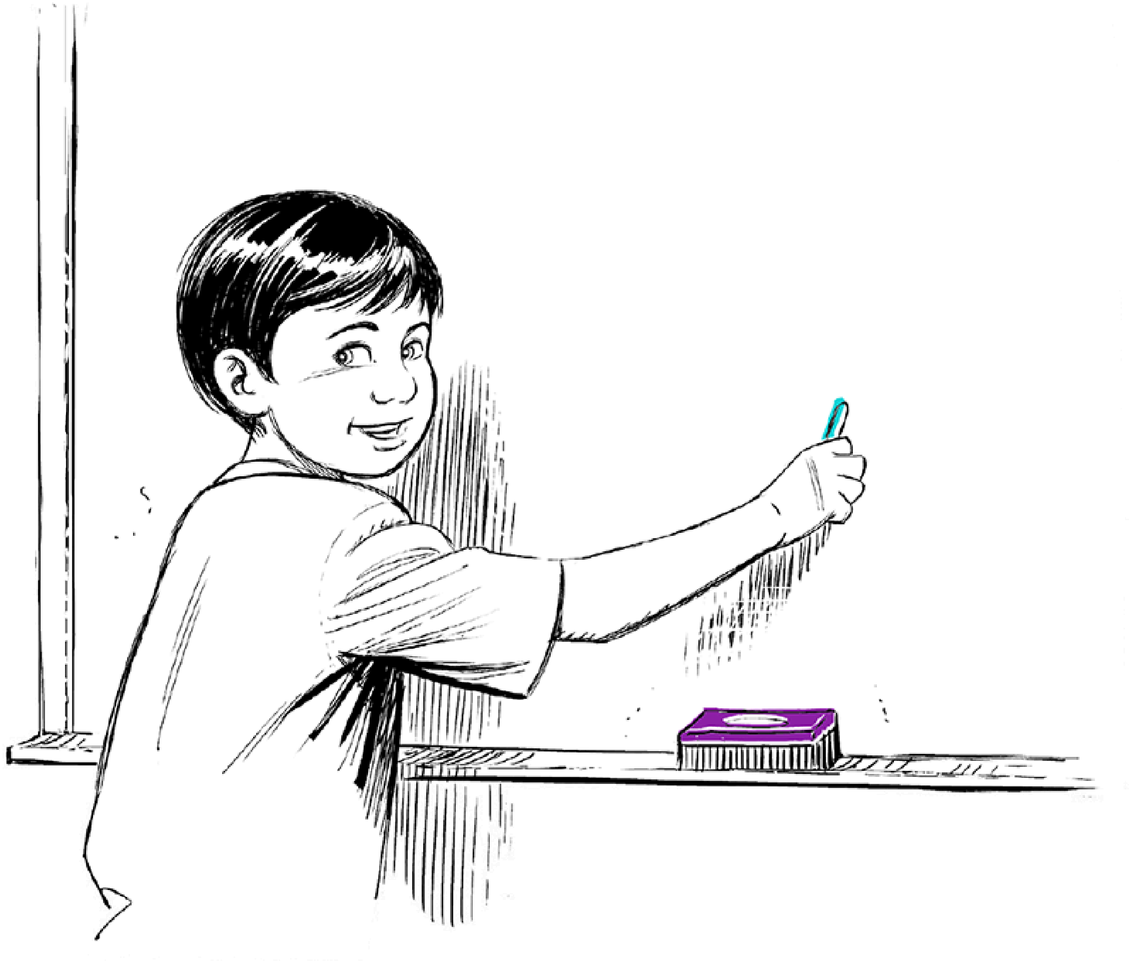
Chapter 3. Introducing Functions: *Getting functional*

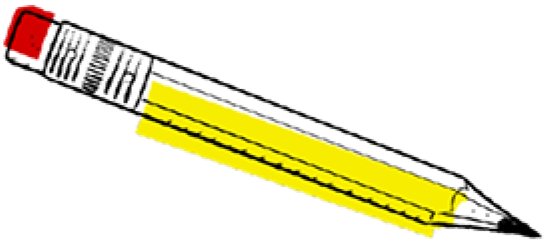


Get ready for your first superpower. You've got some programming under your belt; now it's time to really move things along with **functions**. Functions give you the power to write code that can be applied to all sorts of different circumstances, code that can be **reused** over and over, code that is much more **manageable,** code that can be **abstracted** away and given a simple name so you can forget all the complexity and get on with the important stuff. You're going to find not only that functions are your gateway from scripter to programmer, but that they're the key to the JavaScript programming style. In this chapter, we're going to start with the basics—the mechanics, the ins and outs of how functions really work—and then you'll keep honing your function skills throughout the rest of the book. So, let's get a good foundation started, *now*.

NOTE

More on this as we progress through the book.

SHARPEN YOUR PENCIL



Do a little analysis of the code below. How does it look? Choose among the options at the bottom of the page, or write in your own analysis.

```
let dogName = "rover";
let dogWeight = 23;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spot";
dogWeight = 13;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spike";
dogWeight = 53;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
```

□ A. The code seems very redundant.

□ B. If we want to change the display of the output, or add another weight for dogs, this is going to require a lot of reworking.

□ C. Looks tedious to type in!

□ D. Not the most readable code I’ve ever seen.

□ E. ______________________________

____

→ **Solution in “Sharpen your pencil Solution”**

What’s wrong with the code, anyway?

We just looked at some code that got used *over and over*. What’s wrong with that? Well, at face value, nothing. After all, it works, right? Let’s take a closer look at the code in question:

```
let dogName = "rover";
let dogWeight = 23;
if (dogWeight > 20) {
    console.log(dogName + " says WOOF WOOF");
} else {
    console.log(dogName + " says woof woof");
}
 ...
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
    console.log(dogName + " says WOOF WOOF");
} else {
    console.log(dogName + " says woof woof");
}
```

What we're doing here is comparing the dog's weight to 20, and if it's greater than 20, we're outputting a big WOOF WOOF. If it's less than 20, we're outputting a smaller woof woof.

And this code is...d'oh! It's doing EXACTLY the same thing. And so on, many times over in the rest of the code.

```
dogName = "spike";
dogWeight = 53;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
```

Sure, this code looks innocent enough, but it's tedious to write, a pain to read, and will be problematic if it needs to change over time. That last point will ring true more and more as you gain experience in programming.All code changes over time, and the code above is a nightmare waiting to happen because the same logic is repeated over and over, so if you need to change that logic, you'll have to change it in multiple places. And the bigger the program gets, the more changes you'll have to make, leading to more opportunities for mistakes. What we really want is a way to take redundant code like this and put it in one place where it can be easily reused whenever we need it.

BRAIN POWER

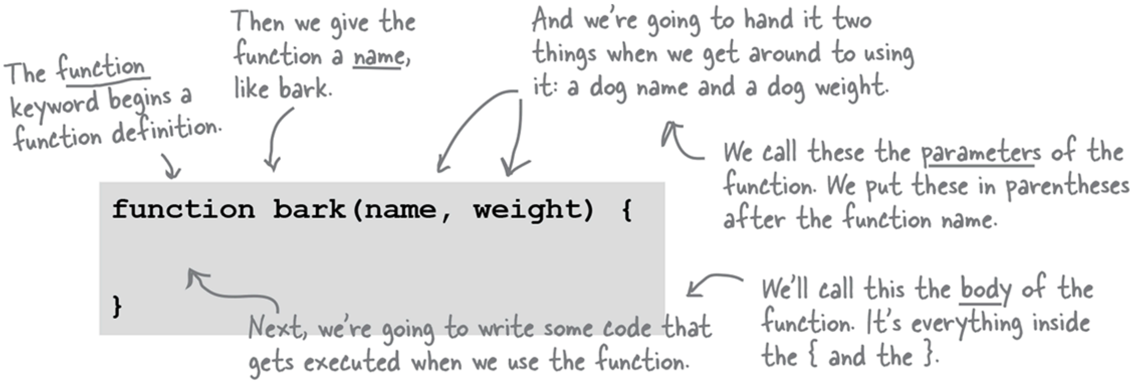
How can we improve this code? Take a few minutes to think of a few possibilities. Does JavaScript have something that could help?

