APPMOB - Javascript Closures

Olivier Liechti & Simon Oulevay
COMEM Applications Mobiles

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Global Variables

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

This is a **global variable**.

```
var a = 0;

function add() {
  a = a + 1;
}

add();
add();
add();

console.log(a);
```

All functions have access to global variables.

3

Local Variables

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
function add() {
  var a = 0;
  a = a + 1;
  return a;
}

add();
add();
add();
```

This is a **local variable**. It can only be used in the function where it's declared.

1

Local Variables

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
function test() {
  var a = "foo";

  if (true) {
    var b = "bar";
  }

  return a + b;
}

test();
```

There is no "block" scope in Javascript. Variables are always **local to the entire function** where they're declared.

Both **a** and **b** have the same visibility.

"foobar"

Closures

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

This is a **global variable**. Any function, like **foo**, can use and modify it.

This is a **local variable**. It can only be used inside **foo**.

When you define a function in another function, the inner function (**bar**) has access to variables in the outer function (**foo**). This is what is called a **closure**.

```
var a = 1;

function foo(b) {

  var c = 3;

  function bar(d) {
    return a + b + c + d;
  }

  return bar(4);
}

foo(2);
```

The **bar** function has access to:

- **a**, a global variable;
- **b**, the argument given to foo;
- **c**, the local variable declared in foo;
- **d**, the argument given to bar.

10 (1 + 2 + 3 + 4)

Closures in Factories

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Closures are often used to make **factories**. This function returns a hello function bound to a specific name.

In the factory function, we create a new function. Since it's an inner function, it has access to the name given as argument.

```
function makeHelloFunction(name) {

  return function() {
    return "Hello " + name + "!";
  };
}

var helloWorld = makeHelloFunction("World");
var helloBob = makeHelloFunction("Bob");

helloWorld();
helloBob();
```

"Hello World!"

"Hello Bob!"

Private Variables with Closures

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Closures can be used to make variables **"private"**. Consider this badly implemented counter function which uses a **global variable**.

```
var n = 0;

function count() {
  n = n + 1;
  return n;
}

count();

n = -37;
count();
```

1

-36

Anyone can modify **n**, which is global, and mess up the counter.

Private Variables with Closures

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

If we use a **factory** with a **closure** to create the counter function.

We can use a **local variable** instead of a **global** one.

We create an inner function which has access to the **local variable**.

Now we can get a counter function by calling our factory. The counter variable is **private** and cannot be modified from anywhere else.

```
function makeBetterCount() {
  var value = 0;
  return function() {
    value = value + 1;
    return value;
  };
}

var betterCount = makeBetterCount();

betterCount();
betterCount();
betterCount();
```

1

2

3

Private Functions with Closures

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

This **private** variable can only be used in the service.

This **private** function can only be used in the service.

You can reuse the private function throughout the service.

```
angular.service("StuffService", function() {

  var defaultNumber = 10;

  function getStuff() {
    return [ ... ];
  }

  var service = {
    getAllTheStuff: function() {
      return getStuff();
    },

    getSomeOfTheStuff: function(n) {
      if (n === undefined) {
        n = defaultNumber;
      }

      return getStuff().slice(0, n);
    }
  };

  return service;
})
```

The Infamous Loop Closure "Bug"

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Here is some code which adds markers to a map in an AngularJS application with Mapbox.

```
var issues = ...;

for (var i = 0; i < issues.length; i++) {
  var issue = issues[i];
  $scope.mapMarkers.push({
    lat: issue.lat,
    lng: issue.lng,
    message: "<p>{{ issue.description }}</p>",
    getMessageScope: function() {
      var scope = $scope.$new();
      scope.issue = issue;
      return scope;
    }
  });
}
```

We iterate over a list of issues.

We create a marker for each issue.

Every time you click on a marker, this function will be used to to create the scope for the message template.

Find the bug!

The Infamous Loop Closure "Bug"

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Remember that variables are always **local to a function**. The **issue** variable is not limited to the **for loop**. This code is strictly equivalent to the one on the previous slide.

```
var issues = ...;

var issue;

for (var i = 0; i < issues.length; i++) {
  issue = issues[i];

  $scope.mapMarkers.push({
    lat: issue.lat,
    lng: issue.lng,
    message: "<p>{{ issue.description }}</p>",
    getMessageScope: function() {
      var scope = $scope.$new();
      scope.issue = issue;
      return scope;
    }
  });
}
```

This function is a **closure** which captures a reference to the **issue** variable.

The problem is that the code in **getMessageScope is not executed now**. It will only be executed later, when you click on the marker. **What will be the value of the issue variable then?**

The Solution

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
var issues = ...;

function createMarkerScope(issue) {
  return function() {
    var scope = $scope.$new();
    scope.issue = issue;
    return scope;
  };
}

for (var i = 0; i < issues.length; i++) {
  var issue = issues[i];

  $scope.mapMarkers.push({
    lat: issue.lat,
    lng: issue.lng,
    message: "<p>{{ issue.description }}</p>",
    getMessageScope: createMarkerScope(issue)
  });
}
```

The solution is to create a **new closure** which will capture the correct value of the **issue** variable **during the execution of the for loop**.

We call **createMarkerScope** while iterating. So the **issue argument in createMarkerScope** will have the correct value for each invocation.