10 first class functions

Liberated functions

Know functions, then rock. Every art, craft, and discipline has a key principle that separates the intermediate players from the rock star virtuosos—when it comes to JavaScript, it's truly understanding **functions** that makes the difference. Functions are fundamental to JavaScript, and many of the techniques we use to **design and organize** code depend on advanced knowledge and use of functions. The path to learning functions at this level is an interesting and often mind-bending one, so get ready... This chapter is going to be a bit like Willy Wonka giving a tour of the chocolate factory—you're going to encounter some wild, wacky and wonderful things as you learn more about JavaScript functions.

We'll spare you the singing Oompa Loompas.

this is a new chapter 429

The mysterious double life of the function keyword

So far we've been declaring functions like this:

```
function quack(num) {
   for (var i = 0; i < num; i++) {
      console.log("Quack!");
   }
}

quack(3);
```

A standard function declaration with the function keyword, a name, a parameter and a block of code.

And we can invoke this function by using its name followed by parentheses that enclose any needed arguments.

There are no surprises here, but let's get our terminology down: formally, the first statement above is a *function declaration*, which creates a function that has a name—in this case `quack`—that can be used to *reference* and *invoke* the function.

So far so good, but the story gets more mysterious because, as you saw at the end of the last chapter, there's another way to use the function keyword:

```
var fly = function(num) {
   for (var i = 0; i < num; i++) {
      console.log("Flying!");
   }
};

fly(3);
```

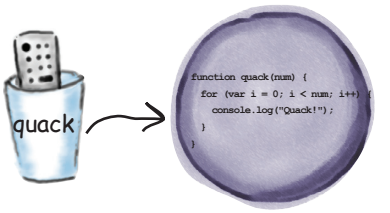
This doesn't look so standard: the function doesn't have a name, and it's on the right hand side of an assignment to a variable.

We can invoke this function too, this time by using the variable fly.

Now when we use the function keyword this way—that is, within a statement, like an assignment statement—we call this a *function expression*. Notice that, unlike the function declaration, this function doesn't have a name. Also, the expression results in a value that is then assigned to the variable `fly`. What is that value? Well, we're assigning it to the variable `fly` and then later invoking it, so it must be a *reference to a function*.

Serious Coding

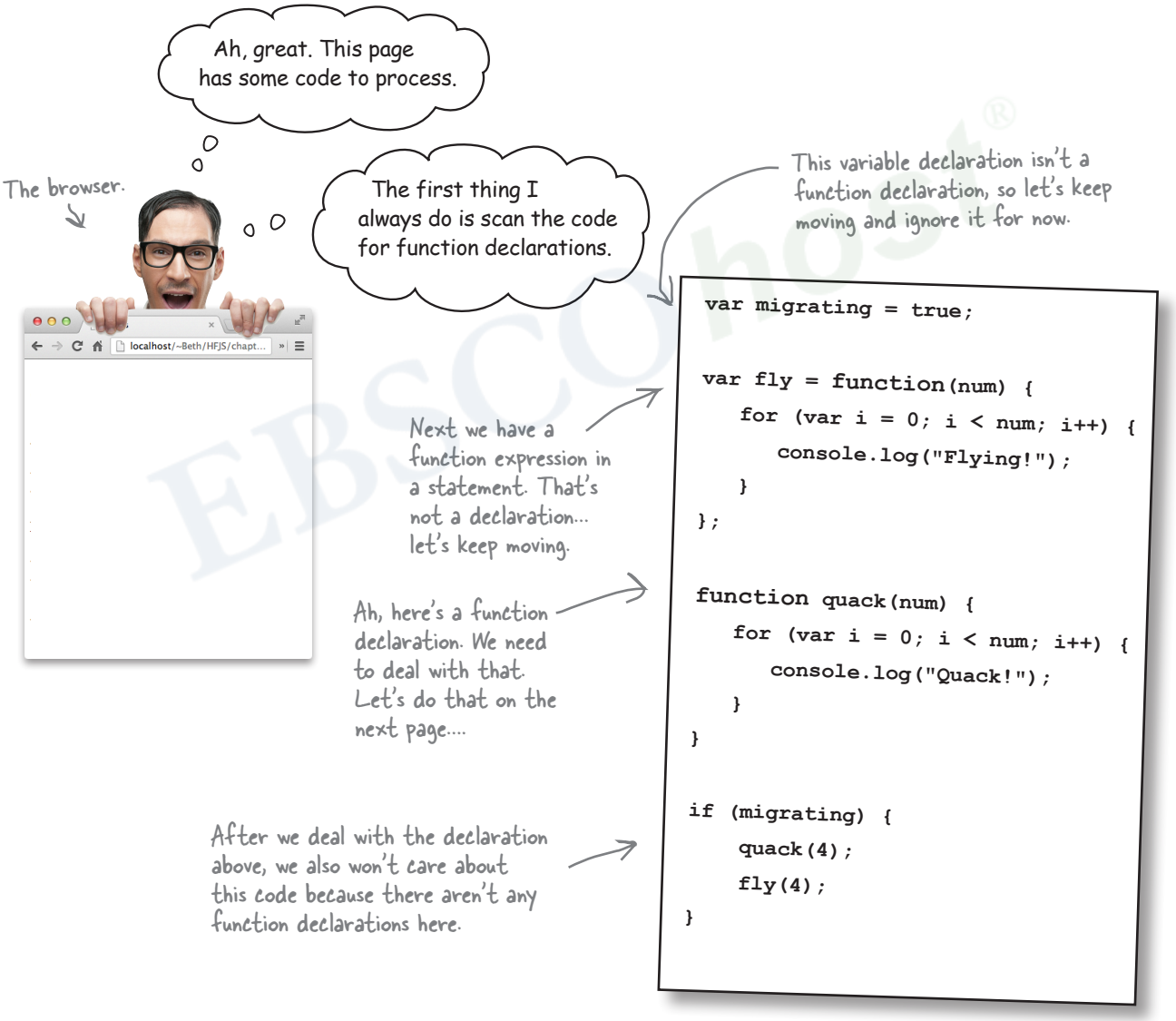
A function reference is exactly what it sounds like: a reference that refers to a function. You can use a function reference to invoke a function or, as you'll see, you can assign them to variables, store them in objects, and pass them to or return them from functions (just like object references).



Function declarations versus function expressions

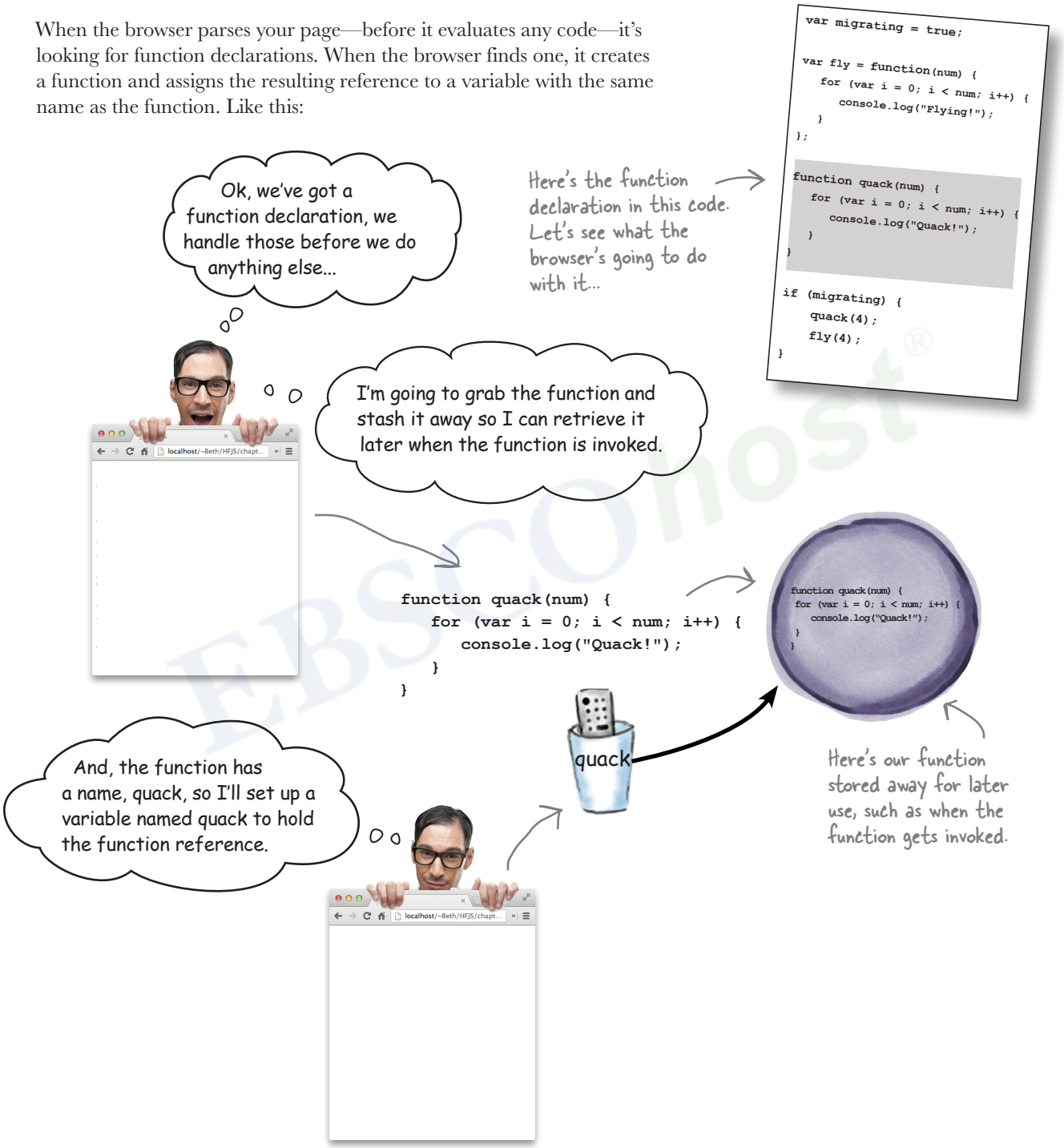
Whether you use a function declaration or a function expression you get the same thing: a function. So what's the difference? Is the declaration just more convenient, or is there something about function expressions that makes them useful? Or are these just two ways to do the same thing?

