

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, 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*.

More on this
as we progress
through the book.





Do a little analysis of the code below. How does it look? Choose as many of the options below as you like, or write in your own analysis:

```
var dogName = "rover";
var 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. _____
-

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 have a closer look at the code in question:

```
var dogName = "rover";
var 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");
}
```

Sure, this code looks innocent enough, but it's tedious to write, a pain to read and will be problematic if your code 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 we've got the same logic repeated over and over, and 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 to put it in one place where it can be easily re-used whenever we need it.

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");
}
```



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 we need it. That sounds like just the medicine we need.

Say you're writing some code that does a lot of "barking." 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) {  
}  
} // Next we're going to write some code that gets executed when we use the function.
```

The function keyword begins a function definition.

Next 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 }.

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");  
    }  
} // ...then output the dog's name with WOOF WOOF or woof woof.  
// Notice the variable names used in the code match the parameters of the function.
```

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

Now you have a function you can use in your code. Let's 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);      ↪ Now all that code becomes just a few calls to the
bark("spot", 13);       ↪ bark function, passing it each dog's name and weight.
bark("spike", 53);      ↪ Wow, now
bark("lady", 17);       ↪ that's simple!
```

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

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.

Okay, 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:

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.

"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.

bark("rover", 23);

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

```
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.

After we know the value for each parameter—like name is "rover" and weight is 23—then 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 then 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");
  }
}
```

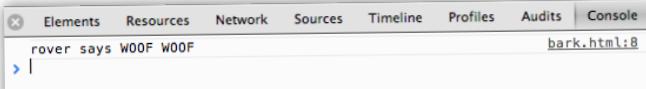
The parameters act like variables in the body, which have been assigned the values of the arguments you passed in.

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, sounds like "WOOF WOOF"), and the function is done. After the function completes, then control is returned to the statement following our call to bark.

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

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

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



"rover says WOOF WOOF"



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 answer 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?



Code Magnets

This working JavaScript code is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working program that produces the output listed below? Notice, 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 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.



The Function Exposed

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 take code, write it 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 `computeShoppingCartTotal`. Then you can pass that function the shopping carts of many users and each time I compute the amount of each specific shopping cart.

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`, `prompt`, `Math.random`, `document.write`. 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 as 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: Ohhhh 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 pretty much any JavaScript value as an argument, like a string, a boolean, or a number:

Pass any JavaScript value as an argument.

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

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

function saveMyProfile(name, birthday, GPA, newuser) {
 if (birthday >= 2004) {
 // 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:

```
var student = "krissy";  

var year = 1991;  

var GPA = 381/100;  

var status = "existinguser";  

var isNewUser = (status == "newuser");  

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:

```
var student = "krissy";  

var status = "existinguser";  

var year = 1991;  

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

Yes, even these expressions will work as arguments!

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

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 one 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.



What does this code output? Are you sure?

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



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 answer at the end of the chapter before you go on.

```
function dogYears(dogName, age) {
    var years = age * 7;
    console.log(dogName + " is " + years + " years old");
}

var myDog = "Fido";
dogYears(myDog, 4);

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

var guests = 3;
makeTea(guests, "Earl Grey");

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

secret();

function speak(kind) {
    var defaultSound = "";
    if (kind == "dog") {
        alert("Woof");
    } else if (kind == "cat") {
        alert("Meow");
    } else {
        alert(defaultSound);
    }
}

var pet = prompt("Enter a type of pet: ");
speak(pet);
```

Variables

Functions

Built-in functions

Arguments

Parameters

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.

- 1 Let's declare a variable `age`, and initialize it to the value 7.

```
var age = 7;
```



- 2 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;
}
```



- 3 Now let's call the function `addOne`, pass it the variable `age` as the argument. The value in `age` is copied into the parameter `x`.

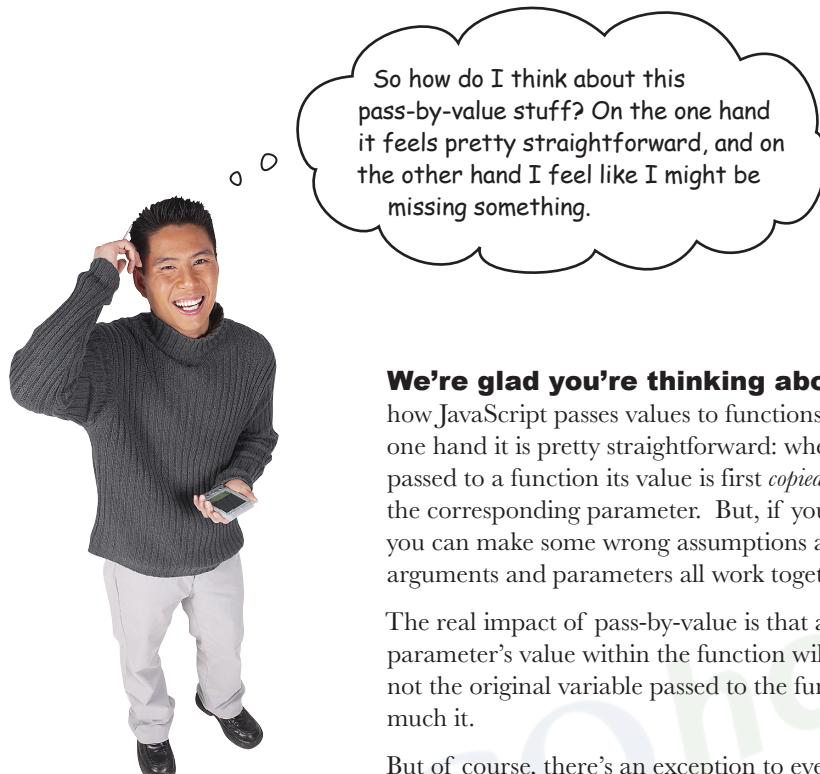
```
addOne(age);
```



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

```
We're incrementing x.
function addOne(x) {
  x = x + 1;
}
```





