JavaScript Cheat Sheet

> Last updated: 07/18/2015

Primitives

Primitives are the basic building blocks of JavaScript.

- `null` - intentionally valueless
- `undefined` - value has yet to be defined
- `string`
- `boolean` – true `and` false`
- `number` - integer or float

Variables

- Variables function as containers for values.
- Allow you to pass values around and refer to them with a set name.
- The name should be meaningful and it must be unique that follow the JavaScript naming conventions.
- Declaration syntax: `var varName = varValue`
- Variables can have their values reassigned later with another assignment statement: `varName = newValue`

Math Operators

- addition (+)
- subtraction (–)
- multiplication (*)
- division (/)
- modulus (%)
- The mathematical operators can be combined with the assignment operator (=). - ex: +=

Logical Operators

- and (`&&`)
- or (`||`)

Comparison Operators

- greater-than (`>`)
- less-than (`<`)
- greater-than or equal (`>=`)
- less-than or equal (`<=`)
- equal (`==`)
- not equal (`!=`)
- strict equal (`===`)
- strict not equal (`!==`)
- More on operators

Conditionals

- Conditional statements control the flow of a program.
- Conditionals are done with `if...else if...else` blocks

```
if (condition) {
  //some code
} else if (condition) {
  //some code
} else {
  //some more code
}
```

- The `else` block is usually your catch-all or default behavior block.
- `switch` statements can also be used to control program flow based on the value of a sentinel variable.

```
switch (expression) {
  case someVal:
```

```
    //do things
    break;
  case 1:
    //do things
    break;
  case "blue":
    //do things
    break;
  default:
    //default behavior
}
```

- The `default` block gets executed if `expression` doesn't match any of the cases.

Loops

- Allow you to keep running a certain piece of code for a certain number of iterations.

`while` loops:

- Will execute and continue to execute as long as the Boolean expression given to the loop evaluates to `true`.

```
while (someBooleanExpression) {
 //do stuff
}
```

`for` loops:

- `for` loops are useful for when you know how many times you want to iterate because you are explicitly setting the number of iterations with either a primitive value or a variable.

```
for (var i = 0; i < stop; i++) {
  //do stuff
}
```

  i. In the above, `var i = 0` is the expression that defines where you want to start the `for` loop.

ii. `i < stop` defines where you want to stop the loop.
iii. `i++` defines how you want to change `i` after each iteration.
iv. The general pattern is: `for (start; stop; change)`.
v. The `start` expression is executed before the first iteration of the loop.

Functions

- Allow us to capture and reuse blocks of code.
- Should have a single defined purpose.
- Can be defined using either *expression syntax* or *declaration syntax*.

- Expression syntax:

```
var myFunction = function(){
  //do stuff
}
```

- Declaration syntax:

```
function myFunction() {
  //do stuff
}
```

- JavaScript functions can take zero arguments, or as many arguments as you want.

- Functions can take and deal with optional arguments as well. If you do not give a parameter that the function is expecting, JavaScript will set that parameter to `undefined`.
- Will return `undefined` unless you use a `return` statement in the function body.
- When a `return` statement is executed, control breaks out of the function.
- **Parameters** are the variable names you use when defining a function - ex: `function myFunction(thing1, thing2)`.
- **Arguments** are the values that you supply to a function when you call it - ex: `myFunction(32, true);`

Scope

- Variables that are defined outside of any functions are part of the *global* scope.
- Global variables can be accessed by any other piece of the script.
- Variables defined within a function are part of that function's *local* scope.
- Local variables are created each time a function is called. The values are not shared between function calls.
- Descendant (child) scopes are always aware of the variables within their ancestors' (parent) scope.
- Ancestor scopes are *not* aware of the variables within their descendants' scopes.
- You can pass variable values outside of the function by returning its value.
- **Hoisting** is when you reassign a global variable's value within a function.
- You can avoid hoisting by always using `var` when declaring variables.

- Hoisting example:

```
var someVar = 0;
console.log(someVar);
>> 0

function myFunction() {
  //hoisting
  someVar = "cat";
  return "No problems here. Move along, move along."
}

myFunction();
console.log(someVar);
>> cat
```

String Concatenation

- Joining two or more strings together.

- Example:

```
var lastName = "Williams";
var midName = "Dee"
var fullName = "Billy " + midName + " " + lastName;
```

- You can build strings with the `+=` operator:

```
var someString = "Mary";
someString += " had a little lamb";
//the above means the same as:
someString = someString + "had a little lamb";
```

- You can concatenate different types of primitives together into one string.

