An Introduction to JavaScript Arrow Function

**Summary**: in this tutorial, you will learn how to use the JavaScript arrow function to write more concise code for function expressions.

Introduction to JavaScript arrow function

The arrow function provides you with an alternative way to write a shorter syntax compared to the [function expression](http://www.javascripttutorial.net/javascript-function-type/). See the following example.

|  |  |
| --- | --- |
| 1  2  3  4 | var add = function(x,y) {    return x + y;  }  console.log(add(10,20)); // 30 |

In this example, the add() function expression returns the sum of two numbers. The following example uses the arrow function that is equivalent to the add() function expression above.

|  |  |
| --- | --- |
| 1  2 | var add = (x,y) => x + y;  console.log(add(10,20)); // 30; |

The  typeof operator returns function indicating the type of the arrow function.

|  |  |
| --- | --- |
| 1 | console.log(typeof add); // function |

Similarly, an arrow function is also an instance of the [Function type](http://www.javascripttutorial.net/javascript-function-type/) as shown in the following example.

|  |  |
| --- | --- |
| 1 | console.log(add instanceof Function); // true |

JavaScript arrow function with multiple parameters

For the arrow function that takes two or more arguments, you use the following syntax:

|  |  |
| --- | --- |
| 1 | (p1, p2, ..., pn) => expression; |

The following expression:

|  |  |
| --- | --- |
| 1 | => expression |

is equivalent to the expression shown below:

|  |  |
| --- | --- |
| 1 | => { return expression; } |

For example, to [sort an array](http://www.javascripttutorial.net/javascript-array-sort/) of numbers in descending orders, you use the sort() method of the array object as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var numbers = [4,2,6];  numbers.sort(function(a,b){      return b - a;  });  console.log(numbers); // [6,4,2] |

By using the arrow function syntax, it is shorter as follows:

|  |  |
| --- | --- |
| 1  2  3 | var numbers = [4,2,6];  numbers.sort((a,b) => b - a);  console.log(numbers); // [6,4,2] |

JavaScript arrow function with a single parameter

For the arrow function that takes a single parameter, you use the following syntax:

|  |  |
| --- | --- |
| 1 | (p1) => { statements } |

Note that you can also omit the parentheses as follows:

|  |  |
| --- | --- |
| 1 | p => { statements } |

The following example uses an arrow function as the argument of the map() method that [transforms an array](http://www.javascripttutorial.net/javascript-array-map/) of strings into an array of the string’s lengths.

|  |  |
| --- | --- |
| 1  2  3  4 | var names = ['John', 'Mac', 'Peter'];  var lengths = names.map(name => name.length);    console.log(lengths); |

JavaScript arrow function with no parameter

If the arrow function has no parameter, you must use the parentheses as follows:

|  |  |
| --- | --- |
| 1 | () => { statements } |

See the following example.

|  |  |
| --- | --- |
| 1  2 | var logDoc = () => console.log(window.document);  logDoc(); |

Line break between parameter definition and arrow

JavaScript does not allow you to use a line break between the parameter definition and the arrow (=>) of the arrow function. For example, the following code causes a SyntaxError:

|  |  |
| --- | --- |
| 1  2 | var multiply = (x,y)  => x \* y; |

However, the following example works perfectly fine.

|  |  |
| --- | --- |
| 1  2 | var multiply = (x,y) =>  x \* y; |

JavaScript allows you to use the line break between parameters as shown in the following example.

|  |  |
| --- | --- |
| 1  2  3  4  5 | var multiply = (    x,    y  ) =>  x \* y; |

Statements & expressions in the arrow function body

In JavaScript, an expression evaluates to a value as shown in the following example.

|  |  |
| --- | --- |
| 1 | 10 + 20; |

An expression also does a specific task such as:

|  |  |
| --- | --- |
| 1  2  3 | if (x === y) {      console.log('x equals y');  } |

If you use an expression in the body of an arrow function, you don’t need to use the curly braces.

