This in Javascript

Context for this

Global

When in global context (not inside any function), *this* refers to the global object.

Function

Inside of a function, the value of *this* depends on how the function was called and what mode is being used.

Strict Mode

Strict Mode

- a way to opt in to a more restricted variant of Javascript
- can help prevent programming mistakes in certain circumstances by converting those mistakes to errors
- the 'use strict'; instruction must go at the top of the file or at the top of the function
- not implemented the same way across all Javascript engines

Specifics

- Prevents accidental creation of global variables by throwing reference errors during assignments that would normally create a new global variable
- Assignments that would normally fail silently throw exceptions
- Attempts to delete undeletable properties will throw type errors rather than failing silently
- Prevents accidental use of duplicate parameter names in functions

Terminology

Lexical Environment

Where something sits physically in the code you write.

```
function greet() {
  var myVar = 'Hello';
}

var myOtherVar = 'Goodbye';
```

Execution Context

A wrapper to help manage the code that is running. There are several lexical environments, and which one currently running is managed via execution contexts.

Global

Global Execution Context

The global execution context is handled by the Javascript engine, and two things area created in this execution context: the global object and *this*.

In the browser, *this* is the current browser window because that is where the Javascript is being run. For Node programs, *this* is a different global object because it is not being run in a browser.

Regardless, *this* is always the global object, whether in strict mode or not.

Global Execution Context Demo

```
<html>
  <head>
  </head>
  <body>
    <script src="app.js"></script>
  </body>
</html>
```

Global Execution Context

- An execution context is created at the global level
- The execution context creates a global object which is available to all code running inside that context
- The execution context also creates a special variable called *this*
- At the global level, *this* is equal to the global object
- In a browser, these are both of these are equal to the window in which the Javascript is being run.

Global Execution Context Demo

```
var myVar = 'I am a string';

function myFunc = {
  console.log('I am a function');
};
```

Functions and Objects

Simple Call

Inside a function, the value of *this* will change depending on how the function is called and what method is being used.

For simple function calls, the value of this depends on whether or not strict mode is being used.

Each time a function is called, a new execution context is created for that function. Code inside that function runs in its own execution context.

Execution Context Demo

```
function myFunc() {
  var myVar = 8;
  console.log(myVar);
  myOtherFunc();
}

function myOtherFunc() {
  var myVar;
  console.log(myVar);
}

var myVar = 3;
console.log(myVar);
myFunc();
console.log(myVar);
```

Execution Context Demo

```
function myFunc() {
  var myVar = 8;
  console.log(myVar);
  myOtherFunc();
}

function myOtherFunc() {
  console.log(myVar);
}

var myVar = 3;
console.log(myVar);
myFunc();
console.log(myVar);
```

Simple Call Demo

```
function simpleCall() {
  return this;
}

function strictCall() {
  'use strict';
  return this;
}

console.log('Simple Call This:');
console.log(simpleCall());
console.log('Strict Call This:');
console.log(strictCall());
```

Handling *this* in strict mode

If you want to use *this* inside functions around while using strict mode, you have to use the call, apply or bind functions available on the Function prototype.

The call and apply functions both invoke whichever function they're used on while allowing us to define *this*, but they have slightly different parameter formats.

Bind is bit more complicated and is discussed further in later slides.

Call and Apply Demo

```
'use strict';

var a = 5;
var b = 6;

function add(c, d) {
  return this.a + this.b + c + d;
}

console.log(add.call(this, 3, 4));
console.log(add.apply(this, [7, 8]));
```

Call and Apply Demo

```
'use strict';

var a = 5;
var b = 6;

var myObj = {
  a: 11,
  b: 15
};

function add(c, d) {
  return this.a + this.b + c + d;
}

console.log(add.call(myObj, 3, 4));
console.log(add.apply(myObj, [5, 6]));
```

bind

Bind is another tool available on the Function prototype. Bind creates a new function with the same body as the function its copying, but the new function is permanently bound to the first argument.

Bind Demo

```
'use strict';

function myFunc() {
  return this.myProp;
}

var aFunc = myFunc.bind({myProp:'I am a property'}
console.log(aFunc());
```

Objects and this

When a function is called as a method on an object, *this* refers to the object itself.

This behavior is not affected by how or where the function is defined.

Object Demo

```
'use strict';

function myFunc() {
  return this.myProp;
}

var aFunc = myFunc.bind({myProp:'I am a property'}
console.log(aFunc());
```

Object Demo

```
'use strict';

var myObject = {
  number: 4,
};

function aNumberFunction () {
  return this.number;
}

myObject.getNumber = aNumberFunction;

console.log(myObject.getNumber());
```

Object Demo

```
'use strict';

var person = {
  firstName: 'John',
  lastName: 'Doe',
  getFullName: function () {
    return this.firstName + ' ' + this.lastName;
  }
}

console.log(person.getFullName());
```

Exercise

Using what we've learned, how could we call getFullName, but force its *this* to be a different context than person? There are multiple ways to do this, try to find multiple solutions.

```
'use strict';

var person = {
  firstName: 'John',
  lastName: 'Doe',
  getFullName: function () {
    return this.firstName + ' ' + this.lastName;
  }
}

console.log(person.getFullName());
```

Exercise Solution

```
'use strict';

var person = {
  firstName: 'John',
  lastName: 'Doe',
  getFullName: function () {
    return this.firstName + ' ' + this.lastName;
  }
}

var anotherPerson = {
  firstName: 'Jane',
  lastName: 'Smith'
}

// Most Straightforward
anotherPerson.getFullName = person.getFullName;
console.log(anotherPerson.getFullName());

// Using call on Function prototype
console.log(person.getFullName());
console.log(person.getFullName.call(anotherPerson));

// Using bind on Function prototype
anotherPerson.getFullName = person.getFullName.bind(anotherPerson);
console.log(anotherPerson.getFullName());
```