Functional JavaScript

Twitter: @djidja8

JavaScript Overview

- **Type System**
- **Primitive Values**
- **Objects**
- **Inheritance**
- **Functions**
- **Closures**

JavaScript Overview

- Type System

Javascript has dynamic and loosely typed type system with two main types:

Primitive (value) types:

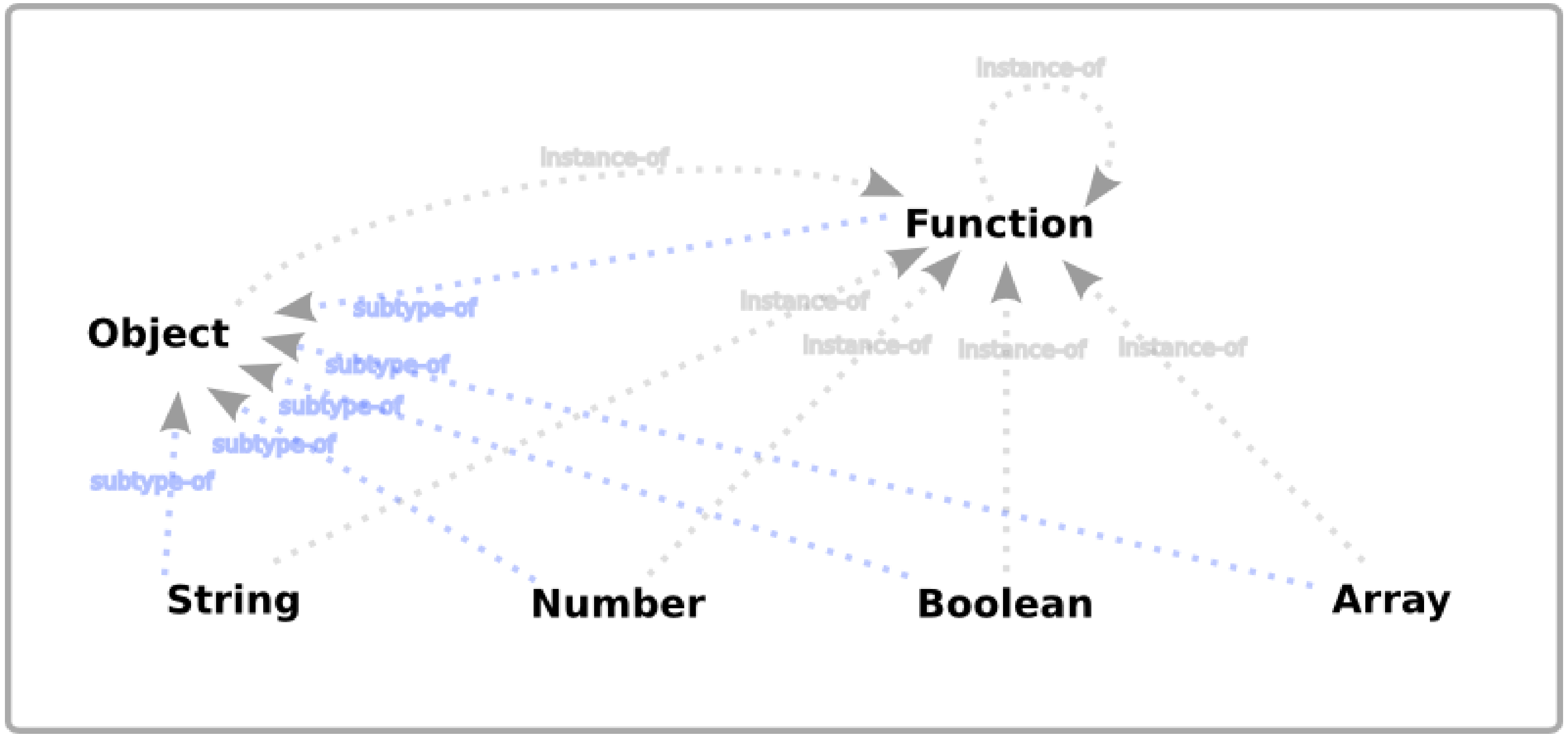
- **number**
- **string**
- **boolean**
- **null**
- **Undefined**

Object (reference) types:

- **Object**
- **Function**
- **Array**
- **Date**
- **RegExp**
- ***Wrappers for primitives: Boolean, Number, String**

JavaScript Overview

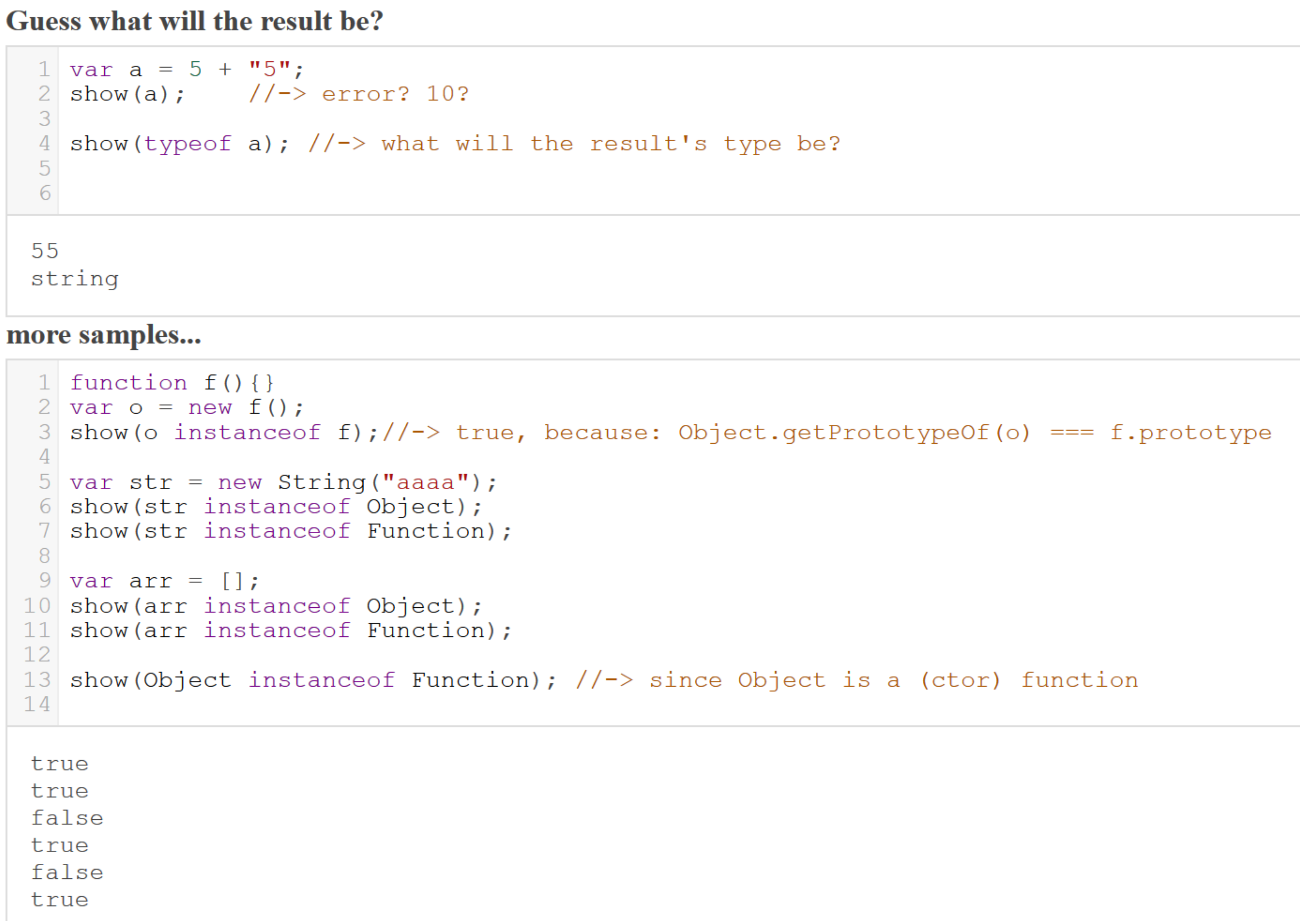
- Type System



@vijayan blog: JavaScript Type Model

JavaScript Overview

- Type System



Guess what will the result be?

```
var a = 5 + "5";
show(a);     //-> error? 10?

show(typeof a); //-> what will the result's type be?


```

```
55
string
```

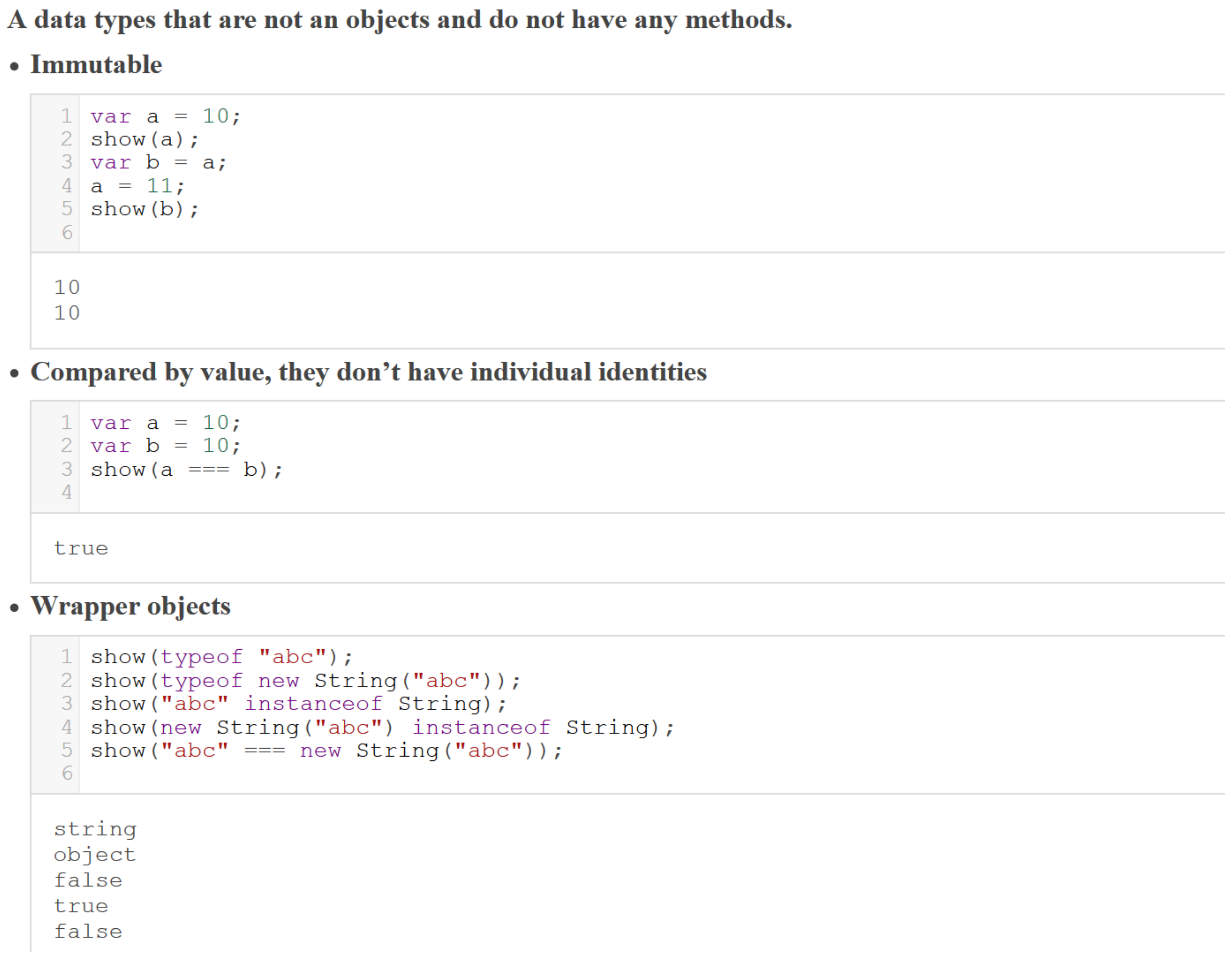
more samples...

```
function f(){}
var o = new f();
show(o instanceof f);//-> true, because: Object.getPrototypeOf(o) === f.prototype

var str = new String("aaaa");
show(str instanceof Object);
show(str instanceof Function);

var arr = [];
show(arr instanceof Object);
show(arr instanceof Function);

show(Object instanceof Function); //-> since Object is a (ctor) function

```

```
true
true
false
true
false
true
```

JavaScript Overview

- Primitive Values



A data types that are not an objects and do not have any methods.

