Introduction to

JavaScript

the programming language of the Web

Functions Blocks

A function is a block of JavaScript code that is defined once but may be executed, or invoked, any number of times.

JS

Defining Functions

A function declaration statement has the following syntax:

```
//function definition statement
function [Name]([param [, param  [..., param ]]]) {
        statements
}
```

- **Name** The function name. Can be omitted, in which case the function becomes known as an anonymous function.
- **Param** The name of an argument to be passed to the function. A function can have up to 255 arguments.
- **Statements** The statements comprising the body of the function.

return

A return statement within a function specifies the value of invocations of that function.

```
function square(x) {                    A function that has a return statement
        return x * x;
 }
square(2);                              This invocation evaluates to 4
```

Defining Functions

```
function definition statement
function Square(x) { return x * x; }
```

OR

```
function declaration expression
Note that we assign it to a variable
var square = function(x) { return x * x; }
```

Variable Scope

```
var scope = "global";              Declare a global variable
function checkscope() {
    var scope = "local";           Declare a local variable with the same name
    return scope;                  Return the local value, not the global one
}
checkscope()                       => "local"
```

Variable Scope

Although you can get away with not using the var statement when you write code in the global scope, you must always use var to declare local variables.

```
scope = "global";          Declare a global variable, even without var.
function checkscope2() {
    scope = "local";       Oops! We just changed the global variable.
    myscope = "local";     This implicitly declares a new global variable.
    return [scope, myscope];  Return two values.
}
checkscope2()              => ["local", "local"]: has side effects!
scope                      => "local": global variable has changed.
myscope                    => "local": global namespace cluttered up.
```

Variable Hoisting

JavaScript code behaves as if all variable declarations in a function (but not any associated assignments) are “hoisted” to the top of the function. Javascript does not use **block scoping** , instead it uses **function scoping**

```
var scope = "global";
function f() {
    console.log(scope);  Prints "undefined", not "global"
    var scope = "local";  Variable initialized here, but defined everywhere
    console.log(scope);  Prints "local"
}
```

Nested Functions

```
function hypotenuse(m, n) {          // outer function
    function square(num) {           // inner function
        return num * num;
    }
    return Math.sqrt(square(m) + square(n));
}
alert(hypotenuse(3,4));              // => 5
```

- The outer function hypotenuse contains an inner function square.
- The square function is visible only within the hypotenuse function body, that is, square has function scope only.
- The easiest way to look at square is as a private helper function to the outer function.

The Arguments Object

JavaScript functions have a built-in object called the arguments object.
The argument object contains an array of the arguments used when the function was called.

```
x = sumAll(1, 123, 500, 115, 44, 88);

function sumAll() {
    var i, sum = 0;
    for (i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}
```

Function Blocks: Default Parameters

JS

Functions with Default Parameter Values

Functions in JavaScript are **unique** in that they allow any number of parameters to be passed regardless of the number of parameters declared in the function definition. This allows you to define functions that can handle different numbers of parameters, often by just filling in **default values** when parameters aren't provided.

```
function getProduct(price, type="HardWare")
{
    //do something
}
//uses default type parameter
getProduct(1000);
//overwrite Default type value
getProduct(1000,"software");
```

```
function makeRequest(url, timeout = 2000, callback = function() {}) {
    // the rest of the function };

// uses default timeout and callback
makeRequest("/foo");
// uses default callback
makeRequest("/foo", 500);
// doesn't use defaults
makeRequest("/foo", 500, function() {
//doSomething; });
function callBackRequest(){/*body*/}
makeRquest("/foo",500,callBackRequest);
```

How Default Parameter Values Affect the arguments Object

```
function makeRequest(url, timeout = 2000, callback = function() {})
                                                {
                                                    console.log(arguments.length); };


// uses default timeout and callback
makeRequest("/foo"); //→ output 1
// uses default callback
makeRequest("/foo", 500); //→ output 2
// doesn't use defaults
makeRequest("/foo", 500, function() {
//doSomething; });          //→ output 3
```

Default Parameter Expressions

The most interesting feature of default parameter values is that the default value need not be a primitive value.

```
let baseDiscount=0.5;
function getProduct(price, type="HardWare",Discount=baseDiscount)
{   //do somthing ; }
getProduct(1000); //Discount→ 0.5
function getBaseDiscount(){ return 0.2 }
function getProduct(price, type="HardWare",Discount=getBaseDiscount())
{  //do something  }
getProduct(1000); //Discount→ 0.2
```

Keep in mind that **getBaseDiscount()** is called only when **getProduct()** is called without a second parameter

You can use a previous parameter as the default for a later parameter.

```
function getProduct(price, type="HardWare",Discount=price*0.2)
{   //do something   }
getProduct(1000); // Discount = 200
```

you can pass **price** into a function to get the value for **Discount**

```
function getBaseDiscount(value){ return value*0.2 }
function getProduct(price, type="HardWare",Discount=getBaseDiscount(price))
{  //do something  }
getProduct(1000); //Discount = 200
```

The ability to reference parameters from default parameter assignments works only for **previous** arguments, so earlier arguments don't have access to later arguments.

```
function getProduct(price=Discount, type="HardWare",Discount=0.3)
{  //do something  }
getProduct(); // ERROR
```

Function Blocks: Arrow Functions

JS

Arrow Function Syntax

All variations begin with function arguments, followed by the arrow, followed by the body of the function. The arguments and the body can take different forms depending on usage.

Function have no input or return

```
let getPrice = () => console.log("testing");
getPrice(); //-->testing
```

This is equivalent to

```
let getPrice =function ()
{
    console.log("testing");
}
```

✓Function take one argument and return one value

```
let getBaseDiscount = (price) => price*0.2;
console.log(getBaseDiscount (1000)); //-->200
```

This is equivalent to

```
let getBaseDiscount=function (price) {
    return price * 0.2
}
```

✓Function with more than one output statement

```
let getPrice = (product, price) => {
            let result;
            if(price>50)
                result = product + " : " + (price * 2)
            else
                result = product + " : " + (price)
            return result;
        }
```

No arguments Binding

Arrow function does not contains arguments object as normal Functions

```
let myFunction=(x)=>console.log(arguments.length)
myFunction(3)  //error -> arguments is not defined
```