# JavaScript Advanced

"Code with Passion!"

# Topics

1. JavaScript functions as first-class objects
2. Self-invoking function
3. Function scope
4. What is Closure? (Closure examples)
5. Why use Closure? (Closure usage examples)
6. Global object, non-global object, and “this”

# Topics

1. JavaScript functions as first-class objects
2. Self-invoking function
3. Function scope
4. What is Closure? (Closure examples)
5. Why use Closure? (Closure usage examples)
6. Global object, non-global object, and “this”

# JavaScript Functions as First-class Objects

# A function is a first-class JavaScript Object (like String or Number object)

- Functions are a bit like Java methods (like in Java)
  - They contain statements for performing some tasks
  - They have arguments and return values
- A function is a first-class object in JavaScript (unlike in Java 7)
  - Can be considered as a descendant of Object object
  - Can do everything a regular JavaScript object can do such as having properties and their values
  - Function objects can have other function objects as methods
- A function behaves like a first-class object (unlike in Java 7)
  1. It can be saved into a variable (like String object)
  2. It can be passed as an argument to another function
  3. It can be returned as a object

# Function Object as First Class Object

- #1: Function object can be assigned to a variable

```
// Define a function – function object gets created
function myMethod(x) {
    console.log("myMethod is invoked with " + x);
}

// Save function object into a variable
var my_function_var = myMethod;

// Invoke the function
my_function_var("Function as a variable");

// Save anonymous function into a variable
var my_function_var2 = function (something){
    console.log("anonymous function is invoked with " + something);
};

// Invoke the function
my_function_var2("Function as a variable");
```

# Function Object as First Class Object

- #2: Function object can be passed as an argument to another function

```
// Define a function – function object gets created
function myFunction(x) {
    console.log("myMethod is invoked with " + x);
}

// Define another function, which takes an argument
function yourMethod(y) {
    y("Function as an argument");
}

// Pass your function object as an argument
yourMethod(myFunction);
```

# Function Object as First Class Object

- #3: Function object can be returned as a return value

```
// Define a function – function object gets created
function myFunction(x) {
    console.log("myFunction is invoked with " + x);
}

// Return function object as a return value
function hisFunction() {
    return myFunction;
}

// Call function, which returns myFunction function
var y = hisFunction();

// Invoke the function
y("Function as a return value");
```

# Lab:

## Exercise 1: Functions as First-class Objects
## 4266_javascript_advanced.zip

# Self-invoking Function

# What is a Self-invoking Function?

- Typically, you define a function and then invoke the function

```
// Declare a function first
function myFunction(something) {
    console.log("Hello " + something);
}

// Then invoke the defined function
myFunction("JPassion"); // Hello JPassion
```

- Self-invoking function lets you define and invoke a function at the same time
  - Self-invoking function is typically anonymous (because you don't need to reference it by name)
  - Sometimes called immediately-invoked function

```
// Self-invoking anonymous function - define and invoke function at the same time
(function (something) {
    console.log("Hello " + something);
})("JPassion");                          // Hello JPassion
```

# Self-Invoking Function Usage example #1

- To avoid global variables conflict
- Problem code:
  - $ is used both in jquery.js and prototype.js
- Code that solves global conflict of $ between jquery.js & prototype.js

```
<script type="text/javascript" src="jquery-1.7.2.js"></script>
<script type="text/javascript" src="prototype.js"></script>
<script type="text/javascript">>

    // Create a plugin - there is no $ namespace conflict with
    // prototype.js since $ is used in private scope here.
    (function($) {
        $.fn.sayGreeting = function() {
            this.prepend("Hello, ");
        };
    })(jQuery);
```

# Self-Invoking Function Usage example #2

- To substitute “setInterval(..)”
- Problem code:
  - If you are in a situation where you want to run a piece of code repeatedly, your first thinking might be using setInterval(..) - The problem is that doSomething() function gets called repeatedly irrespective of whether doSomething() function actually finished doing what it is supposed to do