At first glance, it might appear as if there isn't a big difference between function declarations and function expressions. But, actually, there is something fundamentally different about the two, and to understand that difference we need to start by looking at how your code is treated by the browser at runtime. So let's drop in on the browser as it parses and evaluates the code in your page:



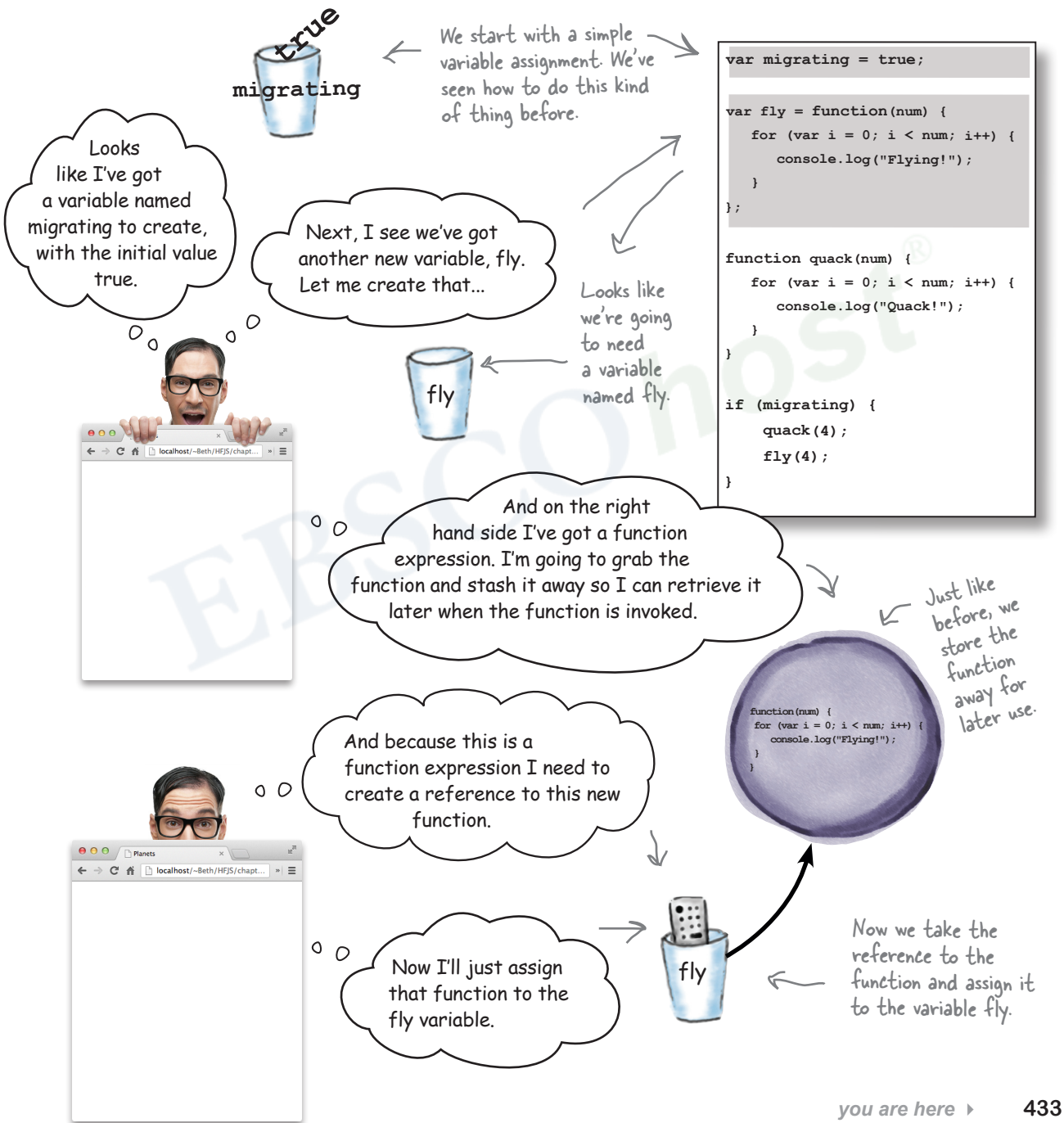
Parsing the function declaration

When the browser parses your page—before it evaluates any code—it's looking for function declarations. When the browser finds one, it creates a function and assigns the resulting reference to a variable with the same name as the function. Like this:



What's next? The browser executes the code

Now that all the function declarations have been taken care of, the browser goes back up to the top of your code and starts executing it, top to bottom. Let's check in on the browser at that point in the execution:



Moving on... The conditional

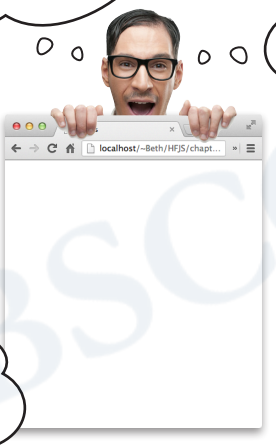
Once the `fly` variable has been taken care of, the browser moves on. The next statement is the function declaration for `quack`, which was dealt with in the first pass through the code, so the browser skips the declaration and moves on to the conditional statement. Let's follow along…

Been there, done that, moving on...

```
var migrating = true;

var fly = function(num) {
  for (i = 0; i < num; i++) {
    console.log("Flying!");
  }
};

function quack(num) {
  for (i = 0; i < num; i++) {
    console.log("Quack!");
  }
}

if (migrating) {
  quack(4);
  fly(4);
}
```

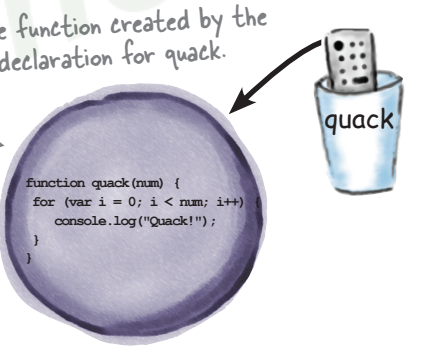
Let's see, the variable migrating is true, so I need to execute the body of the if statement. Inside it looks like there's a call to quack. I know it's a function call because we're using the function name, quack, followed by parentheses.

```
quack(4);
```

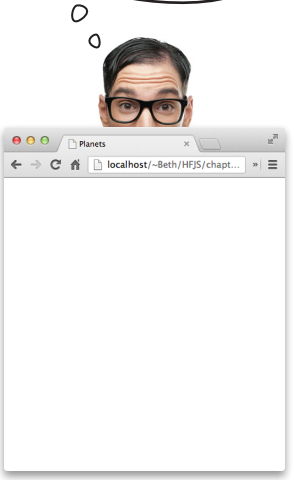


Remember, the quack variable is a reference to the function I stashed away earlier...

Here's the function created by the function declaration for quack.



There's an argument in the function call, so I'll pass that into the function...



4

To invoke the function, we pass a copy of the argument value to the parameter...

```
function quack(num) {
  for (var i = 0; i < num; i++) {
    console.log("Quack!");
  }
}
```

...and then execute the body of the function.

... and execute the code in the body of the function, which prints "Quack!" four times to the console log.

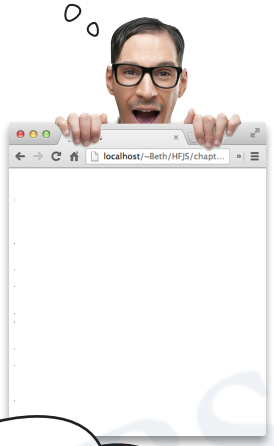
And finishing up...

All that's left is to invoke the `fly` function created by the function expression. Let's see how the browser handles this:

```
var migrating = true;

var fly = function(num) {
  for (var i = 0; i < num; i++) {
    console.log("Flying!");
  }
};

function quack(num) {
  for (var i = 0; i < num; i++) {
    console.log("Quack!");
  }
}

if (migrating) {
  quack(4);
  fly(4);
}
```

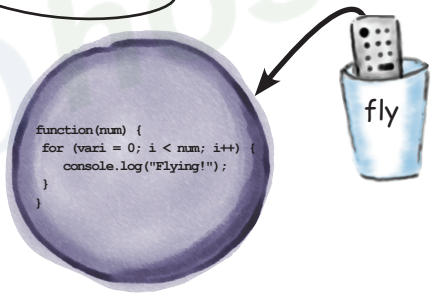
Hey look, another function call. I know it's a function call because we're using the variable name, fly, followed by parentheses.

```
fly(4);
```

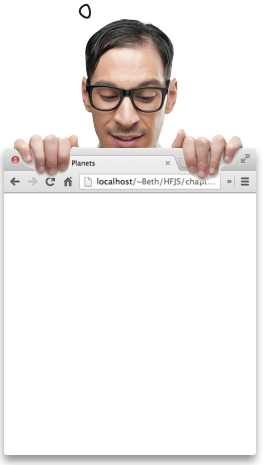


Remember the fly variable is a reference to the function I stashed away earlier...

Here's the function referenced by the fly variable.



There's an argument in the function call, so I'll pass that into the function...



4

To invoke the function, we pass a copy of the argument value to the parameter...

```
function(num) {
  for (var i = 0; i < num; i++) {
    console.log("Flying!");
  }
}
```

...and then execute the body of the function.

