8 bringing it all together

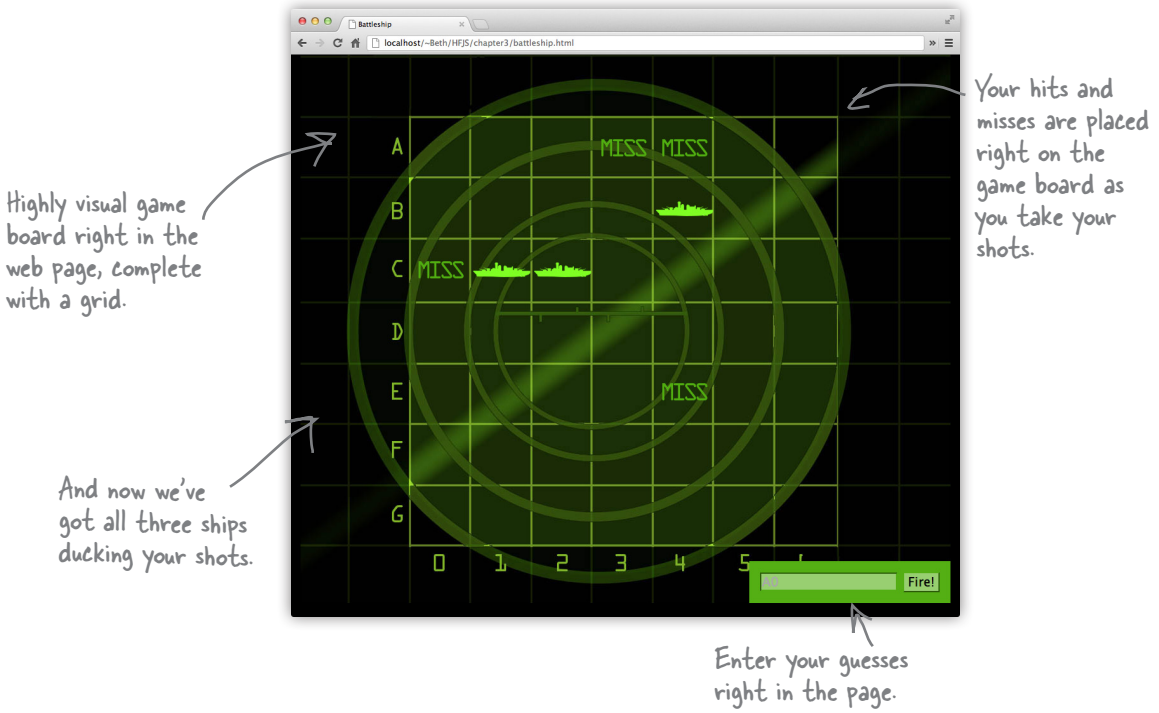
Building an app

Put on your toolbelt. That is, the toolbelt with all your new coding skills, your knowledge of the DOM, and even some HTML & CSS. We're going to bring everything together in this chapter to create our first true **web application**. No more **silly toy games** with one battleship and a single row of hiding places. In this chapter we're building the **entire experience**: a nice big game board, multiple ships and user input right in the web page. We're going to create the page structure for the game with HTML, visually style the game with CSS, and write JavaScript to code the game's behavior. Get ready: this is an all out, pedal to the metal development chapter where we're going to lay down some serious code.

This time, let's build a REAL Battleship game

Sure, you can feel good because back in Chapter 2 you built a nice little battleship game from scratch, but let's admit it: that was a bit of a *toy* game—it worked, it was playable, but it wasn't exactly the game you'd impress your friends with, or use to raise your first round of venture capital. To really impress, you'll need a visual game board, snazzy battleship graphics, and a way for players to enter their moves right in the game (rather than a generic browser dialog box). You'll also want to improve the previous version by supporting all three ships.

In other words, you'll want to create something like this:



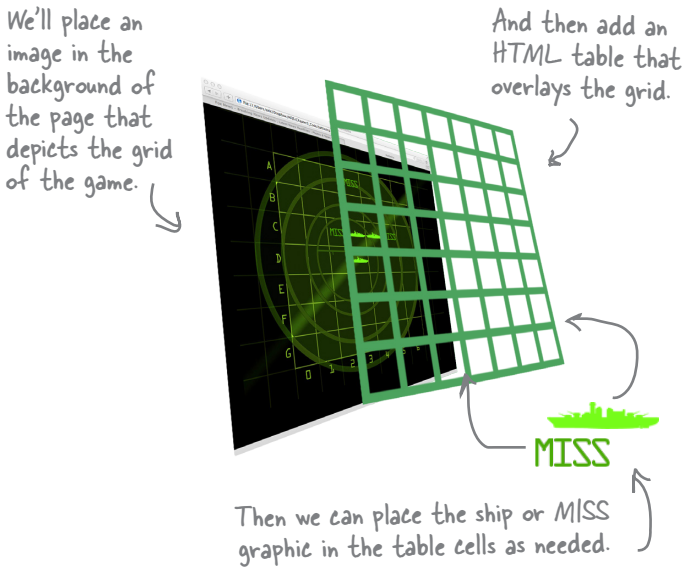
Forget JavaScript for a minute... look at the Battleship mockup above. If you focus on the structure and visual represention of the page, how would you create it using HTML and CSS?

Stepping back... to HTML and CSS

To create a modern, interactive web page, or *app*, you need to work with three technologies: HTML, CSS and JavaScript. You already know the mantra "HTML is for structure, CSS is for style and JavaScript is for behavior." But rather than just stating it, in this chapter we're going to fully embody it. And we're going to start with the HTML and CSS first.

Our first goal is going to be to reproduce the look of the game board on the previous page. But not *just* reproduce it; we need to implement the game board so it has a structure we can use in JavaScript to take player input and place hits, misses and messages on the page.

To pull that off we're going to do things like use an image in the background to give us the slick grid over a radar look, and then we'll lay a more functional HTML table over that so we can place things (like ships) on top of it. We'll also use an HTML form to get the player input.

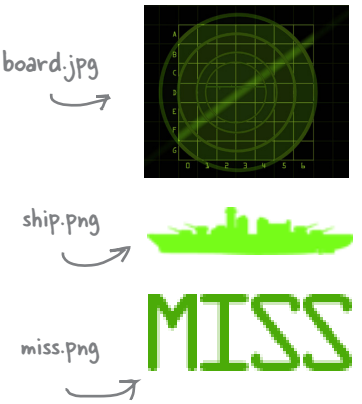


So, let's build this game. We're going to take a step back and spend a few pages on the crucial HTML and CSS, but once we have that in place, we'll be ready for the JavaScript.

GET YOUR BATTLESHIP TOOLKIT

Here's a toolkit to get you started on this new version of Battleship.

INVENTORY includes...



This toolkit contains three images, "board.jpg", which is the main background board for the game including the grid; "ship.png", which is a small ship for placement on the board—notice that it is a PNG image with transparency, so it will lay right on top of the background—and finally we have "miss.png", which is also meant to be placed on the board. True to the original game, when we hit a ship we place a ship in the corresponding cell, and when we miss we place a miss graphic there.

Download everything you need for the game at http://wickedlysmart.com/hfjs

Creating the HTML page: the Big Picture

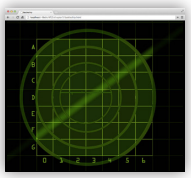
Here's the plan of attack for creating the Battleship HTML page:

1. First we'll concentrate on the background of the game, which includes setting the background image to black and placing the radar grid image in the page.

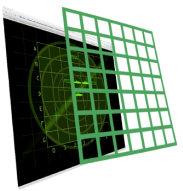
2. Next we'll create an HTML table and lay it on top of the background image. Each cell in the table will represent a board cell in the game.

3. Then we'll add an HTML form element where players can enter their guesses, like "A4". We'll also add an area to display messages, like "You sank my battleship!"

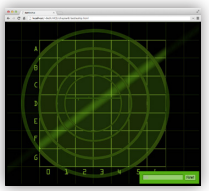
4. Finally, we'll figure out how to use the table to place the images of a battleship (for a hit) and a MISS (for a miss) into the board.



We're placing an image in the background to give the game its cool, green phosphorus radar feel.



An HTML table on top of the background creates a game board for the game to play out in.



An HTML form for player input.

We'll use these images and place them into the table as needed.

Step 1: The Basic HTML

Let's get started! First we need an HTML page. We're going to start by creating a simple HTML5-compliant page; we'll also add some styling for the background image. In the page we're going to place a `<body>` element that contains a single `<div>` element. This `<div>` element is going to hold the game grid.

Go ahead and check out the next page that contains our starter HTML and CSS.

A little rusty?

If you're feeling a bit rusty on your HTML and CSS, *Head First HTML and CSS* was written to be the companion to this book.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Battleship</title>
    <style>
      body {
        background-color: black;
      }

      div#board {
        position: relative;
        width: 1024px;
        height: 863px;
        margin: auto;
        background: url("board.jpg") no-repeat;
      }
    </style>
  </head>
  <body>
    <div id="board">

    </div>
    <script src="battleship.js"></script>
  </body>
</html>
```

Just a regular HTML page.

And we want the background of the page to be black.

We want the game board to stay in the middle of the page, so we're setting the width to 1024px (the width of the game board), and the margins to auto.

Here's where we add the "board.jpg" image to the page, as the background of the "board" element. We're positioning this relative, so that we can position the table we add in the next step relative to this .

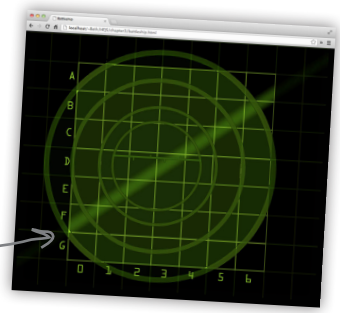
We're going to put the table for the game board and the form for getting user input here.

We'll put our code in the file "battleship.js". Go ahead and create a blank file for that.

A Test Drive

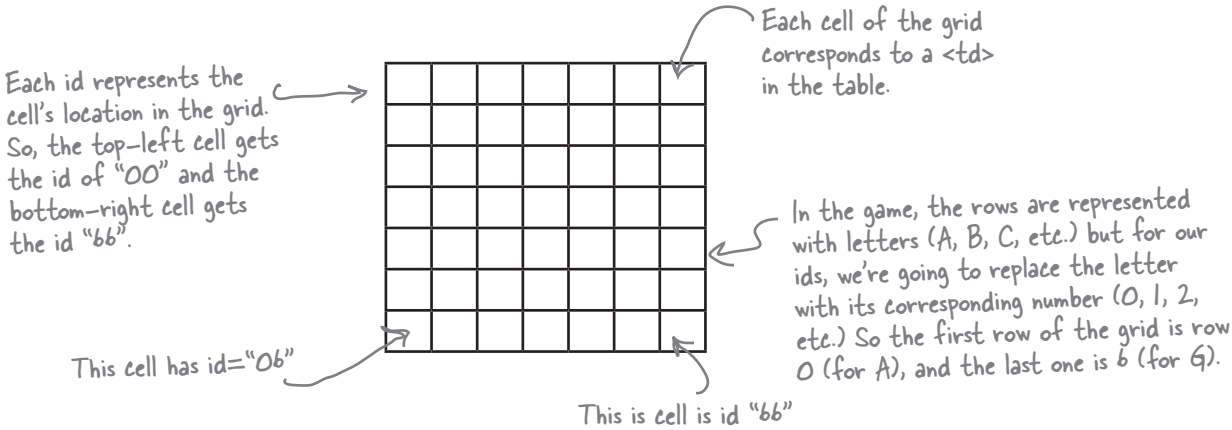
Go ahead and enter the code above (or download all the code for the book from http://wickedlysmart.com/hfjs) into the file "battleship.html" and then load it in your browser. Our test run is below.

Here's what the web page looks like so far...



Step 2: Creating the table

Next up is the table. The table will overlay the visual grid in the background image, and provide the area to place the hit and miss graphics where you play the game. Each cell (or if you remember your HTML, each <td> element) is going to sit right on top of a cell in the background image. Now here is the trick: we'll give each cell its own id, so we can manipulate it later with CSS and JavaScript. Let's check out how we're going to create these ids and add the HTML for the table:



Here's the HTML for the table. Go ahead and add this between the tags:

```
<div id="board">
```
We're nesting the table inside the "board" .

```
  <table>
    <tr>
      <td id="00"></td><td id="01"></td><td id="02"></td><td id="03">
</td><td id="04"></td> <td id="05"></td><td id="06"></td>
    </tr>
    <tr>
      <td id="10"></td><td id="11"></td><td id="12"></td><td id="13"></td>
<td id="14"></td> <td id="15"></td><td id="16"></td>
    </tr>
    ...
    <tr>
      <td id="60"></td><td id="61"></td><td id="62"></td><td id="63"></td>
<td id="64"></td><td id="65"></td><td id="66"></td>
    </tr>
  </table>
</div>
```

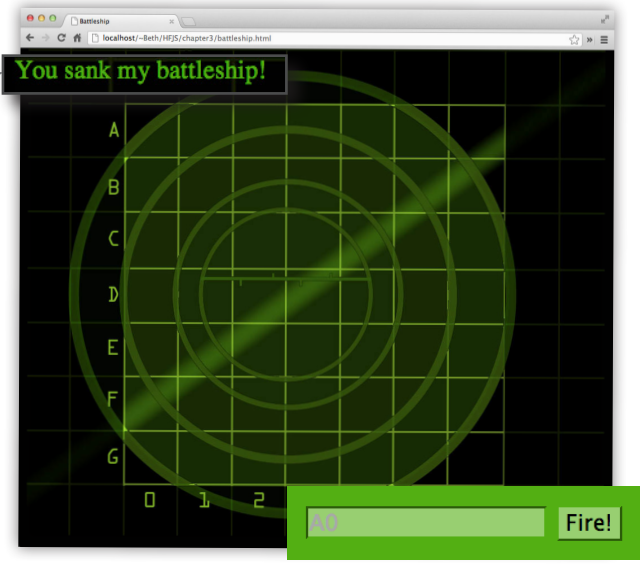
Make sure each <td> gets the correct id corresponding to its row and column in the grid.

We've left out a few rows to save some trees, but we're sure you can fill these in on your own.

Step 3: Player interaction

Okay, now we need an HTML element to enter guesses (like "A0" or "E4"), and an element to display messages to the player (like "You sank my battleship!"). We'll use a `<form>` with a text `<input>` for the player to submit guesses, and a `<div>` to create an area where we can message the player:

We'll notify players when they've sunk battleships with a message up in the top left corner.



And here's where players can enter their guesses.

```
<div id="board">
  <div id="messageArea"></div>
  <table>
    ...
  </table>
  <form>
    <input type="text" id="guessInput" placeholder="A0">
    <input type="button" id="fireButton" value="Fire!">
  </form>
</div>
```

The messageArea will be used to display messages from code.

The <form> has two inputs: one for the guess (a text input) and one for the button. Note the ids on these elements. We'll need them later when we write the code to get the player's guess.

Notice that the message area , the <table>, and the <form> are all nested within the "board" . This is important for the CSS on the next page.

Adding some more style

If you load the page now (go ahead, give it a try), most of the elements are going to be in the wrong places and the wrong size. So we need to provide some CSS to put everything in the right place, and make sure all the elements, like the table cells, have the right size to match up with the game board image.

To get the elements into the right places, we're going to use CSS positioning to lay everything out. We've positioned the "board" `<div>` element using position relative, so we can now position the message area, table, and form at specific places within the "board" `<div>` to get them to display exactly where we want them.

Let's start with the "messageArea" `<div>`. It's nested inside the "board" `<div>`, and we want to position it at the very top left corner of the game board:

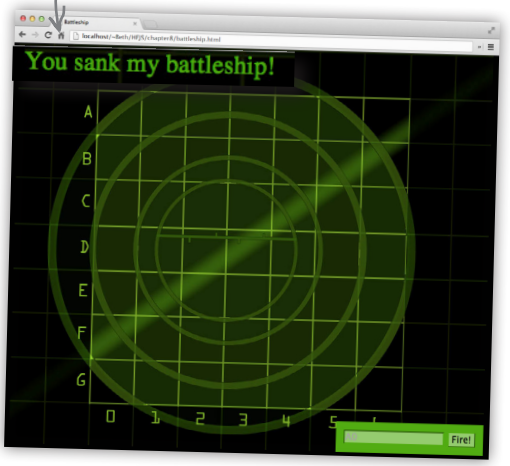
```
body {
    background-color: black;
}
div#board {
    position: relative;
    width: 1024px;
    height: 863px;
    margin: auto;
    background: url("board.jpg") no-repeat;
}
div#messageArea {
    position: absolute;
    top: 0px;
    left: 0px;
    color: rgb(83, 175, 19);
}
```

The "board" is positioned relative, so everything nested within this can be positioned relative to it.

We're positioning the message area at the top left of the board.

The messageArea is nested inside the board , so its position is specified relative to the board . So it will be positioned 0px from the top and 0px from the left of the top left corner of the board .

We want the message area to be positioned at the top left corner of the game board.



BULLET POINTS

- "position: relative" positions an element at its normal location in the flow of the page.
- "position: absolute" positions an element based on the position of its most closely positioned parent.
- The top and left properties can be used to specify the number of pixels to offset a positioned element from its default position.

We can also position the table and the form within the "board" , again using absolute positions to get these elements precisely where we want them. Here's the rest of the CSS:

```
body {
    background-color: black;
}
div#board {
    position: relative;
    width: 1024px;
    height: 863px;
    margin: auto;
    background: url("board.jpg") no-repeat;
}
div#messageArea {
    position: absolute;
    top: 0px;
    left: 0px;
    color: rgb(83, 175, 19);
}
table {
    position: absolute;
    left: 173px;
    top: 98px;
    border-spacing: 0px;
}
td {
    width: 94px;
    height: 94px;
}
form {
    position: absolute;
    bottom: 0px;
    right: 0px;
    padding: 15px;
    background-color: rgb(83, 175, 19);
}
form input {
    background-color: rgb(152, 207, 113);
    border-color: rgb(83, 175, 19);
    font-size: 1em;
}
```

We position the <table> 173 pixels from the left of the board and 98 pixels from the top, so it aligns with the grid in the background image.

Each <td> gets a specific width and height so that the cells of the <table> match up with the cells of the grid.

We're placing the <form> at the bottom right of the board. It obscures the bottom right numbers a bit, but that's okay (you know what they are). We're also giving the <form> a nice green color to match the background image.

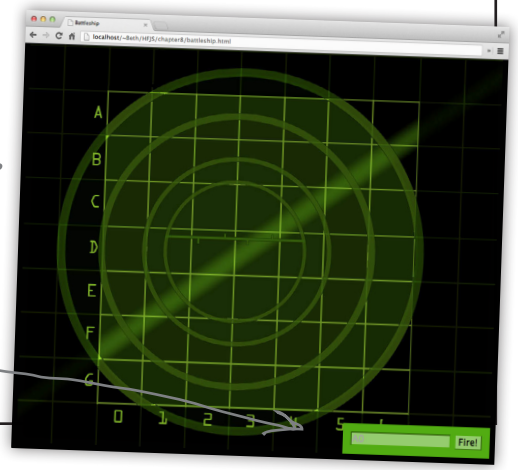
And finally, a bit of styling on the two <input> elements so they fit in with the game theme, and we're done!

A Test Drive

It's time for another game checkpoint. Get all the HTML and CSS entered into your HTML file and then reload the page in your browser. Here's what you should see:

Even though you can't see it (because it's invisible), the table is sitting right on top of the grid.

The form input is ready to take your guesses, although nothing will happen until we write some code.



Step 4: Placing the hits and misses

The game board is looking great don't you think? However, we still need to figure out how to visually add hits and misses to the board—that is, how to add either a "ship.png" image or a "miss.png" image to the appropriate spot on the board for each guess. Right now we're only going to worry about how to craft the right markup or style to do this, and then later we'll use the same technique in code.

ship.png

miss.png

So how do we get a "ship.png" image or a "miss.png" image on the board? A straightforward way is to add the appropriate image to the background of a `<td>` element using CSS. Let's try that by creating two classes, one named "hit" and the other "miss". We'll use the `background` CSS property with these images so an element styled with the "hit" class will have the "ship.png" in its background, and an element styled with the "miss" class will have the "miss.png" image in its background. Like this:

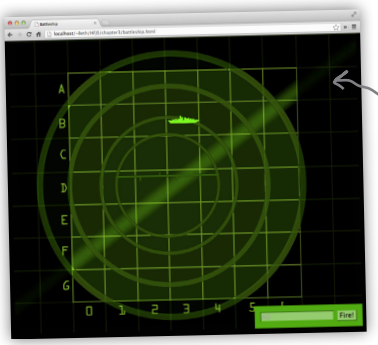
If an element is in the hit class it gets the ship.png image. If the element is in the miss class, it gets the miss.png image in its background.

```
.hit {
    background: url("ship.png") no-repeat center center;
}
.miss {
    background: url("miss.png") no-repeat center center;
}
```

Each CSS rule places a single, centered image in the selected element.

Using the hit and miss classes

Make sure you've added the hit and miss class definitions to your CSS. You may be wondering how we're going to use these classes. Let's do a little experiment right now to demonstrate: imagine you have a ship hidden at "B3", "B4" and "B5", and the user guesses "B3"—a hit! So, you need to place a "ship.png" image at B3. Here's how you can do that: first convert the "B" into a number, 1 (since A is 0, B is 1, and so on), and find the `<td>` with the id "13" in your table. Now, add the class "hit" to that `<td>`, like this:



Here we've added the "hit" class to the <td>.

```
<tr>
<td id="10"></td> <td id="11"></td> <td id="12"></td> <td id="13" class="hit"></td>
<td id="14"></td> <td id="15"></td> <td id="16"></td>
</tr>
```

Make sure you've added the hit and miss classes from the previous page to your CSS.

What we see when we add the class "hit" to element with id "13".

Now when you reload the page, you'll see a battleship at location "B3" in the game board.

PRACTICE DRILLS

Before we write the code that's going to place hits and misses on the game board, get a little more practice to see how the CSS works. Manually play the game by adding the "hit" and "miss" classes into your markup, as dictated by the player's moves below. Be sure to check your answers!

```
Ship 1: A6, B6, C6
Ship 2: C4, D4, E4
Ship 3: B0, B1, B2
```

Remember, you'll need to convert the letters to numbers, with A = 0, ... G = 6.

and here are the player's guesses:

```
A0, D4, F5, B2, C5, C6
```

When you're done, remove any classes that you've added to your <td> elements so you'll have an empty board to use when we start coding.

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

there are no Dumb Questions

Q: I didn't know it was okay to use a string of numbers for the id attributes in our table?

A: Yes. As of HTML5, you are allowed to use all numbers as an element id. As long as there are no spaces in the id value, it's fine. And for the Battleship application, using numbers for each id works perfectly as a way to keep track of each table position, so we can access the element at that position quickly and easily.

Q: So just to make sure I understand, we're using each td element as a cell in the gameboard, and we'll mark a cell as being a hit or a miss with the class attribute?

A: Right, there are a few pieces here: we have a background image grid that is just for eye candy, we have a transparent HTML table overlaying that, and we use the classes "hit" and "miss" to put an image in the background of each table cell when needed. This last part will all be done from code, when we're going to dynamically add the class to an element.

Q: It sounds like we're going to need to convert letters, as in "A6", to numbers so we get "06". Will JavaScript do this automatically for us?

A: No, we're going to have to do that ourselves, but we have an easy way to do it—we're going to use what you know about arrays to do a quick conversion… stay tuned.

Q: I'm not sure I completely remember how CSS positioning works.

A: Positioning allows you to specify an exact position for an element. If an element is positioned "relative", then the element is positioned based on its normal location in the flow of the page. If an element is positioned "absolute", then that element is positioned at a specific location, relative to its most closely positioned parent. Sometimes that's the entire page, in which case the position you specify could be its top left position based on the corner of the web browser. In our case, we're positioning the table and message area elements absolutely, but in relation to the game board (because the board is the most closely positioned parent of the table and the message area).

If you need a more in-depth refresher on CSS positioning, check out Chapter 11 of *Head First HTML and CSS*.

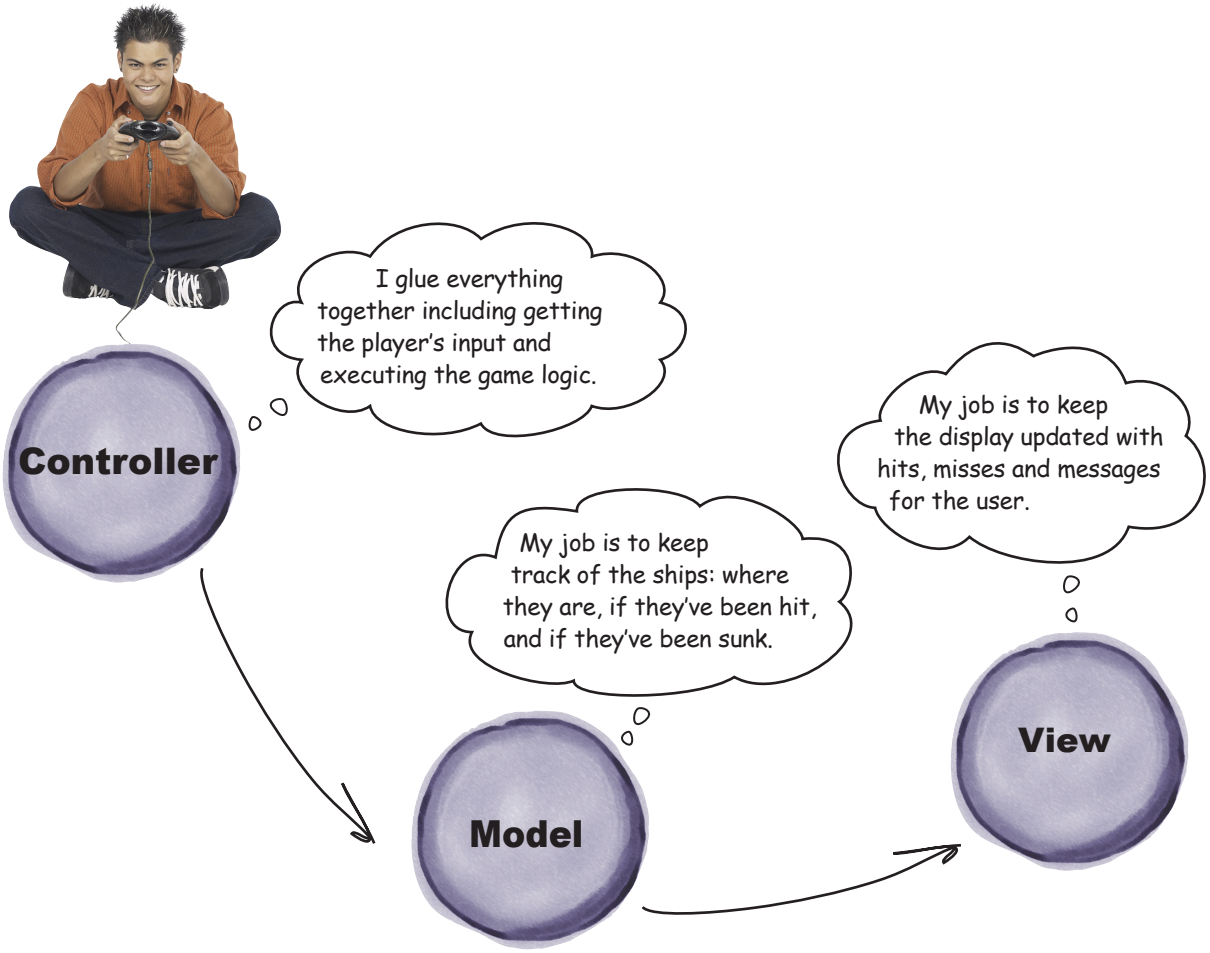
Q: When I learned about the HTML form element, I was taught there is an action attribute that submits the form. Why don't we have one?

A: We don't need the action attribute in the <form> because we're not submitting the form to a server-side application. For this game, we're going to be handling everything in the browser, using code. So, instead of submitting the form, we're going to implement an event handler to be notified when the form button is clicked, and when that happens, we'll handle everything in our code, including getting the user's input from the form. Notice that the type of the form button is "button", not "submit", like you might be used to seeing if you've implemented forms that submit data to a PHP program or another kind of program that runs on the server. It's a good question; more on this later in the chapter.

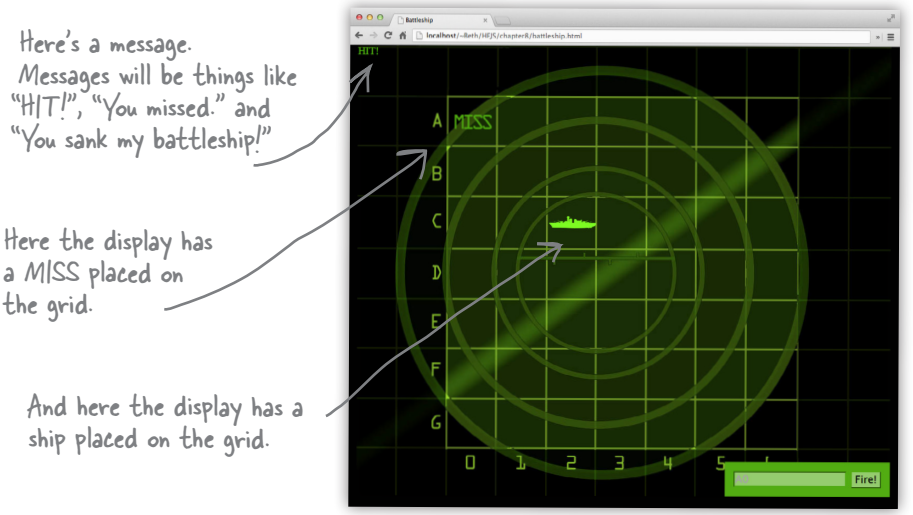
How to design the game

With the HTML and CSS out of the way, let's get to the real game design. Back in Chapter 2, we hadn't covered functions or objects or encapsulation or learned about object-oriented design, so when we built the first version of the Battleship game, we used a procedural design—that is, we designed the game as a series of steps, with some decision logic and iteration mixed in. You also hadn't learned about the DOM, so the game wasn't very interactive. This time around, we're going to organize the game into a set of objects, each with its own responsibilities, and we're going to use the DOM to interact with the user. You'll see how this design makes approaching the problem a lot more straightforward.

Let's first get introduced to the objects we're going to design and implement. There are three: the *model*, which will hold the state of the game, like where each ship is located and where it's been hit; the *view*, which is responsible for updating the display; and the *controller*, which glues everything together by handling the user input, making sure the game logic gets played and determining when the game is over.



It's time for some object design. We're going to start with the view object. Now, remember, the view object is responsible for updating the view. Take a look at the view below and see if you can determine the methods we want the view object to implement. Write the declarations for these methods below (just the declarations; we'll code the bodies of the methods in a bit) along with a comment or two about what each does. We've done one for you. *Check your answers before moving on:*




```
var view = {
```

Notice we're defining an object and assigning it to the variable view.

```
   // this method takes a string message and displays it
   // in the message display area
   displayMessage: function(msg) {
          // code to be supplied in a bit!
   }
```

Your methods go here!

```
};
```

Implementing the View

If not, shame on you. Do it now!

If you checked the answer to the previous exercise, you've seen that we've broken the view into three separate methods: `displayMessage`, `displayHit` and `displayMiss`. Now, there is no one right answer. For instance, you might have just two methods, `displayMessage` and `displayPlayerGuess`, and pass an argument into `displayPlayerGuess` that indicates if the player's guess was a hit or a miss. That is a perfectly reasonable design. But we're sticking with our design for now... so let's think through how to implement the first method, `displayMessage`:

Here's our view object.

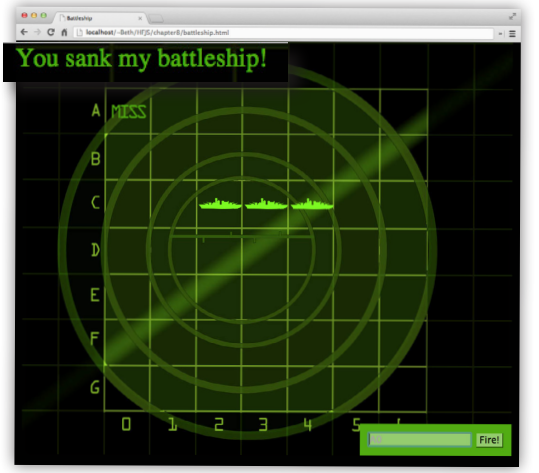
```
var view = {
   displayMessage: function(msg) {

   },
   displayHit: function(location) {
   },
   displayMiss: function(location) {
   }
};
```

We're going to start here.

How displayMessage works

To implement the `displayMessage` method you need to review the HTML and see that we have a `<div>` with the id "messageArea" ready for messages:

```
<div id="board">
  <div id="messageArea"></div>
    ...
</div>
```



We'll use the DOM to get access to this `<div>`, and then set its text using `innerHTML`. And remember, whenever you change the DOM, you'll see the changes immediately in the browser. Here's what we're going to do…

That's one great thing about objects. We can make sure objects fulfill their responsibility without worrying about every other detail of the program. In this case the view just needs to know how to update the message area and place hit and miss markers on the grid. Once we've correctly implemented that behavior, we're done with the view object and we can move on to other parts of the code.

The other advantage of this approach is we can test the view in isolation and make sure it works. When we test many aspects of the program at once, we increase the odds something is going to go wrong and at the same time make the job of finding the problem more difficult (because you have to examine more areas of the code to find the problem).

To test an isolated object (without having finished the rest of the program yet), we'll need to write a little testing code that we'll throw away later, but that's okay.

So let's finish the view, test it, and then move on!

Implementing displayMessage

Let's get back to writing the code for `displayMessage`. Remember it needs to:

- Use the DOM to get the element with the id "messageArea".
- Set that element's `innerHTML` to the message passed to the `displayMessage` method.

So open up your blank "battleship.js" file, and add the view object:

```
var view = {
    displayMessage: function(msg) {
        var messageArea = document.getElementById("messageArea");
        messageArea.innerHTML = msg;
    },
    displayHit: function(location) {
    },
    displayMiss: function(location) {
    }
};
```

The displayMessage method takes one argument, a msg.

We get the messageArea element from the page...

...and update the text of the messageArea element by setting its innerHTML to msg.

Now before we test this code, let's go ahead and write the other two methods. They won't be incredibly complicated methods, and this way we can test the entire object at once.

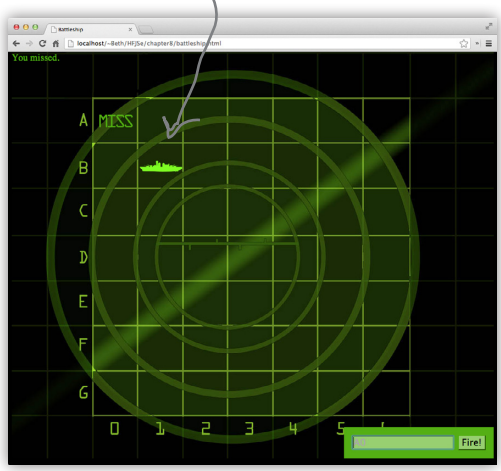
How displayHit and displayMiss work

So we just talked about this, but remember, to have an image appear on the game board, we need to take a `<td>` element and add either the "hit" or the "miss" class to the element. The former results in a "ship.png" appearing in the cell and the latter results in "miss.png" being displayed.

We can affect the display by adding the "hit" or "miss" class to the <td> elements. Now we just need to do this from code.

