Into JavaScript

Atwood's Law: Any application that can be written in JavaScript, will eventually be written in JavaScript.

Try It Yourself

It cannot be emphasized enough: you should practice each of these concepts by typing the code yourself.

The easiest way to do that is to open up the developer tools console in your nearest browser (Firefox, Chrome, IE, etc.).

Try It Yourself

Tip: Typically, you can launch the developer console with a keyboard shortcut or from a menu item. For more detailed information about launching and using the console in your favorite browser, see "Mastering The Developer Tools Console" http://blog.teamtreehouse.com/mastering-developer-tools-console. To type multiple lines into the console at once, use `<shift>` + `<enter>` to move to the next new line. Once you hit `<enter>` by itself, the console will run everything you've just typed.

Try It Yourself

Elements Network Sources »

<top frame> ▼ Preserve log

```
> a = 21;

  b = a * 2;

  console.log( b );
  42                                    VM855:6
<· undefined
>
```

Values & Types

The following built-in types are available:

- String
- Number
- Boolean
- null and undefined
- Object

JavaScript provides a `typeof` operator that can examine a value and tell you what type it is.

```
var a;
typeof a;                    // "undefined"

a = "hello world";
typeof a;                    // "string"

a = 42;
typeof a;                    // "number"

a = true;
typeof a;                    // "boolean"

a = null;
typeof a;                    // "object" -- weird, bug

a = undefined;
typeof a;                    // "undefined"

a = { b: "c" };
typeof a;                    // "object"
```

Objects

Objects

The object type refers to a compound value where you can set properties (named locations) that each hold their own values of any type. This is perhaps one of the most useful value types in all of JavaScript.

Object Example

```
var obj = {
    a: "hello world",
    b: 42,
    c: true
};

obj.a;       // "hello world"
obj.b;       // 42
obj.c;       // true

obj["a"];    // "hello world"
obj["b"];    // 42
obj["c"];    // true
```

Objects

It may be helpful to think of this obj value visually:

obj

a:	b:	c:
"hello world"	42	true

Objects

Properties can either be accessed with dot notation (i.e., `obj.a`) or bracket notation (i.e., `obj["a"]`). Dot notation is shorter and generally easier to read, and is thus preferred when possible.

Bracket notation is useful if you have a property name that has special characters in it, like `obj["hello world!"]` -- such properties are often referred to as keys when accessed via bracket notation. The [ ] notation requires either a variable (explained next) or a string literal (which needs to be wrapped in " .. " or ' .. ').

Objects

Of course, bracket notation is also useful if you want to access a property/key but the name is stored in another variable, such as:

```
var obj = {
    a: "hello world",
    b: 42
};

var b = "a";

obj[b];          // "hello world"
obj["b"];        // 42
```

Exercise

1. Create a `person` object. Person has `firstName`, `lastName` and `age` properties (props).
2. Log `person` `firstName` and `lastName` in console.

Arrays

Arrays

An array is an object that holds values (of any type) not particularly in named properties/keys, but rather in numerically indexed positions.

Arrays

```
var arr = [
    "hello world",
    42,
    true
];

arr[0];          // "hello world"
arr[1];          // 42
arr[2];          // true
arr.length;      // 3

typeof arr;      // "object"
```

Arrays

It may be helpful to think of array visually

arr

0:	1:	2:
"hello world"	42	true

Arrays

Because arrays are special objects (as typeof implies), they can also have properties, including the automatically updated length property.

You theoretically could use an array as a normal object with your own named properties, or you could use an object but only give it numeric properties (0, 1, etc.) similar to an array. However, this would generally be considered improper usage of the respective types.

The best and most natural approach is to use arrays for numerically positioned values and use objects for named properties.

Exercise

1. Create two arrays. `array1` has "Banana" and "Orange". `array2` has "Apple" and "Mango".
2. Join two arrays with method `concat` and log result to console.
3. Add new element "Lemon" to `array1` using method `push` and log result to console.
4. Add an element "Strawberry" to position `1` in `array2` using method `splice` and log result to console.

Functions

Functions

The other object subtype you'll use all over your JS programs is a function.

Functions are a subtype of objects -- `typeof` returns "function", which implies that a function is a main type -- and can thus have properties, but you typically will only use function object properties (like `foo.bar`) in limited cases.

Function Example

```
function foo() {
    return 42;
}

foo.bar = "hello world";

typeof foo;          // "function"
typeof foo();        // "number"
typeof foo.bar;      // "string"
```

Built-In Type Methods

The built-in types and subtypes we've just discussed have behaviors exposed as properties and methods that are quite powerful and useful.

Built-In Type Methods Example

```
var a = "hello world";
var b = 3.14159;

a.length;                    // 11
a.toUpperCase();             // "HELLO WORLD"
b.toFixed(4);                // "3.1416"
```

Built-In Type Methods Example

The "how" behind being able to call `a.toUpperCase()` is more complicated than just that method existing on the value.

Briefly, there is a `String` (capital S) object wrapper form, typically called a "native," that pairs with the primitive string type; it's this object wrapper that defines the `toUpperCase()` method on its prototype.

When you use a primitive value like "hello world" as an object by referencing a property or method (e.g., `a.toUpperCase()` in the previous snippet), JS automatically "boxes" the value to its object wrapper counterpart (hidden under the covers).

Exercise

1. Create `function` `hello` and put `console.log('Hello, world!')` inside `function` block. Call the `function` `hello` and see the result.
2. Add `name` argument to `function` `hello` and change `console.log` to display `name` instead of 'world'.
3. Create function `add` with arguments `a` and `b`. Return the sum of `a` and `b`. Log the result to console.

Appendix

Truthy & Falsy, Variables, Conditionals, Loops

Truthy & Falsy

The specific list of "falsy" values in JavaScript is as follows:

`""` (empty string)

`0, -0, NaN` (invalid `number`)

`null, undefined`

`false`

Truthy & Falsy

Any value that's not on this "falsy" list is "truthy." Here are some examples of those:

`"hello"`

`42`

`true`

`[ ], [ 1, "2", 3 ]` (arrays)

`{ }, { a: 42 }` (objects)

`function foo() { .. }` (functions)

Variables

In JavaScript, variable names (including function names) must be valid identifiers.

An identifier must start with `a-z`, `A-Z`, `$`, or `_`. It can then contain any of those characters plus the numerals `0-9`.

Conditionals

The most common one is the if statement. Essentially, you're saying, "*If* this condition is true, do the following...". For example:

```
var bank_balance = 302.13;
var amount = 99.99;

if (amount < bank_balance) {
    console.log( "I want to buy this phone!" );
}
```

Loops

The for loop has three clauses: the initialization clause (`var i=0`), the conditional test clause (`i <= 9`), and the update clause (`i = i + 1`). So if you're going to do counting with your loop iterations, for is a more compact and often easier form to understand and write.

```
for (var i = 0; i <= 9; i = i + 1) {
    console.log( i );
}
// 0 1 2 3 4 5 6 7 8 9
```

Exercise

1. Loop through `array1` from 'Arrays' exercise and log every element to console.
2. Do the same thing but use `forEach` method.

Bonus:

1. Join `array1` and `array2` from 'Arrays' exercise, then do `forEach` on joined array and log every fruit that has 6 characters. Do not create third variable for this. Hint: you can call multiple methods one after another (also called 'chaining').