

## Chapter 8. Building an app

### Bringing it All Together

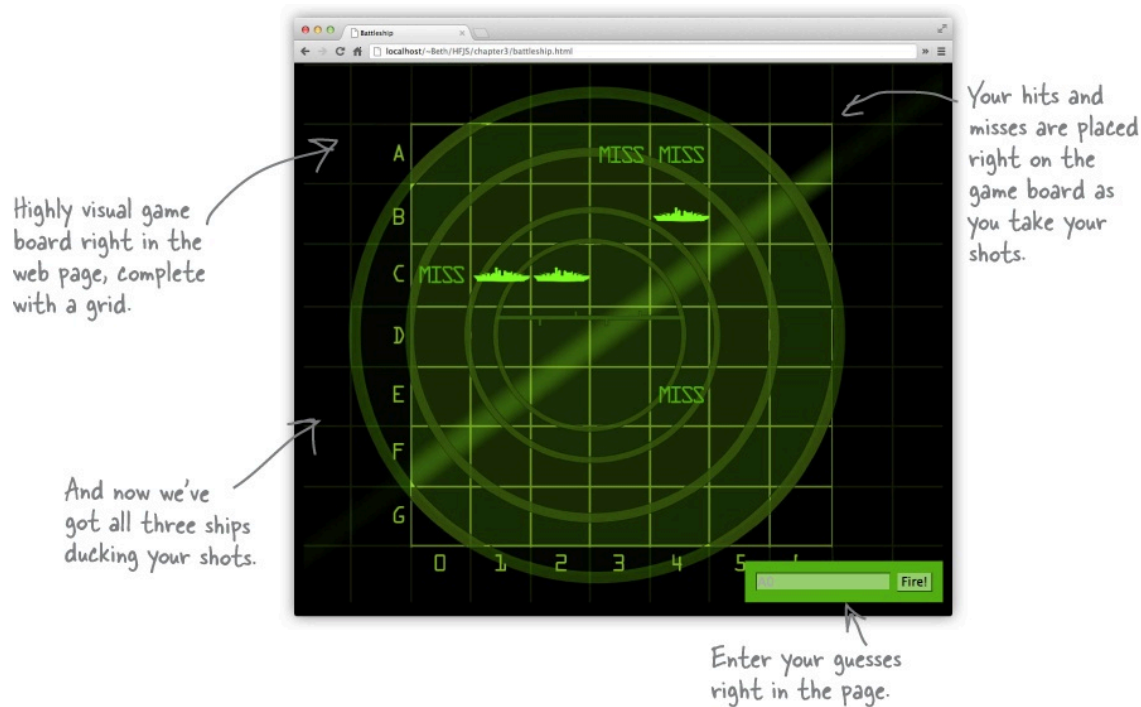


**Put on your toolbelt.** That is, the toolbelt with all your new coding skills, your knowledge of the DOM, and even some HTML & CSS. We're going to bring everything together in this chapter to create our first true **web application**. No more **silly toy games** with one battleship and a single row 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:



