

Functional programming

in JavaScript

Functions

Properties of Functions

- Normal objects
- Inherit from `Function.prototype`
- Can have a name but don't need one
- Create their own scope
- Can be invoked in different ways

Creating them

```
function foo() { }
```

```
var anonymous = function() {};
```

```
var named = function named () {};
```

Scope

- Only functions create scopes
- No scopes for if, for, ...!
- Always use “var” keyword to put variable in local scope

Closures

- Functions inherit all variables from the scope they were defined in
- “Enclose” the scope - Closures

Scope

```
function example(param) {  
    var a;  
    setTimeout(function() {  
        // available here: a, param  
    }, 2000);  
}
```

this

- always depends on the function call
- have to save it for closures
- can be modified using call and apply

```
var Greetings = {  
  name: "Michael",  
  sayHello: function() {  
    alert('Hello, ' + this.name);  
  }  
}
```

```
Greetings.sayHello();
```

```
var Greetings = {  
  name: "Michael",  
  sayHello: function() {  
    alert('Hello, ' + this.name);  
  }  
}
```

```
var f = Greetings.sayHello;  
f();
```

```
var Greetings = {  
  name: "Michael",  
  sayHello: function() {  
    alert('Hello, ' + this.name);  
  }  
}
```

```
var f = Greetings.sayHello;  
f.call(Greetings);
```

```
var Greetings = {  
  name: "Michael",  
  sayHelloLater: function() {  
    var that = this;  
    setTimeout(function() {  
      alert('Hello, ' + that.name);  
    }, 1000);  
  }  
}
```

```
Greetings.sayHelloLater();
```

Arguments

- Special keyword “arguments”
- Available in every function
- Is sort-of like an array...
- arguments.callee

