Intro to Javascript

Functions

(Part 1)

FUNCTIONS

A *function* is a block of code that has a name. Some languages call them *subroutines* or *procedures*. We'll also call them *methods* (although that's a precise definition that we'll explore in a bit). We can execute a function simply by saying—or *calling*—its name. We've seen functions before:

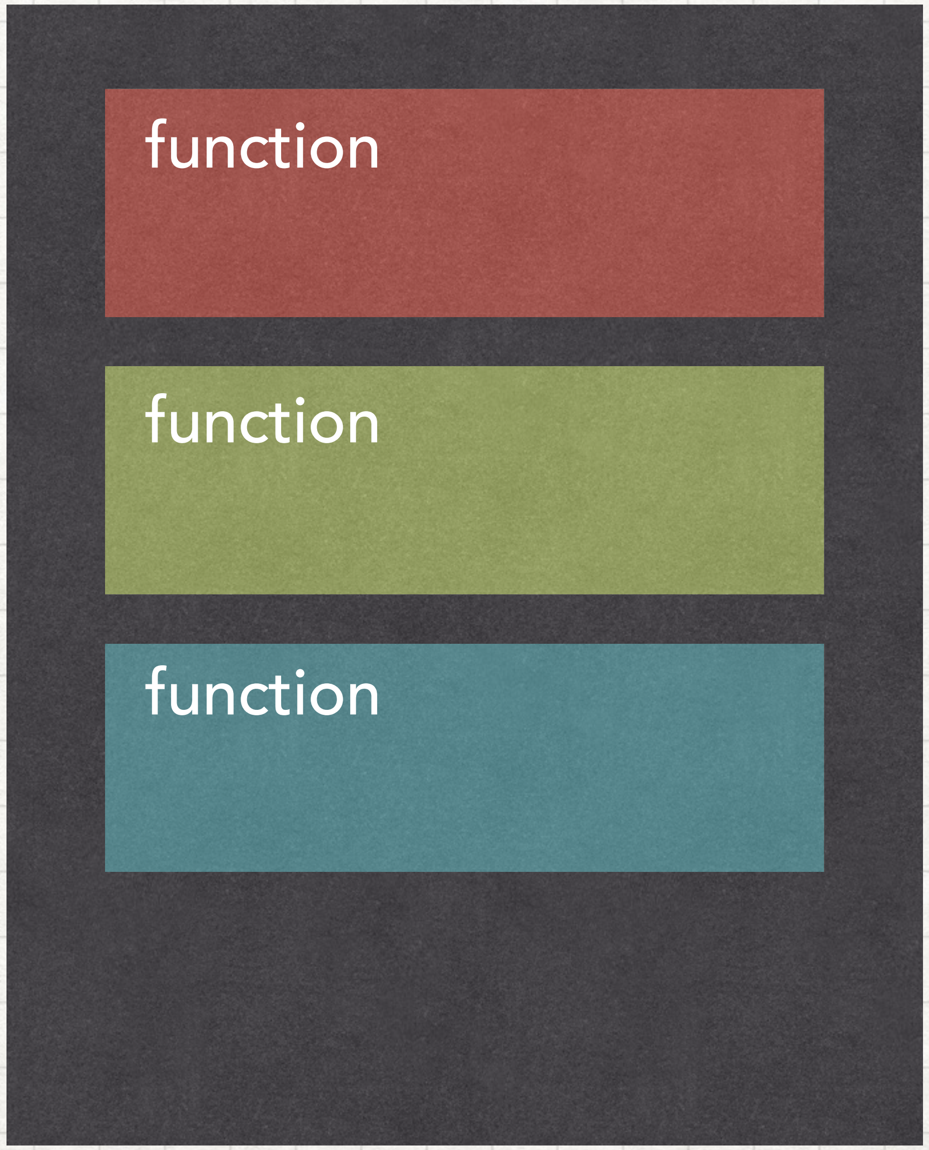
```
coin = Math.random()
document.write()
document.writeln()
```

In that code, `random()` and `write()` and `writeln()` are all functions. They are built-in to Javascript and we can call them at any time from anywhere in our code. Functions typically take some *input* and then either perform an action or deliver to us some *output*. If you like grammar, you might think of them as verbs.

Functions are also one way our programs can break complex actions down into smaller sub-tasks that are easier to code and to control. Thinking about *modularity* and *how to break complex things into smaller parts* are essential skills that programmers need to develop.

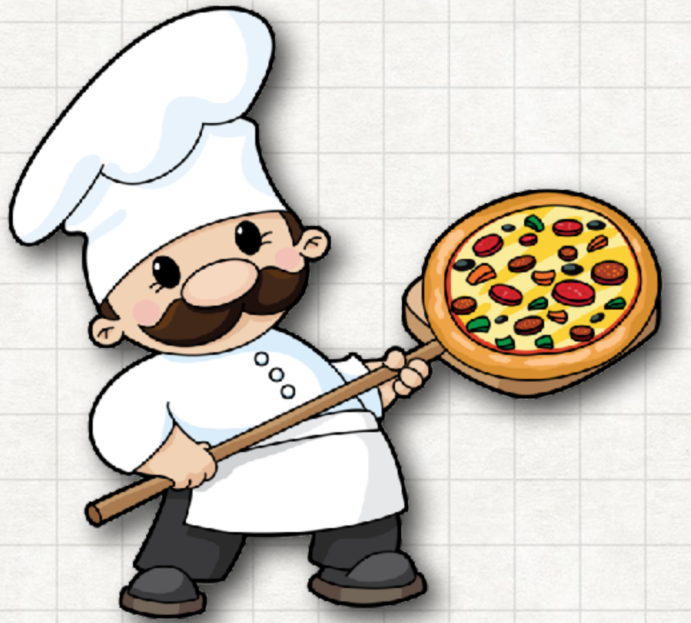
MODULARITY

But modularity is one of the most important things that functions do. We can write long programs that are thousands of lines long, but more experienced programmers divide tasks into smaller units and write functions to perform those activities.



