Introduction to Javascript

Further reading:

http://eloquentjavascript.net/

https://developer.mozilla.org/en-US/docs/Web/JavaScript

(italiano) https://developer.mozilla.org/it/docs/Web/JavaScript

JavaScript History

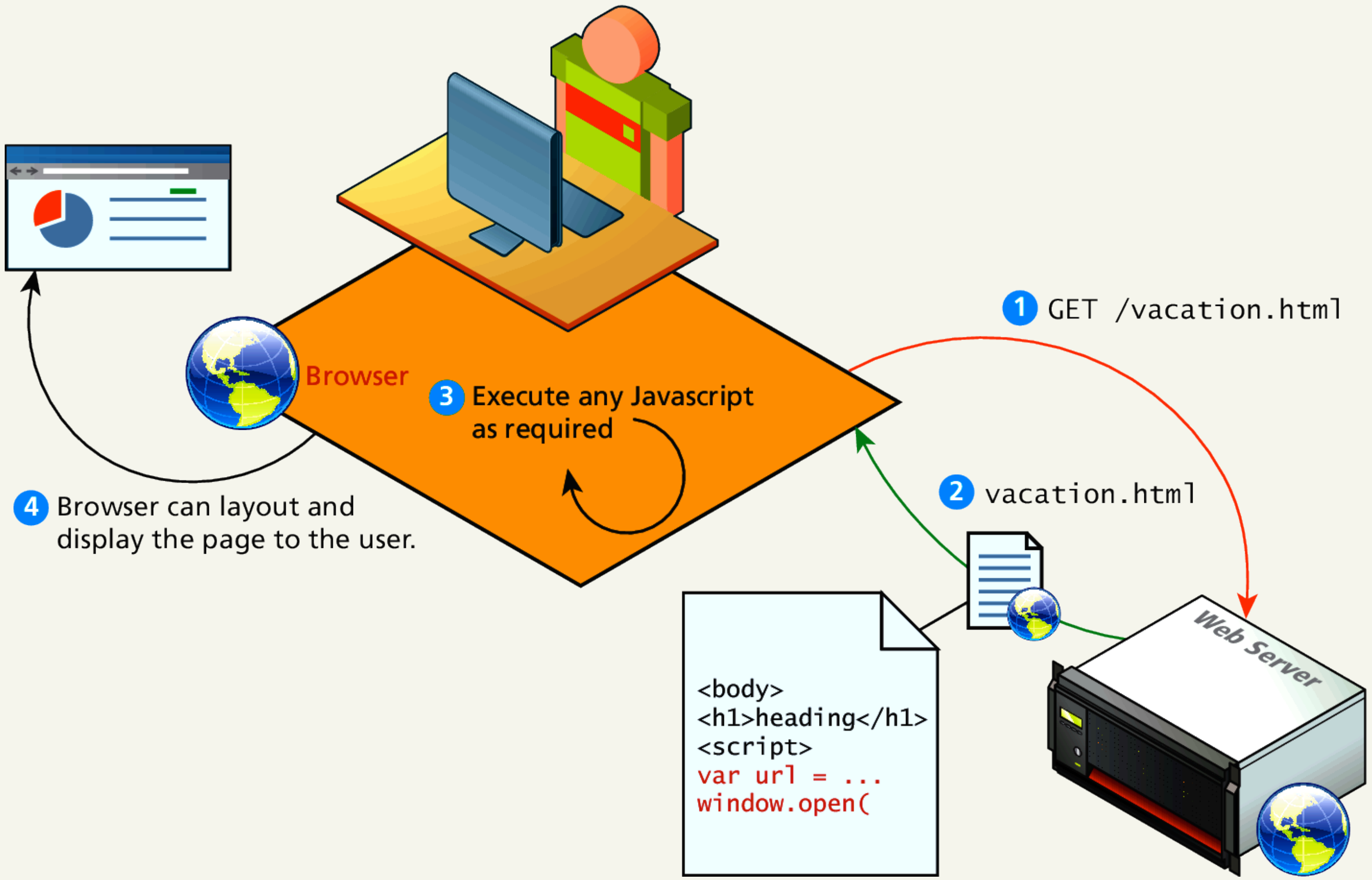
- JavaScript was introduced by Netscape in their Navigator browser back in 1996.
- It was originally going to be called LiveScript, but it was renamed to capitalize on the popularity of Sun Microsystem's Java language
- JavaScript is in fact an implementation of a standardized scripting language called **ECMAScript, today at 6th edition (June 2015)**

What is JavaScript

- Object-oriented dynamic language
  - types and operators, standard built-in objects, and methods.
- Syntax is based on the Java and C languages
  - many structures from those languages apply to JavaScript as well
- Designed to run as a scripting language in a host environment
  - Browser, Node.js, database CouchDB, embedded computers, GNOME, Adobe Photoshop, ect.

Client-Side Scripting

Let the client compute


Browser
1 GET /vacation.html
2 vacation.html
3 Execute any Javascript as required
4 Browser can layout and display the page to the user.
Web Server

```
<body>
<h1>heading</h1>
<script>
var url = ...
window.open(
```

Client-Side Scripting

It’s good

There are many **advantages** of client-side scripting:

- Processing can be offloaded from the server to client machines, thereby reducing the load on the server.
- The browser can respond more rapidly to user events than a request to a remote server ever could, which improves the user experience.
- JavaScript can interact with the downloaded HTML in a way that the server cannot, creating a user experience more like desktop software than simple HTML ever could.

Where does JavaScript go?

JavaScript can be linked to an HTML page in a number of ways.