```
<tr>
<td id="10"></td> <td class="hit" id="11"></td> <td id="12"></td> ...
</tr>
```

Now in code, we're going to use the DOM to get access to a `<td>`, and then set its class attribute to "hit" or "miss" using the `setAttribute` element method. As soon as we set the class attribute, you'll see the appropriate image appear in the browser. Here's what we're going to do:



- Get a string id that consists of two numbers for the location of the hit or miss.
- Use the DOM to get the element with that id.
- Set that element's class attribute to "hit" if we're in `displayHit`, and "miss" if we're in `displayMiss`.

Implementing displayHit and displayMiss

Both `displayHit` and `displayMiss` are methods that take the location of a hit or miss as an argument. That location should match the id of a cell (or `<td>` element) in the table representing the game board in the HTML. So the first thing we need to do is get a reference to that element with the `getElementById` method. Let's try this in the `displayHit` method:

```
displayHit: function(location) {
    var cell = document.getElementById(location);
},
```

Remember the location is created from the row and column and matches an id of a <td> element.

The next step is to add the class "hit" to the cell, which we can do with the `setAttribute` method like this:

```
displayHit: function(location) {
    var cell = document.getElementById(location);
    cell.setAttribute("class", "hit");
},
```

We then set the class of that element to "hit". This will immediately add a ship image to the <td> element.

Now let's add this code to the view object, and write `displayMiss` as well:

```
var view = {
    displayMessage: function(msg) {
        var messageArea = document.getElementById("messageArea");
        messageArea.innerHTML = msg;
    },

    displayHit: function(location) {
        var cell = document.getElementById(location);
        cell.setAttribute("class", "hit");
    },

    displayMiss: function(location) {
        var cell = document.getElementById(location);
        cell.setAttribute("class", "miss");
    }
};
```

We're using the id we created from the player's guess to get the correct element to update.

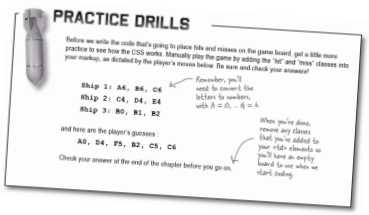
And then setting the class of that element to "hit".

We do the same thing in displayMiss, only we set the class to "miss" which adds a miss image to the element.

Make sure you add the code for `displayHit` and `displayMiss` to your "battleship.js" file.

Another Test Drive...

Let's put the code through its paces before moving on…in fact, let's take the guesses from the previous Practice Drills exercise and implement them in code. Here's the sequence we want to implement:



To represent that sequence in code, add this to the bottom of your "battleship.js" JavaScript file:

```
view.displayMiss("00");
view.displayHit("34");
view.displayMiss("55");
view.displayHit("12");
view.displayMiss("25");
view.displayHit("26");
```

"A0"
"D4"
"F5"
"B2"
"C5"
"C6"

Remember, displayHit and displayMiss take a location in the board that's already been converted from a letter and a number to a string with two numbers that corresponds to an id of one of the table cells.

And, let's not forget to test `displayMessage`:

```
view.displayMessage("Tap tap, is this thing on?");
```

Any message will do for simple testing...

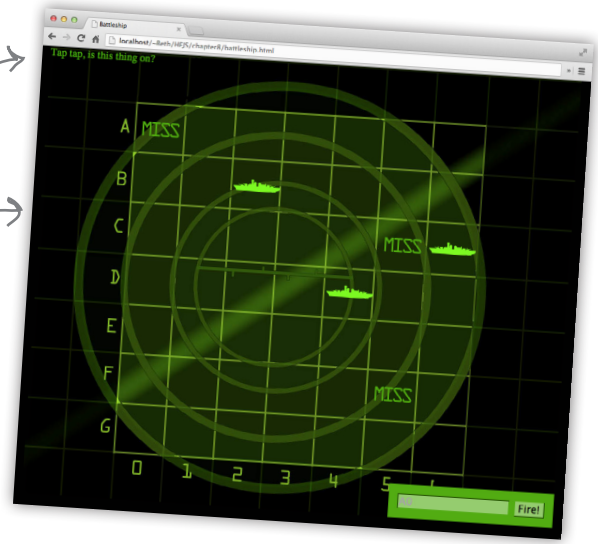
After all that, reload the page in your browser and check out the updates to the display.

One of the benefits of breaking up the code into objects and giving each object only one responsibility is that we can test each object to make sure it's doing its job correctly.

The "tap tap" message is displayed up here at the top left of the view.

And the hits and misses we displayed using the view object are displayed in the game board.

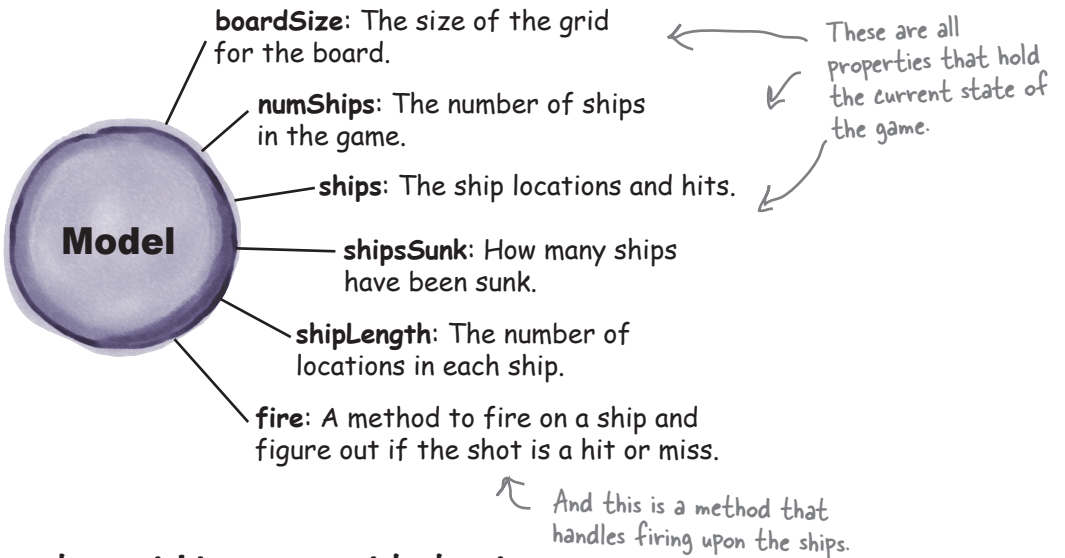
Check each one to make sure it's in the right spot.



The Model

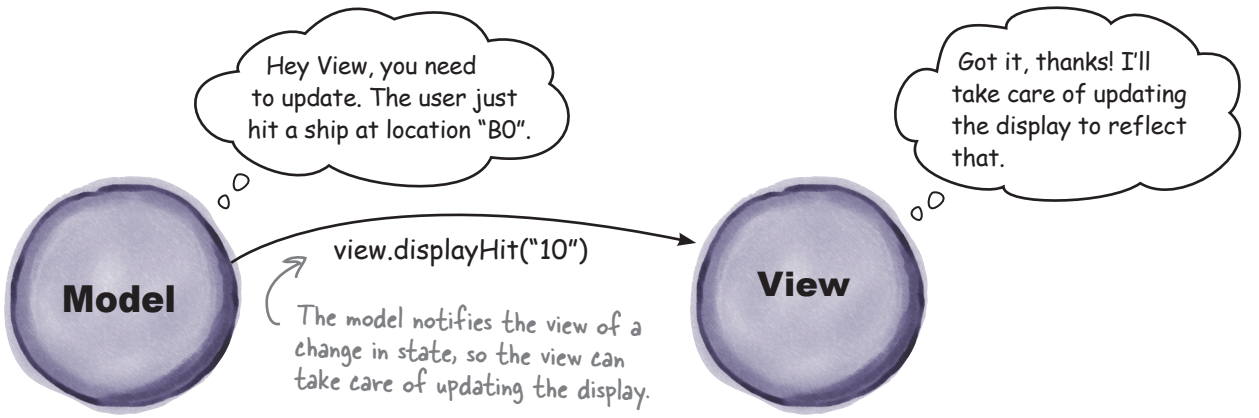
With the view object out of the way, let's move on to the model. The model is where we keep the *state* of the game. The model often also holds some *logic* relating to how the state changes. In this case the state includes the location of the ships, the ship locations that have been hit, and how many ships have been sunk. The only logic we're going to need (for now) is determining when a player's guess has hit a ship and then marking that ship with a hit.

Here's what the model object is going to look like:



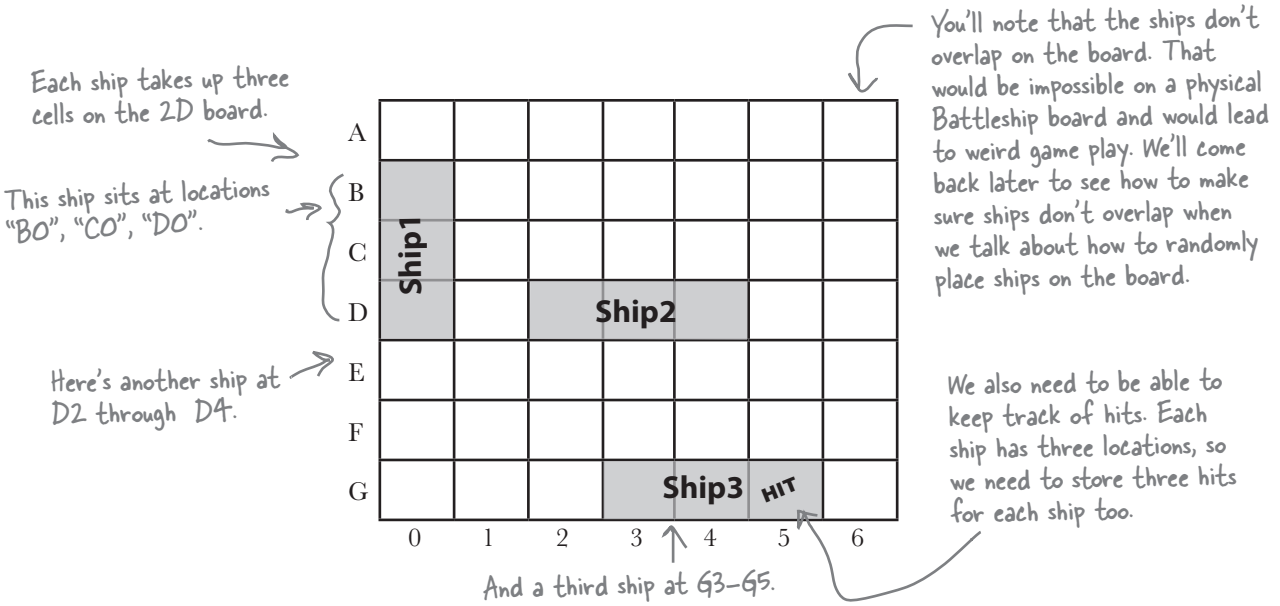
How the model interacts with the view

When the state of the game changes—that is, when you hit a ship, or miss—then the view needs to update the display. To do this, the model needs to talk to the view, and luckily we have a few methods the model can use to do that. We'll get our game logic set first in the model, then we'll add code to update the view.



You're gonna need a bigger boat... and game board

Before we start writing model code, we need to think about how to represent the state of the ships in the model. Back in Chapter 2 in the simple Battleship game, we had a single ship that sat on a 1x7 game board. Now things are a little more complex: we have *three* ships on a 7x7 board. Here's how it looks now:



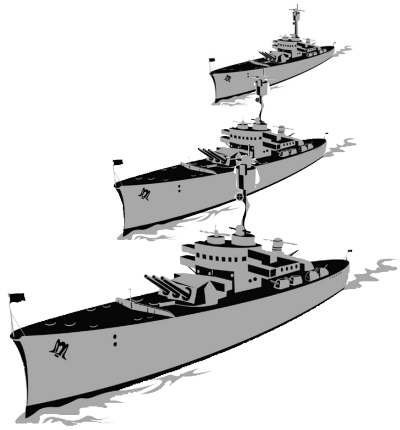
Sharpen your pencil

Given how we've described the new game board above, how would you represent the ships in the model (just the locations, we'll worry about hits later). Check off the best solution below.

- ❑ Use nine variables for the ship locations, similar to the way we handled the ships in Chapter 2.
- ❑ Use an array with an item for each cell in entire board (49 items total). Record the ship number in each cell that holds part of a ship.
- ❑ Use an array to hold all nine locations. Items 0-2 will hold the first ship, 3-5 the second, and so on.
- ❑ Use three different arrays, one for each ship, with three locations contained in each.
- ❑ Use an object named ship with three location properties. Put all the ships in an array named ships.
- ❑ _____

Or write in your own answer.

How we're going to represent the ships



As you can see there are many ways we can represent ships, and you may have even come up with a few other ways of your own. You'll find that no matter what kind of data you've got, there are many choices for storing that data, with various tradeoffs depending on your choice—some methods will be space efficient, others will optimize run time, some will just be easier to understand, and so on.

We've chosen a representation for ships that is fairly simple—we're representing each ship as an object that holds the locations it sits in, along with the hits it's taken. Let's take a look at how we represent one ship:

Each ship is an object.

The ship has a locations property and a hits property.

```
var ship1 = {
              locations: ["10", "20", "30"],
              hits: ["", "", ""]
            };
```

The locations property is an array that holds each location on the board.

The hits property is also an array that holds whether or not a ship is hit at each location. We'll set the array items to the empty string initially, and change each item to "hit" when the ship has taken a hit in the corresponding location.

Note that we've converted the ship locations to two numbers, using 0 for A, 1 for B, and so on.

Here's what all three ships would look like:

Each ship has an array of three locations and an array to track hits.

```
var ship1 = { locations: ["10", "20", "30"], hits: ["", "", ""] };
var ship2 = { locations: ["32", "33", "34"], hits: ["", "", ""] };
var ship3 = { locations: ["63", "64", "65"], hits: ["", "", "hit"] };
```

And, rather than managing three different variables to hold the ships, we'll create a single array variable to hold them all, like this:

We're assigning to ships an array that holds all three ships.

Note the plural name, ships.

```
var ships = [{ locations: ["10", "20", "30"], hits: ["", "", ""] },
             { locations: ["32", "33", "34"], hits: ["", "", ""] },
             { locations: ["63", "64", "65"], hits: ["", "", "hit"] }];
```

Here's the first ship...

...and the second...

...and the third.

Note this ship has a hit at location "65" on the grid.

Ship Magnets

Use the following player moves, along with the data structure for the ships, to place the ship and miss magnets onto the game board. Does the player sink all the ships? We've done the first move for you.

Here are the moves:

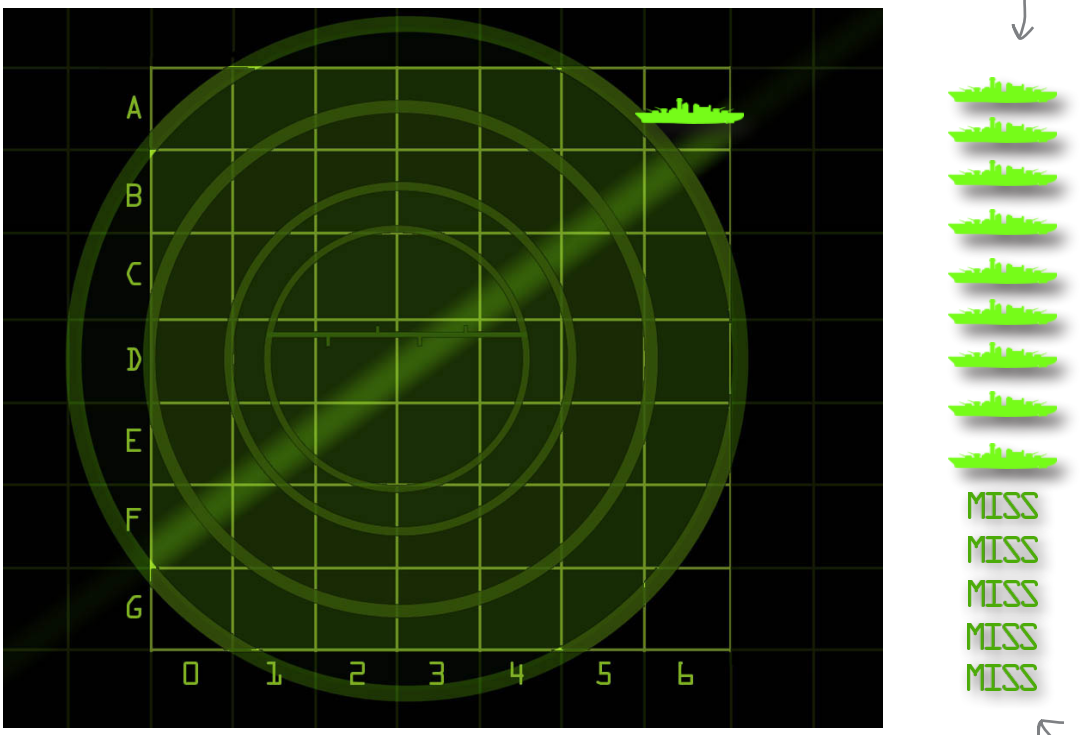
A6, B3, C4, D1, B0, D4, F0, A1, C6, B1, B2, E4, B6

Execute these moves on the game board.

```
var ships = [{ locations: ["06", "16", "26"], hits: ["hit", "", ""] },
             { locations: ["24", "34", "44"], hits: ["", "", ""] },
             { locations: ["10", "11", "12"], hits: ["", "", ""] }];
```

Here is the data structure. Mark each ship with a hit as the game is played.

And here's the board and your magnets.



You might have leftover magnets.

Sharpen your pencil

Let's practice using the ships data structure to simulate some ship activities. Using the ships definition below, work through the questions and the code below and fill in the blanks. Make sure you check your answers before moving on, as this is an important part of how the game works:

```
var ships = [{ locations: ["31", "41", "51"], hits: ["", "", ""] },
             { locations: ["14", "24", "34"], hits: ["", "hit", ""] },
             { locations: ["00", "01", "02"], hits: ["hit", "", ""] }];
```

Which ships are already hit?_____ And, at what locations? _____

The player guesses "D4", does that hit a ship?_____ If so, which one? _____

The player guesses "B3", does that hit a ship?_____ If so, which one? _____

Finish this code to access the second ship's middle location and print its value with console.log.

```
var ship2 = ships[____];
var locations = ship2.locations;
console.log("Location is " + locations[____]);
```

Finish this code to see if the third ship has a hit in its first location:

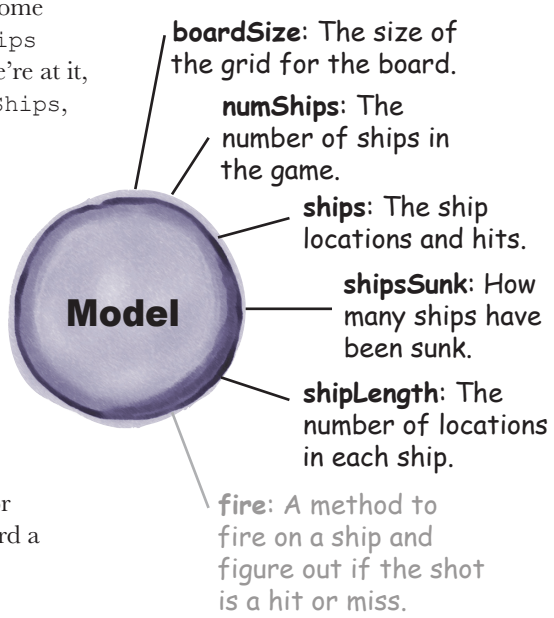
```
var ship3 = ships[____];
var hits = ship3._____;
if (_____ === "hit") {
   console.log("Ouch, hit on third ship at location one");
}
```

Finish this code to hit the first ship at the third location:

```
var ______ = ships[0];
var hits = ship1._______;
hits[____] = ________;
```

Implementing the model object

Now that you know how to represent the ships and the hits, let's get some code down. First, we'll create the model object, and then take the `ships` data structure we just created, and add it as a property. And, while we're at it, there are a few other properties we're going to need as well, like `numShips`, to hold the number of ships we have in the game. Now, if you're asking, "What do you mean, we know there are three ships, why do we need a `numShips` property?" Well, what if you wanted to create a new version of the game that was more difficult and had four or five ships? By not "hardcoding" this value, and using a property instead (and then using the property throughout the code rather than the number), we can save ourselves a future headache if we need to change the number of ships, because we'll only need to change it in one place.



Now, speaking of "hardcoding", we *are* going to hardcode the ships' initial locations, for now. By knowing where the ships are, we can test the game more easily, and focus on the core game logic for now. We'll tackle the code for placing random ships on the game board a little later.

So let's get the model object created:

The model is an object.

```
var model = {
    boardSize: 7,
    numShips: 3,
    shipLength: 3,
    shipsSunk: 0,

    ships: [{ locations: ["06", "16", "26"], hits: ["", "", ""] },
            { locations: ["24", "34", "44"], hits: ["", "", ""] },
            { locations: ["10", "11", "12"], hits: ["", "", ""] }]

};
```

These three properties keep us from hardcoding values. They are: boardSize (the size of the grid used for the board), numShips (the number of ships in the game), and shipLength (the number of locations in each ship, 3).

shipsSunk (initialized to 0 for the start of the game) keeps the current number of ships that have been sunk by the player.

We've got quite a bit of state already!

Later on, we'll generate these locations for the ships so they're random, but for now, we'll hardcode them to make it easier to test the game.

The property ships is the array of ship objects that each store the locations and hits of one of the three ships. (Notice that we've changed ships from a variable, which we used before, to a property for the model object.)

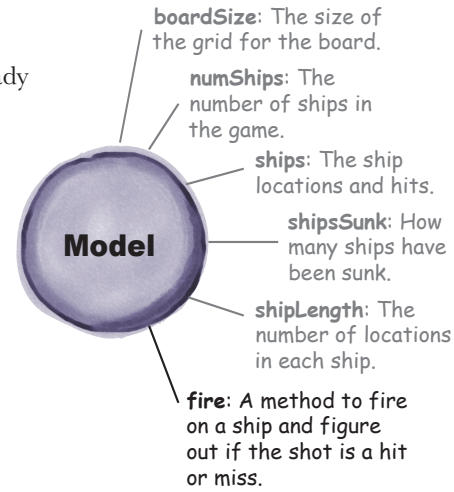
Note we're also hardcoding the sizes of the locations and hits arrays. You'll learn how to dynamically create arrays later in the book.

Thinking about the fire method

The `fire` method is what turns a player's guess into a hit or a miss. We already know the `view` object is going to take care of displaying the hits and misses, but the `fire` method has to provide the game logic for determining if a hit or a miss has occurred.

Knowing that a ship is hit is straightforward: given a player's guess, you just need to:

- Examine each ship and see if it occupies that location.
- If it does, you have a hit, and we'll mark the corresponding item in the `hits` array (and let the view know we got a hit). We'll also return `true` from the method, meaning we got a hit.
- If no ship occupies the guessed location, you've got a miss. We'll let the view know, and return `false` from the method.



Now the `fire` method should also determine if a ship isn't just hit, but if it's sunk. We'll worry about that once we have the rest of the logic worked out.

Setting up the fire method

Let's get a basic skeleton of the `fire` method set up. The method will take a guess as an argument, and then iterate over each ship to determine if that ship was hit. We won't write the hit detection code just yet, but let's get the rest set up now:

```
var model = {
    boardSize: 7,
    numShips: 3,
    shipsSunk: 0,
    shipLength: 3,
    ships: [{ locations: ["06", "16", "26"], hits: ["", "", ""] },
            { locations: ["24", "34", "44"], hits: ["", "", ""] },
            { locations: ["10", "11", "12"], hits: ["", "", ""] }],

    fire: function(guess) {

        for (var i = 0; i < this.numShips; i++) {
            var ship = this.ships[i];
        }
    }
};
```

Don't forget to add a comma here!

The method accepts a guess.

Then, we iterate through the array of ships, examining one ship at a time.

Here we have our hands on a ship. We need to see if the guess matches any of its locations.

Looking for hits

So now, each time through the loop, we need to see if the guess is one of the locations of the ship:

```
for (var i = 0; i < this.numShips; i++) {
    var ship = this.ships[i];
    locations = ship.locations;

}
```

And we're stepping through each ship.

And we've accessed the ship's set of locations. Remember this is a property of the ship that contains an array.

What we need is the code that determines if the guess is in this ship's locations.

Here's the situation: we have a string, `guess`, that we're looking for in an array, `locations`. If `guess` matches one of those locations, we know we have a hit:

```
guess = "16";
locations = ["06", "16", "26"];
```

We need to find out if the value in guess is one of the values in the ship's locations array.

We could write yet another loop to go through each item in the `locations` array, compare the item to `guess`, and if they match, we have a hit.

But rather than write another loop, we have an easier way to do this:

```
var index = locations.indexOf(guess);
```

The indexOf method searches an array for a matching value and returns its index, or –1 if it can't find it.

So, using `indexOf`, we can write the code to find a hit like this:

```
for (var i = 0; i < this.numShips; i++) {
    var ship = this.ships[i];
    locations = ship.locations;
    var index = locations.indexOf(guess);
    if (index >= 0) {
        // We have a hit!
    }
}
```

Notice that the indexOf method for an array is similar to the indexOf string method. It takes a value and returns the index of that value in the array (or –1 if it can't find the value).

So if we get an index greater than or equal to zero, the user's guess is in the location's array, and we have a hit.

Using `indexOf` isn't any more efficient than writing a loop, but it is a little clearer and it's definitely less code. We'd also argue that the intent of this code is clearer than if we wrote a loop: it's easier to see what value we're looking for in an array using `indexOf`. In any case, you now have another tool in your programming toolbelt.

Putting that all together...

To finish this up, we have one more thing to determine here: if we have a hit, what do we do? All we need to do, for now, is mark the hit in the model, which means adding a "hit" string to the `hits` array. Let's put all the pieces together:

```
var model = {
    boardSize: 7,
    numShips: 3,
    shipsSunk: 0,
    shipLength: 3,
    ships: [ { locations: ["06", "16", "26"], hits: ["", "", ""] },
             { locations: ["24", "34", "44"], hits: ["", "", ""] },
             { locations: ["10", "11", "12"], hits: ["", "", ""] } ],

     fire: function(guess) {
         for (var i = 0; i < this.numShips; i++) {
             var ship = this.ships[i];
             var locations = ship.locations;
             var index = locations.indexOf(guess);
             if (index >= 0) {
                 ship.hits[index] = "hit";
                 return true;
             }
         }
         return false;
     }
};
```

For each ship...

If the guess is in the locations array, we have a hit.

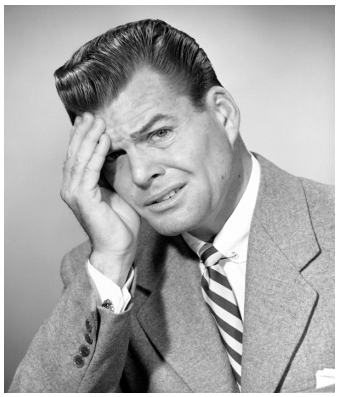
So mark the hits array at the same index.

Oh, and we need to return true because we had a hit.

Otherwise, if we make it through all the ships and don't have a hit, it's a miss, so we return false.

That's a great start on our model object. There are only a couple of other things we need to do: determine if a ship is sunk, and let the view know about the changes in the model so it can keep the player updated. Let's get started on those…

Wait, can we talk about your verbosity again?



Sorry, we have to bring this up again. You're being a bit verbose in some of your references to objects and arrays. Take another look at the code:

```
for (var i = 0; i < this.numShips; i++) {
  var ship = this.ships[i];
  var locations = ship.locations;
  var index = locations.indexOf(guess);
  ...
}
```

First we get the ship...
Then we get the locations in the ship...
Then we get the index of the guess in the locations.

Some would call this code overly verbose. Why? Because some of these references can be shortened using *chaining*. Chaining allows us to string together object references so that we don't have to create temporary variables, like the `locations` variable in the code above.

Now you might ask why `locations` is a temporary variable? That's because we're using `locations` only to temporarily store the `ship.locations` array so we can then turn around and call the `indexOf` method on it to get the `index` of the `guess`. We don't need `locations` for anything else in this method. With chaining, we can get rid of that temporary `locations` variable, like this:

```
var index = ship.locations.indexOf(guess);
```

We've combined the two lines highlighted above into a single line.

How chaining works...

Chaining is really just a shorthand for a longer series of steps to access properties and methods of objects (and arrays). Let's take a closer look at what we just did to combine two statements with chaining.

```
var ship = { locations: ["06", "16", "26"], hits: ["", "", ""] };
var locations = ship.locations;
var index = locations.indexOf(guess);
```

Here's a ship object.
We were grabbing the locations array from the ship
And then using it to access the indexOf method.

We can combine the bottom two statements by chaining together the expressions (and getting rid of the variable `locations`):

```
ship.locations.indexOf(guess)
```

1. Evaluates to the ship object.
2. Which has a locations property, which is an array.
3. Which has a method named indexOf.

Meanwhile back at the battleship...

Now we need to write the code to determine if a ship is sunk. You know the rules: a battleship is sunk when all of its locations are hit. We can add a little helper method to check to see if a ship is sunk:

We'll call the method isSunk. It's going to take a ship and return true if it's sunk and false if it is still floating.

This method takes a ship, and then checks every possible location for a hit.

```
isSunk: function(ship) {
    for (var i = 0; i < this.shipLength; i++)  {
        if (ship.hits[i] !== "hit") {
            return false;
        }
    }
    return true;
}
```

If there's a location that doesn't have a hit, then the ship is still floating, so return false.

Otherwise this ship is sunk! Return true.

Go ahead and add this method to your model object, just below fire.

Now, we can use that method in the `fire` method to find out if a ship is sunk:

```
fire: function(guess) {
    for (var i = 0; i < this.numShips; i++) {
        var ship = this.ships[i];
        var index = ship.locations.indexOf(guess);
        if (index >= 0) {
            ship.hits[index] = "hit";
            if (this.isSunk(ship)) {
                this.shipsSunk++;
            }
            return true;
        }
    }
    return false;
},
isSunk: function(ship) { ... }
```

We'll add the check here, after we know for sure we have a hit. If the ship is sunk, then we increase the number of ships that are sunk in model's shipsSunk property.

Here's where we added the new isSunk method, just below fire. Don't forget to make sure you've got a comma between each of the model's properties and methods!

A view to a kill...

That's about it for the model object. The model maintains the state of the game, and has the logic to test guesses for hits and misses. The only thing we're missing is the code to notify the view when we get a hit or a miss in the model. Let's do that now:

This is the whole model object so you can see the entire thing in one piece.

```
var model = {
    boardSize: 7,
    numShips: 3,
    shipsSunk: 0,
    shipLength: 3,
    ships: [ { locations: ["06", "16", "26"], hits: ["", "", ""] },
             { locations: ["24", "34", "44"], hits: ["", "", ""] },
             { locations: ["10", "11", "12"], hits: ["", "", ""] } ],
    fire: function(guess) {
        for (var i = 0; i < this.numShips; i++) {
            var ship = this.ships[i];
            var index = ship.locations.indexOf(guess);
            if (index >= 0) {
                ship.hits[index] = "hit";
                view.displayHit(guess);
                view.displayMessage("HIT!");
                if (this.isSunk(ship)) {
                    view.displayMessage("You sank my battleship!");
                    this.shipsSunk++;
                }
                return true;
            }
        }
        view.displayMiss(guess);
        view.displayMessage("You missed.");
        return false;
    },
    isSunk: function(ship) {
        for (var i = 0; i < this.shipLength; i++)  {
            if (ship.hits[i] !== "hit") {
                return false;
            }
        }
        return true;
    }
};
```

Notify the view that we got a hit at the location in guess.

And ask the view to display the message "HIT!".

Let the player know that this hit sank the battleship!

Notify the view that we got a miss at the location in guess.

And ask the view to display the message "You missed.".

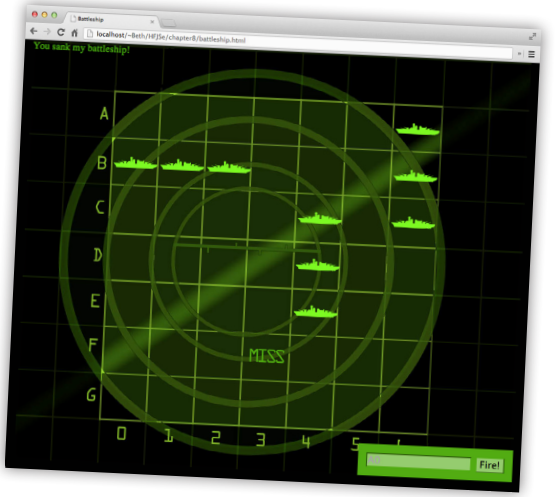
Remember that the methods in the view object add the "hit" or "miss" class to the element with the id at row and column in the guess string. So the view translates the "hit" in the hits array into a "hit" in the HTML. But keep in mind, the "hit" in the HTML is just for display; the "hit" in the model represents the actual state.

A Test Drive

Add all the model code to "battleship.js". Test it by calling the model's fire method, passing in a row and column of a guess each time. Our ships are hardcoded still, so it'll be easy for you to hit them all. Try adding some of your own as well (a few more misses). (Download "battleship_tester.js" to see our version of the test code.)

You'll need to remove or comment out the previous view testing code to get the same results as we show here. You can see how to do that in battleship_tester.js.

```
model.fire("53");

model.fire("06");
model.fire("16");
model.fire("26");

model.fire("34");
model.fire("24");
model.fire("44");

model.fire("12");
model.fire("11");
model.fire("10");
```



Reload "battleship.html". You should see your hits and misses appear on the game board.

there are no Dumb Questions

Q: Is using chaining to combine statements better than keeping statements separate?

A: Not necessarily better, no. Chaining isn't much more efficient (you save one variable), but it does make your code shorter. We'd argue that short chains (2 or 3 levels at most) are easier to read than multiple lines of code, but that's our preference. If you want to keep your statements separate, that's fine. And if you do use chaining, make sure you don't create really long chains; they will be harder to read and understand if they're too long.

Q: We have arrays (locations) inside an object (ship) inside an array (ships). How many levels deep can you nest objects and arrays like this?

A: Pretty much as deep as you want. Practically, of course, it's unlikely you'll ever go too deep (and if you find yourself with more than three or four levels of nesting, it's likely your data structure is getting too complex and you should rethink things a bit).

Q: I noticed we added a property named boardSize to the model, but we haven't used it in the model code. What is that for?

A: We're going to be using model.boardSize, and the other properties in model, in the code coming up. The model's responsibility is to manage the state of the game, and boardSize is definitely part of the state. The controller will access the state it needs by accessing the model's properties, and we'll be adding more model methods later that will use these properties too.

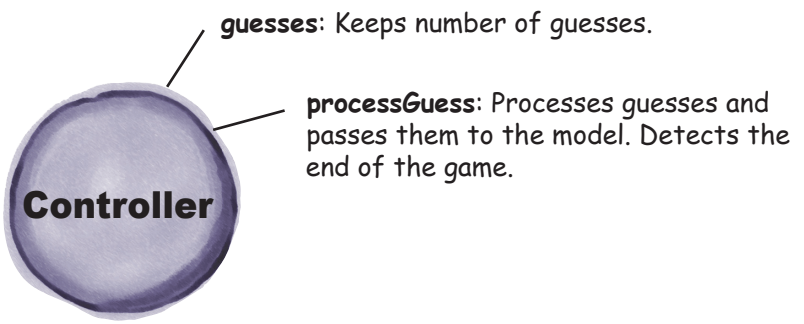
Implementing the Controller

Now that you have the view and the model complete, we're going to start to bring this app together by implementing the controller. At a high level, the controller glues everything together by getting a guess, processing the guess and getting it to the model. It also keeps track of some administrative details, like the current number of guesses and the player's progress in the game. To do all this the controller relies on the model to keep the state of the game and on the view to display the game.

More specifically, here's the set of responsibilities we're giving the controller:

- Get and process the player's guess (like "A0" or "B1").
- Keep track of the number of guesses.
- Ask the model to update itself based on the latest guess.
- Determine when the game is over (that is, when all ships have been sunk).

Let's get started on the controller by first defining a property, `guesses`, in the controller object. Then we'll implement a single method, `processGuess`, that takes an alpha-numeric guess, processes it and passes it to the model.



Here's the skeleton of the controller code; we'll fill this in over the next few pages:

```
var controller = {
   guesses: 0,

   processGuess: function(guess) {
     // more code will go here
   }
};
```

Here we're defining our controller object, with a property, guesses, initialized to zero.

And here's the beginning of the processGuess method, which takes a guess in the form "A0".

Processing the player's guess

The controller's responsibility is to get the player's guess, make sure it's valid, and then get it to the model object. But, where does it get the player's guess? Don't worry, we'll get to that in a bit. For now we're just going to assume, at some point, some code is going to call the controller's `processGuess` method and give it a string in the form:

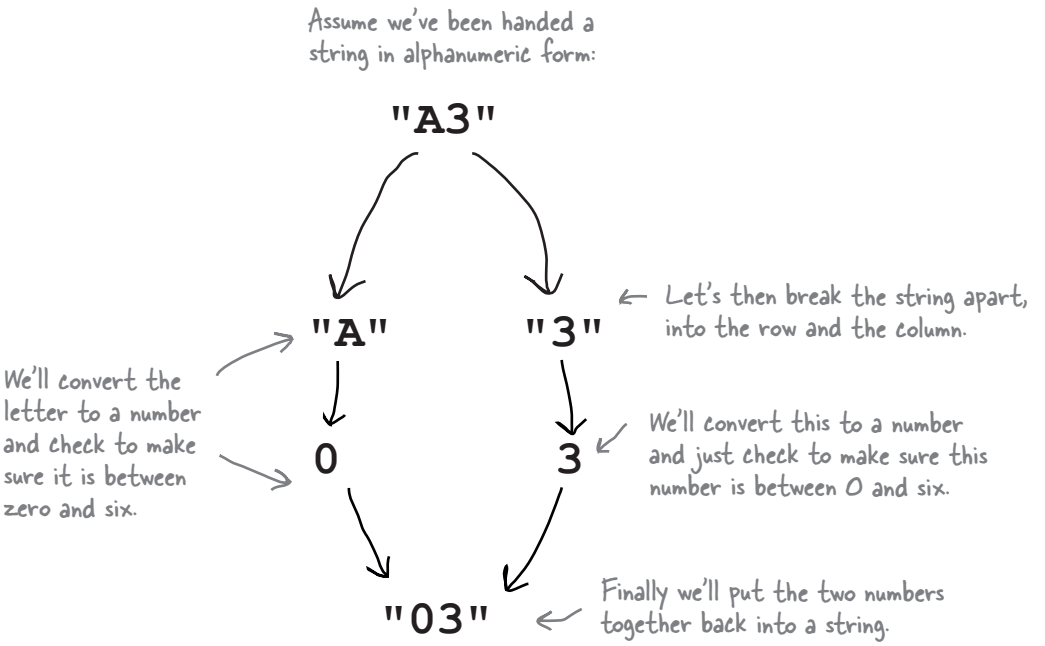
This is a great technique when you are coding. Focus on the requirements for the specific code you're working on. Thinking about the whole problem at once is often a less successful technique.

```
"A3"
```

You know the Battleship–style guess format at this point: it's a letter followed by a number.

Now after you receive a guess in this form (an alpha-numeric set of characters, like "A3"), you'll need to transform the guess into a form the model understands (a string of two numeric characters, like "03"). Here's a high level view of how we're going to convert a valid input into the number-only form:

Surely a player would never enter in an invalid guess, right? Ha! We'd better make sure we've got valid input.



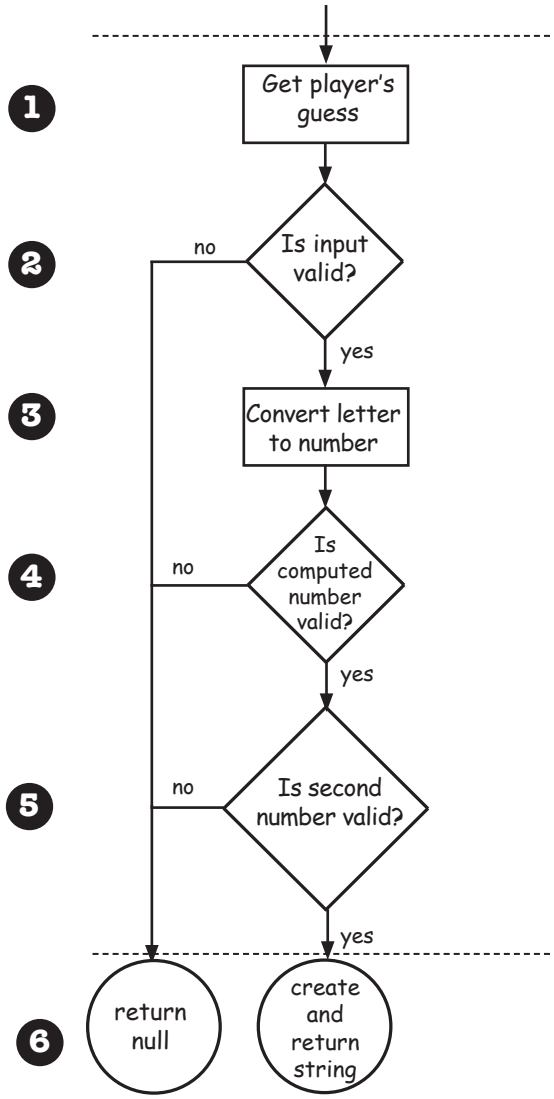
But first things first. We also need to check that the input is valid. Let's plan this all out before we write the code.

Planning the code...

Rather than putting all this guess-processing code into the `processGuess` method, we're going to write a little helper function (after all we might be able to use this again). We'll name the function `parseGuess`.

Let's step through how it is going to work before we start writing code:

1. We get a player's guess in classic Battleship-style as a single letter followed by a number.
2. Check the input to make sure it is valid (not null or too long or too short).
3. Take the letter and convert it to a number: A to 0, B to 1, and so on.
4. See if the number from step 3 is valid (between 0 and 6).
5. Check the second number for validity (also between 0 and 6).
6. If any check failed, return null. Otherwise concatenate the two numbers into a string and return the string.



Implementing parseGuess

We have a solid plan for coding this, so let's get started:

1—2 Let's tackle steps one and two. All we need to do is accept the player's guess and check to make sure it is valid. At this point we're just going to define validity as accepting a non-null string and a string that has exactly two characters in it.

The guess is passed into the guess parameter.

And then we check for null and to make sure the length is 2 characters.

```
function parseGuess(guess) {
   if (guess === null || guess.length !== 2) {
      alert("Oops, please enter a letter and a number on the board.");
   }
}
```

If not, we alert the player.

3 Next, we take the letter and convert it to a number by using a helper array that contains the letters A-F. To get the number, we can use the indexOf method to get the index of the letter in the array, like this:

An array loaded with each letter that could be part of a valid guess.

```
function parseGuess(guess) {
   var alphabet = ["A", "B", "C", "D", "E", "F", "G"];

   if (guess === null || guess.length !== 2) {
      alert("Oops, please enter a letter and a number on the board.");
   } else {
      firstChar = guess.charAt(0);
      var row = alphabet.indexOf(firstChar);
   }
}
```

Grab the first character of the guess.

Then, using indexOf, we get back a number between zero and six that corresponds to the letter. Try a couple of examples to see how this works.

4—5 Now we'll handle checking both characters of the guess to see if they are numbers between zero and six (in other words, to make sure they are both valid positions on the board).

