Closure Design Patterns
The power of functions in JavaScript

Qafoo GmbH

October 25, 2012
What comes next?

Welcome
About Me

Jakob Westhoff

- More than 11 years of professional PHP experience
- More than 8 years of professional JavaScript experience
- Open source enthusiast
- Regular speaker at (inter)national conferences
- Consultant, Trainer and Author

Working with Qafoo
About Me

Jakob Westhoff

▸ More than 11 years of professional PHP experience
▸ More than 8 years of professional JavaScript experience
▸ Open source enthusiast
▸ Regular speaker at (inter)national conferences
▸ Consultant, Trainer and Author

Working with

We help people to create high quality web applications.

Closure Design Patterns 2 / 54
About Me

Jakob Westhoff

- More than 11 years of professional PHP experience
- More than 8 years of professional JavaScript experience
- Open source enthusiast
- Regular speaker at (inter)national conferences
- Consultant, Trainer and Author

Working with

We help people to create high quality web applications.

http://qafoo.com
Goals of this session

- Special role of functions in JavaScript
- The concept of closures
- Utilize those features
Goals of this session

- Special role of functions in JavaScript
- The concept of closures
- Utilize those features
  - Closure/Function Design Patterns
What comes next?

Functions
First level citizens

- Functions are **first level citizens** in JavaScript
  - Can be passed like any other variable
  - Can be created inline
  - Can be defined at any nesting level
  - Can be assigned like any other variable
First level citizens

- Can be passed like any other variable

```javascript
1 function foo(callback) {}
2
3 function bar() {}
4
5 foo(bar);
```
First level citizens

- Can be created inline

```javascript
function foo(callback) {}

foo(function () {
    // .
});
```
First level citizens

- Can be defined at any nesting level

```javascript
function foo () {
    function bar () {
        function baz () {
            // . . .
        }
    }
}
```
First level citizens

- Can be assigned like any other variable

```javascript
function baz(callback) {}  

var foo = function() {}  
var bar = foo;  
baz(bar);  
```
What comes next?

Scope Basics
JavaScript Scoping Basics

- Scoping in JavaScript isn’t trivial
Scoping in JavaScript isn’t trivial
To understand closures only a part of JavaScript’s scoping rules are essential
JavaScript Scoping Basics

- Scoping in JavaScript isn’t trivial
- To understand closures only a part of JavaScripts scoping rules are essential
- Especially Scope Isolation and the Scope Chain
Scope Isolation

- JavaScript does only provide scope isolation on a function level

```javascript
var i = 100;

(function (){
    for (var i = 1; i < 3; ++i)
    {
        alert(i); // 1, 2, 3
    }
})();

alert(i) // 100
```
Scope Isolation

- JavaScript does only provide scope isolation on a function level
- In contrast to block level isolation in other languages (C, C++, Java, ...)

```javascript
var i = 100;

(function () {
    for (var i = 1; i < 3; ++i) {
        alert(i); // 1, 2, 3
    }
})();

alert(i) // 100
```
Scope Isolation

- JavaScript does only provide scope isolation on a function level
- In contrast to block level isolation in other languages (C, C++, Java, ...)

```javascript
var i = 100;

for (var i = 1; i <= 3; ++i) {
    alert(i); // 1, 2, 3
}

alert(i) // 100 or 4?
```
Scope Isolation

- JavaScript does only provide scope isolation on a function level
- In contrast to block level isolation in other languages (C, C++, Java, ...)

```javascript
var i = 100;
for (var i = 1; i <= 3; ++i) {
    alert(i); // 1, 2, 3
}
alert(i) // 4
```
Scope Isolation

- JavaScript does only provide scope isolation on a function level
- In contrast to block level isolation in other languages (C, C++, Java, ...)

```javascript
var i = 100;
for(var i=1; i<=3; ++i) {
  alert(i); // 1, 2, 3
}
alert(i) // 4
```

```javascript
var i = 100;
(function() {
  for(var i=1; i<=3; ++i) {
    alert(i); // 1, 2, 3
  }
})();
alert(i) // 100
```
Scope Chain