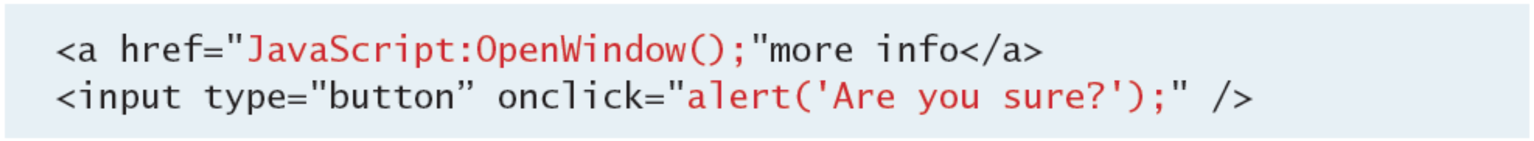
- Inline
- Embedded
- External

Inline JavaScript

Mash it in

Inline JavaScript refers to the practice of including JavaScript code directly within certain HTML attributes

Inline JavaScript is a real maintenance nightmare



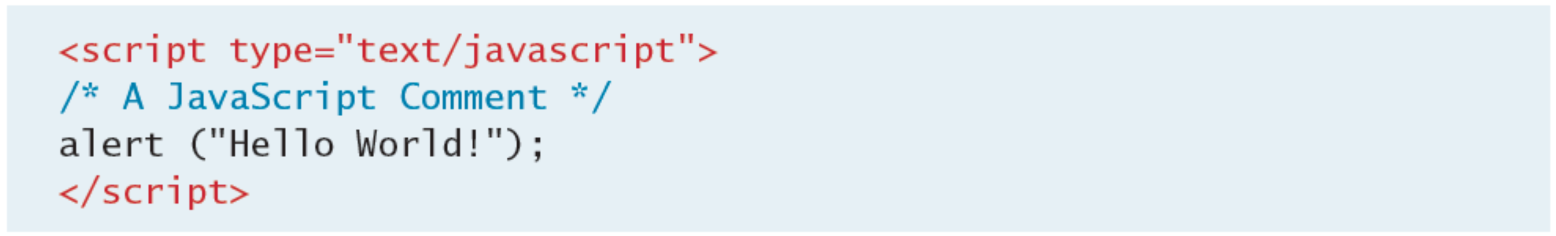
```
<a href="JavaScript:OpenWindow();"more info</a>
<input type="button" onclick="alert('Are you sure?');" />
```

LISTING 6.1 Inline JavaScript example

Embedded JavaScript

Better

Embedded JavaScript refers to the practice of placing JavaScript code within a <script> element



```
<script type="text/javascript">
/* A JavaScript Comment */
alert ("Hello World!");
</script>
```

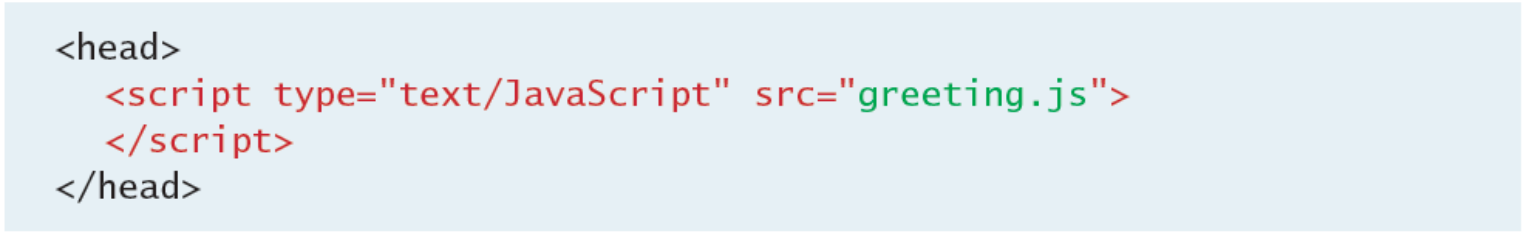
LISTING 6.2 Embedded JavaScript example

External JavaScript

Better

JavaScript supports this separation by allowing links to an external file that contains the JavaScript.

By convention, JavaScript external files have the extension .js.



```html
<head>
  <script type="text/JavaScript" src="greeting.js">
  </script>
</head>
```

LISTING 6.3 External JavaScript example

Basic Concepts
About Values & Types

Definition

A value represents the most basic data we can deal with

value

A type is a ***set*** of data values, and there are exactly 6 types

Type :: { v1 , v2 , v3 , … }

There are 5 primitive (non-Object) types

Undefined `undefined`

Null `null`

Boolean `true` `false`

Number `.33` `-3.14` `011` `7e-2` `-Infinity` `0x101` `NaN`

(IEEE754 64-bit doubles)

String `""` `"Hello world!"` `"哈囉。"` `"\n"` `"\""` `'w Single Quotes'`

(Any finite ordered sequence of 16-bit unsigned integers)

Any value here is called ***a primitive value***

Numbers

1. Numbers are numeric values such as **13**
2. Numbers in Javascript are made up of **64bits** which is about 18 quintillion.
3. 64-bit floating point ("double")
4. Negative numbers and nonwhole numbers use bits which shrink the maximum number available.
5. Negative numbers use a bit to display the sign
6. Decimal numbers use some bits to store the position of the decimal point so the actual whole number is around 9 quadrillion (15 zeros).
7. Decimal numbers are written with a dot. **9.81**

3 Special Numbers

```
> 0/1
< 0
> 1/0
< Infinity
> 0/0
< NaN
> Infinity - 1
< Infinity
```

Infinity and -**Infinity**

Infinity - 1 is still Infinity

NaN stands for “not a number”

Any number of other numeric operations that don’t yield a precise, meaningful result.

Strings

Strings are text wrapped in quotes.

```
> "Patch my boat with chewing gum"
> 'Monkeys wave goodbye'
```

Special characters are escaped using the \
such as \n for a newline.

```
> "This is the first line\nAnd this is the second"
< "This is the first line
  And this is the second"
```

If you actually need a backslash then you
add two backslashes. \\

Unary Operators

Not all operators are symbols such as ***** or **/**.

Some are written as words.

typeof is one of those

```
> console.log(typeof 4.5)
< number
> console.log(typeof "x")
< string
```

Unary operators take 1 params

Binary operators take 2 params

Boolean Values

Comparison

The **>** and **<** signs are binary operators that return a Boolean value.

```
> console.log(3 > 2)
< true
> console.log(3 < 2)
< false
```

Other similar operators are **>=** (greater than or equal to), **<=** (less than or equal to), **==** (equal to), and **!=** (not equal to).

Boolean Values

Logical operators

JavaScript supports three logical operators:

and (**&&**), *or* (**||**), and *not* (**!**)

```
> true && false
< false
> true && true
< true
> false || true
< true
> false || false
< false
> !true
< false
> !false
< true
> NaN == NaN
< false
```

Boolean Values

ternary operator

JavaScript supports a *conditional* or *ternary* operator:

: ?

```
> true ? 1 : 2
< 1
> false ? 1 : 2
< 2
```

Undefined Values

There are two special values, **null** and **undefined**, that are used to denote the absence of a meaningful value.