- **Immutable**

```
var a = 10;
show(a);
var b = a;
a = 11;
show(b);

```

```
10
10
```

- **Compared by value, they don’t have individual identities**

```
var a = 10;
var b = 10;
show(a === b);

```

```
true
```

- **Wrapper objects**

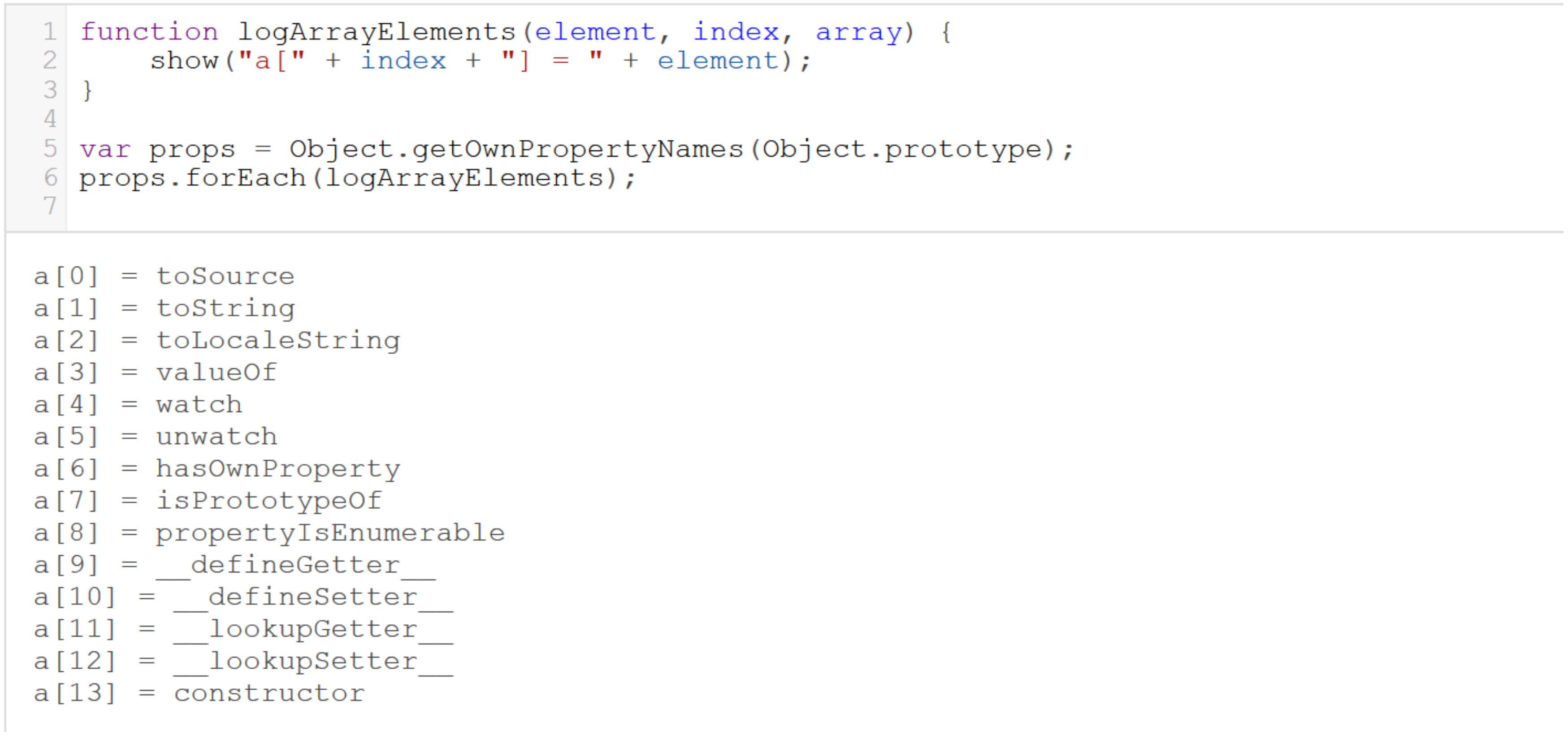
```
show(typeof "abc");
show(typeof new String("abc"));
show("abc" instanceof String);
show(new String("abc") instanceof String);
show("abc" === new String("abc"));

```

```
string
object
false
true
false
```

JavaScript Overview

- Objects

JavaScript objects can be thought of as simple collections of name-value pairs, similar to dictionaries

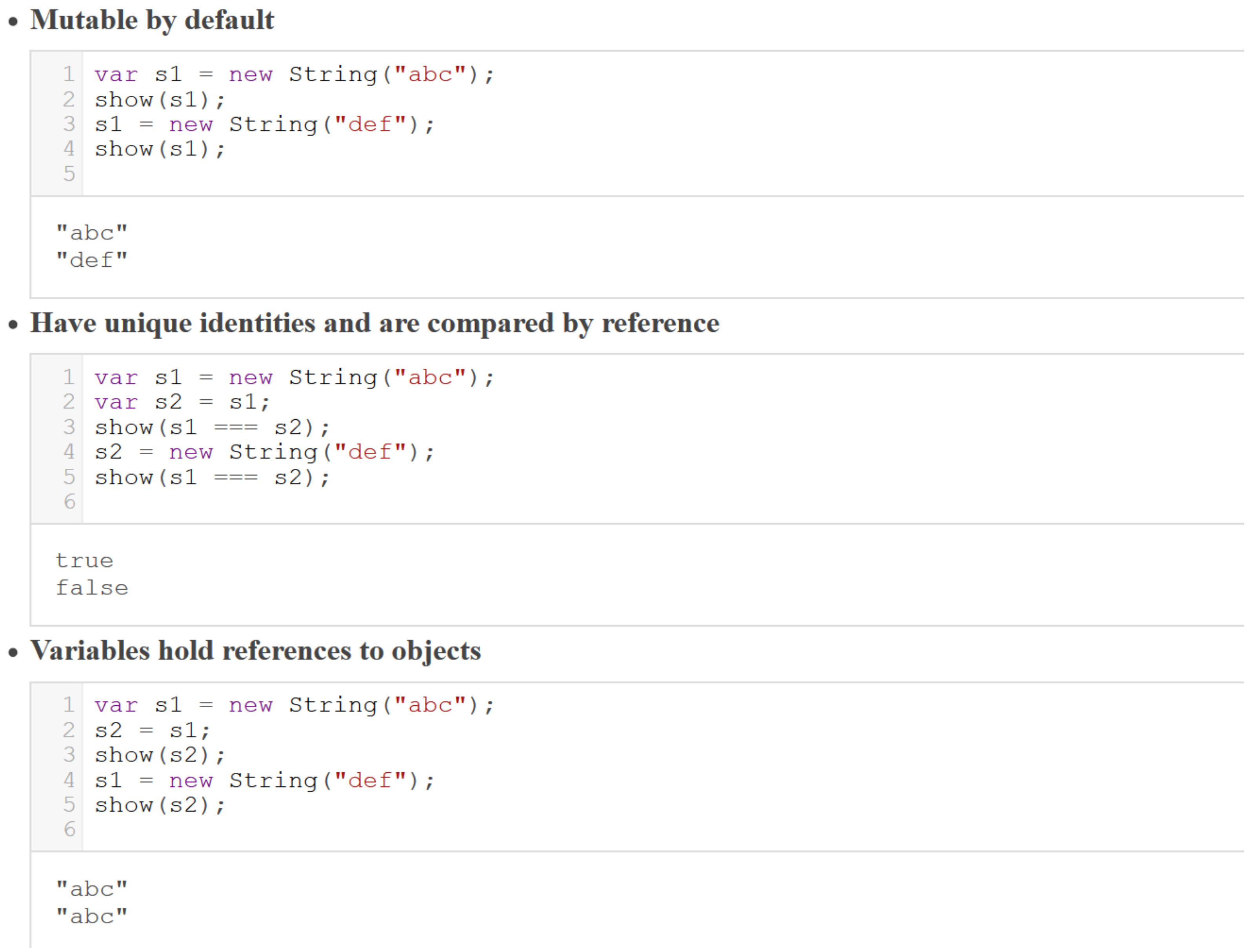


```
function logArrayElements(element, index, array) {
    show("a[" + index + "] = " + element);
}

var props = Object.getOwnPropertyNames(Object.prototype);
props.forEach(logArrayElements);

```

```
a[0] = toSource
a[1] = toString
a[2] = toLocaleString
a[3] = valueOf
a[4] = watch
a[5] = unwatch
a[6] = hasOwnProperty
a[7] = isPrototypeOf
a[8] = propertyIsEnumerable
a[9] = __defineGetter__
a[10] = __defineSetter__
a[11] = __lookupGetter__
a[12] = __lookupSetter__
a[13] = constructor
```

JavaScript Overview

- Objects



- **Mutable by default**

```
var s1 = new String("abc");
show(s1);
s1 = new String("def");
show(s1);

```

```
"abc"
"def"
```

- **Have unique identities and are compared by reference**

```
var s1 = new String("abc");
var s2 = s1;
show(s1 === s2);
s2 = new String("def");
show(s1 === s2);

```

```
true
false
```

- **Variables hold references to objects**

```
var s1 = new String("abc");
s2 = s1;
show(s2);
s1 = new String("def");
show(s2);

```

```
"abc"
"abc"
```

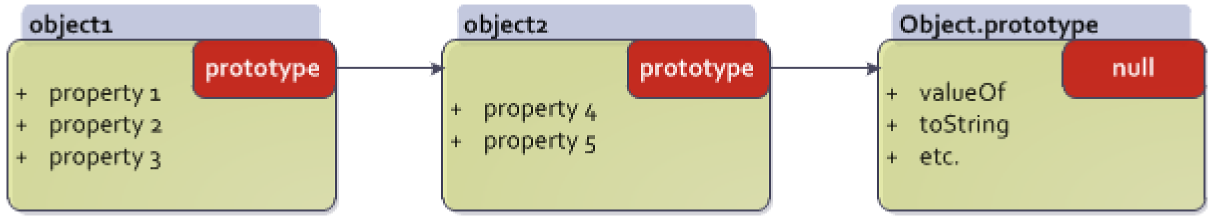
JavaScript Overview

- Inheritance

JavaScript supports a prototype-based inheritance.

Inheritance is performed by creating new objects directly, based on existing ones, existing object is assigned as the prototype of the new object and then inherited behavior can be changed or new behavior can be added to the new object.