Then I'll execute the code in the body of the function, which prints "Flying!" four times to the console log.

Sharpen your pencil

What deductions can you make about function declarations and function expressions given how the browser treats the quack and fly code? Check each statement that applies. Check your answer at the end of the chapter before you go on.

- ❑ Function declarations are evaluated before the rest of the code is evaluated.
- ❑ Function expressions get evaluated later, with the rest of the code.
- ❑ A function declaration doesn't return a reference to a function; rather it creates a variable with the name of the function and assigns the new function to it.
- ❑ A function expression returns a reference to the new function created by the expression.
- ❑ You can hold function references in variables.
- ❑ Function declarations are statements; function expressions are used in statements.
- ❑ The process of invoking a function created by a declaration is exactly the same for one created with an expression.
- ❑ Function declarations are the tried and true way to create functions.
- ❑ You always want to use function declarations because they get evaluated earlier.

there are no Dumb Questions

Q: We've seen expressions like 3+4 and Math.random() * 6, but how can a function be an expression?

A: An expression is anything that evaluates to a value. 3+4 evaluates to 7, Math.random() * 6 evaluates to a random number, and a function expression evaluates to a function reference.

Q: But a function declaration is not an expression?

A: No, a function declaration is a statement. Think of it as having a hidden assignment that assigns the function reference to a variable for you. A function expression doesn't assign a function reference to anything; you have to do that yourself.

Q: What good does it do me to have a variable that refers to a function?

A: Well for one thing you can use it to invoke the function:

```
myFunctionReference();
```

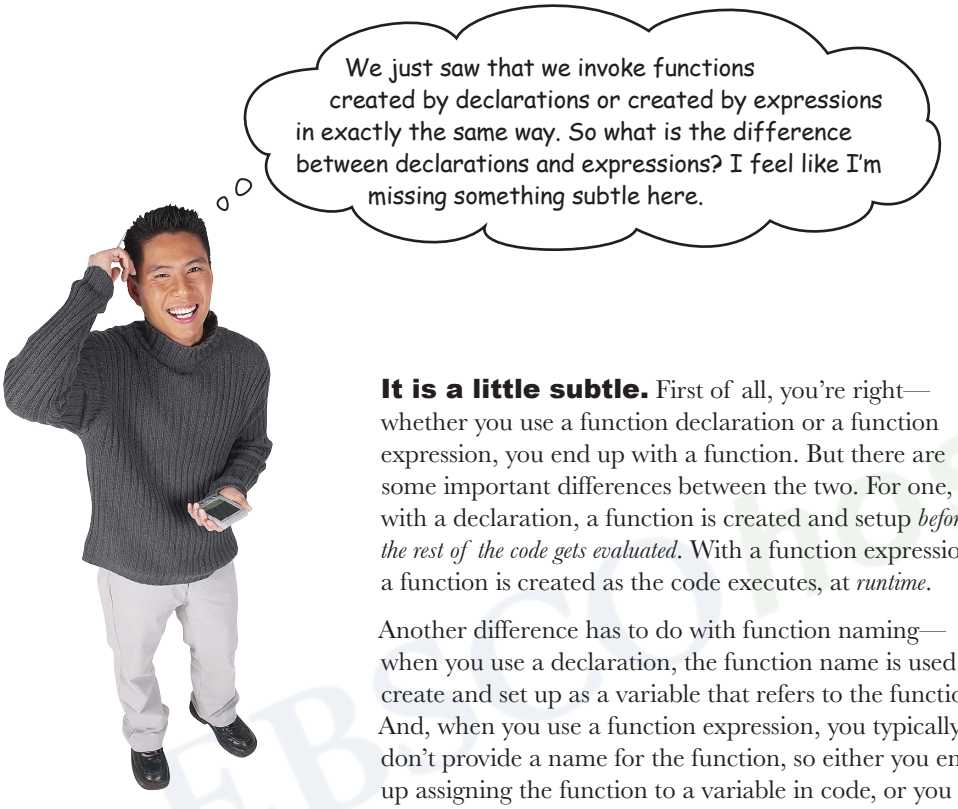
But you can also pass a reference to a function or return a reference from a function. But, we're getting a little ahead of ourselves. We'll come back to this in a few pages.

Q: Can function expressions only appear on the right hand side of an assignment statements?

A: Not at all. A function expression can appear in many different places, just like other kind of expressions can. Stay tuned because this is a really good question and we'll be coming back to this in just a bit.

Q: Okay, a variable can hold a reference to a function. But what is the variable really referencing? Just some code that is in the body of the function?

A: That's a good way to begin thinking about functions, but think of them more as a little crystallized version of the code, all ready to pull out at any time and invoke. You're going to see later that this crystallized function has a bit more in it than just the code from the body.



It is a little subtle. First of all, you're right—whether you use a function declaration or a function expression, you end up with a function. But there are some important differences between the two. For one, with a declaration, a function is created and setup *before the rest of the code gets evaluated*. With a function expression, a function is created as the code executes, at *runtime*.

Another difference has to do with function naming—when you use a declaration, the function name is used to create and set up as a variable that refers to the function. And, when you use a function expression, you typically don't provide a name for the function, so either you end up assigning the function to a variable in code, or you use the function expression in other ways.

We'll take a look at what those are later in the chapter.

Now take these differences and stash them in the back of your brain as this is all going to become useful shortly. For now, just remember how function declarations and expressions are evaluated, and how names are handled.

BE the Browser

Below, you'll find JavaScript code. Your job is to play like you're the browser evaluating the code. In the space to the right, record each function as it gets created. Remember to make two passes over the code: the pass that processes declarations, and the second pass that handles expressions.

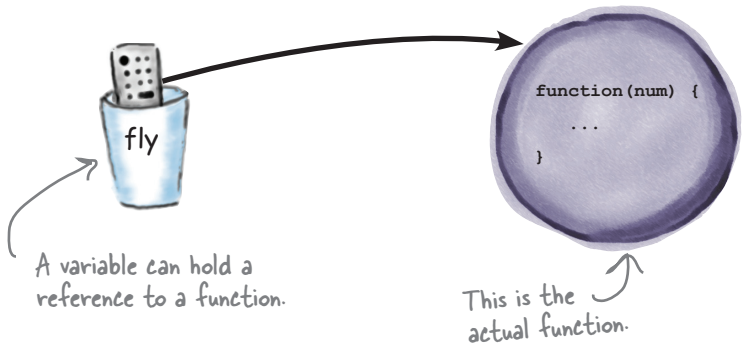
```
var midi = true;
var type = "piano";
var midiInterface;

function play(sequence) {
    // code here
}
var pause = function() {
    stop();
}
function stop() {
    // code here
}

function createMidi() {
   // code here
}

if (midi) {
    midiInterface = function(type) {
        // code here
    };
}
```

Write, in order, the names of the functions as they are created. If a function is created with a function expression put the name of the variable it is assigned to. We've done the first one for you.

play

How functions are values too

Sure, we all think of functions as things we invoke, but you can think of functions as *values* too. That value is actually a reference to the function, and as you've seen, whether you define a function with a function declaration or a function expression, you get a reference to that function.



One of the most straightforward things we can do with functions is assign them to variables. Like this:

```
function quack(num) {
   for (var i = 0; i < num; i++) {
      console.log("Quack!");
   }
}
var fly = function(num) {
   for (var i = 0; i < num; i++) {
      console.log("Flying!");
   }
}
```

Our two functions again. Remember quack is defined with a function declaration, and fly with a function expression. Both result in function references, which are stored in the variables quack and fly, respectively.

The function declaration takes care of assigning the reference to a variable with the name you supply, in this case quack.

When you have a function expression, you need to assign the resulting reference to a variable yourself. Here we're storing the reference in the fly variable.

```
var superFly = fly;
superFly(2);
```

After we assign the value in fly to superFly, superFly holds the function reference, so by adding some parentheses and an argument we can invoke it!

```
var superQuack = quack;
superQuack(3);
```

And even though quack was created by a function declaration, the value in quack is a function reference too, so we can assign it to the variable superQuack and invoke it.

In other words, references are references, no matter how you create them (that is, with a declaration or an expression)!

JavaScript console

```
Flying!
Flying!
Quack!
Quack!
Quack!
```

Sharpen your pencil

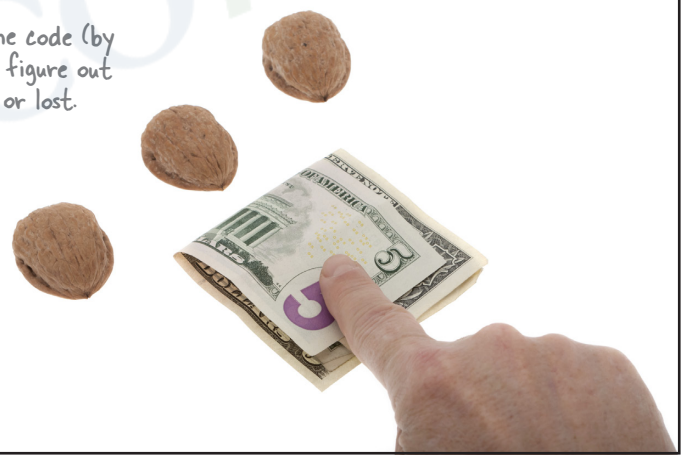
To get the idea of functions as values into your brain, let's play a little game of chance. Try the shell game. Will you win or lose? Give it a try and find out.

```
var winner = function() { alert("WINNER!") };
var loser = function() { alert("LOSER!") };
// let's test as a warm up
winner();
// let's assign to other variables for practice
var a = winner;
var b = loser;
var c = loser;
a();
b();
// now let's try your luck with a shell game
c = a;
a = b;
b = c;
c = a;
a = c;
a = b;
b = c;
a();
```

Remember, these variables hold references to the winner and loser functions. We can assign and reassign these references to other variables, just like with any value.