If only I could find a way to **reuse** code so that anytime I needed it, I could just **use** it rather than **retyping** it. And a way to give it a nice memorable **name** so that I could remember it. And a way to make changes in just **one** place instead of many if something changes. That would be dreamy. But I know it's just a fantasy...

By the way, did we happen to mention FUNCTIONS?

Meet *functions*. JavaScript functions allow you to take a bit of code, give it a name, and then refer to it over and over whenever you need it. That sounds like just the medicine we're looking for.

Say you're writing some code that does a lot of "barking" (like in the previous exercise). If your code is dealing with a big dog, then the bark is a big "WOOF WOOF." And if it's a small dog, the bark is a tiny "woof woof." You're going to need to use this barking functionality many times in your code. Let's write a `bark` function you can use over and over:



```
function bark(name, weight) {


}
```

The function keyword begins a function definition.

Then we give the function a name, like bark.

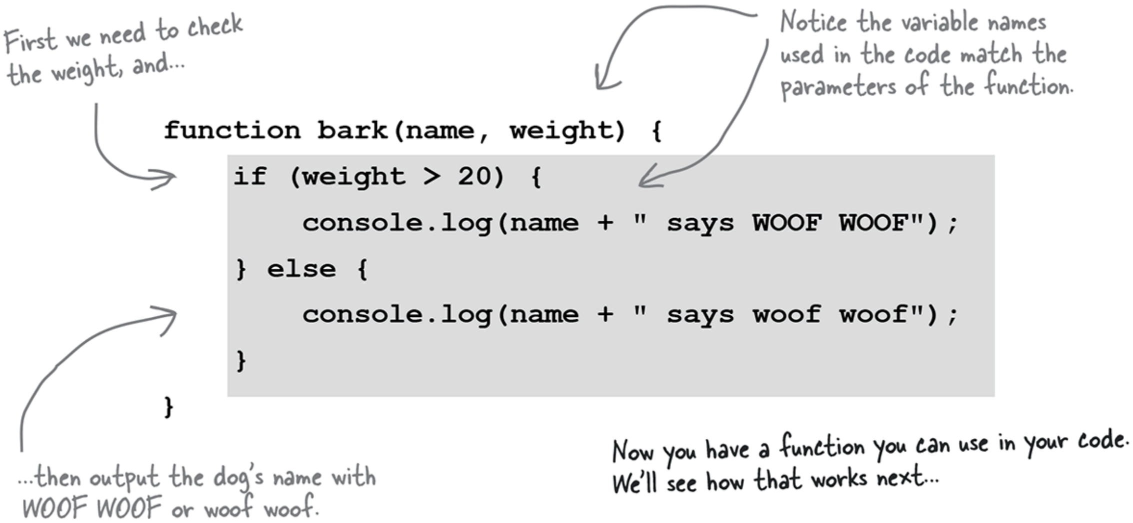
And we're going to hand it two things when we get around to using it: a dog name and a dog weight.

We call these the parameters of the function. We put these in parentheses after the function name.

We'll call this the body of the function. It's everything inside the { and the }.

Next, we're going to write some code that gets executed when we use the function.

Now we need to write the code for the function. Our code will check the weight and output the appropriate-sized bark:



```
function bark(name, weight) {
    if (weight > 20) {
        console.log(name + " says WOOF WOOF");
    } else {
        console.log(name + " says woof woof");
    }
}
```

First we need to check the weight, and...

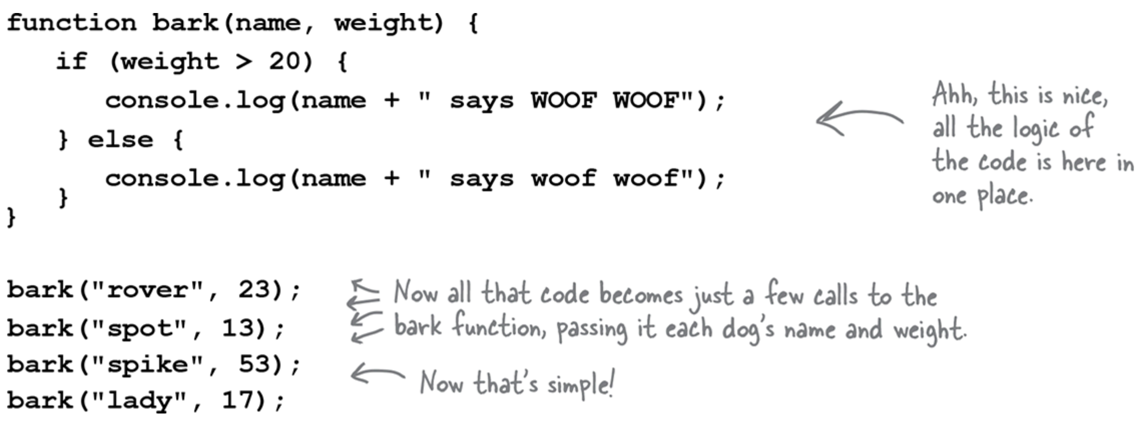
Notice the variable names used in the code match the parameters of the function.

...then output the dog's name with WOOF WOOF or woof woof.

Now you have a function you can use in your code. We'll see how that works next...

Okay, but how does it actually work?

First, let’s rework our code using the new function, `bark`:



```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}

bark("rover", 23);
bark("spot", 13);
bark("spike", 53);
bark("lady", 17);
```

Ahh, this is nice, all the logic of the code is here in one place.

Now all that code becomes just a few calls to the bark function, passing it each dog’s name and weight.

Now that’s simple!

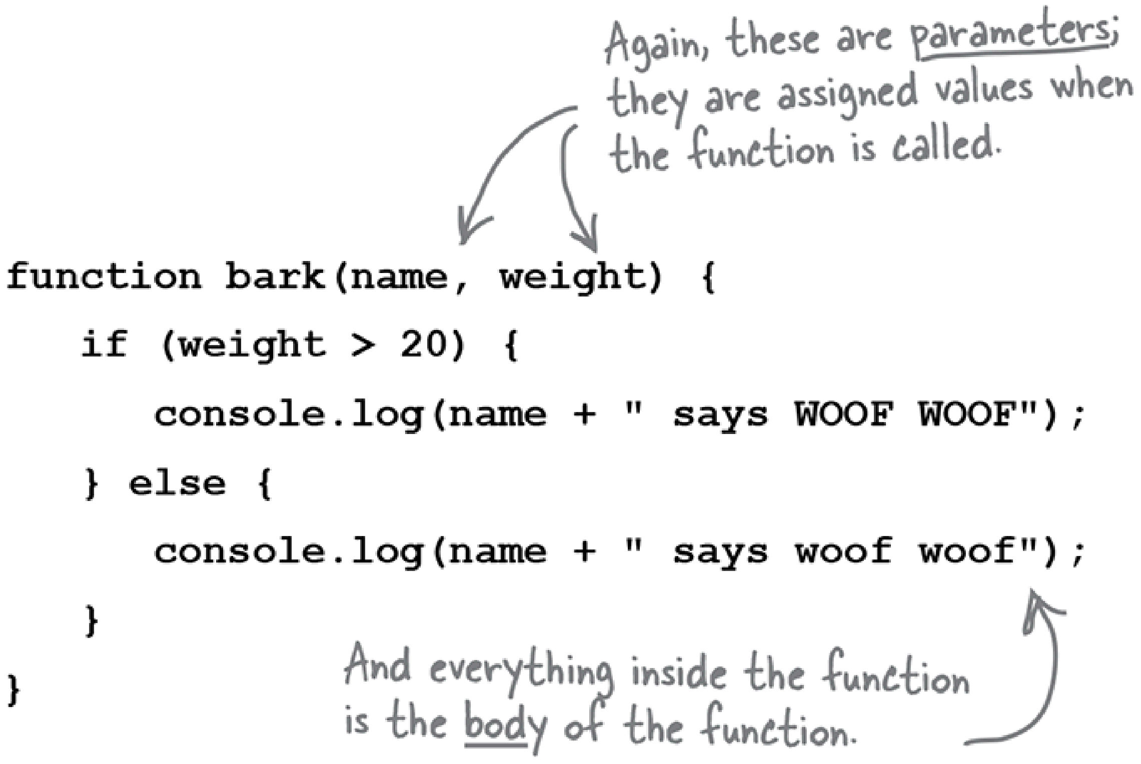
Wow, that’s a lot less code—and it’s so much more readable to your co-worker who needs to go into your code and make a quick change. We’ve also got all the logic in one convenient location.