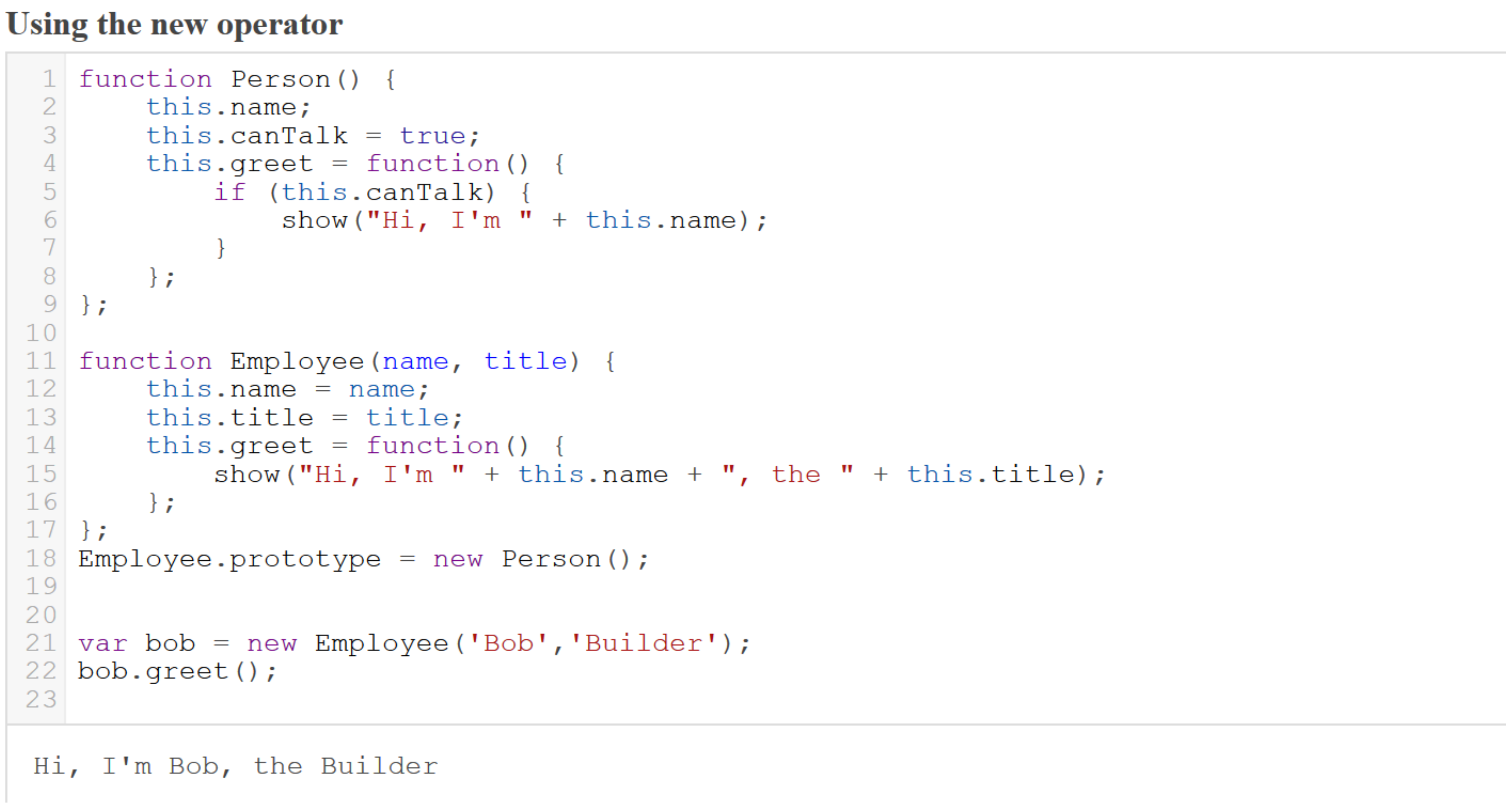
- **The prototype of an object is a simple reference to another object which has its own prototype reference set to yet another object**
- **All objects are descended from 'Object' and they inherit properties from Object.prototype, but they may be overridden**
- **'null', by definition, has no prototype, and acts as the final link in this prototype chain**
- **New objects are created by copying the structure of an existing prototype object**



Inheritance in JavaScript

JavaScript Overview

- Inheritance



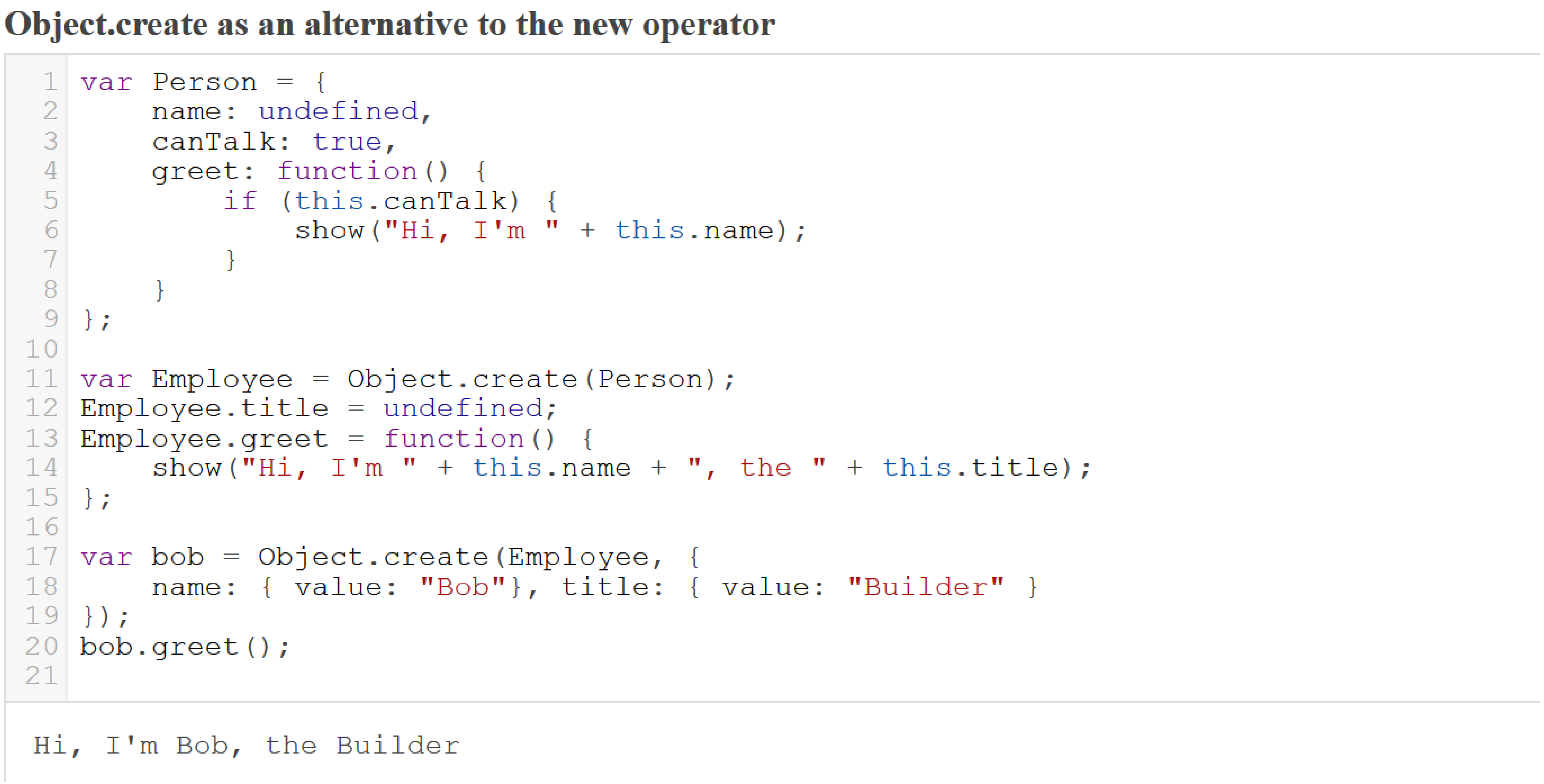
Using the new operator

```javascript
function Person() {
    this.name;
    this.canTalk = true;
    this.greet = function() {
        if (this.canTalk) {
            show("Hi, I'm " + this.name);
        }
    };
};

function Employee(name, title) {
    this.name = name;
    this.title = title;
    this.greet = function() {
        show("Hi, I'm " + this.name + ", the " + this.title);
    };
};
Employee.prototype = new Person();


var bob = new Employee('Bob','Builder');
bob.greet();

```

```
Hi, I'm Bob, the Builder
```

JavaScript Overview

- Inheritance



Object.create as an alternative to the new operator

```javascript
var Person = {
    name: undefined,
    canTalk: true,
    greet: function() {
        if (this.canTalk) {
            show("Hi, I'm " + this.name);
        }
    }
};

var Employee = Object.create(Person);
Employee.title = undefined;
Employee.greet = function() {
    show("Hi, I'm " + this.name + ", the " + this.title);
};

var bob = Object.create(Employee, {
    name: { value: "Bob"}, title: { value: "Builder" }
});
bob.greet();

```

```
Hi, I'm Bob, the Builder
```

JavaScript Overview

- Functions

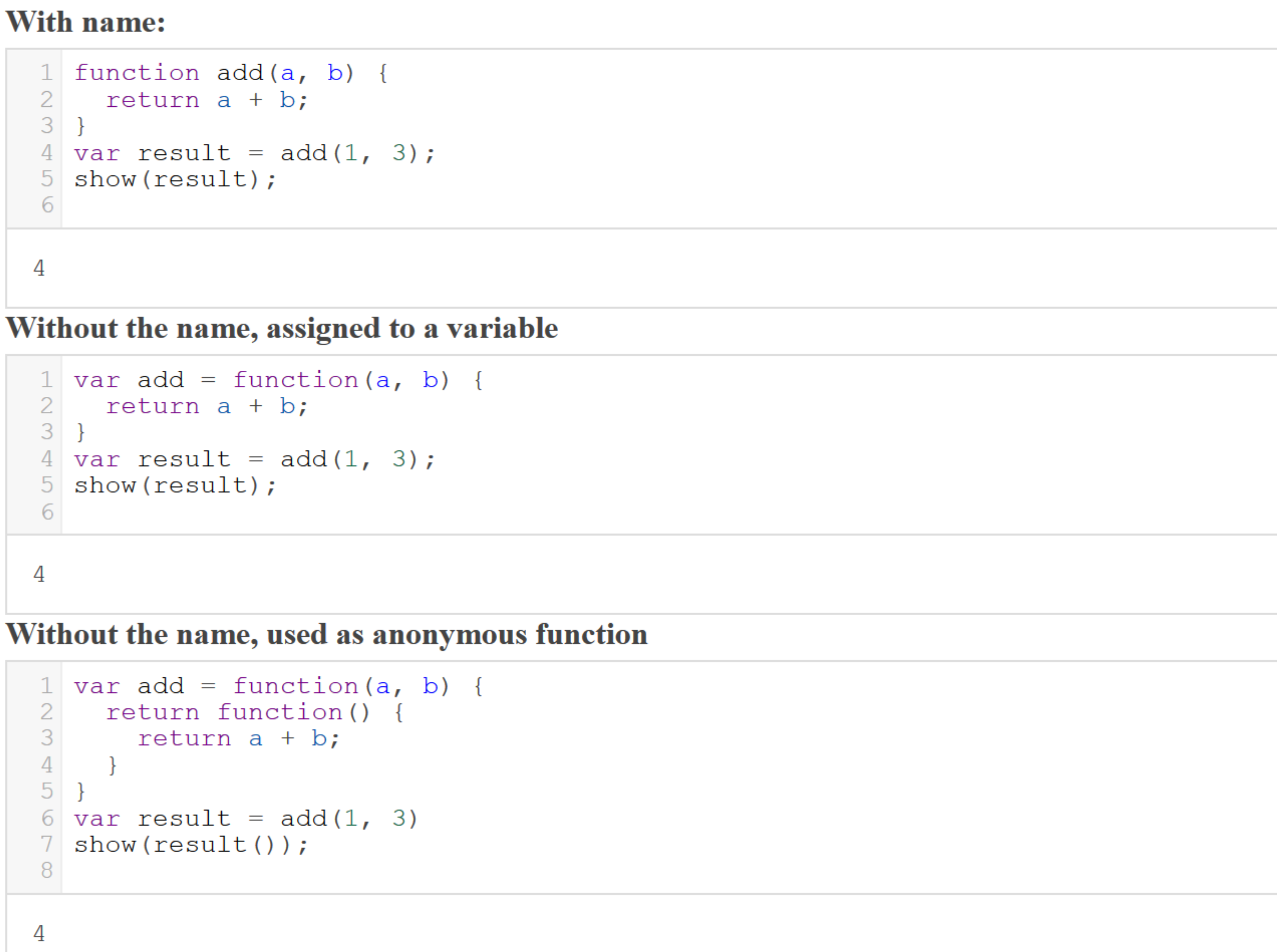
**In JavaScript, functions are objects and are used to perform a task or calculates a value.
Functions have properties and methods that they inherit from the Function object. It is possible to add new properties and methods to functions.**