FUNCTIONS

Modularity is important. For example, programming a robot to make pizza could be done in one long Javascript program. But we gain advantages by dividing the process into sub-tasks:



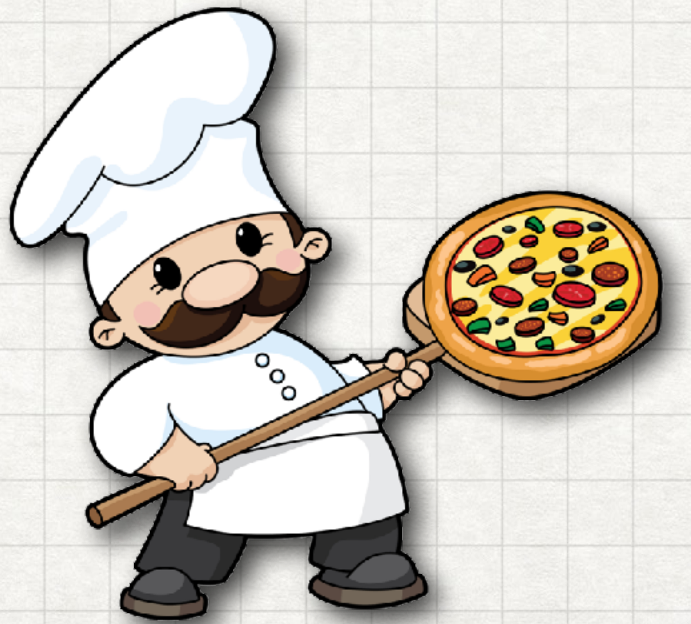
- mix the dough
- roll out the dough
- add toppings
- bake the pizza
- slice it
- box it up

In our code, we'll make our robot more modular by making each one of these processes a function.

FUNCTIONS

Functions allow us to modify the individual steps in the pizza-making process. We can now provide input parameters that can alter the kinds of pizzas we make:

mix the dough	(traditional, whole wheat, gluten-free)
roll out the dough	(traditional, thin, pan)
add toppings	(any combo of ingredients from menu)
bake the pizza	(time and temp)
slice it	(how many)
box it up	(box size)



FUNCTIONS IN JAVASCRIPT

It makes sense to say that functions are the verbs of a programming language. A big part of programming, then, is writing our own functions. A function is an ***activity*** or an ***action*** or a ***behaviour*** or a ***transaction*** that has a name.

We've seen before how to simulate the flipping of a coin. Here's a slight variation on the action that we performed before:

```
function flipCoin() {
  if (Math.random() < 0.5) {
    return 'heads';
  } else {
    return 'tails';
  }
}
```

FUNCTIONS IN JAVASCRIPT

By wrapping that activity inside a function, we can create a new verb. Our function has a body that's enclosed between `{` and `}`. It's denoted by the keyword `function` and it has a custom name: here, that's `flipCoin`. It also can possibly take some input inside `(` and `)`. This function, however, takes no input:

```
function flipCoin() {
  if (Math.random() < 0.5) {
    return 'heads';
  } else {
    return 'tails';
  }
}
```

FUNCTIONS IN JAVASCRIPT

Our program can call — or execute — this function as often as we'd like:

```
function flipCoin() {
   if (Math.random() < 0.5) {
      return 'heads';
   } else {
      return 'tails';
   }
}

coin1 = flipCoin();
coin2 = flipCoin();
coin3 = flipCoin();
```

ANOTHER FUNCTION

Let’s write another very simple function. Usually, functions will be more complex than this, but let’s keep it simple so that we can focus on the function definition itself:

```
function fullname(first, last) {
   name = first + ' ' + last;
   return name;
}
```

ANOTHER FUNCTION

Let's write another very simple function. Usually, functions will be more complex than this, but let's keep it simple so that we can focus on the function definition itself:

```
function fullname(first, last) {
   name = first + ' ' + last;
   return name;
}
```

The keyword `function` tells Javascript what we're doing here. The name of this function is `fullname`.

ANOTHER FUNCTION

Let's write another very simple function. Usually, functions will be more complex than this, but let's keep it simple so that we can focus on the function definition itself:

```
function fullname(first, last) {
   name = first + ' ' + last;
   return name;
}
```

The body of the function goes inside { and } curly brackets. The function can be short or long, but Javascript knows that every statement inside the brackets belongs to the function. Notice that we indent our statements for clarity.

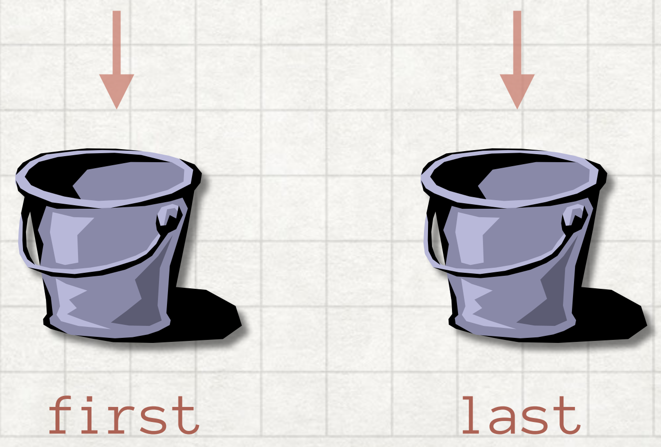
ANOTHER FUNCTION

Let’s write another very simple function. Usually, functions will be more complex than this, but let’s keep it simple so that we can focus on the function definition itself:

```
function fullname(first, last) {
    name = first + ' ' + last;
    return name;
}
```

These variables receive the data sent to the function from the caller (whoever or whatever that is!). For example, if the function is called like this, then the data lines up like so:

```
fullname('Bill', 'Shakespeare')
```