But how exactly does it all come together and actually work? Let’s go through it step by step.

First we have the function.

So, we’ve got the `bark` function right at the top of the code. The browser reads this code, sees it’s a function, and then takes a look at the statements in the body. The browser knows it isn’t executing the function statements now; it’ll wait until the function is called from somewhere else in the code.

Notice too that the function is *parameterized*, meaning it takes a dog’s name and weight when it is called. That allows you to call this function for as many different dogs as you like. Each time you do, the logic applies to the name and weight you pass to the function call.



Again, these are parameters; they are assigned values when the function is called.

```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}
```

And everything inside the function is the body of the function.

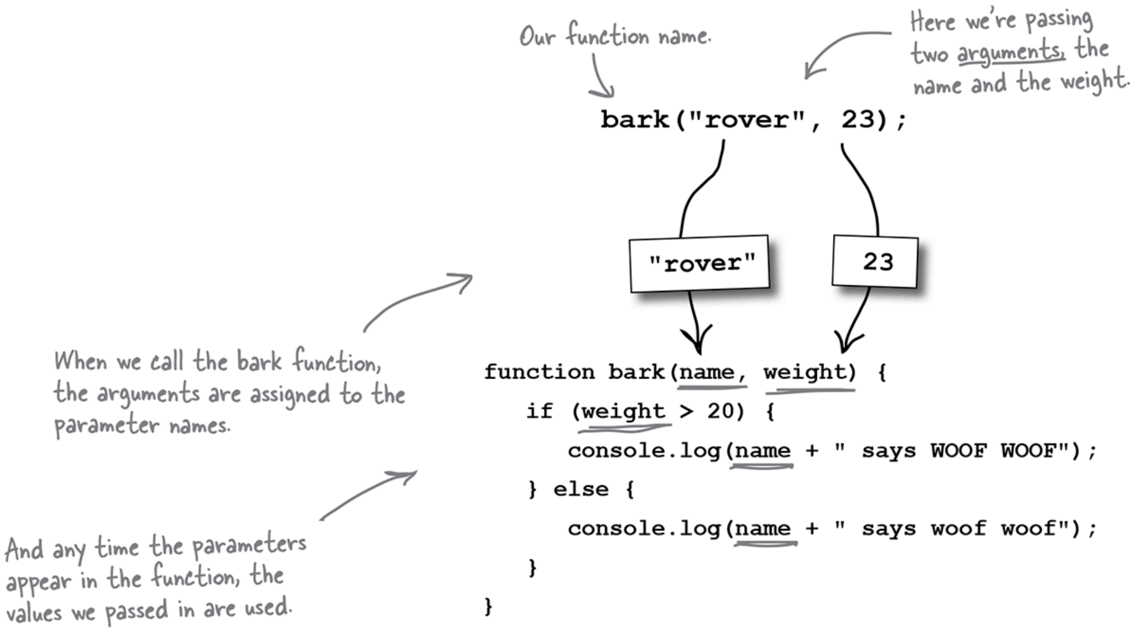
Now let’s call the function.

To call, or *invoke*, a function, just use its name, followed by an open parenthesis, then any values you need to pass it (separated by commas), and finally a closing parenthesis. The values in the parentheses are *arguments*. For the `bark` function, we need two arguments: the dog’s name and the dog’s weight.

Here’s how the call works.

NOTE

“Invoking a function” is just a fancy way of saying “calling a function.” Feel free to mix and match, especially when your new boss is around.



Our function name.

Here we're passing two arguments, the name and the weight.

```
bark("rover", 23);
```

When we call the bark function, the arguments are assigned to the parameter names.

And any time the parameters appear in the function, the values we passed in are used.

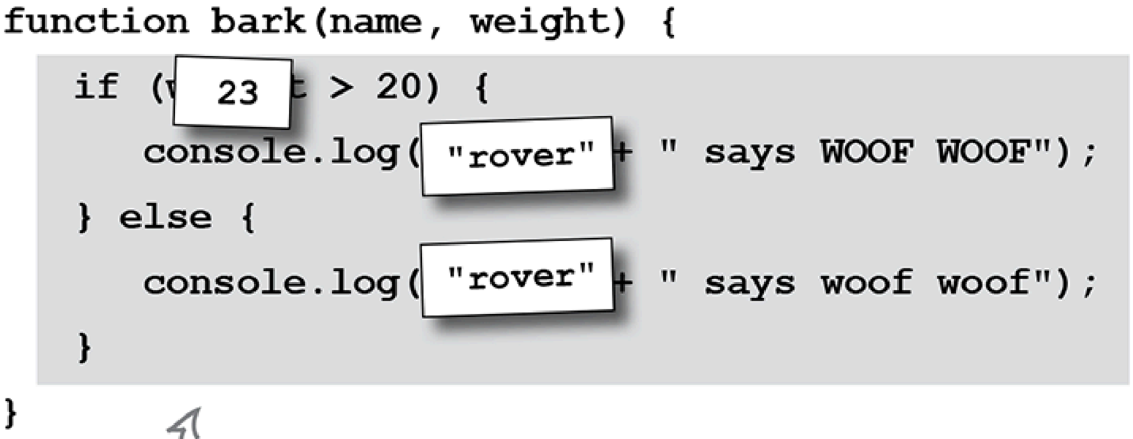
```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}
```

After you call the function, the body of the function does all the work.

Once we know the value for each parameter—like `name` is “rover” and `weight` is 23—we’re ready to execute the function body.

Statements in the function body are executed from top to bottom, just like all the other code you’ve been writing. The only difference is that the parameter names `name` and `weight` have been assigned the values of the arguments you passed into the function.

After we've assigned the argument values to the parameter names, we execute the statements in the body.

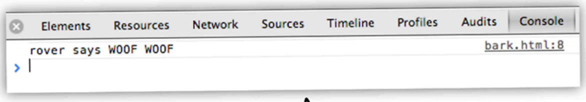


```
function bark(name, weight) {
    if (23 > 20) {
        console.log("rover" + " says WOOF WOOF");
    } else {
        console.log("rover" + " says woof woof");
    }
}
```

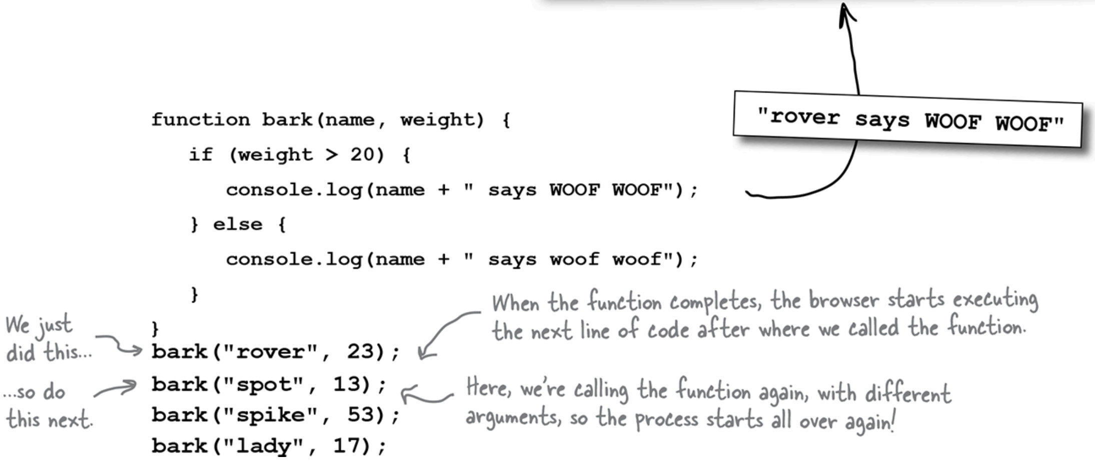
The parameters act like variables in the body, which have been assigned the values of the arguments you passed in.