Many operations in the language that don't produce a meaningful value (you'll see some later) yield **undefined** simply because they have to yield some value.

```
> var test; test
< undefined
> var test=null; test
< null
```

Automatic Type Conversion

JavaScript goes out of its way to accept almost any program you give it, even programs that do odd things

```
> 8 * null
< 0
> "5" - 1
<  4
> "5" + 1
< "51"
> "five" * 2
< NaN
> false == 0
< true
```

JavaScript will quietly convert that value to the type it wants, using a set of rules that often aren't what you want or expect.

This is called *type coercion.*

Automatic Type Conversion

However, when null or undefined occurs on either side of the operator, it produces true only if both sides are one of null or undefined.

```
> null == undefined
< true
> null == 0
< false
```

This is useful if you want to test to see if a value has a real value instead null or undefined.

```
> var val;
> val == null
< true
```

Automatic Type Conversion

How do you determine false?

0, **NaN**, and the empty string (**""**) count as false.

```
> 0 == false
< true
> "" == false
< true
```

If you don't want automatic type conversion you can use **===** and **!==**.

It is highly recommended that you use **===** and **!==** to prevent unexpected type conversions.

```
> 0 === false
< false
> "" === false
< false
```

Variables

- With the keyword **var.** For example, *var x = 42.* This syntax can be used to declare both local and global variables.
- By simply assigning it a value. For example, *x = 42*. This always declares a global variable. You shouldn't use this variant.

After a variable has been defined,

it can be used as an expression

```
> var x = 42;
> y = 10;
> x * y
< 420
```

Variables

A variable without a value will be set to **undefined**.

A single var statement may define multiple variables.

```
var nothing;
console.log(nothing);
// → undefined

var one = 1, two = 2;
console.log(one + two);
// → 3

var input;
if(input === undefined){
  doThis();
} else {
  doThat();
}
```

Variable scope

- Variables outside any function are called **global**, because available to any code.
- Variables declared within a function (using **var**) are **local** because available only to that function
- Javascript **does not have block statement scope** like in java

```
if (true) {
  var x = 5;
}
console.log(x);  // 5

window.x = 1;
console.log(x); // 1
console.log(x == window.x); // true
```

Global variables are part of the *global object*. In web pages the global object is window, so you can set and access global variables using the window.*variable* syntax.

Variable hoisting

- Variable can be referred even if declared later.
- Variables are "lifted" to the top of the function or statement, and initialized as *undefined*

```
// What is i, $, p, and q afterwards?

var i = -1;

for (var i = 0; i < 10; i += 1) {
    var $ = -i;
}
if (true) {
    var p = 'FOO';
} else {
    var q = 'BAR';
}

// Check the next slide for an answer...
```

The code in previous page actually works like this one:

```
var i, $, p, q; // all undefined

i = -1;

for (i = 0; i < 10; i += 1) {
    $ = -i;
}
if (true) {
    p = 'FOO';
} else {
    q = 'BAR';
}

// i=10, $=-9, p='FOO', q=undefined
```

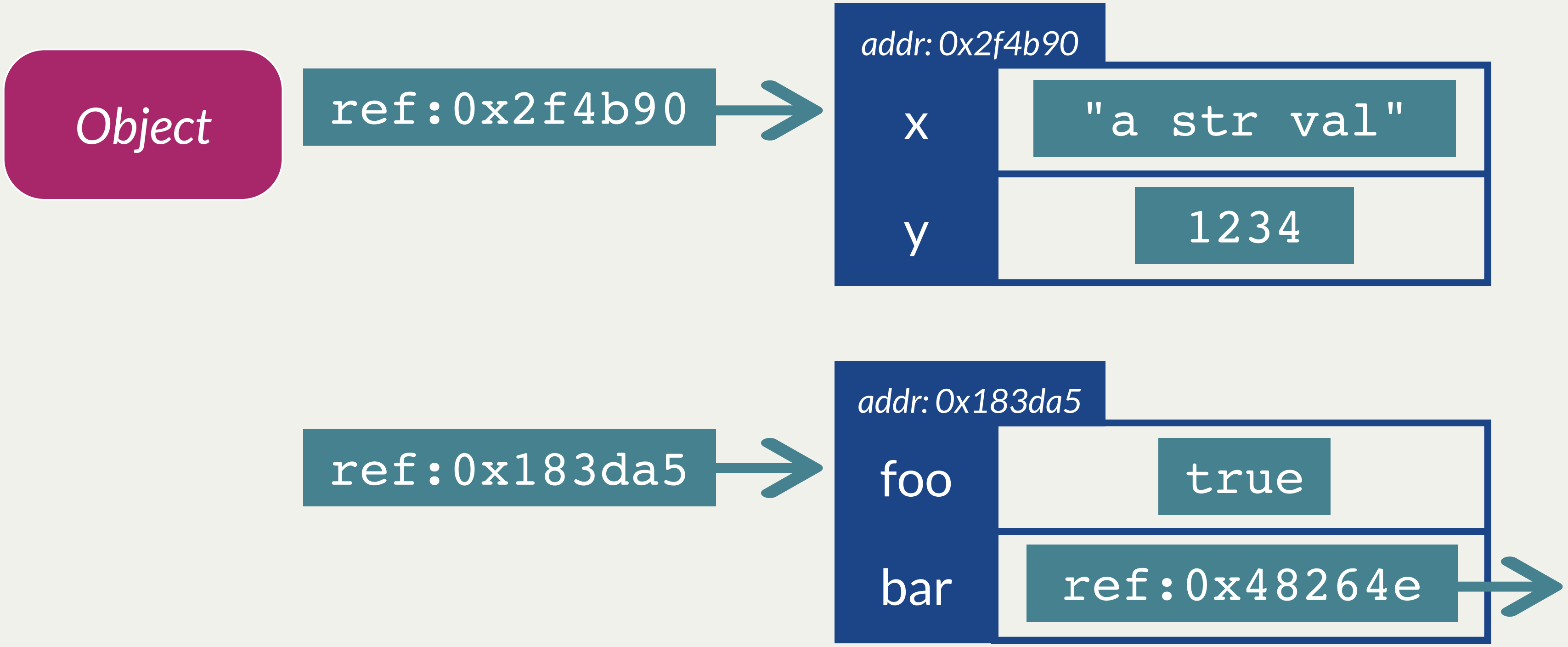
When the program runs, all variable declarations are moved up ***to the top of the current scope***.