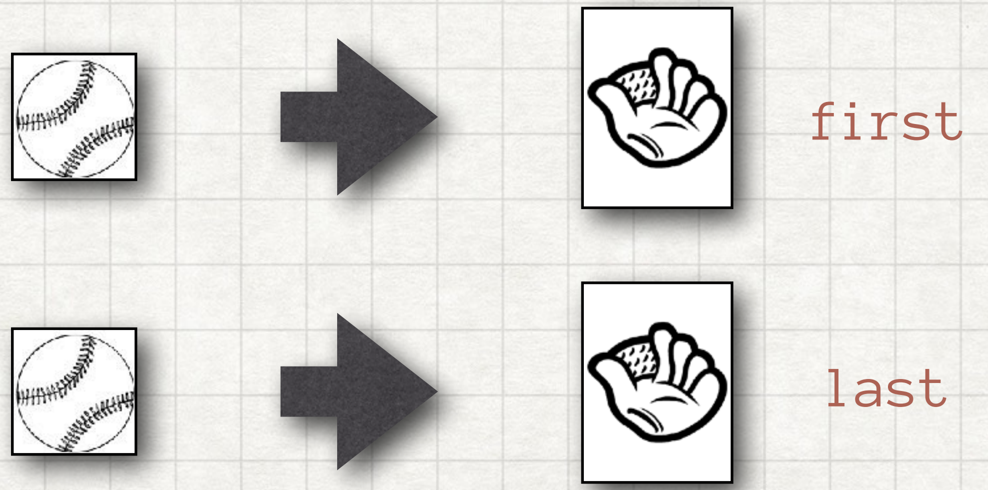


EXAMPLE FUNCTION

Let’s write a very simple function. Usually, functions will be more complex than this, but let’s keep it simple so that we can focus on the function definition itself:

```
function fullname(first, last) {
   name = first + ' ' + last;
   return name;
}
```

Think of these variables as catcher’s mitts. The main program “throws” data to the function and the function “catches” the data in these variables.

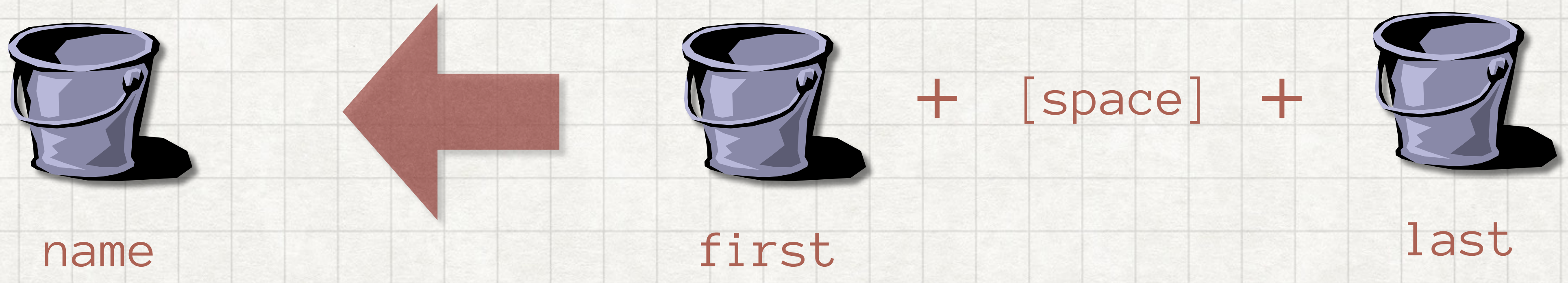


EXAMPLE FUNCTION

Let's write a very simple function. Usually, functions will be more complex than this, but let's keep it simple so that we can focus on the function definition itself:

```
function fullname(first, last) {
   name = first + ' ' + last;
   return name;
}
```

This function concatenates three strings together and assigns the result to a variable called name.

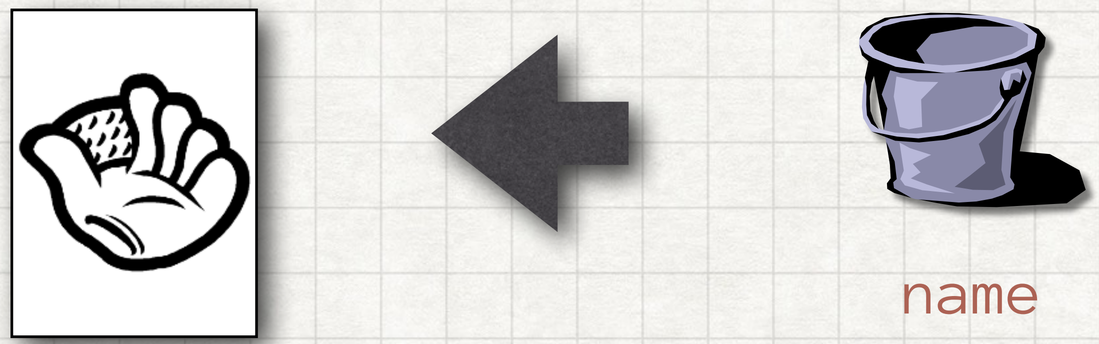


EXAMPLE FUNCTION

Let's write a very simple function. Usually, functions will be more complex than this, but let's keep it simple so that we can focus on the function definition itself:

```
function fullname(first, last) {
   name = first + ' ' + last;
   return name;
}
```

name is "thrown" back to the main program, which is responsible for "catching" it.



FUNCTION IN ACTION

In a program, the whole thing might look like this:

```
"use strict";

document.write(name);

function fullname(first, last) {
   var name = first + ' ' + last;
   return name;
}

var writer = fullname('Bill', 'Shakespeare');
document.write("Writer: " + writer + "<br>");

document.write("hello," + name);
```

The function is defined here, but it does not execute until the main program "calls" it.

FUNCTION IN ACTION

In a program, the whole thing might look like this:

```
"use strict";

document.write(name);

function fullname(first, last) {
   var name = first + ' ' + last;
   return name;
}

var writer = fullname('Bill', 'Shakespeare');
document.write("Writer: " + writer + "<br>");

document.write("hello," + name);
```

Here, we call the function, sending two strings.

FUNCTION IN ACTION

In a program, the whole thing might look like this:

```
"use strict";

document.write(name);

function fullname(first, last) {
   var name = first + ' ' + last;
   return name;
}

var writer = fullname('Bill', 'Shakespeare');
document.write("Writer: " + writer + "<br>");

document.write("hello," + name);
```

The function returns data to the main program, which "catches" it in the variable called `writer`.

