**Advanced Javascript**

1. **Arrow functions**
2. **Template Literals**

**How to Use JavaScript Arrow Functions – Explained in Detail**

**Hello everyone! In this article, I’m going to explain one of the most useful features in JavaScript: the arrow function.**

**I’ll compare the arrow function with the regular function syntax, I'll show you how to convert a regular function into an arrow function easily, and I'll discuss why the arrow function syntax is recommended over the regular function.**

**Here's what we'll cover:**

1. [**What Is the Arrow Function Syntax?**](https://www.freecodecamp.org/news/javascript-arrow-functions-in-depth/#heading-what-is-the-arrow-function-syntax)
2. [**How to Convert a Regular Function to an Arrow Function Easily**](https://www.freecodecamp.org/news/javascript-arrow-functions-in-depth/#heading-how-to-convert-a-regular-function-to-an-arrow-function-easily)
3. [**Why Arrow Functions Are Recommended Over Regular Functions**](https://www.freecodecamp.org/news/javascript-arrow-functions-in-depth/#heading-why-arrow-functions-are-recommended-over-regular-functions)
4. [**Arrow Functions Are Better for Short Functions**](https://www.freecodecamp.org/news/javascript-arrow-functions-in-depth/#heading-arrow-functions-are-better-for-short-functions)
5. [**Arrow Functions Have an Implicit Return Statement**](https://www.freecodecamp.org/news/javascript-arrow-functions-in-depth/#heading-arrow-functions-have-an-implicit-return-statement)
6. [**Arrow Functions Don’t Have this Binding**](https://www.freecodecamp.org/news/javascript-arrow-functions-in-depth/#heading-arrow-functions-dont-have-this-binding)
7. [**When You Should Not Use Arrow Functions?**](https://www.freecodecamp.org/news/javascript-arrow-functions-in-depth/#heading-when-you-should-not-use-arrow-functions)
8. [**Conclusion**](https://www.freecodecamp.org/news/javascript-arrow-functions-in-depth/#heading-conclusion)

**Let’s dive in!**

**What Is the Arrow Function Syntax?**

**When you need to create a function in JavaScript, the main method is to use the function keyword followed by the function name as shown below:**

**function greetings(name) {**

**console.log(`Hello, ${name}!`);**

**}**

**greetings('John'); // Hello, John!**

**The arrow function syntax allows you to create a function expression that produces the same result as the code above.**

**Here’s the greetings() function again, but using the arrow function syntax:**

**const greetings = name => {**

**console.log(`Hello, ${name}!`);**

**};**

**greetings('John'); // Hello, John!**

**When you declare a function with the arrow function syntax, you need to assign the declaration to a variable so that the function has a name.**

**Basically, the arrow function syntax looks as follows:**

**const myFunction = (param1, param2, ...) => {**

**// function body**

**}**

**In the code above, the myFunction is the variable that holds the function. You can call the function as myFunction() later in your code.**

**(param1, param2, ...) are the function parameters. You can define as many parameters as required by the function.**

**Then you have the arrow => to indicate the beginning of the function. After that, you can write curly brackets {} to indicate the function body, or remove them if you have a single-line function. More on this later.**

**At first, the arrow function may seem weird as you are used to seeing the function keyword. But as you start using the arrow syntax, you will see that it’s very convenient and easier to write.**

**Let me show you an easy way to convert a regular function to an arrow function next.**

**How to Convert a Regular Function to an Arrow Function Easily**

**You can follow these three easy steps to convert a regular function to an arrow function:**

1. **Replace the function keyword with the variable keyword const**
2. **Add the = symbol after the function name and before the parentheses**
3. **Add the => symbol after the parentheses**