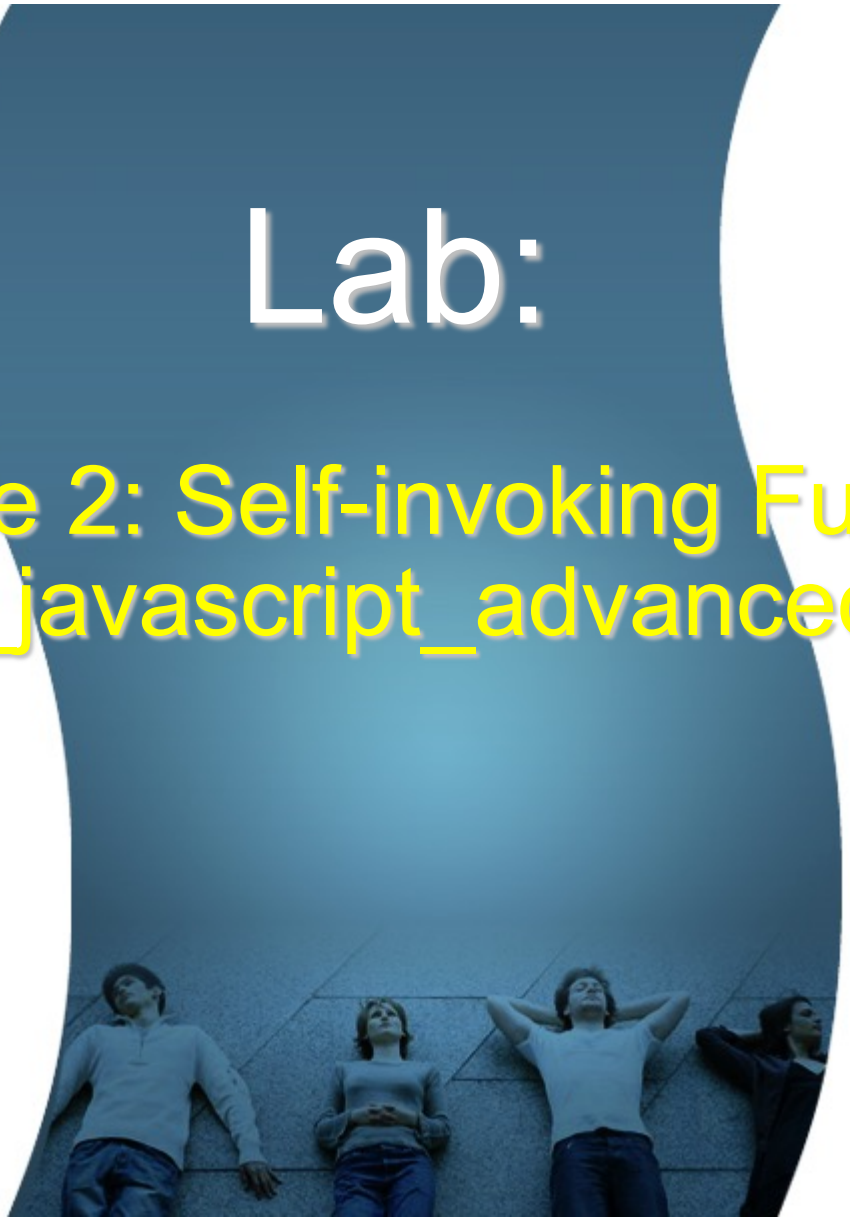
```
setInterval(doSomething, 3000);
```

- Code that uses self-invoking function
  - This code will also repeat itself again and again with one difference. setTimeout will never get triggered unless the task is finished.

```
(function doSomething(){
  // Do some task

  // Wait until the above task is done then schedule the task again in 3 seconds
  setTimeout(doSomething, 3000);
})()
```

