1) Function Objects
2) Function Literal
3) Invocation
4) Arguments
5) Return
6) Exceptions
7) Augmenting Types
8) Recursion
9) Scope
10) 'this' in JavaScript
11) Predefined functions

JavaScript Function Syntax

A function is written as a code block (inside curly { } braces), preceded by the function keyword:

function *functionname*()
{
some code to be executed
}

1. Function Objects

Functions in JavaScript are objects.

Objects are collections of name/value pairs having a hidden link to a prototype object.
Objects produced from object literals are linked to Object.prototype.

Function objects are linked to Function.prototype (which is itself linked to Object.prototype). Every function is also created with two additional hidden properties: the function's context and the code that implements the function's behavior.

Since functions are objects, they can be used like any other value. Functions can be stored in variables, objects, and arrays. Functions can be passed as arguments to functions, and functions can be returned from functions.

Also, since functions are objects, functions can have methods.

2. Function Literal

Function objects are created with function literals:

A function literal has four parts.

1) The first part is the reserved word function.
2) The optional second part is the function's name.

3) The third part is the set of parameters of the function, wrapped in **parentheses**.
4) The fourth part is a set of statements wrapped in **curly braces**.

```
// Create a variable called add and store a function
// in it that adds two numbers.

var add = function(a, b) {
   return a + b;
};
```

3. Invocation

There are four patterns of invocation in JavaScript:

1. The method invocation pattern
2. The function invocation pattern
3. The constructor invocation pattern
4. The apply invocation pattern.

1. The method invocation pattern

Create myObject. It has a value and an increment method. The increment method takes an optional parameter. If the argument is not a number, then 1 is used as the default.

```
var myObject = {
   value: 0,
   increment: function(inc) {
      this.value += typeof inc === 'number' ? inc : 1;
   }
};
myObject.increment( );
document.writeln(myObject.value); // 1
myObject.increment(2);
document.writeln(myObject.value); // 3
```

2. The function invocation pattern

```
function add(x, y) {
   return x + y;
}

var myObject = {
   value: 0,
   increment: function(inc) {
      this.value += typeof inc === 'number' ? inc : 1;
   }
};

myObject.increment(2);
```

```
document.writeln(myObject.value); // 2

myObject.double = function() {
   var that = this;    // Workaround.

   var helper = function() {
      that.value = add(that.value, that.value);
   };

   helper();    // Invoke helper as a function.
};

myObject.double(  );
document.writeln(myObject.value);    // 4
```

3. The constructor invocation pattern

JavaScript is a prototypal inheritance language. That means that objects can inherit properties directly from other objects. The language is class-free.

Create a constructor function called Quo. It makes an object with a status property.

```
var Quo = function(string) {
   this.status = string;
};

// Give all instances of Quo a public method
// called get_status.
Quo.prototype.get_status = function( ) {
   return this.status;
};

// Make an instance of Quo.
var myQuo = new Quo("confused");
document.writeln(myQuo.get_status( )); // confused
```

4. The Apply Invocation Pattern

Because JavaScript is a functional object-oriented language, functions can have methods.

The apply method lets us construct an array of arguments to use to invoke a function.

It also lets us choose the value of this. The apply method takes two parameters.

The first is the value that should be bound to this. The second is an array of parameters.

// Make an array of 2 numbers and add them.

```
var add = function(a, b) {
   return a + b;
};

var array = [3, 4];
var sum = add.apply(null, array);
document.writeln(sum);
```

Javascript Call Vs Apply Method

```
function theFunction(name, profession) {
   alert("My name is " + name + " and I am a " + profession + ".");
}
theFunction("John", "fireman");
theFunction.apply(undefined, ["Susan", "school teacher"]);
theFunction.call(undefined, "Claude", "mathematician");
```

4. Arguments

Make a function that adds a lot of stuff. Note that defining the variable sum inside of the function does not interfere with the sum defined outside of the function. The function only sees the inner one.

```
var sum = function( ) {
   var i, sum = 0;
   for (i = 0; i < arguments.length; i += 1) {
      sum += arguments[i];
   }
   return sum;
};
document.writeln(sum(4, 8, 15, 16, 23, 42)); // 108
```

5. Return

The return statement can be used to cause the function to return early. When return is executed, the function returns immediately without executing the remaining statements. A function always returns a value. If the return value is not specified, then undefined is returned.

6. Exceptions

JavaScript provides an exception handling mechanism. Exceptions are unusual (but not completely unexpected) mishaps that interfere with the normal flow of a program.

When such a mishap is detected, your program should throw an exception:

```
var add = function(a, b) {
```

```
    if (typeof a !== 'number' || typeof b !== 'number') {
        throw {
            name: 'TypeError',
            message: 'add needs numbers'
        };
    }
    return a + b;
};
```

The throw statement interrupts execution of the function. It should be given an exception object containing a name property that identifies the type of the exception, and a descriptive message property. You can also add other properties.