**Usually, a function is never changed after the declaration, so we use the const keyword instead of let.**

**The code below should help you visualize the steps:**

**function greetings(name) {**

**return `Hello, ${name}!`;**

**}**

**// step 1: replace function with const**

**const greetings(name) {**

**return `Hello, ${name}!`;**

**}**

**// step 2: add = after the function name**

**const greetings = (name) {**

**return `Hello, ${name}!`;**

**}**

**// step 3: add => after the parentheses**

**const greetings = (name) => {**

**return `Hello, ${name}!`;**

**}**

**The three steps above are enough to convert any old JavaScript function syntax to the new arrow function syntax.**

**When you have a single line function, there’s a fourth optional step to remove the curly brackets and the return keyword as follows:**

**// from this**

**const greetings = (name) => {**

**return `Hello, ${name}!`;**

**};**

**// to this**

**const greetings = (name) => `Hello, ${name}!`;**

**When you have exactly one parameter, you can also remove the parentheses:**

**// from this**

**const greetings = (name) => `Hello, ${name}!`;**

**// to this**

**const greetings = name => `Hello, ${name}!`;**

**But the last two steps are optional. Only the first three steps are required to convert any JavaScript function created using the function keyword into the arrow function syntax.**

**Why Arrow Functions Are Recommended Over Regular Functions**

**The arrow function syntax offers improvements to the way you write a function in JavaScript, such as:**

* **You can write short functions in a more straightforward manner**
* **For single-line functions, the return statement can be implicit**
* **The this keyword is not bound to the function.**

**Let’s see how these improvements work with practical examples next.**

**Arrow Functions Are Better for Short Functions**

**Suppose you have a single-line function that prints a string to the console. Using the function keyword, here’s how you would write the function:**

**function greetings(name) {**

**console.log(`Hello, ${name}!`);**

**}**

**If you use the arrow function syntax, you can omit the curly brackets, creating a single-line function as shown below:**

**const greetings = (name) => console.log(`Hello, ${name}!`);**

**Even more, you can remove the parentheses that surround the function parameters when you have exactly one parameter:**

**const greetings = name => console.log(`Hello, ${name}!`);**

**If your function has no parameter, then you need to pass empty parentheses between the assignment and the arrow syntax as shown below:**

**const greetings = () => console.log(`Hello, World!`);**

**When using the arrow function syntax, the curly brackets are required only when your function is more than a single line. For example:**

**const greetings = () => {**

**console.log('Hello World!');**

**console.log('How are you?');**

**};**

**When you use the regular function keyword, you can’t omit the curly brackets no matter what.**

**Arrow functions are also great for situations where you don’t need to name the function, such as callbacks:**

**const myArray = [1, 2, 3];**

**// From this:**

**myArray.forEach(function (item) {**

**console.log(item);**

**});**

**// To this:**

**myArray.forEach(item => console.log(item));**

**Or when you need to create an Immediately Invoked Function Expression (IIFE):**

**// From this:**

**(function () {**

**console.log('Hello World');**

**})();**

**// To this:**

**(() => console.log('Hello World'))();**

**As you can see, using the arrow function syntax makes your code much more clean and concise.**

**Arrow Functions Have an Implicit Return Statement**

**When you have a single-line arrow function, the return statement will be added implicitly by JavaScript. This means you shouldn't add the return keyword explicitly.**

**To show you what I mean, suppose you have a function that sums two numbers as follows:**

**function sum(a, b) {**

**return a + b;**

**}**

**When you write the function above using the arrow function syntax, you need to remove the curly brackets and the return keyword:**

**const sum = (a, b) => a + b;**

**If you didn’t remove the return keyword, then JavaScript will throw an error, saying an opening curly bracket { is expected.**

**When you use arrow functions, only write the return statement explicitly when you have multi-line statements:**

**const sum = (a, b) => {**

**const result = a + b;**

**return result;**

**};**

**When you remove the curly brackets, don’t forget to remove the return keyword if you use it.**

**Arrow Functions Don’t Have this Binding**

**One significant difference between the arrow function and the regular function syntax is in how they handle the this keyword.**

**In a regular function, the this keyword refers to the object from which you call the function. In an arrow function, the this keyword refers to the object from which you define the function.**