- **Functions are values that can be bound to names, passed as arguments, returned from other functions**
- **Functions are applied to arguments**
- **The arguments are passed by sharing, which is also called “pass by value”**
- **Function bodies have zero or more expressions**
- **Function application evaluates whatever is returned with the return keyword, or to 'undefined'**
- **Function application creates a scope. Scopes are nested and free variable references are closed over**
- **Variables can shadow variables in an enclosing scope**

JavaScript Overview

- Functions

Function declarations (they are hoisted — moved in their entirety to the beginning of the current scope)



With name:

```
function add(a, b) {
  return a + b;
}
var result = add(1, 3);
show(result);

```

```
4
```

Without the name, assigned to a variable

```
var add = function(a, b) {
  return a + b;
}
var result = add(1, 3);
show(result);

```

```
4
```

Without the name, used as anonymous function

```
var add = function(a, b) {
  return function() {
    return a + b;
  }
}
var result = add(1, 3)
show(result());

```

```
4
```

JavaScript Overview

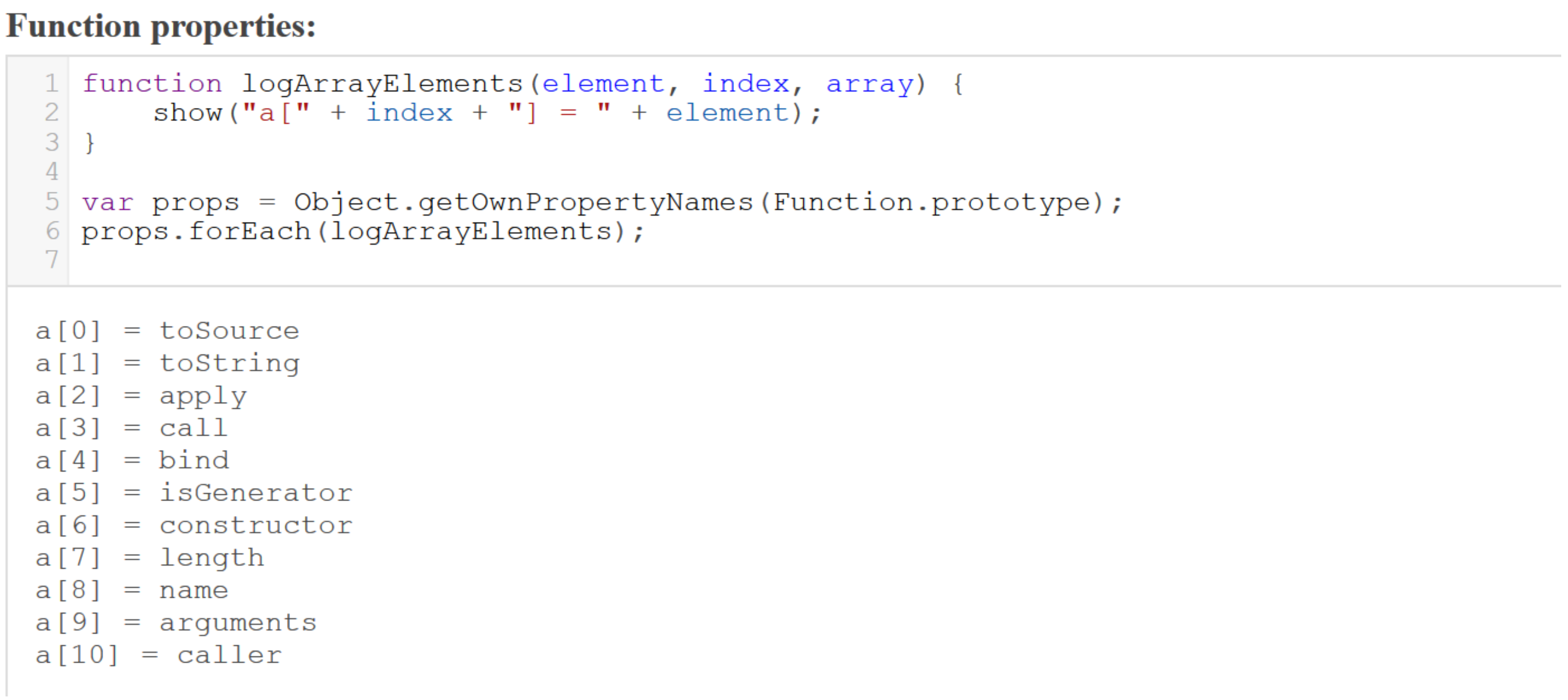
- Functions

Function object (all functions inherit from it) has the following properties:

- **arguments: An Array/object containing the arguments passed to the function**
    - **arguments.length: Stores the number of arguments in the array**
    - **arguments.callee: Pointer to the executing function (allows anonymous functions to recurse)**
- **length: The number of arguments the function was expecting**
- **constructor: function pointer to the constructor function**
- **prototype: allows the creation of prototypes**
- **Function object has the following methods:**
- **apply: A method that lets you more easily pass function arguments**
- **call: Allows you to call a function within a different context**
- **bind: creates a new function that, when called, has its 'this' set to the provided value, with a given sequence of arguments preceding any provided when the new function is called**
- **toString: Returns the source of the function as a string**

JavaScript Overview

- Functions



Function properties:

```
function logArrayElements(element, index, array) {
    show("a[" + index + "] = " + element);
}

var props = Object.getOwnPropertyNames(Function.prototype);
props.forEach(logArrayElements);

```

```
a[0] = toSource
a[1] = toString
a[2] = apply
a[3] = call
a[4] = bind
a[5] = isGenerator
a[6] = constructor
a[7] = length
a[8] = name
a[9] = arguments
a[10] = caller
```

JavaScript Overview

- Closures

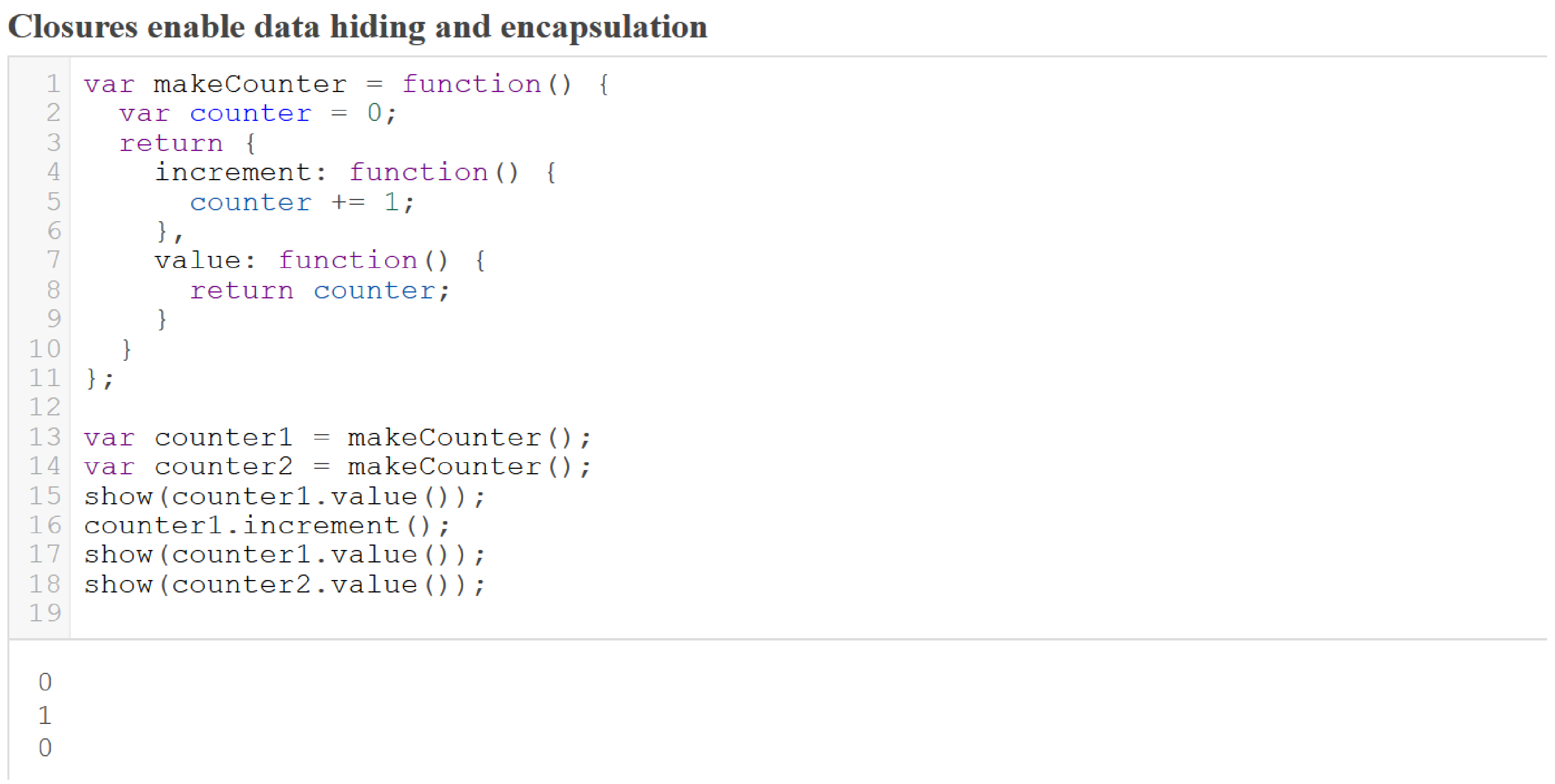
Closures are functions that refer to independent (free) variables.

Function defined in the closure 'remembers' the environment in which it was created. A closure, unlike a plain function pointer, allows a function to access those free variables even when invoked outside its immediate lexical scope.

- **Simply accessing variables outside of your immediate lexical scope creates a closure**
- **Inner function get full access to all the values defined in the outer function, not the other way around**

JavaScript Overview

- Closures



Closures enable data hiding and encapsulation

```javascript
var makeCounter = function() {
  var counter = 0;
  return {
    increment: function() {
      counter += 1;
    },
    value: function() {
      return counter;
    }
  }
};

var counter1 = makeCounter();
var counter2 = makeCounter();
show(counter1.value());
counter1.increment();
show(counter1.value());
show(counter2.value());
```

```
0
1
0
```

Functional Programming Overview

- **What is functional programming?**
- **Why Functional?**
- **A motivating example(?)**

Functional Programming Overview

- What is functional programming?