---

## BRAIN POWER

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

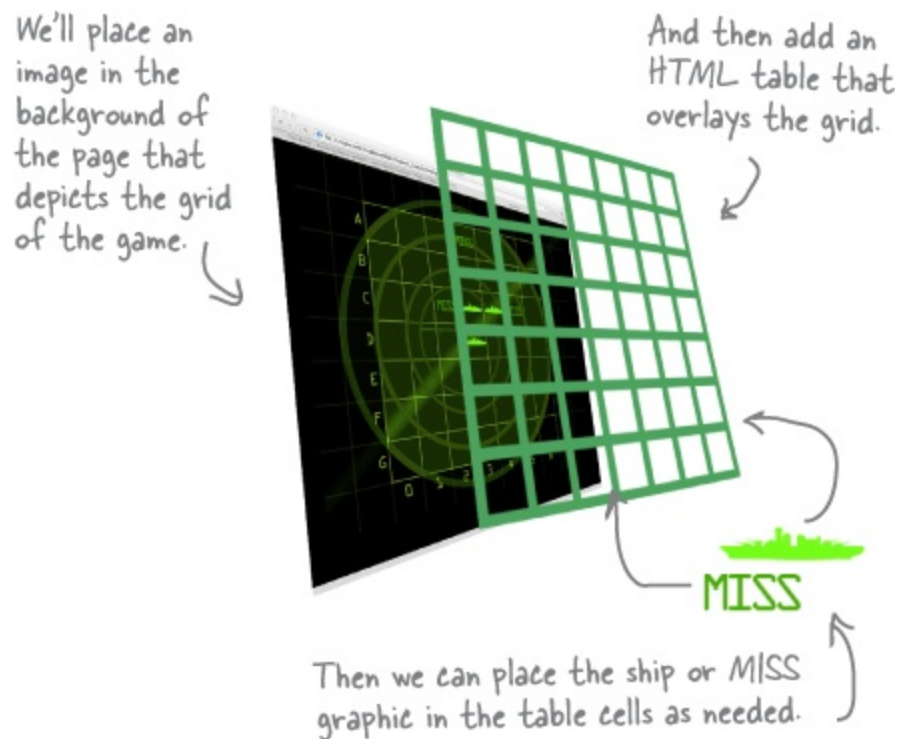
---

# 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.

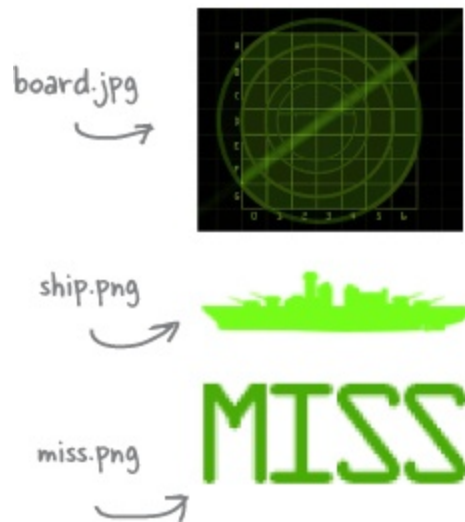
To pull that off we’re going to do things like use an image in the background to give us the slick grid over a radar look, and then we’ll lay a more functional HTML table over that so we can place things (like ships) on top of it. We’ll also use an HTML form to get the player input.



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.

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.

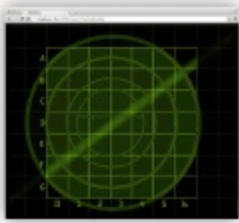
Download everything you need for the game at <http://wickedlysmart.com/hfjs>

---

## Creating the HTML page: the Big Picture

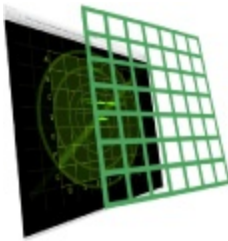
Here's the plan of attack for creating the Battleship HTML page:

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.



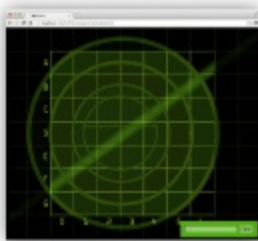
We're placing an image in the background to give the game its cool, green phosphorus radar feel.

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.



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

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!"



An HTML form for player input.

