Chapter 2. Writing Real Code: *Going further*

You already know about variables, types, expressions... we could go on. The point is, you already know a few things about JavaScript. In fact, you know enough to write some **real code**. Some code that does something interesting, some code that someone would want to use. What you're lacking is the **real experience** of writing code, and we're going to remedy that right here and now. How? By jumping in head first and coding up a casual game, all written in JavaScript. Our goal is ambitious, but we're going to take it one step at a time. Come on, let's get this started, and if you want to launch the next startup, we won't stand in your way; the code is yours.

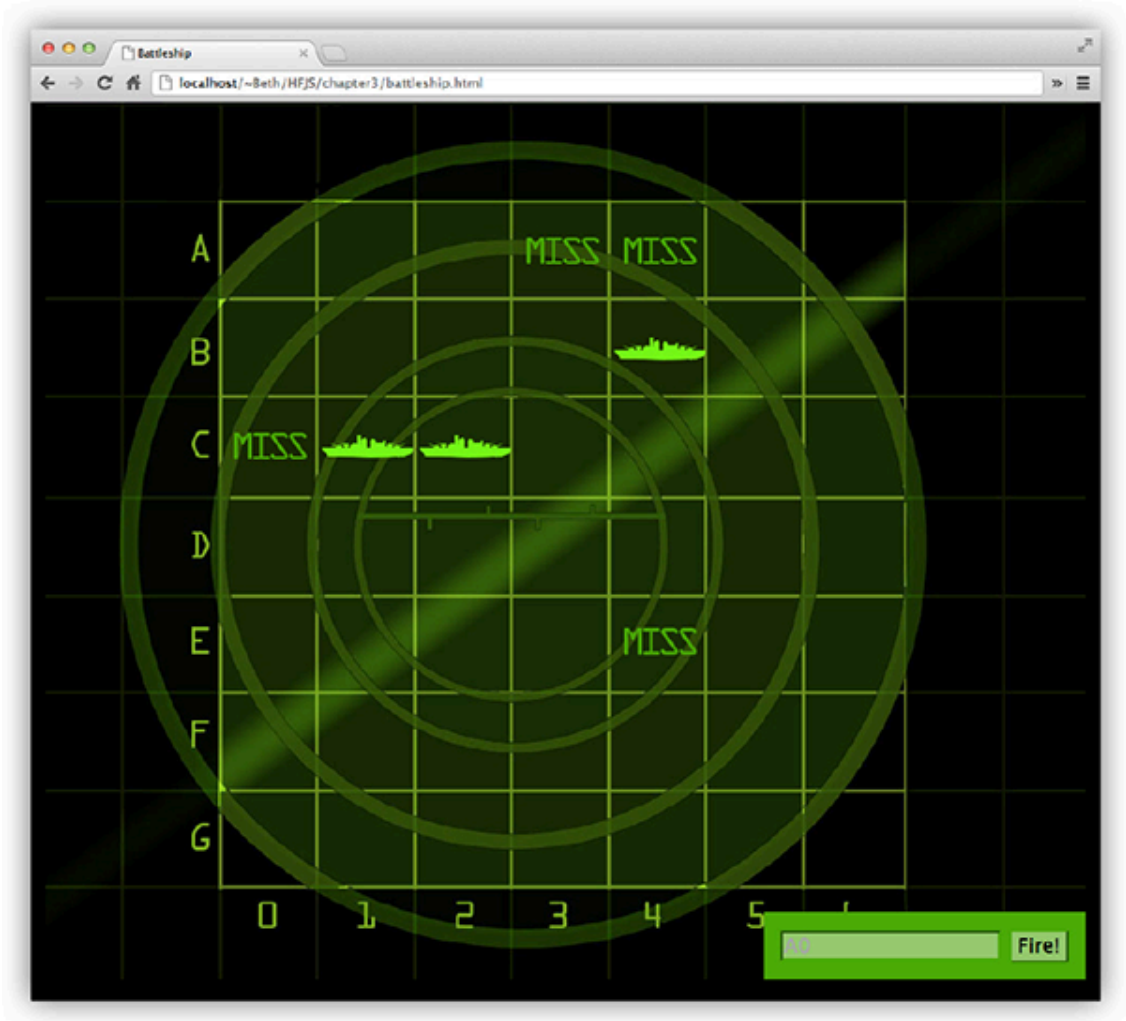
Let’s build a Battleship game

It’s you against the browser: the browser hides ships and your job is to seek them out and destroy them. Of course, unlike the real Battleship game, in this one you don’t place any ships of your own. Instead, your job is to sink the computer’s ships in the fewest number of guesses.

Goal: Sink the browser’s ships in the fewest number of guesses. You’re given a rating, based on how well you perform.

Setup: When the game program is launched, the computer places ships on a virtual grid. When that’s done, the game asks for your first guess.

How you play: The browser will prompt you to enter a guess, and you’ll type in a grid location. In response to your guess, you’ll see a result of “HIT!” “MISS” or “You sank my battleship!” When you sink all the ships, the game ends by displaying your rating.