And when it's done... The logic of the body has been carried out (and, in this example, you'll see that Rover, being 23 pounds, says "WOOF WOOF"), and the function is done. After the function completes, control is returned to the statement following our call to `bark`.

Use your browser's Developer Tools to access the console so you can see the output of bark.



```
rover says WOOF WOOF                                   bark.html:8
>
```



```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}
bark("rover", 23);
bark("spot", 13);
bark("spike", 53);
bark("lady", 17);
```

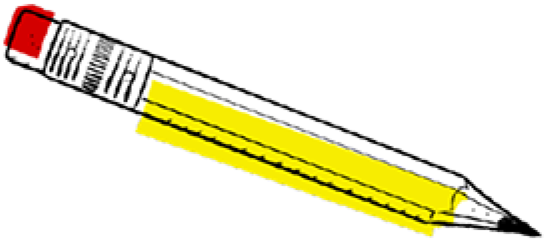
We just did this...

...so do this next.

When the function completes, the browser starts executing the next line of code after where we called the function.

Here, we're calling the function again, with different arguments, so the process starts all over again!

SHARPEN YOUR PENCIL



We’ve got some more calls to bark below. Next to each call, write what you think the output should be, or if you think the code will cause an error. Check your answers at the end of the chapter before you go on.

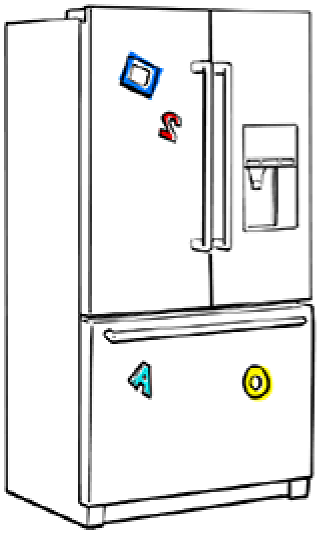
```
bark("juno", 20); ______________________________

bark("scottie", -1); ___________________________

bark("dino", 0, 0); ____________________________

bark("fido", "20"); ____________________________

bark("lady", 10); ______________________________

bark("bruno", 21); _____________________________
```

Write what you think the console log will display here.

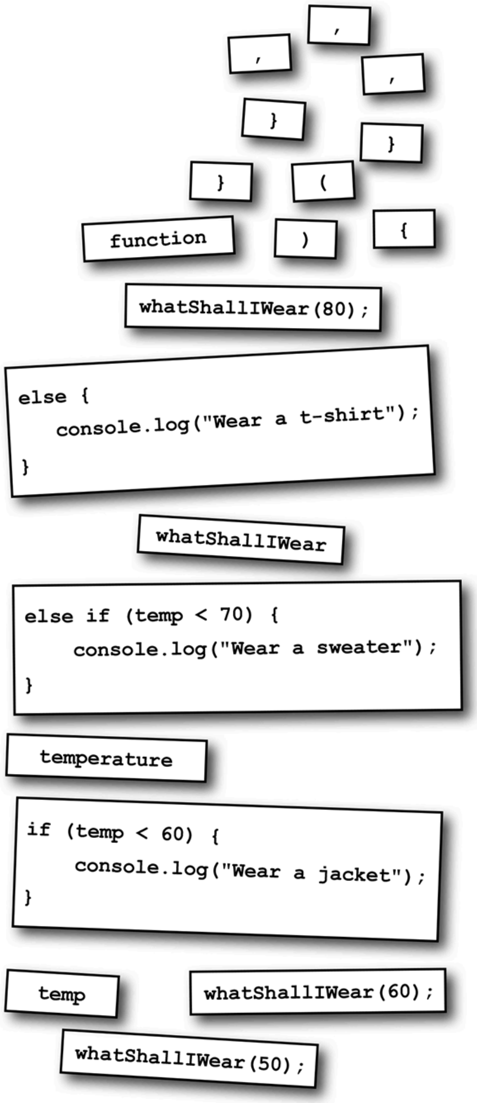
Hmm, any ideas what these do?

Solution in “Sharpen your pencil Solution”

CODE MAGNETS



This working JavaScript code is all scrambled up on the fridge. Can you reorder the code snippets to make a working program that produces the output listed below? Note that there may be some extra code on the fridge, so you may not use all the magnets. Check your answer at the end of the chapter.



```
,
,
,
}
}
}
(
function
)
{
whatShallIWear(80);
```

```
else {
   console.log("Wear a t-shirt");
}
```

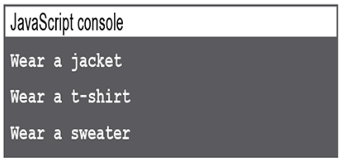
```
whatShallIWear
```

```
else if (temp < 70) {
    console.log("Wear a sweater");
}
```

```
temperature
```

```
if (temp < 60) {
    console.log("Wear a jacket");
}
```

```
temp
whatShallIWear(60);
whatShallIWear(50);
```



JavaScript console

```
Wear a jacket
Wear a t-shirt
Wear a sweater
```

We're using this to represent a generic console.

Solution in “Code Magnets Solution”

This week's interview: The intimate side of functions...

Head First: Welcome, Function! We're looking forward to digging in and finding out what you're all about.

Function: Glad to be here.

Head First: Now, we've noticed many JavaScript newbies tend to ignore you. They just get in and write their code, line by line, top to bottom, no functions at all. Are you really needed?

Function: Those newbies are missing out. That's unfortunate, because I'm powerful. Think about me like this: I give you a way to write code once and then reuse it over and over.

Head First: Well, excuse me for saying this, but if you're just giving them the ability to do the same thing, over and over...that's a little boring isn't it?

Function: No no, functions are parameterized—in other words, each time you use the function, you pass it arguments so it can compute something that's relevant to what you need.

Head First: Err, example?

Function: Let's say you need to show a user how much the items in his shopping cart are going to cost, so you write a function named `computeShoppingCartTotal`. I can compute the amount for any shopping cart you give me.

Head First: If you're so great, why aren't more new coders using you?

Function: That's not even a true statement; they use me all the time! `alert`, `console.log`, `prompt`, `Math.random`...it's hard to write anything meaningful without using functions. It's not so much that new users don't use functions, they just aren't defining *their own* functions.

Head First: Well, right, `alert` and `prompt`, those make sense, but take `Math.random`—that doesn't look quite like a function.

Function: `Math.random` is a function, but it happens to be attached to another powerful thing new coders don't make a lot of use of: *objects*.

Head First: Oh yes, objects. I believe our readers are learning about those in a later chapter.

Function: Fair enough, I'll save my breath on that one for later.

Head First: Now, this argument/parameter stuff all seems a little confusing.

Function: Think about it like this: each parameter acts like a variable throughout the body of the function. When you call the function, each value you pass in is assigned to a corresponding parameter.

Head First: And arguments are what?

Function: Oh, that's just another name for the values you pass into a function...they're the arguments of the function call.

Head First: Well, you don't seem all that great; I mean, okay, you allow me to reuse code, and you have this way of passing values into parameters. Is that it? I don't get the mystery around you.

Function: Oh, that's just the basics. There's so much more: I can return values, I can masquerade around your code anonymously, I can do a neat trick called closures, and I have an *intimate* relationship with objects.

Head First: Ohhhhh REALLY?! Can we get an exclusive on that relationship for our next interview?

Function: We'll talk...

What can you pass to a function?

When you call a function, you pass it arguments, and those arguments then get matched up with the parameters in the function definition. You can pass any JavaScript value as an argument, like a string, a boolean, or a number:

Pass any JavaScript value as an argument.

Each argument is passed to its corresponding parameter in the function.

```
saveMyProfile("krissy", 1991, 3.81, false);

function saveMyProfile(name, age, GPA, newuser) {
    if (age <= 17) {
        // code for handling a child
    }
   // rest of code for this function here
}
```

And each parameter acts as a variable within the function.

You can also pass variables as arguments, and that's often the more common case. Here's the same function call using variables:

```
    let student = "krissy";
    let year = 2024;
    let GPA = 381/100;
    let status = "existing user";
    let isNewUser = (status == "new user");
    saveMyProfile(student, year, GPA, isNewUser);
```

Now, each of the values we're passing is stored in a variable. When we call the function, the variable's values are passed as the arguments.