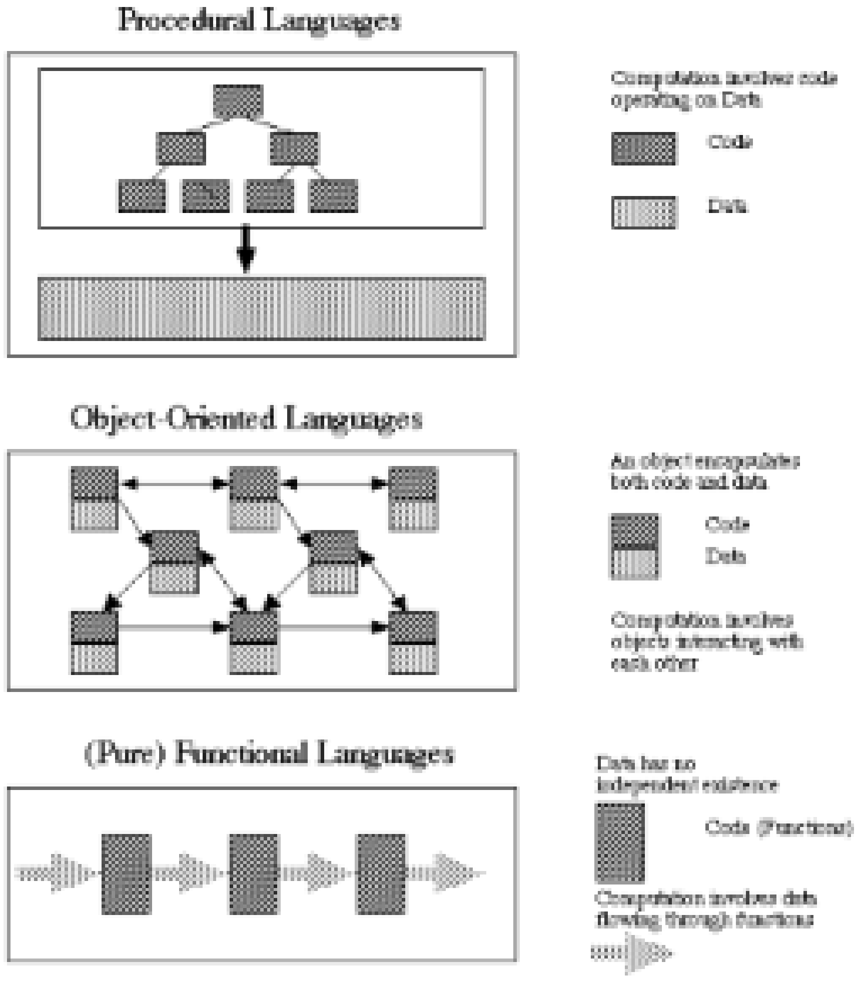
- **First Class Functions:**
  stored in variables, passed as arguments to functions, created within functions and returned from functions
- **Higher Order Functions:**
  Function that can accept functions as arguments, and/or can return a function
- **No Side Effects:**
  Function that does something other than returning the result is said to have side effects
- **Referential Transparency:**
  For a given set of arguments, the same code should always output the same value, only by changing arguments can a output value be different
- **Immutability:**
  Inability for variables to change their values once created. In other words, all things created stay constant
- **Currying / Partial Application:**
  Ability of a function to return a new function until it receives all it's arguments. Calling a curried function with only some of its arguments is called partial application
- **Tail Call Optimization:**
  Ability to avoid allocating a new stack frame for a function call. The most common use is tail-recursion, where a recursive function uses constant stack space.

Functional Programming Overview

- Why functional programming?

"Functional programming isn't the goal. The goal is to simplify the complicated."

- **Write less code**
- **Fewer errors**
- **Easy concurrency**
- **Testability**


Procedural Languages
Computation involves code operating on Data
Code
Data
Object-Oriented Languages
An object encapsulates both code and data
Code
Data
Computation involves objects interacting with each other
(Pure) Functional Languages
Data has no independent existence
Code (Functions)
Computation involves data flowing through functions

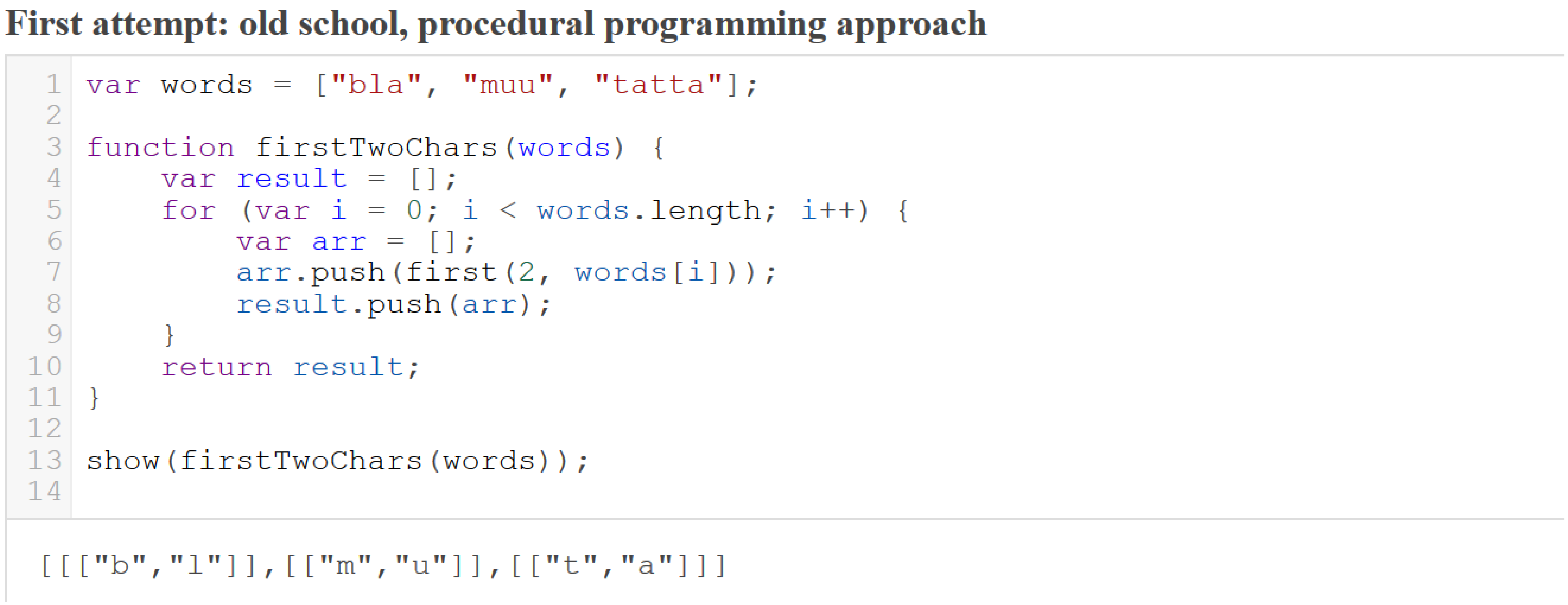
Functional Programming Overview

- Why functional programming?

... but who has time to learn new stuff (all the time)?

Functional Programming Overview

- Motivating example? *



First attempt: old school, procedural programming approach

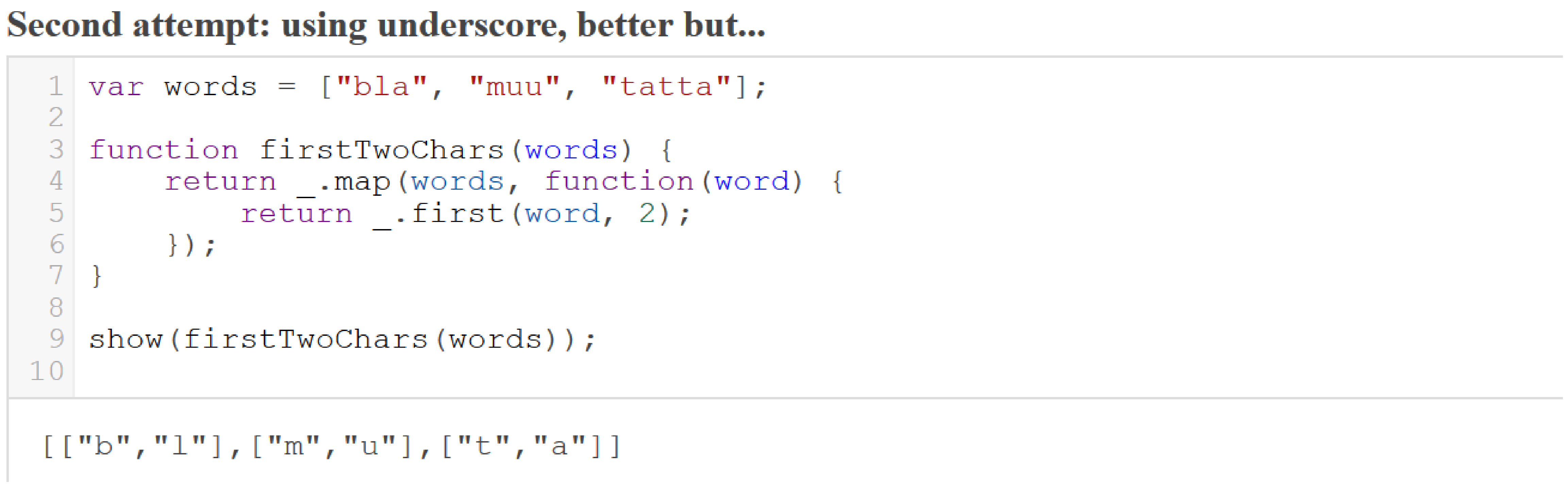
```
var words = ["bla", "muu", "tatta"];

function firstTwoChars(words) {
    var result = [];
    for (var i = 0; i < words.length; i++) {
        var arr = [];
        arr.push(first(2, words[i]));
        result.push(arr);
    }
    return result;
}

show(firstTwoChars(words));
```

```
[[["b","l"]],[["m","u"]],[["t","a"]]]
```

***Hat tip to Brian Lonsdorf, aka @drboolean. Watch his videos, great fun and motivating stuff!**
Hey Underscore, You're Doing It Wrong!
Functional programming patterns for the non-mathematician!

Functional Programming Overview

- Motivating example?



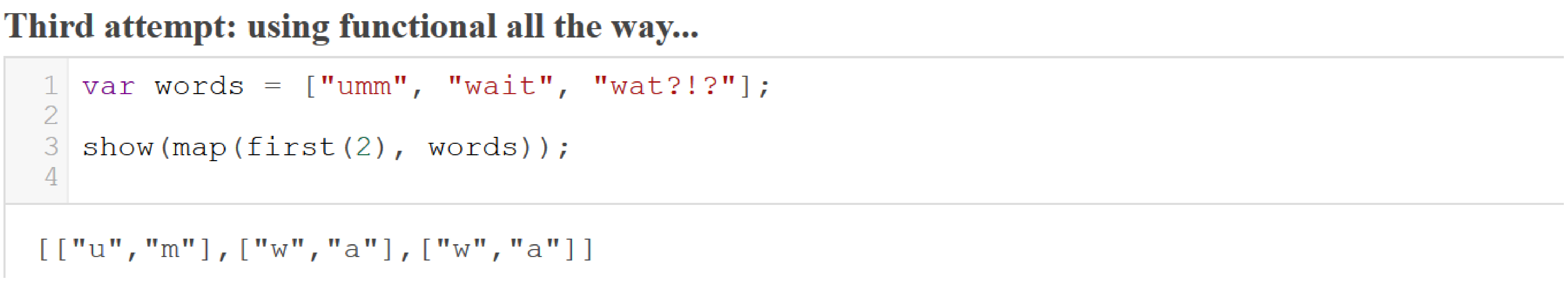
Second attempt: using underscore, better but...

```
var words = ["bla", "muu", "tatta"];

function firstTwoChars(words) {
    return _.map(words, function(word) {
        return _.first(word, 2);
    });
}

show(firstTwoChars(words));

```

```
[["b","l"],["m","u"],["t","a"]]
```

Functional Programming Overview

- Motivating example?