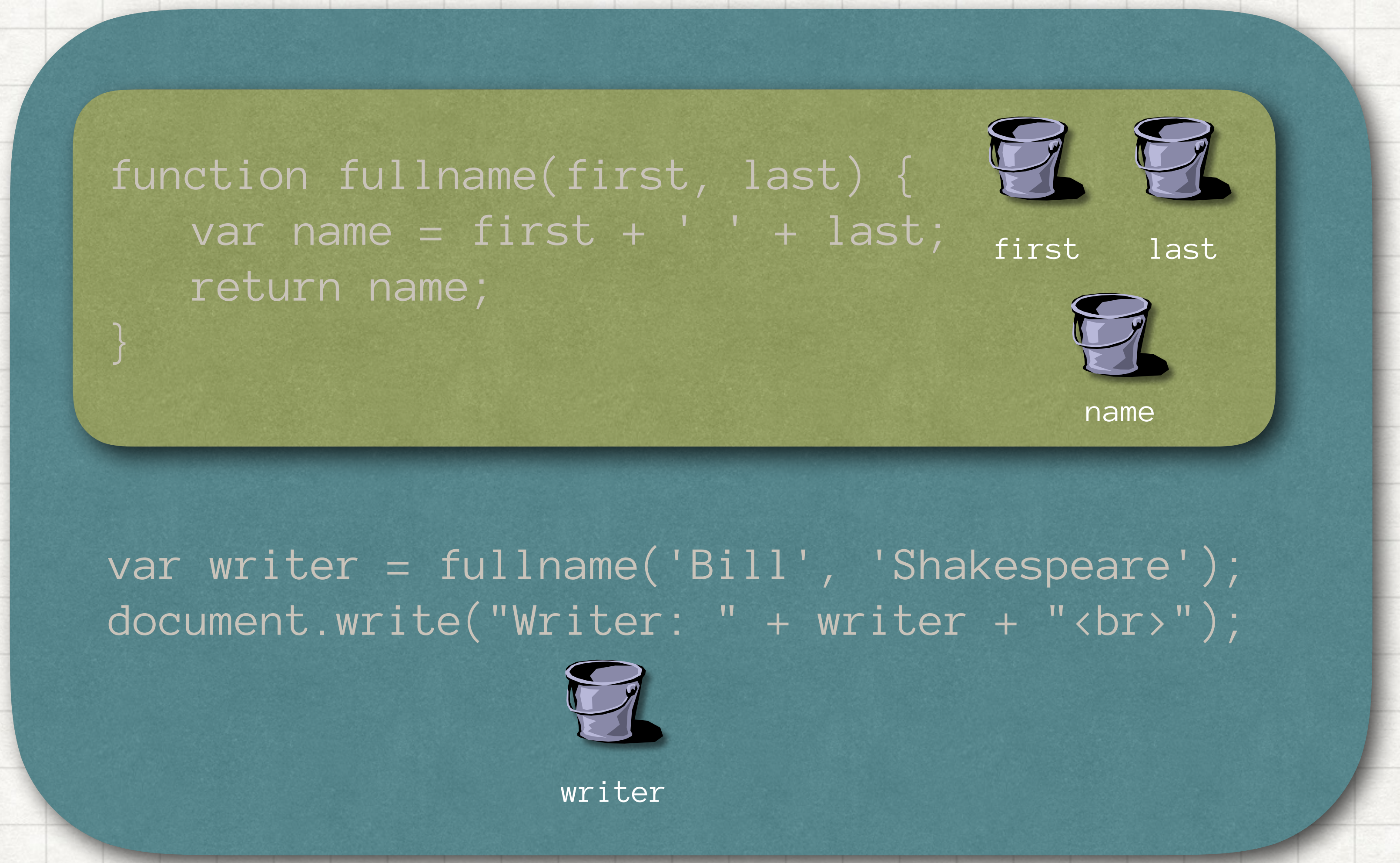
writer

Intro to Javascript

Variable Scope

VARIABLE SCOPE

We might diagram the relationship between the main program and the function like this. The function exists as a self-contained "space" within the program.



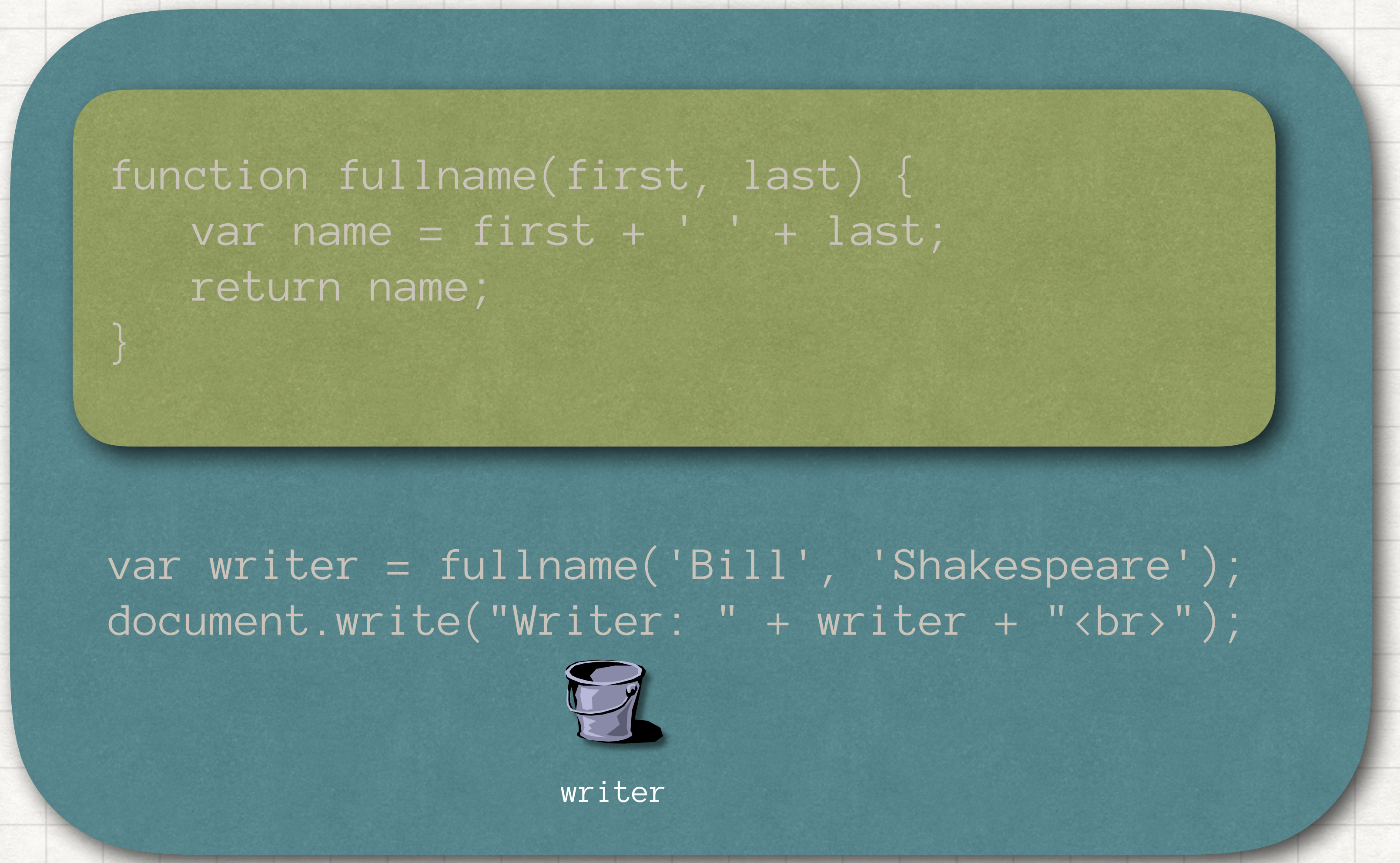
```
function fullname(first, last) {
    var name = first + ' ' + last;
    return name;
}
```

```
var writer = fullname('Bill', 'Shakespeare');
document.write("Writer: " + writer + "<br>");
```