So, in this case we're passing the value in the variable student, "krissy", as the argument to the name parameter.

And we're also using variables for these other arguments.

And you can even use expressions as arguments:

```
let student = "krissy";
let status = "existing user";
let year = 2024;

saveMyProfile(student, year, 381/100, status == "new user");
```

In each case we first evaluate the expression to a value, and then that value is passed to the function.

Yes, even these expressions will work as arguments!

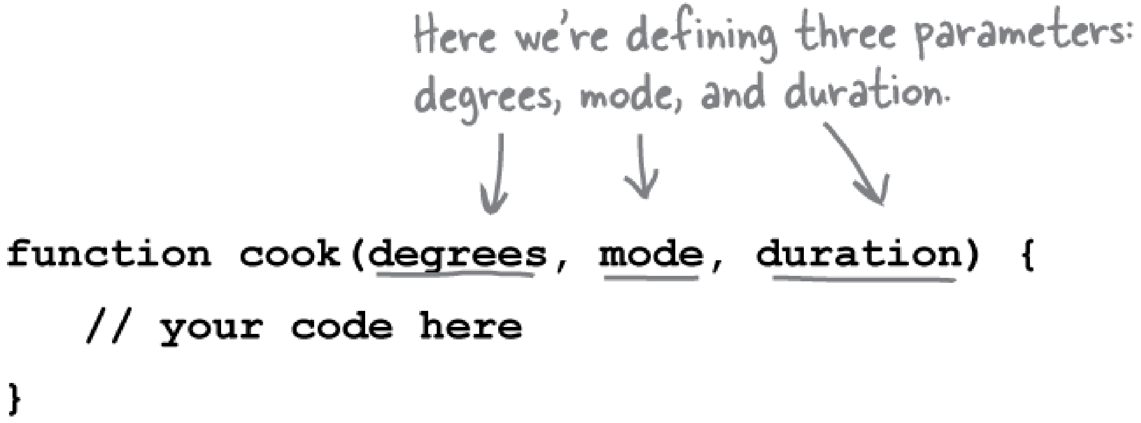
We can evaluate a numeric expression...

...or a boolean expression, like this one that results in passing false to the function.

I'm still not sure I get the difference between a parameter and an argument—are they just two names for the same thing?

No, they're different.

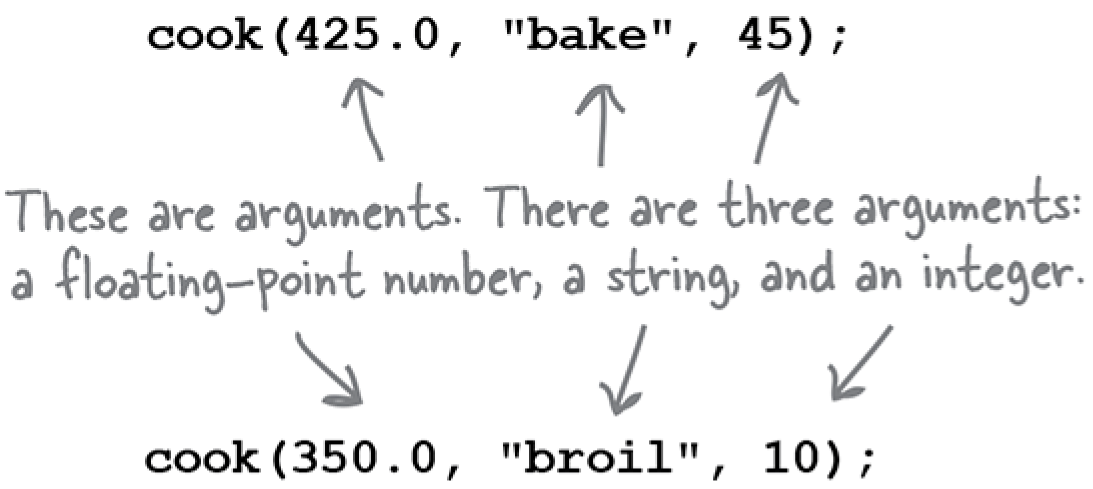
When you define a function, you can *define* it with zero or more *parameters*:



Here we're defining three parameters: degrees, mode, and duration.

```
function cook(degrees, mode, duration) {
   // your code here
}
```

When you call a function, you *call* it with *arguments*:



```
cook(425.0, "bake", 45);
```

These are arguments. There are three arguments: a floating-point number, a string, and an integer.

```
cook(350.0, "broil", 10);
```

So you'll only define your parameters once, but you'll probably call your function with many different arguments.

BRAIN POWER

What does this code output? Are you sure?

```
function doIt(param) {
    param = 2;
}
let test = 1;
doIt(test);
console.log(test);
```

SHARPEN YOUR PENCIL

Below you'll find some JavaScript code, including variables, function definitions, and function calls. Your job is to identify all the variables, functions, arguments, and parameters. Write the names of each in the appropriate boxes on the right. Check your answers at the end of the chapter before you go on.

```
function dogYears(dogName, age) {
    let years = age * 7;
    console.log(dogName + " is " + years + " years old");
}
const myDog = "Fido";
dogYears(myDog, 4);

function makeTea(cups, tea) {
    console.log("Brewing " + cups + " cups of " + tea);
}
let guests = 3;
makeTea(guests, "Earl Grey");

function secret() {
    console.log("The secret of life is 42");
}
secret();

function speak(kind) {
    const defaultSound = "";
    if (kind == "dog") {
        alert("Woof");
    } else if (kind == "cat") {
        alert("Meow");
    } else {
        alert(defaultSound);
    }
}
let pet = prompt("Enter a type of pet: ");
speak(pet);
```

Variables (and constants)

Functions

Built-in functions

Arguments

Parameters

Solution in “Exercise Solution”

JavaScript is pass-by-value

(that means pass-by-copy)

It’s important to understand how JavaScript passes arguments. JavaScript passes arguments to a function using *pass-by-value*. What that means is that each argument is *copied* into the parameter variable. Let’s look at a simple example to see how this works:

Let’s declare a variable age, and initialize it to the value 7.

```
let age = 7;
```

Now let’s declare a function addOne, with a parameter named x, that adds 1 to the value of x.

```
function addOne(x) {
    x = x + 1;
}
```

Now let’s call the function addOne and pass it the variable age as the argument. The value in age is copied into the parameter x.

```
addOne(age);
```

Now the value of x is incremented by one. But remember x is a copy, so only x is incremented, not age.

```
function addOne(x) {
    x = x + 1;
}
```

NOTE

We’re incrementing x.

We’re glad you’re thinking about it. Understanding how JavaScript passes values to functions is important. When an argument is passed to a function, its value is first *copied* and then assigned to the corresponding parameter. It is indeed pretty straightforward, but if you don’t understand this, you can make some wrong assumptions about how functions, arguments, and parameters all work together.

The real impact of pass-by-value is that any changes to a parameter’s value within the function will affect *only the parameter*, not the original variable passed to the function. That’s pretty much it.

But of course, there’s an exception to every rule, and we’re going to have to revisit this topic when we talk about objects, which we’ll get to in a couple of chapters. Don’t worry, though; with a solid understanding of pass-by-value, you’re in good shape to have that discussion.

And for now, just remember that because of pass-by-value, *whatever happens to a parameter in the function, stays in the function.* Kinda like Vegas.

BRAIN POWER REVISITED

Remember this Brain Power from “Brain Power”? Do you think about it differently now, knowing about pass-by-value? Or did you guess correctly the first time?

```
function doIt(param) {
    param = 2;
}
let test = 1;
doIt(test);
console.log(test);
```

NOTE

Remember this Brain Power?

```
function doIt(param) {
    param = 2;
}
const test = 1;
doIt(test);
console.log(test);
```

NOTE

What about this one? It uses const.

Weird Functions

So far you've seen the normal way to use functions, but what happens when you experiment a little by, say, passing too many arguments to a function? Or not enough? Sounds dangerous. Let's see what happens.

EXPERIMENT #1: What happens when we don't pass enough arguments?