- JavaScript Engines chain scopes during their creation
- Inner scopes are always allowed to access outer scopes variables
- Outer scopes can not access inner scopes variables
- Outer scope access is done by reference not by value
Scope Chain

```
1 var a = 42;
```
```javascript
var a = 42;
```
1. `var a = 42;`
**Scope Chain**

```
1 var a = 42;

2 function somefunc () {
3     var b = 23;
4 }
```

![Scope Chain Diagram]

where:
- `a = 42`
- `b = 23`
- `null`
```javascript
var a = 42;

function somefunc () {
    var b = 23;
}
```
```javascript
var a = 42;

function somefunc() {
    var b = 23;

    function otherfunc() {
        var c = "foo";
    }
}
```

Diagram:
- `null` pointing to `a = 42`
- `b = 23` pointing to `null`
- `c = "foo"` pointing to `null`
var a = 42;

function somefunc() {
    var b = 23;

    function otherfunc() {
        var c = "foo";
    }
}

null
null
null
null
b = 23
c = "foo"
Scope Chain

```javascript
var a = 42;

function somefunc() {
    var b = 23;

    function otherfunc() {
        var c = "foo";
        var a = "bar";
    }
}

null
```

```
a = 42  b = 23  c = "foo"
null  a = "bar"
```
Scope Chain

```javascript
var a = 42;

function somefunc () {
  var b = 23;

  function otherfunc () {
    var c = "foo";
    var a = "bar";
    a = "baz";
  }
}
```

null

- `a = 42`
- `b = 23`
- `c = "foo"`
- `a = "baz"`
var a = 42;

function somefunc() {
    var b = 23;

    function otherfunc() {
        var c = "foo";
        var a = "bar";
        a = "baz";
        b = 5;
    }
}

null

a = 42
b = 5
c = "foo"
a = "baz"
```javascript
var a = 42;

function somefunc() {
    var b = 23;

    function otherfunc() {
        var c = "foo";
        a = "baz";
    }
}
```

null → a = "baz" → b = 23 → c = "foo"
What comes next?

Closures
Closures in computer science

- Closures are functions
- They are closed over their free variables
  - Variables from an outside scope can be accessed (*upvalues*)
  - Still accessible if outer scope ceases to exist
- Upvalues are passed by reference not by value
var greeting = "Hello World!";

function showGreetings() {
    alert(greeting);
}

showGreetings();
Closures in JavaScript

JavaScript-Warnmeldung
Hello World!
function createAlertMessage(message) {
    var showMessage = function() {
        alert(message);
    }
    return showMessage;
}
Closures in JavaScript

```javascript
function createAlertMessage(message) {
    var showMessage = function() {
        alert(message);
    }
    return showMessage;
}

var greetTheWorld = createAlertMessage("Hello World!");
greetTheWorld();
```
Closures in JavaScript

JavaScript-Warnmeldung
Hello World!
Closures in JavaScript

```javascript
function createAlertMessage ( message ) {
    var showMessage = function () {
        alert( message );
    }

    return showMessage;
}

var greetTheWorld = createAlertMessage ( "Hello World!" );
var greetTheAudience = createAlertMessage ( "Hello Audience. You are great!" );

greetTheWorld();
greetTheAudience();
```
Closures in JavaScript
Closures in JavaScript

JavaScript-Warnmeldung
Hello Audience. You are great!
The scope chain is created during function declaration
  ▶ Which function may access which scope
Closures in JavaScript - Why?

- The **scope chain** is created during function declaration
  - Which function may access which scope
- A fresh scope is **created** every time a function is **invoked** (activated)
  - Where a function stores its inner variables
Closures in JavaScript - Why?

- The scope chain is created during function declaration
  - Which function may access which scope
- A fresh scope is created every time a function is invoked (activated)
  - Where a function stores its inner variables
- All outer scopes will be kept in memory while at least one inner scope references them.
function createAlertMessage(message) {
    var showMessage = function() {
        alert(message);
    }
    return showMessage;
}

var greetTheWorld = createAlertMessage("Hello World!");
var greetTheAudience = createAlertMessage("Hello Audience. You are great!");
greetTheWorld();
greetTheAudience();
What comes next?

