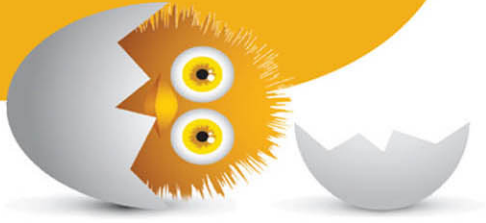
FULL COLOR

JavaScript

ABSOLUTE BEGINNER'S GUIDE

No experience necessary!



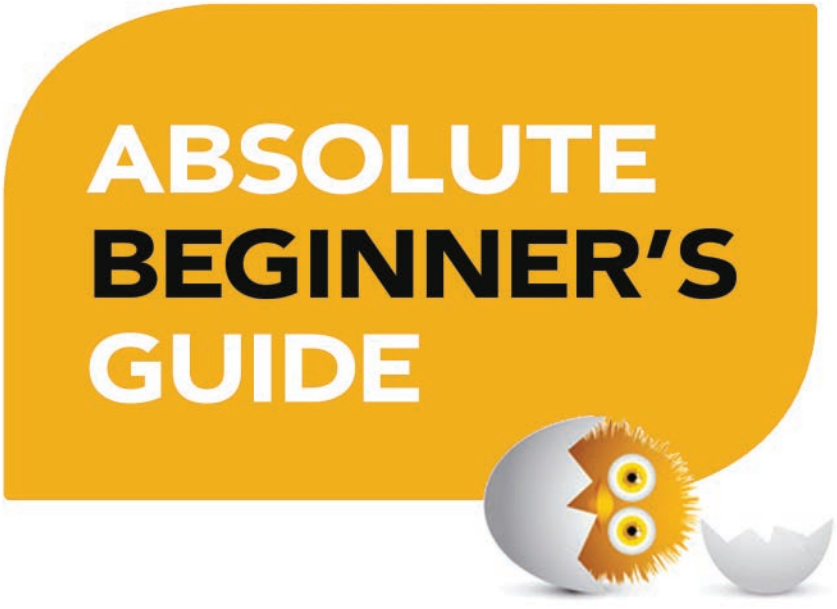
Kirupa Chinnathambi

que

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

JavaScript



Kirupa Chinnathambi

que®

Pearson Education
800 East 96th Street
Indianapolis, Indiana 46240
USA

JavaScript Absolute Beginner's Guide

ISBN-13: 978-0-7897-5806-4
ISBN-10: 0-7897-5806-7

Library of Congress Control Number: 2016939721

Printed in the United States of America

First Printing: July 2016

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Pearson cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearsoned.com.

Acquisitions Editor
Mark Taber

Development Editor
Chris Zahn

Copy Editor
Autumn Spalding

Production Editor
Lori Lyons

Technical Editors
Trevor McCauley
Kyle Murray

This page intentionally left blank

Contents at a Glance

Table of Contents

Dedication

To Meena!

(Who still laughs at the jokes found in these pages despite having read them a bazillion times!)

Acknowledgments

As I found out, getting a book like this out the door is no small feat. It involves a bunch of people in front of (and behind) the camera who work tirelessly to turn my ramblings into the beautiful pages that you are about see. To everyone at Pearson who made this possible, thank you!

With that said, there are a few people I'd like to explicitly call out. First, I'd like to thank Mark Taber for giving me this opportunity, Chris Zahn for patiently answering my numerous questions, and Loretta Yates for helping make the connections that made all of this happen. The technical content of this book has been reviewed in great detail by my long-time friends and online collaborators, Kyle Murray and Trevor McCauley. I can't thank them enough for their thorough (and occasionally, humorous!) feedback.

Lastly, I'd like to thank my parents for having always encouraged me to pursue creative hobbies like painting, writing, playing video games, and writing code. I wouldn't be half the rugged indoorsman I am today without you both.☺

About the Author

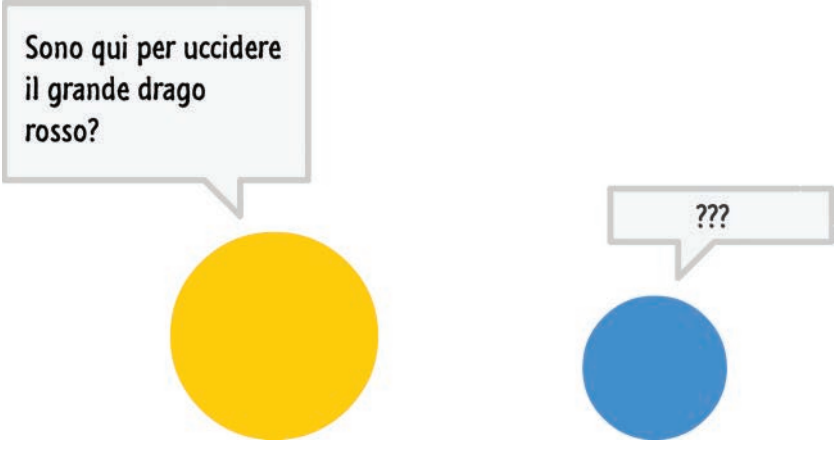
Kirupa Chinnathambi has spent most of his life trying to teach others to love web development as much as he does.