Sounds dicey, but all that really happens is each parameter that doesn't have a matching argument is set to undefined. Here's an example:

```
function makeTea(cups, tea) {
  console.log("Brewing " + cups + " cups of " + tea);
}
makeTea(3);
```

EXPERIMENT #2: What happens when we pass too many arguments?

Ah, in this case JavaScript just ignores the extra arguments. Here's an example:

```
function makeTea(cups, tea) {
  console.log("Brewing " + cups + " cups of " + tea);
}
makeTea(3, "Earl Grey", "hey ma!", 42);
```

NOTE

There's actually a way to get at the extra arguments, but we won't worry about that just now...

EXPERIMENT #3: What happens when we have NO parameters?

No worries, many functions have no parameters!

```
function barkAtTheMoon() {
  console.log("Wooooooooooooooo!");
}
barkAtTheMoon();
```

Functions can return things too

You know how to communicate with your functions in one direction; that is, you know how to pass arguments *to functions*. But what about the other way? Can a function communicate back? Let’s check out the `return` statement:

Tracing through a function with a return statement

Now that you know how arguments and parameters work, and how you can return a value from a function, let’s trace through a function call from start to finish to see what happens at every step along the way. Be sure to follow the steps in order.

> First, we declare a constant radius and initialize it to 5.2.
>
> Next, we call the calculateArea function and pass the radius variable as the argument.
>
> The argument is sent to the parameter r, and the calculateArea function begins executing with r containing the value 5.2.

We compute the area of the circle using the value 5.2 in the parameter r.

We return the value of area from the function. This stops the execution of the function and returns the value.

NOTE

Developers often call this “tracing the flow of execution,” or just “tracing.” As you can see, the flow can jump around when you’re calling functions and returning values. Just take it slow, one step at a time.

The value returned from the function is stored in the variable theArea.

Execution continues on the next line.

ANATOMY OF A FUNCTION

Now that you know how to define and call a function, let’s make sure you’ve got the syntax down cold. Here are all the parts of a function’s anatomy:

THERE ARE NO DUMB QUESTIONS

Q: What happens if I mix up the order of my arguments, so that I'm passing the wrong arguments into the parameters?

A: All bets are off; in fact, we'd guess you're pretty much guaranteed either an error at runtime or an incorrect result. Always take a careful look at a function's parameters, so you know what arguments the function expects to be passed in.

Q: Can I pass a constant to a function?

A: You sure can. Its value will be passed to the function, just like with a variable declared with let.

Q: Why don't the parameter names have let or const in front of them? A parameter is a new variable, right?

A: Effectively, yes. The function does all the work of instantiating the variables for you, so you don't need to supply the let or const keyword in front of your parameter names.

Q: What are the rules for function names?

A: The rules for naming a function are the same as the rules for naming a variable. Just like with variables, you'll want to use names that make sense to you when you read them and provide some indication of what the function does, and you can use camel case (e.g., camelCase) to combine words in function names, just like with variables.

Q: What happens if I use the same name for an argument variable as the parameter? Like if I use the name x for both?

A: Even if your argument and parameter have the same name, like x, the parameter x gets a copy of the argument x, so they are two different variables. Changing the value of the parameter x does not change the value of the argument x.

Q: What does a function return if it doesn't have a return statement?

A: A function without a return statement returns `undefined`.

Good catch. Yes and no.

These declarations work exactly the same within a function as they do outside a function, in the sense that you are initializing a new variable to a value. However, the difference between a variable declared *outside a function* and a variable declared *inside a function* is where that variable can be used—in other words, where in your JavaScript code you can reference the variable. If the variable is declared outside a function, then you can use it *anywhere* in your code. If a variable is declared inside a function, then you can use it only *within that function*. This is known as a variable's *scope*. There are two kinds of scope: global and local.

Global and local variables

Know the difference or risk humiliation

You already know that you can declare a variable by using the `let` or `const` keyword and a name anywhere in your script:

```
let avatar;
const levelThreshold = 1000;
```

NOTE

These are global variables and constants; they're accessible everywhere in your JavaScript code.

And you've seen that you can also declare variables inside a function:

But what does it matter? Variables are variables, right? Well, *where* you declare your variables determines *how visible* they are to other parts of your code, and later, understanding how these two kinds of variables operate will help you write more maintainable code (not to mention helping you understand the code of others).

> **If a variable is declared outside a function, it's GLOBAL. If it's declared inside a function, it's LOCAL.**

Another good catch.

There's a long history of using the letter i as the variable you iterate with. This convention developed back in the days when space was limited (like when we used punched cards to write code), and there was an advantage to short variable names. Now it's a convention all programmers understand. You'll also commonly see j, k, and sometimes even x and y used in this manner. However, this is one of the only exceptions to the best practice of choosing meaningful variable names.

Knowing the scope of your local and global variables

Where you define your variables determines their *scope*; that is, where they're visible to your code and where they aren't. Let's look at an example of both locally and globally scoped variables—remember, the variables you define outside a function are globally scoped, and the function variables are locally scoped.

There’s more to the story

Any block can act as the scope of a variable

So far we’ve looked at global variables—variables declared at the top level—and local variables—variables declared inside functions. But we can declare a variable inside any code block, like the code block of an if statement. So how does scope work in that case? Take a look at the function `playTurn`. Here we have three code blocks: a block for the body of the function, a block for the if, and a block for the else. As we’ll see, any of these code blocks can define the scope of a variable. Let’s take a look:

> **Think of a block of code as any set of statements grouped within curly braces, { and }.**

There are only two kinds of scope.

We have global scope for variables declared outside a function, and then we have block scope. For block scope, a variable’s visibility is defined as within a block of code. That block may be an entire function (in which case we often call it “local scope”), or it could be the block of an if or while statement. Either way, a variable declared in a block is defined only within that block.

That said, you’ll often hear programmers refer to variables as either global or local. Just remember that a variable is visible only within the block it’s declared in, and most often, that’s a function.

Don’t forget to declare your locals!

If you use a variable without declaring it first, that variable will be global. That means that even if you use a variable for the first time inside a function (because you meant for it to be local), the variable will actually be global and be available outside the function too (which might cause confusion later). So, don’t forget to declare your locals!

> **If you forget to declare a variable before using it, the variable will always be global (even if the first time you use it is in a function).**

The short lives of variables

When you’re a variable, you work hard and life can be short. That is, unless you’re a global variable, although even with globals, life has its limits. But what determines the life of a variable? Think about it like this:

Globals live as long as the page. A global variable begins life when its JavaScript is loaded into the page. But your global variable’s life ends when the page goes away. Even if you reload the same page, all your global variables are destroyed and then re-created in the newly loaded page.

Local variables typically disappear when your function ends. Local variables are created when your function is first called and live until the function returns (with a value or not). That said, you can take the values of your local variables and return them from the function before the variables meet their digital maker.

NOTE

We say “typically” because there are some advanced ways to retain locals a little longer, but we’re not going to worry about them now.

Local block variables disappear even faster: when the block ends. Local variables in blocks are created when the block begins executing and live until the end of that block (i.e., when you hit the enclosing curly brace, }).

NOTE

This applies to any statement with one or more blocks, like if/else and while statements and other statements with blocks that you haven’t seen yet.

So, there really is NO escape from the page, is there? If you’re a local variable, your life comes and goes quickly, and if you’re lucky enough to be a global, you’re good only as long as the browser doesn’t reload the page.

You “shadow” your global.

Here’s what that means. Say you have a global variable `beanCounter` and you then declare a function, like this:

When you do this, any references to `beanCounter` within the function refer to the local variable and not the global. So, we say the global variable is *in the shadow* of the local variable (in other words, we can’t see the global variable because the local version is in our way).

NOTE

Note that the local and global variables have no effect on each other: if you change one, it does not change the other. They are independent variables.