**To show you what I mean, suppose you have a person with the following properties and methods:**

**const person = {**

**name: 'Nathan',**

**skills: ['HTML', 'CSS', 'JavaScript'],**

**showSkills() {**

**this.skills.forEach(function (skill) {**

**console.log(`${this.name} is skilled in ${skill}`);**

**});**

**},**

**};**

**person.showSkills();**

**If you run the code above, the result of calling the showSkills() method would be:**

**undefined is skilled in HTML**

**undefined is skilled in CSS**

**undefined is skilled in JavaScript**

**Here, the this keyword refers to the global Window object because we called the showSkills() method outside of the person object.**

**In the global object, the name property is undefined. Now, let’s rewrite the callback function using the arrow syntax:**

**const person = {**

**name: 'Nathan',**

**skills: ['HTML', 'CSS', 'JavaScript'],**

**showSkills() {**

**this.skills.forEach(skill => {**

**console.log(`${this.name} is skilled in ${skill}`);**

**});**

**},**

**};**

**person.showSkills();**

**Run the code again, and the result would be:**

**Nathan is skilled in HTML**

**Nathan is skilled in CSS**

**Nathan is skilled in JavaScript**

**Here, the this keyword refers to the object from which the arrow function is defined, which is the person object.**

**This one behavior is what makes people prefer arrow functions, because it makes more sense to have this refer to the object from which you define that function rather than from which you call it.**

**When You Should Not Use Arrow Functions?**

**Arrow functions are typically preferred over standard functions, but there are a few situations when you shouldn't use the arrow function.**

**One of these situations is when you define an object method. Back to our person object example above, suppose you write the showSkills() method as an arrow function like this:**

**const person = {**

**name: 'Nathan',**

**skills: ['HTML', 'CSS', 'JavaScript'],**

**showSkills: () => {**

**this.skills.forEach(skill => {**

**console.log(`${this.name} is skilled in ${skill}`);**

**});**

**},**

**};**

**person.showSkills();**

**Running the above code will cause an error:**

**TypeError: Cannot read properties of undefined (reading 'forEach')**

**When inside an object, the this keyword refers to the current object only when you declare the method using the standard syntax (methodName() or methodName: function(){ })**

**When you declare an object method using the arrow function, the this keyword refers to the global object, and the skills property is undefined there. Never use the arrow function when declaring a method.**

**JavaScript Template Literals**

Template Literal in ES6 provides new features to create a string that gives more control over dynamic strings. Traditionally, String is created using **single quotes (‘)**or **double quotes (“)** quotes. Template literal is created using the **backtick (`)** character.

**Syntax:**

let s=`some string`;

**Multiline Strings**

In order to create a multiline string an escape sequence **\n** was used to give a new line character. However, Template Literals there is no need to add **\n** string ends only when it gets **backtick (`)** character.

**Example:**This example shows the use of the multiline strings.

javascript

*// Without template literal*

console.log('Some text that I want \non two lines!');

*// With template literal*

console.log(`Some text that I want

on two lines!`);

**Output:**

Some text that I want  
on two lines!  
Some text that I want  
on two lines!

**Expressions**

To dynamically add values into new Template Literals expressions are used. The ${} syntax allows an expression in it that produces the value. This value can be a string stored in a variable or a computation operation.

${expression}

**Example:** The code below shows the use of expressions in template literals.

JavaScript

**let** principal = 1000;

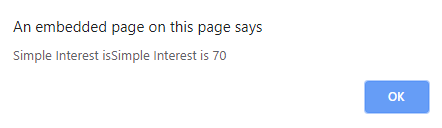
**let** noofyears = 1;

**let** rateofinterest = 7;

**let** SI = `Simple Interest is **${**(principal \*

noofyears \* rateofinterest) / 100**}**`;

alert("Simple Interest is" + SI);

**Output:**

**Tagged Templates**

One of the features of Template Literals is its ability to create Tagged Template Literals. Tagged Literal is written like a function definition, but the difference is when this literal is called. There is no parenthesis() to a literal call. An array of Strings are passed as a parameter to a literal.

