mph \* h

}

var mph = 60;

var h = 2;

var distance = getDistance(mph, h);





In functional programming, functions return new values and then use those values somewhere else in the code.

1

2

3

4

5

6

7

8

function getDistance(mph, h) {

    return mph \* h

}

var mph = 60;

var h = 2;

var distance = getDistance(mph, h);

console.log(distance); // <====== THIS HERE!





Another style is **object-oriented programming (OOP)**. In this style, we group data and functionality as properties and methods inside objects.

For example, if I have a **virtualPet** object, I can give it a **sleepy** property and a **nap()** method:

1

2

3

4

var virtualPet = {

    sleepy: true,

    nap: function() {}

}





In OOP, methods **update properties** stored in the object instead of generating new return values.

For example, if I check the **sleepy** property on the **virtualPet** object, I can confirm that it's set to **true**.

However, once I've ran the **nap()** method on the **virtualPet** object, will the **sleepy** property's value change?

1

2

3

4

5

6

7

8

9

10

//creating an object

var virtualPet = {

    sleepy: true,

    nap: function() {

        this.sleepy = false

    }

}

console.log(virtualPet.sleepy) // true

virtualPet.nap()

console.log(virtualPet.sleepy) // false





OOP helps us model real-life objects. It works best when the grouping of properties and data in an object makes logical sense - meaning, the properties and methods "belong together".

Note that the goal here is not to discuss OOP in depth; instead, I just want to show you the simplest explanation of what it is and how it works, in order to make the single most important distinction between FP and OOP.

To summarize this point, we can say that the Functional Programming paradigm works by keeping the data and functionality separate. It's counterpart, OOP, works by keeping the data and functionality grouped in meaningful objects.

There are many more concepts and ideas in functional programming.

Here are some of the most important ones:

* First-class functions
* Higher-order function
* Pure functions and side-effects

There are many other concepts and priciples in functional programming, but for now, let's stick to these three.

### First-class functions

It is often said that functions in JavaScript are “first-class citizens”. What does that mean?

It means that a function in JavaScript is just another value that we can:

* pass to other functions
* save in a variable
* return from other functions

In other words, a function in JavaScript is just a value - from this vantage point, almost no different then a string or a number. For example, in JavaScript, it's perfectly normal to pass a function invocation to another function. To explain how this works, consider the following program.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

function addTwoNums(a, b) {

    console.log(a + b)

}

function randomNum() {

    return Math.floor((Math.random() \* 10) + 1);

}

function specificNum() { return 42 };

var useRandom = true;

var getNumber;

if(useRandom) {

    getNumber = randomNum

} else {

    getNumber = specificNum

}

addTwoNums(getNumber(), getNumber())





I start the program with the **addTwoNums()** function whose definition I've already used earlier in various variations. The reason why this function is a recurring example is because it's so simple that it helps explain concepts that otherwise might be a bit harder to grasp. Next, I code a function named **randomNum()** which returns a random number between 0 and 10. I then code another function named **specificNum()** which returns a specific number, the number 42. Next, I save a variable named **useRandom**, and I set it to the boolean value of **true**. I declare another variable, named **getNumber**. This is where things get interesting. On the next several lines, I have an if else statement. The if condition is executed when the value of **useRandom** is set to **true**. If that's the case, the entire **randomNum()** function's declaration is saved into the **getNumber** variable. Otherwise, I'm saving the entire **specificNum()** function's declaration into the **getNumber** variable. In other words, based on the **useRandom** being set to **true** or **false**, the **getNumber** variable will be assigned either the **randomNum()** function declaration or the **specificNum()** function declaration. With all this code set, I can then invoke the **addTwoNums()** function, passing it the invocation of the **getNumber()** variables as its first and second arguments. **This works because functions in JavaScript are truly first-class citizens, which can be assigned to variable names and passed around just like I would pass around a string, a number, an object, etc.** Note: most of the code inside the **randomNum()** function declaration comes from a previous lesson, namely the lesson that discussed the Math object in JavaScript. This brings me to the second foundational concept of functional programming, which is the concept of higher-order functions.

**Higher-order functions**

A higher-order function is a function that has either one or both of the following characteristics:

* It accepts other functions as arguments
* It returns functions when invoked

There's no "special way" of defining higher-order functions in JavaScript. It is simply a feature of the language. The language itself allows me to pass a function to another function, or to return a function from another function. Continuing from the previous section, consider the following code, in which I'm re-defining the **addTwoNums()** function so that it is a higher-order function:

1

2

3

function addTwoNums(getNumber1, getNumber2) {

    console.log(getNumber1() + getNumber2());

}





You can think of the above function declaration of **addTwoNums** as describing how it will deal with the **getNumber1** and **getNumber2** inputs: once it receives them as arguments, it will then attempt invoking them and concatenating the values returned from those invocations. For example:

1

2

addTwoNums(specificNum, specificNum); // returned number is 84

addTwoNums(specificNum, randomNum); // returned number is 42 + some random number





### Pure functions and side-effects

Another concept of functional programming are pure functions.

A pure function returns the exact same result as long as it's given the same values.

An example of a pure function is the **addTwoNums()** function from the previous section:

1

2

3

function addTwoNums(a, b) {

    console.log(a + b)

}





This function will always return the same output, based on the input. For example, as long as we give it a specific value, say, a **5**, and a **6**:

1

addTwoNums(5,6); // 11





... the output will always be the same.

Another rule for a function to be considered pure is that it should not have side-effects. A side-effect is any instance where a function makes a change outside of itself.

This includes:

* changing variable values outside of the function itself, or even relying on outside variables
* calling a Browser API (even the console itself!)
* calling **Math.random()** - since the value cannot be reliably repeated

The topic of pure and impure functions can get somewhat complex.

For now, it's sufficient to know that this concept exists and that it is related to functional programming.

# Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[MDN Functions Guide](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions)

[MDN Glossary: Recursion](https://developer.mozilla.org/en-US/docs/Glossary/Recursion)

[MDN Glossary: Scope](https://developer.mozilla.org/en-US/docs/Glossary/Scope)

[Functional Programming in JavaScript](https://www.toptal.com/javascript/functional-programming-javascript)

[MDN: First-class functions](https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function)