Closure Design Patterns
Closure based design patterns

- As with object orientation certain design patterns can be extracted from working with closures/lamda functions

- Callback Iteration
- Pluggable Behaviour
- Transparent Lazy-Loading
- Function Wrapping
- Composition
- Memoization
- Currying

Be advised, as this are no strict design patterns their names may vary in literature
As with object orientation certain design patterns can be extracted from working with closures/lambda functions

- Callback Iteration
- Pluggable Behaviour
- Transparent Lazy-Loading
- Function Wrapping
- Composition
- Memoization
- Currying
As with object orientation certain design patterns can be extracted from working with closures/lamda functions

- Callback Iteration
- Pluggable Behaviour
- Transparent Lazy-Loading
- Function Wrapping
- Composition
- Memoization
- Currying

Be advised, as this are no strict design patterns their names may vary in literature
What comes next?

Callback Iteration
Callback Iteration

- Callback iteration is a technique to isolate traversal logic from operation logic.
- Its OO counterpart would be the Visitor pattern.
### Callback Iteration - Example

```javascript
var traverseObject = function(object, operation)
{
    var key;
    for (key in object)
    {
        if (object.hasOwnProperty(key))
        {
            operation(object[key], key);
        }
    }
}

traverseObject({one: 1, two: 2, three: 3}, function(value, key)
{ alert(key + "has the value" + value ); });
```
Callback Iteration - Practical use

- Already present in JavaScript (ES5)
  - `Array.forEach`

- Don't stop there. You can use it to iterate complex structures like, trees, jumplists, dual lists, ...

- The visitor pattern is quite useful, but might be overkill in a lot of situations
Callback Iteration - Practical use

- Already present in JavaScript (ES5)
  - Array.forEach
- Available in mostly any framework on objects as well
  - jQuery: jQuery.each
  - ExtJs: Ext.each
  - ...

Don't stop there. You can use it to iterate complex structures like, trees, jumplists, dual lists, ...

The visitor pattern is quite useful, but might be overkill in a lot of situations
Callback Iteration - Practical use

- Already present in JavaScript (ES5)
  - `Array.forEach`
- Available in mostly any framework on objects as well
  - jQuery: `jQuery.each`
  - ExtJs: `Ext.each`
  - ...
- Don’t stop there. You can use it to iterate complex structures like, trees, jumplists, dual lists, ...
Callback Iteration - Practical use

- Already present in JavaScript (ES5)
  - Array.forEach
- Available in mostly any framework on objects as well
  - jQuery: jQuery.each
  - ExtJs: Ext.each
  - ...
- Don’t stop there. You can use it to iterate complex structures like, trees, jumplists, dual lists, ...
- The visitor pattern is quite useful, but might be overkill in a lot of situations
What comes next?

Pluggable Behaviour
Pluggable Behaviour

- Technique to create a generic process, which is configured later on by injecting decision logic

```javascript
var alertFromArray = function(input, decision) {
    var i,
        length = input.length;

    for (i = 0; i < length; i++) {
        if (decision(input[i], i)) {
            alert(input[i]);
        }
    }
}

alertFromArray([1, 2, 3, 4, 5], function(value, index) {
    return value % 2 === 0;
}); // 2, 4
```
Pluggable Behaviour - Practical Use

- Simple replacement for the strategy pattern
- Creation and configuration of filter chains
- Dynamic User-Choice limitation
  - Dropdowns, Options, Checkboxes, ...
What comes next?

Transparent Lazy-Loading
Transparent Lazy-Loading

- Transparent Lazy-Loading is a technique, which allows the lazy initialization of resources and or program code, without the calling context knowing about this.
Imagine a simple Event registration abstraction
  - Modern browsers support the DOM Level 2 Events Model: addEventListener(...)
  - Older Internet Explorer version do not: attachEvent(...)
Imagine a simple Event registration abstraction
  ▶ Modern browsers support the DOM Level 2 Events Model: addEventListener(...)  
    ▶ Older Internet Explorer version do not: attachEvent(...)  