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

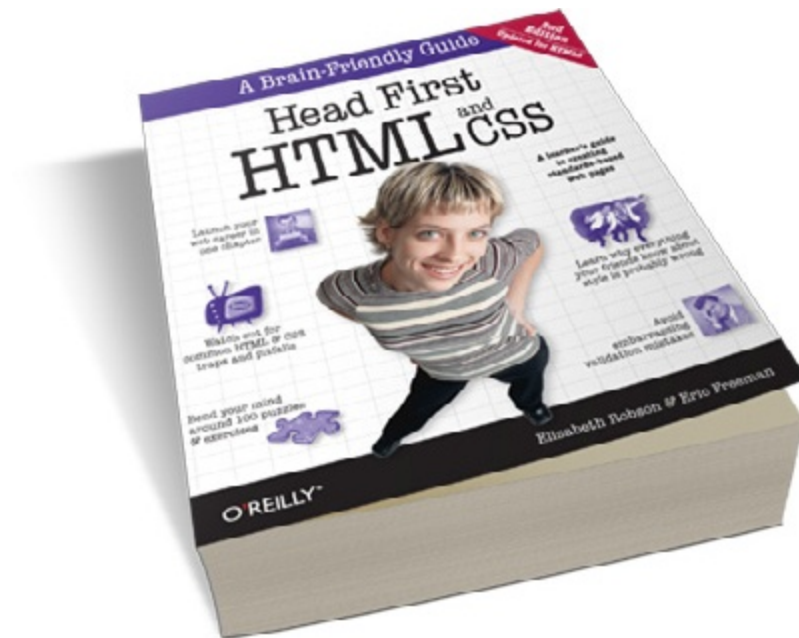
## Step 1: The Basic HTML

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.

---

RELAX



### 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" <div> element. We're positioning this <div> relative, so that we can position the table we add in the next step relative to this <div>.

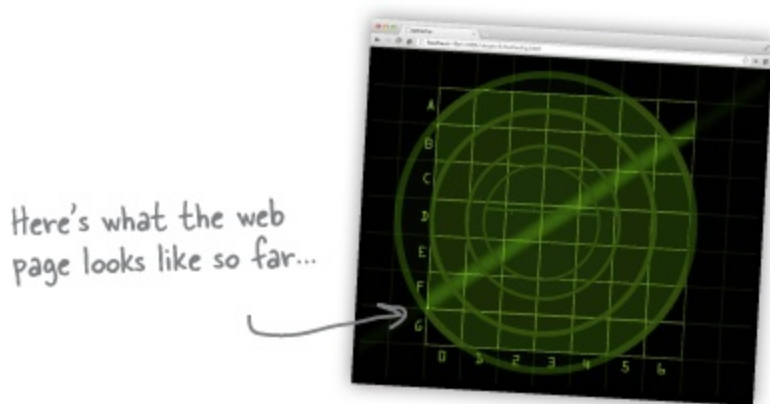
We're going to put the table for the game board and the form for getting user input here.

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

## A TEST DRIVE

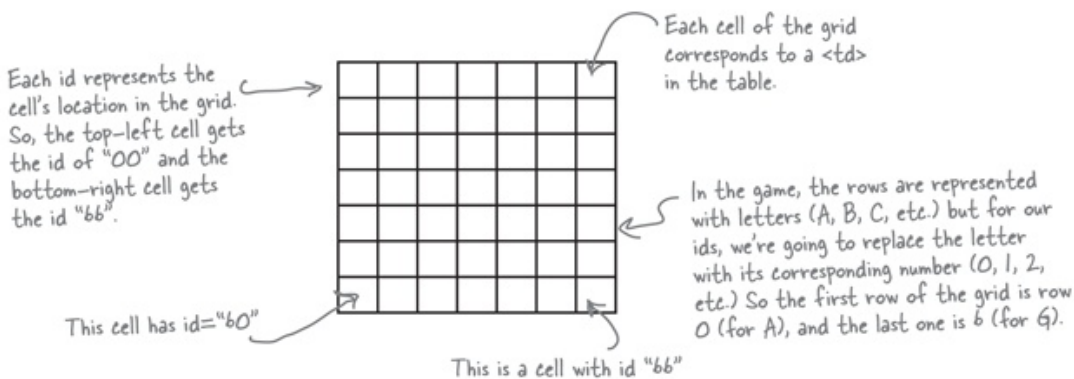


Go ahead and enter the code above (or download all the code for the book from <http://wickedlysmart.com/hfjs>) into the file "battleship.html" and then load it in your browser. Our test run is below.