We're glad you're thinking about it. Understanding how JavaScript passes values to functions is important. On the one hand it is pretty straightforward: when an argument is passed to a function its value is first *copied* and then assigned to the corresponding parameter. 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 talk about this topic again when we learn objects, which we'll talk about in a couple of chapters. But no worries, 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

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

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

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);
```

Notice that the value of the parameter tea is undefined because we didn't pass in a value.

JavaScript console

```
Brewing 3 cups of undefined
```



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

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

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

Works fine, the function ignores the extras.

JavaScript console

```
Brewing 3 cups of Earl Grey
```

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("Woooooooooooooo!");
}
barkAtTheMoon();
```

JavaScript console

```
Woooooooooooooo!
```

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:

```
function bake(degrees) {
    var message;

    if (degrees > 500) {
        message = "I'm not a nuclear reactor!";
    } else if (degrees < 100) {
        message = "I'm not a refrigerator!";
    } else {
        message = "That's a very comfortable temperature for me.";
        setMode("bake");
        setTemp(degrees);
    }

    return message;
}

var status = bake(350);
```

Here we've got a new `bake` function that takes the temperature in degrees for the oven.

It then sets a variable to a string that depends on the temperature requested in the `degrees` parameter.

And presumably some real work is getting done here, but we won't worry about those details for now...

What we care about is that a `return` statement is returning the message as the result of this function.

Now, when the function is called and returns, the string that is returned as a result will be assigned to the `status` variable.

And in this case, if the `status` variable was printed, it would hold the string "That's a very comfortable temperature for me." Work through the code and make sure you see why!

350 degrees is the perfect temperature for good cookies. Feel free to make some and return to the next page.



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.

- ① First, we declare a variable `radius` and initialize it to 5.2.
- ② Next, we call the `calculateArea` function, and pass the `radius` variable as the argument.
- ③ The argument is sent to the parameter `r`, and the `calculateArea` function begins executing with `r` containing the value 5.2.
- ④ The body of the function executes starting with declaring a variable, `area`. We then test to see if the parameter `r` has a value ≤ 0 .
- ⑤ If $r \leq 0$, then we return 0 from the function and the function stops executing. But we passed in 5.2 so this line does NOT execute.
- ⑥ We execute the else clause instead.
- ⑦ We compute the area of the circle using the value 5.2 in the parameter `r`.
- ⑧ We return the value of `area` from the function. This stops the execution of the function and returns the value.
- ⑨ The value returned from the function is stored in the variable `theArea`.
- ⑩ Execution continues on the next line.

```

③ function calculateArea(r) {
    var area;
    if (r <= 0) {
        ⑤ return 0;
    } else {
        area = Math.PI * r * r;
        ⑧ return area;
    }
}

① var radius = 5.2;
② var theArea = calculateArea(radius);

⑩ console.log("The area is: " + theArea);

```

Output here!

JavaScript console

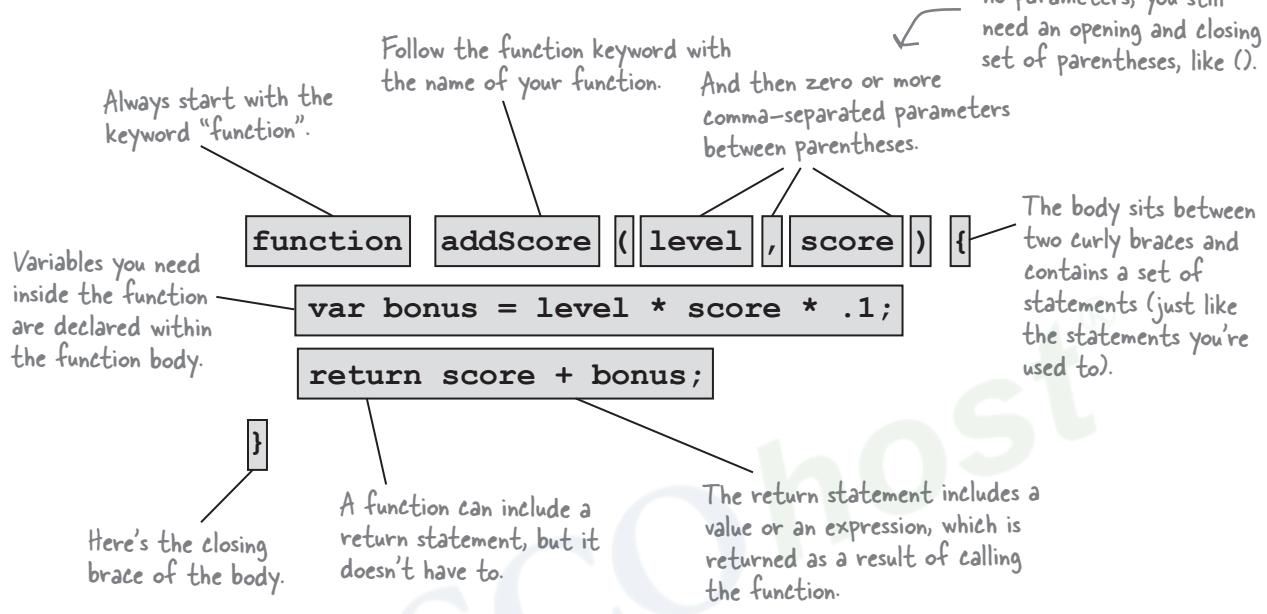
The area is: 84.94866535306801

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.