In fact, the variables defined inside the function exist only there. They are created when the function begins executing, and they are erased when the function returns.

VARIABLE SCOPE

We might diagram the relationship between the main program and the function like this. The function exists as a self-contained “space” within the program.



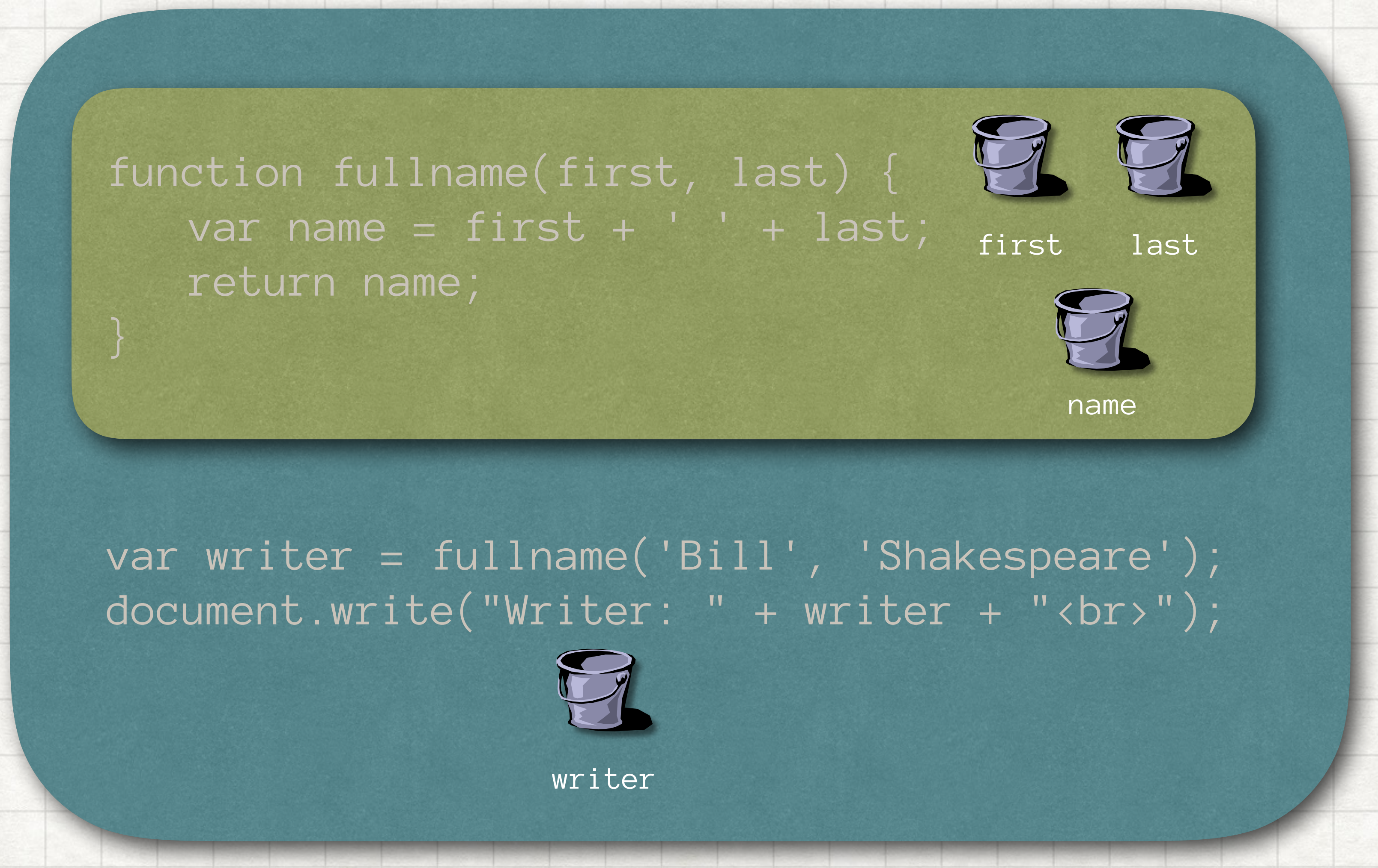
```
function fullname(first, last) {
    var name = first + ' ' + last;
    return name;
}
```

```
var writer = fullname('Bill', 'Shakespeare');
document.write("Writer: " + writer + "<br>");
```

In fact, the variables defined inside the function exist only there. They are created when the function begins executing, and they are erased when the function returns.

VARIABLE SCOPE

But more: the main program (in dark green) *cannot see that the function has any variables at all*. Variables inside the function are invisible to the main program.



```
function fullname(first, last) {
    var name = first + ' ' + last;
    return name;
}

var writer = fullname('Bill', 'Shakespeare');
document.write("Writer: " + writer + "<br>");
```

This dynamic is called **variable scope** (like micro*scope*, tele*scope*). The main program cannot “see” the function’s variables.