The exception object will be delivered to the catch clause of a try statement:

```
// Make a try_it function that calls the new add
// function incorrectly.
var try_it = function( ) {
    try {
        add("seven");
    } catch (e) {
        document.writeln(e.name + ': ' + e.message);
    }
};
try_it( );
```

7. Augmenting Types

JavaScript allows the basic types of the language to be augmented. we seen that adding a method to Object.prototype makes that method available to all objects. This also works for functions, arrays, strings, numbers, regular expressions, and booleans.

For example, by augmenting Function.prototype, we can make a method available to all functions:

```
Function.prototype.method = function(name, func) {
    this.prototype[name] = func;
    return this;
};

Number.method('integer', function( ) {
    return Math[this < 0 ? 'ceil' : 'floor'](this);
});

document.writeln((-10 / 3).integer( )); // -3

String.method('trim', function( ) {
    return this.replace(/^\s+|\s+$/g, '');
```

```
});

document.writeln('"' + " neat ".trim( ) + '"');

Function.prototype.method = function(name, func) {
   if (!this.prototype[name]) {
      this.prototype[name] = func;
   }
};
```

8. Recursion

A recursive function is a function that calls itself, either directly or indirectly. Recursion is a powerful programming technique in which a problem is divided into a set of similar subproblems, each solved with a trivial solution. Generally, a recursive function calls itself to solve its subproblems.

The Towers of Hanoi is a famous puzzle. The equipment includes three posts and a set of discs of various diameters with holes in their centers. The setup stacks all of the discs on the source post with smaller discs on top of larger discs. The goal is to move the stack to the destination post by moving one disc at a time to another post, never placing a larger disc on a smaller disc. This puzzle has a trivial recursive solution:

```
var hanoi = function (disc, src, aux, dst) {
if (disc > 0) {
hanoi(disc - 1, src, dst, aux);
document.writeln('Move disc ' + disc +
' from ' + src + ' to ' + dst);
hanoi(disc - 1, aux, src, dst);
}
};
hanoi(3, 'Src', 'Aux', 'Dst');
```

It produces this solution for three discs:

```
Move disc 1 from Src to Dst
Move disc 2 from Src to Aux
Move disc 1 from Dst to Aux
Move disc 3 from Src to Dst
Move disc 1 from Aux to Src
Move disc 2 from Aux to Dst
Move disc 1 from Src to Dst
```

9. Scope

Scope in a programming language controls the visibility and lifetimes of variables and parameters. This is an important service to the programmer because it reduces naming collisions and provides automatic memory management:

```
var foo = function( ) {
   var a = 3, b = 5;
   var bar = function( ) {
      var b = 7, c = 11;
// At this point, a is 3, b is 7, and c is 11
      a += b + c;
// At this point, a is 21, b is 7, and c is 11
   };
// At this point, a is 3, b is 5, and c is not defined
   bar( );
// At this point, a is 21, b is 5
};
```

10 'this' in JavaScript

JavaScript is a great programming language. That would have been a controversial statement a few years ago, but developers have rediscovered its beauty and elegance. If you dislike JavaScript, it's probably because:

- You've encountered browser API differences or problems — which isn't really JavaScript's fault.
- You're comparing it to a class-based language such as C++, C# or Java — and JavaScript doesn't behave in the way you expect.

One of the most confusing concepts is the 'this' keyword. In most languages, 'this' is a reference to the current object instantiated by the class. In JavaScript, 'this' normally refers to the object which 'owns' the method, but it depends on how a function is called.

Global Scope

If there's no current object, 'this' refers to the global object. In a web browser, that's 'window' — the top-level object which represents the document, location, history and a few other useful properties and methods.

```
window.WhoAmI = "I'm the window object";
alert(window.WhoAmI);
alert(this.WhoAmI); // I'm the window object
alert(window === this); // true
```

Calling a Function

'this' remains the global object if you're calling a function:

```
window.WhoAmI = "I'm the window object";
```

```
function TestThis() {
    alert(this.WhoAmI); // I'm the window object
    alert(window === this); // true
}
TestThis();
```

Calling Object Methods

When calling an object constructor or any of its methods, 'this' refers to the instance of the object — much like any class-based language:

```
window.WhoAmI = "I'm the window object";
function Test() {
    this.WhoAmI = "I'm the Test object";
    this.Check1 = function() {
        alert(this.WhoAmI); // I'm the Test object
    };
}
Test.prototype.Check2 = function() {
    alert(this.WhoAmI); // I'm the Test object
};

var t = new Test();
t.Check1();
t.Check2();
```

11 Predefined functions

Function.prototype.call
Function.prototype.apply
Function.prototype.toString
Function.prototype.bind
Function.prototype.constructor

12 Important points to remember

http://stackoverflow.com/questions/7755088/what-does-href-expression-a-href-javascript-a-do

```
<script>
function doSomething() {
  alert("hello")
}
</script>
<a href="javascript:doSomething();">click me</a>
```

http://stackoverflow.com/questions/5195933/with-form-validation-why-onsubmit-return-functionname-instead-of-onsubmit