Anatomy of a Function

Now that you know how to define and call a function, let's make sure we'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 run time or incorrect code. Always take a careful look at a function's parameters, so you know what arguments the function expects to be passed in.

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

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

Q: What are the rules for function names?

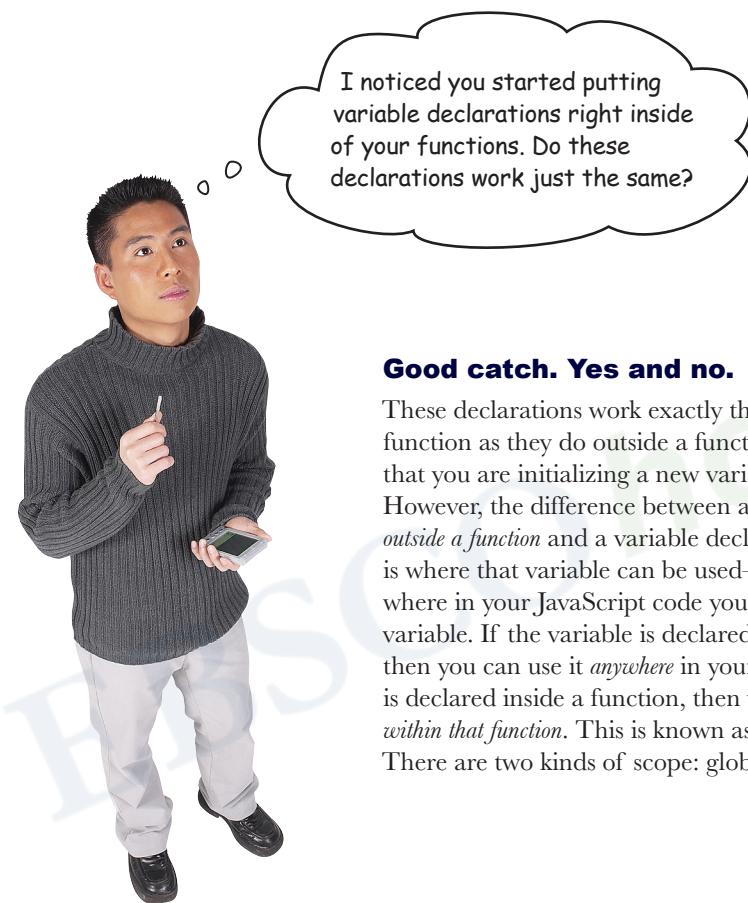
A: The rules for naming a function are the same as the rules for naming a variable. Just like with variables, you'll want to use names that make sense to you when you read them, and provide some indication of what the function does, and you can use camel case (e.g. camelCase) to combine words in function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

```
var avatar;
var levelThreshold = 1000;
```

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

```
function getScore(points) {
  var score;
  var i = 0;
  while (i < levelThreshold) {
    //code here
    i = i + 1;
  }
  return score;
}
```

Even if we use `levelThreshold` inside the function, it's global because it's declared outside the function.

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

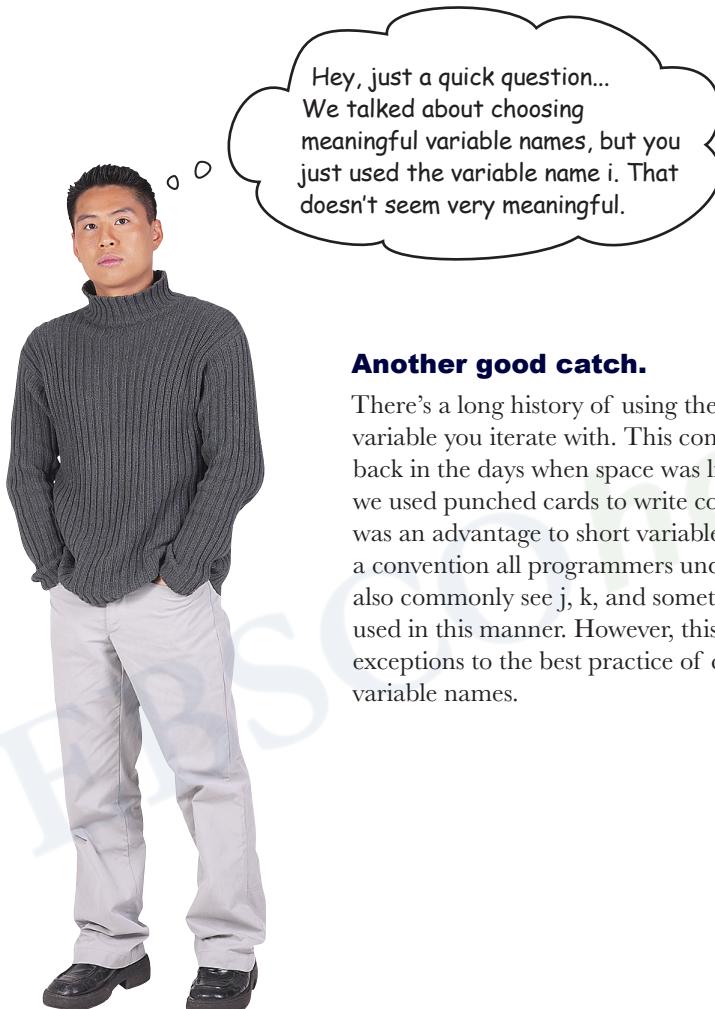
The `points`, `score` and `i`
variables are all declared
within a function.