Although it seems confusing at first, variable scope turns out to be a Very Good Idea.

The variables inside the function are called *local variables*. Variables that belong to the main program itself are called *global variables*.

OK, that's how it's *supposed* to work.

But Javascript goofed it up.

So we need to fix it.

STRICT MODE

We force Javascript to use proper scoping by telling the browser to run our script in “strict mode.” As long as we declare variables with `var`, all will be well.

```
"use strict";

document.write(name);

function fullname(first, last) {
   var name = first + ' ' + last;
   return name;
}

var writer = fullname('Bill', 'Shakespeare');
document.write("Writer: " + writer + "<br>");

document.write("hello," + name);
```

Notice that “use strict”; is just a string. It’s not even assigned to anything. In older browsers, it’ll simply be interpreted as a “no op”!

STRICT MODE

Now the variable name exists only inside the function. Outside the function, Javascript doesn’t even know it exists:

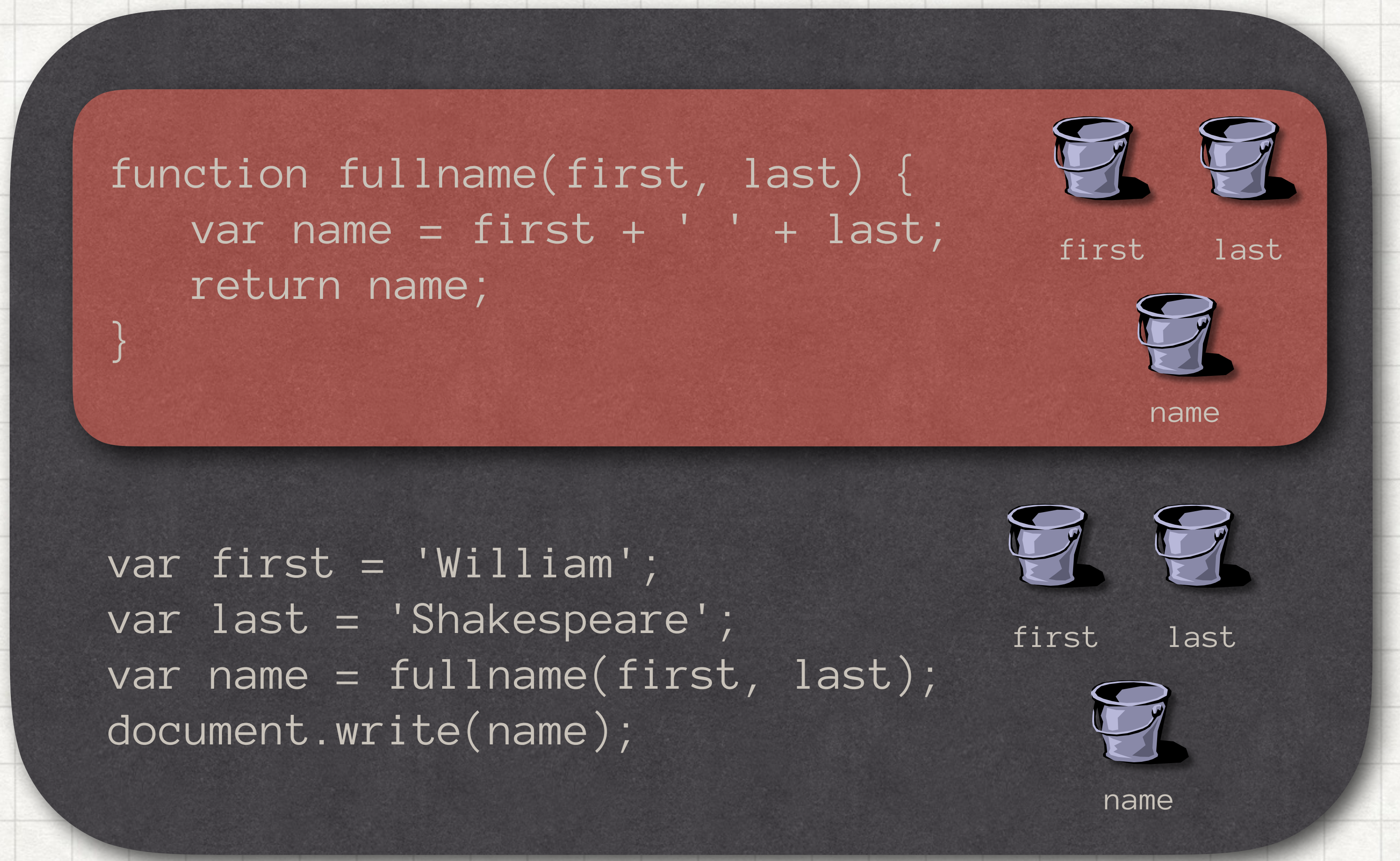
```
"use strict";

document.write(name);

function fullname(first, last) {
   var name = first + ' ' + last;
   return name;
}

var writer = fullname('Bill', 'Shakespeare');
document.write("Writer: " + writer + "<br>");

document.write("hello," + name);
```

VARIABLE SCOPE: A GOOD IDEA

Variable scope prevents data "collisions" between the main program and any functions. What if the function and the program used the same variable names?

Without proper variable scope, we'd have a mess.

Variable scope prevents collisions from happening. In this program, we have two different variables called `first`, two different variables called `last`, and two different variables called `name`.



```
function fullname(first, last) {
    var name = first + ' ' + last;
    return name;
}
```

```
var first = 'William';
var last = 'Shakespeare';
var name = fullname(first, last);
document.write(name);
```

Technically, the variables in the function and in the main program live in different *namespaces*.

