https://slides.com/concise/js/

# concise JavaScript

A concise and accurate JavaScript tutorial/notes written for those entering the JavaScript world for the first time but already have experience with other languages

Some slides extracted from above reference

# Basic Concepts About Variables

Definition

# A variable is a named container for a ***value***

The ***name*** that refers to a variable
is sometime called *an* ***identifier***

```
var x;
var y = "Hello JS!";
var z;
```

These red boxes are ***variables***, and each of them has a ***name (identifier)***


Any ***JavaScript value*** can be contained within these boxes

```
var x;
var y = "Hello JS!";
var z;
z = false;
z = 101;
```


We can ***assign*** another ***value*** to a variable later after its creation

# Curly-brace blocks do *not* introduce new variable scopes in JavaScript

```
// 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.***

**let** & **const** do NOT behave like **var**

They introduce ‘Block Scoped’ variables that:

- cannot be redefined
- can only be used in the scope they are declared in

I.E. They closely match the way Java Local Variables are scoped.

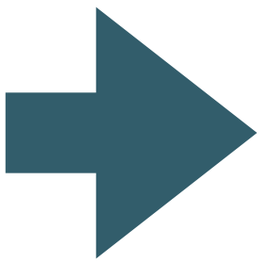
# const & let are Block Scoped

```
const greeting = 'hello';

{
  const greeting = ‘howdy’;
  console.log(greeting);
}

console.log(greeting);
```

- 2 variables called **greeting** defined in two separate scopes

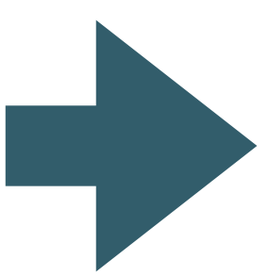


```
howdy
hello
```

# var is not Block Scoped

```
var greeting = 'hello';
{
  var greeting = 'howdy';
  console.log(greeting);
}
console.log(greeting);
```

- 1 variable called **greeting** defined.
- Second **greeting** is ***Hoisted*** to the outer scope



```
howdy
howdy
```

## **let** & **const** VS **var**

Because they are more predictable, we will always prefer **let** & **const** to **var**

# Reserved Words

Some keywords can not be used as variable names:

```
null true false break do instanceof typeof
case else new var catch finally return void
continue for switch while debugger function
this with default if throw delete in try
class enum extends super const export import