Here's what we're shooting for: a nice 7x7 grid with three ships to hunt down. Right now we're going to start a little simpler, but once you know a bit more JavaScript we'll complete the implementation so it looks just like this, complete with graphics and everything...we'll leave the sound to you as extra credit.

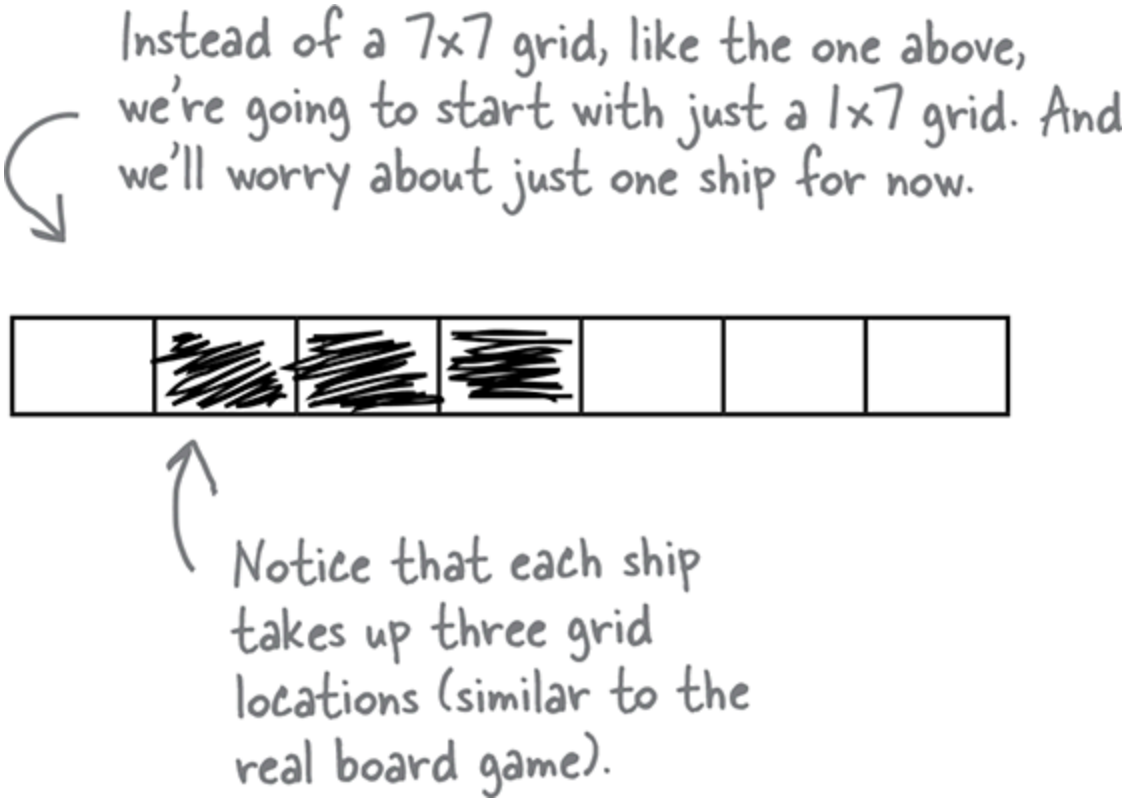
Our first attempt...

...a simplified Battleship

For our first attempt, we're going to start simpler than the full-blown 7 by 7 graphical version with three ships. Instead, we're going to start with a nice 1D grid with seven locations and one ship to find. It will be crude,

but our focus is on designing the basic code for the game, not the look and feel (at least for now).

Don't worry; by starting with a simplified version of the game, you get a big head start on building the full game later. This also gives us a nice chunk to bite off for your first real JavaScript program (not counting the serious business application from Chapter 1—99 bottles of rootbeer on the wall—of course). So, we'll build the simple version of the game in this chapter, and we'll get to the deluxe version later in the book, after you've learned a bit more about JavaScript.



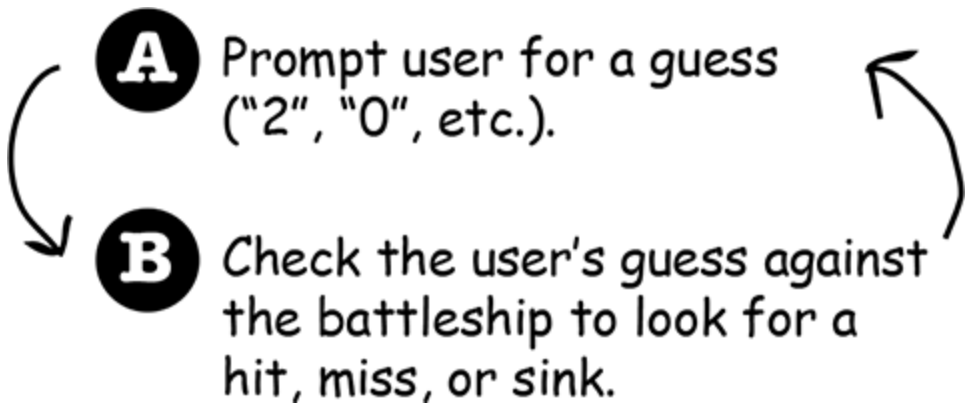
First, a high-level design

We know we'll need variables, and some numbers and strings, and if statements, and conditional tests, and loops...but where, and how many? And how do we put all this together? To answer these questions, we need more information about what the game should do.