```
function parseGuess(guess) {
   var alphabet = ["A", "B", "C", "D", "E", "F", "G"];

   if (guess === null || guess.length !== 2) {
      alert("Oops, please enter a letter and a number on the board.");
   } else {
      firstChar = guess.charAt(0);
      var row = alphabet.indexOf(firstChar);
      var column = guess.charAt(1);

      if (isNaN(row) || isNaN(column)) {
         alert("Oops, that isn't on the board.");
      } else if (row < 0 || row >= model.boardSize ||
                 column < 0 || column >= model.boardSize) {
         alert("Oops, that's off the board!");
      }
   }
}
```

Here we've added code to grab the second character in the string, which represents the column.

And we're checking to see if either of the row or column is not a number using the isNAN function.

We're also making sure that the numbers are between zero and six.

Actually we're being even more general here. Instead of hardcoding the number six, we're asking the model to tell us how big the board is and using that number for comparison.

Notice we're using type conversion like crazy here! column is a string, so when we check to make sure its value is 0–6, we rely on type conversion to convert it to a number for comparison.

Rather than hard-coding the value six as the biggest value a row or column can hold, we used the model's boardSize property. What advantage do you think that has in the long run?

6 Now for our final bit of code for the parseGuess function... If any check for valid input fails, we'll return null. Otherwise we'll return the row and column of the guess, combined into a string.

```
function parseGuess(guess) {
   var alphabet = ["A", "B", "C", "D", "E", "F", "G"];

   if (guess === null || guess.length !== 2) {
      alert("Oops, please enter a letter and a number on the board.");
   } else {
      firstChar = guess.charAt(0);
      var row = alphabet.indexOf(firstChar);
      var column = guess.charAt(1);

      if (isNaN(row) || isNaN(column)) {
         alert("Oops, that isn't on the board.");
      } else if (row < 0 || row >= model.boardSize ||
                 column < 0 || column >= model.boardSize) {
         alert("Oops, that's off the board!");
      } else {
         return row + column;
      }
   }
   return null;
}
```

At this point, everything looks good, so we can return a row and column.

Notice we're concatenating the row and column together to make a string, and returning that string. We're using type conversion again here: row is a number and column is a string, so we'll end up with a string.

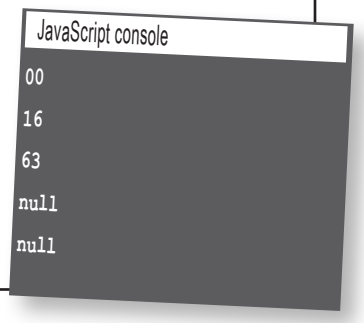
If we get here, there was a failed check along the way, so return null.

A Test Drive

Okay, make sure all this code is entered into "battleship.js" and then add some function calls below it all that look like this:

```
console.log(parseGuess("A0"));
console.log(parseGuess("B6"));
console.log(parseGuess("G3"));
console.log(parseGuess("H0"));
console.log(parseGuess("A7"));
```

Reload "battleship.html", and make sure your console window is open. You should see the results of parseGuess displayed in the console and possibly an alert or two.



Meanwhile back at the controller...

Now that we have the `parseGuess` helper function written we move on to implementing the controller. Let's first integrate the `parseGuess` function with the existing controller code:

```
var controller = {
   guesses: 0,

   processGuess: function(guess) {
      var location = parseGuess(guess);
      if (location) {

      }
   }
};
```

We'll use parseGuess to validate the player's guess.

And as long as we don't get null back, we know we've got a valid location object.

Remember null is a falsey value.

And the rest of the code for the controller will go here.

That completes the first responsibility of the controller. Let's see what's left:

- ~~Get and process the player's guess (like "A0" or "B1").~~
- Keep track of the number of guesses.
- Ask the model to update itself based on the latest guess.
- Determine when the game is over (that is, when all ships have been sunk).

We'll tackle these next.

Counting guesses and firing the shot

The next item on our list is straightforward: to keep track of the number of guesses we just need to increment the `guesses` property each time the player makes a guess. As you'll see in the code, we've chosen not to penalize players if they enter an invalid guess.

Next, we'll ask the model to update itself based on the guess by calling the model's `fire` method. After all, the point of a player's guess is to fire hoping to hit a battleship. Now remember, the `fire` method takes a string, which contains the row and column, and by some luck we get that string by calling `parseGuess`. How convenient.

Let's put all this together and implement the next step…

```
var controller = {
   guesses: 0,

   processGuess: function(guess) {
      var location = parseGuess(guess);
      if (location) {
         this.guesses++;
         var hit = model.fire(location);
      }
   }
};
```

If the player entered a valid guess we increase the number of guesses by one.

Remember, this.guesses++ just adds one to the value of the guesses property. It works just like i++ in for loops.

Also notice if the player enters an invalid board location, we don't penalize them by counting the guess.

And then we pass the row and column in the form of a string to the model's fire method. Remember, the fire method returns true if a ship is hit.

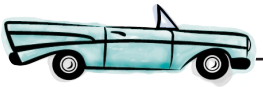
Game over?

All we have left is to determine when the game is complete. How do we do that? Well, we know that when three ships are sunk the game is over. So, each time the guess is a hit, we'll check to see if there are three sunken ships, using the `model.shipsSunk` property. Let's generalize this a bit, and instead of just comparing it to the number 3, we'll use the model's `numShips` property for the comparison. You might decide later to set the number of ships to, say, 2 or 4, and this way, you won't need to revisit this code to make it work correctly.

```
var controller = {
   guesses: 0,

   processGuess: function(guess) {
      var location = parseGuess(guess);
      if (location) {
         this.guesses++;
         var hit = model.fire(location);
         if (hit && model.shipsSunk === model.numShips) {
               view.displayMessage("You sank all my battleships, in " +
                                   this.guesses + " guesses");
         }
      }
   }
};
```

If the guess was a hit, and the number of ships that are sunk is equal to the number of ships in the game, then show the player a message that they've sunk all the ships.

We'll show the player the total number of guesses they took to sink the ship. The guesses property is a property of "this" object, the controller.

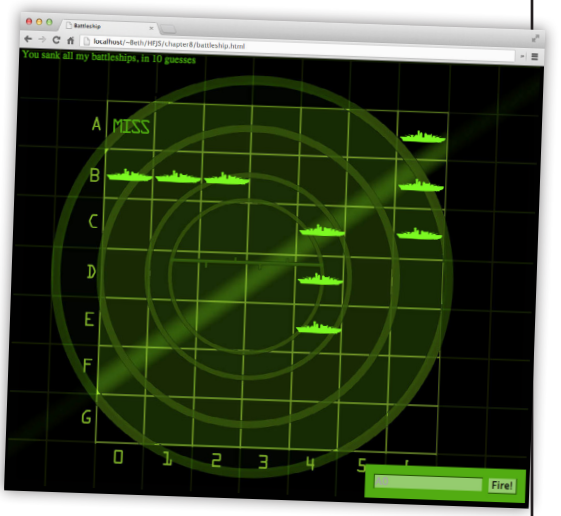


A Test Drive

Okay, make sure all the controller code is entered into your "battleship.js" file and then add some function calls below it all to test your controller. Reload your "battleship.html" page and note the hits and misses on the board. Are they in the right places? (Download "battleship_tester.js" to see our version.)

Again, you'll need to remove or comment out the previous testing code to get the same results as we show here. You can see how to do that in battleship_tester.js.

```
controller.processGuess("A0");

controller.processGuess("A6");
controller.processGuess("B6");
controller.processGuess("C6");

controller.processGuess("C4");
controller.processGuess("D4");
controller.processGuess("E4");

controller.processGuess("B0");
controller.processGuess("B1");
controller.processGuess("B2");
```

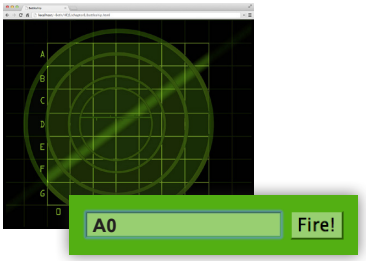


We're calling the controller's processGuess method and passing in guesses in Battleship format.

We let the player know the game ended in the message area, after they sink all three ships. But the player can still enter guesses. If you wanted to fix this so a player isn't allowed to enter guesses after they've sunk all the ships, how would you handle that?

Getting a player's guess

Now that you've implemented the core game logic and display, you need a way to enter and retrieve a player's guesses so the game can actually be played. You might remember that in the HTML we've already got a `<form>` element ready for entering guesses, but how do we hook that into the game?

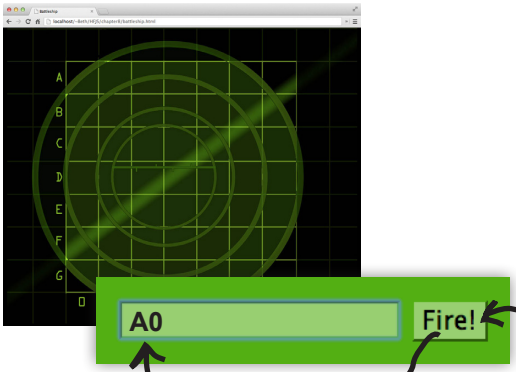


Here's our HTML form element, ready to take user input.

To do that we need an *event handler*. We've talked a little about event handlers already. For now, we're going to spend just enough time with event handlers again to get the game working, and we'll undertake learning the nitty-gritty details of event handlers in the next chapter. Our goal is for you to get a high-level understanding of how event handlers work with form elements, but not necessarily understand everything about how it works at the detailed level, right now.

Here's the big picture:

1. The player enters a guess and clicks on the Fire! button.
2. When Fire! is clicked, a pre-assigned event handler is called.
3. The handler for the Fire! button grabs the player's input from the form and hands it to the controller.



3

```
function handleFireButton() {
    // get the player's guess from the form
    // and get it to the controller.
}
```

How to add an event handler to the Fire! button

To get this all rolling the first thing we need to do is add an event handler to the Fire! button. To do that, we first need to get a reference to the button using the button's id. Review your HTML again, and you'll find the Fire! button has the id "fireButton". With that, all you need to do is call `document.getElementById` to get a reference to the button. Once we have the button reference, we can assign a handler function to the `onclick` property of the button, like this:

We need somewhere for this code to go, so let's create an init function.

First, we get a reference to the Fire! button using the button's id:

```
function init() {
   var fireButton = document.getElementById("fireButton");
   fireButton.onclick = handleFireButton;
}
```

Then we can add a click handler function named handleFireButton to the button.

And let's not forget to get a handleFireButton function started:

Here's the handleFireButton function. This function will be called whenever you click the Fire! button.

```
function handleFireButton() {
   // code to get the value from the form
}

window.onload = init;
```

We'll write this code in just a sec.

Just like we learned in Chapter 6, we want the browser to run init when the page is fully loaded.

Getting the player's guess from the form

The Fire! button is what initiates the guess, but the player's guess is actually contained in the "guessInput" form element. We can get the value from the form input by accessing the input element's `value` property. Here's how you do it:

First, we get a reference to the input form element using the input element's id, "guessInput".

```
function handleFireButton() {
    var guessInput = document.getElementById("guessInput");
    var guess = guessInput.value;
}
```

Then we get the guess from the input element. The guess is stored in the value property of the input element.

We have the value, now all we need is to do something with it. Luckily we have lots of code already that's ready to do something with it. Let's add that next.

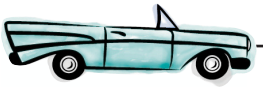
Passing the input to the controller

Here's where it all comes together. We have a controller waiting—just dying—to get a guess from the player. All we need to do is pass the player's guess to the controller. Let's do that:

```
function handleFireButton() {
    var guessInput = document.getElementById("guessInput");
    var guess = guessInput.value;
    controller.processGuess(guess);

    guessInput.value = "";
}
```

We're passing the player's guess to the controller, and then everything should work like magic!

This little line just resets the form input element to be the empty string. That way you don't have to explicitly select the text and delete it before entering the next guess, which would be annoying.

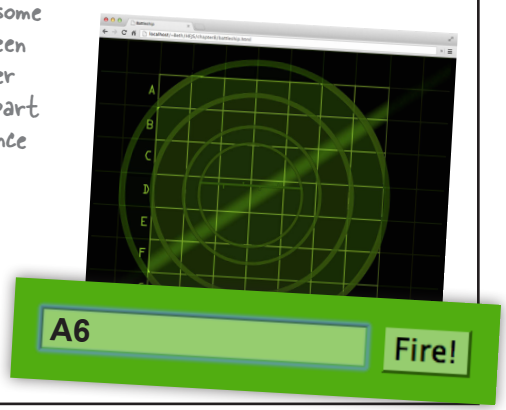


A Test Drive

This is no mere test drive. You're finally ready to play the real game! Make sure you've added all the code to "battleship.js", and reload "battleship.html" in your browser. Now, remember the ship locations are hardcoded, so you'll have a good idea of how to win this game. Below you'll find the winning moves, but be sure to fully test this code. Enter misses, invalid guesses and downright incorrect guesses.

```
A6
B6
C6

C4
D4
E4

B0
B1
B2
```

These are the winning guesses, in order by ship. But you don't have to enter them all in order. Try mixing them up a bit. Enter some invalid guesses in between the correct ones. Enter misses too. That's all part of the Quality Assurance testing for the game.



Serious Coding

Finding it clumsy to have to click the Fire! button with every guess? Sure, clicking works, but it's slow and inconvenient. It would be so much easier if you could just press RETURN, right? Here's a quick bit of code to handle a RETURN key press:

```
function init() {
    var fireButton = document.getElementById("fireButton");
    fireButton.onclick = handleFireButton;
    var guessInput = document.getElementById("guessInput");
    guessInput.onkeypress = handleKeyPress;
}
```

Add a new handler. This one handles key press events from the HTML input field.

Here's the key press handler. It's called whenever you press a key in the form input in the page.

The browser passes an event object to the handler. This object has info about which key was pressed.

```
function handleKeyPress(e) {
    var fireButton = document.getElementById("fireButton");
    if (e.keyCode === 13) {
        fireButton.click();
        return false;
    }
}
```

If you press the RETURN key, the event's keyCode property will be set to 13. If that's the case, then we want to cause the Fire! button to act like it was clicked. We can do that by calling the fireButton's click method (basically tricking it into thinking it was clicked).

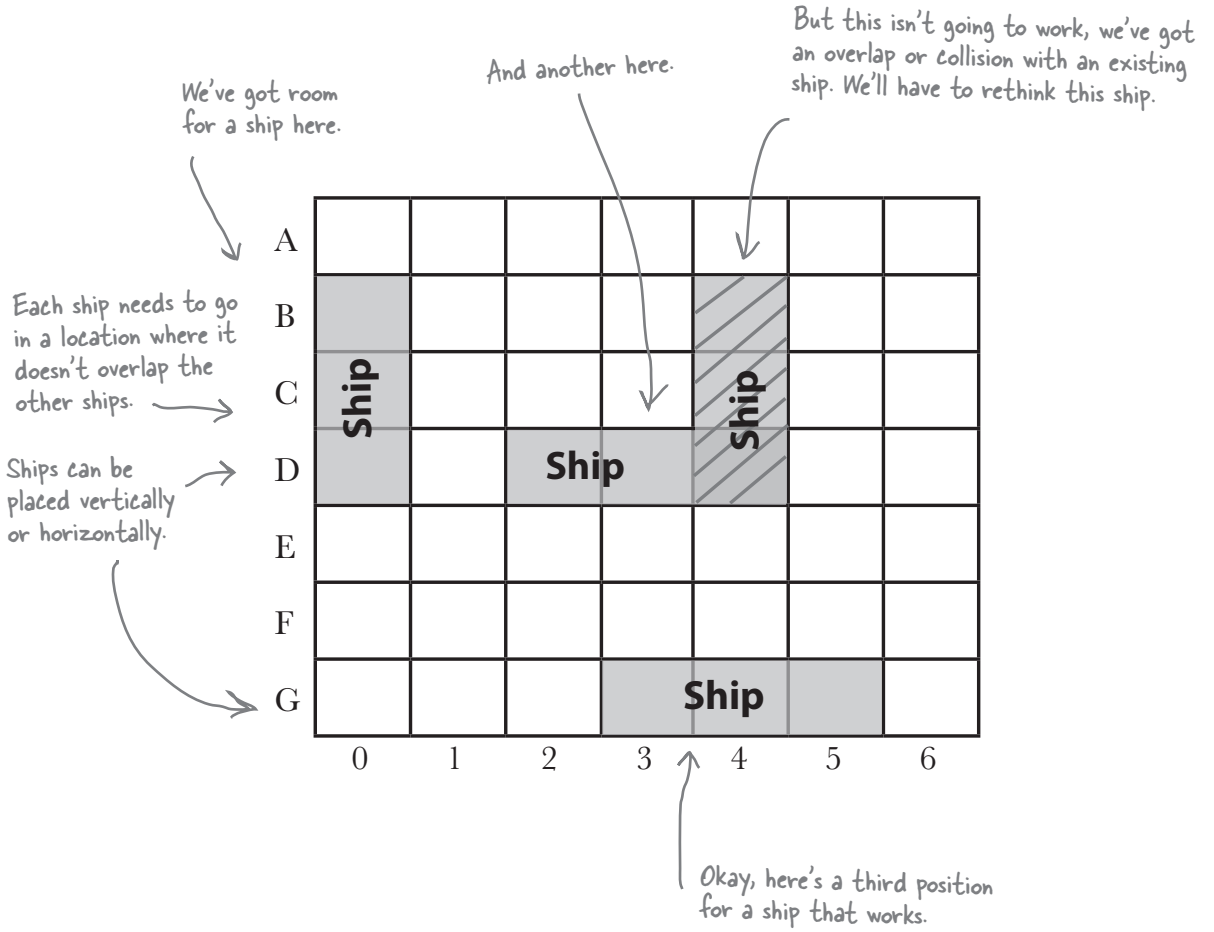
And we return false so the form doesn't do anything else (like try to submit itself).

Update your init function and add the handleKeyPress function anywhere in your code. Reload and let the game play begin!

What's left? Oh yeah, darn it, those hardcoded ships!

At this point you've got a pretty amazing browser-based game created from a little HTML, some images, and roughly 100 lines of code. But, the one aspect of this game that is a little unsatisfying is that the ships are always in the same location. You still need to write the code to generate random locations for the ships every time we start a new game (otherwise, it'll be a pretty boring game).

Now, before we start, we want to let you know that we're going to cover this code at a slightly faster clip—you're getting to the point where you can read and understand code better, and there aren't a lot of new things in this code. So, let's get started. Here's what we need to consider:



Code Magnets

An algorithm to generate ships is all scrambled up on the fridge. Can you put the magnets back in the right places to produce a working algorithm? Check your answer at the end of the chapter before you go on.

An algorithm is just a fancy word for a sequence of steps that solve a problem.

```
Generate a random
location for the new ship.
```

```
Loop for the number of
ships we want to create.
```

```
Generate a random direction
(vertical or horizontal)
for the new ship.
```

```
Add the new ship's locations
to the ships array.
```

```
Test to see if the new ship's
locations collide with any
existing ship's locations.
```

How to place ships

There are two things you need to consider when placing ships on the game board. The first is that ships can be oriented either vertically or horizontally. The second is that ships don't overlap on the board. The bulk of the code we're about to write handles these two constraints. Now, as we said, we're not going to go through the code in gory detail, but you have everything you need to work through it, and if you spend enough time with the code you'll understand each part in detail. There's nothing in it that you haven't already encountered so far in the book (with one exception that we'll talk about). So let's dive in…

We're going to organize the code into three methods that are part of the model object:

- **generateShipLocations**: This is the master method. It creates a `ships` array in the model for you, with the number of ships in the model's `numShips` property.
- **generateShip**: This method creates a single ship, located somewhere on the board. The locations may or may not overlap other ships.
- **collision**: This method takes a single ship and makes sure it doesn't overlap with a ship already on the board.

The generateShipLocations function

Let's get started with the `generateShipLocations` method. This method iterates, creating ships, until it has filled the model's `ships` array with enough ships. Each time it generates a new ship (which it does using the `generateShip` method), it uses the `collision` method to make sure there are no overlaps. If there is an overlap, it throws that ship away and keeps trying.

One thing to note in this code is that we're using a new iterator, the **do while** loop. The do while loop works almost exactly like **while**, except that you *first* execute the statements in the body, and *then* check the condition. You'll find certain logic conditions, while rare, work better with do while than with the while statement.

We're adding this method to the model object.

```
generateShipLocations: function() {
    var locations;
    for (var i = 0; i < this.numShips; i++) {
        do {
            locations = this.generateShip();
        } while (this.collision(locations));
        this.ships[i].locations = locations;
    }
},
```

For each ship we want to generate locations for.

We generate a new set of locations...

... and check to see if those locations overlap with any existing ships on the board. If they do, then we need to try again. So keep generating new locations until there's no collision.

We're using a do while loop here!

Once we have locations that work, we assign the locations to the ship's locations property in the model.ships array.

Writing the generateShip method

The `generateShip` method creates an array with random locations for one ship without worrying about overlap with other ships on the board. We'll go through this method in a couple of steps. The first step is to randomly pick a direction for the ship: will it be horizontal or vertical? We're going to determine this with a random number. If the number is 1, then the ship is horizontal; if it's 0, then the ship is vertical. We'll use our friends the `Math.random` and `Math.floor` methods to do this as we've done before:

Generating a random 0 or 1 is kind of like tossing a coin.

This method also is added to the model object.

```
generateShip: function() {
    var direction = Math.floor(Math.random() * 2);
    var row, col;

    if (direction === 1) {
        // Generate a starting location for a horizontal ship
    } else {
        // Generate a starting location for a vertical ship
    }

    var newShipLocations = [];
    for (var i = 0; i < this.shipLength; i++) {
        if (direction === 1) {
            // add location to array for new horizontal ship
        } else {
            // add location to array for new vertical ship
        }
    }
    return newShipLocations;
},
```

We use Math.random to generate a number between 0 and 1, and multiply the result by 2, to get a number between 0 and 2 (not including 2). We then turn that into a 0 or a 1 using Math.floor.

We're saying that if the direction is a 1, that means we'll create a horizontal ship...

... and if direction is 0, that means we'll create a vertical ship.

First, we'll create a starting location, like row = 0 and column = 3, for the new ship. Depending on the direction, we need different rules to create the starting location (you'll see why in just a sec).

For the new ship locations, we'll start with an empty array, and add the locations one by one.

We'll loop for the number of locations in a ship...

... and add a new location to the newShipLocations array each time through the loop. Again we need slightly different code to generate a location depending on the direction of the ship.

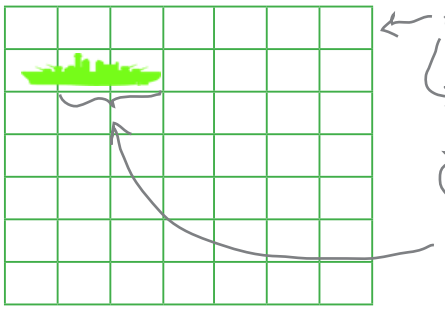
Once we've generated all the locations, we return the array.

We'll be filling in the rest of this code starting on the next page...

Generate the starting location for the new ship

Now that you know how the ship is oriented, you can generate the locations for the ship. First, we'll generate the starting location (the first position for the ship) and then the rest of the locations will just be the next two columns (if the ship is horizontal) or the next two rows (if it's vertical).

To do this we need to generate two random numbers—a row and a column—for the starting location of the ship. The numbers both have to be between 0 and 6, so the ship will fit on the game board. But remember, if the ship is going to be placed *horizontally*, then the starting *column* must be between 0 and 4, so that we have room for the rest of the ship:



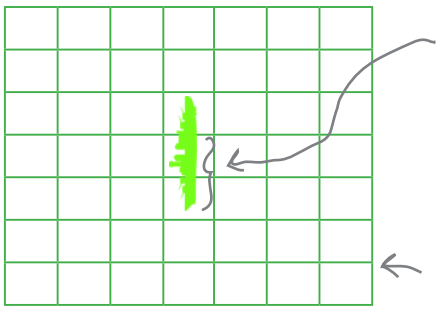
A horizontal ship can be located in any row...

```
row = Math.floor(Math.random() * this.boardSize);
col = Math.floor(Math.random() * (this.boardSize - 3));
```

... but the first column must leave room for the other two locations of the ship.

So we subtract 3 from the boardSize (7), so the starting column is always between 0 and 4. (Remember, boardSize is a property of the model.)

And, likewise, if the ship is going to be placed *vertically*, then the starting *row* must be between 0 and 4, so that we have room for the rest of the ship:



A vertical ship must start at row 0–4 to leave room for the next two locations...

```
row = Math.floor(Math.random() * (this.boardSize - 3));
col = Math.floor(Math.random() * this.boardSize);
```

... but can be located in any column.

Completing the generateShip method

Plugging that code in, now all we have to do is make sure we add the starting location along with the next two locations to the `newShipLocations` array.

```
generateShip: function() {
    var direction = Math.floor(Math.random() * 2);
    var row, col;

    if (direction === 1) {
        row = Math.floor(Math.random() * this.boardSize);
        col = Math.floor(Math.random() * (this.boardSize - this.shipLength));
    } else {
        row = Math.floor(Math.random() * (this.boardSize - this.shipLength));
        col = Math.floor(Math.random() * this.boardSize);
    }

    var newShipLocations = [];
    for (var i = 0; i < this.shipLength; i++) {
        if (direction === 1) {
            newShipLocations.push(row + "" + (col + i));
        } else {
            newShipLocations.push((row + i) + "" + col);
        }
    }
    return newShipLocations;
},
```

Here's the code to generate a starting location for the ship on the board.

We replaced 3 (from the previous page) with this.shipLength to generalize the code, so we can use it for any ship length.

This is the code for a horizontal ship. Let's break it down...

Here, we use parentheses to make sure i is added to col before it's converted to a string.

We're pushing a new location onto the newShipLocations array.

That location is a string made up of the row (the starting row we just computed above)...

... and the column + i. The first time through the loop, i is 0, so it's just the starting column. The second time, it's the next column over, and the third, the next column over again. So we'll get something like "01", "02", "03" in the array.

Same thing here only for a vertical ship.

So now, we're increasing the row instead of the column, adding i to the row each time through the loop.

For a vertical ship, we'll get something like "31", "41", "51" in the array.

Remember, when we add a string and a number, + is concatenation not addition, so we get a string.

Once we've filled the array with the ship's locations, we return it to the calling method, generateShipLocations.

Avoiding a collision!

The `collision` method takes a ship and checks to see if any of the locations overlap—or collide—with any of the existing ships already on the board.

Look back at page 364 to see where we call the collision method.

We've implemented this using two nested for loops. The outer loop iterates over all the ships in the model (in the `model.ships` property). The inner loop iterates over all the new ship's locations in the `locations` array, and checks to see if any of those locations is already taken by an existing ship on the board.

locations is an array of locations for a new ship we'd like to place on the board.

```
collision: function(locations) {
    for (var i = 0; i < this.numShips; i++) {
        var ship = model.ships[i];
        for (var j = 0; j < locations.length; j++) {
            if (ship.locations.indexOf(locations[j]) >= 0) {
                return true;
            }
        }
    }
    return false;
}
```

For each ship already on the board...

...check to see if any of the locations in the new ship's locations array are in an existing ship's locations array.

We're using indexOf to check if the location already exists in a ship, so if the index is greater than or equal to 0, we know it matched an existing location, so we return true (meaning, we found a collision).

Returning from inside a loop that's inside another loop stops the iteration of both loops immediately, exiting the function and returning true.

If we get here and haven't returned, then we never found a match for any of the locations we were checking, so we return false (there was no collision).

In this code, we have two loops: an outer loop to iterate over all the ships in the model, and an inner loop to iterate over each of the locations we're checking for a collision. For the outer loop, we used the loop variable i, and for the inner loop, we used the loop variable j. Why did we use two different loop variable names?

Two final changes

We've written all the code we need to generate random locations for the ships; now all we have to do is integrate it. Make these two final changes to your code, and then take your new Battleship game for a test drive!

```
var model = {
    boardSize: 7,
    numShips: 3,
    shipLength: 3,
    shipsSunk: 0,
    ships: [ { locations: ["06", "16", "26"], hits: ["", "", ""] },
             { locations: ["24", "34", "44"], hits: ["", "", ""] },
             { locations: ["10", "11", "12"], hits: ["", "", ""] } ],
    ships: [ { locations: [0, 0, 0], hits: ["", "", ""] },
             { locations: [0, 0, 0], hits: ["", "", ""] },
             { locations: [0, 0, 0], hits: ["", "", ""] } ],
    fire: function(guess) { ... },
    isSunk: function(ship) { ... },
    generateShipLocations: function() { ... },
    generateShip: function() { ... },
    collision: function(locations) { ... }
};

function init() {
    var fireButton = document.getElementById("fireButton");
    fireButton.onclick = handleFireButton;
    var guessInput = document.getElementById("guessInput");
    guessInput.onkeypress = handleKeyPress;

    model.generateShipLocations();
}
```

Remove the hardcoded ship locations...

... and replace them with arrays initialized with 0's instead.

And of course, add the call to generate the ship locations, which will fill in those empty arrays in the model.

We're calling model.generateShipLocations from the init function so it happens right when you load the game, before you start playing. That way all the ships will have locations ready to go when you start playing.

Don't forget you can download the complete code for the Battleship game at http://wickedlysmart.com/hfjs.

A Final Test Drive

This is the FINAL test drive of the real game, with random ship locations. Make sure you've got all the code added to "battleship.js", reload "battleship.html" in your browser, and play the game! Give it a good run through. Play it a few times, reloading the page each time to generate new ship locations for each new game.

Oh, and how to cheat!

To cheat, open up the developer console, and type `model.ships`. Press return and you should see the three ship objects containing the locations and hits arrays. Now you have the inside scoop on where the ships are sitting in the game board. But, you didn't hear this from us!

JavaScript console

```
> model.ships
[ Object,                Object,                Object ]
  hits: Array[3],          hits: Array[3],        hits: Array[3]
  locations: Array[3],     locations: Array[3],   locations: Array[3]
     0: "63"                  0: "20"                0: "60"
     1: "64"                  1: "21"                1: "61"
     2: "65"                  2: "22"                2: "62"
```

Beat the computer every time.

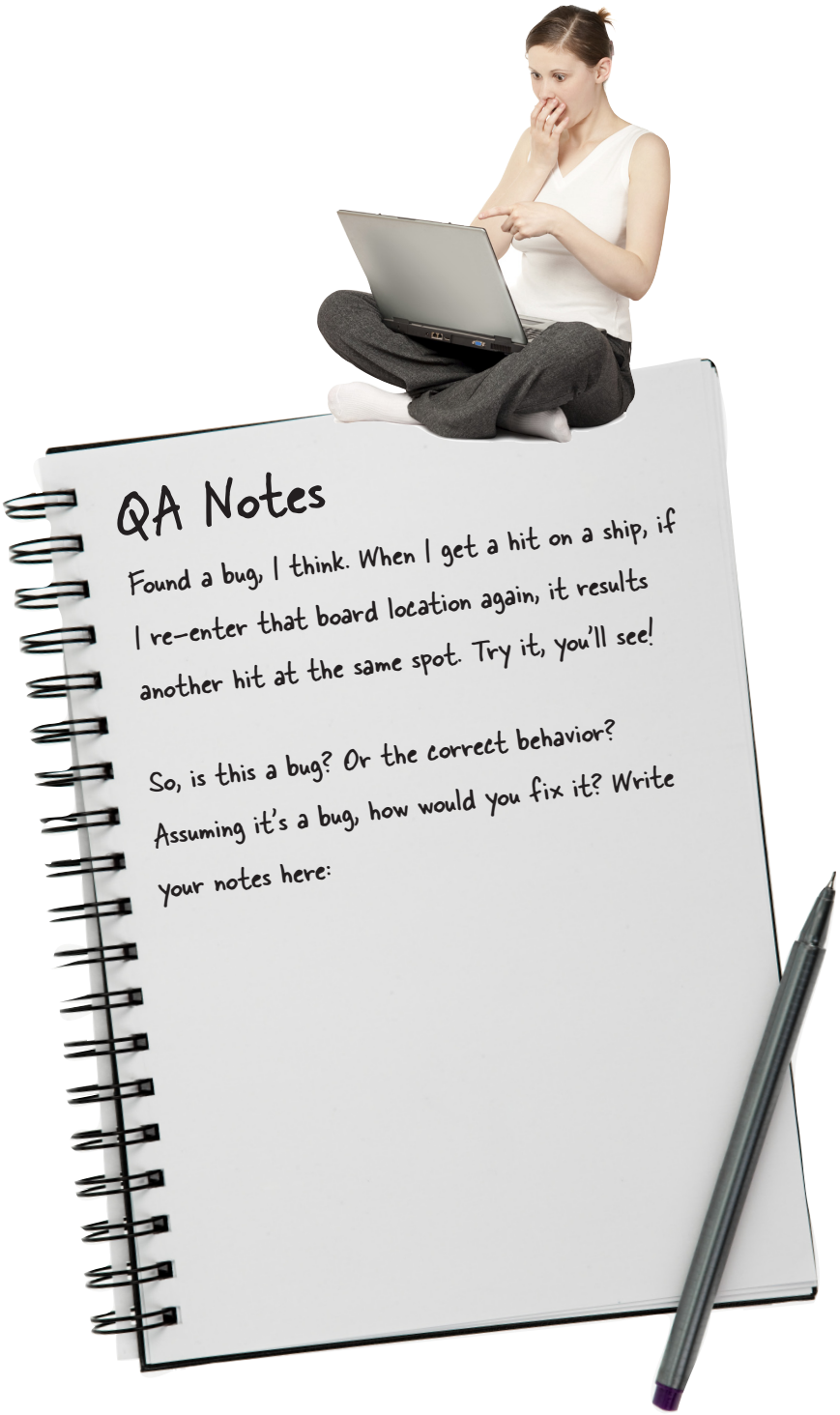
Congrats, It's Startup Time!

You've just built a great web application, all in 150 (or so) lines of code and some HTML & CSS. Like we said, the code is yours. Now all that's standing between you and your venture capital is a real business plan. But then again, who ever let that stand in their way!?

So now, after all the hard work, you can relax and play a few rounds of Battleship. Pretty darn engaging, right?

Oh, but we're just getting started. With a little more JavaScript horse power we're going to be able to take on apps that rival those written in native code.

For now, we've been through a lot of code in this chapter. Get some good food and plenty of rest to let it all sink in. But before you do that, you've got some bullet points to review and a crossword puzzle to do. Don't skip them; repetition is what really drives the learning home!

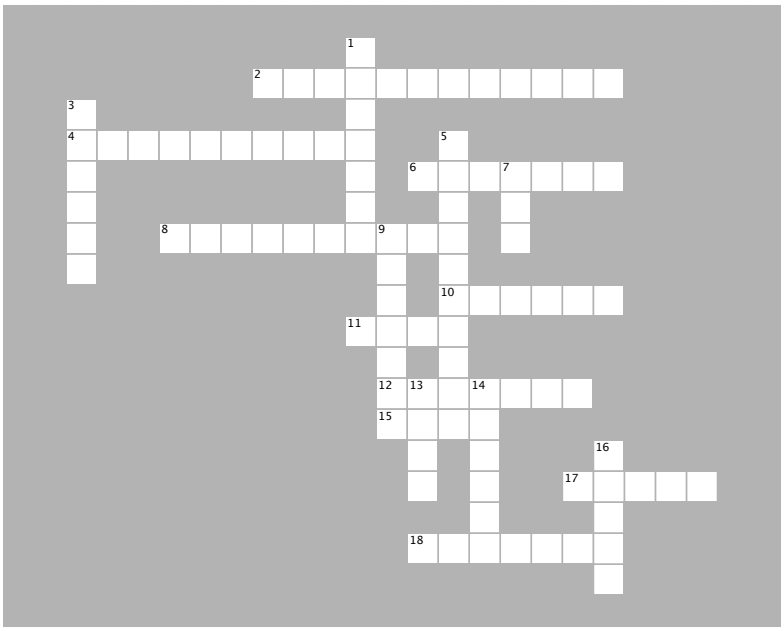

QA Notes
Found a bug, I think. When I get a hit on a ship, if I re-enter that board location again, it results another hit at the same spot. Try it, you'll see!
So, is this a bug? Or the correct behavior? Assuming it's a bug, how would you fix it? Write your notes here:

BULLET POINTS

- We use HTML to build the structure of the Battleship game, CSS to style it, and JavaScript to create the behavior.
- The id of each <td> element in the table is used to update the image of the element to indicate a HIT or a MISS.
- The form uses an input with type “button”. We attach an **event handler** to the button so we can know in the code when a player has entered a guess.
- To get a value from a form input text element, use the element’s **value** property.
- CSS positioning can be used to position elements precisely in a web page.
- We organized the code using three objects: a **model**, a **view**, and a **controller**.
- Each object in the game has one **primary responsibility**.
- The responsibility of the model is to store the state of the game and implement logic that modifies that state.
- The responsibility of the view is to update the display when the state in the model changes.
- The responsibility of the controller is to glue the game together, to make sure the player’s guess is sent to the model to update the state, and to check to see when the game is complete.
- By designing the game with objects that each have a **separate responsibility**, we can build and test each part of the game independently.
- To make it easier to create and test the model, we initially hardcoded the locations of the ships. After ensuring the model was working, we replaced these hardcoded locations with random locations generated by code.
- We used properties in the model, like numShips and shipLength, so we don’t hardcode values in the methods that we might want to change later.
- Arrays have an **indexOf** method that is similar to the string indexOf method. The array indexOf method takes a value, and returns the index of that value if it exists in the array, or -1 if it does not.
- With **chaining**, you can string together object references (using the dot operator), thus combining statements and eliminating temporary variables.
- The **do while** loop is similar to the while loop, except that the condition is checked after the statements in the body of the loop have executed once.
- **Quality assurance** (QA) is an important part of developing your code. QA requires testing not just valid input, but invalid input as well.

JavaScript cross

Your brain is frying from the coding challenges in this chapter. Do the crossword to get that final sizzle.



ACROSS

2. We use the _____ method to set the class of an element.

4. To add a ship or miss image to the board, we place the image in the _____ of a <td> element.

6. The _____ loop executes the statements in its body at least once.

8. Modern, interactive web apps use HTML, CSS and _____.

10. We represent each ship in the game with an _____.

11. The id of a <td> element corresponds to a _____ on the game board.

12. The responsibility of the collision function is to make sure that ships don't _____.

15. We call the _____ method to ask the model to update the state with the guess.

17. Who is responsible for state?

18. You can cheat and get the answers to Battleship using the _____.

DOWN

1. To get the guess from the form input, we added an event _____ for the click event.

3. Chaining is for _____ references, not just jailbirds.

5. The _____ is good at gluing things together.

7. To add a "hit" to the game board in the display, we add the ____ class to the corresonding <td> element.

9. Arrays have an _____ method too.

13. The three objects in our game design are the model, _____, and controller.

14. 13 is the keycode for the _____ key.

16. The _____ notifies the view when its state changes.

PRACTICE DRILLS SOLUTION

In just a few pages, you're going to learn how to add the MISS and ship images to the game board with JavaScript. But before we get to the real thing, you need to practice in the HTML simulator. We've got two CSS classes set up and ready for you to practice with. Go ahead and add these two rules to your CSS, and then imagine you've got ships hidden at the following locations:

Ship 1: A6, B6, C6

Ship 2: C4, D4, E4

Ship 3: B0, B1, B2

and that the player has entered the following guesses:

A0, D4, F5, B2, C5, C6

You need to add one of the two classes below to the correct cells in the grid (the correct <td> elements in the table) so that your grid shows MISS and a ship in the right places.

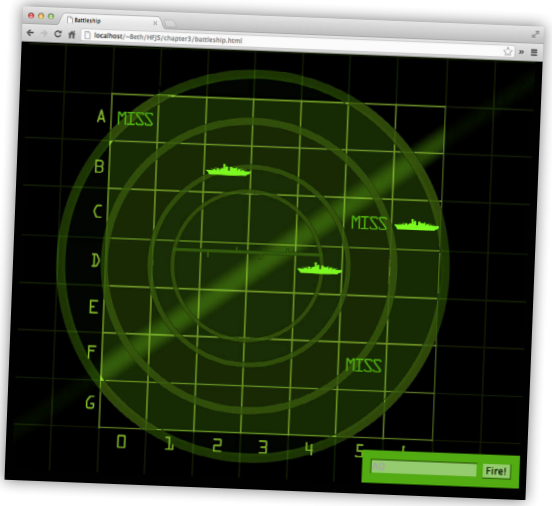
Make sure you've downloaded everything you need, including the two images you'll need for this exercise.


