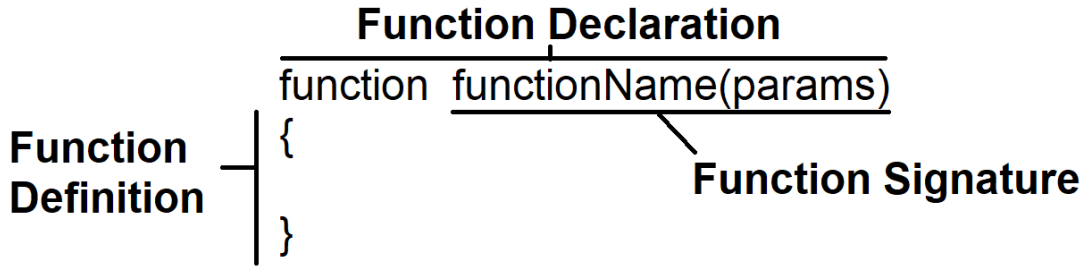
Functions and Methods in JavaScript:

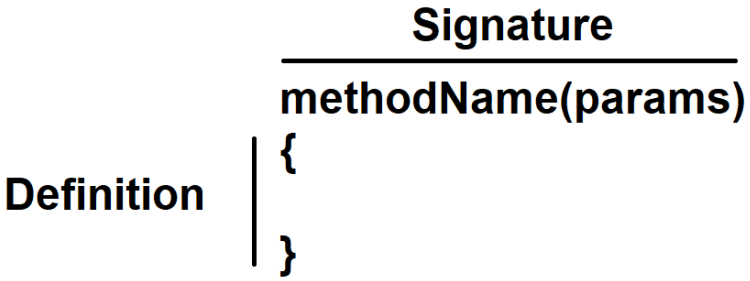
- Functions and methods define the functionality for any object or class.
- Functions are declared outside the class by using “function” keyword.



"Function is accessed by using its signature"

functionName(args)

- Methods are defined with in the class by using a method signature.



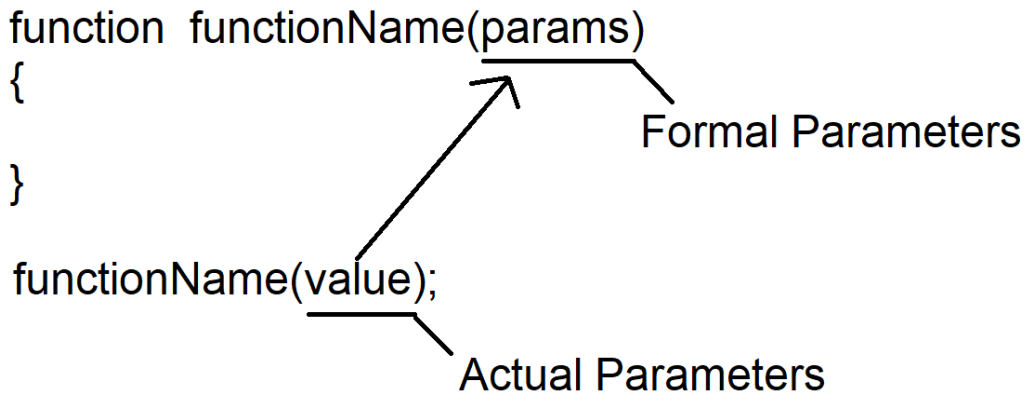
- You can access the functions and use in methods and vice versa.

```
<script>
   //function
  function Hello(){
     document.write("Hello is a function");
  }
```

```
class Demo
{
   //method
  Print(){
     document.write("Print is a Method<br>");
     Hello();
  }
}
function bodyload(){
   var obj = new Demo;
   obj.Print();
}
bodyload();
</script>
```

Parameters in Methods and Functions

- Functions and methods can be parameter less or parameterized.
- The parameters are used to modify the functionality.
- The parameters defined in function declaration are known as “Formal Parameters”.
- The parameters defined while calling the function as known as “Actual Parameters”.



Ex:

```
<script>
  function PrintNumbers(start, end){
     for(var i=start; i<=end; i++){
        document.write(i + "<br>");
     }
  }
  class Demo
  {
    Print(){
       PrintNumbers(100,120);
       PrintNumbers(200,210);
    }
  }
  var obj = new Demo;
```

```
  obj.Print();
</script>
```

Multiple Parameters:

- A function or method can be defined with multiple parameters.
- Every parameter is mandatory.
- If any formal parameter is not defined with actual value, then it returns “undefined”.

Ex:

```
<script>
  function PrintProduct(id, name, price){
    if(id==undefined) {
      document.write(`
      Name=${name} <br>
      Price=${price}
    `);
    } else {
    document.write(`
      ID=${id} <br>
      Name=${name} <br>
      Price=${price}
    `);
```

```
    }
  }
  class Product
  {
    Print(){
      PrintProduct(undefined,"TV",34000.44);
    }
  }
  var obj = new Product;
  obj.Print();
</script>
```

Function Expressions

Another option is to use a function expression. This just involves creating the function as a variable:

```
const logCompliment = function() {
console.log("You're doing great!");
};
logCompliment();
```

Note: Function declarations are hoisted and function expressions are not.

```
// Invoking the function before it's declared
hey();
// Function Declaration
function hey() {
alert("hey!");
}
```

```
// Invoking the function before it's declared
hey();
// Function Expression
const hey = function() {
alert("hey!");
};
```

TypeError: hey is not a function

Passing arguments:

```
const logCompliment = function(firstName) {
console.log(`You're doing great, ${firstName}`);
};
```

```
logCompliment("Molly");
```

Now, we won’t hard-code the message. We’ll pass in a dynamic value as a parameter:

```
const logCompliment = function(firstName, message) {
console.log(`${firstName}: ${message}`);
};
logCompliment("Molly", "You're so cool");
```

Function returns

A return statement specifies the value returned by the function.

```
const createCompliment = function(firstName, message) {
return `${firstName}: ${message}`;
};
createCompliment("Molly", "You're so cool");
```

Default Parameters

Default parameters are included in the ES6 spec, so in the event that a value is not provided for the argument, the default value will be used.

```
function logActivity(name = "Shane McConkey", activity = "skiing") {
console.log(`${name} loves ${activity}`);
}
```

If no arguments are provided to the logActivity function, it will run correctly using the default values.