In 1999, before blogging was even a word, he started posting tutorials on **kirupa.com**. In the years since then, he has written hundreds of articles, written a few books (none as good as this one, of course!), and recorded a bunch of videos you can find on YouTube. When he isn't writing or talking about web development, he spends his waking hours helping make the Web more awesome as a Program Manager in Microsoft. In his non-waking hours, he is probably sleeping...or writing about himself in the third person.

You can find him on Twitter (**twitter.com/kirupa**), Facebook (**facebook.com/kirupa**), or e-mail (**kirupa@kirupa.com**). Feel free to contact him anytime.

INTRODUCTION

Have you ever tried learning to read, speak, or write in a language different from the one you grew up with? If you were anything like me, your early attempts probably looked something like the following:



Unless you are Jason Bourne (or Roger Federer), you barely survived learning your first language. This is because learning languages is hard. It doesn't matter if you are learning your first language or a second or third. Being good at a language to a point where you are useful in a non-comical way takes a whole lotta time and effort.

It requires starting with the basics:

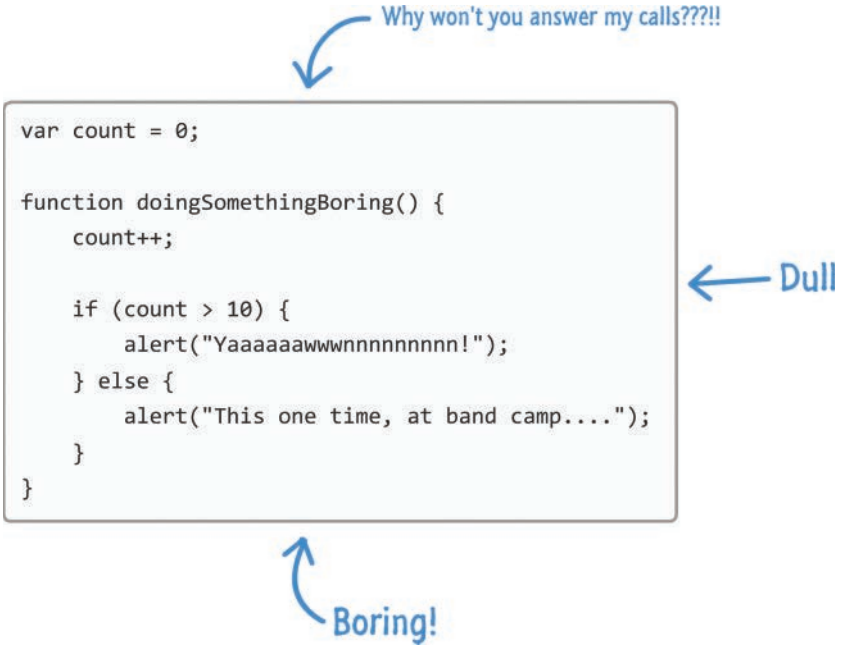
It requires a boatload of practice and patience. It's one of those few areas where there really aren't any shortcuts for becoming proficient.

Parlez-vous JavaScript?

Successfully learning a *programming* language is very similar to how you would approach learning a *real world* language. You start off with the basics. Once you've gotten good at that, you move on to something a bit more advanced. This whole process just keeps repeating itself, and it never really ends. None of us ever truly stop learning. It just requires starting somewhere. To help you with the

"starting somewhere" part is where this book comes in. This book is filled from beginning to end with all sorts of good (and hilarious—I hope!) stuff to help you learn JavaScript.

Now, I hate to say anything bad about a programming language behind its back, but JavaScript is pretty dull and boring:



There is no other way to describe it. Despite how boring JavaScript might most certainly be,[1] it doesn't mean that learning it has to be boring as well.