**Example 1:**This example shows the use of the tagged templates.

JavaScript

**function** TaggedLiteralEg(strings) {

document.write(strings);

}

TaggedLiteralEg`GeeksforGeeks`;

**Output:**

GeeksforGeeks

**Example 2:** It is also possible to pass values to a tagged literal. This value can be a result of some expression or a value fetched from the variable. The code below shows the use of Tagged Literal.

JavaScript

**function** TaggedLiteralEg(strings, value, value2) {

console.log(strings);

console.log(value2 + " " + value);

}

**let** text = 'GeeksforGeeks';

TaggedLiteralEg`test **${**text**}** **${**2 + 3**}**`;

**Output:**

test , ,  
5 GeeksforGeeks

**Raw String**

Raw method of template literal allows access of raw strings as they were entered, without processing escape sequences. In addition, the String.raw() method exists to create raw strings just like the default template function, and string concatenation would create.

**Example:** This example shows the use of the raw strings.

javascript

**let** s = String.raw`Welcome to GeeksforGeeks Value of expression is **${**2 + 3**}**`;

console.log(s);

**Output:**

Welcome to GeeksforGeeks Value of expression is 5

**Nested Templates**

Templates can be nested if it contains multiple expression evaluation or multiple condition checking. Instead of using else if ladder this is readable and gives ease to the developer. The code below finds the maximum of three numbers using conditional operator and nested template literal.

**Example:** This example shows the use of the nested templates.

javascript

**function** maximum(x, y, z) {

**let** c = `value **${**(y > x && y > z) ? 'y is greater' :

`**${**x > z ? 'x is greater' : 'z is greater'**}**`**}**`;

**return** (c);

}

console.log(maximum(5, 11, 15));

console.log(maximum(15, 11, 3));

console.log(maximum(11, 33, 2));

**Output:**

value z is greater  
value x is greater  
value y is greater

Want to be a Software Developer or a Working Professional looking to enhance your **Software Development Skills**? Then, master the concepts of Full-Stack Development. Our [**Full Stack Development - React and Node.js Course**](https://gfgcdn.com/tu/Q3I/) will help you achieve this quickly. Learn everything from **Front-End to Back-End Development** with hands-on **Projects**and real-world examples. This course enables you to build scalable, efficient, dynamic web applications that stand out. Ready to become an expert in Full-Stack? Enroll Now and Start Creating the Future!

**JavaScript Destructuring Assignment**

**JavaScript Destructuring**

The destructuring assignment introduced in [ES6](https://www.programiz.com/javascript/ES6) makes it easy to assign [array](https://www.programiz.com/javascript/array) values and [object properties](https://www.programiz.com/javascript/object#properties) to distinct [variables](https://www.programiz.com/javascript/variables-constants). For example,  
  
**Before ES6:**

// assigning object attributes to variables

const person = {

name: 'Sara',

age: 25,

gender: 'female'

}

let name = person.name;

let age = person.age;

let gender = person.gender;

console.log(name); // Sara

console.log(age); // 25

console.log(gender); // female

[Run Code](https://www.programiz.com/javascript/online-compiler)

**From ES6:**

// assigning object attributes to variables

const person = {

name: 'Sara',

age: 25,

gender: 'female'

}

// destructuring assignment

let { name, age, gender } = person;

console.log(name); // Sara

console.log(age); // 25

console.log(gender); // female

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Note**: The order of the name does not matter in object destructuring.