There are 5 primitive (non-Object) types

Undefined	`undefined`
Null	`null`
Boolean	`true` `false`
Number (IEEE754 64-bit doubles)	`.33` `-3.14` `011` `7e-2` `-Infinity` `0x101` `NaN`
String	`""` `"Hello world!"` `"哈囉。"` `"\n"` `"\""` `'w Single Quotes'` (Any finite ordered sequence of 16-bit unsigned integers)

Any value here is called ***a primitive value***

And then there is the "Object" type



Any value of this type is ***a reference*** ***to some “object”***;
sometimes we would simply call such value ***an object***

Definition

An object is a
collection of ***properties***

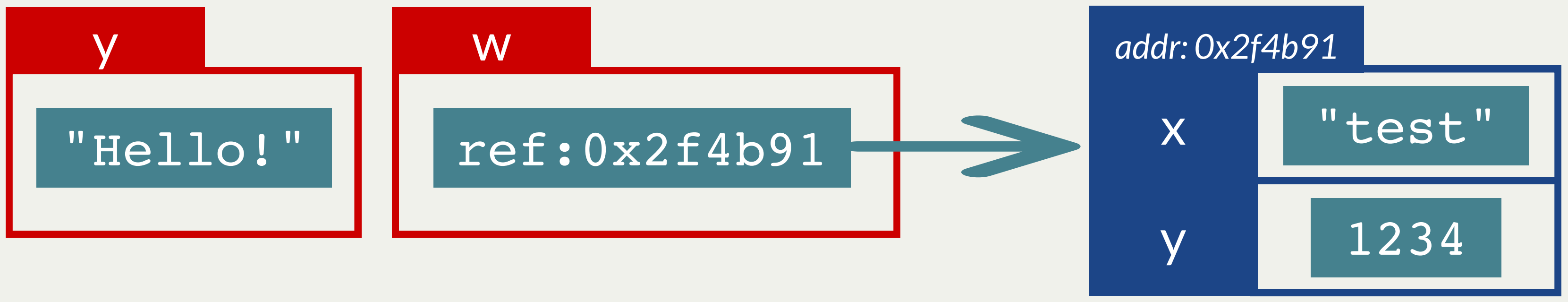
A property is a
named container for a ***value***
w/ some additional attributes

Definition

The ***name of a property*** is called ***a key***; thus, ***an object*** can be considered as ***a collection of key-value pairs***.

There are similar concepts in other programming languages, e.g., *Map*, *Dictionary*, *Associative Array*, *Symbol Table*, *Hash Table*, ...

A “*variable*” vs a “*property*” in an object



```
// Value containers
var y = "Hello!";
var w = {
    x: "test",
    y: 1234
};
```

```
// To get the values
y;       // "Hello!"
w;       // (the object ref)
w.x;     // "test"
w['x'];  // "test"
w.y;     // 1234
w["y"];  // 1234
```

Object Initialiser (Object Literal)

The notation using a pair of curly braces
to ***initialize*** a new JavaScript object.

```
var w = {
    x: "test",
    y: 1234,
    z: {},
    w: {},
    "": "hi"
};
```

```
var w = new Object();
w.x = "test";
w.y = 1234;
w.z = new Object();
w.w = new Object();
w[""] = "hi";
```

The code on the left-hand side has exactly the
same result as the one on the right-hand side

Properties

The two most common ways to access properties in JavaScript are with a **dot** and with **square brackets**.

Both **value.x** and **value[x]** access a property on value—but not necessarily the same property.

The difference is in how **x** *is interpreted.*

Named Properties - dot

When using a dot, the part after the dot must be a valid variable name, and it ***names*** the property.

```
var events = ["work","television"];

var day = {
    events: events,
    squirrel: false
};

console.log(day.events);
// → ["work","television"]

console.log(screen.squirrel);
// → false

console.log(events.length);
// → 2
```

Properties - bracket

When using square brackets, the expression between the brackets is ***evaluated*** to get the property name.

"2" and **"john doe"** aren't valid variable names so they can't be accessed through the dot notation.

```
var random = {
  1: "one"
  "2": 2
  "john doe": "unknown"
};

var x = 1;
console.log(random[x])
// → "one"

console.log(random["2"])
// → 2

console.log(random["john doe"]);
// → "unknown"
```

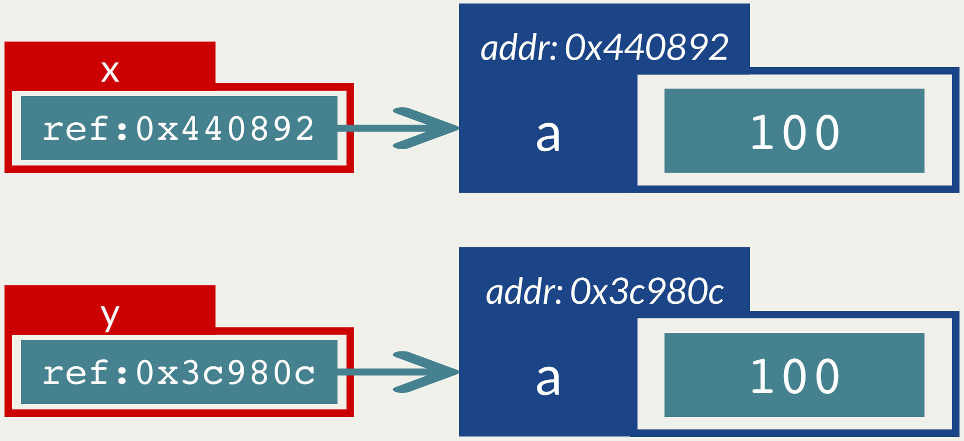
Add/Get/Set/Remove A Property

We can dynamically modify an object after its creation

```
var obj = {
    1  : "Hello",
    "3": "Good",
    x  : "JavaScript",
    foo: 101,
    bar: true,
    "" : null
};

obj["2"] = "World";      // *1 Add & Set
obj["1"];                // *2 Get        -> "Hello"
obj[2];                  // *3 Get        -> "World"
obj[3];                  // *4 Get        -> "Good"
obj.foo = 202;           // *5 Set
delete obj.bar;          // *6 Remove
delete obj[""];          // *7 Remove
```

