Scopes & Closures JavaScript

Scopes

The scope of a variable is controlled by the location of the ***variable declaration****, and defines the part of the program where a particular variable is accessible.*

Scopes

JavaScript has two scopes – global and local.

Any variable declared outside of a function belongs to the global scope, and is therefore accessible from anywhere in your code.

Each function has its own local scope, and any variable declared within that function is only accessible from that function and any nested functions.

Scopes

A global scope and a local scope

```
var x = 5;
function one() {
    var x = 1;
    console.log(x);
}
one();
```

Scopes

A global scope and a local scope

```
var x = 5;
function oneAndAHalf() {
   var x;
    x = 1;
   console.log(x);
}
oneAndAHalf();
```

Scopes

A global variable passed as a parameter

```
var x = 5;
function two(x) {
   console.log(x);
}
two();
```

Scopes

A global variable called within a function

```
var x = 5;
function three() {
    console.log(x);
}
three();
```

Scopes

A global scope

```
var x = 5;
function three() {
     console.log(x);
}

function four() {
     x = 4;
     console.log(x);
}
four();
three();
```

Scopes

An unknown variable

```
function five() {
     var y = 5;
     console.log(y);
}
five();
console.log(y);
```

Closures

A closure wraps up an entire environment, binding necessary variables from other scopes.

```
function testClosure() {
    var x = 4;
     return x;
}
testClosure();
x;
```

This is a local variable

4

undefined

Function's local variables aren't available once the function's scope is closed!!!

OBJ

Closures

A closure wraps up an entire environment, binding necessary variables from other scopes.

The inner function can access the outer function's variables, because they "feel" like global variables.

```
function testClosure() {
    var x = 4;
    function closeX() {
        return x;
    }
    return closeX;
}
```

Notice `x` *does not need to be "stored" anywhere in* `closeX`
We don't even set it as a parameter when we call the function!

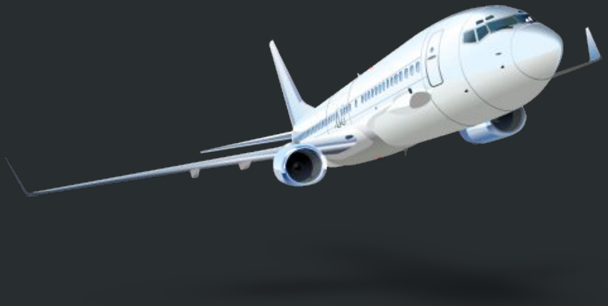
Closures

A closure wraps up an entire environment, binding necessary variables from other scopes.

```
function testClosure() {
    var x = 4;
    function closeX() {
        return x;
    }
    return closeX;
}
var checkLocalX = testClosure();
checkLocalX();          4
```

Even though `testClosure` *has finished operating, its local variable is now bound within* `checkLocalX`.

Closures



A closure can make the creation of very similar functions ultra-efficient.

```
function ticketBuilder(transport) {
        return function(name) {
                console.log("Welcome, " + name + ". Here is your ticket for the " + transport + "!");
        }
}


var getPlaneTicket = ticketBuilder("plane");
var getTrainTicket = ticketBuilder("train");
```

`ticketBuilder` *receives the* `transport` *variable and it is 'closed' in the returned anonymous function where we create the alert.*

Closures

Wait! We are missing something. We have the values for the **`transport`** *variable but, what about the* **`name`** *variable? It is still* `undefined`

```
function ticketBuilder(transport) {
      return function(name) {
            console.log("Welcome, " + name + ". Here is your ticket for the " + transport + "!");
      }
}


var getPlaneTicket = ticketBuilder("plane");
var getTrainTicket = ticketBuilder("train");
```

Closures

BEWARE! Bound variables won't be evident in the stored function.

Passing a `name` to any of our ticket makers will complete our ticket-making process.

Closures

Passing a name *to any of our ticket makers will complete our ticket-making process.*

```
function ticketBuilder(transport) {
      return function(name) {
            console.log("Welcome, " + name + ". Here is your ticket for the " + transport);
      }
}


var getPlaneTicket = ticketBuilder("plane");
var getTrainTicket = ticketBuilder("train");

getPlaneTicket("John Smith");
getPlaneTicket("Patty Bishop");
```

Closures

Closure functions can modify bound variables in the background

Let’s add a passenger tracking for our ticket builder

Closures

Adding a passenger tracking

We will start every ticket maker's tracker at `0` passengers

When a particular ticket maker is called, we know a new passenger should be added, so we'll increase the tracker.