Third attempt: using functional all the way...

```
var words = ["umm", "wait", "wat?!?"];

show(map(first(2), words));

```

```
[["u","m"],["w","a"],["w","a"]]
```

Functional Programming Techniques

- **Functional JavaScript**
- **Pure Function, Higher Order Functions**
- **Composition, Combinators**
- **Currying/Partial Application**
- **Filter, Map, Reduce**
- **Try it out yourself...**

Functional Programming Techniques

JavaScript allows a variety of different programming paradigms: OO, functional, procedural
Although not a pure functional programming language, it allows one to program in a functional way.

Supports:

- **First Class Functions**
- **Higher Order Functions**
- **Anonymous functions**
- **Closures**

Functional Programming Techniques

JavaScript allows a variety of different programming paradigms: OO, functional, procedural
Although not a pure functional programming language, it allows one to program in a functional way.

Does not support directly, but possible with some discipline :)

- **Pure functions**
- **Immutability**
- **No Side Effects**

Functional Programming Techniques

JavaScript allows a variety of different programming paradigms: OO, functional, procedural
Although not a pure functional programming language, it allows one to program in a functional way.

Does not support directly, but possible with use of libraries or with ES6:

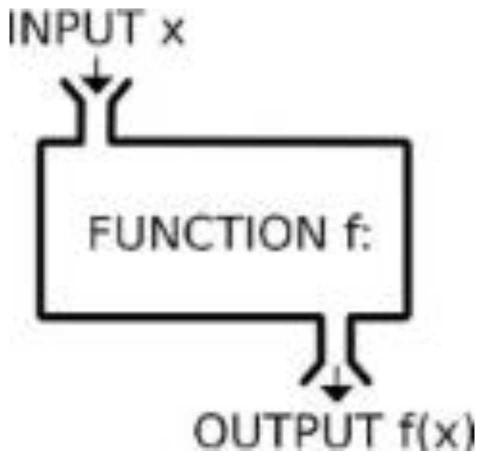
- **Currying/Partial Application**
- **Tail call optimization**
- **Pattern matching**
- **Lazy Evaluation**

Functional Programming Techniques

- Pure function

A function is considered pure if the result depends only on the arguments, and it has no side effects
The only result of invoking a pure function is the return value.

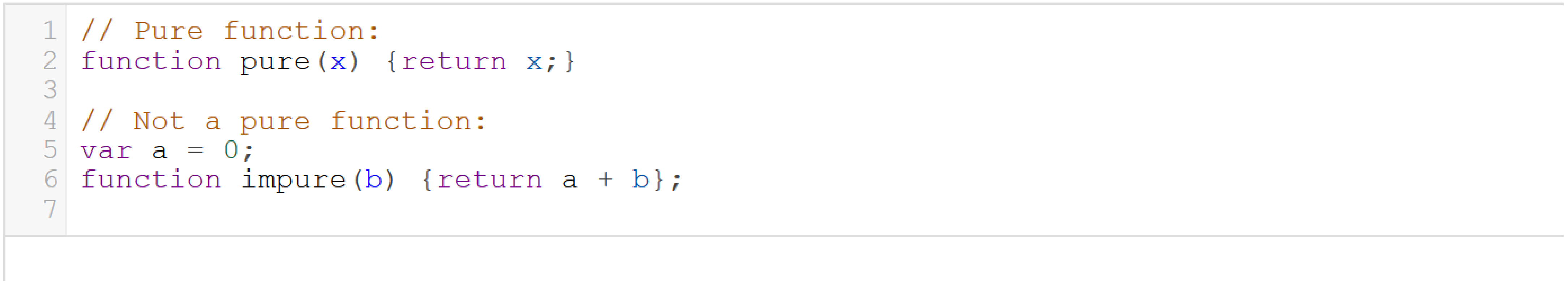
- **Same input, same ouput**
- **No side effects**
- **Can be cached**
- **Easy to test**
- **Allows code to run in parallel**



Functional Programming Techniques

- Pure function

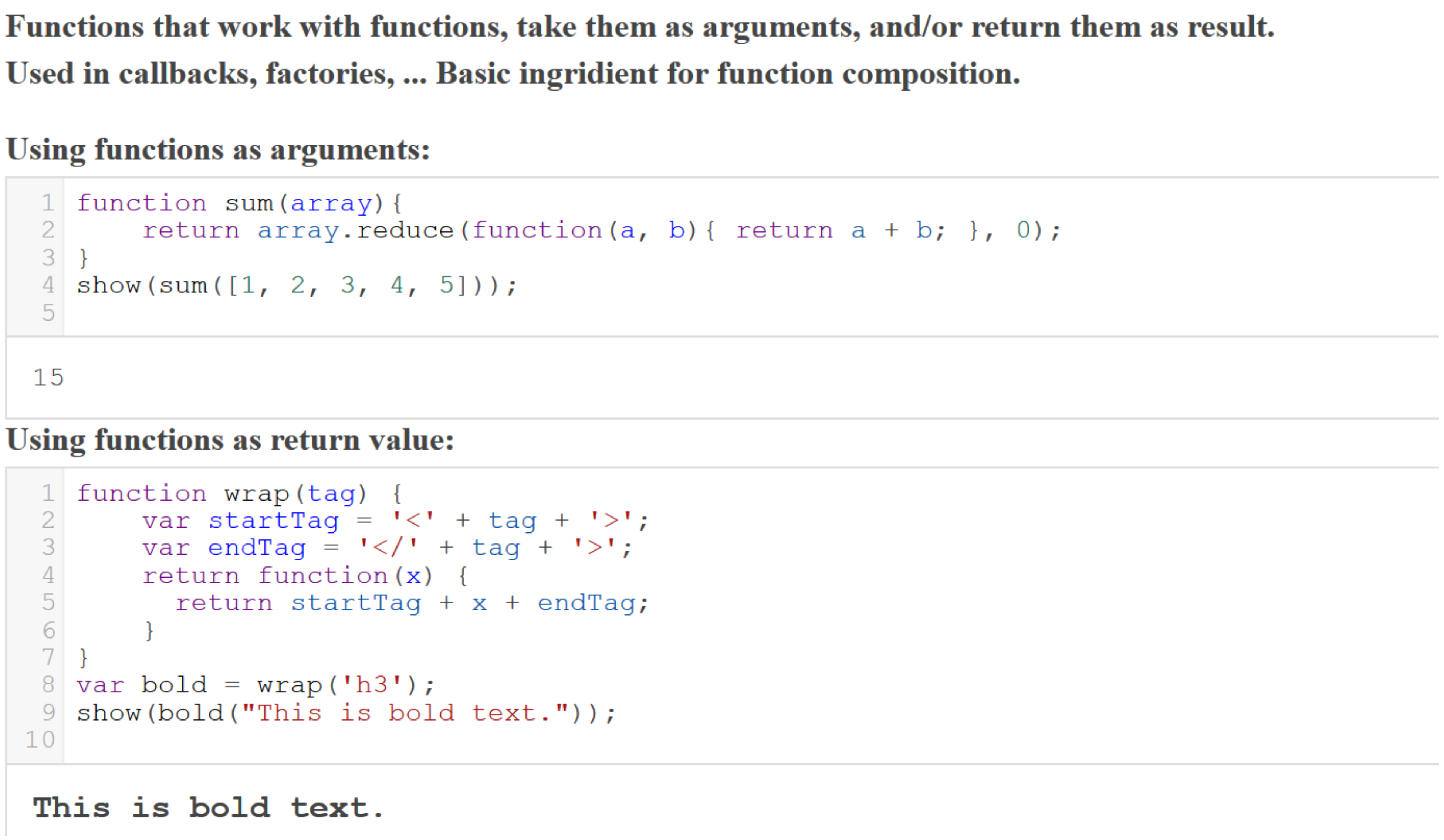
A function is considered pure if the result depends only on the arguments, and it has no side effects
The only result of invoking a pure function is the return value.



```
// Pure function:
function pure(x) {return x;}

// Not a pure function:
var a = 0;
function impure(b) {return a + b};

```

Functional Programming Techniques

- Higher order function



Functions that work with functions, take them as arguments, and/or return them as result.

Used in callbacks, factories, ... Basic ingridient for function composition.

Using functions as arguments:

```
function sum(array){
    return array.reduce(function(a, b){ return a + b; }, 0);
}
show(sum([1, 2, 3, 4, 5]));

```

```
15
```

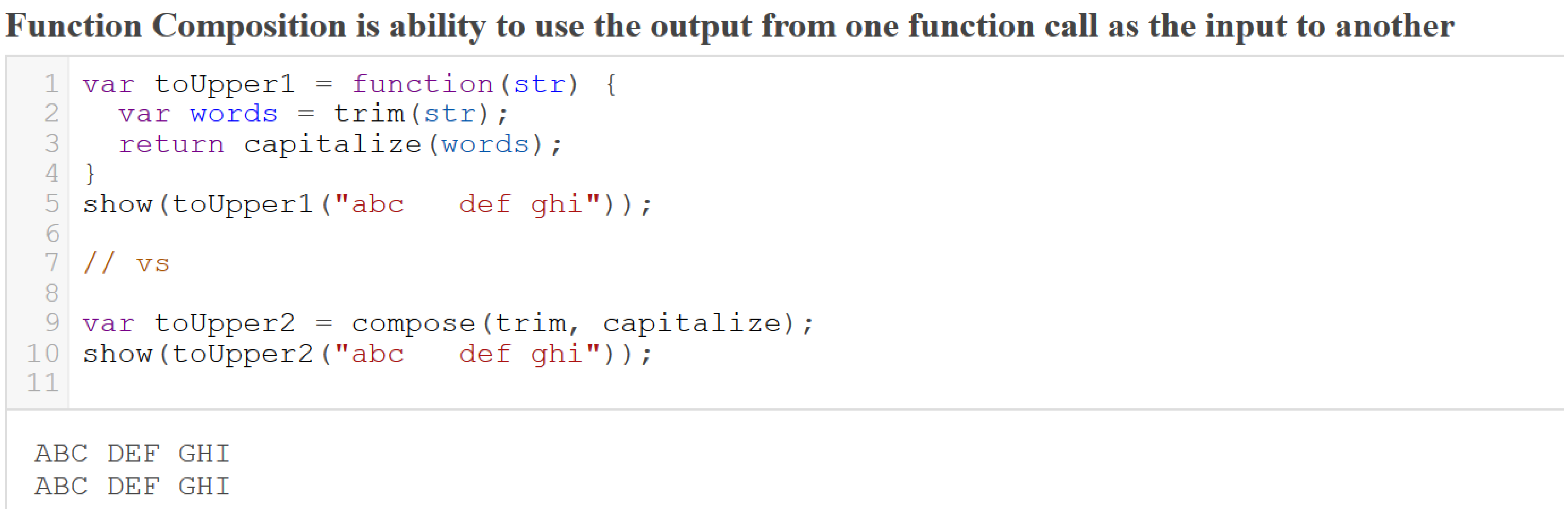
Using functions as return value:

```
function wrap(tag) {
    var startTag = '<' + tag + '>';
    var endTag = '</' + tag + '>';
    return function(x) {
      return startTag + x + endTag;
    }
}
var bold = wrap('h3');
show(bold("This is bold text."));

```

```
This is bold text.
```

Functional Programming Techniques

- Composition



Function Composition is ability to use the output from one function call as the input to another

```
var toUpper1 = function(str) {
  var words = trim(str);
  return capitalize(words);
}
show(toUpper1("abc   def ghi"));

// vs

var toUpper2 = compose(trim, capitalize);
show(toUpper2("abc   def ghi"));