```
.hit {
    background: transparent url("ship.png") no-repeat center center;
}
.miss {
    background: transparent url("miss.png") no-repeat center center;
}
```

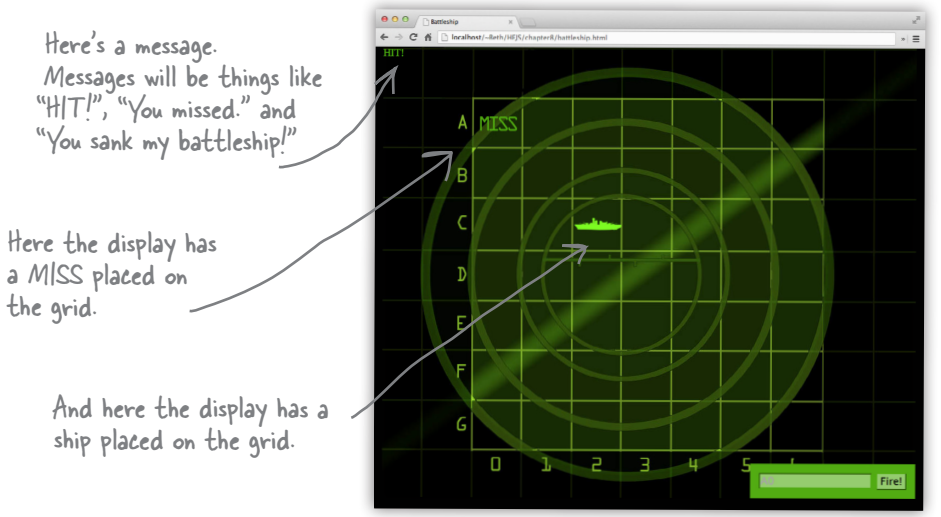
Here's our solution. The right spots for the .hit class are in <td>s with the ids: "00", "34", "55", "12", "25" and "26". To add a class to an element, you use the class attribute, like this:

```
<td class="miss" id="55">
```

After adding the classes in the right spots, your game board should look like this.



It's time for some object design. We're going to start with the view object. Now, remember, the view object is responsible for updating the view. Take a look at the view below and see if you can determine the methods we want the view object to implement. Write the declarations for these methods below (just the declarations; we'll code the bodies of the methods in a bit) along with a comment or two about what each does. Here's our solution:




```
var view = {
```

Notice we're defining an object and assigning it to the variable view.

```
    // this method takes a string message and displays it
    // in the message display area
    displayMessage: function(msg) {
        // code to be supplied in a bit!
    },

    displayHit: function(location) {
        // code will go here
     },

     displayMiss: function(location) {
        // code will go here
     }
};
```

Your methods go here!

Sharpen your pencil Solution

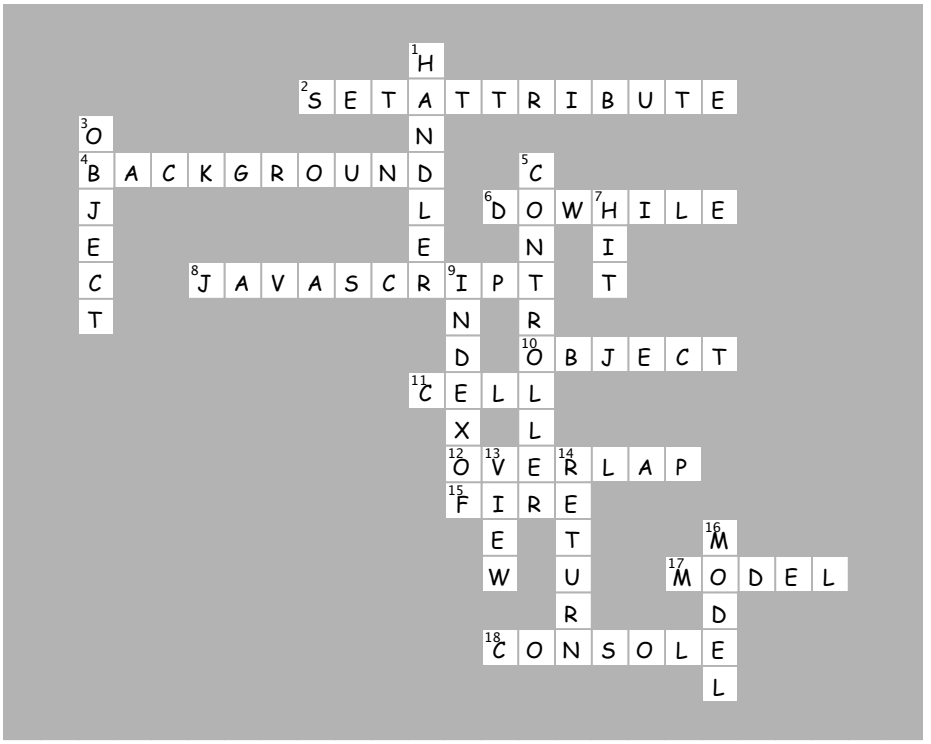
Given how we've described the new game board above, how would you represent the ships in the model (just the locations, we'll worry about hits later). Check off the best solution below.

- ❑ Use nine variables for the ship locations, similar to the way we handled the ships in Chapter 2.
- ❑ Use an array with an item for each cell in entire board (49 items total). Record the ship number in each cell that holds part of a ship.
- ❑ Use an array to hold all nine locations. Items 0-2 will hold the first ship, 3-5 the second, and so on.
- ❑ Use three different arrays, one for each ship, with three locations contained in each.
- ☑ Use an object named ship with three location properties. Put all the ships in an array named ships.
- ❑ _____

Or write in your own answer.

Any of these solutions could work! (In fact we tried each one when we were figuring out the best way to do it.) This is the one we use in the chapter.

JavaScript cross solution



Ship Magnets Solution

Use the following player moves, along with the data structure for the ships, to place the ship and miss magnets onto the game board. Does the player sink all the ships? We've done the first move for you.

Here are the moves:

A6, B3, C4, D1, B0, D4, F0, A1, C6, B1, B2, E4, B6

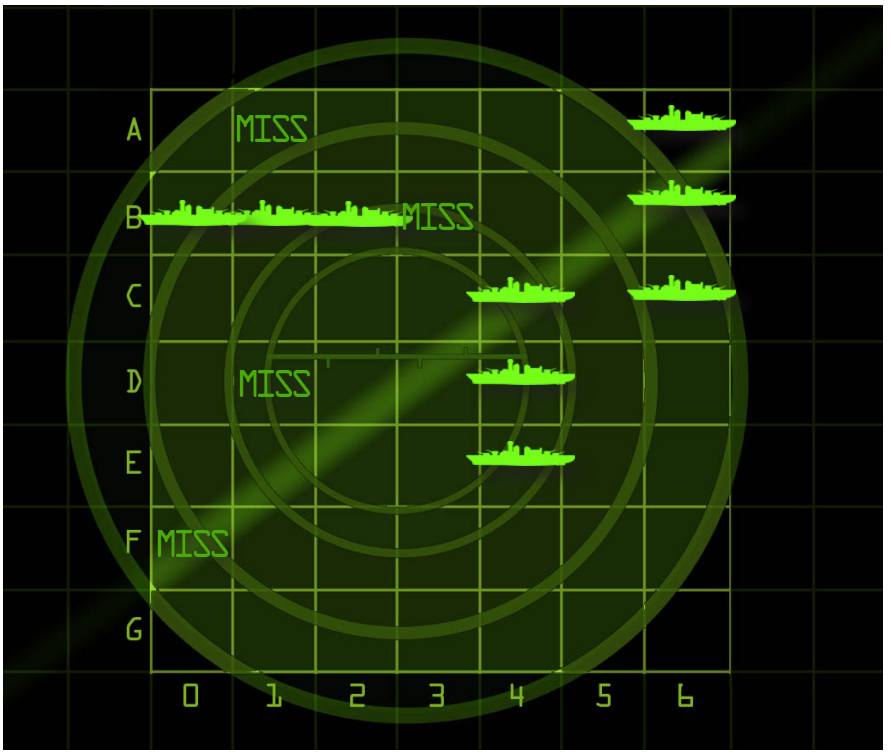
Execute these moves on the game board.

And here's our solution:

```
var ships = [{ locations: ["06", "16", "26"], hits: ["hit", "hit", "hit"] },
             { locations: ["24", "34", "44"], hits: ["hit", "hit", "hit"] },
             { locations: ["10", "11", "12"], hits: ["hit", "hit", "hit"] }];
```

All three ships are sunk!

And here's the board and your magnets.



Leftover magnets.

Sharpen your pencil Solution

Let's practice using the ships data structure to simulate some ship activities. Using the ships definition below, work through the questions and the code below and fill in the blanks. Make sure you check your answers before moving on, as this is an important part of how the game works:

```
var ships = [{ locations: ["31", "41", "51"], hits: ["", "", ""] },
             { locations: ["14", "24", "34"], hits: ["", "hit", ""] },
             { locations: ["00", "01", "02"], hits: ["hit", "", ""] }];
```

Which ships are already hit? Ships 2 and 3 And at what locations? C4, A0

The player guesses "D4", does that hit a ship? yes If so, which one? Ship 2

The player guesses "B3", does that hit a ship? no If so, which one? _____

Finish this code to access the second ship's middle location and print its value with console.log.

```
var ship2 = ships[ 1 ];
var locations = ship2.locations;
console.log("Location is " + locations[ 1 ]);
```

Finish this code to see if the third ship has a hit in its first location:

```
var ship3 = ships[ 2 ];
var hits = ship3.hits;
if (hits[0] === "hit") {
   console.log("Ouch, hit on third ship at location one");
}
```

Finish this code to hit the first ship at the third location:

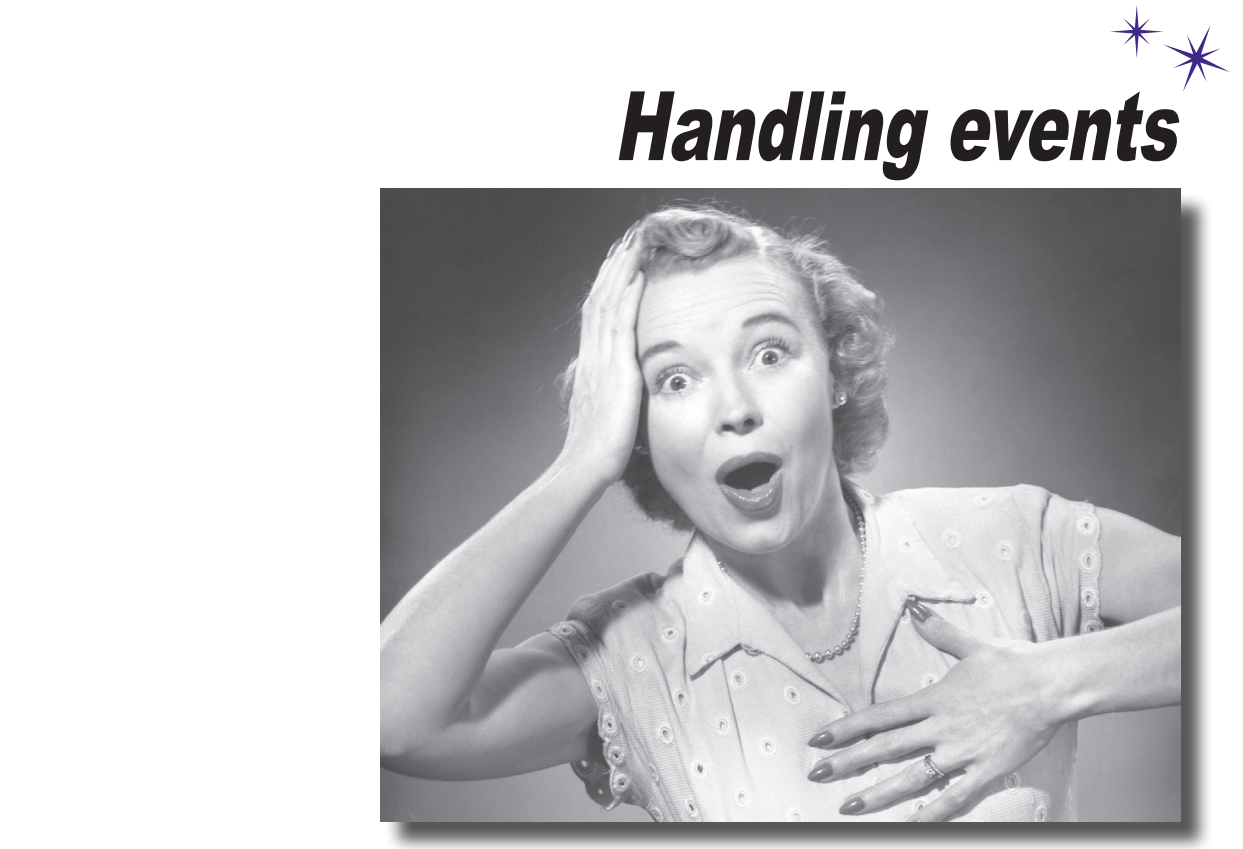
```
var ship1 = ships[0];
var hits = ship1.hits;
hits[ 2 ] = "hit";
```

Code Magnets Solution

An algorithm to generate ships is all scrambled up on the fridge. Can you put the magnets back in the right places to produce a working algorithm? Here's our solution.

```
Loop for the number of
ships we want to create.

    Generate a random direction
    (vertical or horizontal)
    for the new ship.

    Generate a random
    location for the new ship.

    Test to see if the new ship's
    locations collide with any
    existing ship's locations.

    Add the new ship's locations
    to the ships array.
```

9 asynchronous coding



Handling events

After this chapter you're going to realize you aren't in Kansas anymore. Up until now, you've been writing code that typically executes from top to bottom—sure, your code might be a little more complex than that, and make use of a few functions, objects and methods, but at some point the code just runs its course. Now, we're awfully sorry to break this to you this late in the book, but that's **not how you typically write JavaScript code**. Rather, most JavaScript is written to **react to events**. What kind of events? Well, how about a user clicking on your page, data arriving from the network, timers expiring in the browser, changes happening in the DOM and that's just a few examples. In fact, all kinds of events are happening **all the time**, behind the scenes, in your browser. In this chapter we're going rethink our approach to JavaScript coding, and learn how and why we should write code that reacts to events.

You know what a browser does, right? It retrieves a page and all that page's contents and then renders the page. But the browser's doing a lot more than just that. What else is it doing? Choose any of the tasks below you suspect the browser is doing behind the scenes. If you aren't sure just make your best guess.

- ☐ Knows when the page is fully loaded and displayed.
- ☐ Keeps track of all the clicks you make to the page, be it on a button, link or elsewhere.
- ☐ Knows when a user submits a form.
- ☐ Knows when the user presses keys on a keyboard.
- ☐ Knows when an element gets user interface focus.
- ☐ Watches all mouse movement.
- ☐ Watches the clock and manages timers and timed events.
- ☐ Retrieves additional data for your page.
- ☐ Tracks when the page has been resized or scrolled.
- ☐ Knows when the cookies are finished baking.

Sharpen your pencil

Pick two of the events above. If the browser could notify your code when these events occurred, what cool or interesting code might you write?

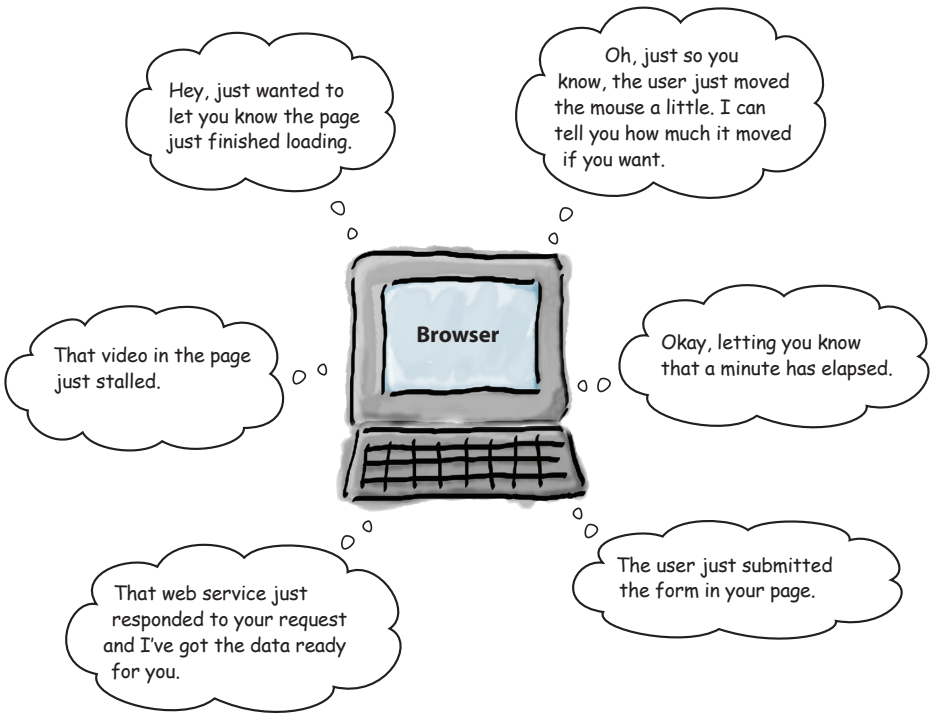
No, you can't use the cookie event as one of your examples!

What are events?

We're sure you know by now that after the browser retrieves and displays your page, it doesn't just sit there. Behind the scenes, a lot is going on: users are clicking buttons, the mouse location is being tracked, additional data is becoming available on the network, windows are getting resized, timers are going off, the browser's location could be changing, and so on. All these things cause *events* to be triggered.

Whenever there's an event, there is an opportunity for your code to *handle it*; that is, to supply some code that will be invoked when the event occurs. Now, you're not required to handle any of these events, but you'll need to handle them if you want interesting things to happen when they occur—like, say, when the button click event happens, you might want to add a new song to a playlist; when new data arrives you might want to process it and display it on your page; when a timer fires you might want to tell a user the hold on a front row concert ticket is going to expire, and so on.

A browser's geo-location, as well as a number of other advanced types of events, is something we cover in Head First HTML5 Programming. In this book we'll stick to the bread & butter foundational types of events.



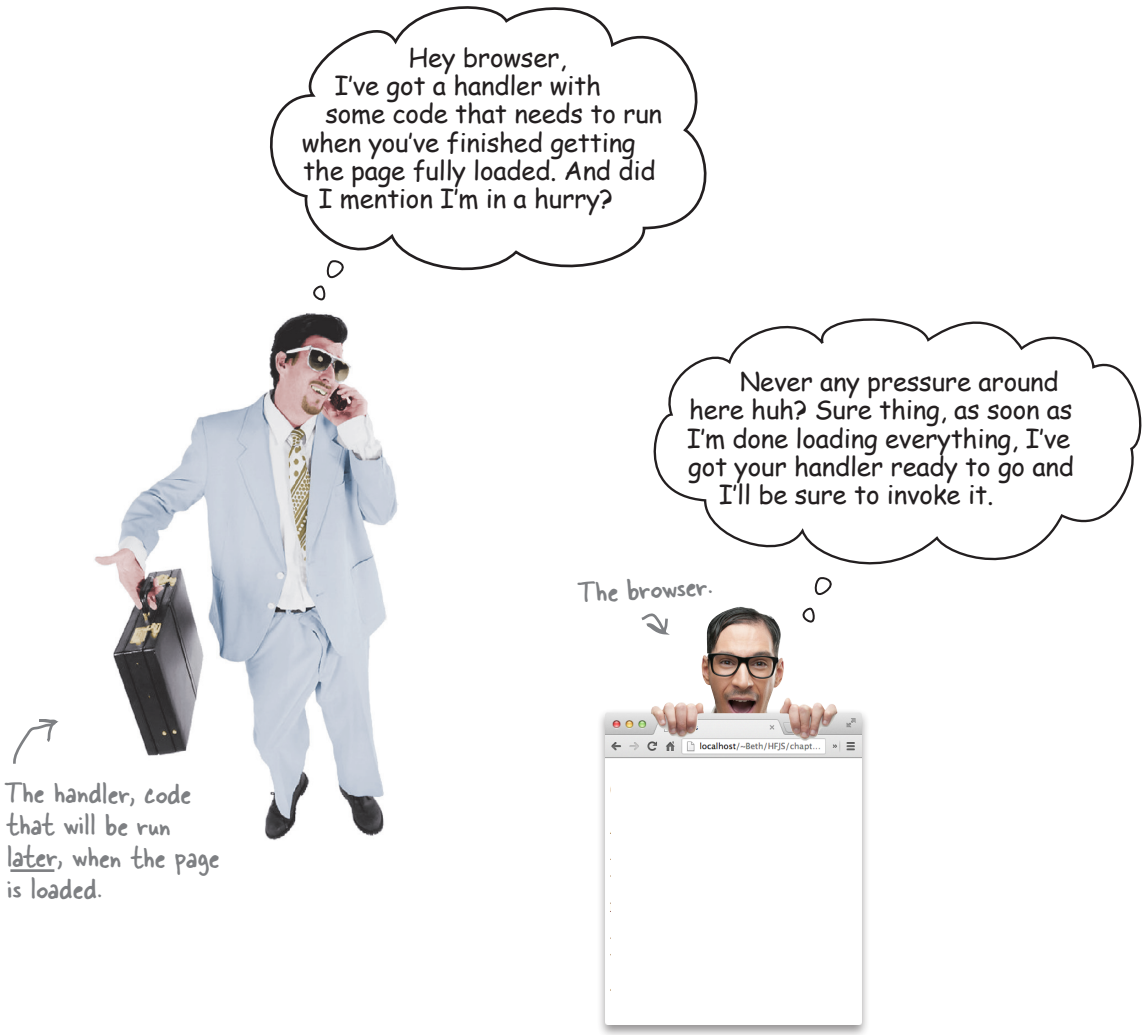
Whenever there's an event, there is an opportunity for your code to handle it.

What's an event handler?

We write *handlers* to handle events. Handlers are typically small pieces of code that know what to do when an event occurs. In terms of code, a handler is just a function. When an event occurs, its handler function is called.

You might also hear developers use the name callback or listener instead of handler.

To have your handler called when an event occurs, you first need to *register it*. As you'll see, there are a few different ways to do that depending on what kind of event it is. We'll get into all that, but for now let's get started with a simple example, one you've seen before: the event that's generated when a page is fully loaded.



How to create your first event handler

There's no better way to understand events than by writing a handler and wiring it up to handle a real, live event. Now, remember, you've already seen a couple of examples of handling events—including the page load event—but we've never fully explained how event handling works. The page load event is triggered when the browser has fully loaded and displayed all the content in your page (and built out the DOM representing the page).

Let's step through what it takes to write the handler and to make sure it gets invoked when the page load event is triggered:

1. First we need to write a function that can handle the page load event when it occurs. In this case, the function is going to announce to the world "I'm alive!" when it knows the page is fully loaded.

A handler is just an ordinary function.

```
function pageLoadedHandler() {
    alert("I'm alive!");
}
```

Here's our function, we'll name it pageLoadedHandler, but you can call it anything you like.

Remember we often refer to this as a handler or a callback.

This event handler doesn't do much. It just creates an alert.

2. Now that we have a handler written and ready to go, we need to wire things up so the browser knows there's a function it should invoke when the load event occurs. To do that we use the onload property of the window object, like this:

```
window.onload = pageLoadedHandler;
```

In the case of the load event, we assign the name of the handler to the window's onload property.

Now when the page load event is generated, the pageLoadedHandler function is going to be called.

We're going to see that different kinds of events are assigned handlers in different ways.

3. That's it! Now, with this code written, we can sit back and know that the browser will invoke the function assigned to the window.onload property when the page is loaded.

Test drive your event

Go ahead and create a new file, "event.html", and add the code to test your load event handler. Load the page into the browser and make sure you see the alert.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title> I'm alive! </title>
  <script>
    window.onload = pageLoadedHandler;
    function pageLoadedHandler() {
        alert("I'm alive!");
    }
  </script>
</head>
<body>
</body>
</html>
```

First the browser loads your page, and starts parsing the HTML and building up the DOM.

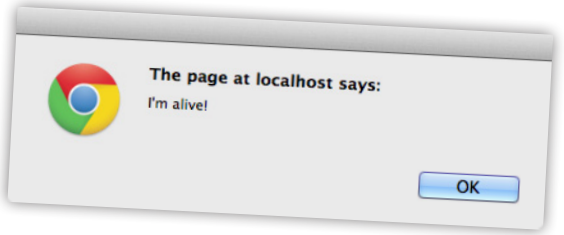
When it gets to your script the browser starts executing the code.

For now, the script just defines a function, and assigns that function to the window.onload property. Remember this function will be invoked when the page is fully loaded.

Then the browser continues parsing the HTML.

When the browser is done parsing the HTML, and the DOM is ready, the browser calls the page load handler.

Which in this case creates the "I'm alive" alert.



If we didn't have functions, could we have event handlers?

As we already mentioned, up until now you've taken a rather, let's say, linear approach to writing code: you took an algorithm, like computing the best bubble solution, or generating the 99 bottles song, and wrote the code stepwise, top to bottom.

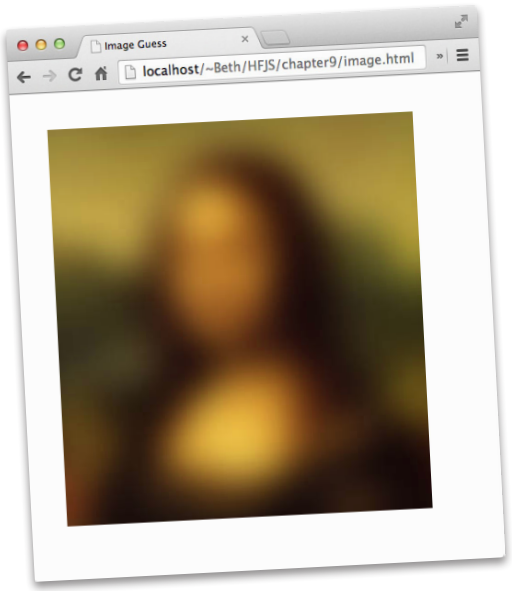
But remember the Battleship game? The code for that game didn't quite fit the linear model—sure, you wrote some code that set up the game, initialized the model, and all that—but then the main part of the game operated in a *different way*. Each time you wanted to fire at another ship you entered your guess into a form input element and pressed the "Fire" button. That button then caused a whole sequence of actions that resulted in the next move of the game being executed. In that case your code was *reacting* to the user input.

Organizing code around reacting to events is a different way of thinking about how you write your code. To write code this way, you need to consider the events that can happen, and how your code should react. Computer science types like to say that this kind of code is *asynchronous*, because we're writing code to be invoked *later*, *if* and *when* an event occurs. This kind of coding also changes your perspective from one of encoding an algorithm step-by-step into code, into one of gluing together an application that is composed of many handlers handling many different kinds of events.

Getting your head around events... by creating a game

The best way to understand events is with experience, so let's get some more by writing a simple game. The game works like this: you load a page and are presented with an image. Not just any image, but a really blurred image. Your job is to guess what the image is. And, to check your answer, you click on the image to unblur it.

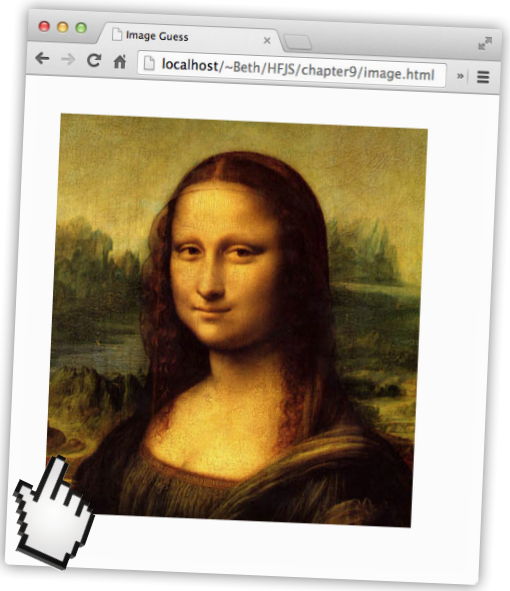
Like this:



Here's the blurred version of the image. Hmmm, what could it be?

Hum the Jeopardy theme song to yourself as you try to figure it out...

When you think you have it, click to reveal the unblurred image.



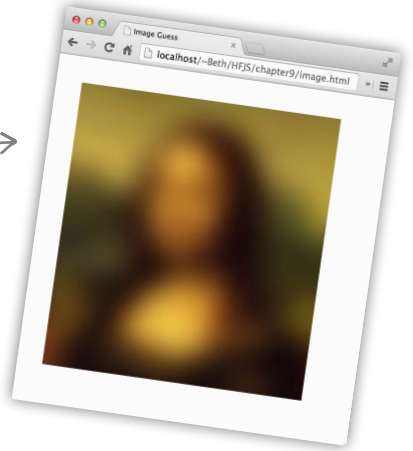
Let's start with the markup. We'll use two JPG images. One is blurred and the other isn't. We've named them "zeroblur.jpg" and "zero.jpg" respectively. Here's the markup:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title> Image Guess </title>
  <style> body { margin: 20px; } </style>
  <script>  </script>
</head>
<body>
    <img id="zero" src="zeroblur.jpg">
</body>
</html>
```

Just some basic HTML, with a <script> element all ready for our code. Rather than use a separate file for the JavaScript, we'll keep it simple and add the script here. As you'll see, there is very little code needed to implement this.

And here's the blurred image, placed in the page. We'll give it an id of "zero". You'll see how we use the id in a sec...

Implementing the game



Go ahead and load this markup in your browser and you'll see the blurred image. To implement the game, we need to react to a click on the image in order to display the unblurred version of the image.

Lucky for us, every time an HTML element in the page is clicked (or touched on a mobile device), an event is generated. Your job is to create a handler for that event, and in it write the code to display the unblurred version of the image. Here's how you're going to do that:

1. **Access the image object in the DOM and assign a handler to its onclick property.**

2. **In your handler, write the code to change the image src attribute from the blurred image to the unblurred one.**

Let's walk through these steps and write the code.

Step 1: access the image in the DOM

Getting access to the image is old hat for you; we just need to use our old friend, the `getElementById` method, to get a reference to it.

```
var image = document.getElementById("zero");
```

Here we're grabbing a reference to the image element and assigning it to the image variable.

Oh, but we also need this code to run only *after* the DOM for the page has been created, so let's use the window's `onload` property to ensure that. We'll place our code into a function, `init`, that we'll assign to the `onload` property.

Remember, we can't get the image from the DOM until the page has finished loading.

We create a function init, and assign it to the onload handler to make sure this code doesn't run until the page is fully loaded.

```
window.onload = init;
function init() {
    var image = document.getElementById("zero");
}
```

In the code of init, we'll grab a reference to the image with id="zero".

Remember in JavaScript the order in which you define your functions doesn't matter. So we can define init after we assign it to the onload property.

Step 2: add the handler, and update the image

To add a handler to deal with clicks on the image, we simply assign a function to the image's `onclick` property. Let's call that function `showAnswer`, and we'll define it next.

```
window.onload = init;
function init() {
    var image = document.getElementById("zero");
    image.onclick = showAnswer;
}
```

Using the image object from the DOM, we're assigning a handler to its onclick property.

Now we need to write the `showAnswer` function, which unblurs the image by resetting the image element's `src` property to the unblurred image:

First, we have to get the image from the DOM again.

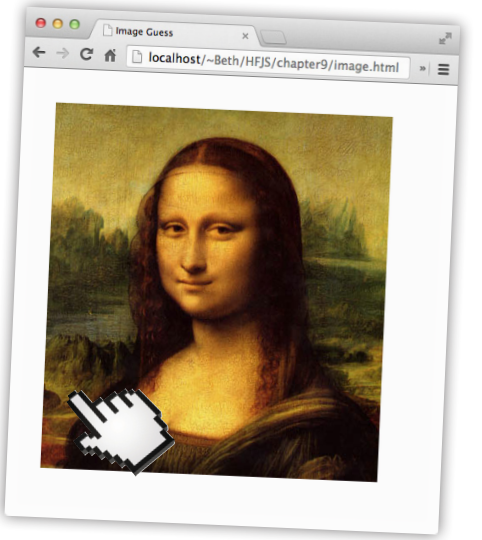
```
function showAnswer() {
    var image = document.getElementById("zero");
    image.src = "zero.jpg";
}
```

Remember the blurred version is named "zeroblur.jpg" and the unblurred is named "zero.jpg".

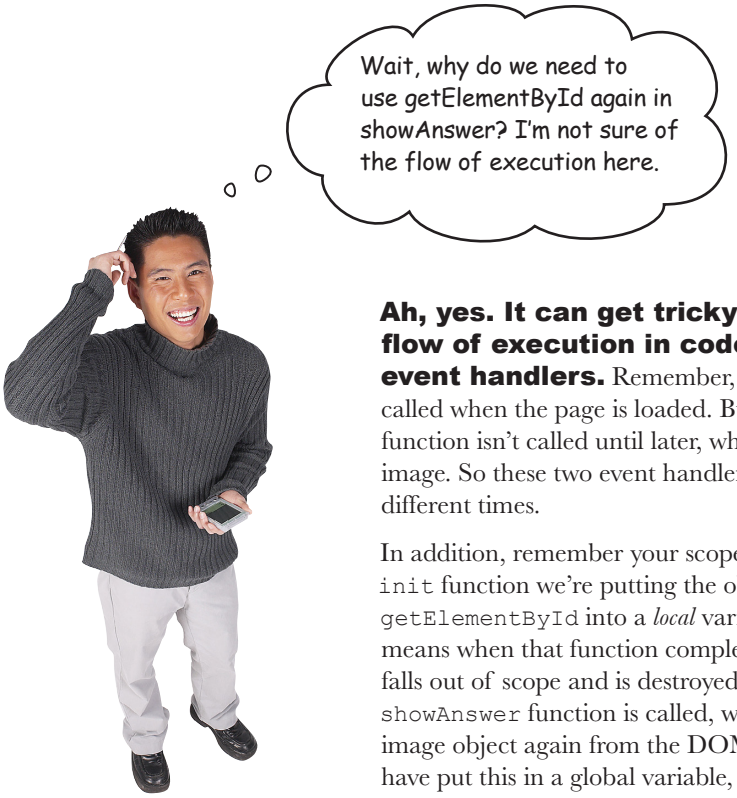
Once we have the image, we can change it by setting its src property to the unblurred image.

Test drive

Let's take this simple game for a test drive. Make sure you've got all the HTML, CSS and JavaScript typed into a file named "image.html", and that you've got the images you downloaded from http://wickedlysmart.com/hfjs in the same folder. Once all that's done, load up the file in your browser and give it a try!



Click anywhere on the image to have the showAnswer handler called. When that happens, the src of the image is changed to reveal the answer.



Ah, yes. It can get tricky to follow the flow of execution in code with a lot of event handlers. Remember, the `init` function is called when the page is loaded. But the `showAnswer` function isn't called until later, when you click the image. So these two event handlers get called at two different times.

In addition, remember your scope rules. In the `init` function we're putting the object returned by `getElementById` into a *local* variable `image`, which means when that function completes, the variable falls out of scope and is destroyed. So later, when the `showAnswer` function is called, we have to get the image object again from the DOM. Sure, we could have put this in a global variable, but over use of globals can lead to confusing and buggy code, which we'd like to avoid.

there are no Dumb Questions

Q: Is setting the src property of the image the same as setting the src attribute using setAttribute?

A: In this case, yes, it is. When you get an HTML element from the DOM using getElementById, you're getting an element object that has several methods and properties. All element objects come with a property, id, that is set to the id of the HTML element (if you've given it one in your HTML). The image element object also comes with a src property that is set to the image file specified in the src attribute of the <img> element.

Not all attributes come with corresponding object properties, however, so you will need to use setAttribute and getAttribute for those. And in the case of src and id, you can use either the properties or get/set them using getAttribute and setAttribute and it does the same thing.

Q: So do we have a handler called within a handler?

A: Not really. The load handler is the code that is called when the page is fully loaded. When the load handler is called, we assign a handler to the image's onclick property, but it won't be called until you actually click on the image. When you do that (potentially a long time after the page has loaded), the showAnswer click handler is called. So the two handlers get called at different times.

BE the Browser

Below, you'll find your game code. Your job is to play like you're the browser and to figure out what you need to do after each event. After you've done the exercise, look at the end of the chapter to see if you got everything. We've done the first bit for you.

```
window.onload = init;
function init() {
    var image = document.getElementById("zero");
    image.onclick = showAnswer;
}

function showAnswer() {
    var image = document.getElementById("zero");
    image.src = "zero.jpg";
}
```

Here's the code you're executing...

When page is being loaded...

First define the functions init and showAnswer

When page load event occurs...

When image click event occurs...

Your answers go here.

BRAIN POWER

What if you had an entire page of images that could each be individually deblurred by clicking? How would you design your code to handle this? Make some notes. What might be the naive way of implementing this? Is there a way to implement this with minimal code changes to what you've already written?

Judy: Hey guys. So far the image guessing game works great. But we really should expand the game to include more images on the page.

Jim: Sure, Judy, that's exactly what I was thinking.

Joe: Hey, I've already got a bunch of images ready to go, we just need the code. I've followed the naming convention of "zero.jpg", "zeroblur.jpg", "one.jpg", "oneblur.jpg", and so on...

Jim: Are we going to need to write a new click event handler for each image? That's going to be a lot of repetitive code. After all, every event handler's going to do exactly the same thing: replace the blurred image with its unblurred version, right?

Joe: That's true. But I'm not sure I know how to use the same event handler for multiple images. Is that even possible?

Judy: What we can do is assign the same handler, which really means the same function, to the `onclick` property of every image in the game.

Joe: So the same function gets called for every image that is clicked on?

Judy: Right. We'll use `showAnswer` as the handler for every image's click event.

Jim: Hmm, but how will we know which image to deblur?

Joe: What do you mean? Won't the click handler know?

Jim: How will it know? Right now, our `showAnswer` function assumes we clicked on the image with the id "zero". But if we're calling `showAnswer` for every image's click event, then our code needs to work for any of the images.

Joe: Oh… right… so how do we know which image was clicked?

Judy: Actually I've been reading up on events, and I think there is a way for the click handler to know the element the user clicked on. But let's deal with that part later. First let's add some more images to the game, and see how to set the same event handler for all of them... then we'll figure out how to determine which image the user clicked.

Joe, Jim: Sounds good!

Let's add some more images

We've got a whole set of new images, so let's start by adding them to the page. We'll add five more images for a total of six. We'll also modify the CSS to add a little whitespace between the images:

Get the images

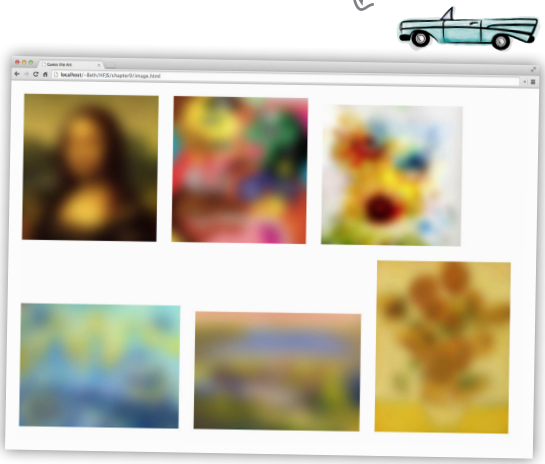
You'll find all the images in the chapter9 folder you downloaded from http://wickedlysmart.com/hfjs.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title> Image Guess </title>
  <style>
    body { margin: 20px; }
    img { margin: 20px; }
  </style>
  <script>
     window.onload = init;
     function init() {
        var image = document.getElementById("zero");
        image.onclick = showAnswer;
    }
    function showAnswer() {
        var image = document.getElementById("zero");
        image.src = "zero.jpg";
    }
  </script>
</head>
<body>
    <img id="zero" src="zeroblur.jpg">
    <img id="one" src="oneblur.jpg">
    <img id="two" src="twoblur.jpg">
    <img id="three" src="threeblur.jpg">
    <img id="four" src="fourblur.jpg">
    <img id="five" src="fiveblur.jpg">
</body>
</html>
```

We're just adding a margin of 20px between the images with this CSS property.