implements let private public yield
interface package protected static
```

We don't need to remember them all. Just be aware of the possible cause for some SyntaxError exceptions in our program.

# 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`

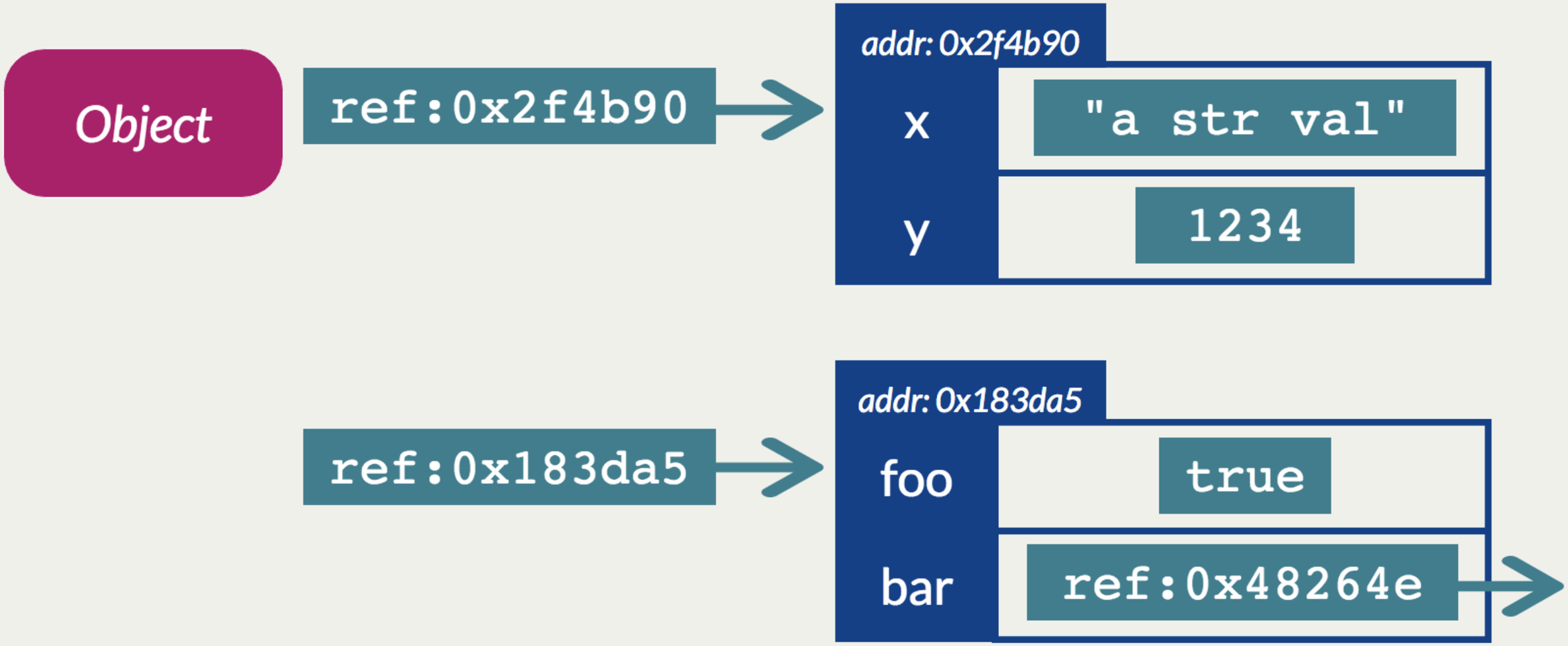
(IEEE754 64-bit doubles)

*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***

# 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*, ...

# To Refer To A Value

- ***Literal notation*** for the value
