Closures

Objectives

- Understand closures
- Understand how to use closures in JavaScript

Closures

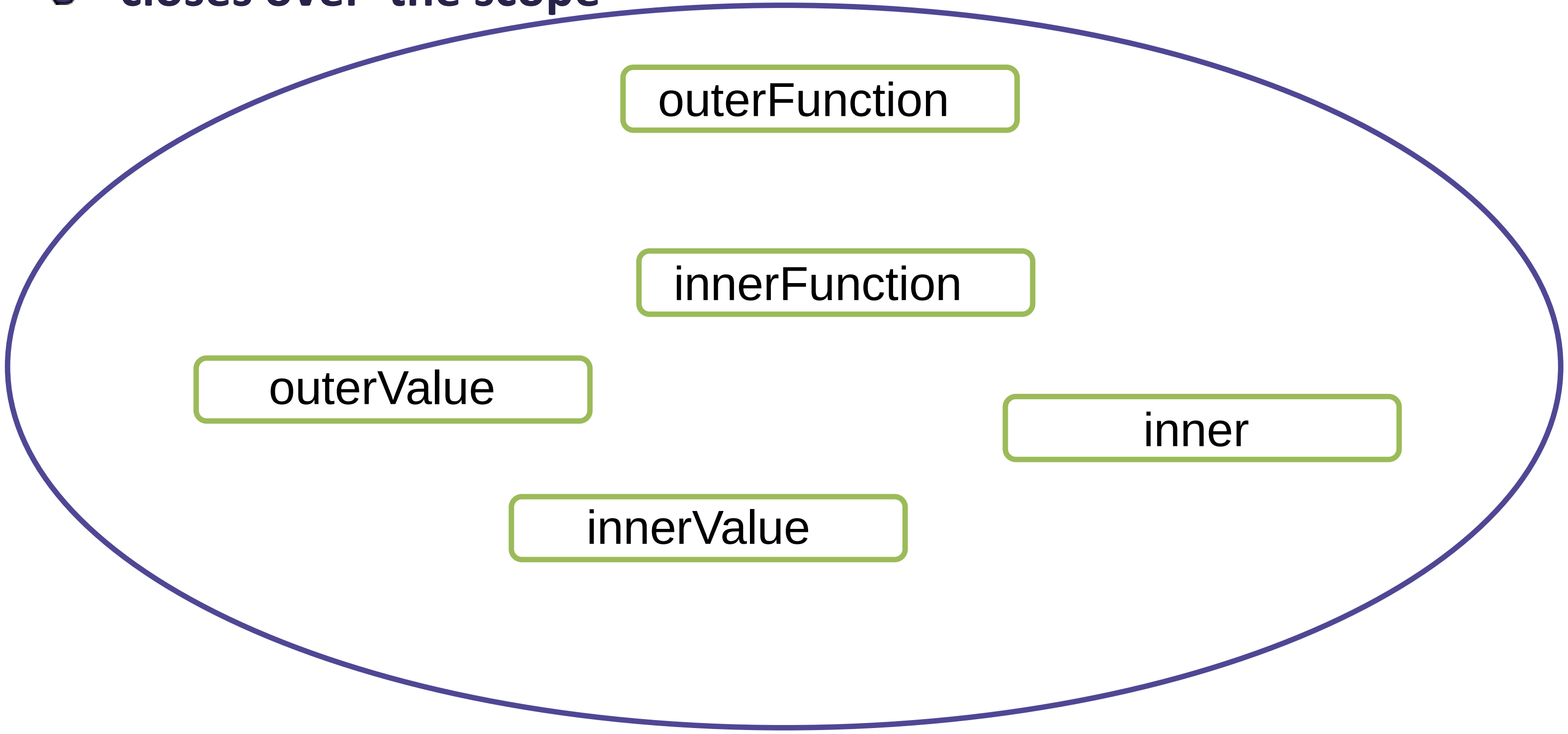
- Closure defines a scope
  - Created when function is declared
  - Allows access to variables defined outside function
  - Variables can still be accessed when function is used
  - Even if their scope has disappeared