Detecting the featureset of the browser at loading time, combined with defining the proper behaviour increases loading time

Detecting and executing the proper registration everytime an event is registered slows down the application significantly as well
Imagine a simple Event registration abstraction
  ▶ Modern browsers support the DOM Level 2 Events Model: `addEventListener(...)`
  ▶ Older Internet Explorer version do not: `attachEvent(...)`

Detecting the featureset of the browser at loading time, combined with defining the proper behaviour increases loading time

Detecting and executing the proper registration everytime an event is registered slows down the application significantly as well

Detect and define proper behaviour once on the first call of the functionality
var addEventListener = function(target, eventType, handler) {
   // Modern browser
   if (target.addEventListener) {
       addEventListener = function(target, eventType, handler) {
           target.addEventListener(target, eventType, handler);
       }
   }
   // Internet Explorer
   else {
       addEventListener = function(target, eventType, handler) {
           target.attachEvent("on" + eventType, handler);
       }
   }

   // Seemlessly call the selected implementation
   addEventListener(target, eventType, handler);
}
What comes next?

Function Wrapping
Function Wrapping

- Function Wrapping is a technique to wrap the behaviour of one function with another one

```javascript
var doSomething = function() {
  alert("Yeah!");
}

var trackOperation = function(operation) {
  alert('Started operation');
  operation();
  alert('Finished operation');
}

trackOperation(doSomething);
```
A modified version of this technique can for example be used to transparently add profiling and/or timing code to the application.
```javascript
var doSomething = function() {
    alert("Yeah!");
}

var timeOperation = function(operation) {
    return function() {
        alert('Started operation: ' + (new Date()).getTime());
        operation();
        alert('Finished operation: ' + (new Date()).getTime());
    }
}

// Transparent wrapping
doSomething = timeOperation(doSomething);
doSomething(); // Will be timed
```
What comes next?

Composition
Composition

- Composition is a technique to combine the result of a chain of operations
Composition - Example

```javascript
var addOne = function(value) {
    return value + 1;
}

var addTen = function(value) {
    return value + 10;
}

var composition = function(operations, initial) {
    var i,
        lastResult = initial,
        length = operations.length;

    for (i = 0; i < length; i++) {
        lastResult = operations[i](lastResult);
    }

    return lastResult;
}

alert( composition( [addOne, addTen, addOne], 0 ) ); // 12
```
Composition - Practical Use

- Composition is an easy way to create complex dataprocessing routines from simple base elements
- The created composition operation can be reused as a callback or new base operation
```javascript
var addOne = function(value) {...
var addTen = function(value) {...

var composition = function(operations) {
    return function(initial) {
        var i,
            lastResult = initial,
            length = operations.length;
        for (i = 0; i < length; i++) {
            lastResult = operations[i](lastResult);
        }
        return lastResult;
    }
}

var addTwelve = composition([addOne, addTen, addOne]);

alert(addTwelve(3)); // 15
```
What comes next?

Memoization
Memoization

- Memoization is a technique to store partial results of complex calculation in order to speedup further calculations.
- May be used as a caching strategy for calling the same calculation over and over again as well.
Calculating the fibonacci sequence (recursively)

```javascript
var fib = function(i) {
    if (i == 0) {
        return 0;
    }
    if (i == 1) {
        return 1;
    }
    return fib(i - 1) + fib(i - 2);
}
```
Fibonacci sequence

- Slow on consecutive calls
- Intermediate results could be cached
function memoize(fn) {
    return (function() {
        var storage = {};
        var memoizedFn = function(arg) {
            if (storage[arg] === undefined) {
                storage[arg] = fn(arg);
            }
            return storage[arg];
        }
        return memoizedFn;
    })();
}
Memoization - Usage

- Memoization can be dynamically applied to any function

```javascript
var fib = function(i) {...}

var memoize = function(fn) {...}

fib = memoize(fib);
```
What comes next?