First, we need to figure out the general flow of the game. Here's the basic idea:

1. **User starts the game.**
    A. **Game places a battleship at a random location on the grid.**
2. **Gameplay begins.**

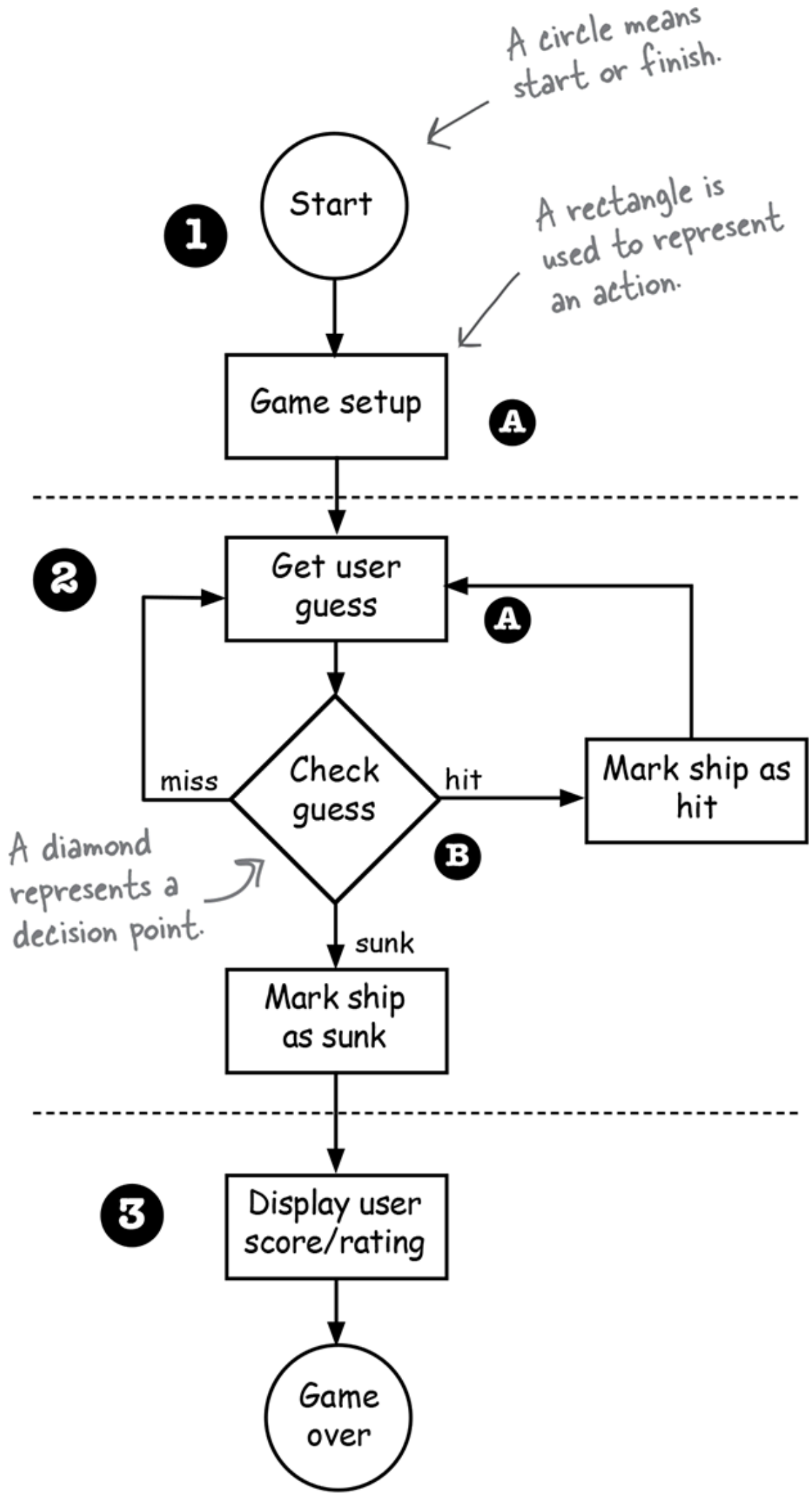
Repeat the following until the battleship is sunk:



3. **Game finishes.**

Give the user a rating based on the number of guesses.

Now we have a high-level idea of the kinds of things the program needs to do. Next, we'll figure out a few more details for the steps.