## 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 `<div>` tags:

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

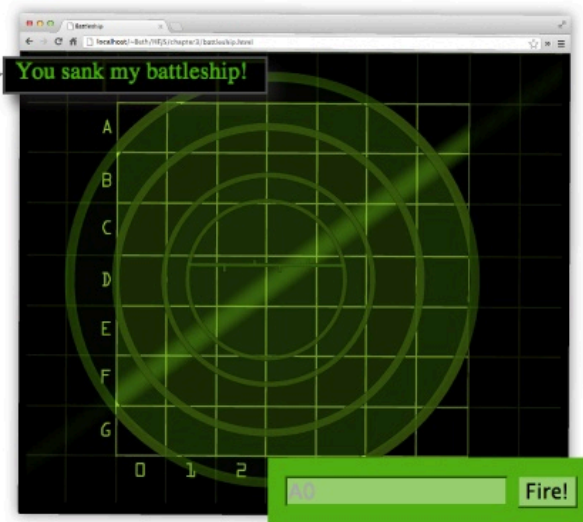
## Step 3: Player interaction

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 sub-



mit 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>
```

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

The messageArea `<div>` will be used to display messages from code.

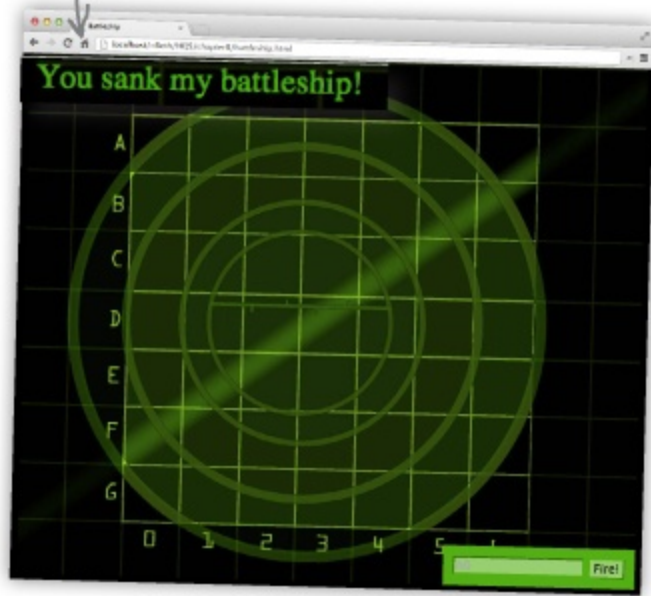
The `<form>` has two inputs: one for the guess (a text input) and one for the button. Note the ids on these elements. We'll need them later when we write the code to get the player's guess.

## 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.

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



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" `<div>` is positioned relative, so everything nested within this `<div>` can be positioned relative to it.

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

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

---

## 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” `<div>`, 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.

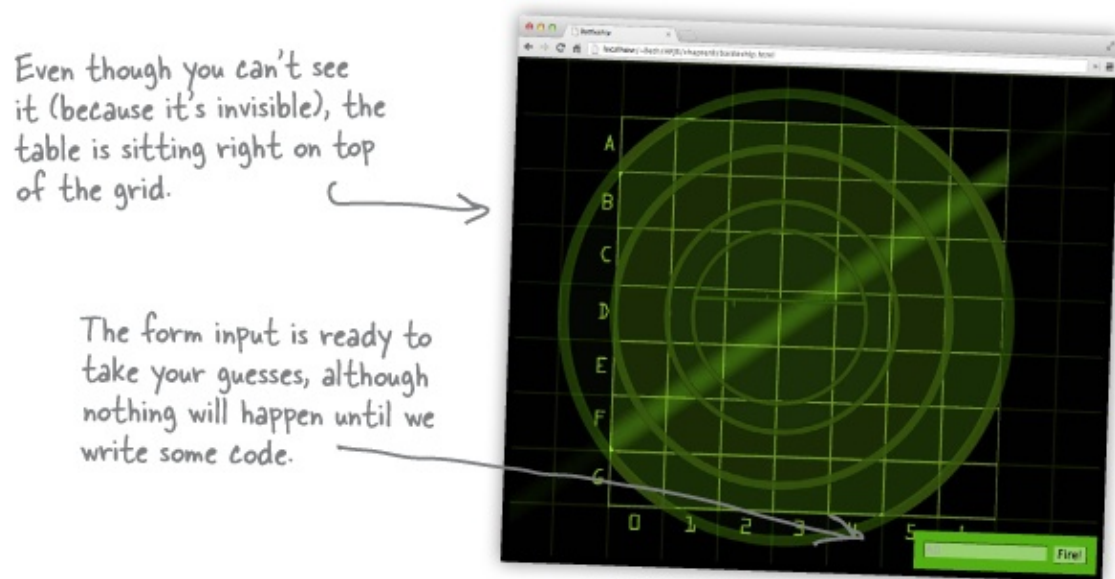
Each <td> gets a specific width and height so that the cells of the <table> match up with the cells of the grid.

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!

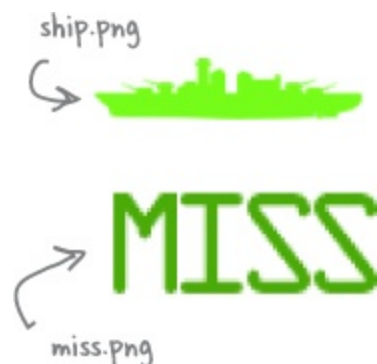


It's time for another game checkpoint. Get all the HTML and CSS entered into your HTML file and then reload the page in your browser. Here's what you should see:



## 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.



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:

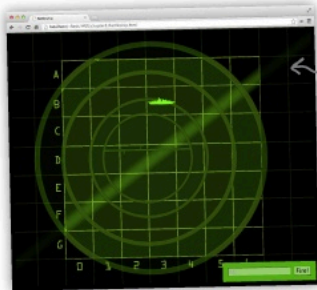
```
.hit {  
    background: url("ship.png") no-repeat center center;  
}  
.miss {  