Eventual Memoization
Memoization does only work with functions, which are **idempotent**

- Every call to the function with the same arguments yields the **same output**
Eventual Memoization

- Memoization does only work with functions, which are idempotent
  - Every call to the function with the same arguments yields the same output
- What to do if this is not true
Eventual Memoization

- Memoization does only work with functions, which are idempotent
  - Every call to the function with the same arguments yields the same output
- What to do if this is not true
  - A result should be shown to the user as soon as possible.
  - Data does not need to be accurate immediately.
  - Eventually data needs to be accurate.
function eventual ( fn ) {
    return ( function () {
        var storage = {};
        var timeout = null;
        return function ( arg ) {
            if ( timeout !== null ) {
                clearTimeout ( timeout );
            }
            setTimeout ( function () {
                storage [ arg ] = fn ( arg );
            }, 1 );

            return storage [ arg ];
        }
    })();
}
Memoization - Usage

- Eventual Memoization can be dynamically applied to any function

```javascript
var addTimestamp = function(number) {
    var now = new Date();
    return number + now.getTime();
}

addTimestamp = eventual(addTimestamp);

addTimestamp(100); // undefined
addTimestamp(200); // now + 100
addTimestamp(500); // now + 100
addTimestamp(7); // now + 500
...
What comes next?

Currying
Currying

- In theory:
  - Currying is the technique of transforming a function that takes multiple arguments in such a way that it can be called as a chain of functions each with a single argument

- Practical application:
  - Take a general function transforming it into a new function with some of its arguments fixed
Simple Currying - Example

```javascript
var sequential = function (start, end) {
    var i;
    for (i = start; i <= end; i++) {
        alert(i);
    }
}

sequential(0,5); // 0,1,2,3,4,5

var fixSequentialStart = function (fixedStart) {
    ...
}

var sequentialStartAt5 = function fixSequentialStart(5);
sequentialStartAt5(10); // 5,6,7,8,9,10
```
Simple Currying - Example

```javascript
var sequential = function(start, end) {...}

var fixSequentialStart = function(fixedStart) {
    return function(end) {
        return sequential(fixedStart, end);
    }
}

var sequentialStartAt5 = function fixSequentialStart(5);

sequentialStartAt5(10); // 5, 6, 7, 8, 9, 10
```
Real world application

- Create highly customizable operations
- Fix certain aspects of this operations to values for a certain module/area of application in a reusable manner
- Example: A generic XHR loader, which is highly flexible, but configured on an application level
Real world application

- Create highly customizable operations
- Fix certain aspects of this operations to values for a certain module/area of application in a reusable manner
- Example: A generic XHR loader, which is highly flexible, but configured on an application level

For this to work in the real world a generic implementation of the concept is needed
function curry ( fn /*, ..., */ ) {
    var curryArgs = Array.prototype.slice.call( arguments, 1 );

    return function( /* ..., */ ) {
        var newArgs = Array.prototype.slice.call( arguments, 0 ),
            mergedArgs = curryArgs.concat( newArgs );

        return fn.apply( this, mergedArgs );
    }
}
What comes next?

Conclusion
Conclusion

- Not every problem in JavaScript needs an object oriented approach
Conclusion

- Not every problem in JavaScript needs an object oriented approach
- You may use known OO patterns if you want to
Conclusion

- Not every problem in JavaScript needs an object oriented approach
- You may use known OO patterns if you want to
- Think outside the box
Conclusion

- Not every problem in JavaScript needs an object oriented approach
- You may use known OO patterns if you want to
- Think outside the box
- Get inspiration from functional programming languages
Conclusion

- Not every problem in JavaScript needs an object oriented approach
- You may use known OO patterns if you want to
- Think outside the box
- Get inspiration from functional programming languages
- Utilize the power of first level citizen functions
Conclusion

- Not every problem in JavaScript needs an object oriented approach
- You may use known OO patterns if you want to
- Think outside the box
- Get inspiration from functional programming languages
- Utilize the power of first level citizen functions
- Closures rock!
Thanks for listening

Questions, comments or annotations?

Rate this talk: https://joind.in/7381

Slides: http://talks.qafoo.com

Contact: Jakob Westhoff <jakob@qafoo.com>
Follow Me: @jakobwesthoff
Hire us: http://qafoo.com