```

```
ABC DEF GHI
ABC DEF GHI
```

Functional Programming Techniques

- Combinators

Ability to use functions as building blocks to make new functions
Function composition is simply one of many combinators.
Tacit programming (point-free programming) is a programming paradigm in which a function definition does not include information regarding its arguments, using combinators and composition in the function declaration instead of arguments

Some Examples*:

- **splat**
- **get**
- **pluck**

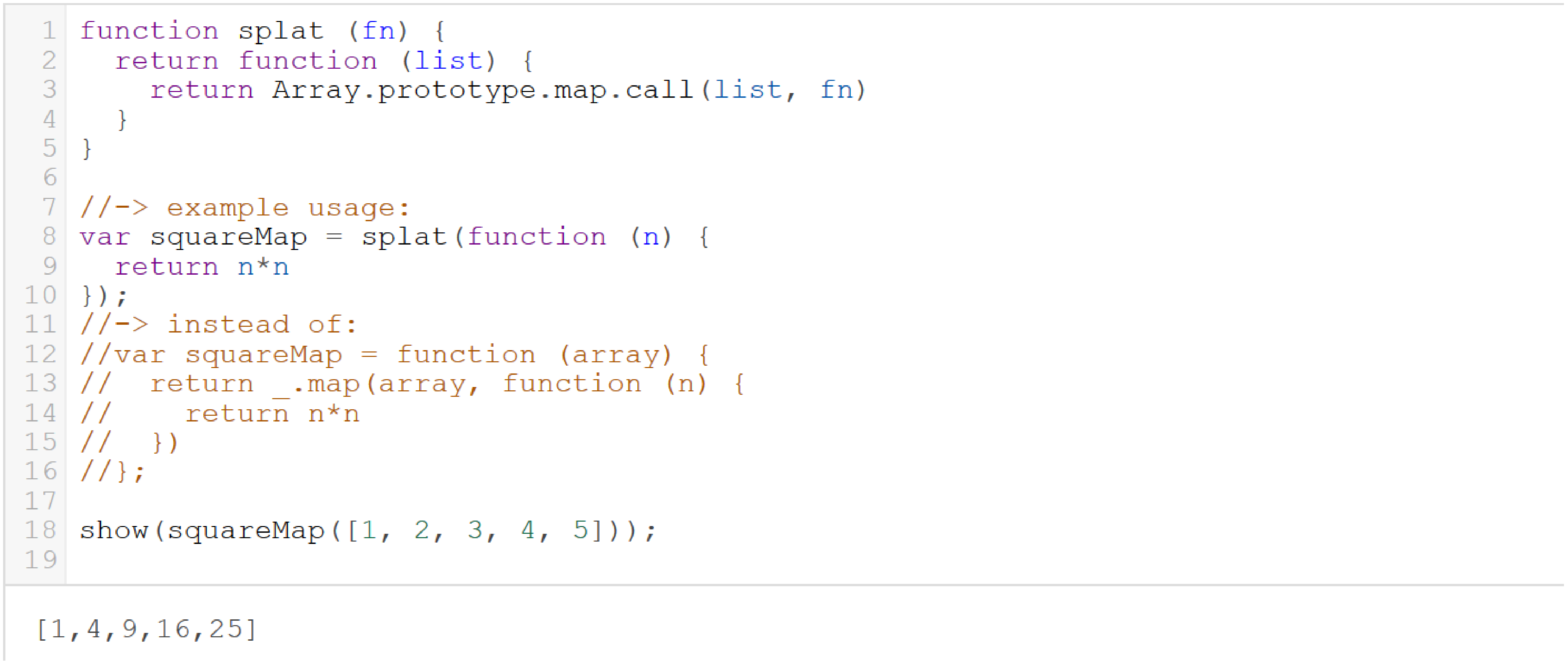
*Taken from Reginald Braithwaite's, aka @raganwald, article: Combinator Recipes for Working With Objects in JavaScript
Be sure to check out his projects and books!

Functional Programming Techniques

- Combinators

'splat' combinator:

Function that wraps around 'map' and turns any other function into a mapping capable function



```
function splat (fn) {
  return function (list) {
    return Array.prototype.map.call(list, fn)
  }
}

//-> example usage:
var squareMap = splat(function (n) {
  return n*n
});
//-> instead of:
//var squareMap = function (array) {
//  return _.map(array, function (n) {
//    return n*n
//  })
//};

show(squareMap([1, 2, 3, 4, 5]));

```

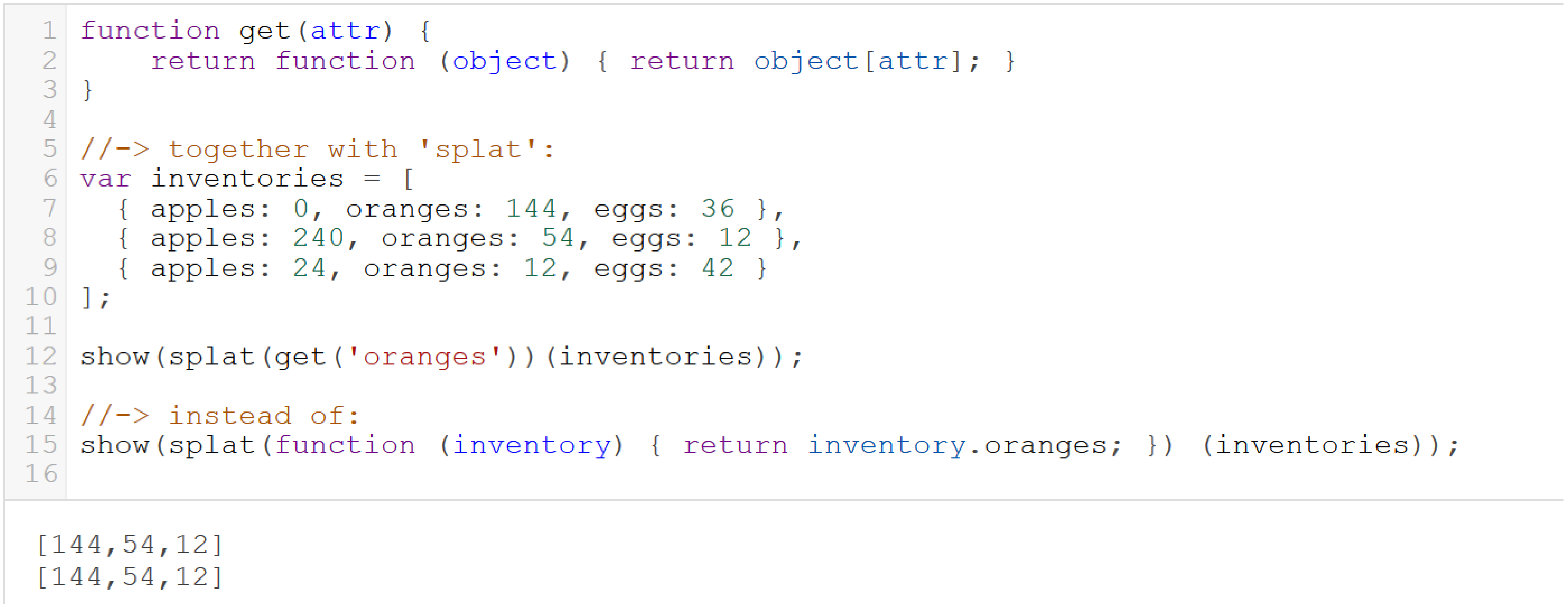
```
[1,4,9,16,25]
```

Functional Programming Techniques

- Combinators

'get' combinator:

Takes the name of a property and returns a function that gets that property from an object



```javascript
function get(attr) {
    return function (object) { return object[attr]; }
}

//-> together with 'splat':
var inventories = [
  { apples: 0, oranges: 144, eggs: 36 },
  { apples: 240, oranges: 54, eggs: 12 },
  { apples: 24, oranges: 12, eggs: 42 }
];

show(splat(get('oranges'))(inventories));

//-> instead of:
show(splat(function (inventory) { return inventory.oranges; }) (inventories));

```

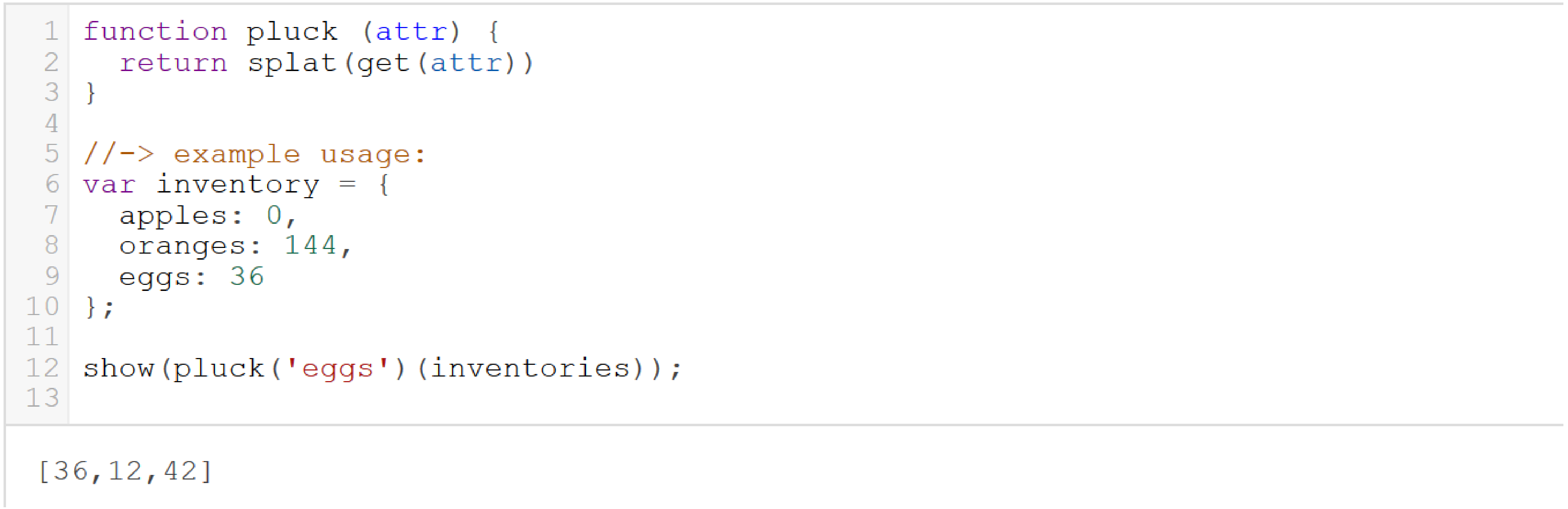
```
[144,54,12]
[144,54,12]
```

Functional Programming Techniques

- Combinators

'pluck' combinator:

Formalizes combining 'get' and 'splat'



```
function pluck (attr) {
  return splat(get(attr))
}

//-> example usage:
var inventory = {
  apples: 0,
  oranges: 144,
  eggs: 36
};

show(pluck('eggs')(inventories));

```

```
[36,12,42]
```

Functional Programming Techniques

- Currying / Partial application

Currying produces a function that will return a new function until it receives all it's arguments

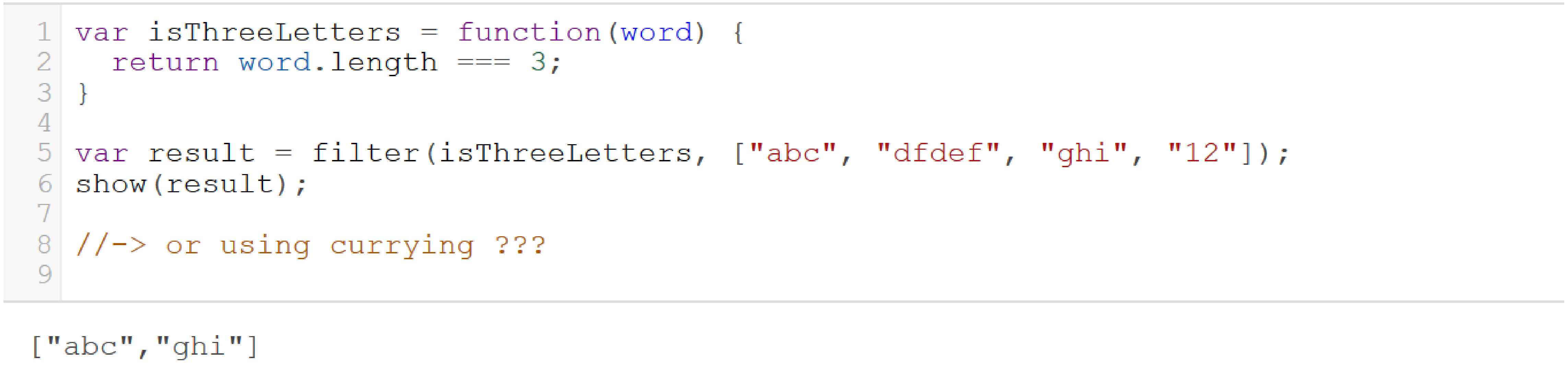
Currying enables Partial Application, and together they help:

- **Making generic functions**
- **Building new functions by applying arguments**
- **Better granularity of functions**
- **More powerful function composition**

Functional Programming Techniques

- Example: Filter

The filter method transforms an array by applying a predicate function to all of its elements, and building a new array from the elements for which predicate returned true.