Each time a ticket is "printed," this passengerNumber will contain the precise amount of times this kind of ticket has been given.

```
function ticketBuilder(transport) {
      var passengerNumber = 0;
      return function(name) {
            passengerNumber ++;
            console.log("Welcome, " + name + ". Here is your ticket for the " + transport +
            " You are passenger #" + passengerNumber + "." );
      }
}
var getPlaneTicket = ticketBuilder("plane");
var getTrainTicket = ticketBuilder("train");

getPlaneTicket("John Smith");
getPlaneTicket("Patty Bishop");
```

Closures

Notice that no initial value for `passengerNumber` *is needed.*

It's value starts at `0` *and is adjusted with each call to* `getPlaneTicket`*.*

```
var getPlaneTicket = ticketBuilder("plane");
var getTrainTicket = ticketBuilder("train");

getPlaneTicket("John Smith");
getPlaneTicket("Patty Bishop");
```

Looping with Closures

Our customers bought all of our tickets to Bali. We will like to implement a function to check in a passenger when they arrive to the counter and give us their names

Looping with Closures

```
function checkInPassenger(name, customersArray) {
    var passengerChecked;
    for (var i = 0; i<customersArray.length; i++) {

    }
}
```

We will loop over the array of customers to find name

Looping with Closures

```
function checkInPassenger(name, customersArray) {
    var passengerChecked;
    for (var i = 0; i<customersArray.length; i++) {
        if (customersArray[i] == name) {
            passengerChecked = function() {

            };
        }
    }
}
```

When we find the name *in our array of passengers, we will make a function that will hold our check-in closure*

Looping with Closures

We'll close up the name *variable and the loop counter* i*, and tell the customer which is his passenger number (adjusted for zero).*

```
function checkInPassenger(name, customersArray) {
      var passengerChecked;
      for (var i = 0; i<customersArray.length; i++) {
            if (customersArray[i] == name) {
                  passengerChecked = function() {
                        console.log ("Hi, " + name + "You're passenger #" + (i+1));
                  };
            }
      }
}
```

Looping with Closures

```
function checkInPassenger(name, customersArray) {
      var passengerChecked;
      for (var i = 0; i<customersArray.length; i++) {
            if (customersArray[i] == name) {
                  passengerChecked = function() {
                        console.log ("Hi, " + name + "You’re passenger #" + (i+1));
                  };
            }
      }
      return passengerChecked;
}
```

Finally, we handle the passenger check-in process back to the global scope

Looping with Closures

```
function checkInPassenger(name, customersArray) {
      var passengerChecked;
      for (var i = 0; i<customersArray.length; i++) {
            if (customersArray[i] === name) {
                  passengerChecked = function() {
                        console.log ("Hi, " + name + "You’re passenger #" + (i+1));
                  };
            }
      }
      return passengerChecked;
}
var flightToBali = ["Wayan", "Putu", "Gede", "Ni Luh", "Nyoman"];
var counterCheckIn = checkInPassenger("Gede", flightToBali);
counterCheckIn();
```

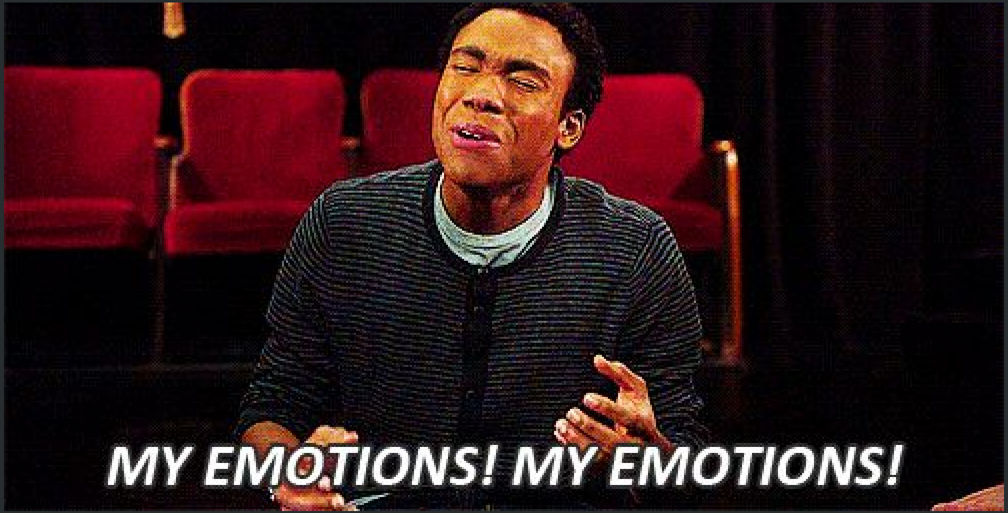
Looping with Closures

Let’s see what happens when we try to check in a passenger. In our example, we are looking for ‘Gede’.

We might have a problem here…

‘Gede’ should be passenger # 3, but the program is returning that the passenger is # 6. There are not 6 passengers in our array!!!