# Lab:

Exercise 2: Self-invoking Function
4266_javascript_advanced.zip



# Function Scope

# Function Scope

- Variables defined inside a function cannot be accessed from anywhere outside the function, because they are defined only in the scope of the function
  - They are called “local scope” variables
- However, a function can access all variables and functions defined in the same scope the function is defined
  - A function (inner function) defined inside another function (outer function) can also access all variables defined in it's outer function and any other variable to which the outer function has access
  - A function defined in the global scope can access all variables and functions defined in the global scope

# Lab:

## Exercise 3: Function Scope
## 4266_javascript_advanced.zip

# What is Closure? (Closure Examples)

# What is a Closure?

- A formal description
  - A "closure" is an expression (typically a function) that can have “free variables” together with an environment that binds those variables (that "closes" the expression) - In computer programming, the term “free variable” refers to variables used in a function that are not local variables nor parameters of that function
- An informal description
  - A “closure” gets created when an inner function X is declared/defined (note that is “declared/defined” not “invoked/executed”) in which, when the function gets executed, it is allowed to access variables and other declared inner functions, within its outer (parent) function, in other words, in the same scope the function X is declared

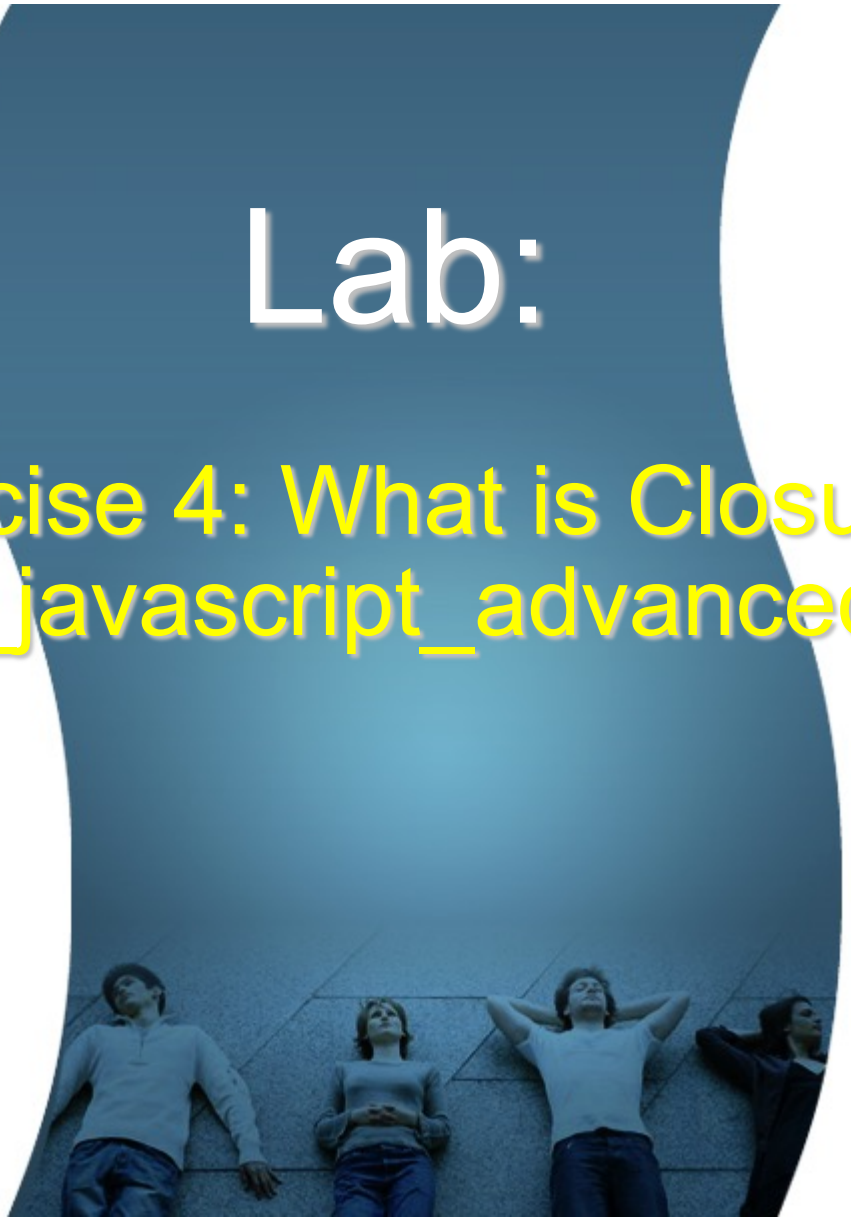
# Closure Example

- When the inner function bar() is declared, which occurs when outer function foo() gets executed, a closure is formed,  in which when the inner function bar() gets executed, it can access variable x that is declared in the same scope of bar()

```
function foo() {
    var x = 10;
    function bar() {
          console.log(x);
    };
    return bar;
}

// "foo" returns inner function
// "bar" and this returned function can
// access variable "x", which is set to 10