Don't Forget Any Value Of The Object Type Is Actually A “Reference”

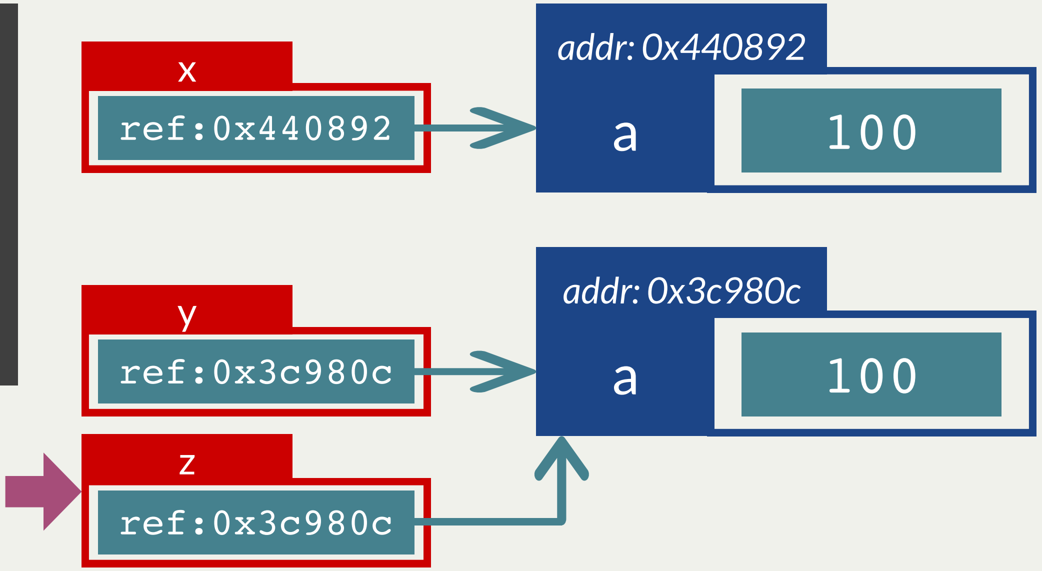
```
var x = { a: 100 };
var y = { a: 100 };
```



Similar to the “pointer” / “address” concept
in programming languages like C or C++

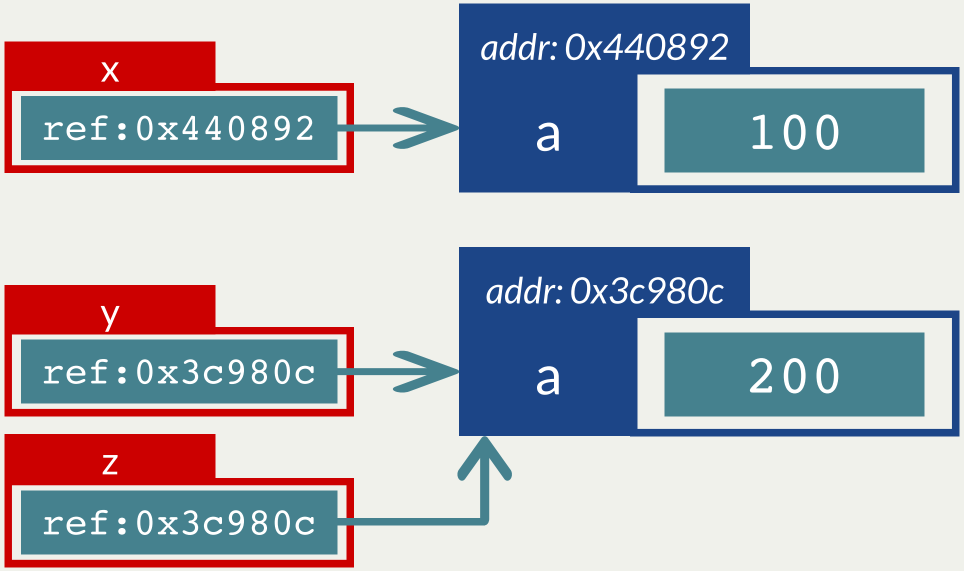
Don't Forget Any Value Of The Object Type Is Actually A “Reference”

```
var x = { a: 100 };
var y = { a: 100 };
var z = y;

x === y; // false
y === z; // true
```



Don't Forget Any Value Of The Object Type Is Actually A “Reference”

```
var x = { a: 100 };
var y = { a: 100 };
var z = y;

x === y; // false
y === z; // true


z.a = 200;

x.a; // 100
y.a; // 200
z.a; // 200
```



Definition

A function is an ***object*** that is ***callable***

Functions

console.log

Most JavaScript systems (including all modern web browsers and Node.js) provide a console.log function that writes out its arguments to *some* text output device

```
var x = 30;
console.log("the value of x is", x);
// → the value of x is 30
```

Notice **console.log** has a period in the middle of it. **console.log** is actually an expression that retrieves the **log** property from the value held by the **console** variable.

Defining a function

A function definition is just a regular variable definition where the value given to the variable happens to be a function.

```
// The variable square is being set to a function
// that has one parameter, x.
var square = function(x) { // Opening brace
  return x * x;            // Function body
};                         // Closing brace

console.log(square(12));
// → 144
```

A function starts with the keyword *function*, can have have a set of *parameters* and a *body* which is wrapped in curly braces {}.

Defining a function

Some functions produce a value
while others produce a side effect.

```
var square = function(x) {
  return x * x;             // produce a value
};
console.log(square(12));
// → 144

var makeNoise = function() {
  console.log("Pling!");  // side-effect
};
makeNoise();
// → Pling!
```

A **return** statement determines the value the function returns and it immediately jumps out of the current function.

The **return** keyword without an expression after it will cause the function to return **undefined**.

Function Invocation

A pair of parentheses invokes the preceding function;
the values of zero or more arguments are passed in.