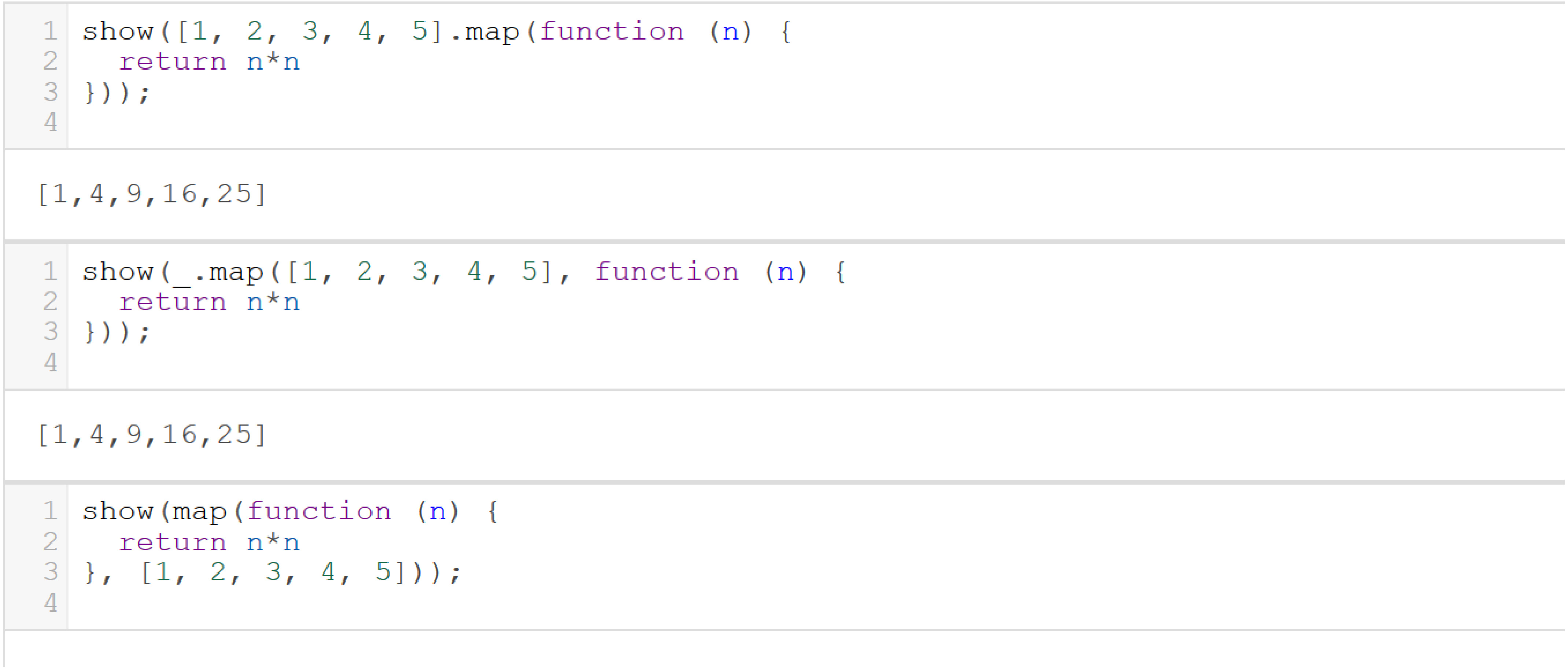
```
var isThreeLetters = function(word) {
  return word.length === 3;
}

var result = filter(isThreeLetters, ["abc", "dfdef", "ghi", "12"]);
show(result);

//-> or using currying ???

```

```
["abc","ghi"]
```

Functional Programming Techniques

- Example: Map

Map takes a function as an argument, and applies it to each of the elements of the array, then returns the results in a new array



```
show([1, 2, 3, 4, 5].map(function (n) {
  return n*n
}));
```

```
[1,4,9,16,25]
```

```
show(_.map([1, 2, 3, 4, 5], function (n) {
  return n*n
}));
```

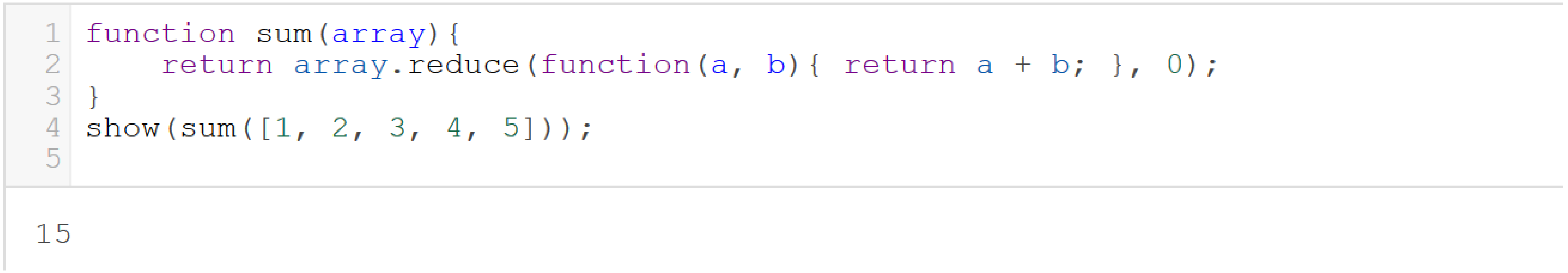
```
[1,4,9,16,25]
```

```
show(map(function (n) {
  return n*n
}, [1, 2, 3, 4, 5]));
```

Functional Programming Techniques

- Example: Reduce (Fold)

Reduce applies a function against an accumulator and each value of the array (from left-to-right) to reduce it to a single value

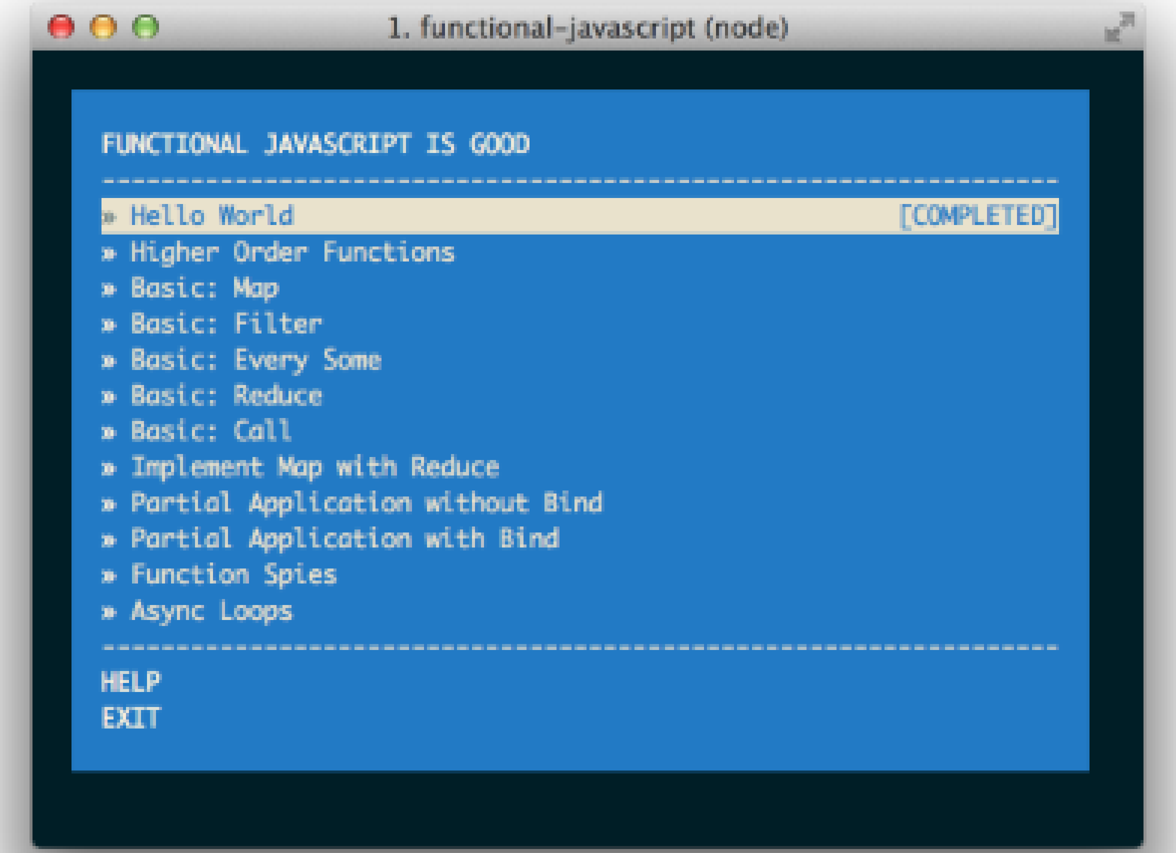


```
function sum(array){
    return array.reduce(function(a, b){ return a + b; }, 0);
}
show(sum([1, 2, 3, 4, 5]));

```

```
15
```

Functional Programming Techniques

- Try it yourself…



```
1. functional-javascript (node)

FUNCTIONAL JAVASCRIPT IS GOOD
------------------------------------------------------------------
» Hello World                                         [COMPLETED]
» Higher Order Functions
» Basic: Map
» Basic: Filter
» Basic: Every Some
» Basic: Reduce
» Basic: Call
» Implement Map with Reduce
» Partial Application without Bind
» Partial Application with Bind
» Function Spies
» Async Loops
------------------------------------------------------------------
HELP
EXIT
```

[@timoxley: functional-javascript-workshop]()

Resources

This presentation: Functional Javascript/
Original Deck/CodeMirror plugin by Irene Ros: deck.js-codemirror

Javascript Functional (and not so functional*) programming libraries:

- **allong.es: http://allong.es/**
- **Ramda: https://github.com/CrossEye/ramda**
- **LambdaJs: https://github.com/loop-recur/lambdajs/**
- **Falktale: http://folktale.github.io/**
- ***Underscore: http://underscorejs.org/**
- ***Lo-Dash: http://lodash.com/**

Books on Functional JavaScript

- **Functional JavaScript, Michael Fogus**
- **Eloquent JavaScript, Marijn Haverbeke**
- **JavaScript Allongé, Reginald Braithwaite**