- Expression involving a ***variable*** or a ***property within some object*** to get the value indirectly
- More complex expression involving function ***calls*** and ***operators***

# 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

# 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”

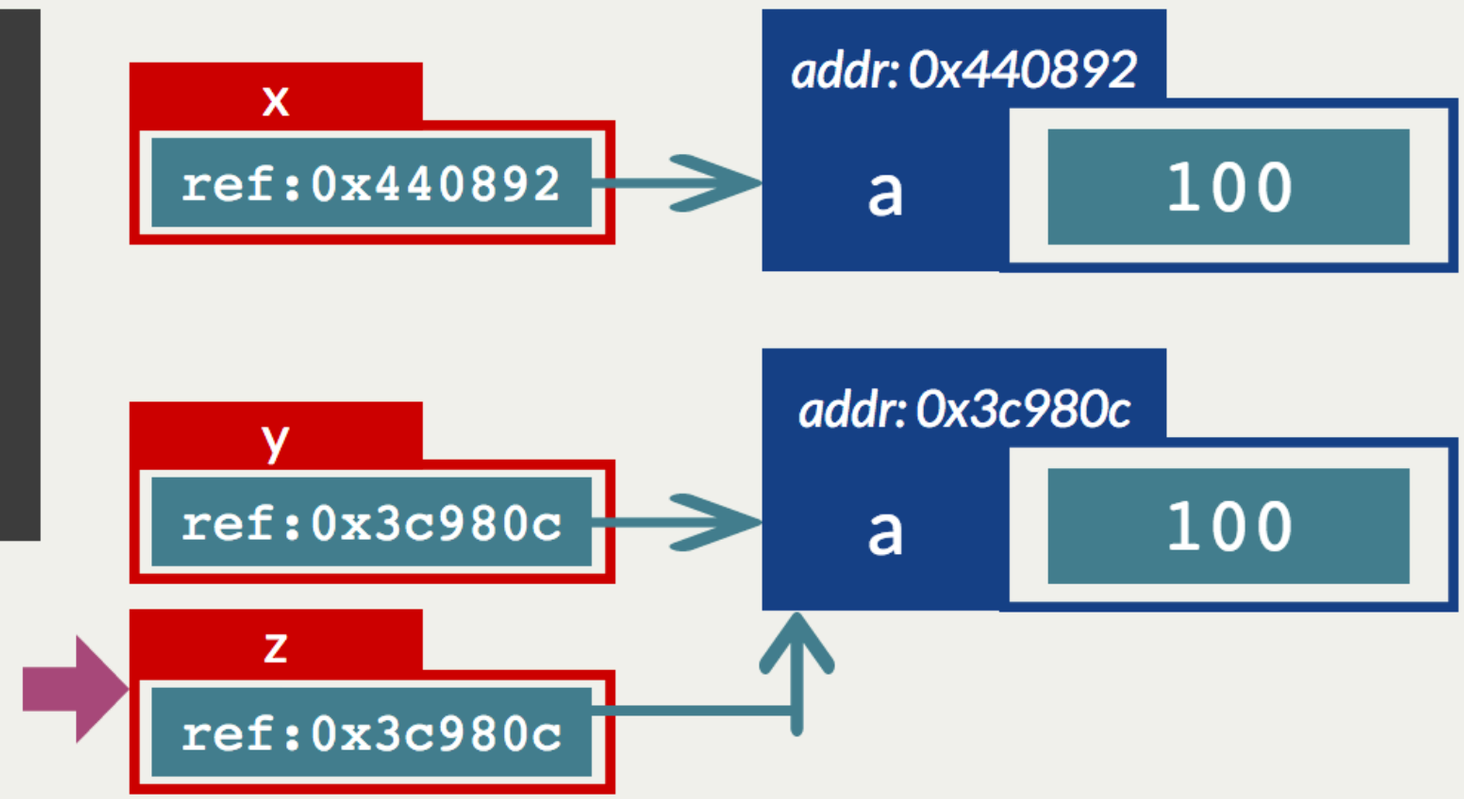
```
var x = { a: 100 };
var y = { a: 100 };
```


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
```