    background: url("miss.png") no-repeat center center;  
}
```

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.

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.





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

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

---

**NOTE**

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.

---

---



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

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

**A:** *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:** 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:** 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

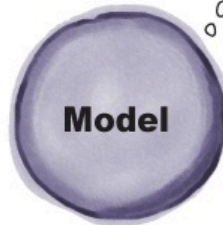
With the HTML and CSS out of the way, let's get to the real game design. Back in [Chapter 2](#), we hadn't covered functions or objects or encapsulation or learned about object-oriented design, so when we built the first version of the Battleship game, we used a procedural design—that is, we designed 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.

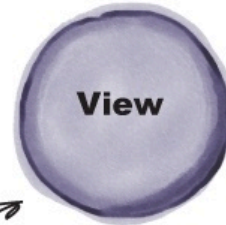


I glue everything together including getting the player's input and executing the game logic.

My job is to keep track of the ships: where they are, if they've been hit, and if they've been sunk.



My job is to keep the display updated with hits, misses and messages for the user.



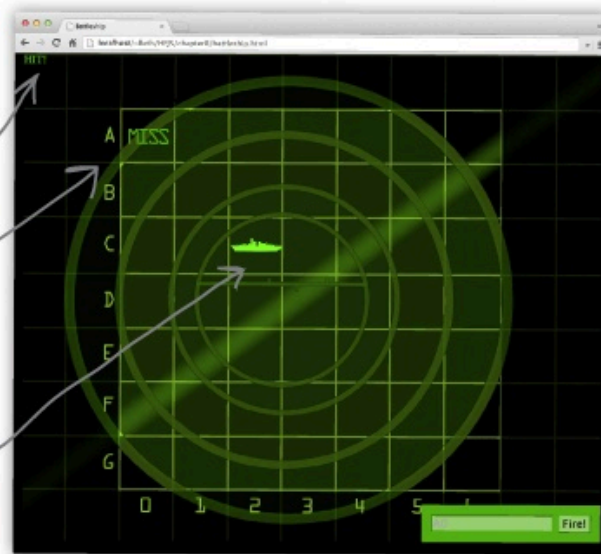
## EXERCISE

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:*

Here's a message.  
Messages will be things like  
"HIT!", "You missed." and  
"You sank my battleship!"

Here the display has  
a MISS placed on  
the grid.

And here the display has a  
ship placed on the grid.



```
var 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!  
  }  
};
```

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

Your methods go here!