For example, you could write the above program as:

let { age, gender, name } = person;

console.log(name); // Sara

**Note**: When destructuring objects, you should use the same name for the variable as the corresponding object key.

For example,

let {name1, age, gender} = person;

console.log(name1); // undefined

If you want to assign different variable names for the object key, you can use:

const person = {

name: 'Sara',

age: 25,

gender: 'female'

}

// destructuring assignment

// using different variable names

let { name: name1, age: age1, gender:gender1 } = person;

console.log(name1); // Sara

console.log(age1); // 25

console.log(gender1); // female

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Array Destructuring**

You can also perform array destructuring in a similar way. For example,

const arrValue = ['one', 'two', 'three'];

// destructuring assignment in arrays

const [x, y, z] = arrValue;

console.log(x); // one

console.log(y); // two

console.log(z); // three

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Assign Default Values**

You can assign the default values for variables while using destructuring. For example,

let arrValue = [10];

// assigning default value 5 and 7

let [x = 5, y = 7] = arrValue;

console.log(x); // 10

console.log(y); // 7

[Run Code](https://www.programiz.com/javascript/online-compiler)

In the above program, arrValue has only one element. Hence,

* the x variable will be **10**
* the y variable takes the default value **7**

In object destructuring, you can pass default values in a similar way. For example,

const person = {

name: 'Jack',

}

// assign default value 26 to age if undefined

const { name, age = 26} = person;

console.log(name); // Jack

console.log(age); // 26

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Swapping Variables**

In this example, two variables are swapped using the destructuring assignment syntax.

// program to swap variables

let x = 4;

let y = 7;

// swapping variables

[x, y] = [y, x];

console.log(x); // 7

console.log(y); // 4

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Skip Items**

You can skip unwanted items in an array without assigning them to local variables. For example,

const arrValue = ['one', 'two', 'three'];

// destructuring assignment in arrays

const [x, , z] = arrValue;

console.log(x); // one

console.log(z); // three

[Run Code](https://www.programiz.com/javascript/online-compiler)

In the above program, the second element is omitted by using the comma separator ,.

**Assign Remaining Elements to a Single Variable**

You can assign the remaining elements of an array to a variable using the spread syntax .... For example,

const arrValue = ['one', 'two', 'three', 'four'];

// destructuring assignment in arrays

// assigning remaining elements to y

const [x, ...y] = arrValue;

console.log(x); // one

console.log(y); // ["two", "three", "four"]

[Run Code](https://www.programiz.com/javascript/online-compiler)

Here, one is assigned to the x variable. And the rest of the array elements are assigned to y variable.

You can also assign the rest of the object properties to a single variable. For example,

const person = {

name: 'Sara',

age: 25,

gender: 'female'

}

// destructuring assignment

// assigning remaining properties to rest

let { name, ...rest } = person;

console.log(name); // Sara

console.log(rest); // {age: 25, gender: "female"}

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Note**: The variable with the spread syntax cannot have a trailing comma ,. You should use this rest element (variable with spread syntax) as the last variable.

For example,

const arrValue = ['one', 'two', 'three', 'four'];

// throws an error

const [ ...x, y] = arrValue;

console.log(x); // eror

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Nested Destructuring Assignment**

You can perform nested destructuring for array elements. For example,

// nested array elements

const arrValue = ['one', ['two', 'three']];

// nested destructuring assignment in arrays

const [x, [y, z]] = arrValue;

console.log(x); // one

console.log(y); // two

console.log(z); // three

[Run Code](https://www.programiz.com/javascript/online-compiler)

Here, the variable y and z are assigned nested elements two and three.

In order to execute the nested destructuring assignment, you have to enclose the variables in an array structure (by enclosing inside []).

You can also perform nested destructuring for object properties. For example,

const person = {

name: 'Jack',

age: 26,

hobbies: {

read: true,

playGame: true

}

}

// nested destructuring

const {name, hobbies: {read, playGame}} = person;

console.log(name); // Jack

console.log(read); // true

console.log(playGame); // true

[Run Code](https://www.programiz.com/javascript/online-compiler)

In order to execute the nested destructuring assignment for objects, you have to enclose the variables in an object structure (by enclosing inside {}).