Remember, at any time, we can invoke a reference to a function.

Execute the code (by hand!) and figure out if you won or lost.



Start thinking about functions as values, just like numbers, strings, booleans or objects. The thing that really makes a function value different from these other values is that we can invoke it.

Function Exposed

This week's interview: Understanding Function

Head First: Function, we're so happy to finally have you back on this show. You're quite a mystery and our readers are dying to know more.

Function: It's true, I'm deep.

Head First: Let's start with this idea that you can be created with a declaration, or created with an expression. Why two ways of defining you? Wouldn't one be enough?

Function: Well, remember, these two ways of defining a function do two slightly different things.

Head First: But the result is the same: a function, right?

Function: Yes, but look at it this way. A function declaration is doing a little bit of work behind the scenes for you: it's creating the function, and then also creating a variable to store the function reference in. A function expression creates the function, which results in a reference and it's up to you to do something with it.

Head First: But don't we always just store the reference we get from a function expression in a variable anyway?

Function: Definitely not. In fact, we usually don't. Remember a function reference is a value. Think of the kinds of things you can do with other kinds of values, like an object reference for instance. I can do all those things too.

Head First: But how can you possibly do everything those other values can do? I declare a function, and I call it. That's about as much as any language allows, right?

Function: Wrong. You need to start thinking of a function as a value, just like objects or the primitive types. Once you get a hold of a function you can do all kinds of things with it. But there is one important difference between a function and other kinds of values, and that is what really makes me what I am: a function can be invoked, to execute the code in its body.

Head First: That sounds very impressive and powerful, but I'd have no idea what to do with you other than define and call you.

Function: This is where we separate the six figure coders from the scripters. When you can treat a function like any other value, all kinds of interesting programming constructs become possible.

Head First: Can you give us just one example?

Function: Sure. Say you want to write a function that can sort *anything*. No problem. All you need is a function that takes two things: the collection of items you need to sort, and *another function* that knows how to compare any two items in your collection. Using JavaScript you can easily create code like that. You write one sort function for every kind of collection, and then just tell that function how to compare items by passing it a function that knows how to do the comparison.

Head First: Err…

Function: Like I said, this is where we separate the six figure coders from the scripters. Again, we're *passing a function* that knows how to do the comparison *to the other function*. In other words we're treating the function like a value by passing it to a function *as a value*.

Head First: And what does that get us other than confused?

Function: It gets you less code, less hard work, more reliability, better flexibility, better maintainability, a higher salary.

Head First: That all sounds good, but I'm still not sure how to get there.

Function: Getting there takes a little bit of work. This is definitely an area where your brain has to expand a bit.

Head First: Well, function, my head is expanding so much it's about to explode, so I'm going to go lie down.

Function: Any time. Thanks for having me!

Did we mention functions have First Class status in JavaScript?

If you're coming to JavaScript from a more traditional programming language you might expect functions to be… well, just functions. You can declare them and call them, but when you're not doing either of those things they just sit around doing nothing.

Now you know that functions in JavaScript are values—values that can be assigned to variables. And you know that with values of other types, like numbers, booleans, strings and even objects, we can do all sorts of things with those values, like pass them to functions, return them from functions or even store them in objects or arrays.

Computer scientists actually have a term for these kinds of values: they're called *first class values*. Here's what you can do with a first class value:

- ❏ Assign the value to a variable (or store it in a data structure like an array or object).
- ❏ Pass the value to a function.
- ❏ Return the value from a function.

Guess what? We can do all these things with functions too. In fact, we can do *everything* with a function that we can do with other values in JavaScript. So consider functions first class values in JavaScript, along with all the values of the types you already know: numbers, strings, booleans and objects.

Here's a more formal definition of first class:

> **First class**: a value that can be treated like any other value in a programming language, including the ability to be assigned to a variable, passed as an argument, and returned from a function.

We're going to see that JavaScript functions easily qualify as first class values—in fact, let's spend a little time working through just what it means for a function to be first class in each of these cases. Here's a little advice first: stop thinking about functions as something special and different from other values in JavaScript. There is great power in treating a function like a value. Our goal in the rest of the chapter is to show you why.

We always thought VIP-access-to-all-areas was a better name, but they didn't listen to us, so we'll stick with first class.

Flying First Class

If that answer lands you the big job, don't forget about us! We take donations in chocolate, pizza or bitcoins.

The next time you're in a job interview and you get asked "What makes JavaScript functions first class?" you're going to pass with flying colors. But before you start celebrating your new career, remember that, so far, all your understanding of first class functions is *book knowledge*. Sure, you can recite the definition of what you can do with first class functions:

- ☑ You can assign functions to variables. ← You've seen this already.
- ❑ You can pass functions to functions. ← We're going to work on this now.
- ❑ You can return functions from functions. ← And we'll cover this in just a bit...

But can you use those techniques in your code, or know when it would help you to do so? No worries; we're going to deal with that now by learning how to pass functions to functions. We're going to start simple, and take it from there. In fact, we're going to start with just a simple data structure that represents passengers on an airline flight:

Here's the data structure representing the passengers:

All passengers are kept in an array.

And here we have four passengers (feel free to expand this list with friends and family).

```
var passengers = [  { name: "Jane Doloop", paid: true },
                    { name: "Dr. Evel", paid: true },
                    { name: "Sue Property", paid: false },
                    { name: "John Funcall", paid: true } ];
```

And each passenger is represented by an object with a name and a paid property.

The name is a simple text string.

And paid is a boolean that represents whether or not the passenger has paid for the flight.

Here our goal: write some code that looks at the passenger list and makes sure that certain conditions are met before the flight is allowed to take off. For instance, let's make sure there are no passengers on a no-fly list. And let's make sure everyone has paid for the flight. We might even want to create a list of everyone who is on the flight.

BRAIN POWER

Think about how you'd write code to perform these three tasks (no-fly list, paid customers and a list of passengers)?

Writing code to process and check passengers

Now typically you'd write a function for each of these conditions: one to check the no-fly-list, one to check that every passenger has paid, and one to print out all the passengers. But if we wrote that code and stepped back to look at it, we'd find that all these functions look roughly the same, like this:

```
function checkPaid(passengers) {
    for (var i = 0; i < passengers.length; i++) {
        if (!passengers[i].paid) {
            return false;
        }
    }
    return true;
}
```

```
function checkNoFly(passengers) {
    for (var i = 0; i < passengers.length; i++) {
        if (onNoFlyList(passengers[i].name)) {
            return false;
        }
    }
    return true;
}
```

```
function printPassengers(passengers) {
    for (var i = 0; i < passengers.length; i++) {
        console.log(passengers[i].name);
        return false;
    }
    return true;
}
```

The only thing different here is the test for paid versus being on a no fly list.

And print is different only in that there is no test (instead we pass the passenger to console.log) and we don't care about the return value, but we're still iterating through the passengers.

That's a lot of duplicated code: all these functions iterate through the passengers doing something with each passenger. And what if there are additional checks needed in the future? Say, checking to make sure laptops are powered down, checking to see if a passenger has an upgrade, checking to see if a passenger has a medical issue, and so on. That's a lot of redundant code.

Even worse, what if the data structure holding the passengers changes from a simple array of objects to something else? Then you might have to open every one of these functions and rewrite it. Not good.

We can solve this little problem with first class functions. Here's how: we're going to write one function that knows how to iterate through the passengers, and pass to that function a second function that knows how to do the check we need (that is, to see if a name is on a no-fly list, to check whether or not a passenger has paid, and so on).

Let's do a little pre-work on this by first writing a function that takes a passenger as an argument and checks to see if that passenger's name is on the no-fly-list. Return true if it is and false otherwise. Write another function that takes a passenger and checks to see if the passenger hasn't paid. Return true if the passenger has not paid, and false otherwise. We've started the code for you below; you just need to finish it. You'll find our solution on the next page, but don't peek!

```
function checkNoFlyList(passenger) {



}

function checkNotPaid(passenger) {



}
```

Hint: assume your no-fly list consists of one individual: Dr. Evel.

Let's get your brain warmed up for passing your first function to another function. Evaluate the code below (in your head) and see what you come up with. Make sure you check your answer before moving on.

```
function sayIt(translator) {
    var phrase = translator("Hello");
    alert(phrase);
}

function hawaiianTranslator(word) {
    if (word === "Hello") return "Aloha";
    if (word === "Goodbye") return "Aloha";
}

sayIt(hawaiianTranslator);
```

Iterating through the passengers

We need a function that takes the passengers and another function that knows how to test a single passenger for some condition, like being on the no-fly list. Here's how we do that:

The function processPassengers has two parameters. The first is an array of passengers.

And the second is a function that knows how to look for some condition in the passengers.

```
function processPassengers(passengers, testFunction) {
    for (var i = 0; i < passengers.length; i++) {
        if (testFunction(passengers[i])) {
            return false;
        }
    }
    return true;
}
```

We iterate through all the passengers, one at a time.

And then we call the function on each passenger.

If the result of the function is true, then we return false. In other words, if the passenger failed the test (e.g. they haven't paid, or they are on the no-fly list), then we don't want the plane to take off!

Otherwise, if we get here then all passengers passed the test and we return true.

Now all we need are some functions that can test passengers (luckily you wrote these in the previous Sharpen Your Pencil exercise). Here they are:

Pay attention: this is one passenger (an object) not the array of passengers (an array of objects).

```
function checkNoFlyList(passenger) {
    return (passenger.name === "Dr. Evel");
}
```

Here's the function to check to see if a passenger is on the no-fly list. Our no-fly list is simple: everyone except Dr. Evel can fly. We return true if the passenger is Dr. Evel; otherwise, we return false (that is, the passenger is not on the no-fly list).

```
function checkNotPaid(passenger) {
    return (!passenger.paid);
}
```

And here's the function to check to see if a passenger has paid. All we do is check the paid property of the passenger. If they have not paid, then we return true.