```
};
```

## Implementing the View

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 indi-



cates 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`:

---

NOTE

If not, shame on you. Do it now!

---

Here's our view object.

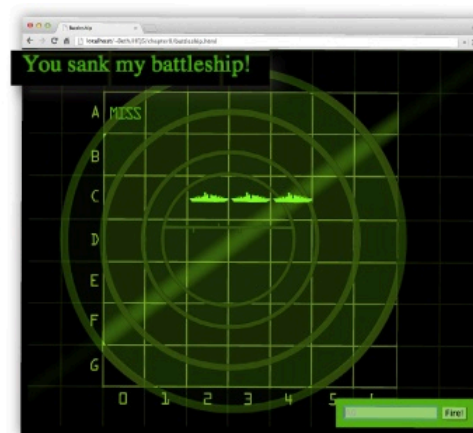
```
var view = {  
  displayMessage: function(msg) {  
    },  
  displayHit: function(location) {  
    },  
  displayMiss: function(location) {  
    }  
};
```

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.

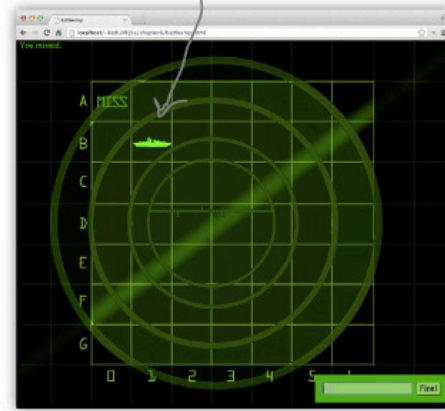
Now before we test this code, let's go ahead and write the other two methods. They won't be incredibly complicated methods, and this way we can test the entire object at once.

## 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
```



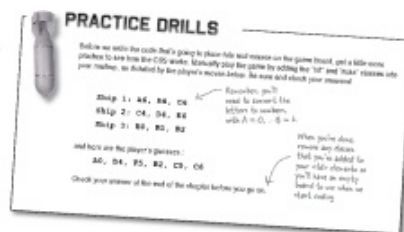
Make sure you add the code for `displayHit` and `displayMiss` to your "battleship.js" file.

## 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:

A0, D4, F5, B2, C5, C6  
↑    ↑    ↑    ↑    ↑    ↑  
MISS HIT MISS HIT MISS HIT



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

`view.displayMiss("00")`



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

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 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:

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.

---

---

---

---

**NOTE**

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:

Here's what all three ships would look like:

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



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

---

**NOTE**

Execute these moves on the game board.

---



---

## 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:

## 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 re-

turn `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:

## 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:

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:

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:

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

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:

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:

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:

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.

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

---

## 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:



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

## **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:



---

## 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.)

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

---

**Q: 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: 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: 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 administra-

tive 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, `process-Guess`, that takes an alphanumeric 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:

# 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:

---

## NOTE

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.

---

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:

---

## NOTE

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

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

- 4- Now we'll handle checking both characters of the guess to see if
- 5 they are numbers between zero and six (in other words, to

make sure they are both valid positions on the board).

---

#### BRAIN POWER

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.





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:

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.

---

**NOTE**

We'll tackle these next.

---

- Determine when the game is over (that is, when all ships have been sunk).

## 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...

## 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.



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.)

---

**BRAIN POWER**

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?

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.



# 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:

## 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:

# 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:

---

## 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.



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:

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:



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.

---

**NOTE**

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

---

---

## 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.

## 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:



## 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:

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:

## 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.

## 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.

---

### NOTE

Look back at [How to place ships](#) 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.

---

#### BRAIN POWER

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!



---

## 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!

---

## 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!





- 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.

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 corresponding `<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.



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.

```
.hit {  
    background: url("ship.png") no-repeat center center;  
}  
.miss {  
    background: 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: "34", "12", and "26", and the ids for the .miss class are "00", "25", and "55". To add a class to an element, you use the class attribute, like this:

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

---

---

#### EXERCISE SOLUTION

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:

---

---

#### 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.

---

**NOTE**

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.

---

---

---

---

---

**NOTE**

Or write in your own answer.

---



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

---

**NOTE**

Execute these moves on the game board.

---

And here's our solution:



---

## 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: The player guesses "D4", does that hit a ship?\_\_\_\_\_ If so, which one? \_\_\_\_\_

```
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.

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

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

---

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.