Each time a function is called/invoked, it has ***its own variable scope*** with ***local variables*** and ***arguments*** filled with corresponding values

```
var a = 7;
var sayhi = function (name) {
    var a = "Hello " + name;
    console.log(a);
    return a;
};

sayhi("J"); // "Hello J"
a; // 7
```

Parameters and Scopes

- Parameters to a function behave like regular variables, but their initial values are given by the *caller* of the function
- Parameters are passed by copy for primitive types, and by copy-reference for objects

```
var square = function(x) {
  return x * x;             // produce a value
};
console.log(square(12)); // → 144

var setName = function(x,newName) {
    x.name = newName;    //will modify x
    console.log(x); //{name: "pluto"}
    x = {name : "new"}; //will not change the original parameter
    console.log(x); //{name: "new"}
}

var you = {name: "pippo"};
setName(you,"pluto");
console.log(you.name); // "pluto"
```

Parameters and Scopes

```
var x = "outside";
var f1 = function() {
  var x = "inside f1";
};
f1();
console.log(x); // → outside

var f2 = function() {
  x = "inside f2";
};
f2();
console.log(x); // → inside f2
```

Variables created with the **var** keyword inside a function including their parameters, are ***local*** to that function

When # of Arguments Not Matched

In JavaScript, it is OK to invoke a function
with a ***mismatched* number of arguments**.

At run time, any ***argument without passing a value*** is filled with the "***undefined***" value, just like a local variable without any assignment.

```
var f = function (a, b) {
    console.log(typeof b);
    return a + b;
};

f(3);         // NaN ("undefined" printed)
f(3, 4);      // 7   ("number" printed)
f(3, 4, 5);   // 7   ("number" printed)
```

Function Not Returning A Value

A function does ***not*** necessarily need to ***return a value explicitly***.

When that is the case, the "***undefined***" value is ***returned***.

```
var f = function () {
    return;
};

var g = function () {
};

var foo = f();
foo; // undefined
g(); // undefined
```

Functions as Values

Function *variables* usually simply act as names for a specific piece of the program. Such a variable is defined once and never changed. This makes it easy to start confusing the function and its name.

```
var func = function() {};
```

function *variable*: Left side of the equation.
function *value*: Right side of the equation.

Functions as Values

But the two are different. A function value can do all the things that other values can do—you can use it in arbitrary expressions, not just call it.

It is possible to store a function value in a new place, pass it as an argument to a function, and so on.

Similarly, a *variable* that holds a function is still just a regular variable and can be assigned a new value.

```
var launchMissiles = function(value) {
  missileSystem.launch("now");
};
if (safeMode)
  launchMissiles = function(value) {/* do nothing */};
```

Function Declaration Statement (1/2)

Functions can be used ***before*** their ***function declaration*** statements, because...

```
// We can invoke a function
// before its "declaration"
//
// Please see the next slide for the reason.......


var x = 100;

plusOne(x); // 101
plusOne(y); // NaN

var y = 999;

function plusOne(n) {
    return n + 1;
}
```

This is a "**function declaration**" statement that ***requires*** a "**name**"

Function Declaration Statement (2/2)

When the program runs, those ***function declaration*** statements work *like* this:

```
var x, y, plusOne;        // Variable declarations

plusOne = function (n) { // Function declarations
    return n + 1;        // are also moved to the
};                       // top of current scope

x = 100;

plusOne(x); // 101
plusOne(y); // NaN

y = 999;
```

When program runs, all function declarations are moved up ***to the top of the current scope***, as well as all variable declaration statements.

Closure

What happens to local variables when the function call that created them is no longer active?

```
function wrapValue(n) {
  var localVariable = n;
  return function() { return localVariable; };
}

var wrap1 = wrapValue(1);
var wrap2 = wrapValue(2);
console.log(wrap1());
// → 1
console.log(wrap2());
// → 2
```

This feature—being able to reference a specific instance of local variables in an enclosing function—is called *closure*. A function that "closes over" some local variables is called a closure.

Closure

With a slight change, we can turn the previous example into a way to create functions that multiply by an arbitrary amount.

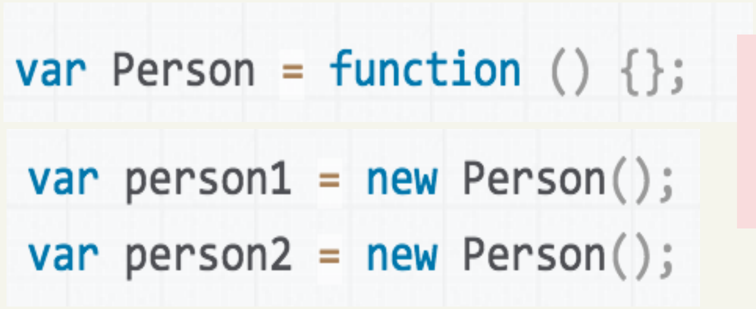
```
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

var twice = multiplier(2);
console.log(twice(5));
// → 10
```

A good mental model is to think of the function keyword as "freezing" the code in its body and wrapping it into a package (the function value). So when you read return function(...) {...}, think of it as returning a handle to a piece of computation, frozen for later use.

Custom Objects (classes)

- JavaScript is a prototype-based language and contains no class statement

- JavaScript uses functions as constructors for classes. Defining a class is as easy as defining a function