Passing a function to a function

Okay, we've got a function that's ready to accept a function as an argument (`processPassengers`), and we've got two functions that are ready to be passed as arguments to `processPassengers` (`checkNoFlyList` and `checkNotPaid`).

It's time to put this all together. Drum roll please…

Passing a function to a function is easy. We just use the name of the function as the argument.

Here, we're passing the checkNoFlyList function. So processPassengers will check each passenger to see if they are on the no-fly list.

```
var allCanFly = processPassengers(passengers, checkNoFlyList);
if (!allCanFly) {
    console.log("The plane can't take off: we have a passenger on the no-fly-list.");
}
```

If any of the passengers are on the no-fly list, we'll get back false, and we'll see this message in the console.

Here, we're passing the checkNotPaid function. So processPassengers will check each passenger to see if they've paid.

```
var allPaid = processPassengers(passengers, checkNotPaid);
if (!allPaid) {
    console.log("The plane can't take off: not everyone has paid.");
}
```

If any of the passengers haven't paid, we'll get back false, and we'll see this message in the console.

Test ~~drive~~ flight

To test drive your code, just add this JavaScript to a basic HTML page, and load it into your browser.

JavaScript console

```
The plane can't take off: we have a passenger on the no-fly-list.
The plane can't take off: not everyone has paid.
```

Well, looks like we won't be taking off after all. We've got problems with our passengers! Good thing we checked...

Your turn again: write a function that prints a passenger's name and whether or not they have paid to console.log. Pass your function to processPassengers to test it. We've started the code for you below; you just need to finish it up. Check your answer at the end of the chapter before you go on.

```
function printPassenger(passenger) {

}

processPassengers(passengers, printPassenger);
```

Write your code here.

Your code should print out the list of passengers when you pass the function to processPassengers.

there are no Dumb Questions

Q: Couldn't we just put all this code into processPassengers? We could just put all the checks we need into one iteration, so each time through we do all the checks and print the list. Wouldn't that be more efficient?

A: If your code is short and simple, yes, that might be a reasonable approach. However, what we're really after is flexibility. What if, in the future, you are constantly adding new checks (has everyone put their laptop away?) or requirements for your existing functions change? Or the underlying data structure for passengers changes? In these cases, the design we used allows you to make changes or additions in a way that reduces overall complexity and that is less likely to introduce bugs into your code.

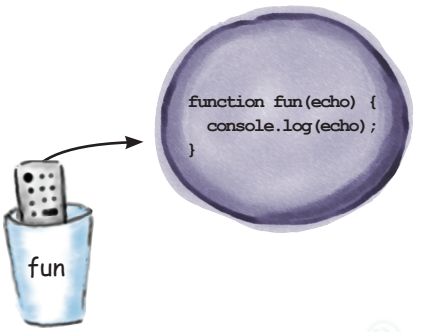
Q: What exactly are we passing when we pass a function to a function?

A: We're passing a reference to the function. Think of that reference like a pointer that points to an internal representation of the function itself. The reference itself can be held in a variable and reassigned to other variables or passed as an argument to a function. And placing parentheses after a function reference causes the function to be invoked.

Sharpen your pencil

Below we've created a function and assigned it to the variable fun.

```
function fun(echo) {
    console.log(echo);
};
```



Work your way through this code and write the resulting output on this page. Do this with your brain before you attempt it with your computer.

```
fun("hello");
```

```
function boo(aFunction) {
    aFunction("boo");
}
```

```
boo(fun);
```

```
console.log(fun);
```

```
fun(boo);
```

```
var moreFun = fun;
```

```
moreFun("hello again");
```

Extra credit! (A preview of what's coming up...)

```
function echoMaker() {
    return fun;
}

var bigFun = echoMaker();
bigFun("Is there an echo?");
```

Super important: check and understand the answers before moving on!

Returning functions from functions

At this point we've exercised two of our first class requirements, assigning functions to variables and passing functions to functions, but we haven't yet seen an example of returning a function from a function.

- ☑ You can assign functions to variables.
- ☑ You can pass functions to functions.
- ❑ You can return functions from functions.

We're doing this now.

Let's extend the airline example a bit and explore why and where we might want to return a function from a function. To do that, we'll add another property to each of our passengers, the ticket property, which is set to "coach" or "firstclass" depending on the type of ticket the passenger has purchased:

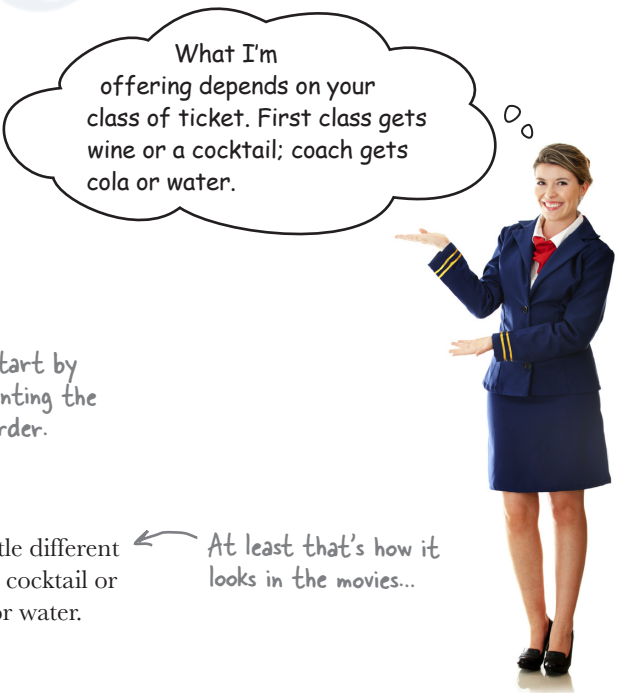
```
var passengers = [  { name: "Jane Doloop", paid: true, ticket: "coach" },
                    { name: "Dr. Evel", paid: true, ticket: "firstclass" },
                    { name: "Sue Property", paid: false, ticket: "firstclass" },
                    { name: "John Funcall", paid: true, ticket: "coach" } ];
```

With that addition, we'll write some code that handles the various things a flight attendant needs to do:

Here are all the things the flight attendant needs to do to serve each passenger.

```
function serveCustomer(passenger) {
    // get drink order
    // get dinner order
    // pick up trash
}
```

Let's start by implementing the drink order.



Now as you might know, service in first class tends to be a little different from the service in coach. In first class you're able to order a cocktail or wine, while in coach you're more likely to be offered a cola or water.

At least that's how it looks in the movies...

Writing the flight attendant drink order code

Now your first attempt might look like this:

```
function serveCustomer(passenger) {
    if (passenger.ticket === "firstclass") {
        alert("Would you like a cocktail or wine?");
    } else {
        alert("Your choice is cola or water.");
    }

    // get dinner order
    // pick up trash
}
```

If the passenger's ticket is a first class ticket then we issue an alert to ask if they'd like a cocktail or a wine.

If they have a coach ticket, then ask if they want a cola or water.

Not bad. For simple code this works well: we're taking the passenger's ticket and then displaying an alert based on the type of ticket they purchased. But let's think through some potential downsides of this code. Sure, the code to take a drink order is simple, but what happens to the `serveCustomer` function if the problem becomes more complex. For instance, we might start serving three classes of passengers (firstclass, business and coach. And what about premium economy, that's four!). What happens if the drink offerings get more complex? Or what if the choice of drinks is based on the location of the originating or destination airport?

For instance they typically only serve Maitais to first class on trips to Hawaii (or so we've been told).

If we have to deal with these complexities then `serveCustomer` is quickly going to become a large function that is a lot more about managing drinks than serving customers, and when we design functions, a good rule of thumb is to have them do only one thing, but do it really well.

Re-read all the potential issues listed in the last two paragraphs on this page. Then, think about what code design would allow us to keep serveCustomer focused, yet also allow for expansion of our drink-serving capability in the future.

The flight attendant drink order code: a different approach

Our first pass wasn't bad, but as you can see this code could be problematic over time as the drink serving code gets more complex. Let's rework the code a little, as there's another way we can approach this by placing the logic for the drink orders in a separate function. Doing so allows us to hide away all that logic in one place, and it also gives us a well-defined place to go if we need to update the drink order code:

Here we're creating a new function createDrinkOrder, which is passed a passenger.

```
function createDrinkOrder(passenger) {
    if (passenger.ticket === "firstclass") {
        alert("Would you like a cocktail or wine?");
    } else {
        alert("Your choice is cola or water.");
    }
}
```

And we'll place all the logic for the drink order here.

Now this code is no longer polluting the serveCustomer function with a lot of drink order logic.

Now we can revisit the `serveCustomer` function and remove all the drink order logic, replacing it with a call to this new function.

```
function serveCustomer(passenger) {
    if (passenger.ticket === "firstclass") {
        alert("Would you like a cocktail or wine?");
    } else {
        alert("Your choice is cola or water.");
    }
    createDrinkOrder(passenger);

    // get dinner order
    // pick up trash
}
```

We're removing the logic from serveCustomer...

And we'll replace the original, inline logic with a call to createDrinkOrder.

The createDrinkOrder function is passed the passenger that was passed into serveCustomer.

That's definitely going to be more readable with a single function call replacing all the inline drink order logic. It's also conveniently put all the drink order code in one, easy-to-find place. But, before we give this code a test drive, hold on, we've just heard about another issue…

Wait, we need more drinks!