As you make your way through the book, hopefully you will find the very casual language and illustrations both informative as well as entertaining (infotaining!). All of this casualness and fun is balanced out by deep coverage of all the interesting things you need to know about JavaScript to become better at using it. *By the time you reach the last chapter, you will be prepared to face almost any JavaScript-related challenge head-on without breaking a sweat.*

1. FYI. All grammatical snafus are carefully and deliberately placed—most of the time!

Contacting Me/Getting Help

If you ever get stuck at any point or just want to contact me, post in the forums at: **forum.kirupa.com**.

For non-technical questions, you can also send e-mail to **kirupa@kirupa.com**, tweet to @kirupa, or message me on Facebook (**facebook.com/kirupa**). I love hearing from readers like you, and I make it a point to personally respond to every message I receive.

And with that, flip the page—it's time to get started!

IN THIS CHAPTER

- Learn how functions help you better organize and group your code
- Understand how functions make your code reusable
- Discover the importance of function arguments and how to use them

3

FUNCTIONS

So far, all of the code we've written contained virtually no structure. It was just…there:

```
alert("hello, world!");
```

There is nothing wrong with having code like this. This is especially true if your code is made up of a single statement. Most of the time, though, that will never be the case. Your code will rarely be this simple when you are using JavaScript in the real world for real-worldy things.

To highlight this, let's say we want to display the distance something has traveled (see Figure 3.1).

FIGURE 3.1

The distance something has traveled.

If you remember from school, distance is calculated by multiplying the speed something has traveled by how long it took as shown in Figure 3.2.

FIGURE 3.2

The formula for calculating distance.

The JavaScript version of that will look as follows:

```
var speed = 10;
var time = 5;
alert(speed * time);
```

We have two variables—`speed` and `time`—and they each store a number. The `alert` function displays the result of multiplying the values stored by the `speed` and `time` variables. Quick note: The `*` character (which I threw in there without warning) between two numbers indicates that a multiplication needs to take place. Anyway, as you can see, our JavaScript is a pretty literal translation of the distance equation you just saw.

Let's say we want to calculate the distance for more values. Using only what we've seen so far, our code would look as follows:

```
var speed = 10;
var time = 5;
alert(speed * time);
```

```
var speed1 = 85;
var time1 = 1.5;
alert(speed1 * time1);

var speed2 = 12;
var time2 = 9;
alert(speed2 * time2);

var speed3 = 42;
var time3 = 21;
alert(speed3 * time3);
```

I don't know about you, but this just looks (as Frank Caliendo impersonating Charles Barkley would say) *turrible*.[1] Our code is unnecessarily verbose and repetitive. As I mentioned earlier, when we looked at variables in the previous chapter, repetition makes your code harder to maintain. It also wastes your time.

This entire problem can be solved very easily by using what you'll be seeing a lot of here—**functions**:

```
function showDistance(speed, time) {
    alert(speed * time);
}

showDistance(10, 5);
showDistance(85, 1.5);
showDistance(12, 9);
showDistance(42, 21);
```

Don't worry too much about what this code does just yet. Just know that this smaller chunk of code does everything all those many lines of code did earlier without all of the negative side effects and calories. We'll learn all about functions, and how they do all the sweet things that they do, starting…right…now!

1. Frank Caliendo FTW: **http://bit.ly/kirupaFrankCB**.

What Is a Function?

At a very basic level, a function is nothing more than a wrapper for some code. A function basically

- Groups statements together
- Makes your code reusable

You will rarely write or use code that doesn't involve functions, so it's important that you become familiar with them and learn all about how well they work.

A Simple Function

The best way to learn about functions is to just dive right in and start using them, so let's start off by creating a very simple function. Creating a function is pretty easy and only requires understanding some little syntactical quirks like using weird parentheses and brackets.

The following is an example of what a very simple function looks like:

```
function sayHello() {
    alert("hello!");
}
```

Just having your function isn't enough, though. Your function needs to actually be *called*, and you can do that by adding the following line at the end of your code block:

```
function sayHello() {
    alert("hello!");
}
sayHello();
```

If you type all this in your favorite code editor and preview your page in your browser, you will see *hello!* displayed. The only thing that you need to know right now is that your code works. Let's look at *why* the code works by breaking it up into individual chunks and looking at them in greater detail.

First, you see the `function` keyword leading things off:

```
function sayHello()   {
    alert("hello!");
}
```

This keyword tells the JavaScript engine that lives deep inside your browser to treat this entire block of code as something to do with functions.

After the `function` keyword, you specify the actual name of the function followed by some opening and closing parentheses, `( )`:

```
function sayHello()    {
    alert("hello!");
}
```

Rounding out your function declaration are the opening and closing brackets that enclose any statements that you may have inside:

```
function sayHello()    {
    alert("hello!");
}
```

The final thing is the contents of your function—the statements that make your function actually…functional:

```
function sayHello()    {
    alert("hello!");
}
```

In our case, the content is the alert function that displays a dialog box with the word *hello!* displayed.

