ES6 - Functions

Functions are the building blocks of readable, maintainable, and reusable code. Functions are defined using the function keyword. Following is the syntax for defining a standard function.

```
function function_name() {
   // function body
}
```

To force execution of the function, it must be called. This is called as function invocation. Following is the syntax to invoke a function.

```
function_name()
```

Example : Simple function definition

```
//define a  function
function test() {
   console.log("function called")
}
//call the function
test()
```

The example defines a function test(). A pair of delimiters ({ }) define the function body. It is also called as the **function scope**. A function must be invoked to force its execution.

The following output is displayed on successful execution of the above code.

```
function called
```

Classification of Functions

Functions may be classified as **Returning** and **Parameterized** functions.

Returning functions

Functions may also return the value along with control, back to the caller. Such functions are called as returning functions.

Following is the syntax for the returning function.

```
function function_name() {
   //statements
   return value;
}
```

- A returning function must end with a return statement.
- A function can return at the most one value. In other words, there can be only one return statement per function.
- The return statement should be the last statement in the function.

The following code snippet is an example of a returning function −

```
function retStr() {
   return "hello world!!!"
}
var val = retStr()
console.log(val)
```

The above Example defines a function that returns the string “hello world!!!” to the caller. The following output is displayed on successful execution of the above code.

```
hello world!!!
```

Parameterized functions

Parameters are a mechanism to pass values to functions. Parameters form a part of the function’s signature. The parameter values are passed to the function during its invocation. Unless explicitly specified, the number of values passed to a function must match the number of parameters defined.

Following is the syntax defining a parameterized function.

```
function func_name( param1,param2 ,…..paramN) {
   ......
   ......
}
```

Example − Parameterized Function

The Example defines a function add that accepts two parameters **n1** and **n2** and prints their sum. The parameter values are passed to the function when it is invoked.

```
function add( n1,n2) {
   var sum = n1 + n2
   console.log("The sum of the values entered "+sum)
}
add(12,13)
```

The following output is displayed on successful execution of the above code.

```
The sum of the values entered 25
```

Default function parameters

In ES6, a function allows the parameters to be initialized with default values, if no values are passed to it or it is undefined. The same is illustrated in the following code.

```
function add(a, b = 1) {
   return a+b;
}
console.log(add(4))
```

The above function, sets the value of b to 1 by default. The function will always consider the parameter b to bear the value 1 unless a value has been explicitly passed. The following output is displayed on successful execution of the above code.

```
5
```

The parameter's default value will be overwritten if the function passes a value explicitly.

```
function add(a, b = 1) {
   return a + b;
}
console.log(add(4,2))
```

The above code sets the value of the parameter b explicitly to 2, thereby overwriting its default value. The following output is displayed on successful execution of the above code.

```
6
```

Rest Parameters

Rest parameters are similar to variable arguments in Java. Rest parameters doesn't restrict the number of values that you can pass to a function. However, the values passed must all be of the same type. In other words, rest parameters act as placeholders for multiple arguments of the same type.

To declare a rest parameter, the parameter name is prefixed with three periods, known as the spread operator. The following example illustrates the same.

```
function fun1(...params) {
   console.log(params.length);
}
fun1();
fun1(5);
fun1(5, 6, 7);
```

The following output is displayed on successful execution of the above code.

```
0
1
3
```

Note − Rest parameters should be the last in a function's parameter list.

Anonymous Function

Functions that are not bound to an identifier (function name) are called as anonymous functions. These functions are dynamically declared at runtime. Anonymous functions can accept inputs and return outputs, just as standard functions do. An anonymous function is usually not accessible after its initial creation.

Variables can be assigned an anonymous function. Such an expression is called a **function expression**.

Following is the syntax for anonymous function.

```
var res = function( [arguments] ) { ... }
```

Example − Anonymous Function

```
var f = function(){ return "hello"}
console.log(f())
```

The following output is displayed on successful execution of the above code.

```
hello
```

Example − Anonymous Parameterized Function

```
var func = function(x,y){ return x*y };
function product() {
   var result;
   result = func(10,20);
   console.log("The product : "+result)
}
product()
```

The following output is displayed on successful execution of the above code.

```
The product : 200
```

The Function Constructor

The function statement is not the only way to define a new function; you can define your function dynamically using Function() constructor along with the new operator.

Following is the syntax to create a function using Function() constructor along with the new operator.

```
var variablename = new Function(Arg1, Arg2..., "Function Body");
```

The Function() constructor expects any number of string arguments. The last argument is the body of the function – it can contain arbitrary JavaScript statements, separated from each other by semicolons.

The Function() constructor is not passed any argument that specifies a name for the function it creates.

Example − Function Constructor

```
var func = new Function("x", "y", "return x*y;");
function product() {
   var result;
   result = func(10,20);
   console.log("The product : "+result)
}
product()
```

In the above example, the Function() constructor is used to define an anonymous function. The function accepts two parameters and returns their product.

The following output is displayed on successful execution of the above code.

```
The product : 200
```

Recursion and JavaScript Functions

Recursion is a technique for iterating over an operation by having a function call itself repeatedly until it arrives at a result. Recursion is best applied when you need to call the same function repeatedly with different parameters from within a loop.