Stop the presses, we've just heard that one drink order is not enough on a flight. In fact, the flight attendants say a typical flight looks more like this:


```
function serveCustomer(passenger) {
    createDrinkOrder(passenger);
    // get dinner order
    createDrinkOrder(passenger);
    createDrinkOrder(passenger);
    // show movie
    createDrinkOrder(passenger);
    // pick up trash
}
```

We've updated the code to reflect the fact we're calling createDrinkOrder a lot during the flight.

Now, on the one hand we designed our code well, because adding additional calls to `createDrinkOrder` works just fine. But, on the other hand, we're unnecessarily recomputing what kind of passenger we're serving in `createDrinkOrder` every time we take an order.

"But it's only a few lines of code." you say? Sure, but this is a simple example in a book. What if in the real world you had to check the ticket type by communicating with a web service from a mobile device? That gets time consuming and expensive.

Don't worry though, because a first class function just rode in on a white horse to save us. You see, by making use of the capability to return functions from functions we can fix this problem.

Sharpen your pencil

What do you think this code does? Can you come up with some examples of how to use it?

```
function addN(n) {
    var adder = function(x) {
                    return n + x;
                };
    return adder;
}
```

Answer here.

Taking orders with first class functions

Now it's time to wrap your head around how a first class function can help this situation. Here's the plan: rather than calling `createDrinkOrder` multiple times per passenger, we're instead going to call it once, and have it hand us back a function that knows how to do a drink order for that passenger. Then, when we need to take a drink order, we just call that function.

Let's start by redefining `createDrinkOrder`. Now when we call it, it will package up the code to take a drink order into a function and return the function for us to use when we need it.

Here's the new createDrinkOrder. It's going to return a function that knows how to take a drink order.

```
function createDrinkOrder(passenger) {
    var orderFunction;

    if (passenger.ticket === "firstclass") {
        orderFunction = function() {
            alert("Would you like a cocktail or wine?");
        };
    } else {
        orderFunction = function() {
            alert("Your choice is cola or water.");
        };
    }
    return orderFunction;
}
```

First, create a variable to hold the function we want to return.

Now, we execute the conditional code to check the passenger's ticket type only once.

If the passenger is first class, we create a function that knows how to take a first class order.

Otherwise create a function to take an coach class order.

And return the function.

Now let's rework `serveCustomer`. We'll first call `createDrinkOrder` to get a function that knows how to take the passenger's order. Then, we'll use that same function over and over to take a drink order from the passenger.

```
function serveCustomer(passenger) {
  var getDrinkOrderFunction = createDrinkOrder(passenger);
  getDrinkOrderFunction();
  // get dinner order
  getDrinkOrderFunction();
  getDrinkOrderFunction();
  // show movie
  getDrinkOrderFunction();
  // pick up trash
}
```

getDrinkOrder now returns a function, which we store in the getDrinkOrderFunction variable.

We use the function we get back from createDrinkOrder whenever we need to get a drink order for this passenger.

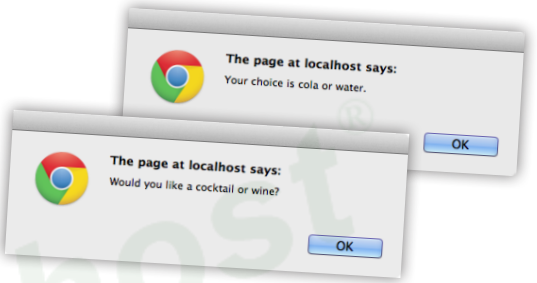
Test ~~drive~~ flight

Let's test the new code. To do that we need to write a quick function to iterate over all the passengers and call `serveCustomer` for each passenger. Once you've added the code to your file, load it in the browser and take some orders.

```
function servePassengers(passengers) {
    for (var i = 0; i < passengers.length; i++) {
        serveCustomer(passengers[i]);
    }
}

servePassengers(passengers);
```

All we're doing here is iterating over the passengers in the passengers array, and calling serveCustomer on each passenger.

And of course we need to call servePassengers to get it all going. (Be prepared, there are a lot of alerts!)



there are no Dumb Questions

Q: Just to make sure I understand... when we call createDrinkOrder, we get back a function that we have to call again to get the drink order?

A: That's right. We first call createDrinkOrder to get back a function, getDrinkOrderFunction, that knows how to ask a passenger for an order, and then we call that function every time we want to take the order. Notice that getDrinkOrderFunction is a lot simpler than createDrinkOrder: all getDrinkOrderFunction does is alert, asking for the passenger's order.

Q: So how does getDrinkOrderFunction know which alert to show?

A: Because we created it specifically for the passenger based on their ticket. Look back at createDrinkOrder again. The function we're returning corresponds to the passenger's ticket type: if the passenger is in first class, then getDrinkOrderFunction is created to show an alert asking for a first class order. But if the passenger is in coach, then getDrinkOrderFunction is created to show an alert asking for a coach order. By returning the correct kind of function for that specific passenger's ticket type, the ordering function is simple, fast, and easy to call each time we need to take an order.

Q: This code serves one passenger a drink, shows the movie, etc. Don't flight attendants usually serve a drink to all the passengers and show the movie to all passengers and so on?

A: See we were testing you! You passed. You're exactly right; this code applies the entire serveCustomer function to a single passenger at a time. That's not really how it works in the real world. But, this is meant to be a simple example to demonstrate a complex topic (returning functions), and it's not perfect. But now that you've *pointed out our mistake*… students, take out a sheet of paper and:

BRAIN POWER

How would you rework the code to serve drinks, dinner and a movie to all the passengers, and do it without endlessly recomputing their order based on their ticket class? Would you use first class functions?

Your job is to add a third class of service to our code. Add "premium economy" class ("premium" for short). Premium economy gets wine in addition to cola or water. Also, implement getDinnerOrderFunction with the following menu:

First class: chicken or pasta

Premium economy: snack box or cheese plate

Coach: peanuts or pretzels

Check your answer at the end of the chapter! And don't forget to test your code.

Make sure you use first class functions to implement this!

Webville Cola

Webville Cola needs a little help managing the code for their product line. To give them a hand, let's take a look at the data structure they use to hold the sodas they produce:

Looks like they're storing their products as an array of objects. Each object is a product.

```
var products = [ { name: "Grapefruit", calories: 170, color: "red", sold: 8200 },
                 { name: "Orange", calories: 160, color: "orange", sold: 12101 },
                 { name: "Cola", calories: 210, color: "caramel", sold: 25412 },
                 { name: "Diet Cola", calories: 0, color: "caramel", sold: 43922 },
                 { name: "Lemon", calories: 200, color: "clear", sold: 14983 },
                 { name: "Raspberry", calories: 180, color: "pink", sold: 9427 },
                 { name: "Root Beer", calories: 200, color: "caramel", sold: 9909 },
                 { name: "Water", calories: 0, color: "clear", sold: 62123 }
               ];
```

In each product they're storing a name, number of calories, color and number of bottles sold per month.

We really need some help sorting these products. We need to sort them by every possible property: name, calories, color, sales numbers. Of course we want to get this done as efficiently as we can, and also keep it flexible so we can sort in lots of different ways.

Webville Cola's analytics guy

Frank: Hey guys, I got a call from Webville Cola and they need help with their product data. They want to be able to sort their products by any property, like name, bottles sold, soda color, calories per bottle, and so on, but they want it flexible in case they add more properties in the future.

Joe: How are they storing this data?

Frank: Oh, each soda is an object in an array, with properties for name, number sold, calories…

Joe: Got it.

Frank: My first thought was just to search for a simple sort algorithm and implement it. Webville Cola doesn't have many products so it just needs to be simple.

Jim: Oh, I've got a simpler way than that, but it requires you use your knowledge of first class functions.

Frank: I like hearing simple! But how do first class functions fit in? That sounds complicated to me.

Jim: Not at all. It's as easy as writing a function that knows how to compare two values, and then passing that to another function that does the real sorting for you.

Joe: How does the function we're writing work exactly?

Jim: Well, instead of handling the entire sort, all you need to do is write a function that knows how to compare two values. Say you want to sort by a product property like the number of bottles sold. You set up a function like this:

```
function compareSold(product1, product2) {
  // code to compare here
}
```

This function needs to take two products and then compare them.

We can fill in the details of the code in a minute, but for now, the key is that once you have this function you just pass it to a sort function and that sort function does the work for you—it just needs you to help it know how to compare things.

Frank: Wait, where is this sort function?

Jim: It's actually a method that you can call on any array. So you can call the `sort` method on the `products` array and pass it this compare function we're going to write. And, when `sort` is done, the `products` array will be sorted by whatever criteria `compareSold` used to sort the values.

Joe: So if I'm sorting how many bottles are sold, those are numbers, so the `compareSold` function just needs to determine which value is less or greater?

Jim: Right. Let's take a closer look at how the array sort works…

How the array sort method works

JavaScript arrays provide a `sort` method that, given a function that knows how to compare two items in the array, sorts the array for you. Here's the big picture of how this works and how your comparison function fits in: sort algorithms are well known and widely implemented, and the great thing about them is that sorting code can be reused for just about any set of items. But there's one catch: to know how to sort a specific set of items, the sort code needs to know how those items compare. Think about sorting a set of numbers versus sorting a list of names versus sorting a set of objects. The way we compare values depends on the type of the items: for numbers we use <, > and == , for strings we compare them alphabetically (in JavaScript, you can do that with <, > and ==) and for objects we'd have some custom way of comparing them based on their properties.