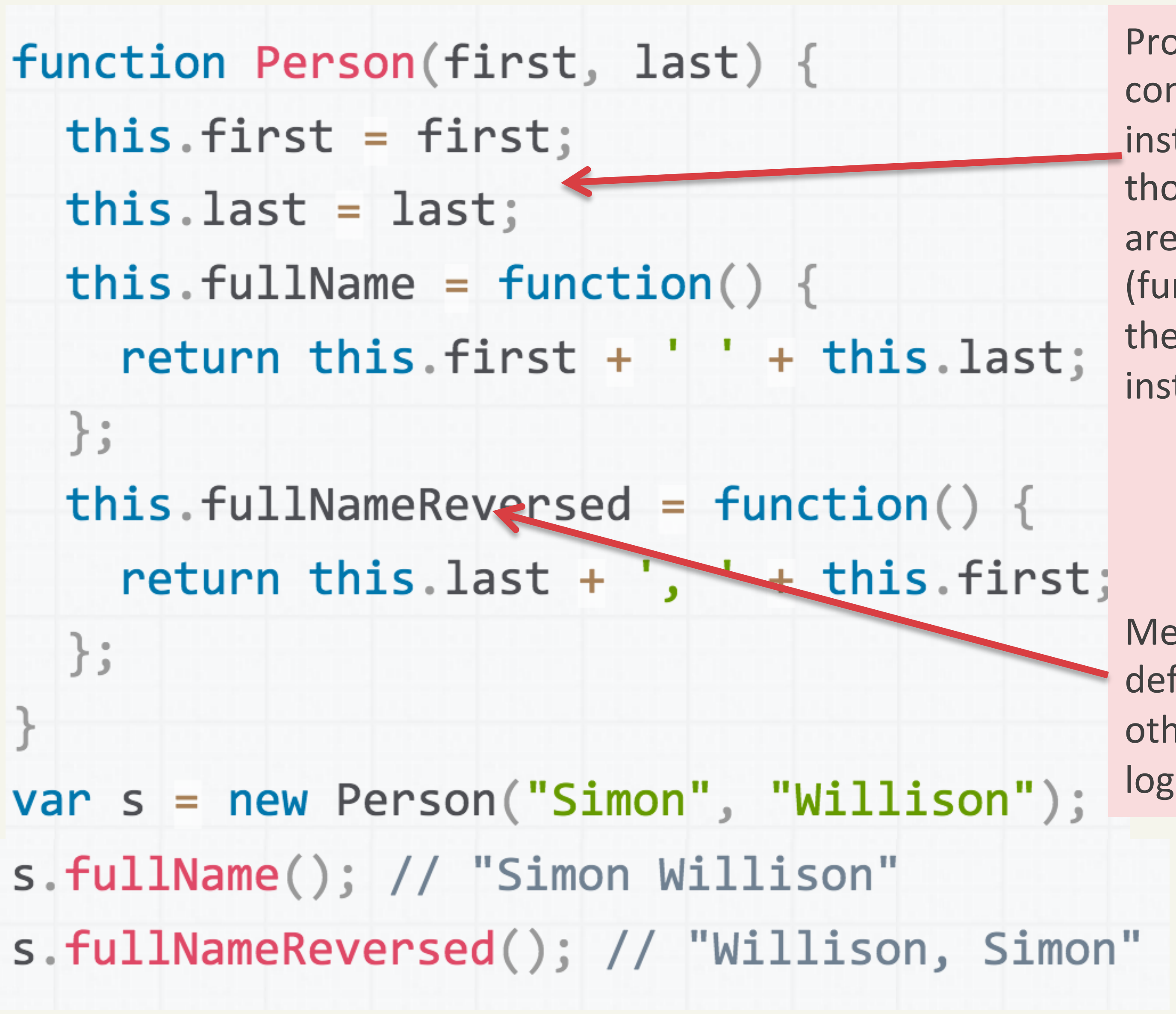
```
var Person = function () {};
```

```
var person1 = new Person();
var person2 = new Person();
```

//constructor

//instances

```
function Person(first, last) {
  this.first = first;
  this.last = last;
  this.fullName = function() {
    return this.first + ' ' + this.last;
  };
  this.fullNameReversed = function() {
    return this.last + ', ' + this.first;
  };
}
var s = new Person("Simon", "Willison");
s.fullName(); // "Simon Willison"
s.fullNameReversed(); // "Willison, Simon"
```



```
function Person(first, last) {
  this.first = first;
  this.last = last;
  this.fullName = function() {
    return this.first + ' ' + this.last;
  };
  this.fullNameReversed = function() {
    return this.last + ', ' + this.first;
  };
}
var s = new Person("Simon", "Willison");
s.fullName(); // "Simon Willison"
s.fullNameReversed(); // "Willison, Simon"
```

Properties are variables contained in the class; every instance of the object has those properties. Properties are set in the constructor (function) of the class so that they are created on each instance.

Methods are functions (and defined like functions), but otherwise follow the same logic as properties

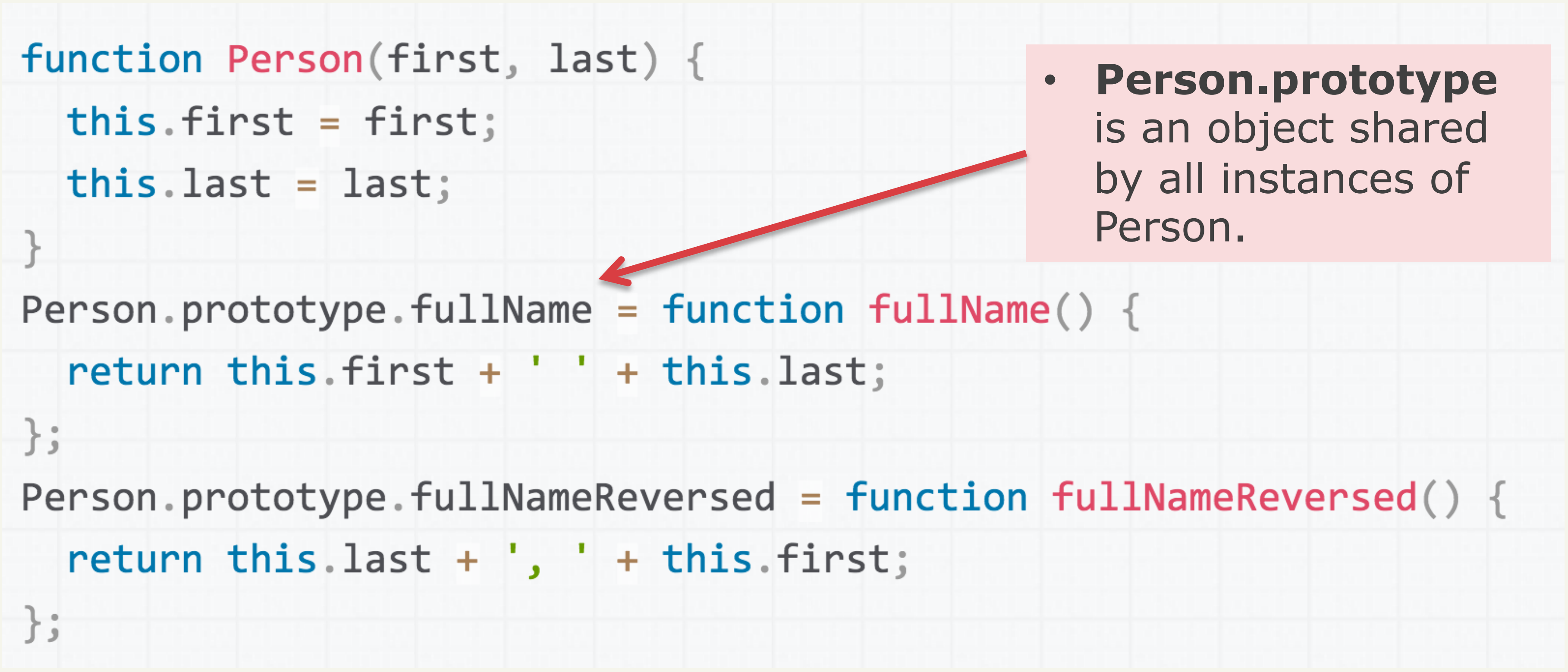
```
function Person(first, last) {
  this.first = first;
  this.last = last;
  this.fullName = function() {
    return this.first + ' ' + this.last;
  };
  this.fullNameReversed = function() {
    return this.last + ', ' + this.first;
  };
}
var s = new Person("Simon", "Willison");
```

