# 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

# 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.**

# 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

```
1  function foo(callback) {}
2
3  function bar() {}
4
5  foo(bar);
```

# First level citizens

- Can be created inline

```
1  function foo(callback) {}
2
3  foo(function() {
4      // ..
5  });
```

# First level citizens

- Can be defined at any nesting level

```
1  function foo() {
2      function bar() {
3          function baz() {
4              // ...
5          }
6      }
7  }
```

# First level citizens

- Can be assigned like any other variable

```
1  function baz(callback) {}
2
3  var foo = function() {}
4  var bar = foo;
5  baz(bar);
```

# Scope Basics

# JavaScript Scoping Basics

- Scoping in JavaScript isn't trivial

# JavaScript Scoping Basics

- Scoping in JavaScript isn't trivial
- To understand closures only a part of JavaScripts 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

# Scope Isolation

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

# Scope Isolation

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

```
1  var i = 100;
2
3  for(var i=1; i<=3; ++i) {
4      alert(i); // 1, 2, 3
5  }
6
7  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, ...)

```
1  var i = 100;
2
3  for(var i=1; i<=3; ++i) {
4      alert(i); // 1, 2, 3
5  }
6
7  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, ...)

```
1  var i = 100;
2
3  for(var i=1; i<=3; ++i) {
4      alert(i); // 1, 2, 3
5  }
6
7  alert(i) // 4
```

```
1  var i = 100;
2
3  (function() {
4      for(var i=1; i<=3; ++i) {
5          alert(i); // 1, 2, 3
6      }
7  })();
8
9  alert(i) // 100
```

# Scope Chain

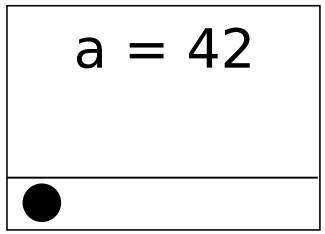
- JavaScript Engines chain scopes during their creation
- Inner scopes are always allowed to access outer scopes variables
- Outer scopes can not access inner scopes variables
- Outer scope access is done by reference not by value

# Scope Chain

```
1  var a = 42;
```

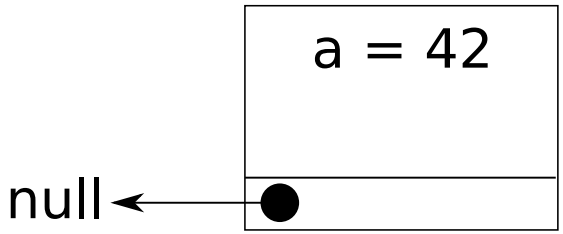
# Scope Chain

```
1  var a = 42;
```



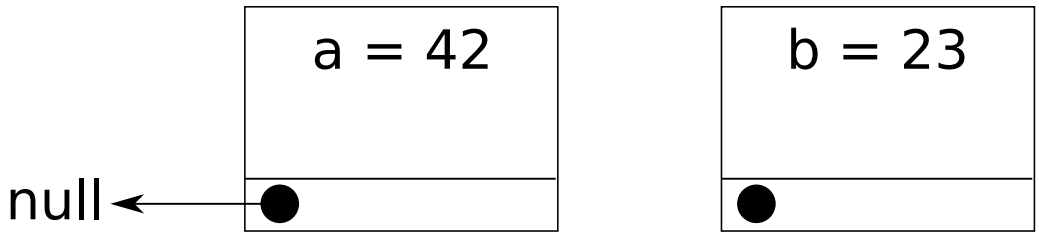
# Scope Chain

```
1  var a = 42;
```



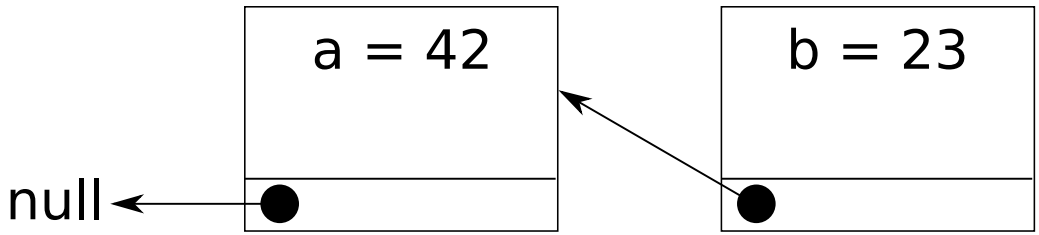
# Scope Chain

```
1  var a = 42;
2
3  function somefunc() {
4      var b = 23;
5  }
```



