## Function call context JS

In JavaScript, the context of a function call refers to the value of the *this* keyword within the function. The context of a function call can be affected by several factors, including the way the function is called, the type of the object that the function is a method of, and the use of the call, apply, and bind methods.

1. When a function is called as a standalone function, the *this* keyword refers to the global object (window in the browser).

```
function myFunction () {
    console.log(this);
}
myFunction (); // logs the global object (window in the browser)
```

2. When a function is called as a method of an object, the *this* keyword refers to the object that the method is a property of.

```
var myObj = {
    property1: "value1",
    method1: function() {
        console.log(this);
    }
};
myObj.method1(); // logs the myObject
```

3. When a function is called using a global reference, the windows object is called.

```
var myObj = {
    property1: "value1",
    method1: function() {
        console.log(this);
    }
};

global_ref = myObj.method1;
global_ref(); // logs the global object (window in the browser)
```

## Functions - First class Citizens

In JavaScript, functions are first-class citizens, meaning that they are treated like any other type of data (such as numbers, strings, and objects). This means that they can be:

1. Assigned to variables
2. Passed as arguments to functions
3. Returned from functions
4. Stored in data structures

Here are a few examples of how functions can be used as first-class citizens in JavaScript:

1. Assigning a function to a variable:

```
var myFunction = function() {
    console.log("I am a function");
};
myFunction(); // logs "I am a function"
```

2. Passing a function as an argument to another function:

```
var myFunction = function() {
    console.log("I am a function");
};
var callFunction = function(func) {
    func();
};
callFunction(myFunction); // logs "I am a function"
```

3. Returning a function from another function:

```
var createFunction = function() {
    return function() {
        console.log ("I am a function");
    };
};
var myFunction = createFunction ();
myFunction (); // logs "I am a function"
```

4. Storing a function in an array:

```
var myFunction = function() {
    console.log("I am a function");
};
var myArray = [myFunction, "string", 42];
console.log(myArray[0]()); // logs "I am a function"
```

## Functions as Objects

In JavaScript, functions are objects. This means that they can have properties and methods just like any other object.

Here are a few examples of how functions can be used as objects in JavaScript:

1. Adding a property to a function:

```
function myFunction () {
    console.log ("I am a function");
}
myFunction.property1 = "value1";
console.log (myFunction.property1); // logs "value1"
```

2. Adding a method to a function:

```
function myFunction() {
    console.log("I am a function");
}
myFunction.method1 = function() {
    console.log("I am a method");
};
myFunction.method1(); // logs "I am a method"
```

3. Using the call method to invoke a function with a specific context:

```
function sum(x, y){
  console.log ("Sum: ", this.num1 + this.num2 + x + y);
}
obj = {
  num1: 10,
  num2: 20
};
sum.call(obj, 5, 10); // logs “Sum: 45”
```

4. Using the apply method when we want to pass an arguments array to the function:

```
// Apply method
function sum(x, y){
console.log ("Sum: ", this.num1 + this.num2 + x + y);
}
obj = {
  num1: 10,
```

```
  num2: 20
};
sum.apply(obj, [5, 10] ); // logs “Sum: 45”
```

5. Using the bind method to create a new function with a specific context:

```
function myFunction() {
    console.log(this);
}
var myObject = { property1: "value1" };
var boundFunction = myFunction.bind(myObject);
boundFunction(); // logs { property1: "value1" }
```

Functions as objects in JavaScript allows you to add properties and methods to them, and to manipulate them in the same way as any other object. This added functionality makes functions more versatile and powerful, and allows you to create more expressive and maintainable code.