Example − Recursion

```
function factorial(num) {
   if(num<=0) {
      return 1;
   } else {
      return (num * factorial(num-1)  )
   }
}
console.log(factorial(6))
```

In the above example the function calls itself. The following output is displayed on successful execution of the above code.

```
720
```

Example − Anonymous Recursive Function

```
(function() {
   var msg = "Hello World"
   console.log(msg)
})()
```

The function calls itself using a pair of parentheses (). The following output is displayed on successful execution of the above code.

```
Hello World
```

Lambda Functions

Lambda refers to anonymous functions in programming. Lambda functions are a concise mechanism to represent anonymous functions. These functions are also called as **Arrow functions**.

Lambda Function - Anatomy

There are 3 parts to a Lambda function −

- **Parameters** − A function may optionally have parameters.
- The **fat arrow notation/lambda notation** (=>): It is also called as the goes to operator.

- **Statements** − Represents the function's instruction set.

Tip − By convention, the use of a single letter parameter is encouraged for a compact and precise function declaration.

Lambda Expression

It is an anonymous function expression that points to a single line of code. Following is the syntax for the same.

```
([param1, parma2,…param n] )=>statement;
```

Example − Lambda Expression

```
var foo = (x)=>10+x
console.log(foo(10))
```

The Example declares a lambda expression function. The function returns the sum of 10 and the argument passed.

The following output is displayed on successful execution of the above code.

```
20
```

Lambda Statement

It is an anonymous function declaration that points to a block of code. This syntax is used when the function body spans multiple lines. Following is the syntax of the same.

```
( [param1, parma2,…param n] )=> {
   //code block
}
```

Example − Lambda Statement

```
var msg = ()=> {
   console.log("function invoked")
}
msg()
```

The function's reference is returned and stored in the variable msg. The following output is displayed on successful execution of the above code.

```
function  invoked
```

Syntactic Variations

Optional parentheses for a single parameter.

```
var msg = x=> {
   console.log(x)
}
msg(10)
```

Optional braces for a single statement. Empty parentheses for no parameter.

```
var disp = ()=>console.log("Hello World")
disp();
```

Function Expression and Function Declaration

Function expression and function declaration are not synonymous. Unlike a function expression, a function declaration is bound by the function name.

The fundamental difference between the two is that, function declarations are parsed before their execution. On the other hand, function expressions are parsed only when the script engine encounters it during an execution.

When the JavaScript parser sees a function in the main code flow, it assumes function declaration. When a function comes as a part of a statement, it is a function expression.

Function Hoisting

Like variables, functions can also be hoisted. Unlike variables, function declarations when hoisted, hoists the function definition rather than just hoisting the function's name.

The following code snippet, illustrates function hoisting in JavaScript.

```
hoist_function();
function hoist_function() {
   console.log("foo");
}
```

The following output is displayed on successful execution of the above code.

```
foo
```

However, function expressions cannot be hoisted. The following code snippet illustrates the same.

```
hoist_function(); // TypeError: hoist_function() is not a function
var hoist_function() = function() {
   console.log("bar");
};
```

Immediately Invoked Function Expression

Immediately Invoked Function Expressions (IIFEs) can be used to avoid variable hoisting from within blocks. It allows public access to methods while retaining privacy for variables defined within the function. This pattern is called as a self-executing anonymous function. The following two examples better explain this concept.

Example 1 : IIFE

```
var main = function() {
   var loop = function() {
      for(var x = 0;x<5;x++) {
         console.log(x);
      }
   }();
   console.log("x can not be accessed outside the block scope x value is :"+x);
}
main();
```

Example 2 : IIFE

```
var main = function() {
   (function() {
      for(var x = 0;x<5;x++) {
         console.log(x);
      }
   })();
   console.log("x can not be accessed outside the block scope x value is :"+x);
}
main();
```

Both the Examples will render the following output.

```
0
1
2
3
4
Uncaught ReferenceError: x is not define
```

Generator Functions

When a normal function is invoked, the control rests with the function called until it returns. With generators in ES6, the caller function can now control the execution of a called function. A generator is like a regular function except that −

- The function can yield control back to the caller at any point.
- When you call a generator, it doesn't run right away. Instead, you get back an iterator. The function runs as you call the iterator's next method.

Generators are denoted by suffixing the function keyword with an asterisk; otherwise, their syntax is identical to regular functions.

The following example illustrates the same.

```
"use strict"
function* rainbow() {
   // the asterisk marks this as a generator
```

```
   yield 'red';
   yield 'orange';
   yield 'yellow';
   yield 'green';
   yield 'blue';
   yield 'indigo';
   yield 'violet';
}
for(let color of rainbow()) {
   console.log(color);
}
```

Generators enable two-way communication between the caller and the called function. This is accomplished by using the **yield** keyword.

Consider the following example −

```
function* ask() {
   const name = yield "What is your name?";
   const sport = yield "What is your favorite sport?";
   return `${name}'s favorite sport is ${sport}`;
}
const it = ask();
console.log(it.next());
console.log(it.next('Ethan'));
console.log(it.next('Cricket'));
```

Sequence of the generator function is as follows −

- Generator started in paused stated; iterator is returned.
- The it.next() yields “What is your name”. The generator is paused. This is done by the yield keyword.
- The call it.next(“Ethan”) assigns the value Ethan to the variable name and yields “What is your favorite sport?” Again the generator is paused.
- The call it.next(“Cricket”) assigns the value Cricket to the variable sport and executes the subsequent return statement.

Hence, the output of the above code will be −

```
{
   value: 'What is your name?', done: false
}
{
   value: 'What is your favorite sport?', done: false
}
{
   value: 'Ethan\'s favorite sport is Cricket', done: true
}
```

Note − Generator functions cannot be represented using arrow functions.