`alert`, `prompt`, and `confirm`

- The `alert` function allows you to show the user pop-up messages in an OK or Cancel message box.
- The `prompt` function works in a similar way, but provides a text input field for the user to enter input.
- Text received by the `prompt` function is received as a string.
- `confirm` produces a message box as well, but when OK is clicked it returns `true` otherwise it returns `false`.

Methods

The JavaScript methods index

String Methods

String Methods

`indexOf(i)`

Takes an integer as argument and returns the character at that position.

`split()`

Splits string objects into an array of strings.

Example 1:

```
var str = "this is a string"
var newStr = str.split(" ")
console.log(newStr)  // returns ["this", "is", "a", "string"]
```

Example 2:

```
var str = "is this string? yes it is."
var newStr = str.split("?")
console.log(newStr)  // returns ["is this string", " yes it is."]
```

`trim()`

Removes the whitespace around a string.

```
var str = "        Hola!    ";
console.log(str.trim());  // returns Hola!
```

`substring(a, b)`

Returns a string that is a peice of the original. It takes 2 arguments, the second one being optional:

1. index to begin substring
2. is the position at which the substring stops, and doesn't include `b`

`length()`

Returns the length of the string `str`.

`toUpperCase()`

Returns the uppercase version of `str`.

`toLowerCase()`

Returns the lowercase version of `str`.

`parseInt()`

Converts a valid string into an integer.

`charAt(i)`

Returns the character in a string at index `i`.

Array Methods

Array Methods

`length()`

Returns the number of elements in the array `data`.

`indexOf(i)`

Given an item, `i`, this method returns either the position, if the item is found, or -1, if the item is not found.

```
var testArray = ["Hello", "World"]
console.log(testArray.indexOf("World"))  // returns 1
console.log(testArray.indexOf("Earth"))  // returns -1
```

`slice()`

Slice returns selected elements in an array as a new array object. Does not change the original array (non-destructive). Selects element at start arguement and ends at *but does not include* the end argument.

Syntax:

```
arrayObj.slice(start, [end])
```

Example:

```
var animals = ["fish", "cow", "chicken", "pig", "moose", "elephant"];
var farmAnimals = animals.slice(1, 4);
console.log("preslice: ", animals);  // returns preslice: [ 'fish', 'cow', 'chi
console.log("postslice: ", farmAnimals);  // returns postslice:  ["cow", "chick
```

`splice()`

Splice returns selected elements in an array. Changes the original array (destructive).

Syntax:

```
arrayObj.splice(index, howmany, item1, ....., itemX)
```

Example:

```
var numbers = [1, 2, 3, 4, 5, 6];
console.log("presplice: ", numbers); // returns presplice:  [1, 2, 3, 4, 5, 6]
console.log("spliced: ", numbers.splice(2, 2)); // returns spliced:  [3, 4]
console.log("postsplice: ", numbers);  // returns postsplice [1, 2, 5, 6]
```

join()

Returns a string of all array items concatenated with the argument given in between each.

```
arr = ["happy", "birthday", "to", "you"]
str = arr.join(", ")
console.log(str)  // returns "happy, birthday, to, you"
```

concat()

```
var arr1 = [1, 2, 3]
var arr2 = [4, 5, 6]
var arr3 = arr1.concat(arr2)
console.log(arr3)  // returns [1, 2, 3, 4, 5, 6]
```

reverse()

- exactly as it sounds - it reverses!
- it reverses inplace - e.g., ot returns the same array in different order (destructive)
- doesn't work on strings, only on arrays

sort()

needs an answer!!!

Objects

Array Methods

`indexOf(i)`

```
indexOf(searchElement, fromIndex)
```

Given an item, `i`, this method returns either the position, if the item is found, or -1, if the item is not found. Optional: can also note the start index in the second argument.

Examples:

```
var testArray = ["Hello", "Sun", "World", "Sun", "Mars"]
console.log(testArray.indexOf("World")) // returns 2
console.log(testArray.indexOf("Earth")) // returns -1
console.log(testArray.indexOf("Sun", 2)) // returns 3
```

Case

`.toUpperCase()`

Returns the string value to uppercase.

```
console.log("hello".toUpperCase()); // returns "HELLO"
```

`.toLowerCase()`

Returns the string value to lowercase

```
console.log("BYE".toLowerCase()); // returns "bye"
```

`toString()`:

Returns the string representation of an object.