And here are the five new images we're adding. Notice we're using the same id & src naming scheme (and image naming scheme) for each one. You'll see how this is going to work in a bit...

If you give this a quick test drive, your page should look like this:



Now we need to assign the same event handler to each image's onclick property

Now we have more images in the page, but we have more work to do. Right now you can click on the first image (of the *Mona Lisa*) and see the unblurred image, but what about the other images?

We *could* write a new, separate handler function for each image, but, from the discussion so far you know that would be tedious and wasteful. Here, have a look:

```
window.onload = init;
function init() {
    var image0 = document.getElementById("zero");
    image0.onclick = showImageZero;
    var image1 = document.getElementById("one");
    image1.onclick = showImageOne;
    ...
}
function showImageZero() {
    var image = document.getElementById("zero");
    image.src = "zero.jpg";
}
function showImageOne() {
    var image = document.getElementById("one");
    image.src = "one.jpg";
}
...
```

We could get each image element from the page and assign a separate click handler to each one. We'd have to do this six times... we're only showing two here.

The other four would be set here.

And we'd need six different click handlers, one for each image.

And we'd need four more handler functions here.

What are the disadvantages of writing a separate handler for each image? Check all that apply:

- ☐ Lots of redundant code in each handler.
- ☐ If we need to change the code in one handler, we're probably going to have to change them all.
- ☐ Generates a lot of code.
- ☐ Hard to keep track of all the images and handlers.
- ☐ Hard to generalize for an arbitrary number of images.
- ☐ Harder for others to work on the code.

How to reuse the same handler for all the images

Clearly writing a handler for each image isn't a good way to solve this problem. So what we're going to do instead is use our existing handler, `showAnswer`, to handle all of the click events for all the images. Of course, we'll need to modify `showAnswer` a little bit to make this work. To use `showAnswer` for all the images we need to do two things:

1. **Assign the showAnswer click handler function to every image on the page.**

2. **Rework showAnswer to handle unblurring any image, not just zero.jpg.**

And we'd like to do both these things in a generalized way that works even if we add more images to the page. In other words, if we write the code right, we should be able to add images to the page (or delete images from the page) without any code changes. Let's get started.

Assigning the click handler to all images on the page

Here's our first hurdle: in the current code we use the `getElementById` method to grab a reference to image "zero", and assign the `showAnswer` function to its `onclick` property. Rather than hardcoding a call to `getElementById` for each image, we're going to show you an easier way: we'll grab all the images at once, iterate through them, and set up the click handler for each one. To do that we'll use a DOM method you haven't seen yet: `document.getElementsByTagName`. This method takes a tag name, like `img` or `p` or `div`, and returns a list of elements that match it. Let's put it to work:

```
function init() {
    var image = document.getElementById("zero");
    image.onclick = showAnswer;

    var images = document.getElementsByTagName("img");
    for (var i = 0; i < images.length; i++) {
        images[i].onclick = showAnswer;
    }
};
```

We'll get rid of the old code to get image "zero" and set its handler.

Now we're getting elements from the page using a tag name, img. This finds every image in the page and returns them all. We store the resulting images in the images variable.

Then we iterate over the images, and assign the showAnswer click handler to each image in turn. Now the onclick property of each image is set to the showAnswer handler.

document.getElementsByTagName Up Close

The `document.getElementsByTagName` method works a lot like `document.getElementById`, except that instead of getting an element by its id, we're getting elements by tag name, in this case the tag name "img". Of course, your HTML can include many `<img>` elements, so this method may return many elements, or one element, or even zero elements, depending on how many images we have in our page. In our image game example, we have six `<img>` elements, so we'll get back a list of six image objects.

What we get back is a list of element objects that match the specified tag name.

```
var images = document.getElementsByTagName("img");
```

What's returned is an array–like list of objects. It's not exactly an array but has qualities similar to an array.

Notice the "s" here. That means we might get many elements back.

Put the tag name in quotes here (and don't include the < and >!).

there are no Dumb Questions

Q: You said getElementsByTagName returns a list. Do you mean an array?

A: It returns an object that you can treat like an array, but it's actually an object called a NodeList. A NodeList is a collection of Nodes, which is just a technical name for the element objects that you see in the DOM tree. You can iterate over this collection by getting its length using the length property, and then access each item in the NodeList using an index with the bracket notation, just like an array. But that's pretty much where the similarities of a NodeList and an array end, so beyond this, you'll need to be careful in how you deal with the NodeList object. You typically won't need to know more about NodeList until you want to start adding and removing elements to and from the DOM.

Q: So I can assign a click handler to any element?

A: Pretty much. Take any element on the page, get access to it, and assign a function to its onclick property. Done. As you've seen, that handler might be specific to that one element, or you might reuse a handler for events on many elements. Of course elements that don't have a visual presence in your page, like the <script> and <head> elements, won't support events like the click event.

Q: Do handler functions ever get passed any arguments?

A: Ah, good question, and very timely. They do, and we're just about to look at the event object that gets passed to some handlers.

Q: Do elements support other types of events? Or is the click the only one?

A: There are quite a few others; in fact, you've already seen another one in the code of the battleship game: the keypress event. There, an event handler function was called whenever the user pressed Enter from the form input. We'll take a look at a few other event types in this chapter.

Okay Judy, now we have a single event handler, showAnswer, to handle clicks on all the images. You said you know how to tell which image was clicked on when showAnswer is called?

Judy: Yes I do. Whenever the click event handler is called, it's passed an *event object*. You can use that object to find out details about the event.

Joe: Like which image was clicked on?

Judy: Well, more generally, the element on which the event occurred, which is known as the *target*.

Joe: What's the target?

Judy: Like I said, it's the element that generated the event. Like if you click on a specific image, the target will be that image.

Joe: So if I click on the image with the id "zero", then the target will be set to that image?

Judy: More precisely, the element object that represents that image.

Joe: Come again?

Judy: Think of the element object as exactly the same thing you get if you call `document.getElementById` with a value of "zero". It's the object that represents the image in the DOM.

Joe: Okay, so how do we get this target? It sounds like that's what we need to know which image was clicked on.

Judy: The target is just a property of the event object.

Joe: Great. That sounds perfect for `showAnswer`. We'll be done in a snap… Wait, so `showAnswer` is passed the event object?

Judy: That's right.

Joe: So how did our code for the `showAnswer` function work up until now? It's being passed this event object, but we don't have a parameter defined for the event object in the function!

Judy: Remember, JavaScript lets you ignore parameters if you want.

Joe: Oh, right.

Judy: Now Joe, don't forget, you'll need to figure out how to change the `src` of the image to the correct name for the unblurred version. Right now we're assuming the name of the unblurred image is "zero.jpg", but that won't work any more.

Joe: Maybe we can use the `id` attribute of the image to figure out the unblurred image name. The ids of all the images match the names of the unblurred version of each image.

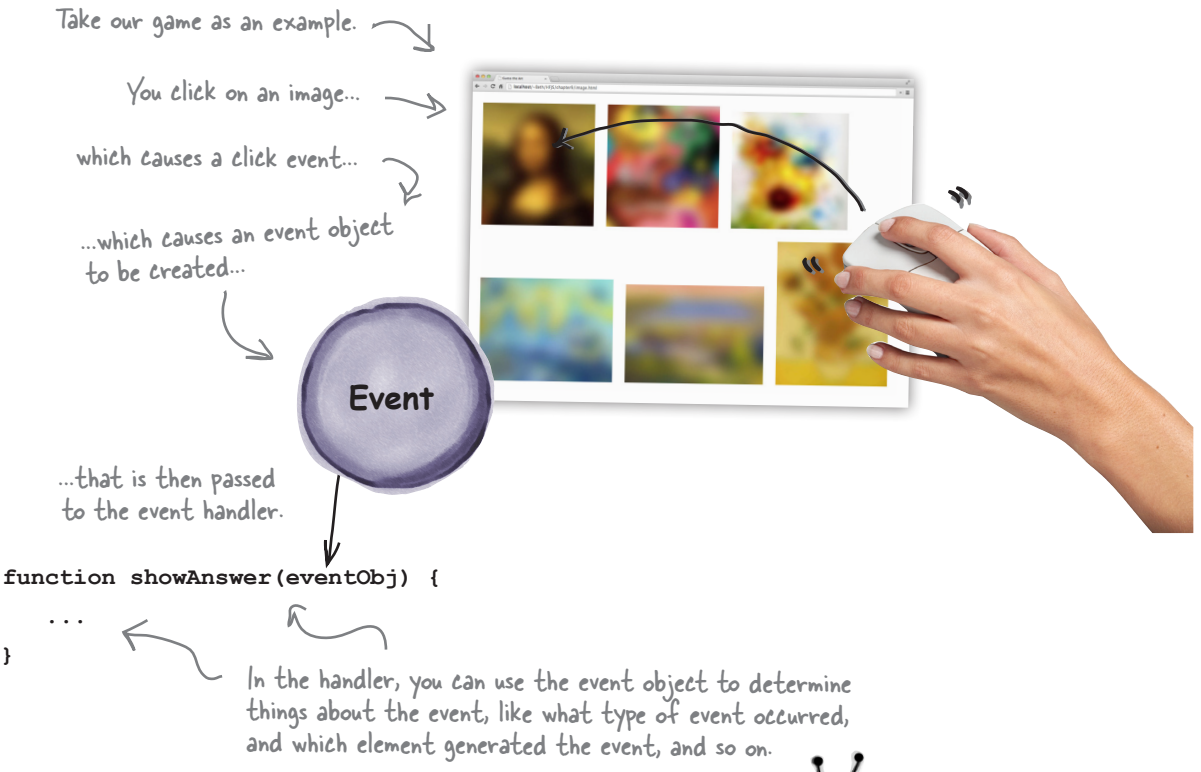
Judy: Sounds like a plan!

How the event object works

When the click handler is called, it's passed an *event object*—and in fact, for most of the events associated with the document object model (DOM) you'll be passed an event object. The event object contains general information about the event, such as what element generated the event and what time the event happened. In addition, you'll get information specific to the event, so if there was a mouse click, for instance, you'll get the coordinates of the click.

There are other kinds of events too (that is, other than DOM events), and we'll see an example later in the chapter...

Let's step through how event objects work:




```
function showAnswer(eventObj) {
  ...
}
```

In the handler, you can use the event object to determine things about the event, like what type of event occurred, and which element generated the event, and so on.

So, what is *in* an event object? Like we said, both general and specific information about the event. The specific information depends on the type of the event, and we'll come back to that a bit. The general information includes the `target` property that holds a reference to the object that generated the event. So, if you click on a page element, like an image, that's the target, and we can access it like this:

```
function showAnswer(eventObj) {
  var image = eventObj.target;
}
```

The target tells us what element generated the event.

Watch it!

If you're running IE8 or older, check the appendix.

With older versions of IE, you need to set up the event object a little differently.

You've already seen that the event object (for DOM events) has properties that give you more information about the event that just happened. Below you'll find other properties that the event object can have. Match each event object property to what it does.

target	I hold the object on which the event occurred. I can be different kinds of objects, but most often I'm an element object.
type	I'm a string, like "click" or "load", that tells you what just happened.
timeStamp	Want to know when your event happened? I'm the property for you.
keyCode	I'll tell you what key the user just pressed.
clientX	Want to know how far from the top of the browser window the user clicked? Use me.
clientY	Want to know how far from the left side of the browser window the user clicked? Use me.
touches	Using a touch device? Then use me to find out how many fingers are touching the screen.

Putting the event object to work

So, now that we've learned a little more about events—or more specifically, how the event object is passed to the click handler—let's figure out how to use the information in the event object to deblur any image on the page. We'll start by revisiting the HTML markup.

```
<!doctype html>
 ...
<body>
    <img id="zero" src="zeroblur.jpg">
    <img id="one" src="oneblur.jpg">
    <img id="two" src="twoblur.jpg">
    <img id="three" src="threeblur.jpg">
    <img id="four" src="fourblur.jpg">
    <img id="five" src="fiveblur.jpg">
</body>
</html>
```

Here's the HTML again.

Each of the images has an id, and the id corresponds to the unblurred image name. So the image with id "zero" has an unblurred image of "zero.jpg". And the image with id "one" has an unblurred image of "one.jpg" and so on...

Notice that the value of each image's id corresponds to the name of the unblurred image (minus the ".jpg" extension). Now, if we can access this id, then we can simply take that name and add on ".jpg" to create the name of the corresponding unblurred image. Once we have that we can change the image `src` property to the unblurred version of the image. Let's see how:

Remember you're getting passed an event object each time an image is clicked on.

```
function showAnswer(eventObj) {
   var image = eventObj.target;

   var name = image.id;
   name = name + ".jpg";
   image.src = name;
}
```

The event object's target property is a reference to the image element that was clicked.

We can then use the id property of that object to get the name of the unblurred image.

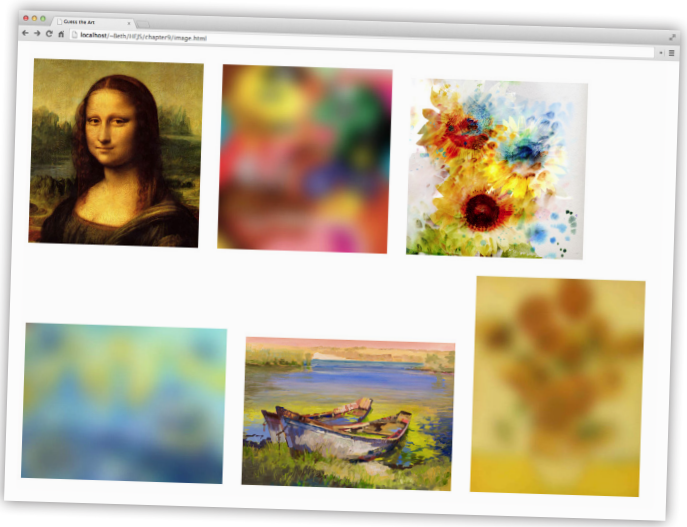
And finally, we'll set the src of the image to that name.

As you know, once you change the src property of the image, the browser will immediately retrieve that new image and display it in the page in place of the blurred version.

Test drive the event object and target

Make sure you've updated all the code in your "image.html" file, and take it for a test drive. Guess the image, click, and see the unblurred version be revealed. Think about how this app is designed not as a top-to-bottom running program, but purely as a set of actions that result from an event being generated when you click on an image. Also think about how you handled the events on all the images with one piece of code that's smart enough to know which image was clicked on. Play around. What happens if you click twice? Anything at all?

Now we can click on any of the images and see the unblurred version. How well did you do?



there are no Dumb Questions

Q: Does the onload event handler get passed an event object too?

A: It does, and it includes information like the target, which is the window object, the time it happened, and the type of the event, which is just the type "load". It's safe to say you don't typically see the event object used much in load handlers because there really isn't anything that is useful in it for this kind of event. You're going to find that sometimes the event object will be useful to you, and sometimes it won't, depending on the type of event. If you're unsure what the event object contains for a specific kind of event, just grab a JavaScript reference.

BRAIN POWER

What if you want to have an image become blurred again a few seconds after you've revealed the answer. How might that work?

Events Exposed

This week's interview:
Talking to the browser about events

Head First: Hey Browser, it's always good to have your time. We know how busy you are.

Browser: My pleasure, and you're right, managing all these events keeps me on my toes.

Head First: Just how do you manage them anyway? Give us a behind-the-scenes look at the magic.

Browser: As you know, events are almost continually happening. The user moves the mouse around, or makes a gesture on a mobile device; things arrive over the network; timers go off… it's like Grand Central. That's a lot to manage.

Head First: I would have assumed you don't need to do much unless there happens to be a handler defined somewhere for an event?

Browser: Even if there's no handler, there's still work to do. Someone has to grab the event, interpret it, and see if there is a handler waiting for it. If there is, I have to make sure that handler gets executed.

Head First: So how do you keep track of all these events? What if lots of events are happening at the same time? There's only one of you after all.

Browser: Well, yes, lots of events can happen over a very short amount of time, sometimes too fast for me to handle all in real time. So what I do is throw them all on a queue as they come in. Then I go through the queue and execute handlers where necessary.

Head First: Boy, that sounds like my days as a short-order cook!

Browser: Sure, if you had orders coming in every millisecond or so!

Head First: You have go through the queue one by one?

Browser: I sure do, and that's an important thing to know about JavaScript: there's one queue and one "thread of control," meaning there is only one of me going through the events one at a time.

Head First: What does that really mean for our readers learning JavaScript?

Browser: Well, say you write a handler and it requires a lot of computation—that is, something that takes a long time to compute. As long as your handler is chugging along computing, I'm sitting around waiting until it's done. Only then can I continue with the queue.

Head First: Oh wow. Does that happen a lot, that is, you end up waiting on slow code?

Browser: It happens, but it also doesn't take long for a web developer to figure out the page or app isn't responsive because the handlers are slow. So, it's not a common problem as long as the web developers know how event queues work.

Head First: And now all our readers do! Now, back to events, are there lots of different kinds of events?

Browser: There are. We've got network-based events, timer events, DOM events related to the page and a few others. Some kinds of events, like DOM events, generate event objects that contain a lot more detail about the event—like a mouse click event will have information about where the user clicked, and a keypress event will have information about which key was pressed, and so on.

Head First: So, you spend a lot of time dealing with events. Is that really the best use of your time? After all, you've got to deal with retrieving, parsing and rendering pages and all that.

Browser: Oh, it's very important. These days you've got to write code that makes your pages interactive and engaging, and for that you need events.

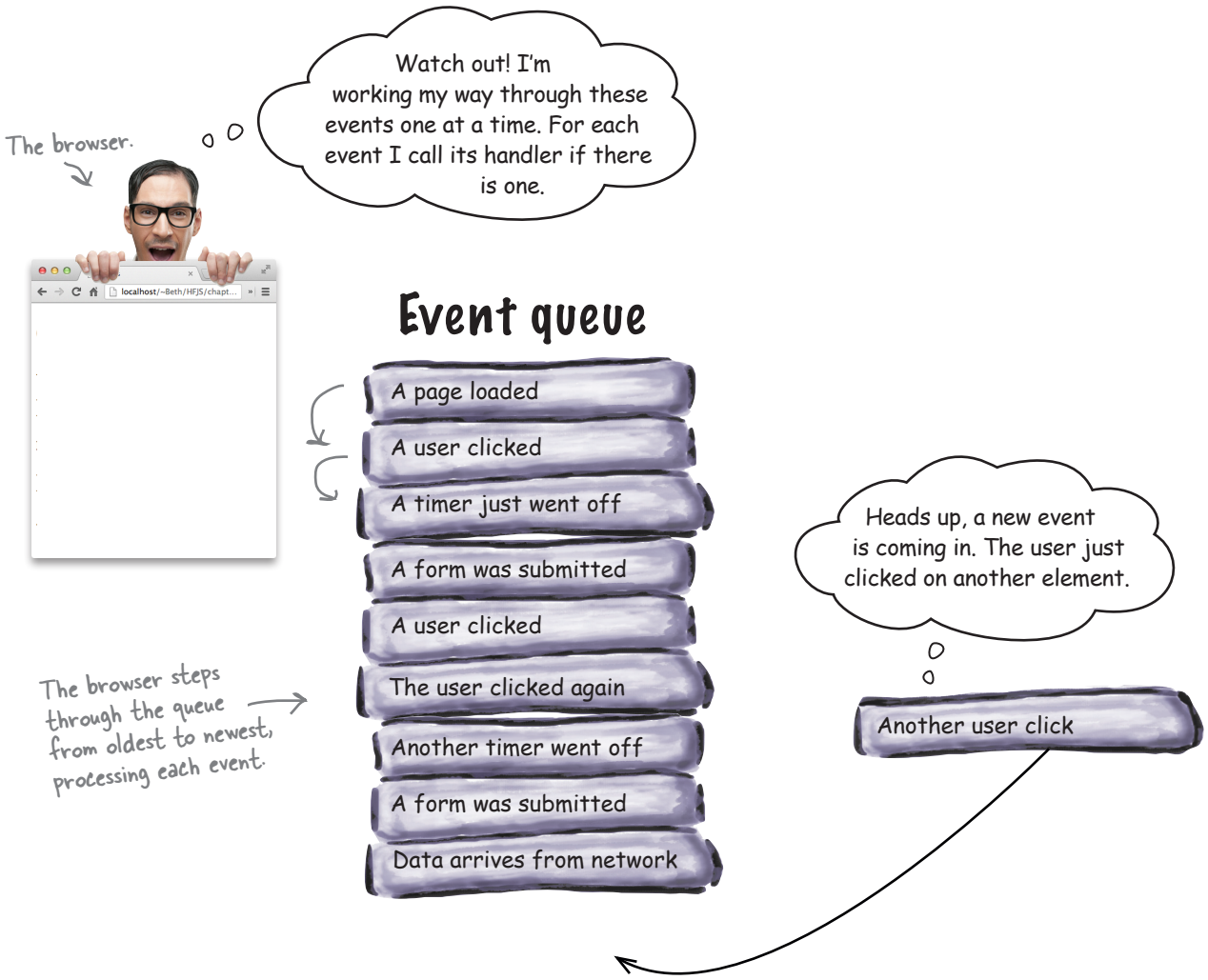
Head First: Oh for sure, the days of simple pages are gone.

Browser: Exactly. Oh shoot, this queue is about to overflow. Gotta run!

Head First: Okay… until next time!

Events and queues

You already know that the browser maintains a queue of events. And that behind the scenes the browser is constantly taking events off that queue and processing them by calling the appropriate event handler for them, if there is one.



It's important to know that the browser processes these events one at a time, so, where possible, you need to keep your handlers short and efficient. If you don't, the whole event queue could stack up with waiting events, and the browser will get backed up dealing with them all. The downside to you? Your interface could really start to become slow and unresponsive.

If things get really bad you'll get the slow script dialog box, which means the browser is giving up!

Ahoy matey! You've got a treasure map in your possession and we need your help in determining the coordinates of the treasure. To do that you're going to write a bit of code that displays the coordinates on the map as you pass the mouse over the map. We've got some of the code on the next page, but you'll have to help finish it.

Here's the map and X marks the spot!

After your code is written, just move the mouse over the X to see the coordinates of the treasure.

Your code will display the coordinates below the map.

COORDINATES: 10, 20

P.S. We highly encourage you to do this exercise, because we don't think the pirates are going to be too happy if they don't get their coordinates... Oh, and you'll need this to complete your code:

The mousemove event

The mousemove event notifies your handler when a mouse moves over a particular element. You set up your handler using the element's **onmousemove** property. Once you've done that you'll be passed an event object that provides these properties:

clientX, clientY: the x (and y) position in pixels of your mouse from the left side (and top) of the browser window.

screenX, screenY: the x (and y) position in pixels of your mouse from the left side (and top) of the user's screen.

pageX, pageY: the x (and y) position in pixels of your mouse from the left side (and top) of the browser's page.

Blimey! The code is below. So far it includes the map in the page and creates a paragraph element to display the coordinates. You need to make all the event code work. Good luck. We don't want to see you go to Davy Jones' locker anytime soon...

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Pirates Booty</title>
  <script>
       window.onload = init;
       function init() {
           var map = document.getElementById("map");
           ____________________________________________
       }

       function showCoords(eventObj) {
           var map = document.getElementById("coords");
           ____________________________
           ____________________________
           map.innerHTML = "Map coordinates: "
                                + x + ", " + y;
       }
  </script>
</head>
<body>
       <img id="map" src="map.jpg">
       <p id="coords">Move mouse to find coordinates...</p>
</body>
</html>
```

Set up your handler here.

Grab the coordinates here.

When you're done get this code in a real page, load it, and write your coordinates here.

_____

Even more events

So far we've seen three types of events: the load event, which occurs when the browser has loaded the page; the click event, which occurs when a user clicks on an element in the page and the mousemove event, which occurs when a user moves the mouse over an element. You're likely to run into many other kinds of events too, like events for data arriving over the network, events about the geolocation of your browser, and time-based events (just to name a few).

For all the events you've seen, to wire up a handler, you've always assigned the handler to some property, like `onload`, `onmouseover` or `onclick`. But not all events work like this—for example, with time-based events, rather than assigning a handler to a property, you call a function, `setTimeout`, instead and pass it your handler.

Here's an example: say you want your code to wait five seconds before doing something. Here's how you do that using `setTimeout` and a handler:

First we write an event handler. This is the handler that will be called when the time event has occurred.

```
function timerHandler() {
    alert("Hey what are you doing just sitting there staring at a blank screen?");
}
```

All we're doing in this event handler is showing an alert.

And here, we call setTimeout, which takes two arguments: the event handler and a time duration (in milliseconds).

```
setTimeout(timerHandler, 5000);
```

Using setTimeout is a bit like setting a stop watch.

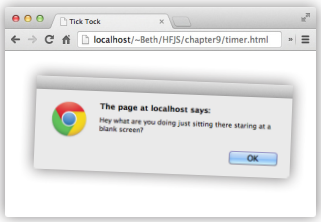
Here we're asking the timer to wait 5000 milliseconds (5 seconds).

And then call the handler timerHandler.

Test drive your timer

Don't just sit there! It's time to test this code! Throw this code into a basic HTML page, and load the page. At first you won't see anything, but after five seconds you'll see the alert.

Be patient, wait five seconds and you'll see what we see. Now if you've been sitting there a couple minutes you might want to give your machine a little kick... just kidding, you'd actually better check your code.



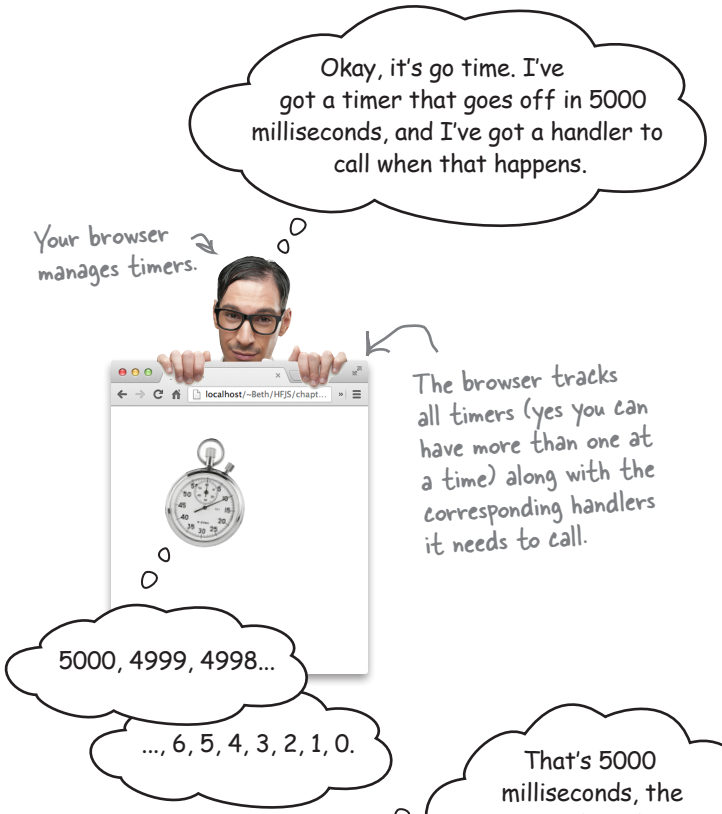
BRAIN POWER

How would you make the alert appear every 5 seconds, over and over?

How setTimeout works

Let's step through what just happened.

1. When the page loads we do two things: we define a handler named timerHandler, and we call setTimeout to create a time event that will be generated in 5000 milliseconds. When the time event happens, the handler will be executed.

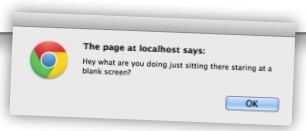


2. The browser continues its normal job as the timer counts down in milliseconds.

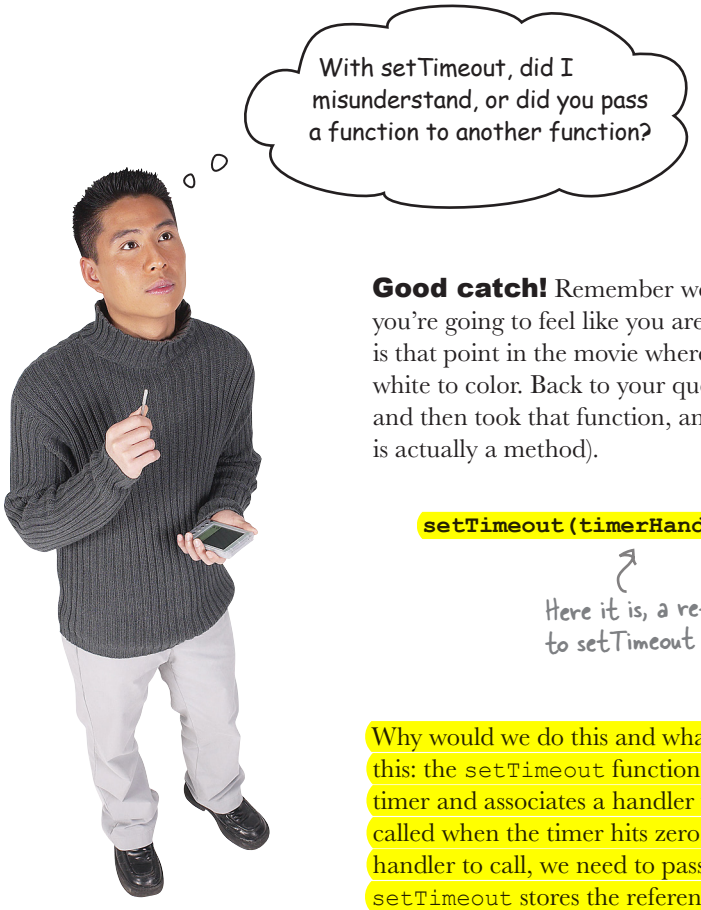
3. When the browser's countdown gets to zero, the browser calls the handler.

4. The handler is called, resulting in an alert being created and displayed in the browser.

```
function timerHandler() {
    alert("Hey what are you doing just sitting there staring at a blank screen?");
}
```



When the browser executes our event handler, we see the alert!



Good catch! Remember we said up front that in this chapter you're going to feel like you aren't in Kansas anymore? Well this is that point in the movie where everything goes from black and white to color. Back to your question; yes, we defined a function and then took that function, and passed it to `setTimeout` (which is actually a method).

```
setTimeout(timerHandler, 5000);
```

Here it is, a reference to a function passed to setTimeout (another function).

Why would we do this and what does it mean? Let's think through this: the `setTimeout` function essentially creates a countdown timer and associates a handler with that timer. That handler is called when the timer hits zero. Now to tell `setTimeout` what handler to call, we need to pass it a *reference to the handler function*. `setTimeout` stores the reference away to use later when the timer has expired.

If you're saying "That makes sense," then great. On the other hand, you might be saying "Excuse me? Pass a function to a function? Say what?" In that case, you probably have experience with a language like C or Java, where *you don't just go around passing functions to other functions like this*… well, in JavaScript, you do, and in fact, being able to pass functions around is incredibly powerful, especially when we're writing code that reacts to events.

More likely at this point you're saying, "I think I sort of get it, but I'm not sure." If so, no worries. For now, just think of this as giving `setTimeout` a reference to the handler it's going to need to invoke when the timer expires. We're going to be talking a lot more about functions and what you can do with them (like passing them to other functions) in the next chapter. So just go with it for now.

Exercise

Here's the code.

```
var tick = true;
function ticker() {
    if (tick) {
        console.log("Tick");
        tick = false;
    } else {
        console.log("Tock");
        tick = true;
    }
}
setInterval(ticker, 1000);
```

Your analysis goes here.

Take a look at the code below and see if you can figure out what `setInterval` does. It's similar to `setTimeout`, but with a slight twist. Check your answer at the end of the chapter.

JavaScript console

```
Tick
Tock
Tick
Tock
Tick
Tock
Tick
Tock
```

Here's the output.

there are no Dumb Questions

Q: Is there a way to stop setInterval?

A: There is. When you call setInterval, it returns a timer object. You can pass that timer object to another function, clearInterval, to stop the timer.

Q: You said setTimeout was a method, but it looks like a function. Where's the object it's a method of?

A: Good catch. Technically we could write window.setTimeout, but because the window object is considered the global object, we can omit the object name, and just use setTimeout, which we'll see a lot in practice.

Q: Can I omit window on the window.onload property too?

A: You can, but most people don't because they are worried onload is a common enough property name (other elements can have the onload property too) that not specifying which onload property might be confusing.

Q: With onload I'm assigning one handler to an event. But with setTimeout, I seem to be able to assign as many handlers as I want to as many timers as I want?

A: Exactly. When you call setTimeout, you are creating a timer and associating a handler with it. You can create as many timers as you like. The browser keeps track of associating each timer with its handler.

Q: Are there other examples of passing functions to functions?

A: Lots of them. In fact, you'll find that passing around functions is fairly common in JavaScript. Not only do lots of built-in functions, like setTimeout and setInterval, make use of function passing, but you'll also discover there's a lot code you'll write yourself that accepts functions as arguments. But that's only part of the story, and in the next chapter we're going to dive deep into this topic and discover that you can do all sort of interesting things with functions in JavaScript.

Joe: That sounds cool… I bet you're using `setTimeout`?

Frank: That's my plan, although I'm not sure how to know which image to reblur.

Jim: What do you mean?

Frank: I have my code so that when you click on an image and reveal it, I set up a timer that fires in two seconds. When the timeout event fires, it calls a new handler I wrote called `reblur`.

Joe: And in `reblur`, you need to know which image to reblur?

Frank: Right. I'm not passing any arguments to the handler, it's just being called by the browser when the time expires, and so I have no way to tell my handler the correct image to reblur. I'm kinda stuck.

Jim: Have you looked at the `setTimeout` API?

Frank: No, I know only what Judy told me: that `setTimeout` takes a function and a time duration in milliseconds.

Jim: You can add an argument to the call to `setTimeout` that is passed on to the handler when the time event fires.

Frank: Oh that's perfect. So I can just pass in a reference to the correct image to reblur and that will get passed on to the handler when it is called?

Jim: You got it.

Frank: See what a little talking through code gets ya Joe?

Joe: Oh for sure. Let's give this a try…

Finishing the image game

Now it's time to put the final polish on the image game. What we want is for an image to automatically reblur a few seconds after it's revealed. And, as we just learned, we can pass along an argument for the event handler when we call `setTimeout`. Let's check out how to do this:

```
window.onload = function() {
    var images = document.getElementsByTagName("img");
    for (var i = 0; i < images.length; i++) {
        images[i].onclick = showAnswer;
    }
};

function showAnswer(eventObj) {
    var image = eventObj.target;
    var name = image.id;
    name = name + ".jpg";
    image.src = name;

    setTimeout(reblur, 2000, image);
}

function reblur(image) {
    var name = image.id;
    name = name + "blur.jpg";
    image.src = name;
}
```

This code is just as we wrote it before. No changes here...

But now when we show the user the clear image, we also call setTimeout to set up an event that will fire in two seconds.

We'll use reblur (below) as our handler, and pass it 2000 milliseconds (two seconds) and also an argument, the image to reblur.

Now when this handler is called, it will be passed the image.

The handler can take the image, get the id of the image, and use that to create the name of the blurred image. When we set the src of the image to that name, it will replace the clear image with the blurred image.

Watch it!

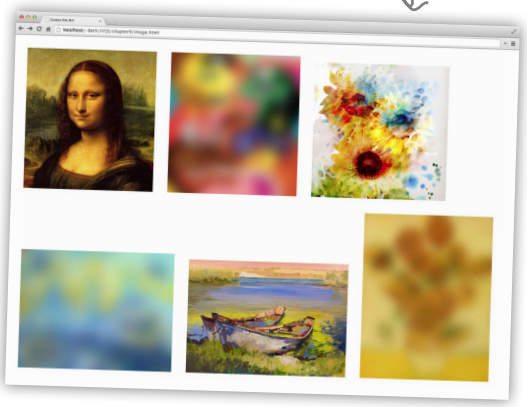
setTimeout does not support extra arguments in IE8 and earlier.

That's right. This code is not going to work for you or your users if you're using IE8 or earlier. But you shouldn't be using IE8 for this book anyway! That said, you'll see another way to do this a little later in the book that will take care of this for IE8 (and earlier).

Test driving the timer

That wasn't much code to add, but it sure makes a big difference in how the image game works. Now when you click, behind the scenes, the browser (through the timer events) is tracking when it needs to call the `reblur` handler, which blurs the image again. Note how *asynchronous* this feels—you're in control of when the images are clicked, but behind-the-scenes code is being invoked at various times based on the click event and on timer events. There's no über algorithm driving things here, controlling what gets called and when; it's just a lot of little pieces of code that set up, create and react to events.

Now when you click, you'll see the image revealed, and then blurred again two seconds later.



Give this a good QA testing by clicking on lots of images in quick succession. Does it always work? Refer back to the code and wrap your brain around how the browser keeps track of all the images that need to be reblurred.

there are no Dumb Questions

Q: Can I pass just one argument to the setTimeout handler?

A: No you can actually pass as many as you like: zero, one or more.

Q: What about the event object? Why doesn't setTimeout pass the event handler one?

A: The event object is mostly used with DOM-related event handlers. setTimeout doesn't pass any kind of event object to its handler, because it doesn't occur on a specific element.

Q: showAnswer is a handler, and yet it creates a new handler, reblur, in its code. Is that right?

A: You've got it. You'll actually see this fairly often in JavaScript. It's perfectly normal to see a handler set up additional event handlers for various events. And this is the style of programming we were referring to in the beginning of the chapter: *asynchronous programming*. To create the image game we didn't just translate an algorithm that runs top down. Rather we're hooking up event handlers to handle the execution of the game as events occur. Trace through a few different examples of clicking on images and the various calls that get made to reveal and reblur the image.

Q: So there are DOM-based events, and timer events... are there lots of different kinds of events?

A: Many of the events you deal with in JavaScript are DOM events (like when you click on an element), or timer events (created with setTmeout or setInterval). There are also API-specific events, like events generated by JavaScript APIs including Geolocation, LocalStorage, Web Workers, and so on (see *Head First HTML5 Programming* for more on these). And finally, there is a whole category of events related to I/O: like when you request data from a web service using XmlHttpRequest (again, see *Head First HTML5 Programming* for more), or Web Sockets.

Hey guys, our desktop users would like to be able to pass their mouse over the image without clicking to reveal the unblurred image. Can we get that implemented?

Judy: To make this work you'll want to make use of the mouseover event. You can set a handler for this event on any element with the `onmouseover` property:

```
myElement.onmouseover = myHandler;
```

Judy: Also, the mouseout event tells you when the mouse leaves your element. You can set its handler with the `onmouseout` property.

Exercise

Rework your code so that an image is revealed and reblurred by moving your mouse over and out of the image elements. Be sure to test your code, and check your answer at the end of the chapter:

JavaScript code goes here.

Exercise

With the image game complete, Judy wrote some code to review in the weekly team meeting. In fact, she started a little contest, awarding the first person to describe what the code does with lunch. Who wins? Jim, Joe, Frank? Or you?

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Don't resize me, I'm ticklish!</title>
<script>
    function resize() {
        var element = document.getElementById("display");
        element.innerHTML = element.innerHTML + " that tickles!";
    }
</script>
</head>
<body>
<p id="display">
    Whatever you do, don't resize this window! I'm warning you!
</p>
<script>
    window.onresize = resize;
</script>
</body>
</html>
```

Put your notes here describing what this code does. What events are involved? How is the handler set up? And when does the event happen? Don't just make notes, give this a try in your browser.