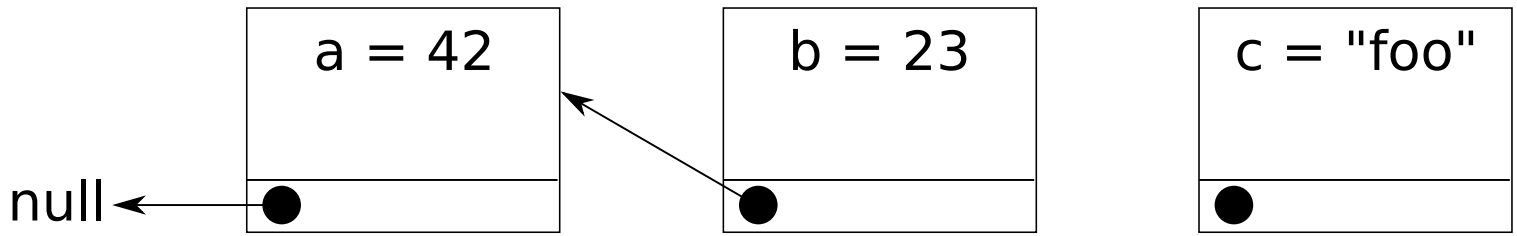
# Scope Chain

```
1  var a = 42;
2
3  function somefunc() {
4      var b = 23;
5  }
```



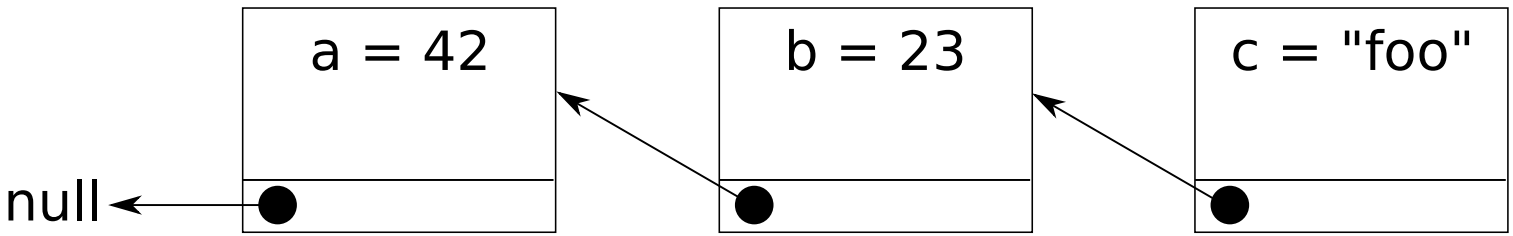
# Scope Chain

```
1  var a = 42;
2
3  function somefunc() {
4      var b = 23;
5
6      function otherfunc() {
7          var c = "foo";
8      }
9  }
```



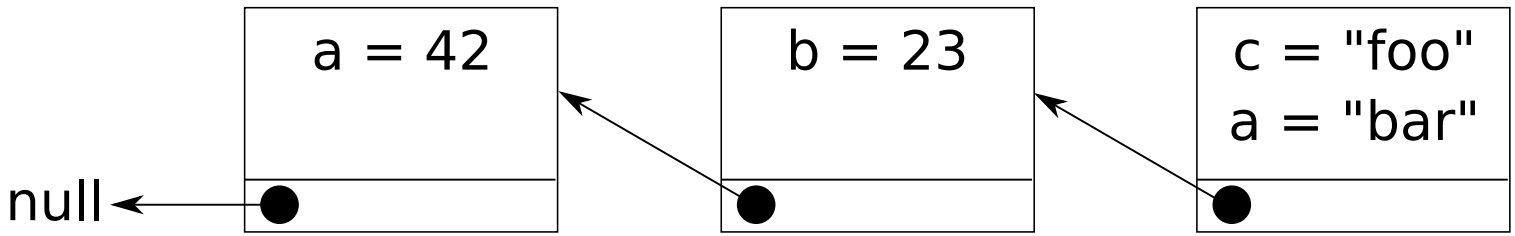
# Scope Chain

```
1  var a = 42;
2
3  function somefunc() {
4      var b = 23;
5
6      function otherfunc() {
7          var c = "foo";
8      }
9  }
```