STRICT MODE: VARIABLE SCOPE

Let’s experiment with these different spaces with the following script:

```
"use strict";

var name = "Kit Marlowe";
document.write("Before function: " + name + "<br>");

function fullname(first, last) {
    var name = first + ' ' + last;
    document.write("In function: " + name + "<br>");
    return name;
}

var writer = fullname('Bill', 'Shakespeare');
document.write("After function: " + name);
```

DEFAULT VALUES

If we do not send a value for every input parameter, we can instruct Javascript to use a "default" value:

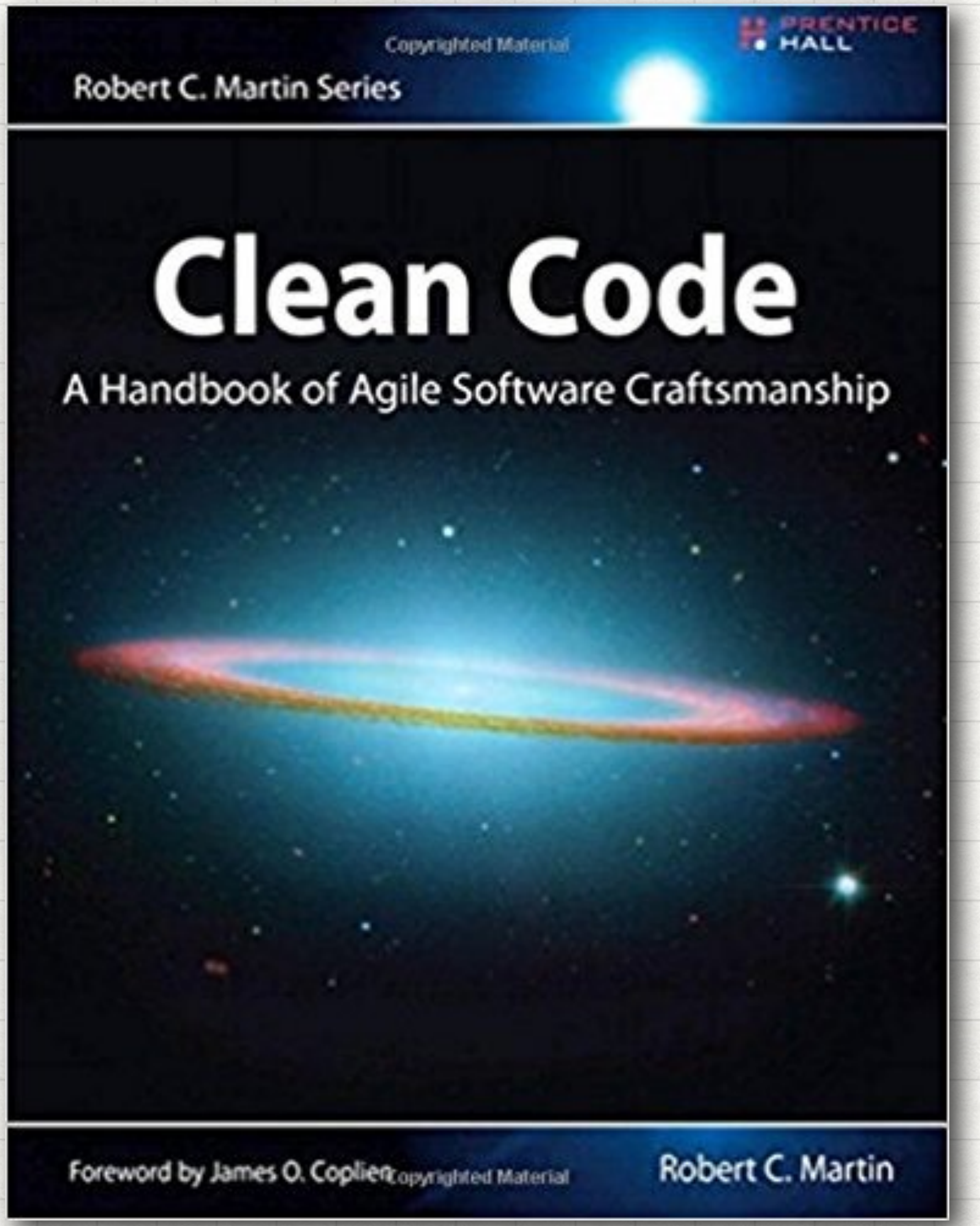
```
function fullname(first, last = 'Shakespeare') {
    var name = first + ' ' + last;
    return name;
}
```

In this case, if I send only a value for the `first` name, Javascript will automatically assign Shakespeare as the value of `last` name.

If I do send a value for `last` name, Javascript will use that instead.

Note: the parameters that have default values must come at *the end of the list!* Otherwise, Javascript would have no idea how to map the inputs to the parameter list.

CODING STYLE



"The following advice has appeared in one form or another for 30 years or more.

Functions should do one thing. They should do it well. They should do it only."

Robert C. Martin: *Clean Code*, p. 35.