CODE LABORATORY

We've found some highly suspicious code we need your help testing. While we've already done an initial analysis on the code and it looks like 100% standard JavaScript, something about it looks odd. Below you'll find two code specimens. For each specimen you'll need to identify what seems odd about the code, test to make sure the code works, and then try to analyze what exactly it does. Go ahead and make your notes on this page. You'll find our analysis on the next page.

Specimen #1

```
var addOne = function(x) {
                return x + 1;
              };
var six = addOne(5);
```

Specimen #2

```
window.onload = function() {
    alert("The page is loaded!");
}
```

CODE LABORATORY: ANALYSIS

Specimen #1

```
var addOne = function(x) {
                    return x + 1;
                  };
var six = addOne(5);
```

At first glance this code appears to simply define a function that adds the number one to any parameter and return it.

Looking closer, this isn't a normal function definition. Rather, we are declaring a variable and assigning to it a function that appears to be missing its name.

Further, we're invoking the function with the variable name, not a name associated with the function as part of its definition.

Odd indeed (although it reminds us a bit of how object methods are defined).

Specimen #2

```
window.onload = function() {
    alert("The page is loaded!");
}
```

Here we appear to have something similar. Instead of defining a function separately and assigning its name to the window.onload property, we're assigning a function directly to that property. And again, the function doesn't define its own name.

We added this code to an HTML page and tested it. The code appears to work as you might expect. With specimen #1, when the function assigned to addOne is invoked, we get a result that is one greater than the number we pass in, which seems right. With specimen #2, when we load the page, we get the alert "The page is loaded!".

From these tests it would appear as if functions can be defined without names, and used in places where you'd expect an expression.

BULLET POINTS

- Most JavaScript code is written to react to **events**.
- There are many different kinds of events your code can react to.
- To react to an event, you write an **event handler** function, and register it. For instance, to register a handler for the click event, you assign the handler function to the onclick property of an element.
- You're not required to handle any specific event. You choose to handle the events you're interested in.
- **Functions** are used for handlers because functions allow us to package up code to be executed later (when the event occurs).
- Code written to handle events is different from code that executes top to bottom and then completes. Event handlers can run at any time and in any order: they are **asynchronous**.
- Events that occur on elements in the DOM (DOM events) cause an event object to be passed to the event handler.
- The **event object** contains properties with extra information about the event, including the type (like "click" or "load") and the target (the object on which the event occurred).
- Older versions of IE (IE 8 and older) have a different event model from other browsers. See the appendix for more details.
- Many events can happen very close together. When too many events happen for the browser to handle them as they occur, the events are stored in an **event queue** (in the order in which they occurred) so the browser can execute the event handlers for each event in turn.
- If an event handler is computationally complex, it will slow down the handling of the events in the queue because only one event handler can execute at a time.
- The functions **setTimeout** and **setInterval** are used to generate time-based events after a certain time has passed.
- The method **getElementsByTagName** returns zero, one or more element objects in a NodeList (which is array-like, so you can iterate over it).

Event Soup

click
Get this event when you click (or tap) in a web page.

load
The event you get when the browser has completed loading a web page.

mousemove
When you move your mouse over an element, you'll generate this event.

keypress
This event is generated every time you press a key.

unload
This event is generated when you close the browser window, or navigate away from a web page.

mouseover
When you put your mouse over an element, you'll generate this event.

mouseout
And you'll generate this event when you move your mouse off an element.

resize
Whenever you resize your browser window, this event is generated.

dragstart
If you drag an element in the page, you'll generate this event.

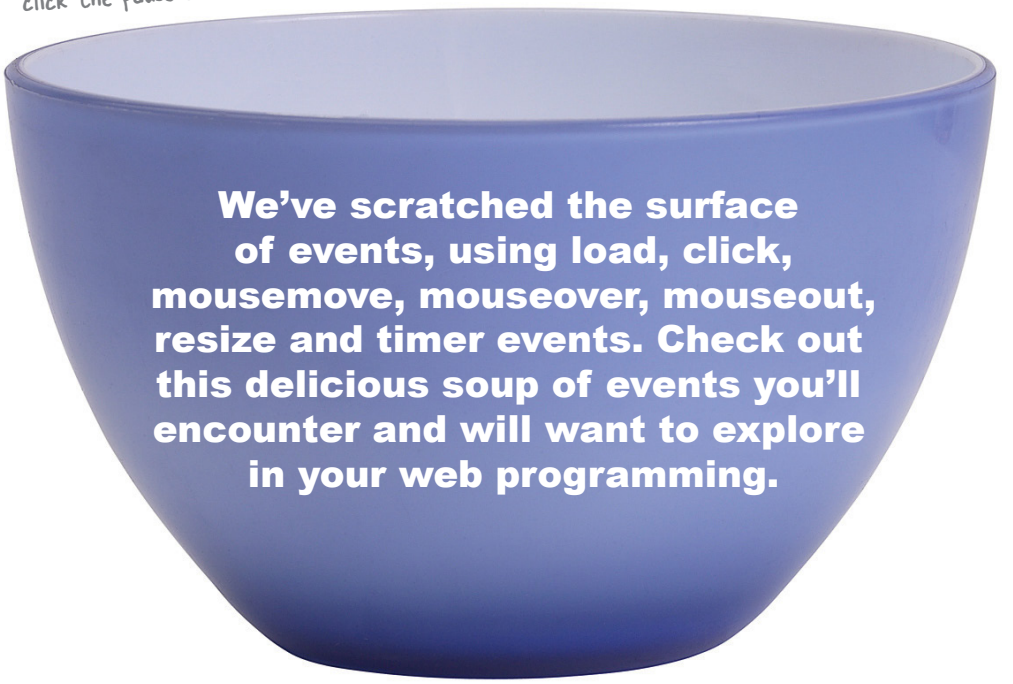
touchstart
On touch devices, you'll generate a touchstart event when you touch and hold an element.

play
Got <video> in your page? You'll get this event when you click the play button.

drop
You'll get this event when you drop an element you've been dragging.

pause
And this one when you click the pause button.

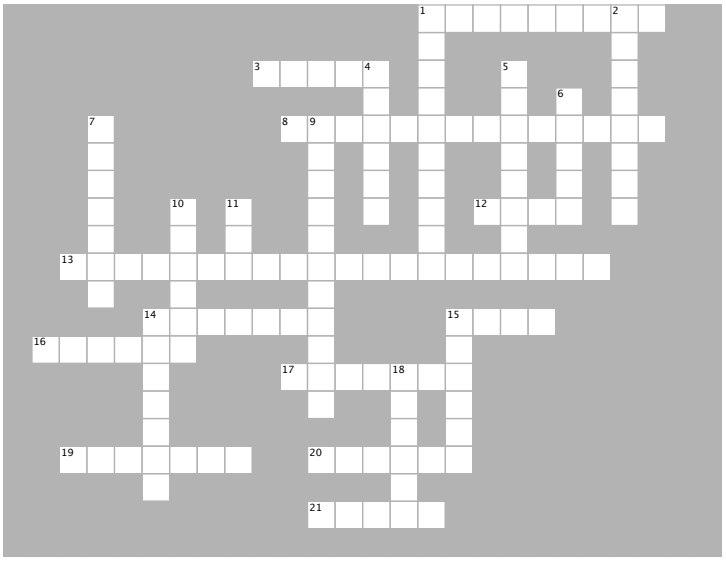
touchend
And you'll get this event when you stop touching.



We've scratched the surface of events, using load, click, mousemove, mouseover, mouseout, resize and timer events. Check out this delicious soup of events you'll encounter and will want to explore in your web programming.

JavaScript cross

Practice your event reaction time by doing this crossword.



ACROSS

1. Use this property of the event object to know when an event happened.
3. When you click your mouse, you'll generate a _____ event.
8. Events are handled _____.
12. 5000 milliseconds is _____ seconds.
13. Use this method to get multiple elements from the DOM using a tag name.
14. A function designed to react to an event is called an event _____.
15. The setTimeout method is used to create a _____ event.
16. The browser has only one _____ of control.
17. The browser can only execute one event _____ at a time.
19. The event object for a mouseover event has this property for the X position of the mouse.
20. The event _____ is passed to an event handler for DOM events.
21. To pass an argument to a time event handler, pass it as the _____ argument to setTimeout.

DOWN

1. You'll generate this event if you touch your touch screen device.
2. zero.jpg is the _____.
4. When you begin programming with events, you might feel like you're not in _____ any more.
5. JavaScript allows you to pass a _____ to a function.
6. If too many events happen close together, the browser stores the events in an event _____.
7. Events are generated for lots of things, but not for baking _____.
9. To make a time event happen over and over, use _____.
10. The window _____ property is for handling a page loaded event.
11. _____ is super ticklish.
14. To assign an event handler for a time event, pass the _____ to setTimeout as the first argument.
15. How you know which image was clicked on in the image game.
18. A program with code to handle events is not this.

Sharpen your pencil Solution

Pick two of the events above. If the browser could notify your code when these events occurred, what cool or interesting code might you write?

Let's take an event that notifies us when a user submits a form. If we're notified of this event, then we could get all the data the user filled into the form and check to make sure it's valid (e.g. the user put something that looks like a phone number into a phone number field, or filled out the required fields). Once we've done that check, then we could submit the form to the server.

How about the mouse movement event? If we're notified whenever a user moves the mouse, then we could create a drawing application right in the browser.

If we're notified when the user scrolls down the page, we could do interesting things like reveal an image as they scroll down.

BE the Browser Solution

Below, you'll find the image game code. Your job is to play like you're the browser and to figure out what you need to do after each event. After you've done the exercise look at the end of the chapter to see if you got everything. Here's our solution.

```
window.onload = init;
function init() {
    var image = document.getElementById("zero");
    image.onclick = showAnswer;
}

function showAnswer() {
    var image = document.getElementById("zero");
    image.src = "zero.jpg";
}
```

When page is being loaded...

- First define the functions init and showAnswer
- Set load handler to init

When page load event occurs...

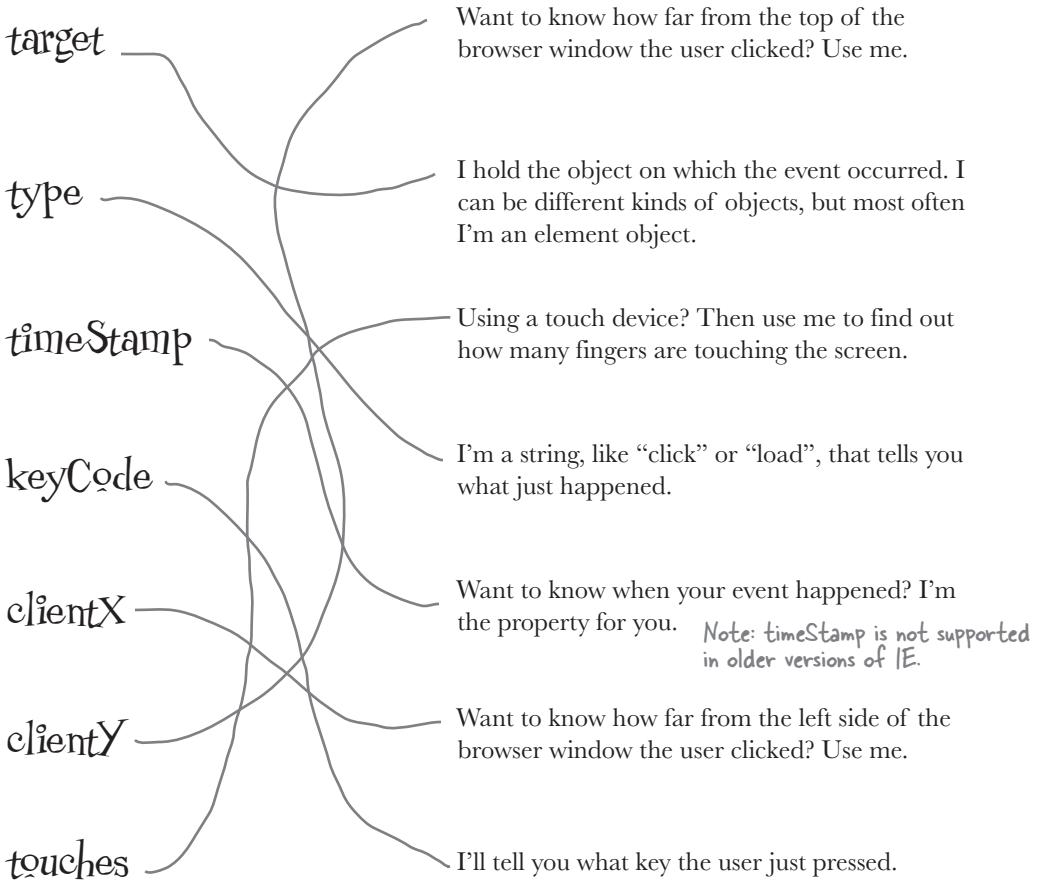
- load handler, init, is called
- we get the image with id "zero"
- set image's click handler to showAnswer

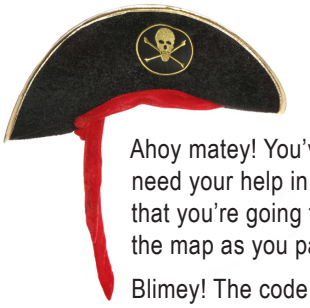
When image click event occurs...

- showAnswer is called
- we get the image with id "zero"
- we set the src attribute to "zero.jpg"

WHO DOES WHAT? SOLUTION

You've already seen that the event object (for DOM events) has properties that give you more information about the event that just happened. Below you'll find other properties that the event object can have. Match each event object property to what it does.





Ahoy matey! You've got a treasure map in your possession and we need your help in determining the coordinates of the treasure. To do that you're going to write a bit of code that displays the coordinates on the map as you pass the mouse over the map.

Blimey! The code is below. So far it includes the map in the page and creates a paragraph element to display the coordinates. You need to make all the event-based code works. Good luck. We don't want to see you go to Davy Jones' locker anytime soon... And here's our solution.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Pirates Booty</title>
  <script>
       window.onload = init;
       function init() {
           var map = document.getElementById("map");
           map.onmousemove = showCoords;
       }

       function showCoords(eventObj) {
           var map = document.getElementById("coords");
           var x = eventObj.clientX;
           var y = eventObj.clientY;
           map.innerHTML = "Map coordinates: "
                                + x + ", " + y;
       }
  </script>
</head>
<body>
       <img id="map" src="map.jpg">
       <p id="coords">Move mouse to find coordinates...</p>
</body>
</html>
```

When we put our mouse right over the X, we got the coordinates: 200, 190

Exercise Solution

Take a look at the code below and see if you can figure out what `setInterval` does. It's similar to `setTimeout`, but with a slight twist.

Here's our solution:

Here's the code.

```
var tick = true;

function ticker() {
    if (tick) {
        console.log("Tick");
        tick = false;
    } else {
        console.log("Tock");
        tick = true;
    }
}

setInterval(ticker, 1000);
```

JavaScript console

```
Tick
Tock
Tick
Tock
Tick
Tock
Tick
Tock
```

Here's the output.

Your analysis goes here.

Just like setTimeout, setInterval also takes an event handler function as its first argument and a time duration as its second argument.

But unlike setTimeout, setInterval executes the event handler multiple times... in fact it keeps going. Forever! (Actually, you can tell it to stop, see below). In this example, every 1000 milliseconds (1 second), setInterval calls the ticker handler. The ticker handler is checking the value of the tick variable to determine whether to display "Tick" or "Tock" in the console.

So setInterval generates an event when the timer expires, and then restarts the timer.

```
var t = setInterval(ticker, 1000);

clearInterval(t);
```

To stop an interval timer, save the result of calling setInterval in a variable...

...and then pass that to clearInterval later, when you want to stop the timer.

Rework your code so that you can reveal and reblur an image by passing your mouse over and out of the image elements. Be sure to test your code. Here's our solution:

```
window.onload = function() {
    var images = document.getElementsByTagName("img");
    for (var i = 0; i < images.length; i++) {
        images[i].onclick = showAnswer;
        images[i].onmouseover = showAnswer;
        images[i].onmouseout = reblur;
    }
};
function showAnswer(eventObj) {
    var image = eventObj.target;
    var name = image.id;
    name = name + ".jpg";
    image.src = name;

    setTimeout(reblur, 2000, image);
}

function reblur(eventObj) {
    var image = eventObj.target;
    var name = image.id;
    name = name + "blur.jpg";
    image.src = name;
}
```

First, we remove the assignment of the event handler to the onclick property.

Then we add the showAnswer event handler to the onmouseover property of the image...

And now we're going to use reblur as the handler for the mouseout event (instead of as a timer event handler). So we assign reblur to the onmouseout property of the image.

We won't use the timer anymore to reblur the image; instead, we'll reblur it when the user moves the mouse out of the image element.

Now we're using reblur as an event handler for the mouseout event, so to get the correct image to reblur, we have to use the event object. Just like in showAnswer, we'll use the target property to get the image object. Once we have that, the rest of reblur is the same.

Exercise Solution

With the image game complete, Judy wrote some code to review in the weekly team meeting. In fact, she started a little contest, awarding the first person to describe what the code does with lunch. Who wins? Jim, Joe, Frank? Or you?

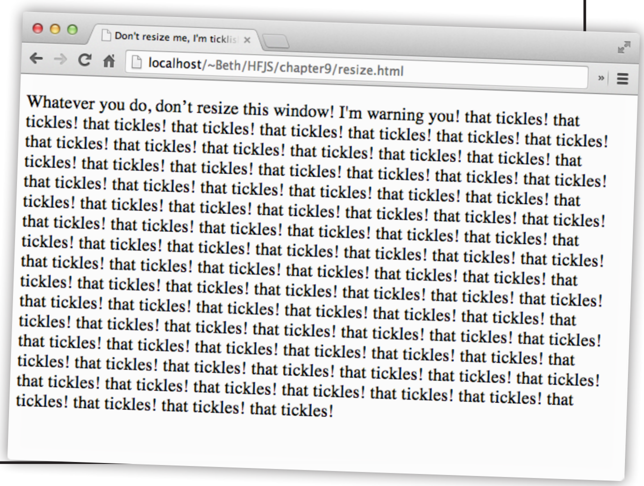
```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Don't resize me, I'm ticklish!</title>
<script>
    function resize() {
        var element = document.getElementById("display");
        element.innerHTML = element.innerHTML + " that tickles!";
    }
</script>
</head>
<body>
<p id="display">
    Whatever you do, don't resize this window! I'm warning you!
</p>
<script>
    window.onresize = resize;
</script>
</body>
</html>
```

Our event handler is named resize. When it's called, it just adds some text to the paragraph with the id "display".

The event we're interested in is the resize event, so we set up a handler function (named resize), and assign it to the onresize property of the window.

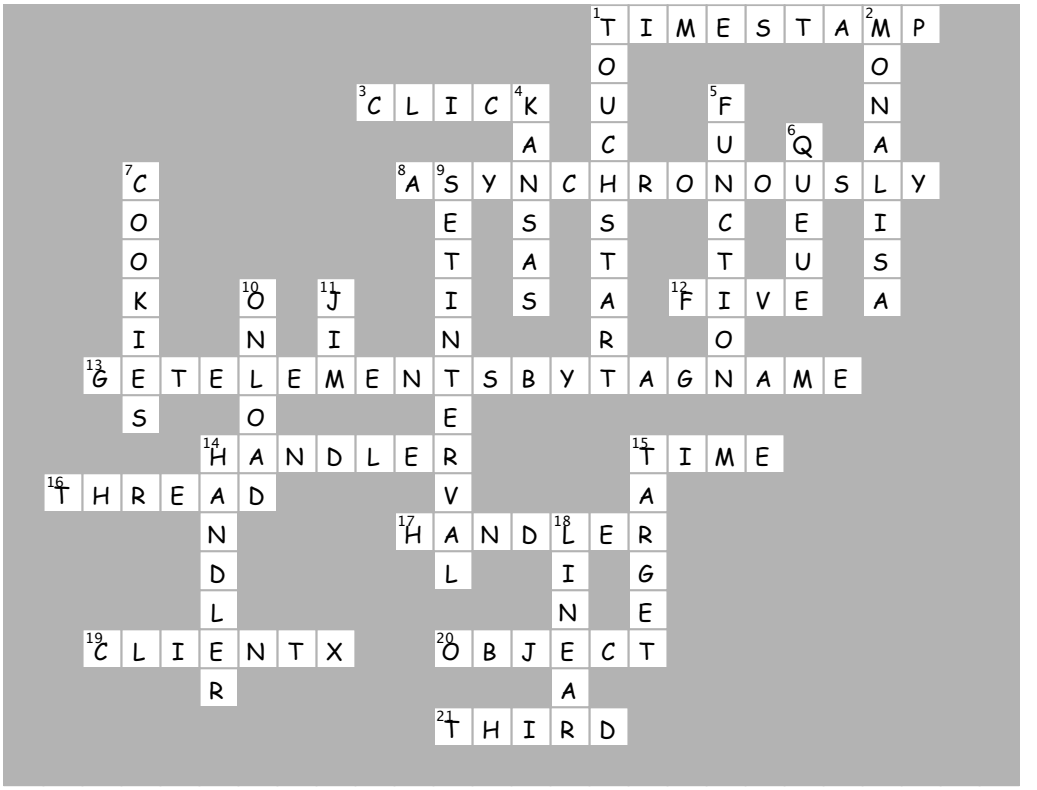
We set up the resize event in the script at the bottom of the page. Remember, this script won't run until the page is fully loaded, so that makes sure we don't set up the event handler too early.

When you resize the browser window, the resize event handler is called, which updates the page by adding new text content (" that tickles") to the "display" paragraph.



JavaScript cross Solution

Practice your event reaction time by doing this crossword. Here's our solution.



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.

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!");
   }
}
```

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

```
quack(3);
```

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);
```

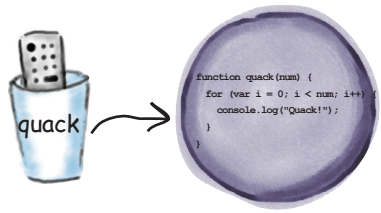
This doesn't look so standard: the function doesn't have a name, and it's on the right hand side of an assignment to a variable.

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

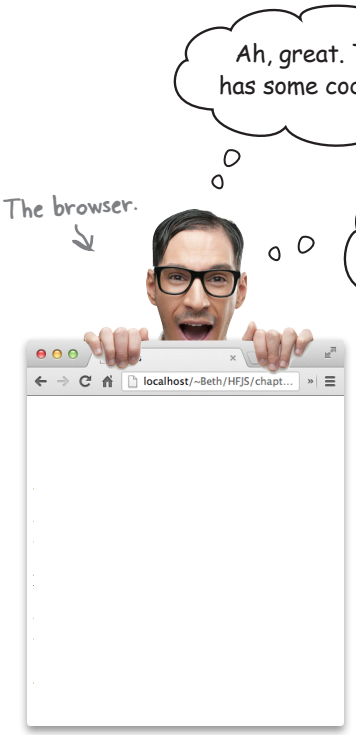
A function reference is exactly what it sounds like: a reference that refers to a function. You can use a function reference to invoke a function or, as you'll see, you can assign them to variables, store them in objects, and pass them to or return them from functions (just like object references).



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:



This variable declaration isn't a function declaration, so let's keep moving and ignore it for now.

```
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);
}
```

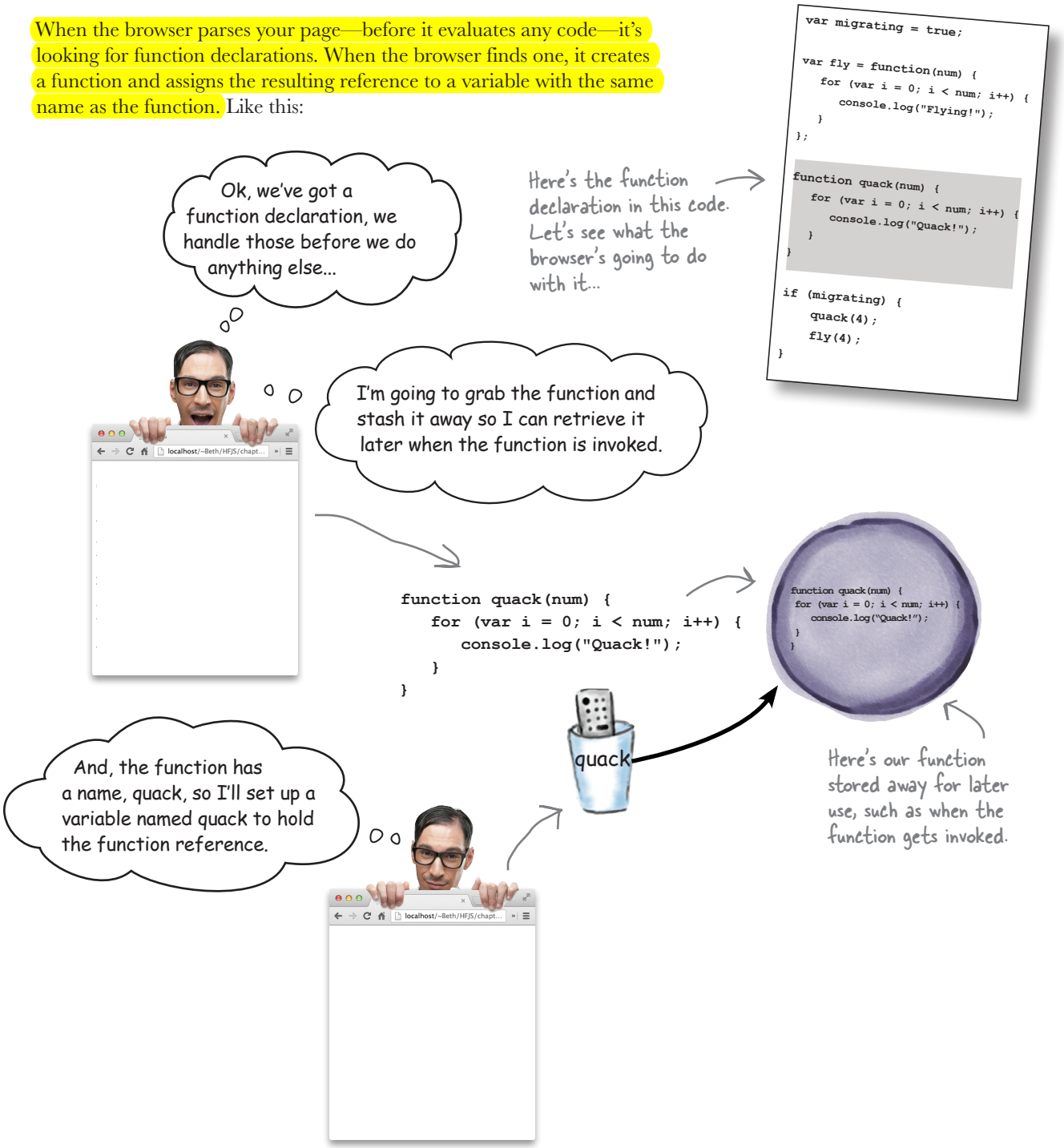
Next we have a function expression in a statement. That's not a declaration... let's keep moving.

Ah, here's a function declaration. We need to deal with that. Let's do that on the next page....

After we deal with the declaration above, we also won't care about this code because there aren't any function declarations here.

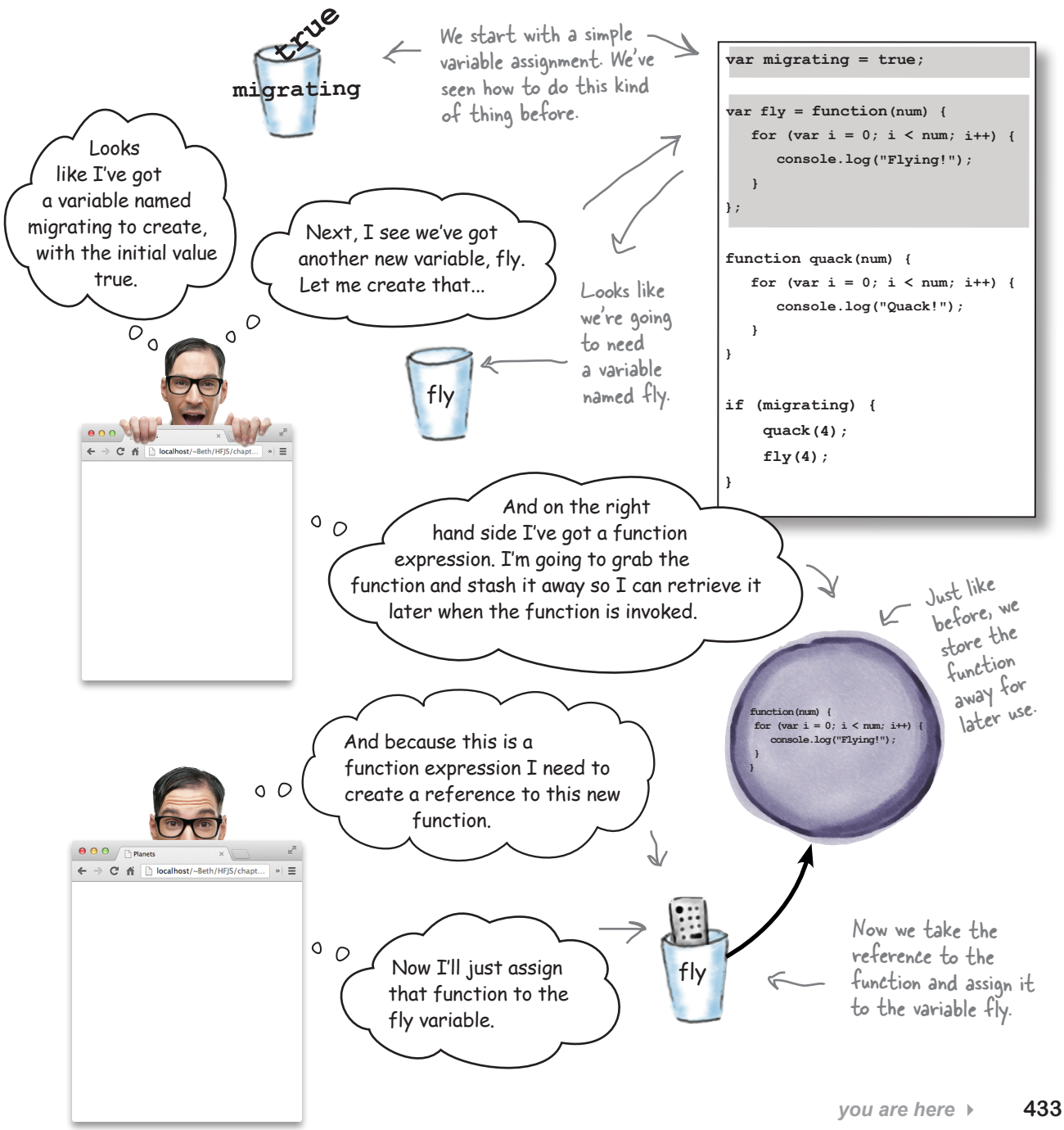
Parsing the function declaration

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

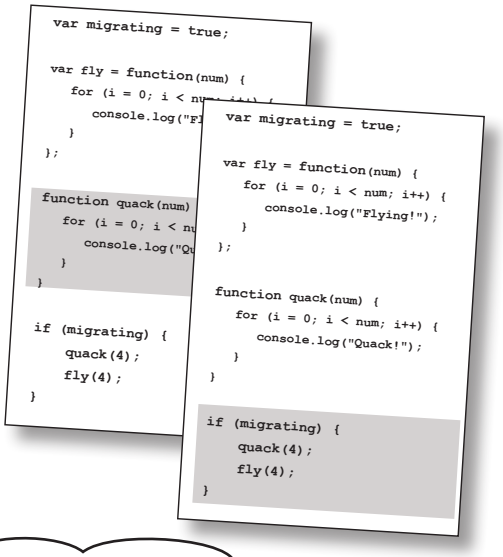
Now that all the function declarations have been taken care of, the browser goes back up to the top of your code and starts executing it, top to bottom.
Let's check in on the browser at that point in the execution:



Moving on... The conditional

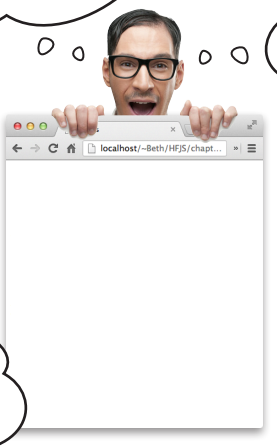
Once the `fly` variable has been taken care of, the browser moves on. The next statement is the function declaration for `quack`, which was dealt with in the first pass through the code, so the browser skips the declaration and moves on to the conditional statement. Let's follow along…

Been there, done that, moving on...



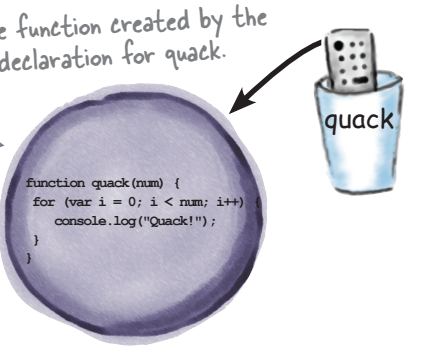
Let's see, the variable migrating is true, so I need to execute the body of the if statement. Inside it looks like there's a call to quack. I know it's a function call because we're using the function name, quack, followed by parentheses.

```
quack(4);
```

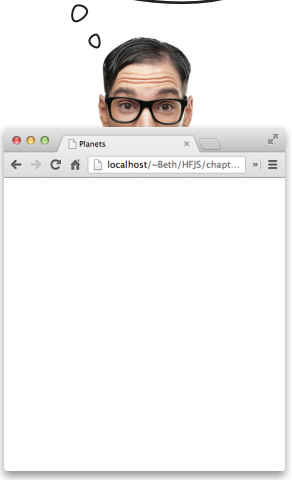


Remember, the quack variable is a reference to the function I stashed away earlier...

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

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

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

...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.

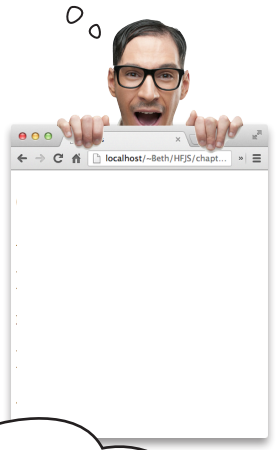
And finishing up...

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);
```

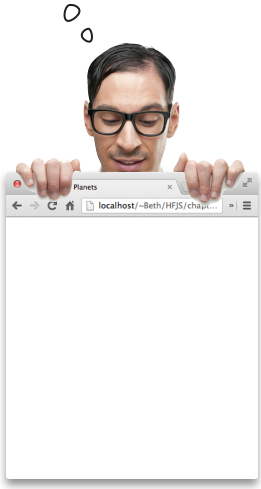


Remember the fly variable is a reference to the function I stashed away earlier...

Here's the function referenced by the fly variable.

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

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.

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.

there are no Dumb Questions

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();
```

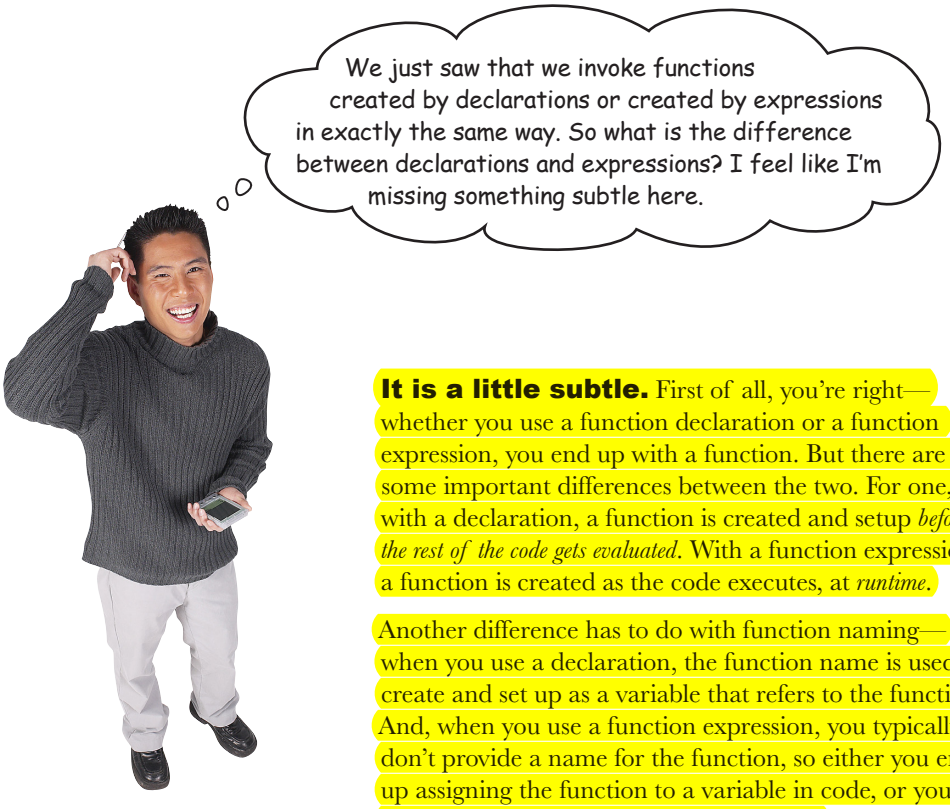
But you can also pass a reference to a function or return a reference from a function. But, we're getting a little ahead of ourselves. We'll come back to this in a few pages.

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.

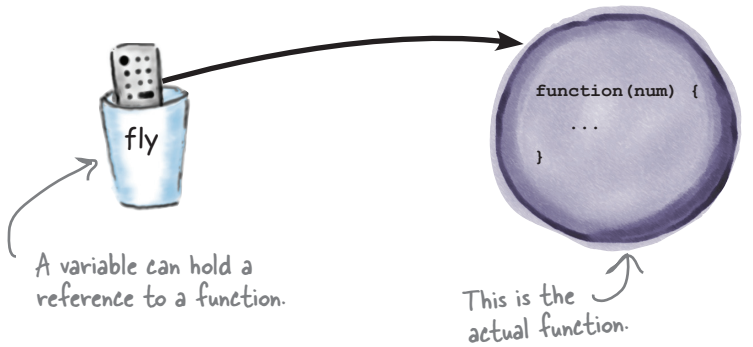
```
var midi = true;
var type = "piano";
var midiInterface;

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).