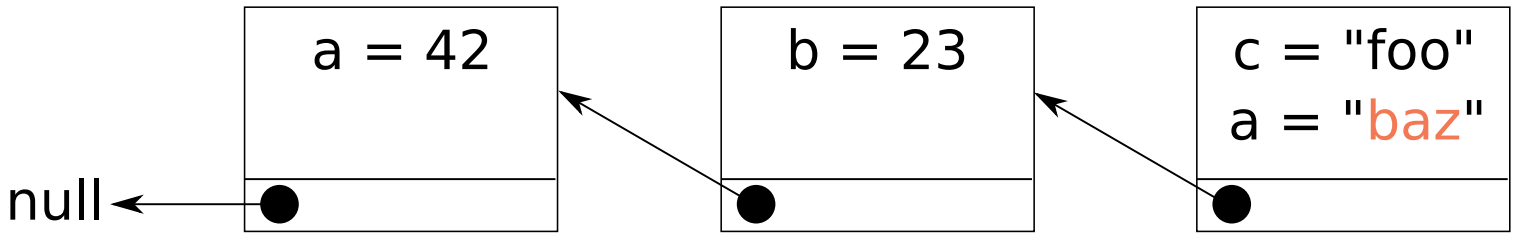
# Scope Chain

```
1  var a = 42;
2
3  function somefunc() {
4      var b = 23;
5
6      function otherfunc() {
7          var c = "foo";
8          var a = "bar";
9      }
10 }
```



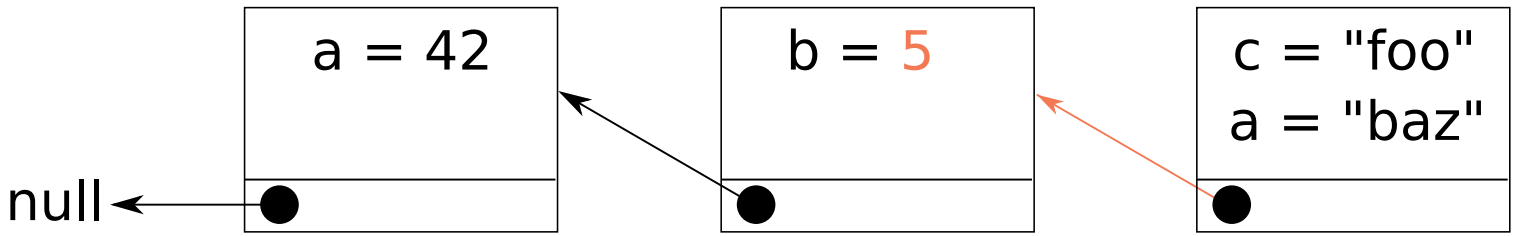
# Scope Chain

```
1  var a = 42;
2
3  function somefunc() {
4      var b = 23;
5
6      function otherfunc() {
7          var c = "foo";
8          var a = "bar";
9          a = "baz";
10     }
11 }
```



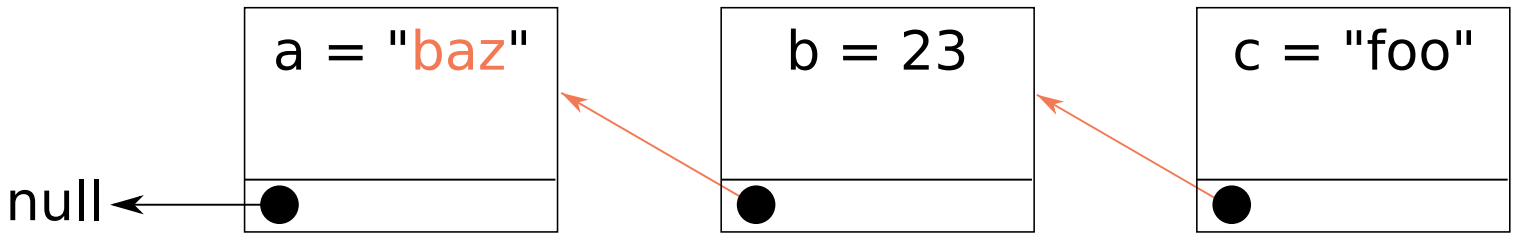
# Scope Chain

```
1  var a = 42;
2
3  function somefunc() {
4      var b = 23;
5
6      function otherfunc() {
7          var c = "foo";
8          var a = "bar";
9          a = "baz";
10         b = 5;
11     }
12 }
```