|  |  |
| --- | --- |
| 1 | var square = x => x \* x; |

However, if you use a statement, you must wrap it inside a pair of curly braces as in the following example:

|  |  |
| --- | --- |
| 1  2  3 | var except = msg => {      throw msg;  }; |

JavaScript arrow function and object literal

Consider the following example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var setColor = function (color) {      return {value: color}  };    var backgroundColor = setColor('Red');  console.log(backgroundColor.value); // "Red" |

The setColor() function expression returns an object that has the value property set to the color argument. If you try the following syntax to return an object literal from an arrow function, you will get an error.

|  |  |
| --- | --- |
| 1 | p => {object:literal} |

For example, the following code causes an error.

|  |  |
| --- | --- |
| 1 | var setColor = color => {value: color }; |

Because both block and object literal use curly brackets, JavasScript engine cannot distinguish between a block and an object.

To fix this, you need to wrap the object literal in parentheses as follows:

|  |  |
| --- | --- |
| 1 | var setColor = color => ({value: color }); |

Arrow function vs. normal function

There are two main differences between an arrow function and a normal function.

1. First, in the arrow function, the this, arguments, super, new.target are lexical. It means that the arrow function uses these variables (or constructs) from the enclosing lexical scope.
2. Second, an arrow function cannot be used as a constructor. If you try to use the new keyword to create a new object from an arrow function, you will get an error.

JavaScript arrow function and this

In JavaScript, a new function defines its own this value. However, it is not the case for the arrow function. See the following example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | function Car() {      this.speed = 0;        this.speedUp = function (speed) {          this.speed = speed;          setTimeout(function () {              console.log(this.speed); // undefined          }, 1000);        };  }    var car = new Car();  car.speedUp(50); |

Inside the anonymous function of the setTimeOut() function, the this.speed is undefined. The reason is that the this of the anonymous function shadows the this of the speedUp() method.

To fix this, you often assign the this to a variable that doesn’t shadow inside the anonymous function as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | function Car() {      this.speed = 0;        this.speedUp = function (speed) {          this.speed = speed;          var self = this;          setTimeout(function () {              console.log(self.speed);          }, 1000);        };  }    var car = new Car();  car.speedUp(50); // 50; |

Unlike an anonymous function, an arrow function captures the this of the enclosing context instead of creating its own this context. The following code should work as expected.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | function Car() {      this.speed = 0;        this.speedUp = function (speed) {          this.speed = speed;          setTimeout(              () => console.log(this.speed),              1000);        };  }    var car = new Car();  car.speedUp(50); // 50; |

JavaScript arrow function and arguments

An arrow function does not have the arguments object. Therefore the arguments is a reference to the name in the enclosing scope. See the following example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function foo() {      return x => x + arguments[0];  }    var bar = foo(10, 20);  var result = bar(5);  console.log(result); // 15 |

The arrow function inside the foo function references to the arguments object. However, this arguments object belongs to the foo() function, not the arrow function.

In practice, it is recommended that you use the arrow function for the callback function because the syntax of the arrow function is cleaner and also it is safer to use the this value.

In this tutorial, you have learned about JavaScript arrow function syntax and when to apply the arrow functions to make the code cleaner.

TypeScript Arrays

An array is a user-defined data type. An array is a homogenous collection of similar types of elements that have a contiguous memory location and which can store multiple values of different data types.

An array is a type of data structure that stores the elements of similar data type and consider it as an object too. We can store only a fixed set of elements and can’t expand its size, once its size is declared.

The array follows Index based storage i.e)the first element of an array is stored at index 0 or at index ‘i’ and the remaining elements are stored at the location ‘i+1’.

**Features of an Array**

 Same data type of elements is stored in an array.

 Array elements are always stored in contiguous memory locations.

 Storage of 2-D array elements are rowed by row in a contiguous memory location.

 The Starting element of address is represented by the array name.

 The size of an array should be declared at the time of initialization.

 The remaining elements of an array can be retrieved by using the starting index of an Array.

Typescript supports array just like that in JavaScript. There are two ways to declare an array in typescript:

**1. Using square brackets.**

let array\_name[:datatype] = [val1, val2, valn..]

**Example:**

filter\_none

brightness\_4

|  |
| --- |
| let fruits: string[] = ['Apple', 'Orange', 'Banana']; |

**2. Using a generic array type.**  
TypeScript array can contain elements of different data types, as shown below.

let array\_name: Array = [val1, val2, valn..]

