levi nine
Technology Services

THIS & PROTOTYPE

Beograd, 31.10.2019.

AGENDA SLIDE

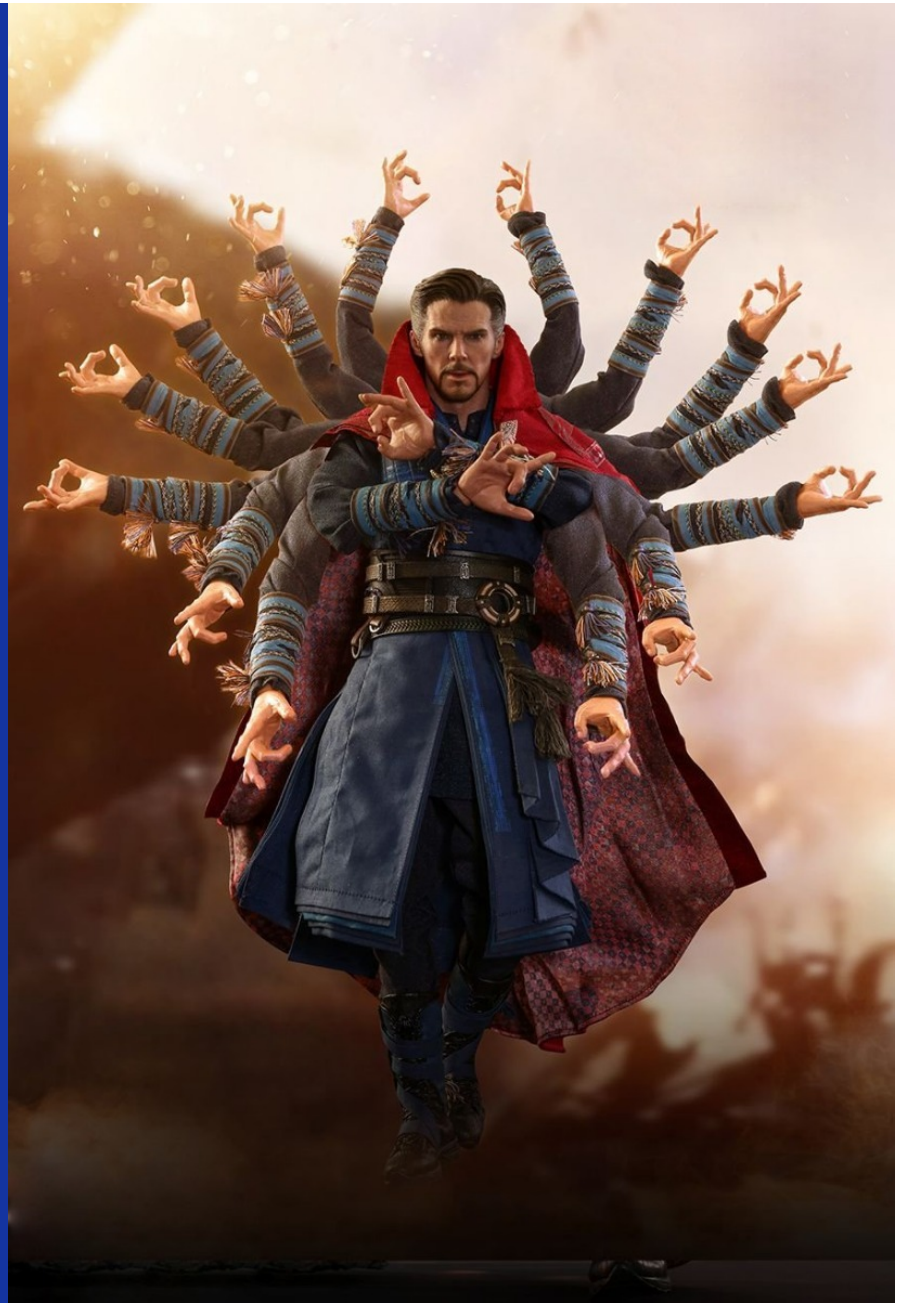
01 This & !this

02 [[Prototypes]]

03 CONSTRUCTORS

01

THIS



!THIS

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times `foo` is called
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}

console.log( foo.count );
```

!THIS

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times `foo` is called
    data.count++;
}

var data = {
    count: 0
};

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}

console.log( data.count ); // 4
```

THIS

- this is a binding that is made when a function is invoked, and what it references is determined entirely by the call-site where the function is called.
- We have to understand the call-site: the location in code where a function is called (not where it's declared)

THIS

```
function baz() {
    // call-stack is: `baz`
    // so, our call-site is in the global scope

    console.log( "baz" );
    bar(); // <-- call-site for `bar`
}

function bar() {
    // call-stack is: `baz` -> `bar`
    // so, our call-site is in `baz`

    console.log( "bar" );
    foo(); // <-- call-site for `foo`
}

function foo() {
    // call-stack is: `baz` -> `bar` -> `foo`
    // so, our call-site is in `bar`

    console.log( "foo" );
}

baz(); // <-- call-site for `baz`
```