A circle means start or finish.
Start
1
A rectangle is used to represent an action.
Game setup
A
2
Get user guess
A
Check guess
B
miss
hit
Mark ship as hit
A diamond represents a decision point.
sunk
Mark ship as sunk
3
Display user score/rating
Game over

Whoa. A real flowchart.

A few more details...

We have a pretty good idea about how this game is going to work from the high-level design and professional-looking flowchart, but let's nail down just a few more of the details before we begin writing the code.

Representing the ships

For one thing, we can start by figuring out how to represent a ship in our grid. Keep in mind that the virtual grid is...well, *virtual*. In other words, it doesn't exist anywhere in the program. As long as both the game and the user know that the battleship is hidden in three consecutive cells out of a possible seven (starting at zero), the row itself doesn't have to be represented in code. You might be tempted to build something that holds all seven locations and then to try to place the ship in those locations. But we don't need to. We just need to know the cells where the ship is located, say, at cells 1, 2, and 3.

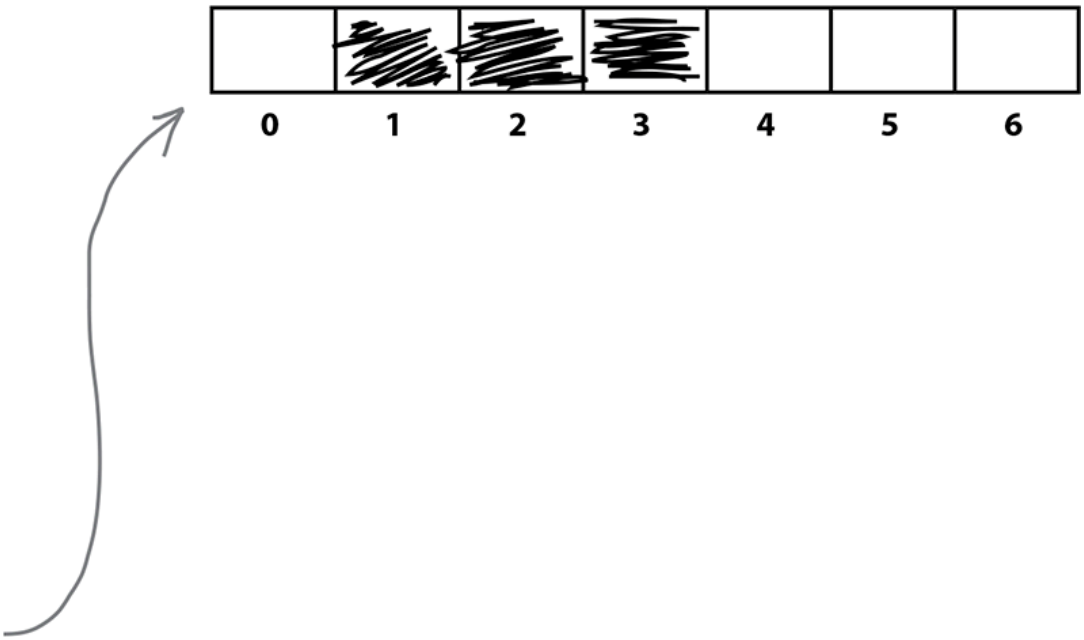
Getting user input

What about getting user input? We can do that with the `prompt` function. Whenever we need to get a new location from the user, we'll use `prompt` to display a message and get the input, which is just a number between 0 and 6, from the user.

Displaying the results

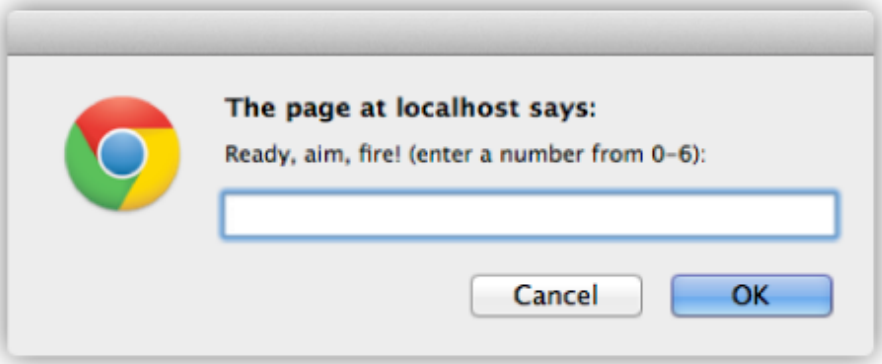
What about output? For now, we'll continue to use `alert` to show the output of the game. It's a bit clunky, but it'll work. (For the real game, later in the book, we'll be updating the web page instead, but we've got a way to go before we get there.)