Exercise

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.

Sharpen your pencil

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.

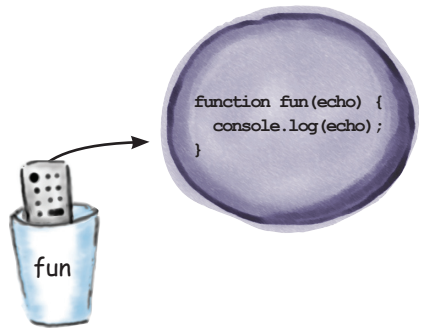
Q: What exactly are we passing when we pass a function to a function?

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:

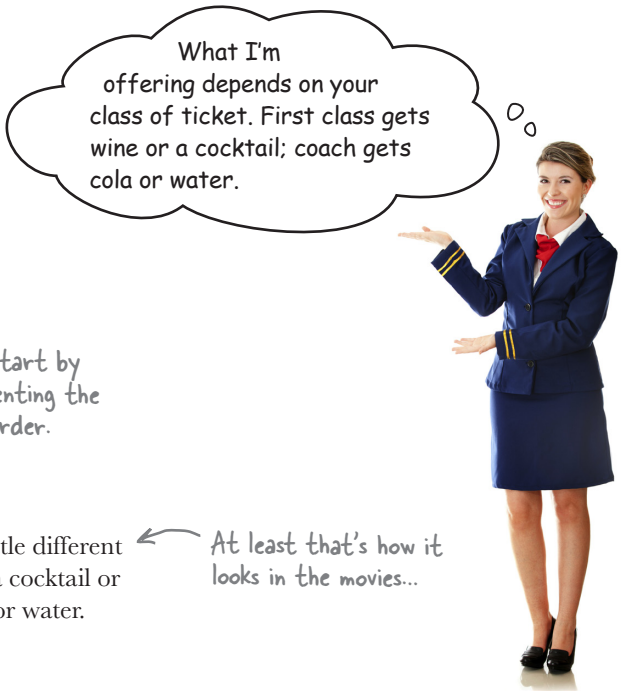
```
var passengers = [  { name: "Jane Doloop", paid: true, ticket: "coach" },
                    { name: "Dr. Evel", paid: true, ticket: "firstclass" },
                    { name: "Sue Property", paid: false, ticket: "firstclass" },
                    { name: "John Funcall", paid: true, ticket: "coach"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

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

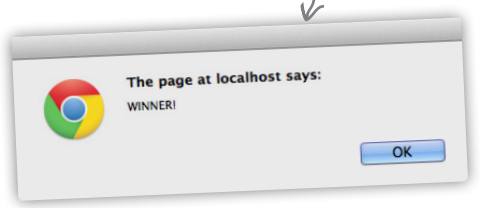
To get the idea of functions as values into your brain, let's play a little game of chance. Try the shell game. Did you win or lose? Give it a try and find out. Here's our solution.

```
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!!!



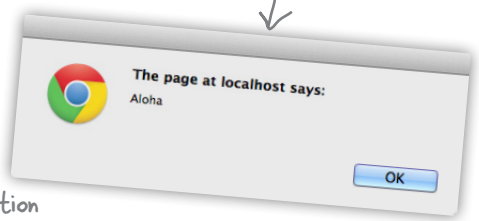
Sharpen your pencil Solution

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.

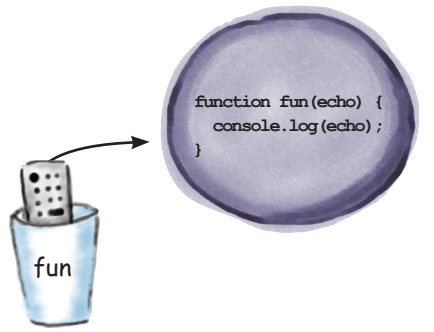
JavaScript console

```
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;
}
```

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.

```
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 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);
```

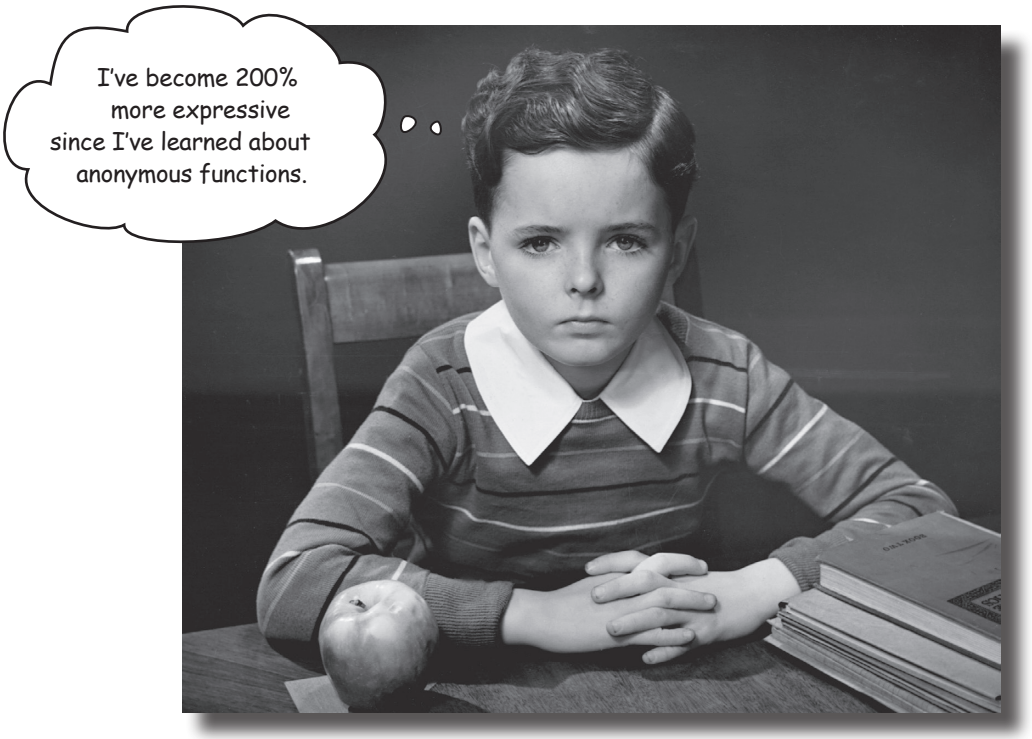
For each new compare function, we call sort, and display the results in the console.

Totally!

You guys nailed it!

11 anonymous functions, scope and closures

Serious functions



You've put functions through their paces, but there's more to learn. In this chapter we take it further; we get hard-core. We're going to show you how to **really handle** functions. This won't be a super long chapter, but it will be intense, and at the end you're going to be more expressive with your JavaScript than you thought possible. You're also going to be ready to take on a coworker's code, or jump into an open source JavasScript library, because we're going to cover some common coding idioms and conventions around functions. And if you've never heard of an **anonymous function** or a **closure**, boy are you in the right place.

And if you have heard of a closure, but don't quite know what it is, you're even more in the right place!

Taking a look at the other side of functions...

You've already seen two sides of functions—you've seen the formal, declarative side of function declarations, and you've seen the looser, more expressive side of function expressions. Well, now it's time to introduce you to another interesting side of functions: *the anonymous side.*

By anonymous we're referring to functions *that don't have names.* How can that happen? Well, when you define a function with a function declaration, your function will *definitely have a name*. But when you define a function using a function expression, *you don't have to give that function a name*.

You're probably saying, sure, that's an interesting fact, maybe it's possible, but so what? By using anonymous functions we can often make our code less verbose, more concise, more readable, more efficient, and even more maintainable.

So let's see how to create and use anonymous functions. We'll start with a piece of code we've seen before, and see how an anonymous function might help out:

Here's a load handler, set up like we've always done in the past.

First we define a function. This function has a name, handler.

```
function handler() { alert("Yeah, that page loaded!"); }
window.onload = handler;
```

Then we assign the function to the onload property of the window object, using its name, handler.

And when the page loads, the handler function is invoked.

Sharpen your pencil

Use your knowledge of functions and variables and check off the true statements below.

- ❑ The handler variable holds a function reference.
- ❑ When we assign handler to window.onload, we're assigning it a function reference.
- ❑ The only reason the handler variable exists is to assign it to window.onload.
- ❑ We'll never use handler again as it's code that is meant to run only when the page first loads.
- ❑ Invoking onload handlers twice is not a great idea—doing so could cause issues given these handlers usually do some initialization for the entire page.
- ❑ Function expressions create function references.
- ❑ Did we mention that when we assign handler to window.onload, we're assigning it a function reference?

How to use an anonymous function

So, we're creating a function to handle the load event, but we know it's a "one time" function because the load event only happens once per page load. We can also observe that the `window.onload` property is being assigned a function reference—namely, the function reference in `handler`. But because `handler` is a one time function, that name is a bit of a waste, because all we do is assign the reference in it to the `window.onload` property.

Anonymous functions give us a way to clean up this code. An anonymous function is just a function expression without a name that's used where we'd normally use a function reference. But to make the connection, it helps to see how we use a function expression in code in an anonymous way:

```
function handler() { alert("Yeah, that page loaded!"); }
window.onload = handler;
```

First remove the handler variable so this becomes a function expression.

```
window.onload = function() { alert("Yeah, that page loaded!"); };
```

Then assign the function expression directly to the window.onload property.

Now the handler has been assigned to the window.onload property without the need for an unnecessary name.

Much more concise now. We're assigning the function we need directly to the onload property. We also aren't creating a function name that might mistakenly be used in other code (after all, handler is a pretty common name!).

Look Ma! No names!

BRAIN POWER

Are there places in your previous code that you've seen anonymous functions and hadn't realized it?

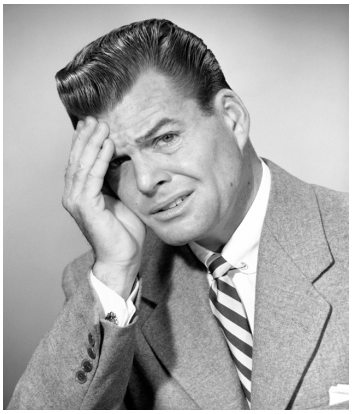
Hint: are they hiding somewhere in your objects?

Sharpen your pencil

There are a few opportunities in the code below to take advantage of anonymous functions. Go ahead and rework the code to use anonymous functions wherever possible. You can scratch out the old code and write in new code where needed. Oh, and one more task: circle any anonymous functions that are already being used in the code.

```
window.onload = init;
var cookies = {
    instructions: "Preheat oven to 350...",
    bake: function(time) {
              console.log("Baking the cookies.");
              setTimeout(done, time);
          }
};
function init() {
    var button = document.getElementById("bake");
    button.onclick = handleButton;
}
function handleButton() {
    console.log("Time to bake the cookies.");
    cookies.bake(2500);
}
function done() {
    alert("Cookies are ready, take them out to cool.");
    console.log("Cooling the cookies.");
    var cool = function() {
        alert("Cookies are cool, time to eat!");
    };
    setTimeout(cool, 1000);
}
```

We need to talk about your verbosity, again



Okay, we hate to bring it up again because you've come a long way with functions—you know how to pass functions around, assign them to variables, pass them to functions, return them from functions—but, well, you're still being a little more verbose than you have to (you could also say you're not being as expressive as you could be). Let's see an example:

Here's a normal-looking function named cookieAlarm that displays an alert about cookies being done.

```
function cookieAlarm() {
    alert("Time to take the cookies out of the oven");
};

setTimeout(cookieAlarm, 600000);
```

Looks like the cookies will be done in 10 minutes, just sayin'.

In case you forgot, these are milliseconds, so 1000 * 60 * 10 = 600,000.

And here we're taking the function and passing it as an argument to setTimeout.

While this code looks fine, we can make it a bit tighter using anonymous functions. How? Well, think about the `cookieAlarm` variable in the call to `setTimeout`. This is a variable that references a function, so when we invoke `setTimeout`, a function reference is passed. Now, using a variable that references a function is one way to get a function reference, but just like with the `window.onload` example a couple of pages back, you can use a function expression too. Let's rewrite the code with a function expression instead:

Now instead of a variable, we're just putting the function, inline, in the call to setTimeout.

Pay careful attention to the syntax here. We write the entire function expression, which ends in a right bracket, and then like any argument, we follow it with a comma before adding the next argument.

```
setTimeout(function() { alert("Time to take the cookies out of the oven");}, 600000);
```

We specify the name of the function we're calling, setTimeout, followed by a parenthesis and then the first argument, a function expression.

Here's the second argument, after the function expression.

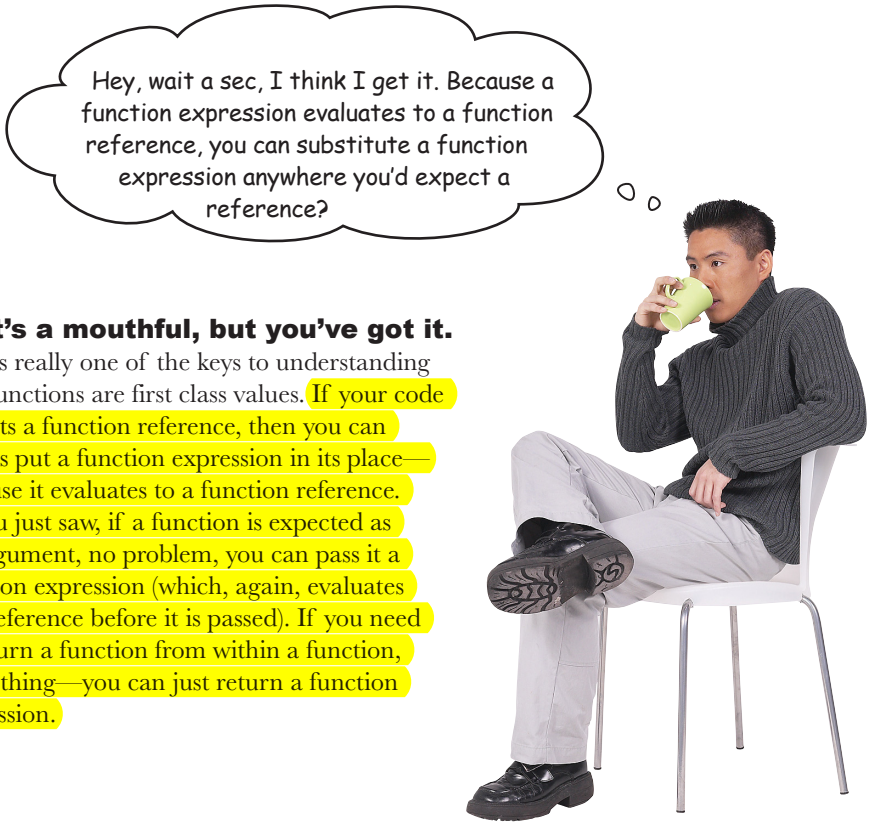
Who are you trying to kid? That's a mess. Who wants to read that one long line? And what if the function is long and complicated?

For a short piece of code, a one liner is just fine. But, beyond that, you're right, it would be rather silly. But as you know, we can use lots of whitespace with JavaScript, so we can insert all the spaces and returns we need to make things more readable. Here's our reformatting of the `setTimeout` code on the previous page.

All we've done is insert some whitespace—that is, some spaces and returns here and there.

```
setTimeout(function() {
                alert("Time to take the cookies out of the oven");
            }, 600000);
```

We're glad you raised this issue because the code is a lot more readable this way.



That's a mouthful, but you've got it.
This is really one of the keys to understanding that functions are first class values. If your code expects a function reference, then you can always put a function expression in its place—because it evaluates to a function reference. As you just saw, if a function is expected as an argument, no problem, you can pass it a function expression (which, again, evaluates to a reference before it is passed). If you need to return a function from within a function, same thing—you can just return a function expression.

Let's make sure you have the syntax down for passing anonymous function expressions to other functions. Convert this code from one that uses a variable (in this case vaccine) as an argument to one that uses an anonymous function expression.

```
function vaccine(dosage) {
    if (dosage > 0) {
        inject(dosage);
    }
}

administer(patient, vaccine, time);
```

Write your version here. And check your answer before moving on!

there are no Dumb Questions

Q: Using these anonymous functions like this seems really esoteric. Do I really need to know this stuff?

A: You do. Anonymous function expressions are used frequently in JavaScript code, so if you want to be able to read other people's code, or understand JavaScript libraries, you're going to need to know how these work, and how to recognize them when they're being used.

Q: Is using an anonymous function expression really better? I think it just complicates the code and makes the code hard to follow and read.

A: Give it some time. Over time, you'll be able to parse code like this more easily when you see it, and there really are lots of cases where this syntax decreases code complexity, makes the code's intention more clear, and cleans up your code. That said, overuse of this technique can definitely lead to code that is quite hard to understand. But stick with it and it'll get easier to read and more useful as you get the hang of it. You're going to encounter lots of code that makes heavy use of anonymous functions, so it's a good idea to incorporate this technique into your code toolbelt.

Q: If first class functions are so useful, how come other languages don't have them?

A: Ah, but they do (and even the ones that don't are considering adding them). For instance, languages like Scheme and Scala have fully first class functions like JavaScript does. Other languages, like PHP, Java (in the newest version), C#, and Objective C have some or most of the first class features that JavaScript does. As more people are recognizing the value of having first class functions in a programming language, more languages are supporting them. Each language does it a little differently, however, so be prepared for a variety of approaches as you explore this topic in other languages.

When is a function defined? It depends...

There's one fine point related to functions that we haven't mentioned yet. Remember that the browser takes two passes through your JavaScript code: in the first pass, all your function declarations are parsed and the functions defined; in the second pass, the browser executes your code top down, which is when function expressions are defined. Because of this, functions created by declarations are defined *before* functions that are created using function expressions. And this, in turn, determines where and when you can invoke a function in your code.

To see what that really means, let's take a look at a concrete example. Here's our code from the last chapter, rearranged just a bit. Let's evaluate it:

IMPORTANT: Read this by following the order of the numbers. Start at 1, then go to 2, and so on.

1 We start at the top of the code and find all the function declarations.

4 We start at the top again, this time evaluating the code.

```
var migrating = true;
```

5 Create the variable migrating and set it to true.

Notice that we moved this conditional up from the bottom of the code.

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

6 The conditional is true, so evaluate the code block.

7 Get the function reference from quack and invoke it with the argument 4.

8 Get the function reference from fly... oh wait, fly isn't defined!

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

2 We found a function declaration. We create the function and assign it to the variable quack.

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

3 We reach the bottom. Only one function declaration was found.

What just happened? Why wasn't fly defined?

Okay, we know the `fly` function is undefined when we try to invoke it, but why? After all, `quack` worked just fine. Well, as you've probably guessed by now, unlike `quack`—which is defined on the first pass through the code because it is a function declaration—the `fly` function is defined along with the normal top-to-bottom evaluation of the code. Let's take another look:

When we evaluate this code to try invoking quack, everything works as expected because quack was defined on the first pass through the code.

```
var migrating = true;
if (migrating) {
   quack(4);
   fly(4);
}
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!");
   }
}
```

But when we try to execute the call to the fly function, we get an error because we haven't yet defined fly...

...because fly doesn't get defined until this statement is evaluated, which is after the call to fly.

JavaScript console

```
Quack!
Quack!
Quack!
Quack!
TypeError: undefined is not a function
```

What happens when you try to call a function that's undefined.

You might see an error like this instead (depending on the browser you're using): TypeError: Property 'fly' of object [object Object] is not a function.

So what does this all mean? For starters, it means that you can place function declarations anywhere in your code—at the top, at the bottom, in the middle—and invoke them wherever you like. Function declarations create functions that are defined everywhere in your code (this is known as *hoisting*).

Function expressions are obviously different because they aren't defined until they are evaluated. So, even if you assign the function expression to a global variable, like we did with `fly`, you can't use that variable to invoke a function until after it's been defined.

Now in this example, both of our functions have *global scope*—meaning both functions are visible everywhere in your code once they are defined. But we also need to consider nested functions—that is functions defined within other functions—because it affects the scope of those functions. Let's take a look.

How to nest functions

It's perfectly legal to define a function within another function, meaning you can use a function declaration or expression inside another function. How does this work? Here's the short answer: the only difference between a function defined at the top level of your code and one that's defined within another function is just a matter of scope. In other words, placing a function in another function affects where the function is visible within your code.

To understand this, let's expand our example a little by adding some nested function declarations and expressions.

```
var migrating = true;
var fly = function(num) {
  var sound = "Flying";
  function wingFlapper() {
    console.log(sound);
  }
  for (var i = 0; i < num; i++) {
    wingFlapper();
  }
};
function quack(num) {
  var sound = "Quack";
  var quacker = function() {
    console.log(sound);
  };
  for (var i = 0; i < num; i++) {
    quacker();
  }
}
if (migrating) {
  quack(4);
  fly(4);
}
```

Here we're adding a function declaration with the name wingFlapper inside the fly function expression.

And here we're calling it.

Here we're adding a function expression assigned to the quacker variable inside the quack function declaration.

And here we're calling it.

We've moved this code back to the bottom so we no longer get that error when we call fly.

Exercise

In the code above, take a pencil and mark where you think the scope of the fly, quack, wingFlapper and quacker functions are. Also, mark any places you think the functions might be in scope but undefined.

How nesting affects scope

Functions defined at the top level of your code have global scope, whereas functions defined within another function have local scope. Let's make a pass over this code and look at the scope of each function. While we're at it, we'll also look at where each function is defined (or, not `undefined`, if you prefer):

```
var migrating = true;
var fly = function(num) {
  var sound = "Flying";
  function wingFlapper() {
    console.log(sound);
  }
  for (var i = 0; i < num; i++) {
    wingFlapper();
  }
};
function quack(num) {
  var sound = "Quack";
  var quacker = function() {
    console.log(sound);
  };
  for (var i = 0; i < num; i++) {
    quacker();
  }
}
if (migrating) {
  quack(4);
  fly(4);
}
```

Everything defined at the top level of the code has global scope. So fly and quack are both global variables.

But remember fly is defined only after this function expression is evaluated.

wingFlapper is defined by a function declaration in the fly function. So its scope is the entire fly function, and it's defined throughout the entire fly function body.

quacker is defined by a function expression in the function quack. So its scope is the entire quack function but it's defined only after the function expression is evaluated, until the end of the function body.

quacker is only defined here.

Notice that the rules for when you can refer to a function are the same within a function as they are at the top level. That is, within a function, if you define a nested function *with a declaration*, that nested function is defined everywhere within the body of the function. On the other hand, if you create a nested function using a *function expression*, then that nested function is defined only after the function expression is evaluated.

there are no Dumb Questions

Q: When we pass a function expression to another function, that function must get stored in a parameter, and then treated as a local variable in the function we passed it to. Is that right?

A: That's exactly right. Passing a function as an argument to another function copies the function reference we're passing into a parameter variable in the function we've called. And just like any other parameter, a parameter holding a function reference is a local variable.

EXTREME JAVASCRIPT CHALLENGE

We need a first class functions expert and we've heard that's you! Below you'll find two pieces of code, and we need your help figuring out what this code does. We're stumped. To us, these look like nearly identical pieces of code, except that one uses a first class function and the other doesn't. Knowing everything we do about JavaScript scope, we expected Specimen #1 to evaluate to 008 and Specimen #2 to evaluate to 007. But they both result in 008! Can you help us figure out why?

We recommend you form a strong opinion, jot it down on this page, and then turn the page.

Specimen #1

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue();
}

getSecret();
```

Specimen #2

```
var secret = "007";

function getSecret() {
  var secret = "008";

  function getValue() {
    return secret;
  }
  return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

Don't look at the solution at the end of the chapter just yet; we'll revisit this challenge a little bit later.

A little review of lexical scope

Lexical just means you can determine the scope of a variable by reading the structure of the code, as opposed to waiting until the code runs to figure it out.

While we're on the topic of scope, let's quickly review how lexical scope works:

Here we have a global variable called justAVar.

```
var justAVar = "Oh, don't you worry about it, I'm GLOBAL";

function whereAreYou() {
    var justAVar = "Just an every day LOCAL";

    return justAVar;
}

var result = whereAreYou();
console.log(result);
```

And this function defines a new lexical scope...

...in which we have a local variable, justAVar, that shadows the global variable of the same name.

When this function is called, it returns justAVar. But which one? We're using lexical scope, so we find the justAVar value by looking in the nearest function scope. And if we can't find it there, we look in the global scope.

So when we call whereAreYou, it returns the value of the local justAVar, not the global one.

JavaScript console

```
Just an every day LOCAL
```

Now let's introduce a nested function:

```
var justAVar = "Oh, don't you worry about it, I'm GLOBAL";

function whereAreYou() {
    var justAVar = "Just an every day LOCAL";

    function inner() {
        return justAVar;
    }

    return inner();
}

var result = whereAreYou();
console.log(result);
```

Here's the same function.

And shadow variable.

But now we have a nested function, that refers to justAVar. But which one? Well, again, we always use the variable from the closest enclosing function. So we're using the same variable as the last time.

Notice that we're calling inner here, and returning its result..

So when we call whereAreYou, the inner function is invoked, and returns the value of the local justAVar, not the global one.

JavaScript console

```
Just an every day LOCAL
```

Where things get interesting with lexical scope

Let's make one more tweak. Watch this step carefully; it's a doozy:

```
var justAVar = "Oh, don't you worry about it, I'm GLOBAL";

function whereAreYou() {
    var justAVar = "Just an every day LOCAL";

    function inner() {
        return justAVar;
    }

    return inner;
}

var innerFunction = whereAreYou();
var result = innerFunction();
console.log(result);
```

No changes at all here, same variables and functions.

But rather than invoking inner, we return the inner function.

So when we call whereAreYou, we get back a reference to inner function, which we assign to the innerFunction variable. Then we invoke innerFunction, capture its output in result and display the result.

So when inner is invoked here (as innerFunction), which justAVar is used? The local one, or the global one?

What matters is when the function is invoked. We invoke inner after it's returned, when the global version of justAVar is in scope, so we'll get "Oh don't worry about it, I'm GLOBAL".

Not so fast. With lexical scope what matters is the structure in which the function is defined, so the result has to be the value of the local variable, or "Just an everyday LOCAL".

Frank: What do you mean you're right? That's like defying the laws of physics or something. The local variable doesn't even exist anymore… I mean, when a variable goes out of scope it ceases to exist. It's derezzed! Didn't you see TRON!?

Judy: Maybe in your weak little C++ and Java languages, but not in JavaScript.

Jim: Seriously, how is that possible? The `whereAreYou` function has come and gone, and the local version of `justAVar` couldn't possibly exist anymore.

Judy: If you'd listen to what I just told you… In JavaScript that's not how it works.

Frank: Well, throw us a bone Judy. How does it work?

Judy: When we define the `inner` function, the local `justAVar` is in the scope of that function. Now lexical scope says how we define things is what matters, so if we're using lexical scope, then *whenever* `inner` is invoked, it assumes it still has that local variable around if it needs it.

Frank: Yeah, but like I already said, that's like defying the laws of physics. The `whereAreYou` function that defined the local version of the `justAVar` variable is over. It doesn't exist any more.

Judy: True. The `whereAreYou` function is done, but the scope is still around for `inner` to use.

Jim: How is that?

Judy: Well, let's see what REALLY happens when we define and return a function…

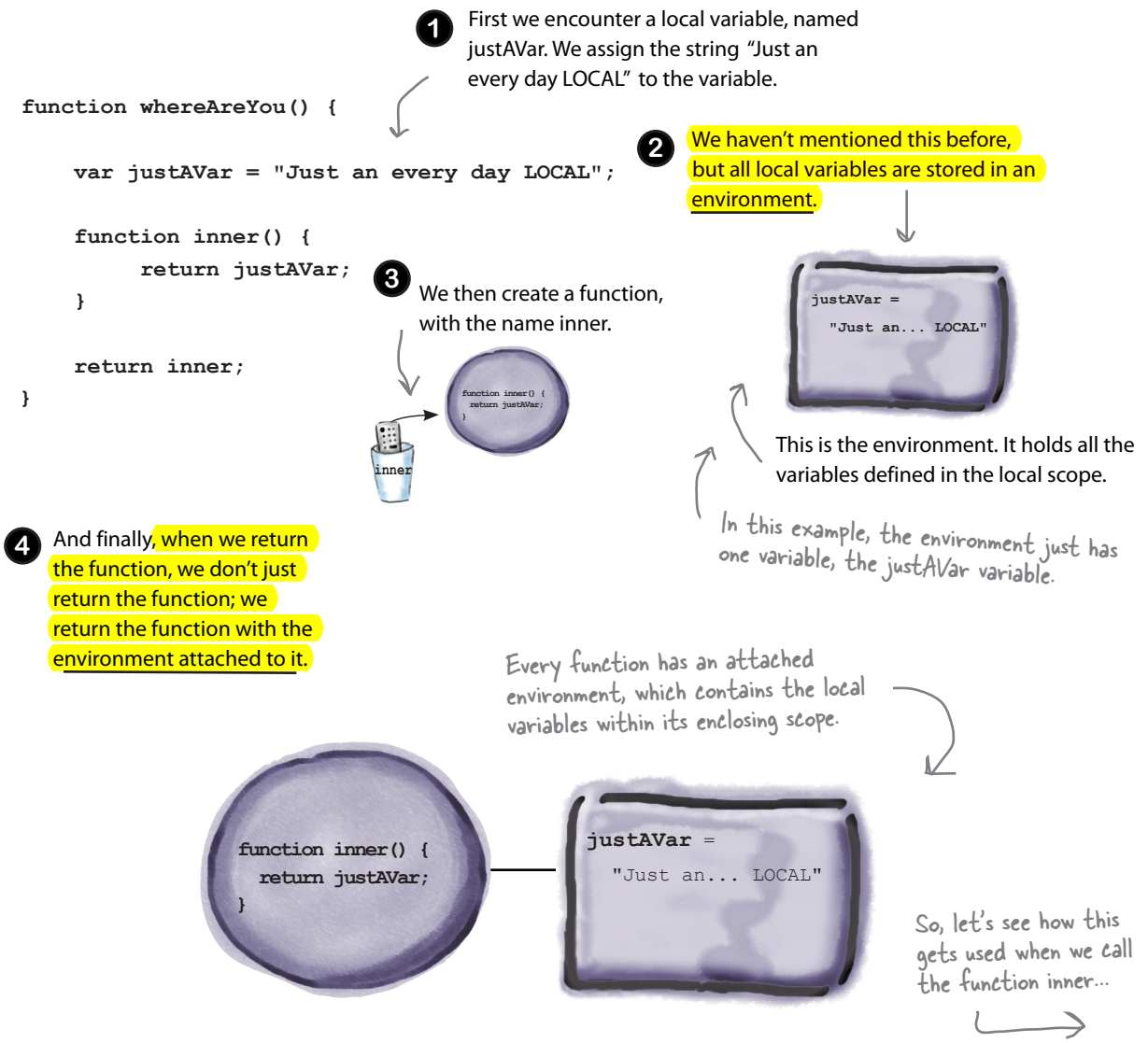
EDITOR'S NOTE: Did Joe change his shirt between pages!?

Functions Revisited

We have a bit of a confession to make. Up until now we haven't told you *everything* about a function. Even when you asked "What does a function reference actually point to?" we kinda skirted the issue. "Oh just think of it like a crystallized function that holds the function's code block," we said.

Well now it's time to show you everything.

To do that, let's walk through what really happens at runtime with this code, starting with the `whereAreYou` function:



Calling a function (revisited)

Now that we have the `inner` function, and its environment, let's invoke `inner` and see what happens. Here's the code we want to evaluate:

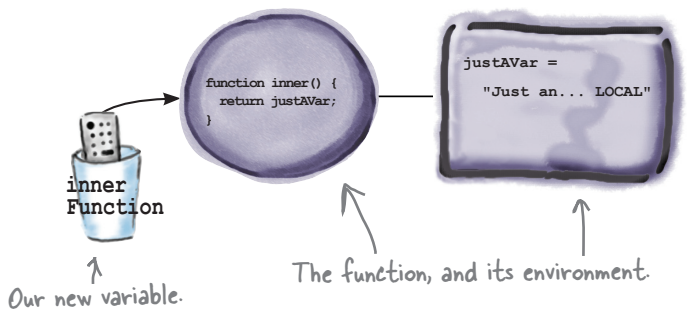
```
var innerFunction = whereAreYou();
var result = innerFunction();
console.log(result);
```

---

1 **First, we call whereAreYou. We already know that returns a function reference. So we create a variable innerFunction and assign it that function. Remember, that function reference is linked to an environment.**

```
var innerFunction = whereAreYou();
```

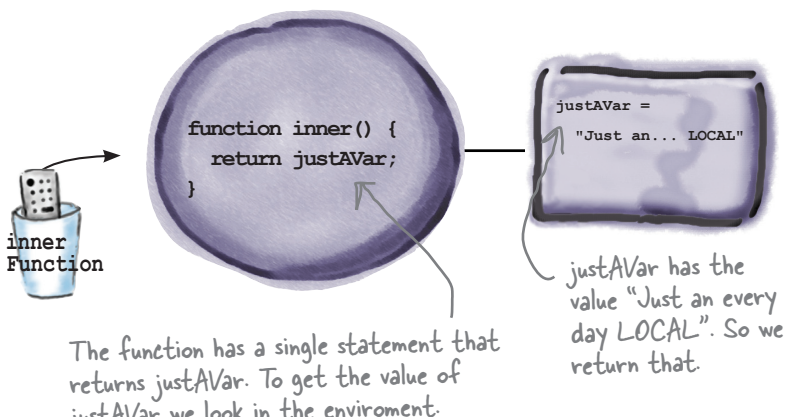
After this statement we have a variable innerFunction that refers to the function (plus an environment) returned from whereAreYou.



---

2 **Next we call innerFunction. To do that we evaluate the code in the function's body, and do that in the context of the function's environment, like this:**

```
var result = innerFunction();
```



3 **Last, we assign the result of the function to the result variable, and then display it in the console.**

```
var result = innerFunction();
console.log(result);
```

innerFunction returns the value "Just an every day LOCAL", which it got from its environment. So, we throw that into the result variable.

And then all we have to do is display that string in the console.

JavaScript console

```
Just an every day LOCAL
```

there are no Dumb Questions

Q: **When you say that lexical scope determines where a variable is defined, what do you mean?**

A: By lexical scope we mean that JavaScript's rules for scoping are based purely on the structure of your code (not on some dynamic runtime properties). This means you can determine where a variable is defined by simply examining your code's structure. Also remember that in JavaScript only functions introduce new scope. So, given a reference to a variable, look for where that variable is defined in a function from the most nested (where it's used) to the least nested until you find it. And if you can't find it in a function, then it must be global, or undefined.

Q: **If a function is nested way down many layers, how does the environment work then?**

A: We used a simplistic way of showing the environment to explain it, but you can think of each nested function as having its own little environment with its own variables. Then, what we do is create a chain of the environments of all the nested functions, from inner to outer.

So, when it comes to finding a variable in the environment, you start at the closest one, and then follow the chain until you find your variable. And, if you don't find it, you look in the global environment.

Q: **Why are lexical scoping and function environments good things? I would have thought the answer in that code example would be "Oh, don't you worry about it, I'm GLOBAL". That makes sense to me. The real answer seems confusing and counterintuitive.**

A: We can see how you might think that, but the advantage of lexical scope is that we can always look at the code to determine the scope that's in place when a variable is defined, and figure out what its value should be from that. And, as we've seen, this is true even if you return a function and invoke it much later in a place totally outside of its original scope.

Now there is another reason you might consider this a good thing, and that is the kind of things we can do in code with this capability. We're going to get to that in just a bit.

Q: **Do parameter variables get included in the environment too?**

A: Yes. As we've said before, you can consider parameters to be local variables in your functions, so they are included in the environment as well.

Q: **Do I need to understand how the environment works in detail?**

A: No. What you need to understand is the lexical scoping rules for JavaScript variables, and we've covered that. But now you know that if you have a function that is returned from within a function, it carries its environment around with it.

Remember that JavaScript functions are always evaluated in the same scoping environment in which they were defined. Within a function, if you want to determine where a variable is coming from, search in its enclosing functions, from the most nested to the least.

What the heck is a closure?

Sure, everyone talks about closures as *the must have* language feature, but how many people actually get what they are or how to use them? Darn few. It's the language feature everyone wants to understand and the feature every traditional language wants to add.

Here's the problem. According to many well-educated folks in the business, *closures are hard*. But that's really not a problem for you. Want to know why? No, no, it's not because this is a "brain friendly book" and no, it's not because we have a killer application that needs to be built to teach closures to you. It's because *you just learned them*. We just didn't call them closures.

So without further ado, we give you the super-formal definition.

If you've been trained well in this book you should be thinking at this point, "Ah, this is the 'get a big raise' knowledge."

Closure, noun: A closure is a function together with a referencing environment.

Okay, we agree that definition isn't totally illuminating. But why is it called *closure*? Let's quickly walk through that, because—seriously—this could be one of those make-or-break job interview questions, or the thing that gets you that raise at some point in the future.

To understand the word *closure*, we need to understand the idea of "closing" a function.