We call them local variables
because they are known
locally only within the
function itself.

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

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, help you understand the code of others).

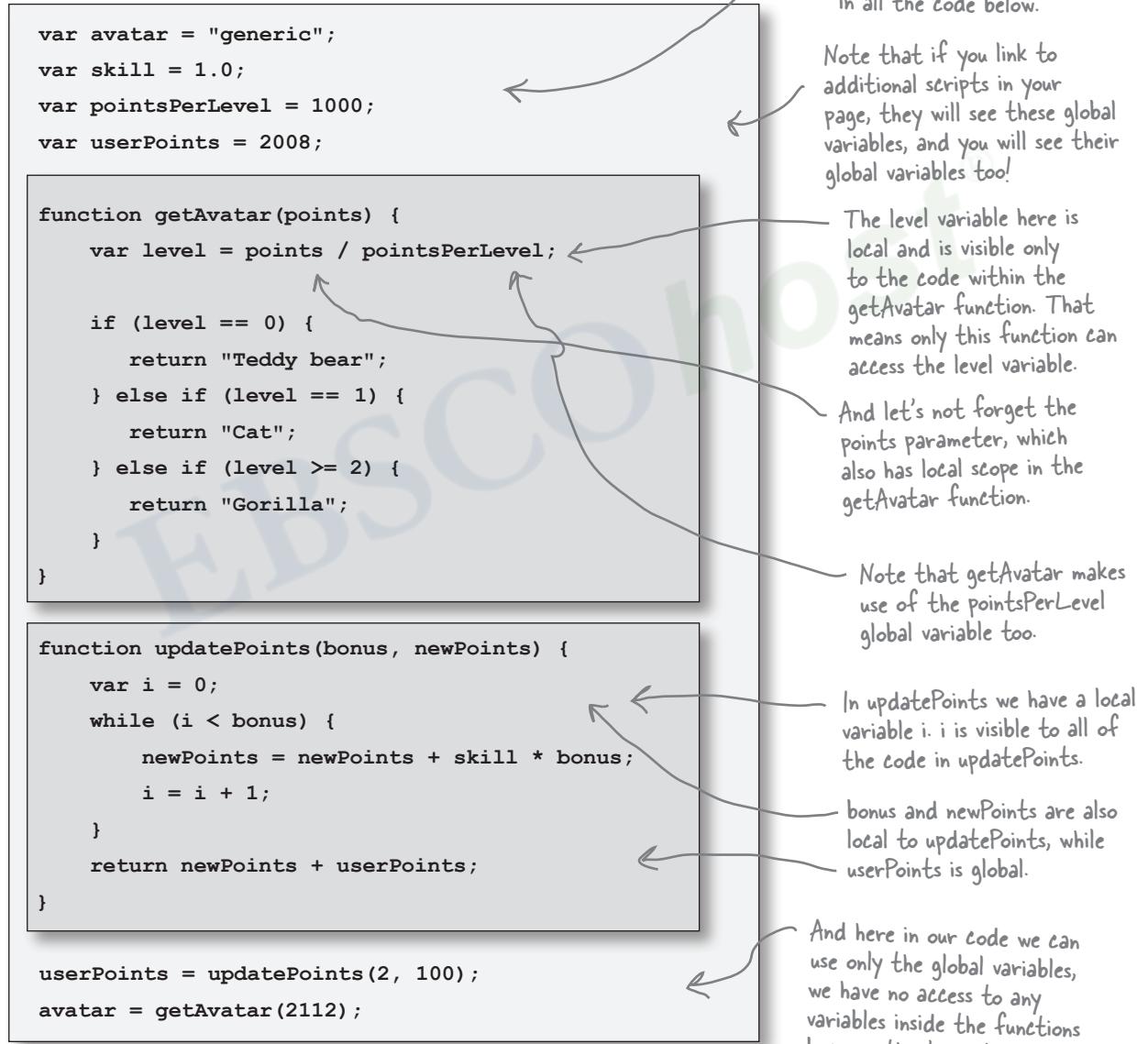


Another good catch.

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:



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, but 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 recreated 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.

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 as long as that browser doesn't reload the page.



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.

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!

```
function playTurn(player, location) {
    points = 0; ← We forgot to declare
    if (location == 1) { points with "var" before
        points = points + 100; we used it. So points is
    } automatically global.
    return points;
}

var total = playTurn("Jai", 1);
alert(points); ← That means we can use points outside the
                function! The value doesn't go away (like it
                should) when the function is done executing.
```

This program behaves as if you'd written this instead:

```
var points = 0; ← JavaScript assumes that we meant for points to be
function playTurn(player, location) { global because we forgot to use "var", and behaves
    points = 0; as if points were declared at the global level.
    if (location == 1) { ←
        points = points + 100;
    }
    return points;
}

var total = playTurn("Jai", 1);
alert(points); ← Forgetting to declare
                your local variables can
                cause problems if you use
                the same name as another
                global variable. You might
                overwrite a value you
                didn't mean to.
```

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).





You “shadow” your global.

Here's what that means: say you have a global variable **beanCounter** and you then declare a function, like this:

```
var beanCounter = 10;  
  
function getNumberOfItems(ordertype) {  
    var beanCounter = 0;  
    if (ordertype == "order") {  
        // do some stuff with beanCounter...  
    }  
    return beanCounter;  
}
```

We've got a global and a local!