var returnedFunction = foo(); // outer function foo() gets executed

// let's define a global variable "x"
var x = 20;

// execution of the returned function
returnedFunction(); // 10, but not 20
```

# Lab:

Exercise 4: What is Closure?
4266_javascript_advanced.zip



# Why Use Closure?
# (Closure Usage Examples)

# Why Use Closure?

- Can reduce the amount and complexity of code
- Can create code that is simply not possible (or too complex) to create without using Closure
- Examples of Closure-enabled code
  - Make variables private
  - Indexing in a loop
  - Timers

# Closure Usage Example #1: Make Variables Private

- JavaScript doesn't have special syntax for private members, but you can make variables private using a closure

```
function Person() {
    // private properties and methods
    var name = 'jPassion';
    var myPrivateGetAgeMethod = function (){
      return 20;
    }

    this.getPersonalInfo = function() {
        return name + " is " + myPrivateGetAgeMethod();
    };
}
var myPerson = new Person();

// 'name' is undefined, it's private
console.log(myPerson.name); // undefined

// 'myPrivateGetAgeMethod' is undefined, since it it's private
//console.log(myPerson.myPrivateGetAgeMethod());

// public method has access to private members
console.log(myPerson.getPersonalInfo()); // "jPassion is 20"
```

**name and myPrivateGetAgeMethod properties cannot be accessed directly**

# Lab:

Exercise 5.1: Why use Closure?
4266_javascript_advanced.zip

# Indexing a loop example #1

- This is a classic problem in JavaScript

```
function addLinksExample1() {
    for ( var i = 0, link; i < 5; i++) {
        link = document.createElement("a");
        link.innerHTML = "LinkWithoutClosure " + i + "<br/>";

        // Indexing a loop without closure
        //
        // Inner anonymous function is defined with
        // the value of variables of outer function
        // when the outer function is executed.
        link.onclick = function() {
            alert(i);
        };
       document.body.appendChild(link);
    }
}
addLinksExample1();
```