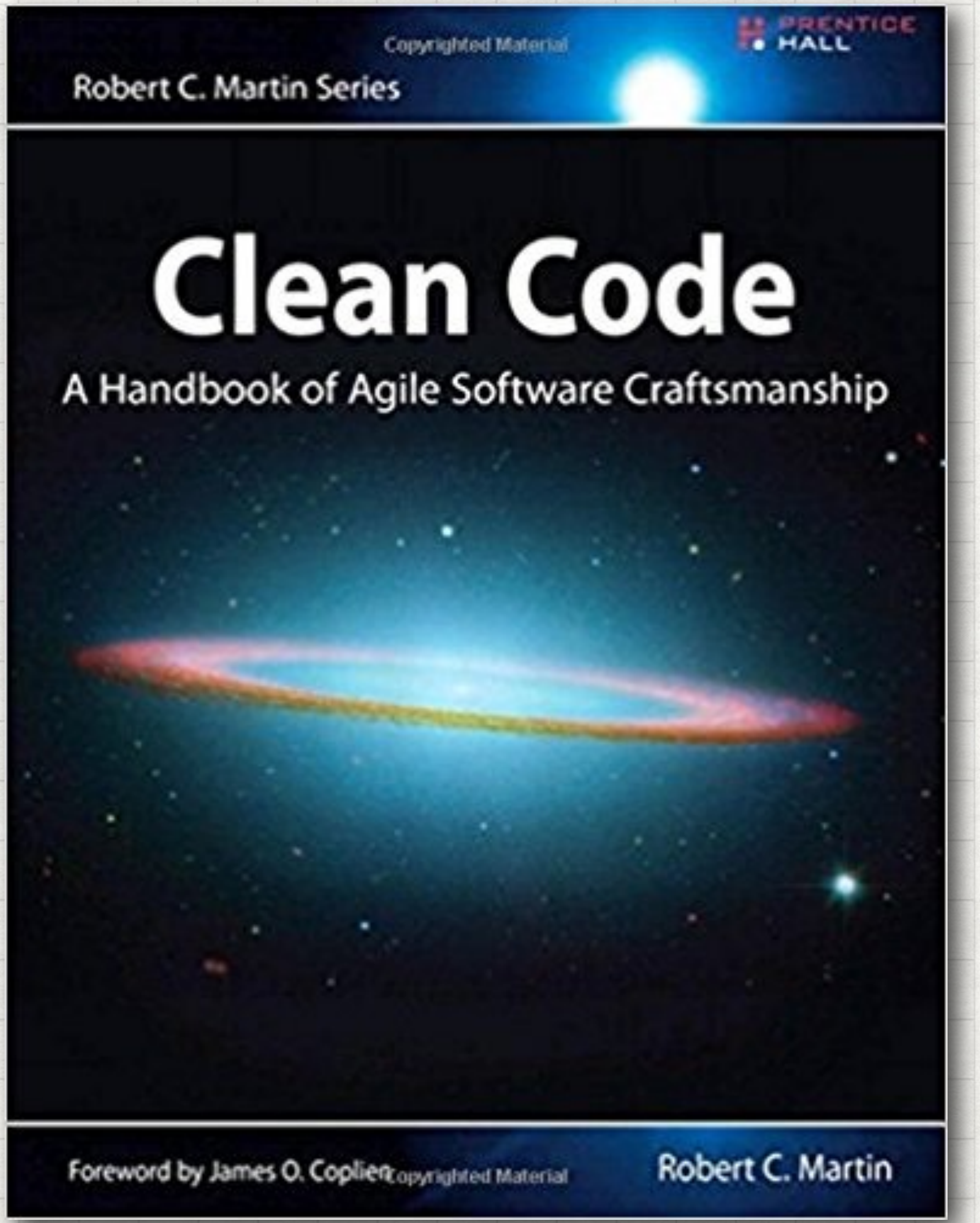
"This also implies that functions should not be large enough to hold nested structures. Therefore, the indent level of a function should not be greater than one or two. This, of course, makes the functions easier to read and understand."

Robert C. Martin: *Clean Code*, p. 35.

CODING STYLE

"The first rule of functions is that they should be small. The second rule of functions is that *they should be smaller than that*."

Robert C. Martin: *Clean Code*, p. 34.


Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

Programming guru Robert "Uncle Bob" Martin says that functions should never be longer than about 6 lines of code.

CODING STYLE

The Stepdown Rule

We want the code to read like a top-down narrative. We want every function to be followed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions. I call this *The Stepdown Rule*.

To say this differently, we want to be able to read the program as though it were a set of *TO* paragraphs, each of which is describing the current level of abstraction and referencing subsequent *TO* paragraphs at the next level down. . . .

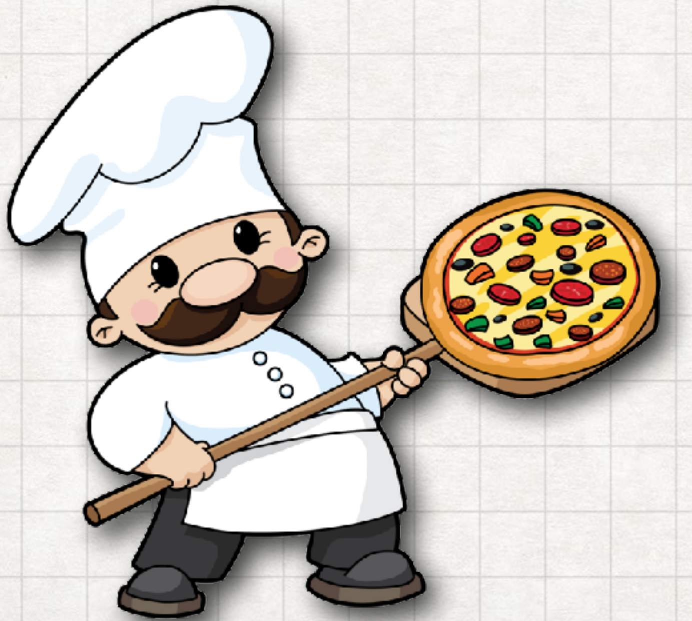
Learning this trick is very important. It is the key to keeping functions short and making sure they do “one thing.” Making the code read like a top-down set of *TO* paragraphs is an effective technique for keeping the abstraction level consistent.

Robert C. Martin: *Clean Code*, p. 37.

FUNCTIONS AS VERBS

Thinking about functions as "to" verbs helps us define their purpose more clearly and helps us see when we need to write "sub-functions."

to mix the dough	(traditional, whole wheat, gluten-free)
	get appropriate ingredients, mix
to roll out the dough	(traditional, thin, pan)
	roll out with pressure, flip in the air
to add toppings	(any combo of ingredients from menu)
	go to fridge, open packages, count, weigh
to bake the pizza	(time and temp)
	check oven temp, set timer
to slice it	(how many)
	square? pie-shaped? how many?
to box it up	(box size)
	small, medium, large, double



FUNCTION RECAP

- A function is, by definition, “a device that groups a set of statements so that they can be run more than once in a program.”
- “Functions should do one thing. They should do it well. They should do it only.” (Robert Martin)
- To run or execute the function is to *call* the function.
- In Javascript, functions must be defined before they can be called.
- Some functions compute a result value based on input data that can be different every time we call the function. We say that the function accepts *arguments* or parameters and *returns* a result.
- Functions minimize *code redundancy*. Copying-and-pasting code is not a good idea. If you’re tempted to copy-and-paste, write a function instead.
- “Functions also provide a tool for splitting systems into pieces that have well-defined roles.” Designing Javascript code with functions allows us to analyze complex problems and break them down into well-defined sub-tasks.
- Robert Martin says functions should be no longer than 6 lines of code. In practice, Quamen thinks 6-10 lines is a good size.

Intro to Javascript

Functions

(Part 1)