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 that the local and global variables have no effect on each other: if you change one, it has no effect on 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 answer at the end of the chapter.

```

var x = 32;
var y = 44;
var radius = 5;

var centerX = 0;
var centerY = 0;
var width = 600;
var height = 400;

function setup(width, height) {
    centerX = width/2;
    centerY = height/2;
}

function computeDistance(x1, y1, x2, y2) {
    var dx = x1 - x2;
    var dy = y1 - y2;
    var d2 = (dx * dx) + (dy * dy);
    var d = Math.sqrt(d2);
    return d;
}

function circleArea(r) {
    var area = Math.PI * r * r;
    return area;
}

setup(width, height);
var area = circleArea(radius);
var distance = computeDistance(x, y, centerX, centerY);
alert("Area: " + area);
alert("Distance: " + distance);

```

Arguments

Parameters

Locals

Globals



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

Global 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!

You have to admit that I could replace all your previous local variables with global ones and your functions would work just the same.

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...

Well, if you'd use better names, then you might be able to keep track of your variables more easily.

True. But why bother with arguments and parameters if you've got all the values you need in globals?

Local variable:

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.

Well, yes and no. If you're extremely careful, sure. But being that careful is difficult, and if you make a mistake, then we'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.

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 what you've used x for before. You have to go searching all over your code to see if you've used x anywhere else! What a nightmare.

And what about parameters? Function parameters are always local. So you can't get around that.

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.

Global variable:

But your variables are just so... temporary. Locals come and go at a moment's notice.

Not at all? Globals are the mainstay of JavaScript programmers!

I think I need a drink.

Local variable:

Face it. It's just good programming practice to use local variables unless you absolutely need globals. And globals can get you into real trouble. I've seen JavaScript programs that barely use globals at all!

Of inexperienced programmers, sure. But as programmers learn to correctly structure their code for correctness, maintainability, and just good coding style, they learn how to stop using globals except when necessary.

They let globals drink? Now, we're *really* in dangerous territory.



there are no Dumb Questions

Q: Keeping track of the scope of all these locals 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 co-workers 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).

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: If I reload a page in the browser, do the global variables all get re-initialized?

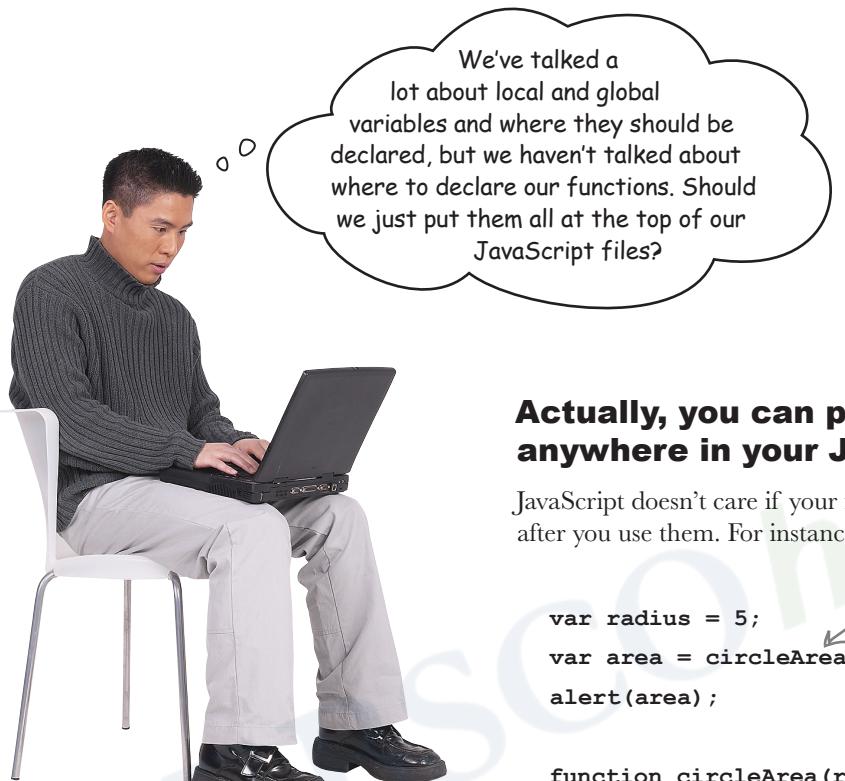
A: Yes. Reloading a page is like starting over from scratch as far as the variables are concerned. And if any code was in the middle of executing when you reload the page, any local variables will disappear, too.

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. However, it's a good programming practice to 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. In addition, 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: Everyone seems to complain about the overuse of global variables in JavaScript. Why is this? Was the language badly designed or do people not know what they're doing, or what? And what do we do about it?

A: Globals are often overused in JavaScript. Some of this is because the language makes it easy to just jump in and start coding—and that's a good thing—because JavaScript doesn't enforce a lot of structure or overhead on you. The downside is when people write serious code this way and it has to be changed and maintained over the long term (and that pretty much describes all web pages). All that said, JavaScript is a powerful language that includes features like objects you can use to organize your code in a modular way. Many books have been written on that topic alone, and we're going to give you a taste of objects in Chapter 5 (which is only a couple of chapters away).



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:

```
var radius = 5;
var area = circleArea(radius);
alert(area);

function circleArea(r) {
    var a = Math.PI * r * r;
    return a;
}
```

Notice that we're using the circleArea function before we're defining it!

The circleArea function isn't actually defined until after we've called it, in the code above. How on earth does this work?

This might seem really odd, especially if you remember 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.



Exercise

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?