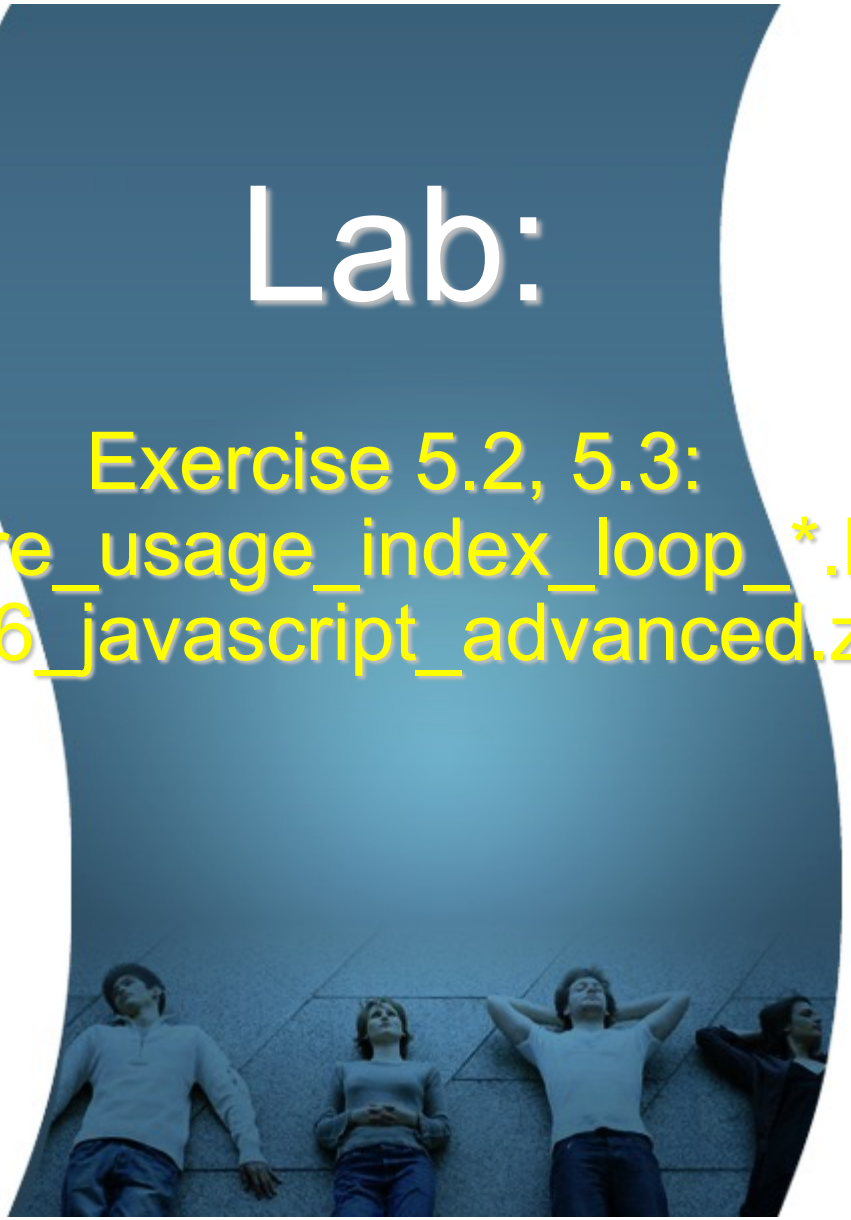
# Example #2: Indexing a loop Example 2

- Use another closure inside a self-invoking function

```
function addLinksExample2() {
    for ( var i = 0, link; i < 5; i++) {
        link = document.createElement("a");
        link.innerHTML = "LinkWithClosure " + i + "<br/>";

        // Indexing a loop with a closure
        //
        // Outer function is self-invoking function.
        // In other words, outer function is defined and invoked.
        // The outer function gets invoked with correct index, which
        // will be the value that inner function takes when it gets executed.
        link.onclick = (function(value) {          // Outer function
            return function() {                    // Inner function
                alert(value);
            }
        })(i);
        document.body.appendChild(link);
    }
}
addLinksExample2();
```

# Lab:

Exercise 5.2, 5.3:
closure_usage_index_loop_*.html
4266_javascript_advanced.zip



# Global Object & Non-global Object & “this”

# Global Object

- All global variables and functions become properties of the global object
  - The global object is the owning object of the global variables and global functions
  - They are in “global” scope
- In browsers, the “window” object is the global object

# What does “this” refer to?

- In JavaScript, “this” refers to the object that a function is a method of
- In global scope, it refers to global object

```
<script type="text/javascript">
    // Global variable, it is a property of global object
    var myGlobalVariable = "John";

    // Global function, it is a property of global object
    function myGlobalFunction() {

    }

    // Display global object, “window” object in browser
    console.dir(this);
</script>
```