**Example: Multi Type Array**

filter\_none

brightness\_4

|  |
| --- |
| var values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];  // or  var values: Array = ['Apple', 2, 'Orange', 3, 4, 'Banana']; |

**Example: Access Array Elements**

 Array elements access on the basis of index i.e.)ArrayName[index].

filter\_none

brightness\_4

|  |
| --- |
| let fruits: string[] = ['Apple', 'Orange', 'Banana'];  fruits[0]; // returns Apple  fruits[1]; // returns Orange  fruits[2]; // returns Banana  fruits[3]; // returns undefined |

 We can access the array elements by using the ‘FOR’ loop:

filter\_none

brightness\_4

|  |
| --- |
| let fruits: string[] = ['Apple', 'Orange', 'Banana'];    for(var index in fruits)  {      console.log(fruits[index]);  // output: Apple Orange Banana  }    for(var i = 0; i < fruits.length; i++)  {      console.log(fruits[i]); // output: Apple Orange Banana  } |

**Advantages**  
Code Optimization: We can retrieve or sort the array data more efficiently.  
Random access: We can randomly access the array data using the location pointer.

**Disadvantages**  
Size Limit: The size of an Array is fixed i.e.)static. We cannot increase the array size once it has been declared.

**There are two types of an array:  
1.Single-Dimensional Array  
2.Multi-Dimensional Array**

**Single-Dimensional Array**  
It is the simplest form of an array that contains only one row for storing data. It contains single set of the square bracket (“[]”).  
**Syntax:**

let array\_name[:datatype];

**Initialization:**

array\_name = [val1, val2, valn..]

**Example:**

filter\_none

brightness\_4

|  |
| --- |
| let arr:number[];  arr = [1, 2, 3, 4]  console.log("Array[0]: " +arr[0]);  console.log("Array[1]: " +arr[1]); |

**Output:**

Array[0]: 1

Array[1]: 2

**Multi-Dimensional Array**  
The data is stored in rows and columns (also known as matrix form) in a Multi-dimensional array.

TypeScript Arrays  
**Syntax:**

let arr\_name:datatype[][] = [ [a1, a2, a3], [b1, b2, b3] ];

**Initialization:**

let arr\_name:datatype[initial\_array\_index][referenced\_array\_index] = [ [val1, val2, val 3], [v1, v2, v3]];

**Example:**

filter\_none

brightness\_4

|  |
| --- |
| var mArray:number[][] = [[10, 20, 30], [50, 60, 70]] ;  console.log(mArray[0][0]);  console.log(mArray[0][1]);  console.log(mArray[0][2]);  console.log();  console.log(mArray[1][0]);  console.log(mArray[1][1]);  console.log(mArray[1][2]); |

**OUTPUT:**

10

20

30

50

60

70

**Array Object**

We can create an Array by using or initializing the Array Object. The Array constructor is used to pass the following arguments to create an Array:

 With the numeric value which represents the size of an array.

 A list of comma separated values.

**Syntax:**

1.let arr\_name:datatype[] = new Array(values);

**Example:**

filter\_none

brightness\_4

|  |
| --- |
| // Initializing an Array by using the Array object.  let arr:string[] = new Array("GEEKSFORGEEKS", "2200", "Java", "Abhishek");  for(var i = 0;i<arr.length;i++) {     console.log(arr[i]); |

**Output:**

GEEKSFORGEEKS

2200

Java

Abhishek

**Passing an Array to a Function  
We can pass an Array to a function by specifying the Array name without an index.  
Example:**

filter\_none

brightness\_4

|  |
| --- |
| let arr:string[] = new Array("GEEKSFORGEEKS", "2300", "Java", "Abhishek");  // Passing an Array in a function  function display(arr\_values:string[]) {     for(let i = 0;i<arr\_values.length;i++) {        console.log(arr[i]);     }  }  // Calling an Array in a function  display(arr); |