Example

```
var outerValue = 'outer';

var inner;

test("closure tests", function () {
        function outerFunction() {
            var innerValue = 'inner';
            assert(outerValue == "outer","ok");

            function innerFunction(){
                assert(innerValue == "inner","ok");
            }
            inner = innerFunction;
        }
        outerFunction();
        inner(); // called but 'outerFunction' has
                 // long gone away
    });
```

How Closures work

- **innerFunction is a closure**
  - **It captures the scope of where it was when it was declared**
  - **'closes over' the scope**



Using closures

- Closures have many uses
  - Private variables
  - Binding 'this'
  - Event handlers

Declaring private variables

- **Can define a constructor**
  - **Variables defined within are 'private' to the constructor**

```
function User(name) {
    var _name = name;

    this.getName = function () {
        return _name;
    };
}

var kevin = new User('kevin');

assert("kevin" == kevin.getName(), "Name is kevin");
```

Callbacks

```
$(function () {
    $.ajaxSetup({'accepts': 'text/JSON'});
    var dataDiv$ = $('#data');
    dataDiv$.html('Loading...');
    $.ajax({
        url: "http://localhost:49578/api/simple",
        success: function (data) {
            var html = $("<span>" + data.firstName +
                             "</span> <span>" +
                               data.lastName + "</span>" )
            dataDiv$.html(html);
        }
    });
})
```

Event handlers can be problematic

- When called the button click method is bound to the element

```
<button id="test">Click Me!</button>

function Button() {
    this.isClicked = false;
    this.click = function () {
        this.isClicked = true;
        alert(button === this);
    };
}

var button = new Button();
var elem = document.getElementById("test");
elem.addEventListener("click", button.click, false);
```

One way to fix

```
function Button() {
    var self = this;
    self.isClicked = false;
    self.click = function () {
        self.isClicked = true;
        alert(button === self);
    };
}
```

Binding contexts (this)

```
function bind(context, name) {
    return function() {
        return context[name].apply(context, arguments);
    };
}

function Button() {
    this.isClicked = false;
    this.click = function () {
        this.isClicked = true;
    };
}

var button = new Button();
$('#clickMe').click(bind(button, "click"));
```

Creating Partial Functions

- Better known as currying
  - Create a function with a predefined set of parameters
  - Apply other parameters to this function

Simple example

- Function that wraps another and stores the arguments
  - Returned function concats its args with stored args …
  - …and calls function

```
function curry(fn) {
    // turn arguments into an array
    var args = Array.prototype.slice.call(arguments, 1);

    return function() {
        return fn.apply(this, args.concat(
        Array.prototype.slice.call(arguments)));
    };
}
```

Using curry

- 'curry' the split function so it splits on ','
  - Create a csv function

```
test("curry tests", function () {
 String.prototype.csv = curry(String.prototype.split, /,\s*/);
    var results = ("Harry, Sam, Alex").csv();
    assert(results[0]=="Harry" &&
        results[1]=="Sam" &&
        results[2]=="Alex",
        "The text values were split properly");
});
```

Another way

- Apply 'curry' method to function prototype

```
Function.prototype.partial = function() {
    var fn = this;
    var args = Array.prototype.slice.call(arguments);
    return function() {
        return fn.apply(this, args.concat(
            Array.prototype.slice.call(arguments)));
    };
}
```

```
test("more closure tests", function () {
   String.prototype.csv
                    = String.prototype.split.partial(/,\s*/);
```

Immediate functions

- IIFE
  - Immediately Invoked Function Expression
  - A way of creating closures
  - Define a function and immediately execute it
  - Use the return value

```
(function(){...})();
```

Scoping

- IIFE used to enforce scope
  - For example, use of $ when using jQuery

```
(function($){
    // do something with jQuery object here
})(jQuery);
```

Module pattern

- Often used as part of the 'revealing module pattern'

Summary

- Closures are used as a scoping tool in JavaScript
    - Enclose over the variables it uses
    - Has many uses
    - Partial functions/currying
    - Data hiding
    - Organisation of code