Let's look at a simple example before we move on to Webville Cola's products array. We'll use a simple array of numbers and use the `sort` method to put them in ascending order. Here's the array:

```
var numbersArray = [60, 50, 62, 58, 54, 54];
```

Next we need to write our own function that knows how to compare two values in the array. Now, this is an array of numbers, so our function will need to compare two numbers at a time. Assume we're going to sort the numbers in ascending order; for that the `sort` method expects us to return something greater than 0 if the first number is greater than the second, 0 if they are equal, and something less than 0 if the first number is less than the second. Like this:

This array is made of numbers, so we're going to be comparing two numbers at a time.

```
function compareNumbers(num1, num2) {
    if (num1 > num2) {
        return 1;
    } else if (num1 === num2) {
        return 0;
    } else {
        return -1;
    }
}
```

We first check to see if num1 is greater than num2. If it is, we return 1.

If they are equal then we return 0.

And finally, if num1 is less than num2, we return –1.

Serious Tip

JavaScript arrays have many useful methods you can use to manipulate arrays in various ways. A great reference on all these methods and how to use them is *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly).

Sharpen your pencil

You know that the comparison function we pass to sort needs to return a number greater than 0, equal to 0, or less than 0 depending on the two items we're comparing: if the first item is greater than second, we return a value greater than 0; if first item is equal to the second, we return 0; and if the first item is less than the second, we return a value less than 0.

Can you use this knowledge with the fact that we're comparing two numbers in compareNumbers to rewrite compareNumbers using much less code?

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

Putting it all together

Now that we've written a compare function, all we need to do is call the `sort` method on `numbersArray`, and pass it the function. Here's how we do that:

```
var numbersArray = [60, 50, 62, 58, 54, 54];
numbersArray.sort(compareNumbers);
console.log(numbersArray);
```

We call the sort method on the array, passing it the compareNumbers function.

And when sort is complete the array is sorted in ascending order, and we display that in the console as a sanity check.

Here's the array sorted in ascending order.

Note that the sort method is destructive, in that it changes the array, rather than returning a new array that is sorted.

JavaScript console

```
[50, 54, 54, 58, 60, 62]
```

Exercise

The sort method has sorted numbersArray in ascending order because when we return the values 1, 0 and -1, we're telling the sort method:

1: place the first item after the second item

0: the items are equivalent, you can leave them in place

-1: place the first item before the second item.

Changing your code to sort in descending order is a matter of inverting this logic so that 1 means place the second item after the first item, and -1 means place the second item before the first item (0 stays the same). Write a new compare function for descending order:

```
function compareNumbersDesc(num1, num2) {
    if (_____ > _____) {
        return 1;
    } else if (num1 === num2) {
        return 0;
    } else {
        return -1;
    }
}
```

Meanwhile back at Webville Cola

It's time to help out Webville Cola with your new knowledge of array sorting. Of course all we really need to do is write a comparison function for them, but before we do that let's quickly review the `products` array again:

But we don't have to tell them that.

```
var products = [ { name: "Grapefruit", calories: 170, color: "red", sold: 8200 },
                 { name: "Orange", calories: 160, color: "orange", sold: 12101 },
                 { name: "Cola", calories: 210, color: "caramel", sold: 25412 },
                 { name: "Diet Cola", calories: 0, color: "caramel", sold: 43922 },
                 { name: "Lemon", calories: 200, color: "clear", sold: 14983 },
                 { name: "Raspberry", calories: 180, color: "pink", sold: 9427 },
                 { name: "Root Beer", calories: 200, color: "caramel", sold: 9909 },
                 { name: "Water", calories: 0, color: "clear", sold: 62123 }
               ];
```

Remember, each item in the products array is an object. We don't want to compare the objects to one another, we want to compare specific properties, like sold, in the objects.

So what are we going to sort first? Let's start with sorting by the number of bottles sold, in ascending order. To do this we'll need to compare the `sold` property of each object. Now one thing you should take note of is, because this is an array of product objects, the compare function is going to be passed two objects, not two numbers:

compareSold takes two cola product objects, and compares the sold property of colaA to the sold property of colaB.

```
function compareSold(colaA, colaB) {
    if (colaA.sold > colaB.sold) {
        return 1;
    } else if (colaA.sold === colaB.sold) {
        return 0;
    } else {
        return -1;
    }
}
```

This function will make the sort method sort the colas by number of bottles sold in ascending order.

Feel free to simplify this code like you did in the earlier exercise if you want!

And of course, to use the `compareSold` function to sort the `products` array, we simply call the `products` array's `sort` method:

```
products.sort(compareSold);
```

Remember that the sort method can be used for any kind of array (numbers, strings, objects), and for any kind of sort (ascending or descending). By passing in a compare function, we get flexibility and code reuse!

Take sorting for a test drive

Time to test the first Webville Cola code. You'll find all the code from the last few pages consolidated below along with some extras to test this out properly. So, just create a simple HTML page with this code ("cola.html") and give it some QA:

```
var products = [ { name: "Grapefruit", calories: 170, color: "red", sold: 8200 },
                 { name: "Orange", calories: 160, color: "orange", sold: 12101 },
                 { name: "Cola", calories: 210, color: "caramel", sold: 25412 },
                 { name: "Diet Cola", calories: 0, color: "caramel", sold: 43922 },
                 { name: "Lemon", calories: 200, color: "clear", sold: 14983 },
                 { name: "Raspberry", calories: 180, color: "pink", sold: 9427 },
                 { name: "Root Beer", calories: 200, color: "caramel", sold: 9909 },
                 { name: "Water", calories: 0, color: "clear", sold: 62123 }
               ];

function compareSold(colaA, colaB) {
    if (colaA.sold > colaB.sold) {
        return 1;
    } else if (colaA.sold === colaB.sold) {
        return 0;
    } else {
        return -1;
    }
}

function printProducts(products) {
    for (var i = 0; i < products.length; i++) {
        console.log("Name: " + products[i].name +
                    ", Calories: " + products[i].calories +
                    ", Color: " + products[i].color +
                    ", Sold: " + products[i].sold);
    }
}

products.sort(compareSold);
printProducts(products);
```

Here's the compare function we'll pass to sort...

... and here's a new function we wrote to print the products so they look nice in the console. (If you just write console.log(products), you can see the output, but it doesn't look very good).

So first we sort the products, using compareSold...

... and then print the results.

JavaScript console

```
Name: Grapefruit, Calories: 170, Color: red, Sold: 8200
Name: Raspberry, Calories: 180, Color: pink, Sold: 9427
Name: Root Beer, Calories: 200, Color: caramel, Sold: 9909
Name: Orange, Calories: 160, Color: orange, Sold: 12101
Name: Lemon, Calories: 200, Color: clear, Sold: 14983
Name: Cola, Calories: 210, Color: caramel, Sold: 25412
Name: Diet Cola, Calories: 0, Color: caramel, Sold: 43922
Name: Water, Calories: 0, Color: clear, Sold: 62123
```

Here's our output running the code using the compareSold sort function. Notice the products are in order by the number of bottles sold.

Exercise

Now that we have a way to sort colas by the sold property, it's time to write compare functions for each of the other properties in the product object: name, calories, and color. Check the output you see in the console carefully; make sure for each kind of sort, the products are sorted correctly. Check the answer at the end of the chapter.

Write your solutions for the remaining three sort functions below.

```
function compareName(colaA, colaB) {

}
function compareCalories(colaA, colaB) {

}

function compareColor(colaA, colaB) {

}
products.sort(compareName);
console.log("Products sorted by name:");
printProducts(products);

products.sort(compareCalories);
console.log("Products sorted by calories:");
printProducts(products);

products.sort(compareColor);
console.log("Products sorted by color:");
printProducts(products);
```

Hint: you can use <, > and == to sort alphabetically too!

For each new compare function, we call sort, and display the results in the console.

BULLET POINTS

- There are two ways to define a function: with a **function declaration** and with a **function expression**.
- A **function reference** is a value that refers to a function.
- Function declarations are handled before your code is evaluated.
- Function expressions are evaluated at runtime with the rest of your code.
- When the browser evaluates a function declaration, it creates a function as well as a variable with the same name as the function, and stores the function reference in the variable.
- When the browser evaluates a function expression, it creates a function, and it's up to you what to do with the function reference.
- **First class** values can be assigned to variables, included in data structures, passed to functions, or returned from functions.
- A function reference is a first class value.
- The array **sort** method takes a function that knows how to compare two values in an array.
- The function you pass to the sort method should return one of these values: a number greater than 0, 0, or a number less than 0.

Exercise Solutions

Sharpen your pencil Solution

What deductions can you make about function declarations and function expressions given how the browser treats the quack and fly code? Check each statement that applies. Check your answer at the end of the chapter before you go on.

- ☑ Function declarations are evaluated before the rest of the code is evaluated.
- ☑ Function expressions get evaluated later, with the rest of the code.
- ☑ A function declaration doesn't return a reference to a function; rather it creates a variable with the name of the function and assigns the new function to it.
- ☑ A function expression returns a reference to the new function created by the expression.
- ☑ The process of invoking a function created by a declaration is exactly the same for one created with an expression.
- ☑ You can hold function references in variables.
- ☑ Function declarations are statements; function expressions are used in statements.
- ❑ Function declarations are the tried and true way to create functions.
- ❑ You always want to use function declarations because they get evaluated earlier.

Not necessarily!