1 **Game starts**, creates one battleship, and gives it a location on three cells in the single row of seven cells.
The locations are just integers; for example, 1, 2, 3 are the cell locations in this picture:



2 **Gameplay begins**. Prompt user for a guess:

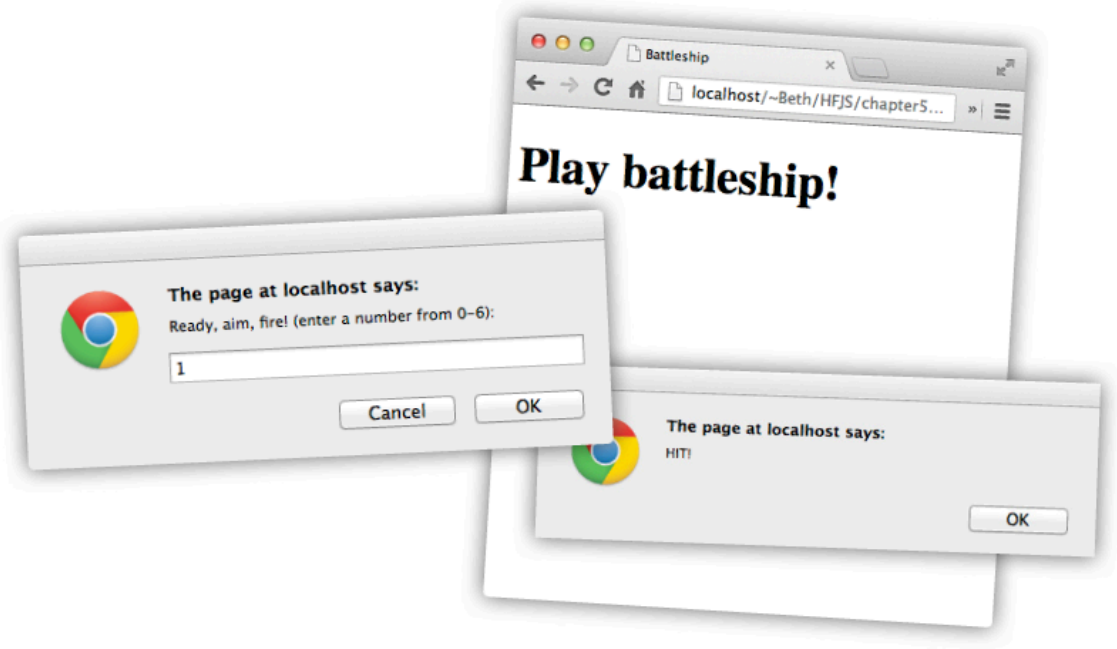
A



B **Check to see if user's input hit any of the ship's three cells. Keep track of how many hits there are in a variable.**

3 **Game finishes** when all three cells have been hit and our number of hits variable's value is 3. We tell the user how many guesses it took to sink the ship.

Sample game interaction



Working through the pseudocode

We need an approach to planning and writing our code. We're going to start by writing *pseudocode.* Pseudocode is halfway between real JavaScript code and a plain English description of the program, and as you'll see, it will help us think through how the program is going to work without having to fully develop the *real code*.

In this pseudocode for Simple Battleship, we've included a section that describes the variables we'll need, and a section describing the logic of the program. The variables will tell us what we need to keep track of in our code, and the logic describes what the code has to faithfully implement to create the game.

DECLARE three *variables* to hold the locations of each cell of the ship. Let's call them `location1`, `location2,` and `location3`.

DECLARE a *variable* to hold the user's current guess. Let's call it `guess`.

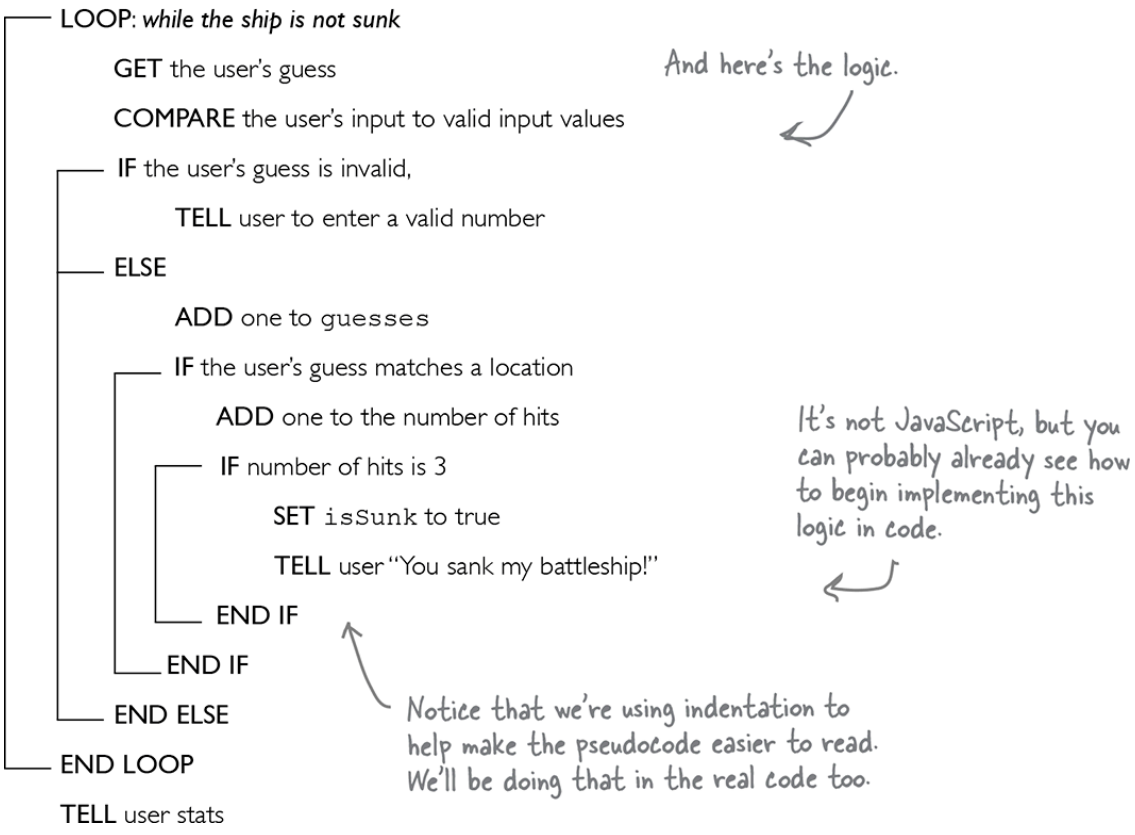
DECLARE a *variable* to hold the number of hits. We'll call it `hits` and *set* it to 0.