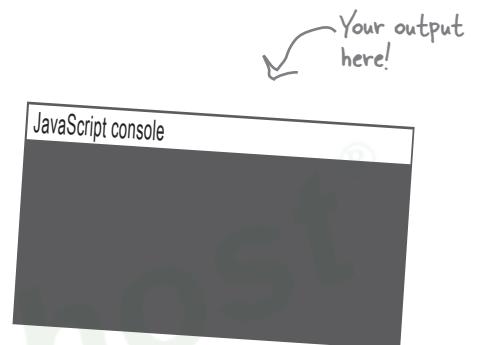
```
function clunk(times) {
    var num = times;
    while (num > 0) {
        display("clunk");
        num = num - 1;
    }
}

function thingamajig(size) {
    var 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);
    }
}

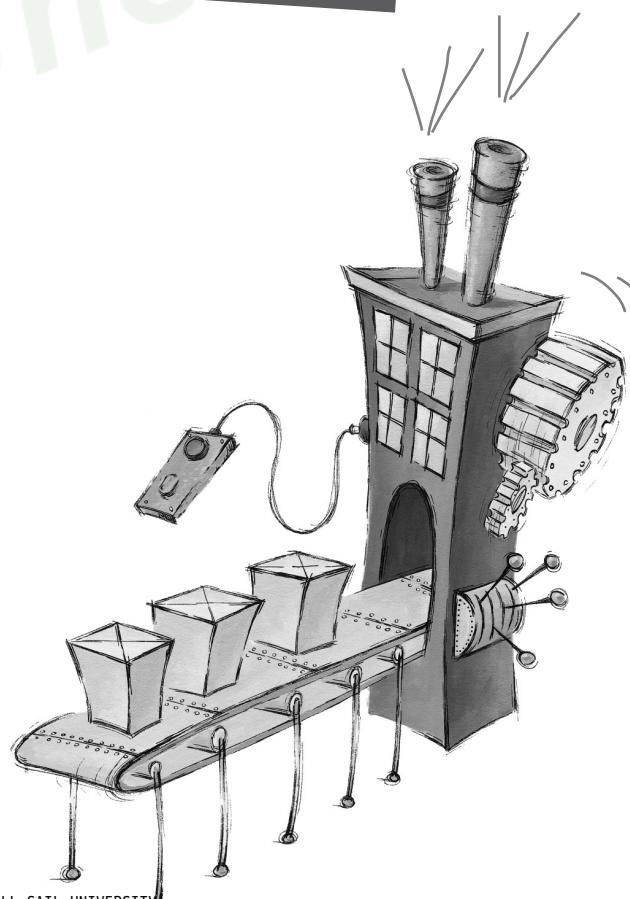
function display(output) {
    console.log(output);
    clunkCounter = clunkCounter + 1;
}

var clunkCounter = 0;
thingamajig(5);
console.log(clunkCounter);
```

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.



Your output here!



Webville Guide to Code Hygiene

In Webville we like to keep things clean, organized and ready for expansion. There's no place that needs to be better maintained 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 Webville. Take one, they're FREE.



Global variables, right at the TOP!

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?

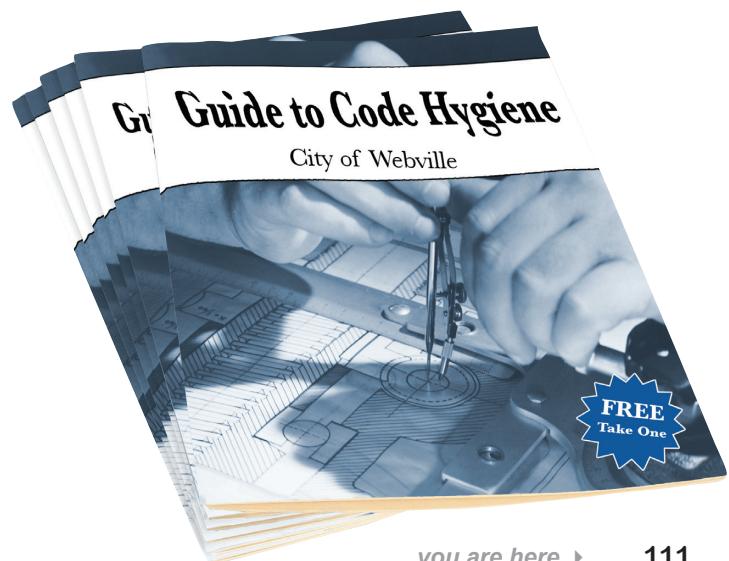
Functions like to sit together.

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, but if you keep them in one place your life will be easier. Here in Webville, we often start with our global variables and then put our functions next.

Let your local variables be declared at the TOP of the function they're in.

Put all your local variable declarations at the beginning of the function body. This makes them easy to find and ensures they are all declared properly before use.

That's it, just be safe and we hope you enjoy your time coding in Webville!



you are here ▶

111



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:

function, argument, return, scope, local variable, global variable, pass-by-value, parameter, function call, 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.

It really means '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.

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.

```
var balance = 10500;
var cameraOn = true;

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

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

```

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?