The last thing to look at is the function call:

```
function sayHello()    {
    alert("hello!");
}
sayHello();
```

The function call is typically the name of the function you want to call (or invoke) followed again by parentheses. Without your function call, the function you created doesn't do anything. It is the function call that wakes your function up and makes it do things.

Now, what you have just seen is a very simple function. In the next couple of sections, we are going to build on what you've just learned and look at increasingly more realistic examples of functions.

Creating a Function That Takes Arguments

Like I mentioned earlier, the previous `sayHello` example was quite simple:

```
function sayHello()    {
    alert("hello!");
}
```

You call a function, and the function does something. That simplification by itself is not out of the ordinary. All functions work just like that. There are differences, however, in the details on how functions get invoked, where they get their data from, and so on. The first such detail we are going to look at involves functions that take **arguments**.

Let's start with a simple example:

```
alert("my argument");
```

What we have here is your `alert` function. You've probably seen it a few (or a few dozen) times already. As you know, this function simply displays some text that you tell it to show (see Figure 3.3).

FIGURE 3.3

The text "my argument" is displayed as a result of the `alert` function.

Let's look at this a little closer. Between your opening and closing parentheses when calling the alert function, you specify the stuff that needs to be displayed. This "stuff" is more formally known as an argument. The `alert` function is just one of many functions that take arguments, and many functions you create will take arguments as well.

To stay local, within this chapter itself, another function that we briefly looked at that takes arguments is our `showDistance` function:

```
function showDistance(speed, time)    {
    alert(speed * time);
}
```

So, you can tell when a function takes arguments by looking at the function declaration itself:

```
function showDistance(speed, time) {
    ...
}
```

Functions that don't take arguments are easy to identify. They *typically* show up with empty parentheses following their name. Functions that take arguments aren't like that. Following their name and between the parentheses, these functions will contain some information about the quantity of arguments they need, along with some hints about what values your arguments will take.

For `showDistance`, you can infer that this function takes two arguments: the first corresponds to the `speed` and the second corresponds to the `time`.

You specify your arguments to the function as part of the function call:

```
function showDistance(speed, time) {
    alert(speed * time);
}
showDistance(10, 5);
```

In our case, we call `showDistance` and specify the values we want to pass to the function inside the parentheses.

```
showDistance(10, 5);
```

Functions that take arguments, however, contain some information about the quantity of arguments they need in the parentheses following their name, along with some hints about what values your arguments will take. To emphasize this, let's look at Figure 3.4.

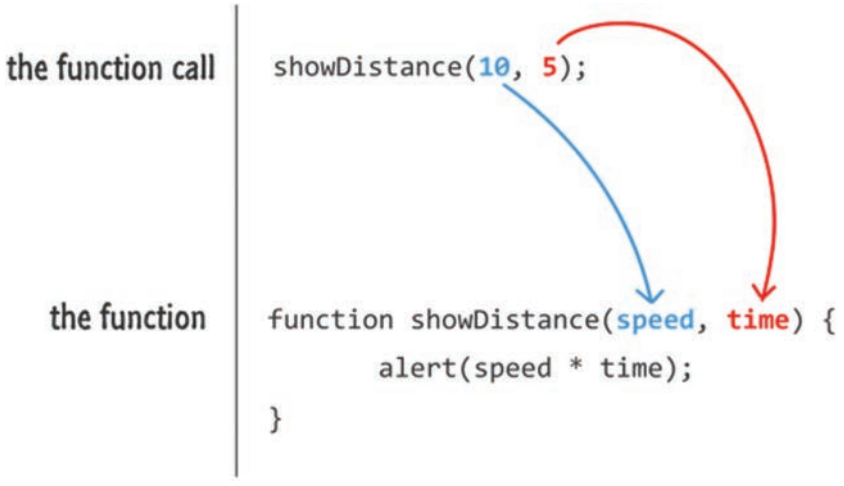


FIGURE 3.4

Order matters.

When the `showDistance` function gets called, the 10 corresponds to the `speed` argument, and the 5 corresponds to the `distance` argument. That mapping is entirely based on order.

Once the values you pass in as arguments reach your function, the names you specified for the arguments are treated just like variable names (see Figure 3.5).

