Chapter 24: Context (this)

Section 24.1: this with simple objects

```
var person = {
  name: 'John Doe',
  age: 42,
  gender: 'male',
  bio: function() {
    console.log('My name is ' + this.name);
  }
};
person.bio(); // logs "My name is John Doe"
var bio = person.bio;
bio(); // logs "My name is undefined"
```

In the above code, `person.bio` makes use of the **context** (`this`). When the function is called as `person.bio()`, the context gets passed automatically, and so it correctly logs "My name is John Doe". When assigning the function to a variable though, it loses its context.

In non-strict mode, the default context is the global object (`window`). In strict mode it is `undefined`.

Section 24.2: Saving this for use in nested functions / objects

One common pitfall is to try and use `this` in a nested function or an object, where the context has been lost.

```
document.getElementById('myAJAXButton').onclick = function(){
    makeAJAXRequest(function(result){
      if (result) { // success
        this.className = 'success';
      }
    })
}
```

Here the context (`this`) is lost in the inner callback function. To correct this, you can save the value of `this` in a variable:

```
document.getElementById('myAJAXButton').onclick = function(){
    var self = this;
    makeAJAXRequest(function(result){
      if (result) { // success
        self.className = 'success';
      }
    })
}
```

Version ≥ 6

ES6 introduced arrow functions which include lexical `this` binding. The above example could be written like this:

```
document.getElementById('myAJAXButton').onclick = function(){
    makeAJAXRequest(result => {
      if (result) { // success
        this.className = 'success';
      }
    })
}
```

Section 24.3: Binding function context

Version ≥ 5.1

Every function has a `bind` method, which will create a wrapped function that will call it with the correct context. See here for more information.

```
var monitor = {
  threshold: 5,
  check: function(value) {
    if (value > this.threshold) {
      this.display("Value is too high!");
    }
  },
  display(message) {
    alert(message);
  }
};

monitor.check(7); // The value of `this` is implied by the method call syntax.


var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so value >
this.threshold is false

var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.

var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

Hard binding

- The object of *hard binding* is to "hard" link a reference to **`this`**.
- Advantage: It's useful when you want to protect particular objects from being lost.
- Example:

```
function Person(){
    console.log("I'm " + this.name);
}

var person0 = {name: "Stackoverflow"}
var person1 = {name: "John"};
var person2 = {name: "Doe"};
var person3 = {name: "Ala Eddine JEBALI"};

var origin = Person;
Person = function(){
    origin.call(person0);
}

Person();
//outputs: I'm Stackoverflow

Person.call(person1);
//outputs: I'm Stackoverflow


Person.apply(person2);
```

```
//outputs: I'm Stackoverflow


Person.call(person3);
//outputs: I'm Stackoverflow
```

- So, as you can remark in the example above, whatever object you pass to *Person*, it'll always use *person0 object*: **it's hard binded**.

Section 24.4: this in constructor functions

When using a function as a constructor, it has a special **`this`** binding, which refers to the newly created object:

```
function Cat(name) {
  this.name = name;
  this.sound = "Meow";
}

var cat = new Cat("Tom"); // is a Cat object
cat.sound; // Returns "Meow"

var cat2 = Cat("Tom"); // is undefined -- function got executed in global context
window.name; // "Tom"
cat2.name; // error! cannot access property of undefined
```