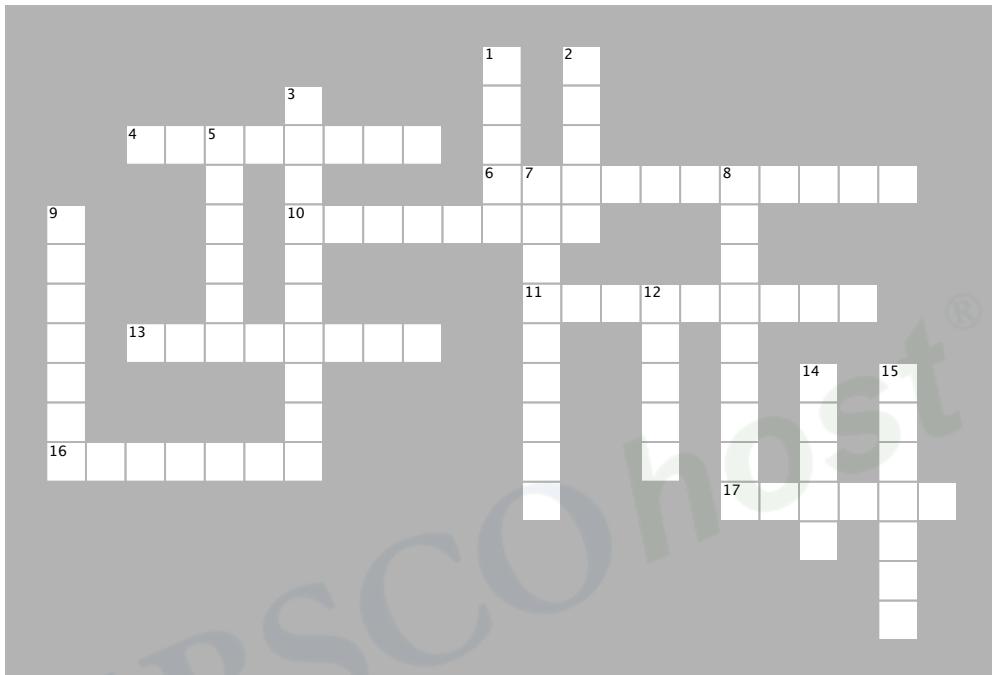
BULLET POINTS

- 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 a 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 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).
- Declare local variables at the top of the body of your function.
- If you forget to declare a local variable using **var**, 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 JavaScript's many 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.



JavaScript cross

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



ACROSS

4. A parameter acts like a _____ in the body of a function.
6. JavaScript uses _____ when passing arguments to functions.
10. You can declare your functions _____ in your JavaScript file.
11. What gets returned from a function without a return statement.
13. Local variables disappear when the _____ returns.
16. If you forget to declare your locals, they'll be treated like _____.
17. A local variable can _____ a global variable.

DOWN

1. A variable with global _____ is visible everywhere.
2. Use functions so you can _____ code over and over again.
3. The variables that arguments land in when they get passed to functions.
5. To get a value back from a function, use the _____ statement.
7. What gets passed to functions.
8. When you reload your page, all your _____ get re-initialized.
9. _____ through your code means following the execution line by line.
12. Watson looked at the bank heist code in Sherlock's _____ on his phone.
14. It's better to use _____ variables whenever you can.
15. Extra arguments to a function are _____.



Sharpen your pencil Solution

Do a little analysis of the code below. How does it look?
Choose as many of the options below as you like, or
write in your own analysis. Here's our solution.

```
var dogName = "rover";
var 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");
}
```

We chose all of them!

- A. The code seems very redundant.
- B. If we wanted to change how this outputs, or if we wanted to 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. *Looks like the developer thought the weights might change over time.*



Sharpen your pencil Solution

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.

`bark("juno", 20);` juno says woof woof

`bark("scottie", -1);` scottie says woof woof

↑ Our bark function doesn't check to make sure dog weights are greater than 0. So this works because -1 is less than 20.

`bark("dino", 0, 0);` dino says woof woof

↑ The bark function just ignores the extra argument, 0. And using 0 as the weight doesn't make sense, but it still works.

`bark("fido", "20");` fido says woof woof

↑ We compare the string "20" to the number 20. "20" isn't greater than 20, so fido says "woof woof". (You'll find out later how JavaScript compares "20" and 20.)

`bark("lady", 10);` lady says woof woof

`bark("bruno", 21);` bruno says WOOF WOOF



Code Magnets Solution

This working JavaScript code is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working program that produces the output listed below? Notice, there may be some extra code on the fridge, so you may not use all the magnets.

Here's our solution.

```

function whatShallIWear ( temp ) {
    if (temp < 60) {
        console.log("Wear a jacket");
    }
    else if (temp < 70) {
        console.log("Wear a sweater");
    }
    else {
        console.log("Wear t-shirt");
    }
}

whatShallIWear(50);
whatShallIWear(80);
whatShallIWear(60);

```

JavaScript console

```

Wear a jacket
Wear a t-shirt
Wear a sweater

```

Leftover magnets.

Diagram showing various code fragments floating around, labeled as 'Leftover magnets'. These include commas (,), braces ({}), and the word 'temperature'.



Exercise Solution

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. Here's our solution.

```
function dogYears(dogName, age) {
    var years = age * 7;
    console.log(dogName + " is " + years + " years old");
}

var myDog = "Fido";
dogYears(myDog, 4);

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

var guests = 3;
makeTea(guests, "Earl Grey");

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

secret();

function speak(kind) {
    var defaultSound = "";
    if (kind == "dog") {
        alert("Woof");
    } else if (kind == "cat") {
        alert("Meow");
    } else {
        alert(defaultSound);
    }
}

var pet = prompt("Enter a type of pet: ");
speak(pet);
```

Variables

myDog, guests, pet, years, defaultSound

Functions

dogYears, makeTea, secret, speak,

Built-in functions

alert, console.log, prompt

Arguments

myDog, 4, guests, "Earl Grey", pet, plus all the string arguments to alert and console.log

Parameters

dogName, age, cups, tea, kind



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. Here's our solution.

```

var x = 32;
var y = 44;
var radius = 5;

var centerX = 0;
var centerY = 0;
var width = 600;
var height = 400;

function setup(width, height) {
    centerX = width/2;
    centerY = height/2;
}

function computeDistance(x1, y1, x2, y2) {
    var dx = x1 - x2;
    var dy = y1 - y2;
    var d2 = (dx * dx) + (dy * dy);
    var d = Math.sqrt(d2);
    return d;
}

function circleArea(r) {
    var area = Math.PI * r * r;
    return area;
}

setup(width, height);
var area = circleArea(radius);
var distance = computeDistance(x, y, centerX, centerY);
alert("Area: " + area);
alert("Distance: " + distance);

```

Don't forget the arguments to the alert function.