# Scope Chain

```
1  var a = 42;
2
3  function somefunc() {
4      var b = 23;
5
6      function otherfunc() {
7          var c = "foo";
8          a = "baz";
9      }
10 }
```



# 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

# Closures in JavaScript

```
1  var greeting = "Hello World!";
2
3  function showGreetings() {
4      alert( greeting );
5  }
6
7  showGreetings();
```

# Closures in JavaScript

# Closures in JavaScript

```
1  function createAlertMessage( message ) {
2      var showMessage = function() {
3          alert( message );
4      }
5
6      return showMessage;
7  }
```

# Closures in JavaScript

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

    return showMessage;
}
```

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

greetTheWorld();
```

# Closures in JavaScript

# Closures in 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

# Closures in JavaScript - Why?

- 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.

# Closures in JavaScript - Why?

```
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();
```

# Closure Design Patterns

# Closure based design patterns

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

# 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

# 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

# Callback Iteration

# Callback Iteration

- Callback iteration is a teqnique, to isolate traversal logic from operation logic
- It's OO counterpart would be the Visitor pattern

# Callback Iteration - Example

```
1  var traverseObject = function(object, operation) {
2      var key;
3      for(key in object) {
4          if (object.hasOwnProperty(key)) {
5              operation(object[key], key);
6          }
7      }
8  }
9
10 traverseObject({one: 1, two: 2, three: 3}, function(value, key) {
11     alert( key + " has the value " + value );
12 });
```

# Callback Iteration - Practical use

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

# 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`
  - ...

# 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 usefull, 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

```
1  var alertFromArray = function(input, decision) {
2      var i,
3          length = input.length;
4
5      for(i = 0; i < length; i++) {
6          if (decision(input[i], i)) {
7              alert(input[i]);
8          }
9      }
10 }
11
12 alertFromArray( [1,2,3,4,5], function(value, index) {
13     return value % 2 === 0;
14 }); // 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 programcode, without the calling context knowing about this.

# Transparent Lazy-loading - Example

- Imagine a simple Event registration abstraction
  - Modern browsers support the DOM Level 2 Events Model: `addEventListener(...)`
  - Older Internet Explorer version do not: `attachEvent(...)`

# Transparent Lazy-loading - Example

- 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

# Transparent Lazy-loading - Example

- 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 functionallity

# Transparent Lazy-loading - Example

```
1  var addEventListener = function(target, eventType, handler) {
2      // Modern browser
3      if (target.addEventListener) {
4          addEventListener = function(target, eventType, handler) {
5              target.addEventListener(target, eventType, handler);
6          }
7      }
8      // Internet Explorer
9      else {
10         addEventListener = function(target, eventType, handler) {
11             target.attachEvent("on" + eventType, handler);
12         }
13     }
14
15     // Seemlessly call the selected implementation
16     addEventListener(target, eventType, handler);
17 }
```

# What comes next?

# Function Wrapping

# Function Wrapping

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

```
1  var doSomething = function() {
2      alert("Yeah!");
3  }
4
5  var trackOperation = function(operation) {
6      alert('Started operation');
7      operation();
8      alert('Finished operation');
9  }
10
11 trackOperation(doSomething);
```

# Function Wrapping - Pratical use

- A modified version of this technique can for example be used to transparently add profiling and/or timing code to the application

# Function Wrapping - Pratical use

```
1  var doSomething = function() {
2      alert("Yeah!");
3  }
4
5  var timeOperation = function(operation) {
6      return function() {
7          alert('Started operation:' + (new Date()).getTime());
8          operation();
9          alert('Finished operation:' + (new Date()).getTime());
10     }
11 }
12
13 // Transparent wrapping
14 doSomething = timeOperation(doSomething);
15 doSomething() // Will be timed
```

# What comes next?

Composition

# Composition

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

# Composition - Example