When a function is invoked ***as a method*** of some object, the ***this*** value during the function call refers to that object

new is strongly related to this. It creates a new empty object, and then calls the function specified, with **this** set to that new object.

```
s.fullName(); // "Simon Willison"
s.fullNameReversed(); // "Willison, Simon"
```

Each method invocation creates a new function object!



```javascript
function Person(first, last) {
  this.first = first;
  this.last = last;
}
Person.prototype.fullName = function fullName() {
  return this.first + ' ' + this.last;
};
Person.prototype.fullNameReversed = function fullNameReversed() {
  return this.last + ', ' + this.first;
};
```

- **Person.prototype** is an object shared by all instances of Person.

- **Prototype chain**: any time you attempt to access a property of Person that isn't set, JavaScript will check **Person.prototype** to see if that property exists there instead.

- As a result, anything assigned to Person.prototype becomes available to all instances of that constructor via the **this** object.

Objects Included in JavaScript

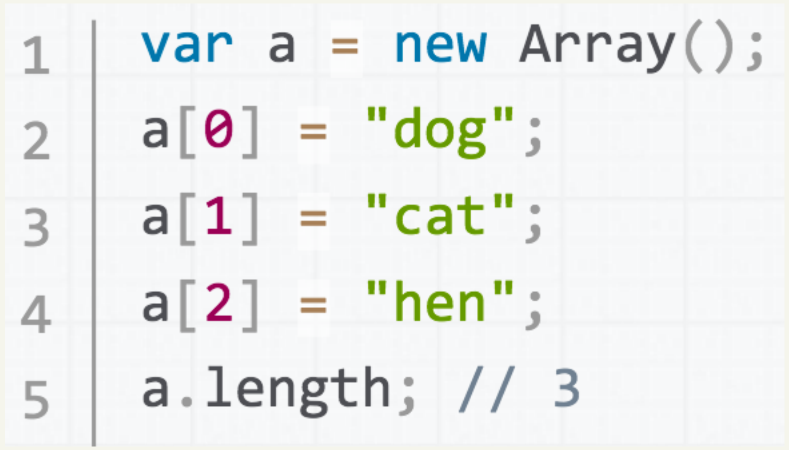
A number of useful objects are included with JavaScript including:

- Array
- Date
- Math
- String, Number, Boolean (primitive wrappers)
- Dom objects

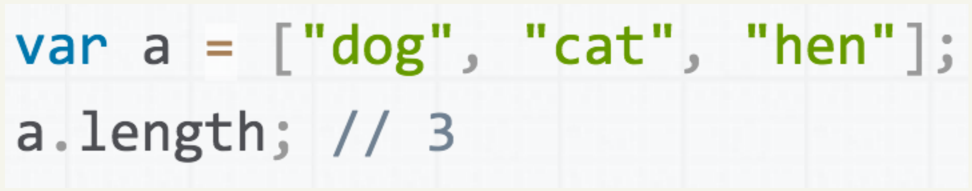
Arrays

Arrays are one of the most used data structures.

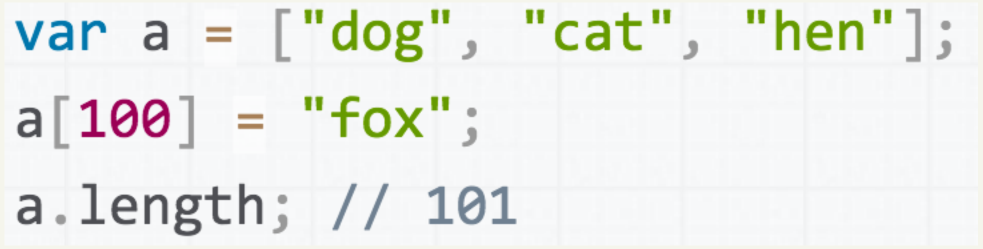
In practice, this class is defined to behave more like a linked list in that it can be resized dynamically



```javascript
var a = new Array();
a[0] = "dog";
a[1] = "cat";
a[2] = "hen";
a.length; // 3
```



```javascript
var a = ["dog", "cat", "hen"];
a.length; // 3
```



```javascript
var a = ["dog", "cat", "hen"];
a[100] = "fox";
a.length; // 101
```

Arrays

Modifying an array

To add an item to an existing array, you can use the **push** method.

greetings.**push**("Good Evening");

The **pop** method can be used to remove an item from the back of an array.

Additional methods: concat(), slice(), join(), reverse(), shift(), and sort()

Math

The **Math class** allows one to access common mathematic functions and common values quickly in one place.

Contains methods such as max(), min(), pow(), sqrt(), and exp(), and trigonometric functions such as sin(), cos(), and arctan().

Many mathematical constants are defined such as PI, E, SQRT2, and some others

Math.PI; *// 3.141592657*

Math.sqrt(4); *// square root of 4 is 2.*

Math.random(); *// random number between 0 and 1*

Date

It allows you to quickly calculate the current date or create date objects for particular dates.

To display today's date as a string, we would simply create a new object and use the toString() method.

var d = **new Date();**

// This outputs Today is Mon Nov 12 2012 15:40:19 GMT-0700

alert ("Today is "+ **d.toString()**);

Window

The window object in JavaScript corresponds to the browser itself. Through it, you can access the current page's URL, the browser's history, and what's being displayed in the status bar, as well as opening new browser windows.

In fact, the alert() function mentioned earlier is actually a method of the window object.