Arguments

width, height, radius,
x, y, centerX,
centerY,
"Area: " + area,
"Distance: " + distance

Parameters

width, height,
x1, y1, x2, y2, r

The local variable
area shadows the
global variable area

Locals

dx, dy, d2, d,
area

Don't forget area
and distance. These
are globals too.

Globals

x, y, radius,
centerX, centerY,
width, height, area,
distance



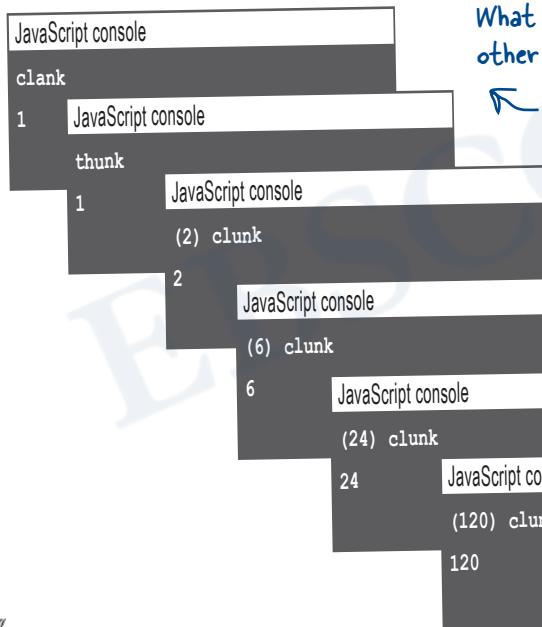
The Thing-A-Ma-Jig

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?

Here's our solution:

```
JavaScript console
(120) clunk
120
```

If you pass 5 to `thingamajig`, you'll see "clunk" in the console 120 times (or you might see your console abbreviate 120 clunks, like above), and then the number 120 at the end.



What about the other values?

Thing-A-Ma-Jig of 0

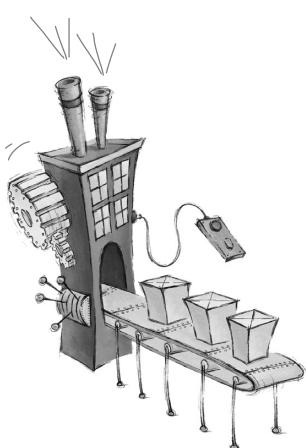
Thing-A-Ma-Jig of 1

Thing-A-Ma-Jig of 2

Thing-A-Ma-Jig of 3

Thing-A-Ma-Jig of 4

Thing-A-Ma-Jig of 5



What does it all mean? We hear the Thing-A-Ma-Jig was invented by a curious chap who was fascinated by rearranging words. You know like DOG rearranged is GOD, OGD, DGO, GDO and ODG. So if a word has three letters the Thing-A-Ma-Jig says you can make six total combinations from those letters. If you use the word "mixes" you can make 120 combinations of letters, wow! Anyway, that's what we heard. And here we just thought it computed mathematical factorials! Who knew!?

Check out 'factorial' in wikipedia for more info!



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.

```

var balance = 10500;           ← balance is a global variable...
var cameraOn = true;

function steal(balance, amount) { ← ... but it's shadowed
    cameraOn = false;
    if (amount < balance) {   ← by this parameter.
        balance = balance - amount;
    }
    return amount;
    cameraOn = true;          ← So when you change the
}                                ← balance in the function steal,
                                 ← you're not changing the actual
                                 ← bank balance!
}

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

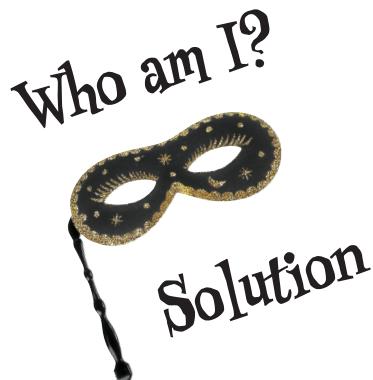
```

We're returning the
amount stolen...

... but we're not using it to
update the real balance in
the account. So the balance
of the bank account is the
same as it was originally.

The criminal thinks he stole
the money, but he didn't!

And, in addition to not actually stealing any money,
the criminal forgets to turn the camera back
on, which is a dead giveaway to the police that
something nefarious is going on. Remember, when you
return from a function, the function stops executing,
so any lines of code after the return are ignored!



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:

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

I get passed into a function.

argument

I send values back to the calling code.

return

I’m the all important keyword.

function

I’m what receives arguments.

parameter

It really means ‘make a copy’.

pass-by-value

I’m everywhere.

global variable

Another phrase for invoking a function.

function call

Example of a function attached to an object.

Math.random

alert and prompt are examples.

built-in functions

What functions are great for.

code reuse

Where I can be seen.

scope

I’m around when my function is.

local variable



JavaScript cross Solution