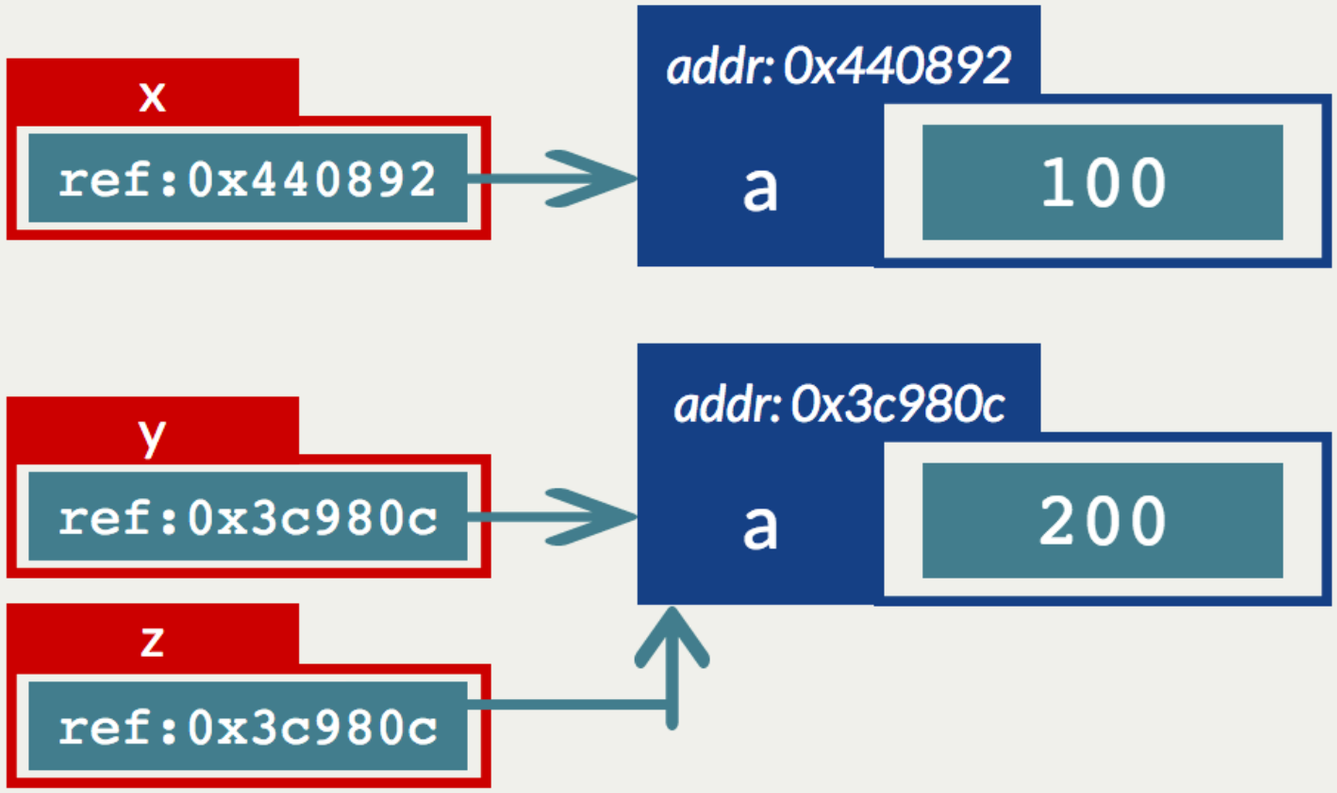


# Don't Forget Any Value Of The Object Type Is Actually A “Reference”

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

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


z.a = 200;
```



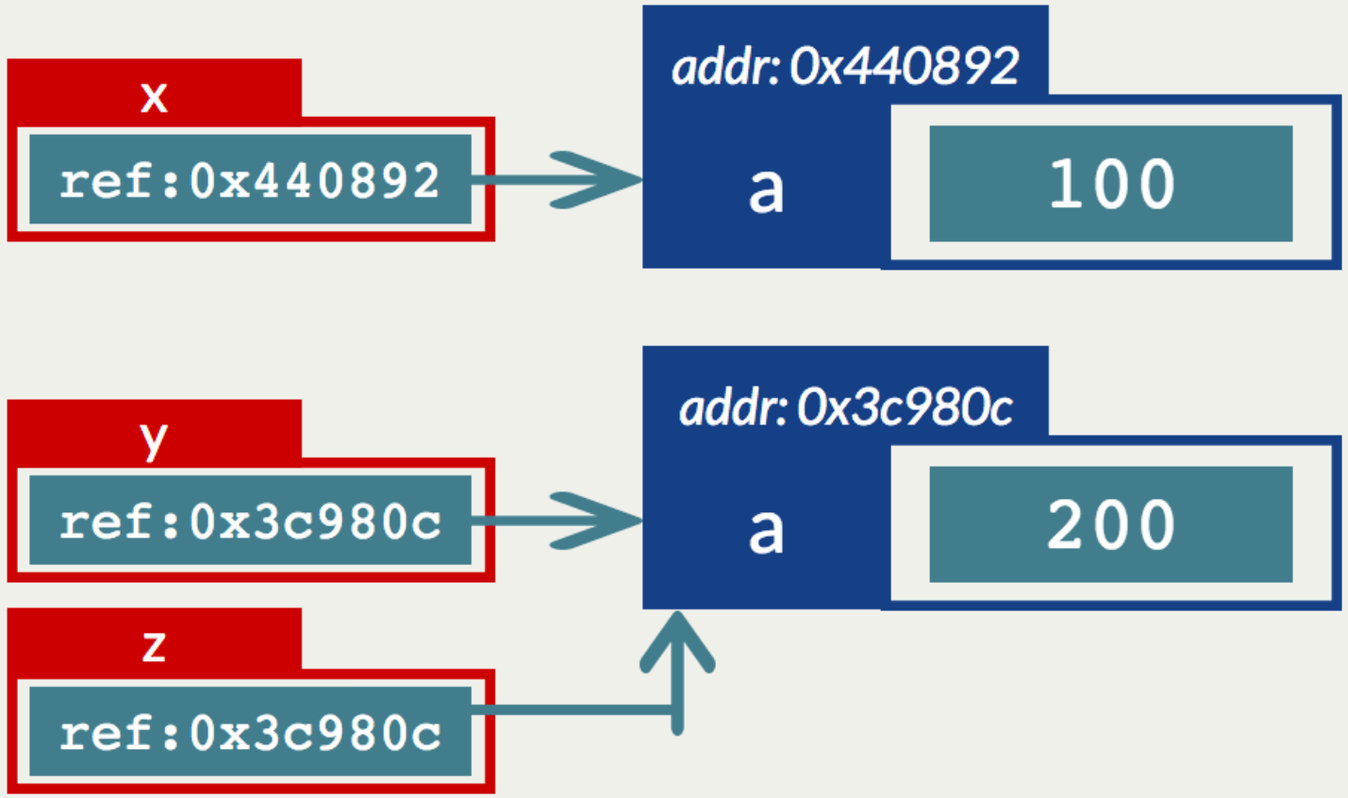
# Don't Forget Any Value Of The Object Type Is Actually A “Reference”

```
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 method is *a* ***function*** as *some object's property*

The property which contains a value that references to some function is called a “method.”