THIS

- You must inspect the call-site and determine which of 4 rules applies

1. Default Binding
2. Implicit Binding
3. Explicit Binding
4. Hard Binding

DEFAULT BINDING

- The first rule we will examine comes from the most common case of function calls: standalone function invocation. Think of this this rule as the default catch-all rule when none of the other rules apply.

```
function foo() {
    console.log( this.a );
}

var a = 2;

foo();
```

DEFAULT BINDING

- If strict mode is in effect, the global object is not eligible for the default binding, so the this is instead set to undefined

```
function foo() {
    "use strict";

    console.log( this.a );
}

var a = 2;

foo();
```

IMPLICIT BINDING

- does the call-site have a context object, also referred to as an owning or containing object, though these alternate terms could be slightly misleading.

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo();
```

IMPLICIT BINDING

```
function foo() {
    console.log( this.a );
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo();
```

IMPLICIT BINDING

- Implicitly Lost

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo;

var a = "oops, global";

bar();
```

IMPLICIT BINDING

- Implicitly Lost

```
function foo() {
    console.log( this.a );
}

function doFoo(fn) {

    fn();
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global";

doFoo( obj.foo );
```

EXPLICIT BINDING

- Call or Apply

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

foo.call( obj );
```

EXPLICIT BINDING

- Call or Apply

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = function(param) {
    return foo.apply( obj, arguments );
};

var b = bar( 3 );
console.log( b );
```

HARD BINDING

- Bind

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

var bar = function() {
    foo.call( obj );
};

bar();
setTimeout( bar, 100 );

bar.call( window );
```

HARD BINDING

- Bind

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

// simple `bind` helper
function bind(fn, obj) {
    return function() {
        return fn.apply( obj, arguments );
    };
}

var obj = {
    a: 2
};

var bar = bind( foo, obj );

var b = bar( 3 );
console.log( b );
```

HARD BINDING

- Bind

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = foo.bind( obj );

var b = bar( 3 );
console.log( b );
```

NEW BINDING

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a );
```

NEW BINDING

1. A brand new object is created (aka constructed) out of thin air.
2. The newly constructed object is [[Prototype]]-linked.
3. The newly constructed object is set as the this binding for that function call.
4. Unless the function returns its own alternate object, the new-invoked function call will *automatically* return the newly constructed object.

OBJECTS

- Objects come in two forms: the declarative (literal) form, and the constructed form.

```
var myObj = {
    key: value
    // ...
};
```

```
var myObj = new Object();
myObj.key = value;
```

[[PROTOTYPES]]

- Objects in JavaScript have an internal property, denoted in the specification as [[Prototype]], which is simply a reference to another object

```
var anotherObject = {
    a: 2
};

// create an object linked to `anotherObject`
var myObject = Object.create( anotherObject );

myObject.a; // 2
```

[[PROTOTYPES]]

```
var anotherObject = {
    a: 2
};

// create an object linked to `anotherObject`
var myObject = Object.create( anotherObject );

for (var k in myObject) {
    console.log("found: " + k);
}
// found: a

("a" in myObject); // true
```

SETTING AND SHADOWING PROPERTIES

```
var anotherObject = {
    a: 2
};

var myObject = Object.create( anotherObject );

anotherObject.a; // 2
myObject.a; // 2

anotherObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "a" ); // false

myObject.a++; // oops, implicit shadowing!

anotherObject.a; // 2
myObject.a; // 3

myObject.hasOwnProperty( "a" ); // true
```

“CLASS” FUNCTIONS

- all functions by default get a public, nonenumerable property on them called prototype, which points at anotherwise arbitrary object

```
function Foo() {
    // ...
}

Foo.prototype; // { }
```

“CONSTRUCTORS”

```
function Foo() {
    // ...
}

Foo.prototype.constructor === Foo; // true

var a = new Foo();
a.constructor === Foo; // true
```

“CONSTRUCTORS”

```
function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

var a = new Foo( "a" );
var b = new Foo( "b" );

a.myName(); // "a"
b.myName(); // "b"
```

(PROTOTYPAL) INHERITANCE

```
function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

function Bar(name,label) {
    Foo.call( this, name );
    this.label = label;
}

Bar.prototype = Object.create( Foo.prototype );


Bar.prototype.myLabel = function() {
    return this.label;
};

var a = new Bar( "a", "obj a" );

a.myName(); // "a"
a.myLabel(); // "obj a"
```

LITERATURE

- Eloquent JavaScript: https://eloquentjavascript.net
- You Don't Know JS: https://github.com/getify/You-Dont-Know-JS/tree/1st-ed

THANK YOU

levi nine

Technology Services