DECLARE a *variable* to hold the number of guesses. We'll call it `guesses` and *set* it to 0.

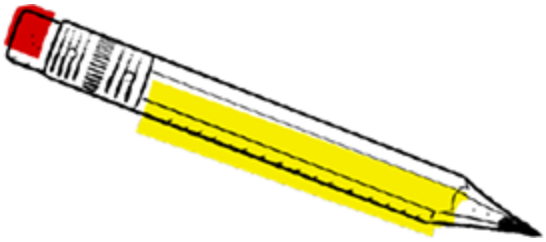
DECLARE a *variable* to keep track of whether the ship is sunk or not. Let's call it `isSunk` and *set* it to `false`.

NOTE

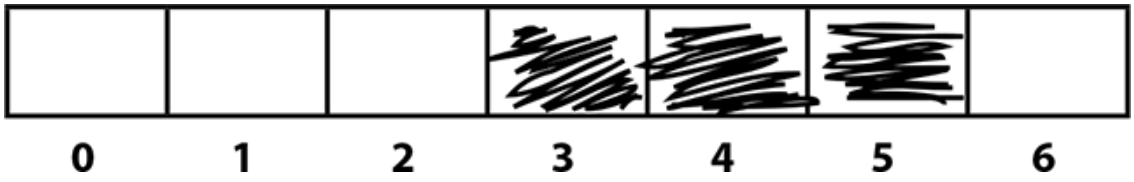
The variables we need.



SHARPEN YOUR PENCIL



Let's say our virtual grid looks like this:



And we've represented the ship's location using our location variables, like this:

```
location1 = 3;
location2 = 4;
location3 = 5;
```

Use the following sequence as your test user input:

```
1, 4, 2, 3, 5
```

Now, using the pseudocode on the previous page, walk through each step of the code and see how this works given the user input. Put your notes below. We've begun the exercise for you. If this is your first time walking through pseudocode, take your time and see how it all works.

NOTE

If you need a hint, take a quick peek at our answer at the end of the chapter.

location1	location2	location3	guess	guesses	hits	isSunk
3	4	5	——	0	0	false
3	4	5	1	1	0	false

The first row shows the initial values of the variables, before the user enters their first guess. We're not initializing the variable guess, so its value is undefined.

Solution in "Sharpen your pencil Solution"

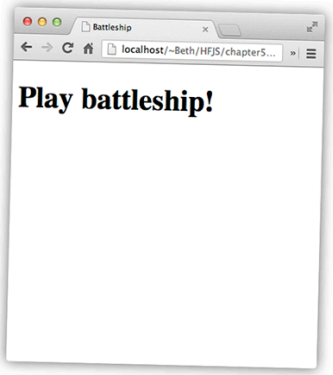
Oh, before we go any further, don't forget the HTML!

You're not going to get very far without some HTML to link to your code. Go ahead and type the markup below into a new file named "battleship.html". After you've done that, we'll get back to writing code.

The HTML for the Battleship game is super simple; we just need a page that links to the JavaScript code, and that's where all the action happens.

```
<!doctype html>
<html lang="en">
  <head>
    <title>Battleship</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Play battleship!</h1>
    <script src="battleship.js"></script>
  </body>
</html>
```

We're linking to the JavaScript at the bottom of the <body> of the page, so the page is loaded by the time the browser starts executing the code in "battleship.js".



Here's what you'll see when you load the page. We need to write some code to get the game going!