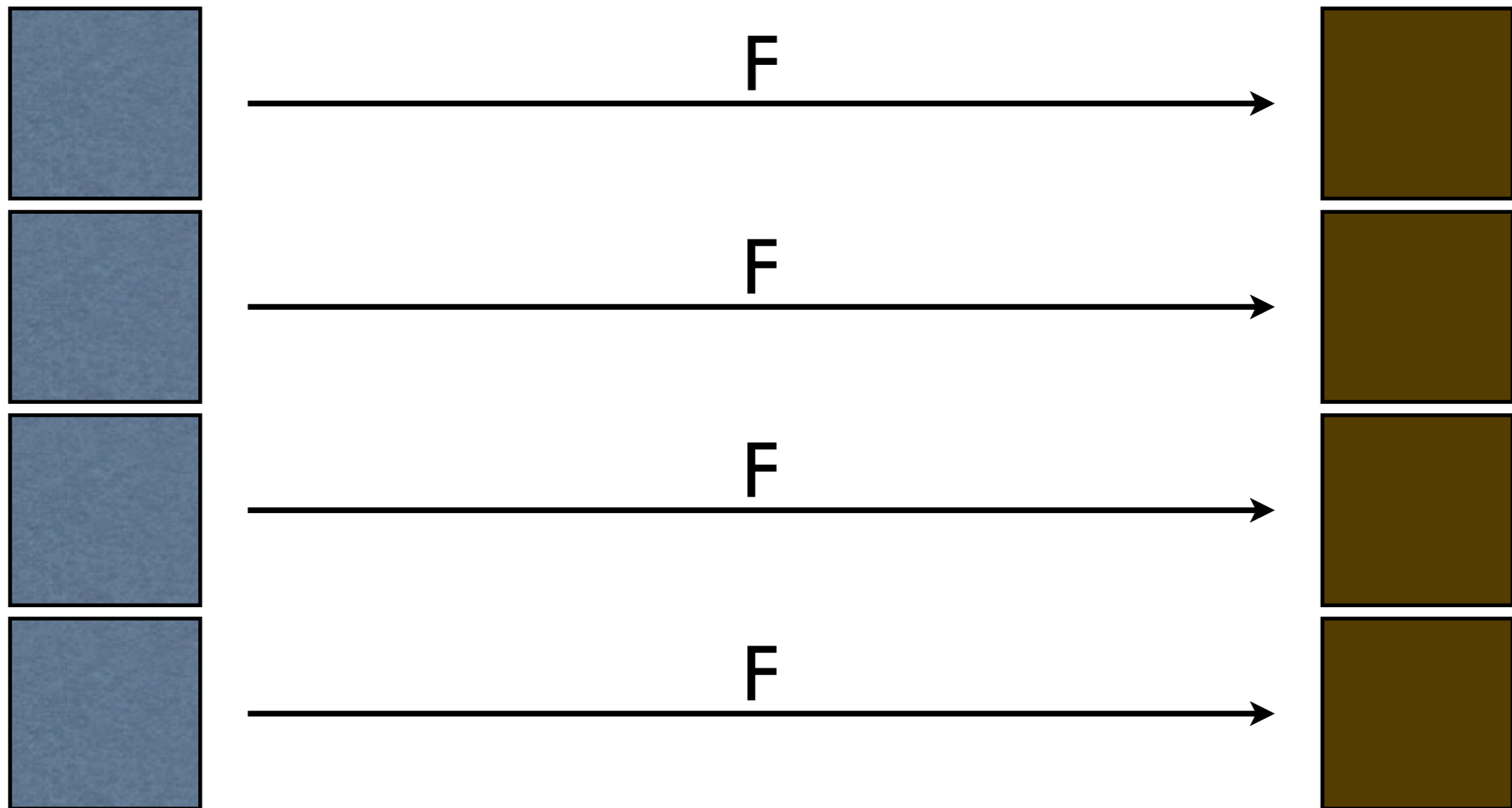
How can we use it

- callbacks
- operations on lists
- encapsulating functions

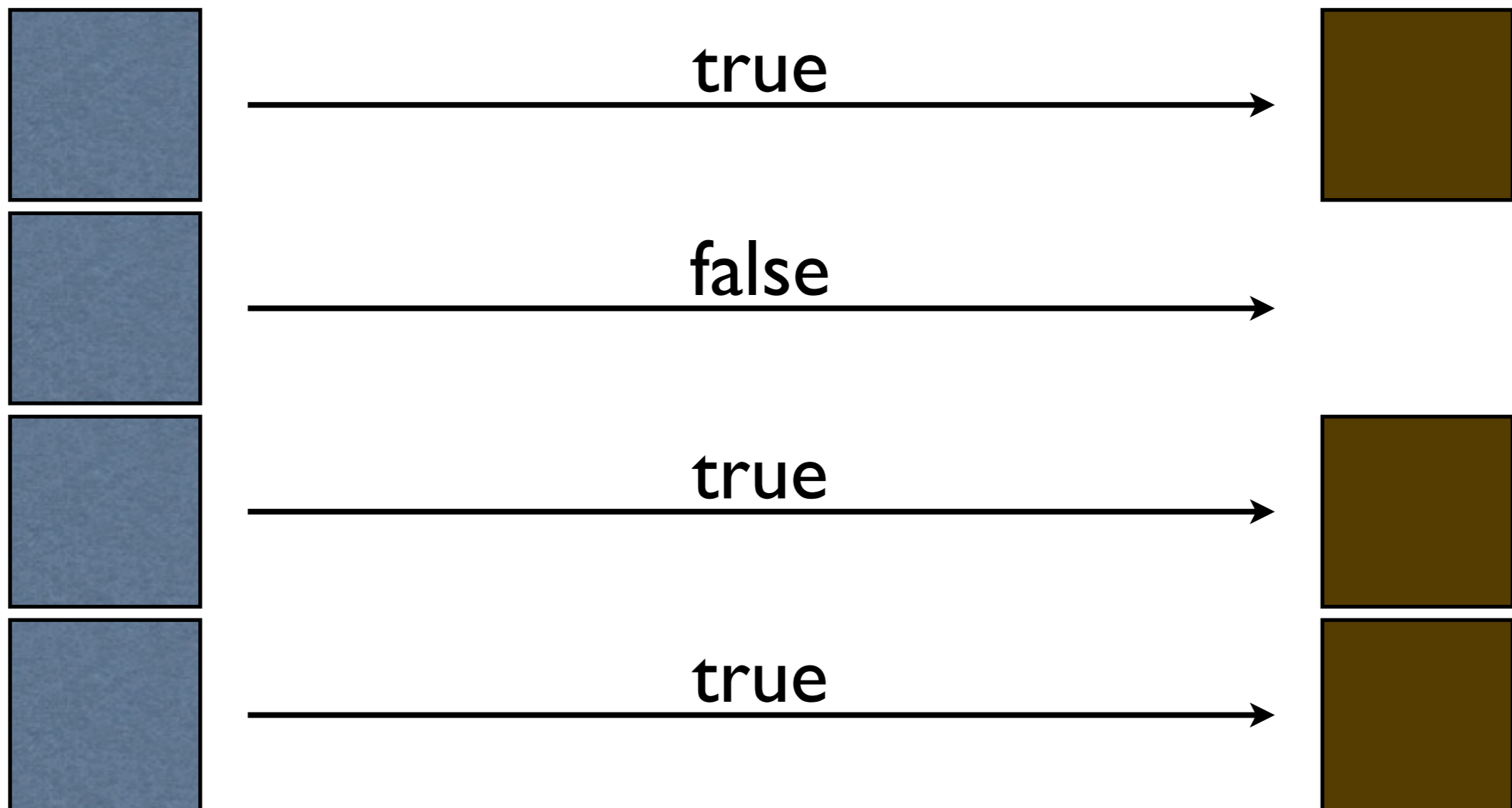
```
function alertResult(callback) {  
    alert(callback());  
}
```

```
alertResult(function() {  
    return 42;  
});
```

