SoftServe

Empowering your Business
through Software Development

Module 3: Program flow and Data collections

D. Petin

07/2014

Agenda

- Program flow control [1]
  - conditions
  - loops
  - switch statement
- Collections [2]
  - array
  - hash table

Program Flow Control

Program flow

Operators in a program processed in linear order: from top to bottom and from left to right.

[1]

Such sequence is called **Program flow**.

There are several methods intended to change standard flow. You already know about *function*. Also JavaScript has *conditions*, *loops* and *switch statement*.

[2]

Conditions: if–else

Most of algorithms have situation when next step related of some conditions depended on previous steps. It's a reason to use **if-else** statement. [1]

```
if (condition) {
    true branch;
} else {
    false branch;
}
```
[2]

```
if (condition) {
    true branch;
}
```
[3]

Conditions: if-else

Example:

Function get a parameter with a information about discount. And if discount is "silver" or "gold", function modifies global variable price.

In this example a shortened form of operator was used.

```
function discount (type) {
    if (type === “silver”) {
        price *= 0.9;
    }
    if (type === “gold”) {
        price *= 0.85;
    }
    return price;
}
```

Conditions: ?:

Sometimes *if-else* too bulky. If we need to initialize a variable modifying it by simple conditions; or we need to return a value from function and this value is dependent on something, we can use ternary

Ternary operator like **?:**.

```
result = (condition)? true action: false action;
```

[1]

Let's rewrite the last example using ternary operator.

Conditions: ?:

```
function discount (type) {
    if (type === “silver”) {
        price *= 0.9;
    }
    if (type === “gold”) {
        price *= 0.85;
    }
    return price;
}
```

We get a more compact but a less readable code. So be careful!

```
function discount (type) {
    price *= (type === “silver”)? 0.9: 1;
    price *= (type === “gold”)? 0.85: 1;
    return price;
}
```

Loops: for

Loops are used when algorithm requires repeating of statements.

[1]

First of them: **for** – loop with counter

```
for (start position; repeat condition; step) {
    body of loop;    // will be repeated
}
```

[2]

One processing of loop's body is called **iteration**. [3]

Loops: while and do-while

Two others types of loops: **while** and **do-while**

```
while (condition) {
    body of loop;
}
```

[1]

```
do {
    body of loop;
} while (condition);
```

The main difference between these loops is the moment of condition calculation. *While* calculates condition, and if the result is true, *while* does iteration. *Do-while* initially does iteration and after that calculates a condition. [2]

Loops: examples

Example 1:

Text with number of current iteration will be print 5 times

```
for (var i = 0; i < 5; i++) {
    console.log("Iteration # %d", i + 1);
}
```
[1]

Example 2:

This loop will be repeated until accumulation reaches 100 or gets grater value.

```
while (accumulation < 100) {
    accumulation += doSomething();
}
```
[2]

Which type of loop to use?

It may be not so simple to decide which type of loops to use in some specific conditions.

There is a simple rule for loop selection: if we know exact counts of iterations, we use *for*, if we know only exiting condition we use *while* or *do-while*.

While loop we use in most cases when we need to check for condition for each iteration, but sometimes we know that at least one iteration should be executed, so we us *do-while* in such case.

Loops: break and continue

There are two keywords for loops control :

- **break** – aborts loop and moves control to next statement after the loop;
- **continue** – aborts current iteration and immediately starts next iteration.

Try not to use this keywords. A good loop have one entering point, one condition and one exit.

Switch

Switch statement allows to select one of many blocks of code to be executed. If all options don't fit, default statements will be processed

```
switch (statement) {
    case value1: some body;
                 break;
    case value2: some body;
                 break;
    . . .
    default: some body;
}
```

Switch

Example:

This switch looks for the word equivalent for a mark in the 5-point system

Default statement is not used.

```
switch (mark) {
    case 5: result = “excellent”;
            break;
    case 4: result = “good”;
            break;
    case 3: result = “satisfactorily”;
            break;
    case 2: result = “bad”;
            break;
}
```

Collections

Collections

Collection is a set of variables grouped under common name.

Usually elements of collections are grouped according to some logical or physical characteristic.

Collections help to avoids situations when we have to declare multiple variables with similar names::

var a1, a2, a3, a4...

There are two types of collections that are typical for JS: arrays and hash tables.

Arrays

Array: creation

There are two ways to create an array:

```
var name = [ ];  //  preferrable
```

and

```
var name = new Array()
```

[1]

First way (with using indexer **[ ]**) is modern and strongly recommended for use.

Second way (with **new** and **Array** constructor) is deprecated and not recommended. [2]

Array: processing

Usage of arrays:

```
var array = [] // declaration of empty array
var array = [5, 8, 16]  // declaration of predefined array

array[0] = 4; // writing value with index 0
tmp = array[2];  // reading value by index (in tmp - 16)

array.length // getting length of array
```

Array: processing

In the sample below we output all elements of the array to the console:

```
var array = [4, 8, 16, 32], i;
for (i = 0; i < array.length; i++) {
    console.log(array[i]);
}
```

[1]

Note: this works only for numerical indexes.

Array: features

Arrays in JavaScript differ from arrays in classical languages.

Arrays in JS are instances of Object.

So Array in JS can be easily resized, can contain data of different types and have string as an index.

And more: if we create empty array it is real empty. And if we insert element with index 5 into this empty array, we get array with only one element but with length equal to 6!

Array: length calculation

Let's discuss *length* calculation.

It’s a virtual property.

Arrays don't review own elements. It takes biggest index, increments it and returns it.

So, if we insert element into an empty array with index 6, property *length* will take false value – 7!

Avoid such errors to help special methods to insert and delete elements in the array. They correctly handle indexes and do not allow free spaces between them.

Array: useful methods

Some useful methods of array:

array.push(value) – add element to the end of an array

array.pop() – extract element from end of an array

array.unshift(value) – insert element before first

array.shift() – extract first element

array.join() – concatenate all elements into a string

array.split() – split a string into an array of substrings

array.sort() – built-in method to sort array

Array: forEach

Not so long ago array received very comfortable method **forEach**. This method circulates around array elements and executes your callback function for each of them.

```
var array = [4, 8, 16, 32];
array.forEach(function (element, i) {

    console.log(element);
});
```

Hash Table

Hash Table: creation

Sometimes we need an Array with string indexes (keys).
There is a special data structure for such case: hash table.

```
var name = { };
```

Hash table is an usual JavaScript object without methods.

To access elements use array syntax with string index instead of numerical index.

Hash Table: creation

We can create hash and initialize it at the same time. For this we should write values separated by a comma like in array. But for all values we have to set key:

```
var name = {
    key: value,
    key: value
};
```

[1]

This format of describing of JS object has its own name: JavaScript Object Notation or short **JSON**. [2]

Hash Table: usage

Usage of hash tables is very similar to arrays:

hash[“good”] = 4; // writing value in element with key “good”

tmp = **hash**[“excellent”]; // reading value by key “excellent”

The difference is in usage of **for-in** statement:

```
for (key in hash) {
    console.log(hash[key]);
}
```

[1]

Array vs Hash

Use **Array** for collections with digital indexes.
Use **Hash** if you want use string keys.

Don't look for property *length* in **Hash**.
Don't look for *forEach* and other **Array** methods in **Hash**.

Always explicitly declare **Array** otherwise you get a **Hash**.
Don't use *for* with hash, use *for-in* instead.

At finally : use collection – be cool :)


THANK
YOU