“let” keyword was introduced in ES6. It lets you define block scope(bracket scope) variables in JavaScript. Initially JavaScript only supported function scope and global scope variables.

Here is an example code

if(true)  
{  
    let x = 12;  
    alert(x); *//alert's 12*  
}  
  
alert(x); *//x is undefined here*

“let” keyword limits the variables accessibility upto a block, statement or expression.

“const” keyword was introduced in ES6. It lets you define read only variables using JavaScript. Variables created using “const” are block scoped(or bracket scoped). Redeclaring a “const” variable in the same scope throws an error.

Here is an code example:

const x = 12;  
  
*//an constant 'x' is already available in this scope therefore the below line throws an error when you are try to create a new x variable.*  
const x = 13;  
  
if(true)  
{  
    *//an constant 'x' is available in this scope but not defined in this scope therefore the below line will not throw error instead define a new "x" inside this scope.*  
    const x = 13;  
      
    *//here 'y' is available inside this scope not outside this scope*  
    const y = 11;  
}  
  
*//here creating a new 'y' will not throw an error because no other 'y' is available in this scope(i.e., global scope)*  
const y = 12;

Just remember that in a scope you cannot redeclare or change value of an “const” variable if a variable with same name is already available for access in that scope.

There were many different ways purposed by JavaScript developers for returning multiple values in an function. But ECMAScript 6 has come up with an easy way to do it.

Here is an code example of how to return multiple values using ES6:

function function\_name()  
{  
    return [1, 6, 7, 4, 8, 0]; *//here we are storing variables in an array and returning the array*  
}  
  
var q, w, e, r, t, y;  
  
*//Here we are using ES6's array destructuring feature to assign the returned values to variables.*  
*//Here we are ignoring 2 and 4 array indexes*  
[q, w, , r, , y] = function\_name();  
  
alert(y);*//y is 0*

ES6 provides a array like syntax to assign multiple variables to values of array indexes. It also lets you ignore some array indexes.

Default Function Arguments Values in JavaScript

ES6 provides a new syntax that can be used to define default values to function parameters:

function myFunction(x = 1, y = 2, z = 3)  
{  
     console.log(x, y, z); *// Output "6 7 3"*  
}  
myFunction(6,7);

Also, passing undefined is considered as missing an argument. Here is an example  
to demonstrate this:

   function myFunction(x = 1, y = 2, z = 3)  
   {  
     console.log(x, y, z); *// Output "1 7 9"*  
   }  
   myFunction(undefined,7,9);

Defaults can also be expressions. Here is an example to demonstrate this:

   function myFunction(x = 1, y = 2, z = 3 + 5)  
   {  
     console.log(x, y, z); *// Output "6 7 8"*  
   }  
   myFunction(6,7);

# JavaScript “…” Operator

ES6 introduced “…” operator which is also called as spread operator. When “…” operator is applied on an array it expands the array into multiple variables in syntax wise. And when its applied to an function argument it makes the function argument behave like array of arguments.

We can use spread operator to take indefinite number of arguments.

Here is an example code of how to use this operator

*//args variable is an array holding the passed function arguments*  
function function\_one(...args)  
{     
    console.log(args);  
    console.log(args.length);  
}  
  
function\_one(1, 4);  
function\_one(1, 4, 7);  
function\_one(1, 4, 7, 0);  
  
  
function function\_two(a, b, ...args)  
{  
    console.log(args);  
    console.log(args.length);  
}  
  
*//"args" holds only 7 and 9*  
function\_two(1, 5, 7, 9);

If we apply “…” to an array it expands it into multiple variables syntax wise. Here is an example code

function function\_name(a, b)  
{  
    console.log(a+b);  
}  
  
var array = [1, 4];  
  
function\_name(...array); *//is equal to function\_name(1, 4)*

# JavaScript Arrow “=>” Function

ECMAScript 6 provides a new way to create functions which just contain one line of statement. This new type of function is called lambda or arrow function.

Here is how to create a arrow function

*//sum is the function name*  
*//x and y are function parameters*  
var sum = (x, y) => x + y;  
  
console.log(sum(2, 900)); *//902*

Here (x, y) => x + y returns a regular JavaScript function object. Here the function body of the returned function object’s body would be function(x, y){return x+ y;}

Arrow functions always return the value of the statement when executed. Here result of x+yis returned.

You can also write multiple statements in an arrow function but arrow functions are mostly used in replacement of single statement functions. Here is code example of multiple statements in an arrow function

var sum = (x, y) => {  
    x = x + 10;  
    y = y + 10;  
    return x + y;  
}  
  
console.log(sum(10, 10)); *//40*

As arrow function actually returns a regular JavaScript function object so they can be used wherever we use regular JavaScript function object. For example, they can be used as callback.

function sum(p, q)  
{  
    console.log(p() + q()); *//87*  
}  
  
sum(a => 20 + 10, b => 1 + 56); *//here we are passing two function objects*

One last and most important feature about arrow function is that the “this” pointer inside an asynchronously executed arrow function points to the scope inside which it was passed as callback. A regular function’s this pointer points to global scope when executed asynchronously.

window.age = 12;  
  
function Person(){  
  this.age = 34;  
  
  setTimeout(() => {  
    console.log(this.age); *//34*  
  }, 1000);  
  
  setTimeout(function(){  
    console.log(this.age); *//12*  
  }, 1000);    
}  
  
var p = new Person();

# JavaScript “yield” Keyword and “function\*()” Syntax

ECMAScript 6 specification introduced a new JavaScript feature called as JavaScript Generators. JavaScript’s yield keyword and function\*() syntax together make JS Generators.

In nutshell JavaScript generators provide a new way for functions to return a collection and also a new way of looping(or iterating) through the elements of the returned collection.

Earlier to JavaScript Generators you would do something like this

function collection\_name()  
{  
    return [1, 3, 5, 7];  
}  
  
var collection = collection\_name();  
  
for(var iii = 0; iii < collection.length; iii++)  
{  
    console.log(collection[iii]);  
}

Here is how you can do the same using Generators

function\* collection\_name()  
{  
    yield 1;  
    yield 3;  
    yield 5;  
    yield 7;  
}  
  
for(var iii  of collection\_name())  
{  
    console.log(iii);  
}

Internally JavaScript creates a object with [Symbol.iterator](http://qnimate.com/javascript-iterators-tutorial/) property from the yielded values which is what for of construct needs for iterating a collection.