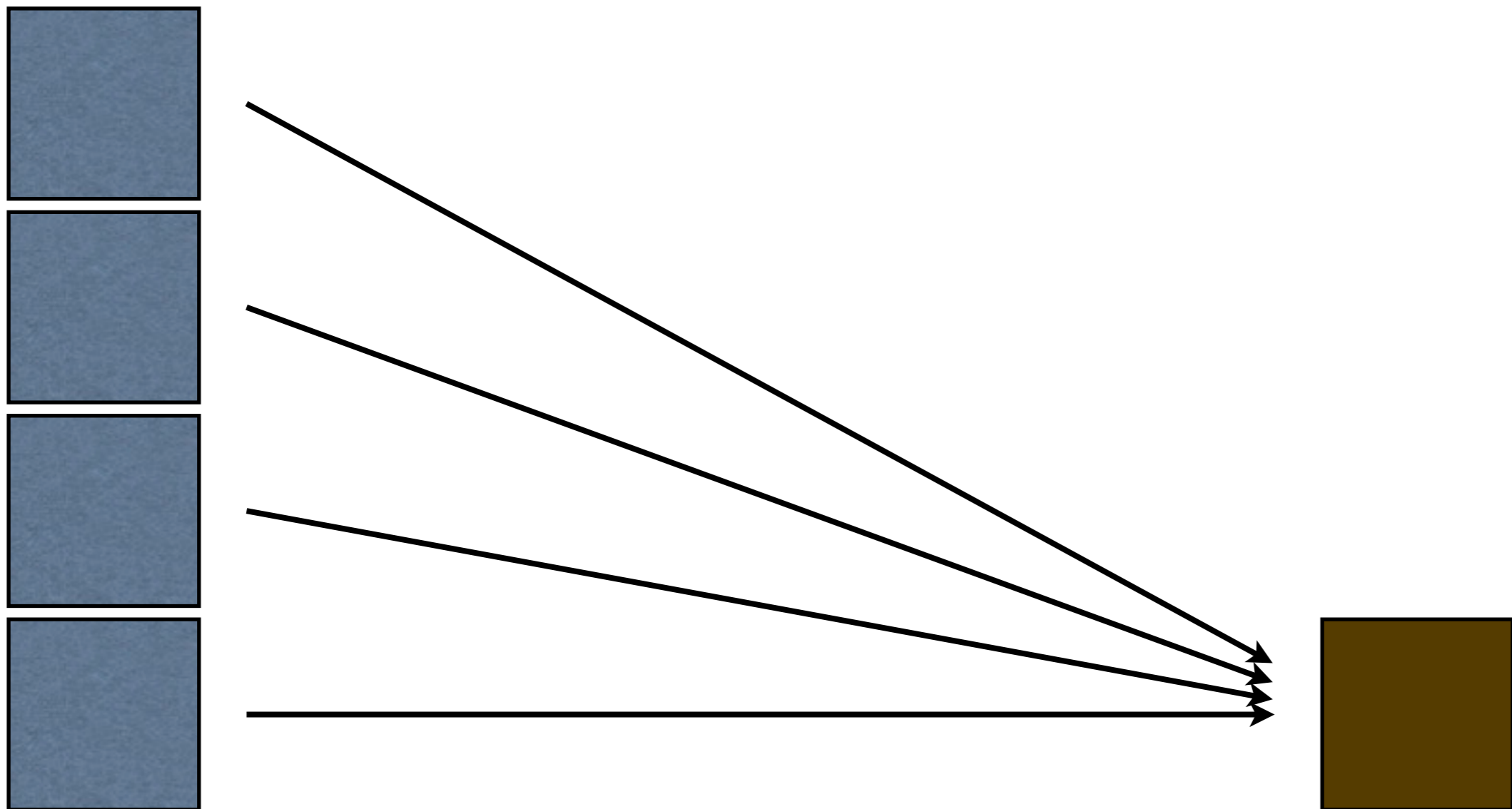
Map



Filter



Reduce



```
function reduce(list, start, op) {  
    var i = start;  
    $(list).each(function(k, element) {  
        i = op(element, i);  
    });  
    return i;  
}
```

```
var list = [12, 20];
```

```
var total = reduce(list, 0, function (a,b) {  
    return a+b;  
});
```

```
var total2 = reduce(list, 0, function (a,b) {  
    return a*b;  
});
```

```
var doSomething = function(a,b) {  
    return a+b;  
}
```

```
doSomething(10, 5);
```

```
var doSomething = function(a,b) {  
    return a+b;  
}
```

```
function logger(func) {  
    return function() {  
        console.log("Function was called. Arguments: ");  
        console.log(arguments);  
        var result = func.apply(this, arguments);  
        console.log("Return value:")  
        console.log(result);  
        return result;  
    }  
}
```

```
doSomething = logger(doSomething);  
doSomething(10, 5);
```

The screenshot shows a web browser's developer console. The console is open to the 'Console' tab, which includes sub-tabs for HTML, CSS, Script, DOM, Net, and YSlow. The console contains the following text:

```
>>> var doSomething = function(a,b) { return a+b...thing = logger(doSomething); doSomething(10, 5);
Function was called. Arguments:
[ 10, 5 ]
Return value:
15
15
```

On the right side of the console, there is a code editor showing the definition of the `logger` function and the execution of the recursive function:

```
function logger(func) {
  return function() {
    console.log("Function was called. Arguments: ");
    console.log(arguments);
    var result = func.apply(this, arguments);
    console.log("Return value:")
    console.log(result);
    return result;
  }
}

doSomething = logger(doSomething);
doSomething(10, 5);
```

At the bottom of the code editor, there are buttons for 'Run', 'Clear', and 'Copy'.