```
1  var addOne = function(value) {
2      return value + 1;
3  }
4  var addTen = function(value) {
5      return value + 10;
6  }
7
8  var composition = function(operations, initial) {
9      var i,
10         lastResult = initial,
11         length = operations.length;
12
13     for(i = 0; i < length; i++) {
14         lastResult = operations[i](lastResult);
15     }
16
17     return lastResult;
18 }
19
20 alert( composition( [addOne, addTen, addOne], 0 ) ); // 12
```

# Composition - Pratical 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

# Composition - Practical Use - Example

```
1  var addOne = function(value) {...}
2  var addTen = function(value) {...}
3
4  var composition = function(operations) {
5      return function(initial) {
6          var i,
7              lastResult = initial,
8              length = operations.length;
9
10         for(i = 0; i < length; i++) {
11             lastResult = operations[i](lastResult);
12         }
13
14         return lastResult;
15     };
16 }
17
18 var addTwelve = composition([addOne, addTen, addOne]);
19
20 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

# Fibonacci sequence

- Calculating the fibonacci sequence (recursively)

```
1  var fib = function(i) {
2      if (i == 0) {
3          return 0;
4      }
5
6      if (i == 1)  {
7          return 1;
8      }
9
10     return
11         fib(i-1) + fib(i-2);
12 }
```

# Fibonacci sequence

- Slow on consecutive calls
- Intermediate results could be cached

# Memoization

```
1  function memoize(fn) {
2      return (function() {
3          var storage = {};
4          var memoizedFn = function(arg) {
5              if ( storage[arg] === undefined ) {
6                  storage[arg] = fn(arg);
7              }
8
9              return storage[arg];
10         }
11
12         return memoizedFn;
13     })();
14 }
```

# Memoization - Usage

- Memoization can be dynamically applied to any function

```
1  var fib = function(i) {...}
2  var memoize = function(fn) {...}
3
4  fib = memoize(fib);
```

# What comes next?

## Eventual Memoization

# 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.

# Eventual Memoization

```
1  function eventual(fn) {
2      return (function() {
3          var storage = {};
4          var timout = null;
5          return function(arg) {
6              if (timeout !== null) { clearTimeout(timeout);
                   }
7              setTimeout(function(){
8                  storage[arg] = fn(arg);
9              }, 1);
10
11             return storage[arg];
12         }
13     })();
14 }
```

# Memoization - Usage

- Eventual Memoization can be dynamically applied to any function

```
1  var addTimestamp = function(number) {
2      var now = new Date();
3      return number + now.getTime();
4  }
5
6  addTimestamp = eventual(addTimestamp);
7
8  addTimestamp(100); // undefined
9  addTimestamp(200); // now + 100
10 addTimestamp(500); // now + 100
11 addTimestamp(7); // now + 500
12 ...
```

# 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
- Pratical application:
  - Take a general function transforming it into a new function with some of its arguments fixed

# Simple Currying - Example

```
1  var sequential = function(start, end) {
2      var i;
3      for(i = start; i <= end; i++) {
4          alert(i);
5      }
6  }
7
8  sequential(0,5); // 0,1,2,3,4,5
9
10 var fixSequentialStart = function(fixedStart) {...}
11
12 var sequentialStartAt5 = function fixSequentialStart(5);
13
14 sequentialStartAt5(10); // 5,6,7,8,9,10
```

# Simple Currying - Example

```
1  var sequential = function(start, end) {...}
2
3  var fixSequentialStart = function(fixedStart) {
4      return function(end) {
5          return sequential(fixedStart, end);
6      }
7  }
8
9  var sequentialStartAt5 = function fixSequentialStart(5);
10
11 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

# Currying for real

```
1  function curry( fn /*, ... */ ) {
2      var curryArgs = Array.prototype.slice.call( arguments, 1 );
3
4      return function( /* ... */ ) {
5          var newArgs = Array.prototype.slice.call( arguments, 0 ),
6              mergedArgs = curryArgs.concat( newArgs );
7
8          return fn.apply( this, mergedArgs );
9      }
10 }
```

# 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