**Output**

**GEEKSFORGEEKS**

**2300**

**Java**

**Abhishek**

**Using TypeScript ‘Spread’ operator  
The spread operator can be used to initialize arrays and objects from another array or object. It can also be used for object destructuring. It is a part of ECMAScript 6 version.**

filter\_none

brightness\_4

|  |
| --- |
| let arr1 = [ 1, 2, 3];  let arr2 = [ 4, 5, 6];  //Create new array from existing array  let copyArray = [...arr1];  console.log("CopiedArray: " +copyArray);  //Create new array from existing array with more elements  let newArray = [...arr1, 7, 8];  console.log("NewArray: " +newArray);  //Create array by merging two arrays  let mergedArray = [...arr1, ...arr2];  console.log("MergedArray: " +mergedArray); |

**Output:**

**CopiedArray: 1, 2, 3**

**NewArray: 1, 2, 3, 7, 8**

**MergedArray: 1, 2, 3, 4, 5, 6**

TypeScript Union

The TypeScript union has the ability to combine one or two different types of data (i.e., number, string, float, double, etc). It is the most powerful way to express a variable with multiple types. Use pipe (‘|’) symbol to combine two or more data types to achieve Union type.

**Syntax:**

(type1|type2|type3|...|type-n)

**Example:**

filter\_none

brightness\_4

|  |
| --- |
| let value: number | string;  value = 190;  console.log("Numeric value of the value: " + value);  value = "Welcome to TypeScript!";  console.log("String value of the value: " + value); |

Compiling the above code to generate the following JavaScript code.

filter\_none

brightness\_4

|  |
| --- |
| let value: number | string;  value = 190;  console.log("Numeric value of the value: "+value);  value = "Welcome to TypeScript!";  console.log("String value of the value: "+value); |

**Output:**

190

Welcome to TypeScript!

**Example:** In this example the geeks is of union type, denoted using (string | number). So, we can assign a string or a number to it nothing else is allowed.

filter\_none

brightness\_4

|  |
| --- |
| let geeks: (string | number);  geeks = 123;   // OK  geeks = "XYZ"; // OK  geeks = true;  // Compiler Error |

**Function Parameter as Union Type:** We can pass the function as a parameter. In this example, parameter geeks is of union type. You can pass either a string value or a number value otherwise the compiler will give an error.  
**Example:**

filter\_none

brightness\_4

|  |
| --- |
| function displayType(geeks: (string | number)) {      if(typeof(geeks) === "number")          console.log('geeks is number.')      else if(typeof(geeks) === "string")          console.log('geeks is string.')  }    // Output: Code is number.  displayType(49);    // Output: Code is string.  displayType("GFG");    // Compiler Error: Argument of type 'true' is not  // assignable to a parameter of type string | number  displayType(true); |

**Array as Union Type:** In union type we can also pass an array. The program declares an array. The array can represent either a numeric collection or a string collection.  
**Example:**

filter\_none

brightness\_4

|  |
| --- |
| //Generated by typescript 1.8.10  var arr = [2, 5, 7, 5, 11, 15];    console.log("Display the array elements");    // Loop to display array elements  for (var i = 0; i < arr.length; i++) {     console.log(arr[i]);  }    // Declare another array  arr = ["Geeks", "G4G", "GFG", "GeeksforGeeks"];    console.log("Display the array elements");    // Loop to display the array elements  for (var i = 0; i < arr.length; i++) {     console.log(arr[i]);  } |

**Output:**

Display the array elements

2

5

7

5

11

15

Display the array elements

Geeks

G4G

GFG

GeeksforGeeks

**Union can Replace enums:** It is a list of constant types that is created by **Enums**. By default, enums have index values (0, 1 ,2, 3, etc). Enums are actually transpiled (taking source code written in one language and transforming into another language that has a similar level of abstraction) and end up into the result like JavaScript.

TypeScript Tuples

As we know array consists value of Homogeneous types but sometimes we need to store a collection of a different type value in a single variable. Then we will go with **Tuples**. They are just like structure in C programming and can also be passed as parameters in a function call.

 To denote a multi-dimensional coordinate system the term used is tuple in abstract mathematics.

 In JavaScript we doesn’t have tuples as data types, but in typescript Tuples facility is available.