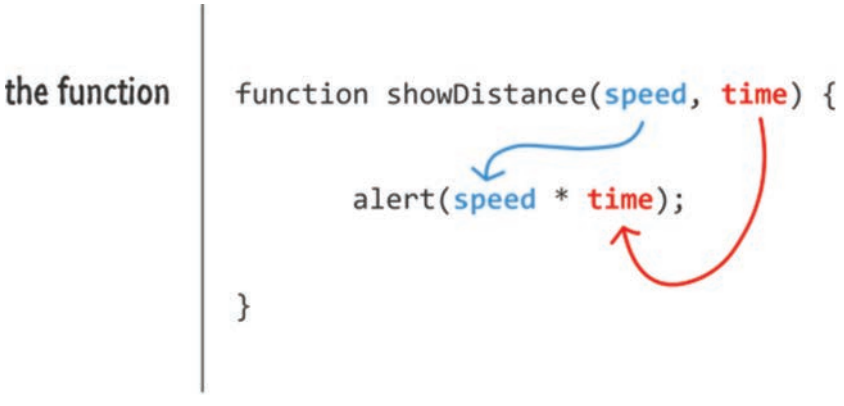


FIGURE 3.5

The names of the arguments are treated as variable names.

You can use these variable names to easily reference the values stored by the arguments inside your function.

NOTE If a function happens to take arguments and you don't provide any arguments as part of your function call, provide too few arguments, or provide too many arguments, things can still work. You can code your function defensively against these cases.

In general, to make the code you are writing clear, just provide the required number of arguments for the function you are calling. Don't complicate things unnecessarily.

Creating a Function That Returns Data

The last function variant we will look at is one that returns some data back to whatever called it. Here is what we want to do. We have our `showDistance` function, and we know that it looks as follows:

```
function showDistance(speed, time)    {
    alert(speed * time);
}
```

Instead of having our `showDistance` function calculate the distance and display it as an `alert`, we actually want to store that value for some future use. We want to do something like this:

```
var myDistance = showDistance(10, 5);
```

The `myDistance` variable will store the results of the calculation done by the `showDistance` function. There are just a few things you need to know about being able to do something like this.

The Return Keyword

The way you return data from a function is by using the `return` keyword. Let's create a new function called `getDistance` that looks identical to `showDistance` with the only difference being what happens when the function runs to completion:

```
function getDistance(speed, time)    {
    var distance = speed * time;
    return distance;
}
```

Notice that we are still calculating the distance by multiplying `speed` and `time`. Instead of displaying an alert, we return the distance (as stored by the `distance` variable).

To call the `getDistance` function, you can just call it as part of initializing a variable:

```
var myDistance = showDistance(10, 5);
```

When the `getDistance` function gets called, it gets evaluated and returns a numerical value that then becomes assigned to the `myDistance` variable. That's all there is to it.

Exiting the Function Early

Once your function hits the `return` keyword, it stops everything it is doing at that point, returns whatever value you specified to whatever called it, and exits:

```
function getDistance(speed, time)    {
    var distance = speed * time;
    return distance;

    if (speed < 0)    {
        distance *= -1;
    }
}
```

Any code that exists after your `return` statement will not be reached, such as the following highlighted lines:

```
function getDistance(speed, time) {
    var distance = speed * time;
    return distance;

    if (speed < 0) {
        distance *= -1;
    }
}
```

It will be as if that chunk of code never even existed. In practice, you will use the `return` statement to terminate a function after it has done what you wanted it to do. That function could return a value to the caller like you saw in the previous examples, or that function could simply exit:

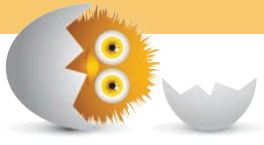
```
function doSomething()    {
    // do something
    return;
}
```

Using the `return` keyword to return a value is optional. The `return` keyword can be used by itself as you see here to just exit the function.

THE ABSOLUTE MINIMUM

Functions are among a handful of things that you will use in almost every single JavaScript application. They provide the much sought-after capability to help make your code reusable. Whether you are creating your own functions or using the many functions that are built into the JavaScript language, you will simply not be able to live without them.

What you have seen so far are examples of how functions are commonly used. There are some advanced traits that functions possess that I did not cover here. Those uses will be covered in the future…a distant future. For now, everything you've learned will take you quite far when it comes to understanding how functions are used in the real world.



TIP Just a quick reminder for those of you reading these words in the print or e-book edition of this book: If you go to **www.quepublishing.com** and register this book, you can receive free access to an online Web Edition that not only contains the complete text of this book but also features a short, fun interactive quiz to test your understanding of the chapter you just read.

If you're reading these words in the Web Edition already and want to try your hand at the quiz, then you're in luck – all you need to do is scroll down!

This page intentionally left blank

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W