Looping with Closures



Looping with Closures

```
function checkInPassenger(name, customersArray) {
     var passengerChecked;
     for (var i = 0; i<customersArray.length; i++) {
          if (customersArray[i] == name) {
               passengerChecked = function() {
                    console.log ("Hi, " + name + "You’re passenger #" + (i+1));
               };
          }
     }
     return passengerChecked;
}


var flightToBali = ["Wayan", "Putu", "Gede", "Ni Luh", "Nyoman"];
var counterCheckIn = checkInPassenger("Gede", flightToBali);
counterCheckIn();
```

Way before `passengerChecked` *is returned, the* `i` *loop counter has progressed in value to 5 and stopped the loop.*

The function’s actual return is the true “moment of closure,” when the environment and all necessary variables are packaged up.

Looping with Closures

How can we solve this?

Looping with Closures

First Solution: return the anonymous function

Looping with Closures

```
fun
                                                                  " + (i+1));




}
```

Get rid of the `passengerChecked` *and return the function. Then it will return immediately, when it finds the passenger*

```
var flightToBali = ["Wayan", "Putu", "Gede", "Ni Luh", "Nyoman"];
var counterCheckIn = checkInPassenger("Gede", flightToBali);
counterCheckIn();
```

Looping with Closures

```
function checkInPassenger(name, customersArray) {
     for (var i = 0; i<customersArray.length; i++) {
          if (customersArray[i] == name) {
               return function() {
                    console.log ("Hi, " + name + "You’re passenger #" + (i+1));
               }
          }
     }
}


var flightToBali = ["Wayan", "Putu", "Gede", "Ni Luh", "Nyoman"];
var counterCheckIn = checkInPassenger("Gede", flightToBali);
counterCheckIn();
```

Now the function is returned when `checkInPassenger` *finds the passenger and it will return the correct passenger number, locking* `i` *in place*

Looping with Closures

Second Solution: a different design

Looping with Closures

```
function checkInPassenger(name, customersArray) {
      function createPrinting(passenger_id){
            return function() {
                  console.log ("Hi, " + name + " You’re passenger #" + passenger_id);
            }
      }
      var result;
      for (var i = 0; i<customersArray.length; i++) {
            if (customersArray[i] == name) {
                  result = createPrinting(i+1);
            }
      }
      return result;
}


var flightToBali = ["Wayan", "Putu", "Gede", "Ni Luh", "Nyoman"];
var counterCheckIn = checkInPassenger("Gede", flightToBali);
counterCheckIn();
```

Looping with Closures

```
function checkInPassenger(name, customersArray) {
    return function() {
        for (var i = 0; i<customersArray.length; i++) {
        }
    };
}
```

At this point, whatever `passengerArray` *got passed in to* `checkInPassenger` *will be bound into the closure. Parameters are part of the environment, too!*

Since we've put the loop inside the returned function, `i` *will come directly from that local scope.*

```
function checkInPassenger(name, customersArray) {
    return function(name) {
        for (var i = 0; i<customersArray.length; i++) {
            if (customersArray[i] == name) {
                console.log ("Hi, " + name + "You're passenger #" + (i+1));
            }
        }
    };
}
```

The only closed variable from the external scope is `customersArray`, *which never changes.*

Hoisting

In JavaScript, variables and functions are "hoisted."

Hoisting

Rather than being available after their declaration, they might actually be available beforehand...

How does that work? Let's take a look at variable hoisting first.

Hoisting

Open a console in your browser and execute:

```
console.log(noSuchVariable);
```

(Yes, your browser is telling you ReferenceError: noSuchVariable is not defined. We wanted that!)

Hoisting

Now, let’s try this:

```
console.log(declaredLater);

var declaredLater = "Now it's defined!";
```

So, the output is now “undefined”. It exists (is not a Reference Error) but is not initialized

Hoisting

JavaScript treats variables that will be declared later differently than variables that are not declared at all.

Basically, the JavaScript interpreter "looks ahead" to find all the variable declarations and "hoists" them to the top of the function.

Hoisting

Now, let's try this:

```
var declaredLater = "Now it's defined!";

console.log(declaredLater);
```

Now its output is "Now it's defined!". We declared the variable and initialized it with a proper value

Hoisting

What about functions?

```
isItHoisted();

function isItHoisted() {
    console.log("Yes!");
}
```

The output is “Yes!”. Unlike variables, a function declaration doesn't just hoist the function's name. It also hoists the actual function definition

Hoisting

Now, let’s try this:

```
isNotHoisted();

var isNotHoisted = function() {
    console.log("Yes!");
}
```

Oops! It throws us a Type Error. It doesn’t recognize the function.
How can we solve this?

Hoisting

This is what JavaScript is actually interpreting:

```
var isNotHoisted;

isNotHoisted();

isNotHoisted = function() {
    console.log("Yes!");
}
```