**Syntax**

let tuple\_name = [val1, val2, val3, ...val n];

**Example:**

let arrTuple = [501, "welcome", 505, "Mohan"];

console.log(arrTuple);

**Output:**

*[501, ‘welcome’, 105, ‘Mohan’]*

Declaration and initialization of a tuple separately by initially declaring the tuple as an empty tuple in Typescript.  
**Example:**

let arrTuple = [];

arrTuple[0] = 501

arrTuple[1] = 506

**Accessing tuple Elements**  
With the help of index basis we can read or access the fields of a tuples, which is the same as an array. An index starts from zero too.  
**Example:**

var employee: [number, string] = [1, "Steve"];

employee[0]; // returns 1

employee[1]; / return Steve

**Output:**

*1  
Steve*

We can declare heterogenous datatypes in tuples like: number and string simultaneously.  
**Example**

let empTuple = ["Vivek Singh", 22, "Honesty"];

console.log("Name of the Employee is : "+empTuple [0]);

console.log("Age of the Employee is : "+empTuple [1]);

console.log(empTuple [0]+" is workinging in "+empTuple [2]);

Output:

*Name of the Employee is : Vivek Singh  
Age of the Employee is : 22  
Vivek Singh is workinging in Microsoft*

**Operations on Tuple**  
A tuple has two operations:

 Push()

 Pop()

**Push()**  
To add an element to the tuple with push operation.  
**Example**

var employee: [number, string] = [1, "Steve"];

employee.push(2, "Bill");

console.log(employee);

**Output:**

*[1, ‘Steve’, 2, ‘Bill’]*

This type of declaration is allowed in tuples because we are adding number and string values to the tuple and they are valid for the **employee** tuple.  
**Example**

let empTuple = ["Vivek Singh", 22, "Honesty"];

console.log("Items: "+empTuple); // here we print tuple elements

empTuple.push(10001); // append value to the tuple

console.log("Length of Tuple Items after push: "+empTuple.length); // After pushing elements in tuples calculate length of tuples.

console.log("Items: "+empTuple);

**Output:**

*Items: Vivek Singh, 22, Honesty  
Length of Tuple Items after push: 4  
Items: Vivek Singh, 22, Honesty, 10001*

To add an element to the tuple with push operation.  
**Example**

let empTuple = ["Mohit Singh", 25, "geeksforgeeks", 10001];

console.log("Items: "+empTuple); // here we print tuple elements

empTuple.pop(); // removed value to the tuple

console.log("Length of Tuple Items after pop: "+empTuple.length); After pushing elements in tuples calculate length of tuples.

console.log("Items: "+empTuple);

**Output:**

*Items: Mohit Singh, 25, geeksforgeeks, 10001  
Length of Tuple Items after pop: 3  
Items: Mohit Singh, 25, geeksforgeeks*

**Update or Modify the Tuple Elements**  
We need to use the index of the fields and assignment operator for modifying the fields of tuple. It can be shown in the following example.  
**Example**

let empTuple = ["Ganesh Singh", 25, "TCS"];

empTuple[1] = 60;

console.log("Name of the Employee is: "+empTuple [0]);

console.log("Age of the Employee is: "+empTuple [1]);

console.log(empTuple [0]+" is workinging in "+empTuple [2]);

**Output:**

*Name of the Employee is: Ganesh Singh  
Age of the Employee is: 60  
Ganesh Singh is workinging in TCS*

**Clear the fields of a Tuple**  
Fields could be cleared but we cannot delete the tuple variables. To clear the fields of a tuple, assign it with an empty tuple field set as shown below:

let empTuple = ["Rohit Sharma", 25, "JavaTpoint"];

empTuple = [];

console.log(empTuple);

**Output:**

*[]*

In TypeScript, To break up the structure of an entity by destructuring.  
**Example**

let empTuple = ["Rohit Sharma", 25, "JavaTpoint"];

let [emp, student] = empTuple;

console.log(emp);

console.log(student);

*Rohit Sharma  
25*

**Passing Tuple to Functions**

//Tuple Declaration

let empTuple = ["JavaTpoint", 101, "Abhishek"];

//Passing tuples in function

function display(tuple\_values:any[]) {

for(let i = 0;i<empTuple.length;i++) {

console.log(empTuple[i]);

}

}

//Calling tuple in function

display(empTuple);