BE the Browser Solution

Below, you'll find JavaScript code. Your job is to play like you're the browser evaluating the code. In the space to the right, record each function as it gets created. Remember to make two passes over the code: the pass that processes declarations, and the second pass that handles expressions.

```
var midi = true;
var type = "piano";
var midiInterface;

function play(sequence) {
    // code here
}
var pause = function() {
    stop();
}
function stop() {
    // code here
}

function createMidi() {
   // code here
}

if (midi) {
    midiInterface = function(type) {
        // code here
    };
}
```

Write, in order, the names of the functions as they are created. If a function is created with a function expression put the name of the variable it is assigned to. We've done the first one for you.

play

stop

createMidi

pause

midiInterface

Sharpen your pencil Solution

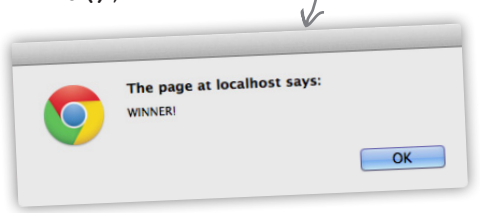
To get the idea of functions as values into your brain, let's play a little game of chance. Try the shell game. Did you win or lose? Give it a try and find out. Here's our solution.

```
var winner = function() { alert("WINNER!") };
var loser = function() { alert("LOSER!") };
// let's test as a warm up
winner();
// let's assign to other variables for practice
var a = winner;
var b = loser;
var c = loser;
a();
b();
// now let's try your luck with a shell game
c = a;
a = b;
b = c;
c = a;
a = c;
a = b;
b = c;
a();
```

Remember, these variables hold references to the winner and loser functions. We can assign and reassign these references to other variables, just like with any value.

Remember, at any time, we can invoke a reference to a function.

c is winner
a is loser
b is winner
c is loser
a is loser
a is winner
b is loser
invoking a...
winner!!!



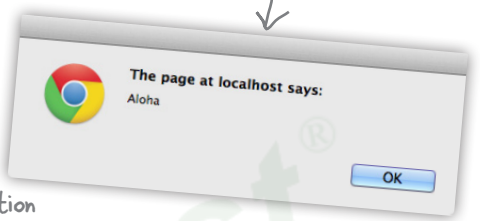
Sharpen your pencil Solution

Let's get your brain warmed up for passing your first function to another function. Evalute the code below (in your head) and see what you come up with. Here's our solution:

```
function sayIt(translator) {
    var phrase = translator("Hello");
    alert(phrase);
}

function hawaiianTranslator(word) {
    if (word == "Hello") return "Aloha";
    if (word == "Goodbye") return "Aloha";
}

sayIt(hawaiianTranslator);
```

We're defining a function that takes a function as an argument, and then calls that function.

We're passing the function hawaiianTranslator to the function sayIt.



Exercise Solution

Your turn again: write a function that prints a passenger's name and whether or not they have paid to console.log. Pass your function to processPassengers to test it. We've started the code for you below; you just need to finish it up. Here's our solution.

```
function printPassenger(passenger) {
    var message = passenger.name;
    if (passenger.paid === true) {
        message = message + " has paid";
    } else {
        message = message + " has not paid";
    }
    console.log(message);
    return false;
}

processPassengers(passengers, printPassenger);
```

This return value doesn't matter that much because we're ignoring the result from processPassengers in this case.

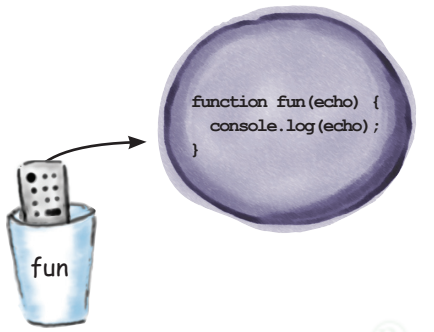
JavaScript console

```
Jane Doloop has paid
Dr. Evel has paid
Sue Property has not paid
John Funcall has paid
```

Sharpen your pencil Solution

Below we've created a function and assigned it to the variable fun.

```
function fun(echo) {
    console.log(echo);
};
```



Warning note: your browser may display different values in the console for the fun and boo functions. Try this exercise in a couple of different browsers.

Work your way through this code and write the resulting output on this page. Do this with your brain before you attempt it with your computer.

```
fun("hello");

function boo(aFunction) {
    aFunction("boo");
}

boo(fun);

console.log(fun);

fun(boo);

var moreFun = fun;

moreFun("hello again");
```

hello

boo

function fun(echo) { console.log(echo); }

function boo(aFunction) { aFunction("boo"); }

hello again

Extra credit! (A preview of what's coming up...)

```
function echoMaker() {
    return fun;
}

var bigFun = echoMaker();
bigFun("Is there an echo?");
```

Is there an echo?

Super important: check and understand the answers before moving on!

Your job is to add a third class of service to our code. Add "premium economy" class ("premium" for short). Premium economy gets wine in addition to cola or water. Also, implement getDinnerOrderFunction with the following menu:

First class: chicken or pasta

Premium economy: snack box or cheese plate

Coach: peanuts or pretzels

Here's our solution.

```
var passengers = [  { name: "Jane Doloop", paid: true, ticket: "coach" },
                    { name: "Dr. Evel", paid: true, ticket: "firstclass" },
                    { name: "Sue Property", paid: false, ticket: "firstclass" },
                    { name: "John Funcall", paid: true, ticket: "premium" } ];

function createDrinkOrder(passenger) {
    var orderFunction;
    if (passenger.ticket === "firstclass") {
        orderFunction = function() {
            alert("Would you like a cocktail or wine?");
        };
    } else if (passenger.ticket === "premium") {
        orderFunction = function() {
            alert("Would you like wine, cola or water?");
        };
    } else {
        orderFunction = function() {
            alert("Your choice is cola or water.");
        };
    }
    return orderFunction;
}
```

We've upgraded John Funcall to premium economy for this flight (so we can test our new code).

Here's the new code to handle the premium economy class. Now we're returning one of three different order functions depending on the ticket type of the passenger.

Notice how handy it is to have all this logic encapsulated in one function that knows how to create the right kind of order function for a customer.

And when we make an order, we don't have to do this logic; we have an order function that is customized for the passenger already!


```
function createDinnerOrder(passenger) {
    var orderFunction;
    if (passenger.ticket === "firstclass") {
        orderFunction = function() {
            alert("Would you like chicken or pasta?");
        };
    } else if (passenger.ticket === "premium") {
        orderFunction = function() {
            alert("Would you like a snack box or cheese plate?");
        };
    } else {
        orderFunction = function() {
            alert("Would you like peanuts or pretzels?");
        };
    }
    return orderFunction;
}

function serveCustomer(passenger) {
    var getDrinkOrderFunction = createDrinkOrder(passenger);
    var getDinnerOrderFunction = createDinnerOrder(passenger);

    getDrinkOrderFunction();

    // get dinner order
    getDinnerOrderFunction();

    getDrinkOrderFunction();
    getDrinkOrderFunction();
    // show movie
    getDrinkOrderFunction();
    // pick up trash
}

function servePassengers(passengers) {
    for (var i = 0; i < passengers.length; i++) {
        serveCustomer(passengers[i]);
    }
}

servePassengers(passengers);
```

We've added a completely new function, createDinnerOrder, to create a dinner ordering function for a passenger.

It works in the same way that createDrinkOrder does: by looking at the passenger ticket type and returning an order function customized for that passenger.

We create the right kind of dinner order function for the passenger...

...and then call it whenever we want to take a passenger's dinner order.

Sharpen your pencil Solution

What do you think this code does? Can you come up with some examples of how to use it? Here's our solution.

```
function addN(n) {
    var adder = function(x) {
                    return n + x;
                };
    return adder;
}
```

This function takes one argument n. It then creates a function that also takes one argument, x, and adds n and x together. That function is returned.

So we used it to create a function that always adds 2 to a number. Like this:

```
var add2 = addN(2);
console.log(add2(10));
console.log(add2(100));
```

Exercise Solution

The sort method has sorted numbersArray in ascending order because when we return the values 1, 0 and -1, we're telling the sort method:

1: place the first item after the second item

0: the items are equivalent, you can leave them in place

-1: place the first item before the second item.

Changing your code to sort in descending order is a matter of inverting this logic so that 1 means place the second item after the first item, and -1 means place the second item before the first item (0 stays the same). Write a new compare function for descending order:

```
function compareNumbersDesc(num1, num2) {
    if ( num2 > num1 ) {
        return 1;
    } else if (num1 == num2) {
        return 0;
    } else {
        return -1;
    }
}
```

Sharpen your pencil Solution

You know that the comparison function we pass to sort needs to return a number greater than 0, equal to 0, or less than 0 depending on the two items we're comparing: if the first item is greater than second, we return a value greater than 0; if first item is equal to the second, we return 0; and if the first item is less than the second, we return a value less than 0.

Can you use this knowledge with the fact we're comparing two numbers in compareNumbers to rewrite compareNumbers using much less code?

Here's our solution:

```
function compareNumbers(num1, num2) {
    return num1 - num2;
}
```

We can make this function a single line of code by simply returning the result of substracting num2 from num1. Run through a couple of examples to see how this works. And remember sort is expecting a number greater than, equal to, or less than 0, not specifically 1, 0, –1 (although you'll see a lot of code return those values for sort).

Exercise Solution

Now that we have a way to sort colas by the sold property, it's time to write compare functions for each of the other properties in the product object: name, calories, and color. Check the output you see in the console carefully; make sure for each kind of sort, the products are sorted correctly. Here's our solution.

Here's our implementation of each compare function.

```
function compareName(colaA, colaB) {
    if (colaA.name > colaB.name) {
        return 1;
    } else if (colaA.name === colaB.name) {
        return 0;
    } else {
        return -1;
    }
}

function compareCalories(colaA, colaB) {
    if (colaA.calories > colaB.calories) {
        return 1;
    } else if (colaA.calories === colaB.calories) {
        return 0;
    } else {
        return -1;
    }
}

function compareColor(colaA, colaB) {
    if (colaA.color > colaB.color) {
        return 1;
    } else if (colaA.color === colaB.color) {
        return 0;
    } else {
        return -1;
    }
}

products.sort(compareName);
console.log("Products sorted by name:");
printProducts(products);

products.sort(compareCalories);
console.log("Products sorted by calories:");
printProducts(products);

products.sort(compareColor);
console.log("Products sorted by color:");
printProducts(products);
```

For each new compare function, we call sort, and display the results in the console.