EXERCISE

Below you'll find some JavaScript code, including variables, function definitions, and function calls. Your job is to identify the variables used in all the arguments, parameters, local variables, and global variables. Write the variable names in the appropriate boxes on the right. Then circle any variables that are shadowed. Check your answers at the end of the chapter.

```
let x = 32;
let y = 44;
const radius = 5;

let centerX = 0;
let centerY = 0;
const width = 600;
const height = 400;

function setup(width, height) {
    centerX = width/2;
    centerY = height/2;
}
function computeDistance(x1, y1, x2, y2) {
    if (x1 != x2 && y1 != y2) {
        let dx = x1 - x2;
        let dy = y1 - y2;
        let d2 = (dx * dx) + (dy * dy);
        let d = Math.sqrt(d2);
        return d;
    } else {
        return 0;
    }
}
function circleArea(r) {
    let area = Math.PI * r * r;
    return area;
}
setup(width, height);
let area = circleArea(radius);
```

```
  let distance = computeDistance(x, y, centerX, centerY);
  alert("Area: " + area);
  alert("Distance: " + distance);
```

Arguments

Parameters

Local or block

Globals

Solution in “Exercise Solution”

Tonight's talk: **Global and Local variables argue over who is most important in a program.**

Global variable:	**Local variable:**
Hey Local, I’m really not sure why you’re here because I can handle any need for a variable a coder might have. After all, I’m visible everywhere!	
	Yes, but using globals everywhere is just bad style. Lots of functions need variables that are local. You know, their own private variables for their own use. Globals can be seen everywhere.
You have to admit that I could replace all of you local variables with global ones and the functions would work just the same.	
	Well, yes and no. If you’re extremely careful, sure. But being that careful is difficult, and if you make a mistake, then you’ve got functions using variables that other functions are using for different purposes. You’d also be littering the program with global variables that you only need inside one function call...that would just make a huge mess.
It wouldn’t have to be a mess. Programmers could just create all the variables they need up at the top of a program, so they’d all be in one place...	
	Yeah, and so what happens if you need to call a function that needs a variable, like, oh I dunno, x, and you can’t remember

Global variable:	**Local variable:**
	what you've used x for before? You'd have to go searching all over your code to see if you've used x anywhere else! What a nightmare.
Well, if you'd use better names, then you might be able to keep track of your variables more easily.	
	And what about parameters? Function parameters are always local. So you can't get around that.
True. But why bother with arguments and parameters if you've got all the values you need in globals?	
	Excuse me, do you hear what you're saying? The whole point of functions is so we can reuse code to compute different things based on different inputs.
But local variables are just so...temporary. You come and go at a moment's notice. Actually, it's even worse; you can declare a variable in a tiny block of code like in an if statement. How silly is that?	
	Face it. It's just good programming practice to minimize the visibility of variables to the extent you can...and globals can get you into real trouble. I've seen JavaScript programs that barely use globals at all!
Not at all? Globals are the mainstay of JavaScript	

Global variable:	Local variable:
programmers!	
	Of inexperienced programmers, sure. But as programmers learn to better structure their code for correctness, maintainability, and just good coding style, they learn to stop using globals except when necessary.
I think I need a drink.	
	They let globals drink? Now, we’re *really* in dangerous territory...

Q: Keeping track of the scope of all these locals, blocks, and globals is confusing, so why not just stick to globals? That's what I've always done.

A: If you're writing code that is complex or that needs to be maintained over a long period of time, then you really have to watch how you manage your variables. When you're overzealous in creating global variables, it becomes difficult to track where your variables are being used (and where you're making changes to your variables' values), and that can lead to buggy code. All this becomes even more important when you're writing code with coworkers or you're using third-party libraries (although if those libraries are written well, they should be structured to avoid these issues).

So, use globals where it makes sense, but use them in moderation, and whenever possible, make your variables local. As you get more experience with JavaScript, you can investigate additional techniques to structure code so that it's more maintainable.

Q: I have global variables in my page, but I'm loading in other JavaScript files as well. Do those files have separate sets of global variables?

A: There is only one global scope, so every file you load sees the same set of variables (and creates globals in the same space). That's why it is so important you be careful with your use of variables to avoid clashes (and reduce or eliminate global variables when you can). We'll look at other techniques for managing your variables later in the book.

Q: If I use the same name for a parameter as I do for a global variable, does the parameter shadow the global?

A: Yes. Just like if you declare a new local variable in a function with the same name as a global, if you use the same name for a parameter as a global, you're also going to shadow the global with that name. It's perfectly fine to shadow a global name as long as you don't want to use the global variable inside your function. But it's a good idea to document what you're doing with comments so you don't get confused later, when you're reading your code.

Q: Do variables I declare at the top level of my function get shadowed by variables in blocks?

A: Yes. Just like a local variable shadows a global variable, a variable declared inside a block will shadow both a local variable with the same name declared at the top of a function and a global variable with the same name.

Q: If I reload a page in the browser, do the global variables all get reinitialized?

A: Yes. Reloading a page is like starting over from scratch as far as the variables are concerned. And if any code was in the middle of executing when you reload the page, any local variables will disappear, too.

Q: Should we always declare our local variables at the top of a function?

A: Just like with global variables, you can declare local variables when you first need to use them in a function. Or you can go ahead and declare them at the top of your function, so someone reading your code can easily find those declarations and get a sense at a glance of all the local variables used within the function. If you delay declaring a variable and then decide to use that variable earlier in the body of the function than you originally anticipated, you might get behavior that you don't expect. JavaScript creates all local variables at the beginning of a function, whether you declare them or not (this is called "hoisting," and we'll come back to it later), but the variables are all undefined until they are assigned a value, which might not be what you want.

Q: Why does everyone complain about the overuse of global variables in JavaScript?

A: Overusing global variables is an issue when your code reaches a level of complexity where you can easily lose track of how variables are being used across your code. You might find you're reusing a global variable name when you didn't mean to. One of the advantages of JavaScript is that it is easy to jump in and start coding, using global variables. As you learn more, it's a best practice to control the use of global variables, but don't stress over it too much in the beginning. As we said, we'll show you more techniques for dealing with this issue later in the book.

Actually, you can put your functions anywhere in your JavaScript file.

JavaScript doesn't care if your functions are declared before or after you use them. For instance, check out this code:

This might seem really odd, especially if you remember that when the browser loads your page, it starts executing the JavaScript from the top to the bottom of your file. But the truth is, JavaScript actually makes two passes over your page: in the first pass it reads all the function definitions, and in the second it begins executing your code. So, that allows you to place functions anywhere in your file.

The Thing-A-Ma-Jig

The Thing-A-Ma-Jig is quite a contraption—it clanks and clunks and even thunks, but what it really does, well, you've got us stumped. Coders claim they know how it works. Can you uncrack the code and find its quirks?

```
function clunk(times) {
    let num = times;
    while (num > 0) {
        display("clunk");
        num = num - 1;
    }
}
```

```
function thingamajig(size) {
    let facky = 1;
    clunkCounter = 0;
    if (size == 0) {
       display("clank");
    } else if (size == 1) {
       display("thunk");
    } else {
        while (size > 1) {
            facky = facky * size;
            size = size - 1;
        }
        clunk(facky);
    }
}

function display(output) {
    console.log(output);
    clunkCounter = clunkCounter + 1;
}
let clunkCounter = 0;
thingamajig(5);
console.log(clunkCounter);
```

NOTE

We recommend passing the Thing-A-Ma-Jig the numbers 0, 1, 2, 3, 4, 5, etc. See if you know what it's doing.

Solution in "The Thing-A-Ma-Jig"

TWO-MINUTE GUIDE TO CODE HYGIENE

There's no place that needs to be well maintained more than your code, and JavaScript can seem pretty loosey-goosey when it comes to organizing your variables and functions. So we've put together a neat little guide for you that makes a few recommendations for those new to coding:

Global variables, right at the TOP!

If you use global variables, and we recommend you don't (once you learn proper techniques not to), it's a good idea to keep your globals grouped together as much as possible, and if they're all up at the top, it's easy to find them. Now, you don't have to do this, but isn't it easier for you and others to locate the variables used in your code if they're all at the top?

Functions like to sit together.

Well, not really; they actually don't care, they're just functions. But if you keep your functions together, it's a lot easier to locate them. As you know, the browser actually scans your JavaScript for the functions before it does anything else, so you can place them at the top or bottom of the file. We often start with our global variables and then put our functions next.

Always declare your variables.

Remember, if you forget to declare your variables inside a function, those variables will be automatically global and may create bugs in your code.

Declare your local variables at the TOP of the function they're in.