So is the referenced function.

# Methods of An Object

```
// The cat object has three properties
// cat.age, cat.meow, and cat.sleep

var cat = {
    age: 3,
    meow: function () {}
};
cat.sleep = function () {};

// We would say that cat.meow and
// cat.sleep are "methods" of cat
```

# Refer To The Object Inside A Method

When a function is invoked ***as a method*** of some object, the ***this*** value during the function call is *(usually)* bound to that object at ***run-time***

```
var cat = {
    age: 3,
    meow: function () {
        console.log(this.sound);
        return this.age;
    },
    sound: 'meow~~'
};

cat.meow(); // 3  ("meow~~" is printed)

var m = cat.meow;
m(); // TypeError or undefined
```

## Methods

```
var cat = {
  age: 3,
  meow: function () {
    console.log(this.sound);
    return this.age;
  },
  sound: 'meow~~'
};

cat.meow();
```

## Shorthand syntax for Methods

```
var cat = {
  age: 3,
  meow () {
    console.log(this.sound);
    return this.age;
  },
  sound: 'meow~~'
};

cat.meow();
```

# Data Types in Javascript

# Review The Data Types We've Seen So Far

*Undefined* | *Null* | *Boolean* | *Number* | *String* | *Object*

There are exactly ***6 types***
of ***values*** in JavaScript

# Review The Data Types We've Seen So Far

*Undefined* | *Null* | *Boolean* | *Number* | *String* | *Object*

These 2 are pretty boring

# Review The Data Types We've Seen So Far

*Undefined* | *Null* | *Boolean* | *Number* | *String* | *Object*

These 3 are more useful primitives

# Review The Data Types We've Seen So Far

*Undefined* | *Null* | *Boolean* | *Number* | *String* | *Object*

This is the most interesting data type where we can start having ***nested*** and ***organized*** program ***structures***

# “Object” Type Can Be Further *Categorized*

*Undefined* | *Null* | *Boolean* | *Number* | *String* | *Object*

*Object* | *Function* | *Array*

# Some Objects Are Called “Arrays”

# Array Initialiser (Array Literal)

The notation using a pair of square brackets
to ***create/initialize*** a JavaScript ***Array*** object.

```
var w = [
    "test",
    1234,
    {},
    [],
    "hi"
];

w[4]; // "hi"
```

```
var w = new Array(5);
w[0] = "test";
w[1] = 1234;
w[2] = new Object();
w[3] = new Array();
w[4] = "hi";


w[4]; // "hi"
```

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

# Enumerate All Elements In An Array (1/3)

There is a special property “length” for any Array object.

```
var arr = [ "test", 1234, {}, [], "hi" ];

for (var i = 0; i < arr.length; i += 1) {
    console.log(arr[i]);
}
```

NOTE: A “For-loop” is *not* always recommanded for enumerating all elements in an array, because...

# Enumerate All Elements In An Array (2/3)

There is a special method “forEach” for any Array object.

```
var arr = [ "test", 1234, {}, [], "hi" ];

arr.forEach(function (val /*, i, arr*/) {
    console.log(val);
});
// undefined
```

The “*forEach*” method is much nicer...

# Enumerate All Elements In An Array (3/3)

There is a special method “map” for any Array object.

```
var arr = [ "test", 1234, {}, [], "hi" ];

arr.map(function (val /*, i, arr*/) {
    return typeof val;
});
// [ "string",
//   "number",
//   "object",
//   "object",
//   "string" ]
```

We even have functional “*map*”, “*every*”, “*some*”, ... See the notes for more info

# Append New Elements To An Array

There is a method “push” for all Array objects.
Or you can just assign a value to the corresponding slot.

```
var arr = [ "test", 1234, {}, [], "hi" ];

arr.push("sixth"); // 6
arr.length;        // 6
arr[5];            // "sixth"

arr[7] = 012;      // 10
arr.length;        // 8

arr[6];            // undefined
arr[7];            // 10

arr[8];            // undefined
arr.length;        // 8
```