Sharpen your pencil

Here's your task: (1) find all the **free variables** in the code below and circle them. A free variable is one that isn't defined in the local scope. (2) Pick one of the environments on the right that **closes the function**. By that we mean that it provides values for all the free variables.

```
function justSayin(phrase) {
    var ending = "";
    if (beingFunny) {
       ending = " -- I'm just sayin!";
    } else if (notSoMuch) {
       ending = " -- Not so much.";
    }
    alert(phrase + ending);
}
```

Circle the free variables in this code. Free variables are not defined in the local scope.

```
beingFunny = true;
notSoMuch = false;
inConversationWith = "Paul";
```

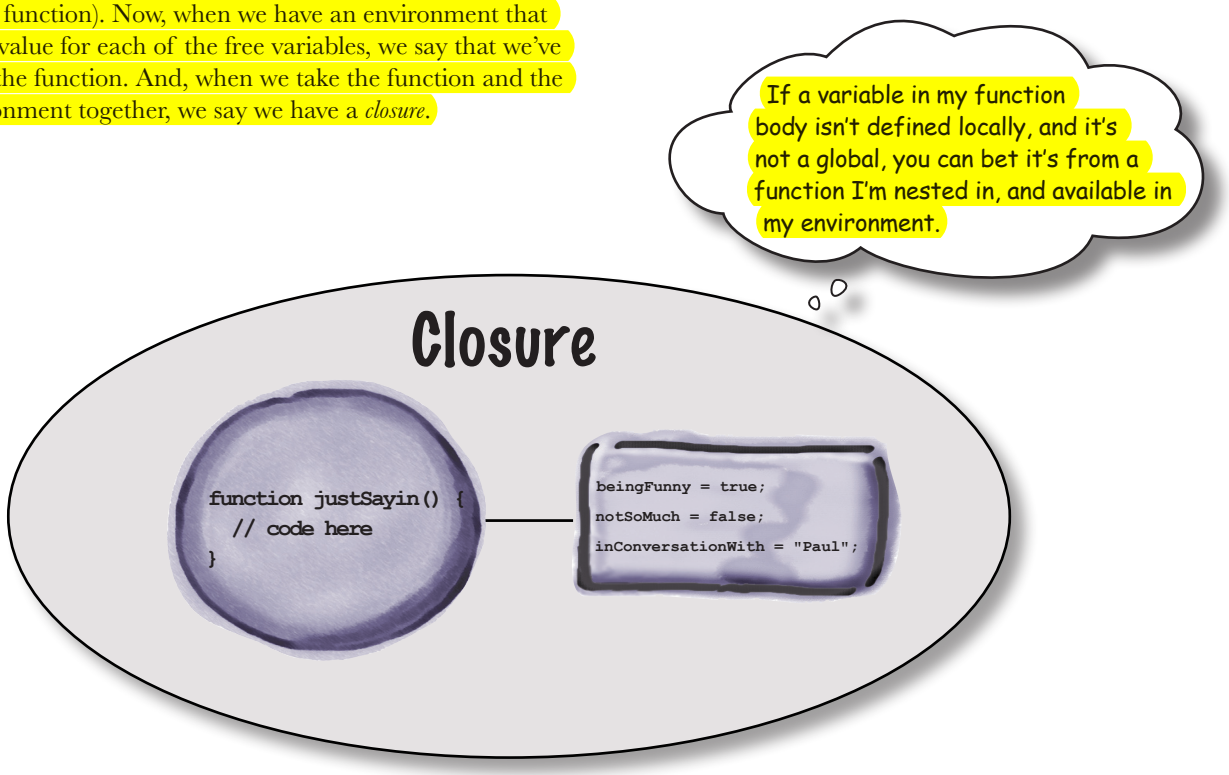
```
beingFunny = true;
justSayin = false;
oocoder = true;
```

```
notSoMuch = true;
phrase = "Do do da";
band = "Police";
```

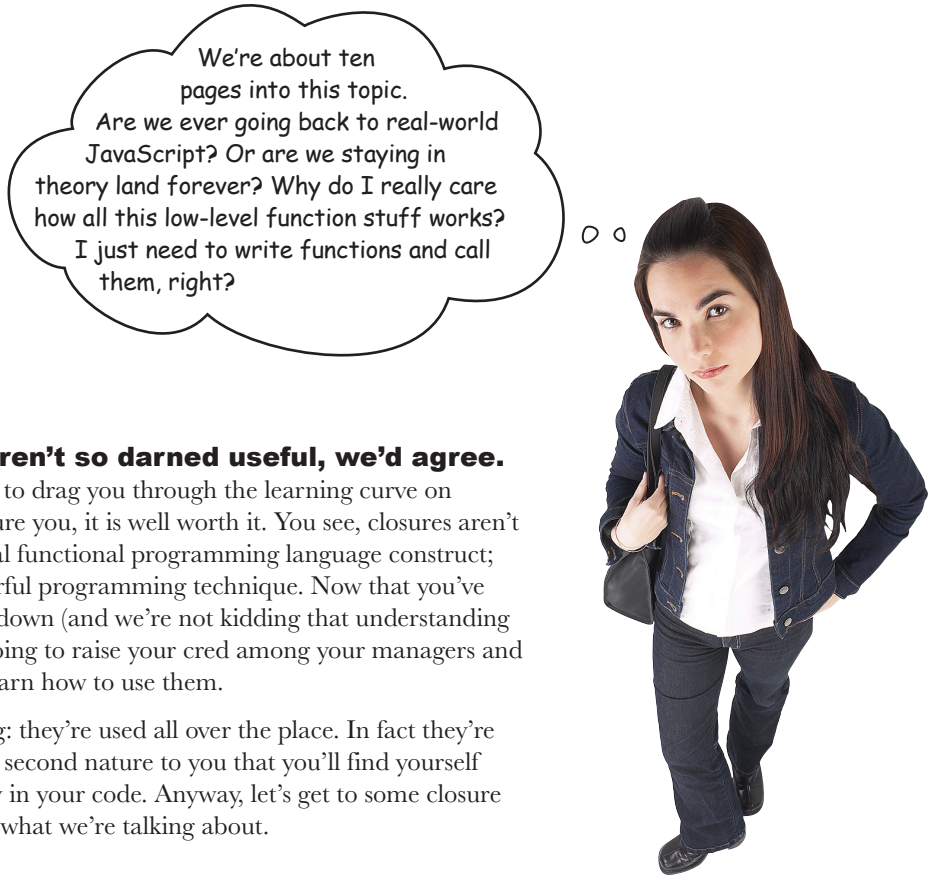
Pick one of these that closes the function.

Closing a function

You probably figured this out in the previous exercise, but let's run through it one more time: a function typically has *local variables* in its code body (including any parameters it has), and it also might have variables that aren't defined locally, which we call *free variables*. The name *free* comes from the fact that within the function body, free variables aren't bound to any values (in other words, they're not declared locally in the function). Now, when we have an environment that has a value for each of the free variables, we say that we've *closed* the function. And, when we take the function and the environment together, we say we have a *closure*.



A closure results when we combine a function that has free variables with an environment that provides variable bindings for all those free variables.



If closures weren't so darned useful, we'd agree.
We're sorry we had to drag you through the learning curve on closures but we assure you, it is well worth it. You see, closures aren't just some theoretical functional programming language construct; they're also a powerful programming technique. Now that you've got how they work down (and we're not kidding that understanding closures is what's going to raise your cred among your managers and peers) it's time to learn how to use them.

And here's the thing: they're used all over the place. In fact they're going to become so second nature to you that you'll find yourself using them liberally in your code. Anyway, let's get to some closure code and you'll see what we're talking about.

Using closures to implement a magic counter

Ever think of implementing a counter function? It usually goes like this:

```
var count = 0;

function counter() {
    count = count + 1;
    return count;
}
```

We have a global variable count.

Each time we call counter, we increment the global count variable, and return the new value.

And we can use our counter like this:

```
console.log(counter());
console.log(counter());
console.log(counter());
```

So we can count and display the value of the counter like this.

JavaScript console

```
1
2
3
```

The only issue with this is that we have to use a global variable for `count`, which can be problematic if you're developing code with a team (because people often use the same names, which end up clashing).

What if we were to tell you there is a way to implement a counter with a totally local and protected `count` variable? That way, you'll have a counter that no other code can ever clash with, and the only way to increment the counter value is through the function (otherwise known as a closure).

To implement this with a closure, we can reuse most of the code above. Watch and be amazed:

```
function makeCounter() {
    var count = 0;

    function counter() {
        count = count + 1;
        return count;
    }
    return counter;
}
```

Here, we're putting the count variable in the function makeCounter. So now count is a local variable, not a global variable.

Now, we create the counter function, which increments the count variable.

And return the counter function.

This is the closure. It holds count in its environment.

Think this magic trick will work? Let's try it and see…

Test drive your magic counter

We added a bit of testing code to test the counter. Give it a try!

```
function makeCounter() {
    var count = 0;

    function counter() {
        count = count + 1;
        return count;
    }
    return counter;
}
var doCount = makeCounter();
console.log(doCount());
console.log(doCount());
console.log(doCount());
```

JavaScript console

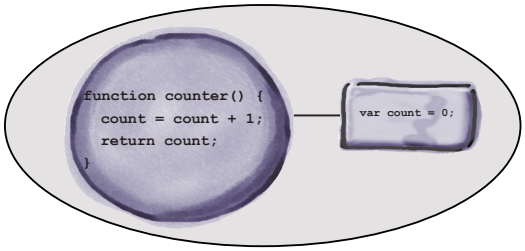
```
1
2
3
```

Our counter works... we get solid counting results.

Looking behind the curtain...

Let's step through the code to see how the counter works.

1. We call makeCounter, which creates a counter function and returns it along with an environment containing the free variable, count. In other words, it creates a closure. The function returned from makeCounter is stored in doCount.
2. We call the function doCount. This executes the body of the counter function.
3. When we encounter the variable count, we look it up in the environment, and retrieve its value. We increment count, save the new value back into the environment, and return that new value to where doCount was called.
4. We repeat steps 2 and 3 each time we call doCount.




```
function makeCounter() {
    var count = 0;

    function counter() {
        count = count + 1;   ③
        return count;
    }
    return counter;
}
① var doCount = makeCounter();
② console.log(doCount());
   console.log(doCount());
④ console.log(doCount());
```

This is a closure.

When we call doCount (which is a reference to counter) and need to get the value of count, we use the count variable that's in the closure's environment. The outside world (the global scope) never sees the variable count. But we can use it anytime we call doCount. And there's no other way to get to count except by calling doCount.

When we call makeCounter, we get back a closure: a function with an environment.

It's your turn. Try creating the following closures. We realize this is not an easy task at first, so refer to the answer if you need to. The important thing is to work your way through these examples, and get to the point where you fully understand them.

First up for 10pts: makePassword takes a password as an argument and returns a function that accepts a password guess and returns true if the guess matches the password (sometimes you need to read these closure descriptions a few times to get them):

```
function makePassword(password) {
    return _____________________________ {
        return (passwordGuess === password);
    };
}
```

Next up for 20pts: the multN function takes a number (call it n) and returns a function. That function itself takes a number, multiplies it by n and returns the result.

```
function multN(n) {
    return _____________________________ {
        return _______________;
    };
}
```

Last up for 30 pts: This is a modification of the counter we just created. makeCounter takes no arguments, but defines a count variable. It then creates and returns an object with one method, increment. This method increments the count variable and returns it.

Creating a closure by passing a function expression as an argument

Returning a function from a function isn't the only way to create a closure. You create a closure *whenever* you have a reference to a function that has free variables, and that function is executed outside of the context in which it was created.

Another way we can create a closure is to pass a function to a function. The function we pass will be executed in a completely different context than the one in which it was defined. Here's an example:

```
function makeTimer(doneMessage, n) {

    setTimeout(function() {
        alert(doneMessage);
    }, n);

}

makeTimer("Cookies are done!", 1000);
```

We have a function...

...with a free variable...

...that we are using as a handler for setTimeout.

...and this function will be executed 1000 milliseconds from now, long after the function makeTimer has completed.

Here, we're passing a function expression that contains a free variable, `doneMessage`, to the function `setTimeout`. As you know, what happens is we evaluate the function expression to get a function reference, which is then passed to `setTimeout`. The `setTimeout` method holds on to this function (which is a function plus an environment—in other words, a closure) and then 1000 milliseconds later it calls that function.

And again, the function we're passing into `setTimeout` is a closure because it comes along with an environment that binds the free variable, `doneMessage`, to the string "Cookies are done!".

BRAIN POWER

What would happen if our code looked like this instead?

```
function handler() {
    alert(doneMessage);
}
function makeTimer(doneMessage, n) {
    setTimeout(handler, n);
}
makeTimer("Cookies are done!", 1000);
```

BRAIN POWER 2

Revisit the code on page 412 in Chapter 9. Can you modify your code to use a closure, and eliminate the need for the third argument to setTimeout?

The closure contains the actual environment, not a copy

One thing that often misleads people learning closures is that they think the environment in the closure must have a copy of all the variables and their values. It doesn't. In fact, the environment references the live variables being used by your code, so if a value is changed by code outside your closure function, that new value is seen by your closure function when it is evaluated.

Let's modify our example to see what that means.

```
function setTimer(doneMessage, n) {

    setTimeout(function() {
        alert(doneMessage);
    }, n);

    doneMessage = "OUCH!";
}
setTimer("Cookies are done!", 1000);
```

The closure is created here.

Now we're changing the value of doneMessage after we call setTimeout.

1. When we call setTimeout and pass to it the function expression, a closure is created containing the function along with a reference to the environment.

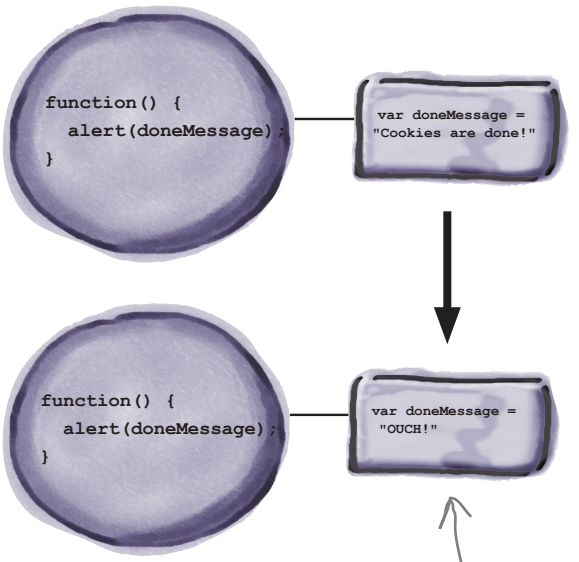
```
setTimeout(function() {
    alert(doneMessage);
}, n);
```

2. Then, when we change the value of doneMessage to "OUCH!" outside of the closure, it's changed in the same environment that is used by the closure.

```
doneMessage = "OUCH!";
```

3. 1000 milliseconds later, the function in the closure is called. This function references the doneMessage variable, which is now set to "OUCH!" in the environment, so we see "OUCH!" in the alert.

```
function() { alert(doneMessage); }
```



When the function is called, it uses the value for doneMessage that's in the environment, which is the new value we set it to earlier, in setTimer.

Creating a closure with an event handler

Let's look at one more way to create a closure. We'll create a closure with an event handler, which is something you'll see fairly often in JavaScript code. We'll start by creating a simple web page with a button and a `<div>` element to hold a message. We'll keep track of how many times you click the button and display the tally in the `<div>`.

Here's the HTML and a tiny bit of CSS to create the page. Go ahead and add the HTML and CSS below into a file named "divClosure.html".

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Click me!</title>
<style>
    body, button { margin: 10px; }
    div { padding: 10px; }
</style>
<script>
    // JavaScript code here
</script>
</head>
<body>
    <button id="clickme">Click me!</button>
    <div id="message"></div>
</body>
</html>
```

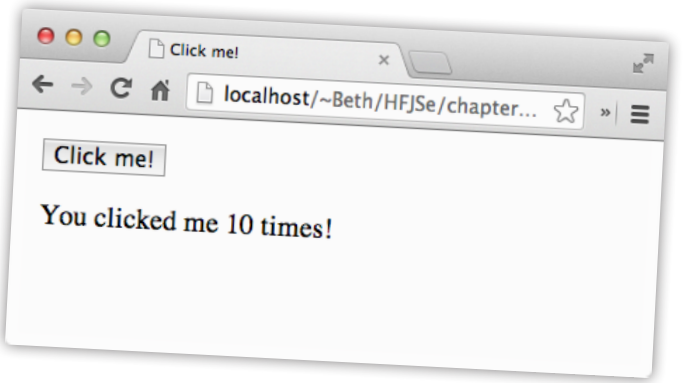
Just your typical, basic web page...

With a little CSS to style the elements in the page.

Here's where our code's going to go.

We have a button, and a to hold the message we'll update each time you click the button.

Here's what we're going for: each time you click the button, the message in the will be updated to show the number of times you've clicked.



Next, let's write the code. Now, you could write the code for this example without using a closure at all, but as you'll see, by using a closure, our code is more concise, and even a bit more efficient.

Click me! without a closure

Let's first take a look at how you'd implement this example *without* a closure.

```
var count = 0;

window.onload = function() {
    var button = document.getElementById("clickme");
    button.onclick = handleClick;
};

function handleClick() {
    var message = "You clicked me ";
    var div = document.getElementById("message");
    count++;
    div.innerHTML = message + count + " times!";
}
```

The count variable will need to be a global variable, because if it's local to handleClick (the click event handler on the button, see below), it'll just get re-initialized every time we click.

In the load event handler function, we get the button element, and add a click handler to the onclick property.

Here's the button's click handler function.

We define the message variable...

...get the element from the page...

...increment the counter...

...and update the with the message containing how many times we've clicked.

Click me! with a closure

The version without a closure looks perfectly reasonable, except for that global variable which could potentially cause trouble. Let's rewrite the code using a closure and see how it compares. We'll show the code here, and take a closer look after we test it.

```
window.onload = function() {
    var count = 0;
    var message = "You clicked me ";
    var div = document.getElementById("message");

    var button = document.getElementById("clickme");
    button.onclick = function() {
        count++;
        div.innerHTML = message + count + " times!";
    };
};
```

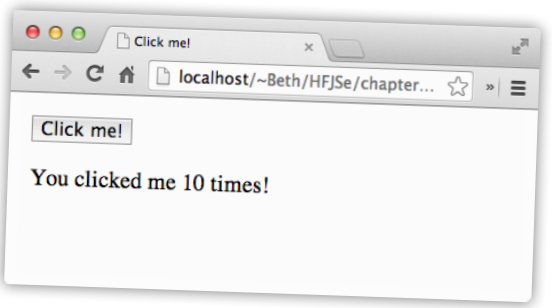
Now, all our variables are local to window.onload. No problems with name clashing now.

We're setting up the click handler as a function expression assigned to the button's onclick property, so we can reference div, message and count in the function. (Remember your lexical scoping!)

This function has three free variables: div, message and count, so a closure is created for the click handler function. So what gets assigned to the button's onclick property is a closure.

Test drive your button counter

Okay, let's bring the HTML and the code together in your "divClosure.html" file and give this a test run. Go ahead and load the page and then click on the button to increment the counter. You should see the message update in the `<div>`. Look at the code again, and make sure you think you know how this all works. After you've done so, turn the page and we'll walk through it together.



Here's what we got.

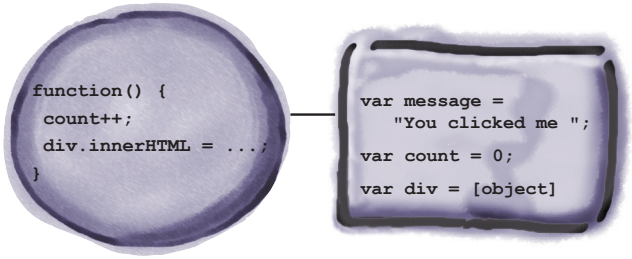
Update your "divClosure.html" file like this.

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Click me!</title>
<style>
    body, button { margin: 10px; }
    div { padding: 10px; }
</style>
<script>
    window.onload = function() {
        var count = 0;
        var message = "You clicked me ";
        var div = document.getElementById("message");

        var button = document.getElementById("clickme");
        button.onclick = function() {
            count++;
            div.innerHTML = message + count + " times!";
        };
    };
</script>
</head>
<body>
    <button id="clickme">Click me!</button>
    <div id="message"></div>
</body>
</html>
```

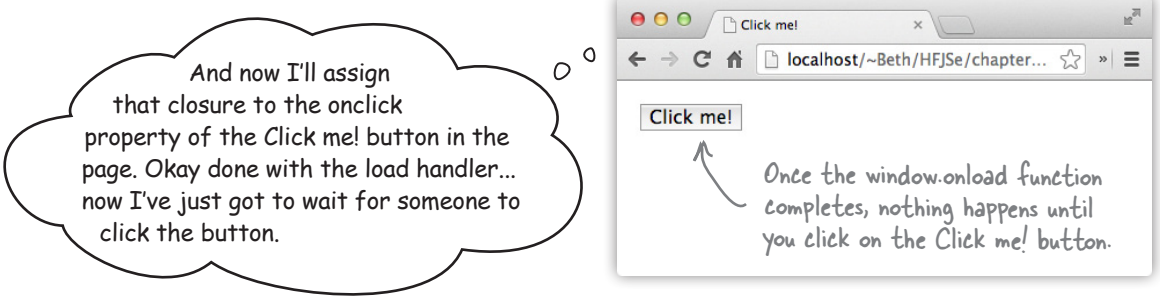
How the Click me! closure works

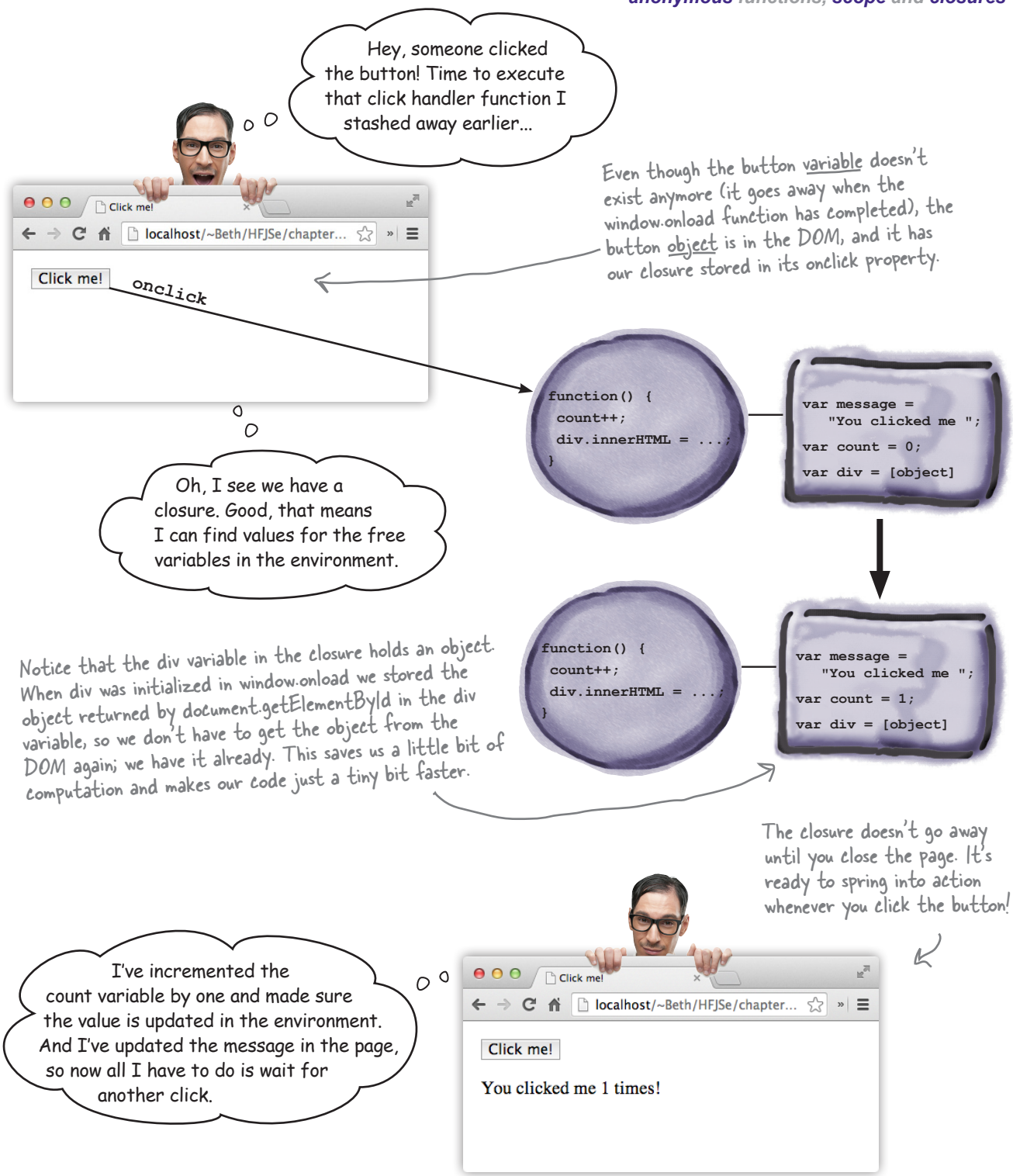
To understand how the closure works, let's follow along with the browser once again, as it evaluates this code…


```
window.onload = function() {
   var count = 0;
   var message = "You clicked me ";
   var div = document.getElementById("message");

   var button = document.getElementById("clickme");
   button.onclick = function() {
      count++;
      div.innerHTML = message + count + " times!";
   };
};
```



The browser creates a closure for the function that's assigned to the button.onclick property. The environment has the div variable, the message variable and the count variable.




Hey, someone clicked the button! Time to execute that click handler function I stashed away earlier...
Click me!
localhost/~Beth/HFJSe/chapter...
Click me!
onclick
Even though the button variable doesn't exist anymore (it goes away when the window.onload function has completed), the button object is in the DOM, and it has our closure stored in its onclick property.
function() {
count++;
div.innerHTML = ...;
}
var message =
"You clicked me ";
var count = 0;
var div = [object]
Oh, I see we have a closure. Good, that means I can find values for the free variables in the environment.
function() {
count++;
div.innerHTML = ...;
}
var message =
"You clicked me ";
var count = 1;
var div = [object]
Notice that the div variable in the closure holds an object. When div was initialized in window.onload we stored the object returned by document.getElementById in the div variable, so we don't have to get the object from the DOM again; we have it already. This saves us a little bit of computation and makes our code just a tiny bit faster.
The closure doesn't go away until you close the page. It's ready to spring into action whenever you click the button!
I've incremented the count variable by one and made sure the value is updated in the environment. And I've updated the message in the page, so now all I have to do is wait for another click.
Click me!
localhost/~Beth/HFJSe/chapter...
Click me!
You clicked me 1 times!

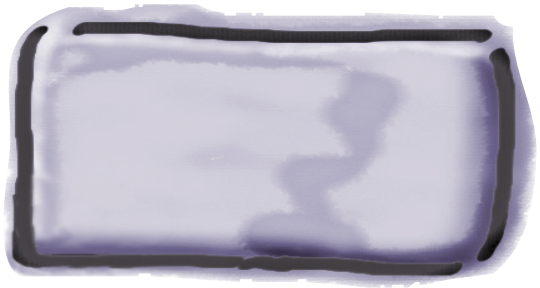
EXTREME JAVASCRIPT CHALLENGE REVISITED

We need a closures expert and we've heard that's you! Now you know how closures work, can you figure out why both specimens below evaluate to 008? To figure it out, write any variables that are captured in the environments for the functions below. Note that it's perfectly fine for an environment to be empty. Check your answer at the end of the chapter.

Specimen #1

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue();
}
getSecret();
```

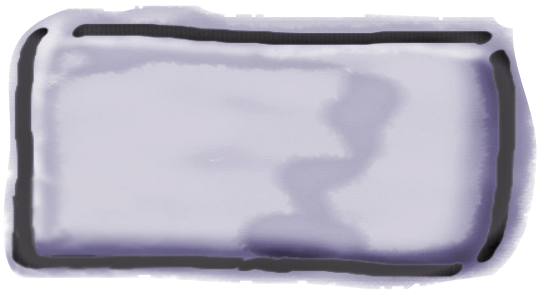
Environment



Specimen #2

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

Environment



Sharpen your pencil

First, check out this code:

```
(function(food) {
    if (food === "cookies") {
        alert("More please");
    } else if (food === "cake") {
        alert("Yum yum");
    }
})("cookies");
```

Using a function expression in place of a reference, taken to the extreme.

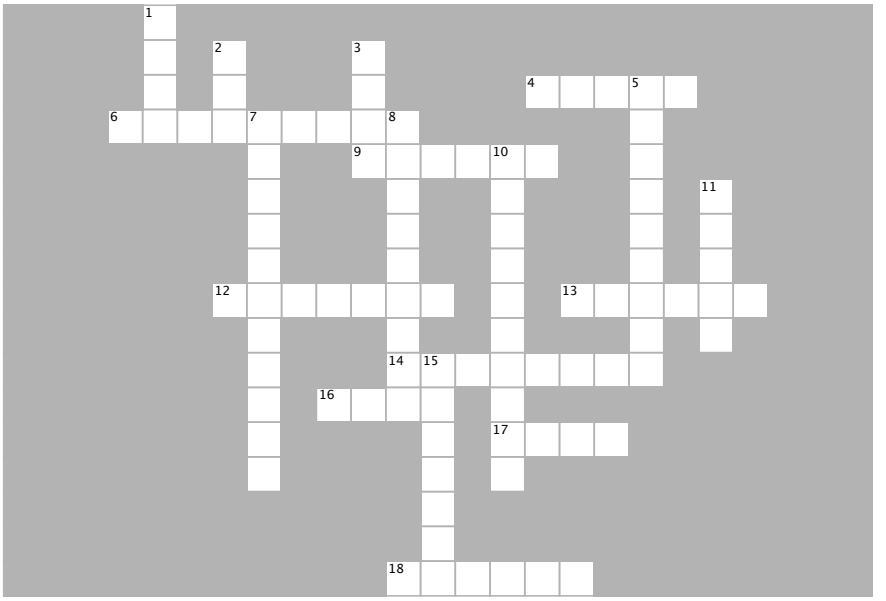
Your task is to figure out not just what this code computes, but *how* it computes. To do that, go in reverse. That is, take out the anonymous function, assign it to a variable, and then use that variable where the function expression used to be. Is the code more obvious now? So, what does it do?

BULLET POINTS

- An **anonymous function** is a function expression that has no name.
- Anonymous functions can make your code more concise.
- A **function declaration** is defined before the rest of your code is evaluated.
- A **function expression** is evaluated at runtime with the rest of your code, and so is not defined until the statement in which it appears is evaluated.
- You can pass a function expression to another function, or return a function expression from a function.
- A function expression evaluates to a **function reference**, so you can use a function expression anywhere you can use a function reference.
- **Nested functions** are functions defined inside another function.
- A nested function has local scope, just like other local variables.
- **Lexical scope** means that we can determine the scope of a variable by reading our code.
- To bind the value of a variable in a nested function, use the value that's defined in the closest enclosing function. If no value is found, then look in the global scope.
- **Closures** are a function along with a referencing environment.
- A closure captures the value of variables in scope at the time the closure is created.
- **Free variables** in the body of a function are variables that are not bound in the body of that function.
- If you execute a function closure in a different context in which it was created, the values of free variables are determined by the referencing environment.
- Closures are often used to capture state for event handlers.

JavaScript cross

Time for another crossword puzzle to burn some JavaScript into those neuron pathways.



ACROSS

4. A function declaration nested in another function has _____ scope.

6. When we tried to call fly before it was defined, we got this kind of error.

9. wingFlapper is a _____ function.

12. We often use setTimeout to create a timer for making _____.

13. A function expression assigned to a variable at the top level of your code has _____ scope.

14. To get a raise, you should understand how _____ work.

16. A _____ variable is one that's not defined in the local scope.

17. We changed the value of doneMessage to _____ in the closure.

18. An environment that provides values for all free variables _____ a function.

DOWN

1. _____ is always right.

2. ____ changed his shirt between pages.

3. Movie the word "derezzed" was used in.

5. An _____ function is a function expression that has no name.

7. A function with an _____ attached to it is called a closure.

8. A function expression evaluates to a function _____.

10. We passed a function _____ to set the cookie alarm.

11. Parameters are _____ variables, so they're included in the environment where variables are defined.

15. _____ scope means you can understand the scope of your variables by reading the structure of your code.

Sharpen your pencil Solution

There are a few opportunities in the code below to make the code more concise by using anonymous functions. Go ahead and rework the code to use anonymous functions wherever possible. You can scratch out the old code and write in new code where needed. Oh, and one more task: circle any anonymous functions that are already being used in the code. Here's our solution.

```
window.onload = init;
var cookies = {
    instructions: "Preheat oven to 350...",
    bake: function(time) {
        console.log("Baking the cookies.");
        setTimeout(done, time);
    }
};
function init() {
    var button = document.getElementById("bake");
    button.onclick = handleButton;
}
function handleButton() {
    console.log("Time to bake the cookies.");
    cookies.bake(2500);
}
function done() {
    alert("Cookies are ready, take them out to cool.");
    console.log("Cooling the cookies.");
    var cool = function() {
        alert("Cookies are cool, time to eat!");
    };
    setTimeout(cool, 1000);
}
```

We reworked the code to create two anonymous function expressions, one for the init function, and one for the handleButton function.

Now we assign a function expression to the window.onload property...

```
window.onload = function() {
    var button = document.getElementById("bake");
    button.onclick = function() {
        console.log("Time to bake the cookies.");
        cookies.bake(2500);
    };
};

var cookies = {
    instructions: "Preheat oven to 350...",
    bake: function(time) {
            console.log("Baking the cookies.");
            setTimeout(done, time);
        }
};

function done() {
    alert("Cookies are ready, take them out to cool.");
    console.log("Cooling the cookies.");
    var cool = function() {
        alert("Cookies are cool, time to eat!");
    };
    setTimeout(cool, 1000);
}
```

...and assign a function expression to the button.onclick property.

Extra credit for you if you figured out you can pass the cool function directly to setTimeout, like this:

```
setTimeout(function() {
    alert("Cookies are cool, time to eat!");
}, 1000);
```

Let's make sure you have the syntax down for passing anonymous function expressions to other functions. Convert this code from one that uses a variable (in this case vaccine) as a parameter to one that uses an anonymous function. Here's our solution.

```
administer(patient, function(dosage) {
    if (dosage > 0) {
        inject(dosage);
    }
}, time);
```

Notice that it's totally fine to use more than one line for a function expression that's used as an argument. But watch your syntax; it's easy to make a mistake!

It's your turn. Try creating the following closures. We realize this is not an easy task at first, so refer to the answer if you need to. The important thing is to work your way through these examples, and get to the point where you fully understand them.

Here are our solutions:

First up for 10pts: makePassword takes a password as an argument and returns a function that accepts a password guess and returns true if the guess matches the password (sometimes you need to read these closure descriptions a few times to get them):

```
makePassword(password) {
    return function guess(passwordGuess) {
        return (passwordGuess === password);
    };
}
var tryGuess = makePassword("secret");
console.log("Guessing 'nope': " + tryGuess("nope"));
console.log("Guessing 'secret': " + tryGuess("secret"));
```

The function that's returned from makePassword is a closure with an environment containing the free variable password.

We pass in the value "secret" to makePassword, so this is the value that's stored in the closure's environment.

Notice here we're using a named function expression! We don't have to, but it's handy as a way to refer to the name of the inner function. But also notice we must invoke the returned function using tryGuess (not guess).

And when we invoke tryGuess, we compare the word we pass in ("nope" or "secret") with the value for password in the environment for tryGuess.

The solutions continue on the next page...

It's your turn. Try creating the following closures. We realize this is not an easy task at first, so refer to the answer if you need to. The important thing is to work your way through these examples, and get to the point where you fully understand them.

Here are our solutions (continued):

Next up for 20pts: the multN function takes a number (call it n) and returns a function. That function itself takes a number, multiplies it by n and returns the result.

```
function multN(n) {
    return function multBy(m) {
        return n*m;
    };
}
var multBy3 = multN(3);
console.log("Multiplying 2: " + multBy3(2));
console.log("Multiplying 3: " + multBy3(3));
```

The function that's returned from multN is a closure with an environment containing the free variable n.

So we invoke multN(3) and get back a function that multiplies any number you give it by 3.

Last up for 30 pts: This is a modification of the counter we just created. makeCounter takes no arguments, but defines a count variable. It then creates and returns an object with one method, increment. This method increments the count variable and returns it.

```
function makeCounter() {
    var count = 0;
    return {
        increment: function() {
            count++;
            return count;
        }
    };
}
var counter = makeCounter();
console.log(counter.increment());
console.log(counter.increment());
console.log(counter.increment());
```

This is similar to our previous makeCounter function, except now we're returning an object with an increment method, instead of returning a function directly.

The increment method has a free variable, count. So, increment is a closure with an environment containing the variable count.

Now, we call makeCounter and get back an object with a method (that is a closure).

We invoke the method in the usual way, and when we do, the method references the variable count in its environment.

Sharpen your pencil Solution

Use your knowledge of functions and variables and check off the true statements below. Here's our solution:

- [x] The handler variable holds a function reference.
- [x] When we assign handler to window.onload, we're assigning it a function reference.
- [x] The only reason the handler variable exists is to assign it to window.onload.
- [x] We'll never use handler again as it's code that is meant to run only when the page first loads.
- [x] Invoking onload handlers twice is not a great idea—doing so could cause issues given these handlers usually do some initialization for the entire page.
- [x] Function expressions create function references.
- [x] Did we mention that when we assign handler to window.onload, we're assigning it a function reference?

Sharpen your pencil Solution

Here's your task: (1) find all the **free variables** in the code below and circle them. A free variable is one that isn't defined in the local scope. (2) Pick one of the environments on the right that **closes the function**. By that we mean that it provides values for all the free variables. Here's our solution.

```
function justSayin(phrase) {
    var ending = "";
    if (beingFunny) {
       ending = " -- I'm just sayin!";
    } else if (notSoMuch) {
       ending = " -- Not so much.";
    }
    alert(phrase + ending);
}
```

Circle the free variables in this code. Free variables are not defined in the local scope.

This environment closes the two free variables beingFunny and notSoMuch

```
beingFunny = true;
notSoMuch = false;
inConversationWith = "Paul";
```

```
beingFunny = true;
justSayin = false;
oocoder = true;
```

```
notSoMuch = true;
phrase = "Do do da";
band = "Police";
```

Pick one of these that closes the function.

SOLUTION

EXTREME JAVASCRIPT CHALLENGE

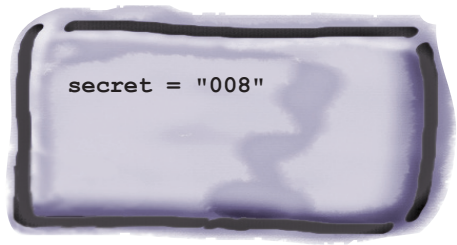
We need a closures expert and we've heard that's you! Now you know how closures work, can you figure out why both specimens below evaluate to 008? To figure it out, write any variables that are captured in the environments for the functions below. Note that it's perfectly fine for an environment to be empty. Here's our solution.

Specimen #1

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue();
}
getSecret();
```

secret is a free variable in getValue...

Environment



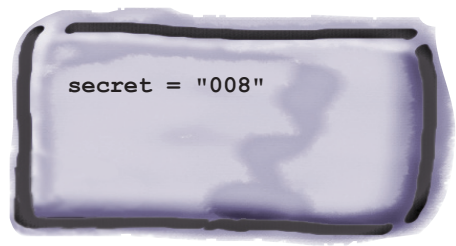
...so it's captured in the environment for getValue. But we don't return getValue from getSecret, so we never see the closure outside the context in which it was created.

Specimen #2

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

secret is a free variable in getValue...

Environment



...and here, we do create a closure that's returned from getSecret. So when we invoke getValueFun (getValue) in a different context (the global scope), we use the value of secret in the environment.

Sharpen your pencil Solution

Here's our solution for this brain twister!

```
(function(food) {
    if (food === "cookies") {
        alert("More please");
    } else if (food === "cake") {
        alert("Yum yum");
    }
})("cookies");
```

Your task is to figure out not just what it computes, but how it computes. To do that, go in reverse, that is, take out the anonymous function, assign it to a variable, and then replace the previous function with the variable. Is the code more obvious now? So what does it do?

Here's the function, extracted. We just called it eat. You could have made this a function declaration if you preferred.

```
var eat = function(food) {
    if (food === "cookies") {
        alert("More please");
    } else if (food === "cake") {
        alert("Yum yum");
    }
};
(eat)("cookies");
```

You would write this as eat("cookies") of course, but we're showing how to substitute eat for the function expression above.

And what we're doing is calling eat on "cookies". But what are the extra parentheses for?

So all this code did was to inline a function expression and then immediately invoke it with some arguments.

Oh, and it returns "More please".

Here's the deal. Remember how a function declaration starts with the word function followed by a name? And remember how a function expression needs to be inside a statement? Well, if you don't use parentheses around the function expression, the JavaScript interpreter wants this to be a declaration rather than a function expression. But we don't need the parentheses to call eat, so you can remove them.

JavaScript cross Solution