Put all your local variable declarations at the beginning of the function body, unless you only need the variable inside one block in the function. This makes them easy to find and ensures they are all declared properly before use.

Use let or const to declare your variables.

For a long time, the only way to declare variables in JavaScript was with `var`. But there were problems with `var`, so a few years ago, JavaScript replaced `var` with `let` and `const`. You don't have to worry about the problems with `var` as long as you always remember to use `let` or `const`.

That's it!

WHO AM I?

A bunch of JavaScript attendees, in full costume, are playing a party game: “Who am I?” They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. Fill in the blank next to each sentence with the name of one attendee.

Tonight’s attendees are:

function, argument, return, scope, local variable, global variable, pass-by-value, parameter, function call, block, Math.random, built-in functions, code reuse

I get passed into a function. ______________________________

I send values back to the calling code. ______________________________

I’m the all-important keyword. ______________________________

I’m what receives arguments. ______________________________

I really mean “make a copy.” ______________________________

I’m everywhere. ______________________________

Another phrase for invoking a function. ______________________________

Example of a function attached to an object. ______________________________

alert and prompt are examples. ______________________________

What functions are great for. ______________________________

Where I can be seen. ______________________________

I’m around when my function is. ______________________________

I’m defined within two curly braces. ______________________________

Solution in “Who am I? Solution”

The case of the attempted robbery not worth investigating

Sherlock finished his phone call with the bumbling chief of police, Lestrade, and sat down in front of the fireplace to resume reading the newspaper. Watson looked at him expectantly.

“What?” said Sherlock, not looking up from the paper.

“Well? What did Lestrade have to say?” Watson asked.

“Oh, he said they found a bit of rogue code in the bank account where the suspicious activity was taking place.”

“And?” Watson said, trying to hide his frustration.

“Lestrade emailed me the code, and I told him it wasn’t worth pursuing. The criminal made a fatal flaw and will never be able to actually steal the money,” Sherlock said.

“How do you know?” Watson asked.

“It’s obvious if you know where to look!” Sherlock exclaimed. “Now stop bothering me with questions and let me finish this paper.”

With Sherlock absorbed in the latest news, Watson snuck a peek at Sherlock’s phone and pulled up Lestrade’s email to look at the code.

```
let balance = 10500;
let cameraOn = true;

function steal(balance, amount) {
    cameraOn = false;
    if (amount < balance) {
        balance = balance - amount;
    }
    return amount;
    cameraOn = true;
}

let amount = steal(balance, 1250);
alert("Criminal: you stole " + amount + "!");
```

NOTE

This is the real, actual bank balance in the account.

Why did Sherlock decide not to investigate the case? How could he know that the criminal would never be able to steal the money just by looking at the code? Is there one problem with the code? Or more?

Solution in “Five Minute Mystery Solution”

BULLET POINTS

- Declare a function using the **function** keyword, followed by the name of the function.
- Use parentheses to enclose any **parameters** a function has. Use empty parentheses if there are no parameters.
- Enclose the **body** of the function in curly braces.
- The statements in the body of a function are executed when you call the function.
- **Calling** a function and **invoking** a function are the same thing.
- You call a function by using its name and passing arguments to the function's parameters (if any).
- A function can optionally return a value using the **return** statement.
- A function creates a block (or local) scope for parameters and any local variables the function uses.
- Variables are either in the **global scope** (visible everywhere in your program) or in the **local scope** (visible only in the function where they are declared).
- Variables can be local to an entire function or local to just one **block** of code, like an if statement or while statement.
- A **block** of code is a group of statements enclosed in curly braces, like in an if statement or while statement.
- Declare local variables at the top of the body of your function, unless you need the variable in only one block.
- If you forget to declare a local variable using **let** or **const**, that variable will be global, which could have unintended consequences in your program.
- Functions are a good way to organize your code and create reusable chunks of code.
- You can customize the code in a function by passing in arguments to parameters (and using different arguments to get different results).
- Functions are also a good way to reduce or eliminate duplicate code.
- You can use **built-in functions**, like alert, prompt, and Math.random, to do work in your programs.
- Using built-in functions means using existing code you don't have to write yourself.
- It's a good idea to organize your code so your functions are together, and your global variables are together, at the top of your JavaScript

file.

In this chapter, you got functional. Now use some brain functions to do this crossword.

ACROSS

3. You can declare your functions _______ in your JavaScript file.

5. When you reload your page, all your _______ get reinitialized.

7. JavaScript uses _______ when passing arguments to functions.

10. What gets returned from a function without a return statement.

11. If you forget to declare your locals, they'll be treated like _______.

12. To get a value back from a function, use the _______ statement.

13. _______ through your code means following the execution line by line.

15. Use functions so you can _______ code over and over again.

17. A local variable can _______ a global variable.

DOWN

1. Local variables disappear when the _______ returns.

2. A parameter acts like a _______ in the body of a function.

4. The variables that arguments land in when they get passed to functions.

6. What gets passed to functions.

8. It’s better to use _______ variables whenever you can.

9. A variable with global _______ is visible everywhere.

14. Extra arguments to a function are _______.

16. Watson looked at the bank heist code in Sherlock’s _______ on his phone.

Solution in “JavaScript cross Solution”

From “Sharpen your pencil”

Do a little analysis of the code below. How does it look? Choose among the options at the bottom of the page, or write in your own analysis. Here’s our solution.

```
let dogName = "rover";
let dogWeight = 23;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spot";
dogWeight = 13;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spike";
dogWeight = 53;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
```

SHARPEN YOUR PENCIL SOLUTION

From "Sharpen your pencil"

We've got some more calls to bark below. Next to each call, write what you think the output should be, or if you think the code will cause an error. Here's our solution.

CODE MAGNETS SOLUTION

From "Code Magnets"

This working JavaScript code is all scrambled up on the fridge. Can you reorder the code snippets to make a working program that produces the output listed below? Note that there may be some extra code on the fridge, so you may not use all the magnets. Here's our solution.

EXERCISE SOLUTION

From “Sharpen your pencil”

Below you’ll find some JavaScript code, including variables, function definitions, and function calls. Your job is to identify all the variables and constants, functions, arguments, and parameters. Write the names of each in the appropriate boxes on the right. Here’s our solution.

```
function dogYears(dogName, age) {
    let years = age * 7;
    console.log(dogName + " is " + years + " years old");
}
const myDog = "Fido";
dogYears(myDog, 4);

function makeTea(cups, tea) {
    console.log("Brewing " + cups + " cups of " + tea);
}
let guests = 3;
makeTea(guests, "Earl Grey");

function secret() {
    console.log("The secret of life is 42");
}
secret();

function speak(kind) {
    const defaultSound = "";
    if (kind == "dog") {
        alert("Woof");
    } else if (kind == "cat") {
        alert("Meow");
    } else {
        alert(defaultSound);
    }
}
let pet = prompt("Enter a type of pet: ");
speak(pet);
```

EXERCISE SOLUTION

From “Exercise”

Below you’ll find some JavaScript code, including variables, function definitions, and function calls. Your job is to identify the variables used in all the arguments, parameters, local or block variables, and global variables. Write the variable names in the appropriate boxes on the right. Then circle any variables that are shadowed. Here’s our solution.

The Thing-A-Ma-Jig

From “The Thing-A-Ma-Jig”

The Thing-A-Ma-Jig is quite a contraption—it clanks and clunks and even thunks, but what it really does, well, you’ve got us stumped. Coders claim they know how it works. Can you uncrack the code and find its quirks?

FIVE MINUTE MYSTERY SOLUTION

From “Five-Minute Mystery”

Why did Sherlock decide not to investigate the case? How could he know that the criminal would never be able to steal the money just by looking at the code? Is there one problem with the code? Or more? Here’s our solution.

WHO AM I? SOLUTION

From “Who am I?”

A bunch of JavaScript attendees, in full costume, are playing a party game: “Who am I?” They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. Fill in the blank next to each sentence with the name of one attendee. Here’s our solution.

`Tonight’s attendees are:`

`function, argument, return, scope, local variable, global variable, pass-by-value, parameter, function call, block, Math.random, built-in functions, code reuse`

From “JavaScript cross”

In this chapter, you got functional. Now use some brain functions to do this crossword. Here’s our solution.