Anonymous Function

- Function without name is known a Anonymous functions.
- It is mostly used in call back technique.
- Anonymous functions are accessed by using “()”
- Function must be enclosed in “()”.

EX:

```
<script>
  (function(){
     document.write("Welcome to JavaScript");
  })();
</script>
```

- You can store anonymous function by using a reference name. and access by using the reference name.

  Ex:

```
<script>
  var hello =  function(){
      document.write("Hello !");
   }
   hello();
```

</script>

- **ES5** introduced **“Rest Parameters”** into JavaScript.
- A Rest parameter allows **multiple values under one reference name**.
- Every function of method can have only one rest parameter.
- **Rest parameter must be the last parameter in formal list.**
- Rest parameter is **defined “…”**

Ex:

```
<script>
   function PrintList(...list){
      for(var item of list){
         document.write(item + "<br>");
      }
   }
   PrintList("TV","Mobile","Shoe");
</script>
<script>
   function PrintList(i,...list){
      for(var item of list){
         document.write(item + "<br>");
      }
      document.write("count = " + i);
```

```
    }
    PrintList(1,"TV","Mobile","Shoe");
</script>
```

Function Closure

- A closure defined function returning a function.
- A function can be defined with in the parent.
- It creates a closure, which will not allow to access the inner function values to parent function.

Ex:

```
<script>
    function Print(){
        function Add(a,b){
            return a + b;
        }
        return Add(10,20);
    }
    document.write(Print());
</script>
```

Arrow Functions

Arrow functions are a useful new feature of ES6. With arrow functions, you can create functions without using the function keyword. You also often do not have to use

the return keyword.

```
const lordify = function(firstName) {
return `${firstName} of Canterbury`;
};
console.log(lordify("Dale")); // Dale of Canterbury
console.log(lordify("Gail")); // Gail of Canterbury
```

With an arrow function, we can simplify the syntax tremendously:

```
const lordify = firstName => `${firstName} of Canterbury`;
console.log(lordify("Gail")); // Gail of Canterbury
```

With the arrow, we now have an entire function declaration in one line.

More than one argument should be surrounded by parentheses:

```
// Arrow Function
const lordify = (firstName, land) => `${firstName} of ${land}`;
console.log(lordify("Don", "Piscataway")); // Don of Piscataway
```

If there are multiple lines, you'll use curly braces:

```
const lordify = (firstName, land) => {if (!firstName) {
throw new Error("A firstName is required to lordify");
}
if (!land) {
throw new Error("A lord must have a land");
}
return `${firstName} of ${land}`;
};
console.log(lordify("Kelly", "Sonoma")); // Kelly of Sonoma
console.log(lordify("Dave")); // ! JAVASCRIPT ERROR
```

Returning objects

```
const person = (firstName, lastName) => ({first: firstName,
last: lastName});
console.log(person("Flad", "Hanson"));
```

Compiling JavaScript

When a new JavaScript feature is proposed and gains support, the community often wants to use it before it's supported by all browsers. The only way to be sure that your **code will work is to convert it to more widely compatible code before running it in the browser**. **This process is called compiling**. **One of the most popular tools for Java-Script compilation is Babel.**

In the past, the only way to use the latest JavaScript features was to wait weeks, months, or even years until browsers supported them. Now, Babel has made it possible to use the latest features of JavaScript right away. The compiling step makes Java-Script similar to other languages. It's not quite traditional compiling: **our code isn't compiled to binary**. **Instead, it's transformed into syntax that can be interpreted by a wider range of browsers.** Also, JavaScript now has source code, meaning that there will be some files that belong to your project that don't run in the browser.

As an example, let's look at an arrow function with some default arguments:

```
const add = (x = 5, y = 10) => console.log(x + y);
```

If we run Babel on this code, it will generate the following:

```
"use strict";
```



```
var add = function add() {
```

```
var x = arguments.length <= 0 || arguments[0] === undefined
? 5 : arguments[0];

var y =arguments.length <= 1 || arguments[1] === undefined
? 10 : arguments[1];

return console.log(x + y);

};
```

Babel added a “use strict” declaration to run in strict mode.

Transpilation

Transpilation is defined as **source-to-source compilation**. Tools have been written to do this and they are called **transpilers. Transpilers take the source code and convert it into another language.** Transpilers are important for two reasons. First, not every browser supports every new syntax in ES6, and second, many developers use programming languages based off of JavaScript, such as CoffeeScript or TypeScript.

A transpiler allows us to write our code in ES6 and translate it into vanilla ES5, which works in every browser. It is critical to ensure that our code works on as many web platforms as possible. Transpilers can be an invaluable tool for ensuring compatibility.

Transpilers also allow us to develop web or server side applications in other programming languages. Languages such as TypeScript and CoffeeScript may not run natively in the browser; however, with a transpiler, we are able to build a full

application in these languages and translate them into JavaScript for server or browser execution.

One of the most popular transpilers for JavaScript is Babel. Babel is a tool that was created to aid in the transpilation between different versions of JavaScript.