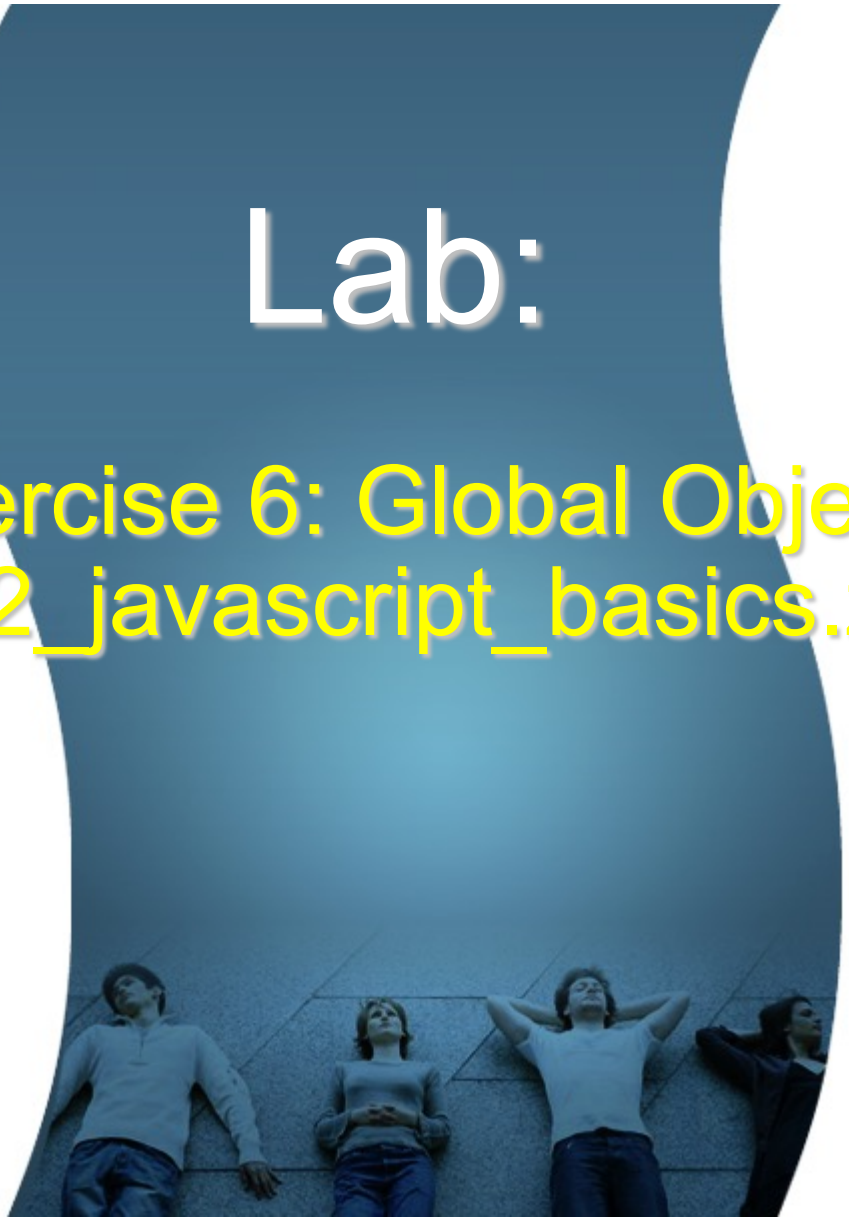
# What does “this” refer to?

- In non-global scope, it refers to non-global object

```
// to personObj variable.
var personObj = {
        firstname : "John",
        lastname : "Doe",
        age : 50,
        tellYourage : function() {
            console.log("The age is " + this.age);
            console.dir(this);  // "this" points to personObj instance
   },
}

personObj.tellYourage();
```

# Lab:

Exercise 6: Global Object
4262_javascript_basics.zip



# Code with Passion!