BRAIN POWER

Flex those dendrites.

This is thinking ahead a bit, but what kind of code do you think it would take to generate a random location for the ship each time you load the page? What factors would you have to take into account in the code to correctly place a ship? Feel free to scribble some ideas here.

Writing the Simple Battleship code

We're going to use the pseudocode as a blueprint for our real JavaScript code. First, let's tackle all the variables we need. Take another look at our pseudocode:

DECLARE three *variables* to hold the locations of each cell of the ship. Let's call them `location1`, `location2`, and `location3`.

NOTE

We need three variables to hold the ship's location.

DECLARE a *variable* to hold the user's current guess. Let's call it `guess`.

DECLARE a *variable* to hold the number of hits. We'll call it `hits` and *set* it to 0.

DECLARE a *variable* to hold the number of guesses. We'll call it `guesses` and *set* it to 0.

NOTE

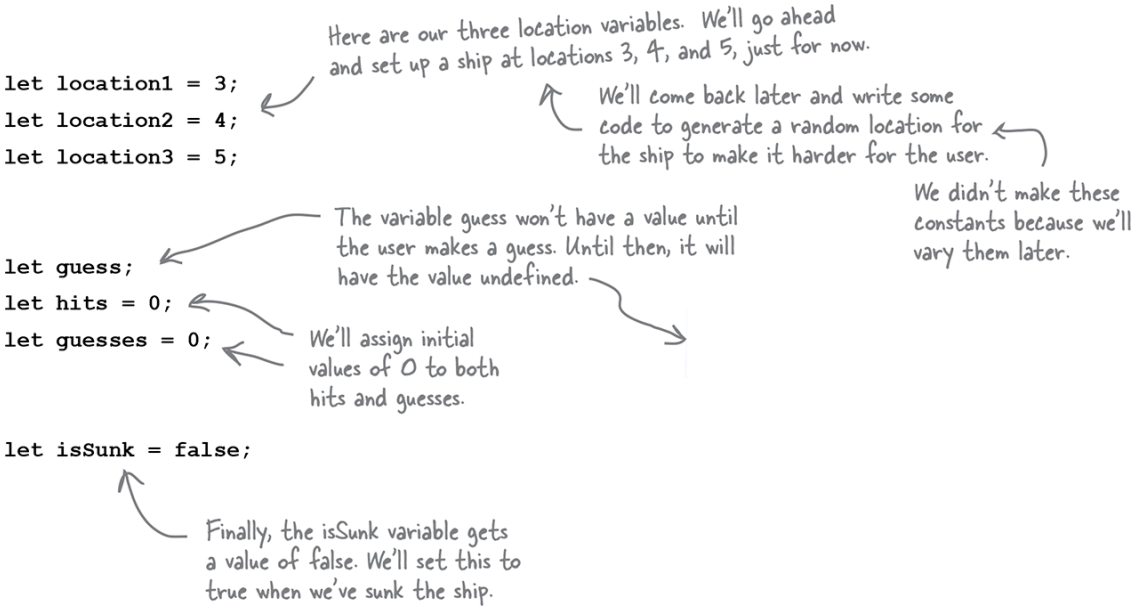
And three more (guess, hits, and guesses) to deal with the user's guess.

DECLARE a *variable* to keep track of whether the ship is sunk or not. Let's call it `isSunk` and *set* it to `false`.

NOTE

And another to track whether or not the ship is sunk.

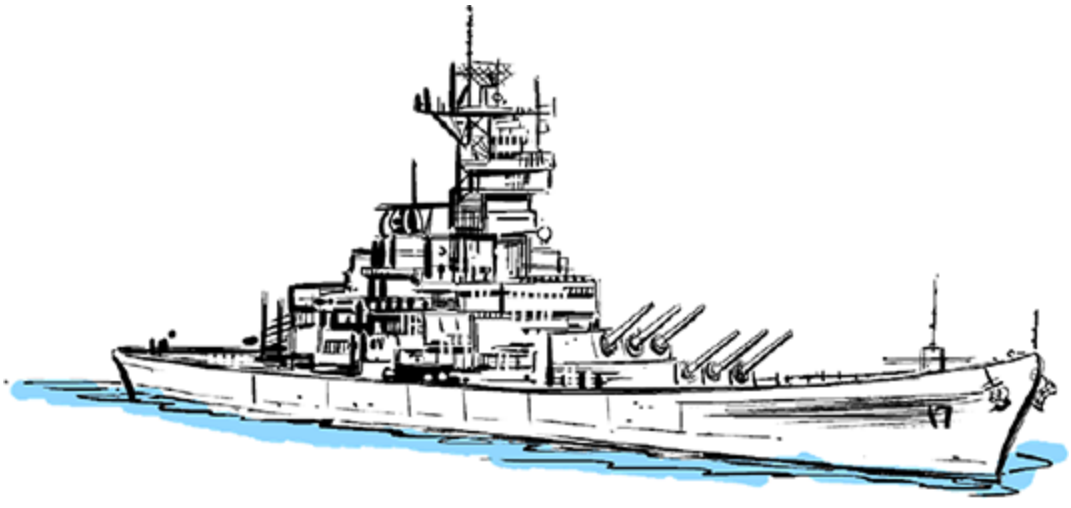
Let's get these variables into a JavaScript file. Create a new file named "battleship.js" in the same folder as "battleship.html" and type in your variable declarations like this:

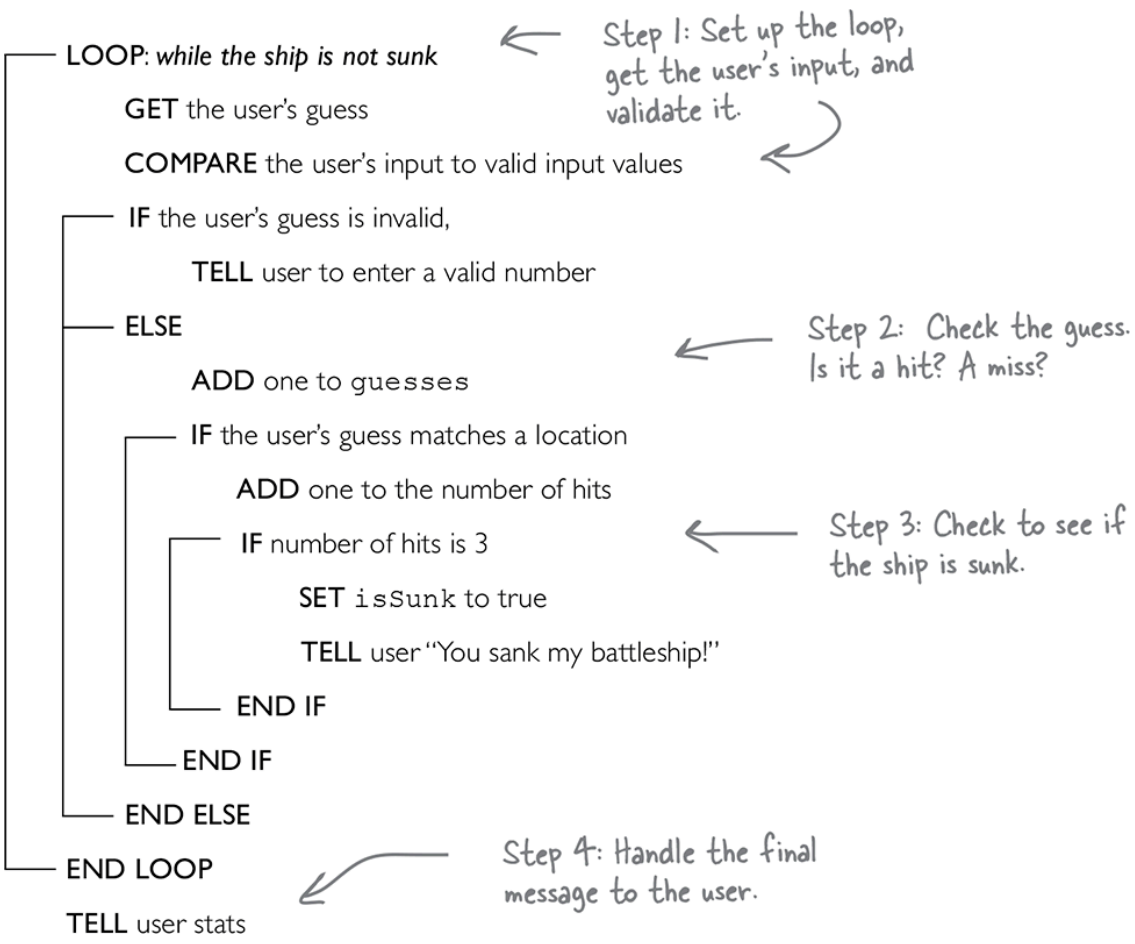


If you don't provide an initial value for a variable, then JavaScript gives it a default value of `undefined`. Think of the value `undefined` as JavaScript's way of saying "this variable hasn't been given a value yet." We'll be talking more about undefined and some other strange values a little later.

Now let's write the game logic

We've got the variables out of the way, so let's dig into the actual pseudocode that implements the game. We'll break this into a few pieces. The first thing you're going to want to do is implement the loop: it needs to keep looping while the ship isn't sunk. From there, we'll take care of getting the guess from the user and validating it—you know, making sure it really is a number between 0 and 6—and then we'll write the logic to check for a hit on the ship and to see if the ship is sunk. Finally, we'll create a little report for the user with the number of guesses it took to sink the ship.





TO DO:

- ☐ Create loop and get user guess
- ☐ Check user guess
- ☐ Check if ship has been sunk
- ☐ Display stats to user

Step 1: Setting up the loop, getting some input

Now we're going to begin to translate the logic of our game into actual JavaScript code. There isn't a perfect mapping from pseudocode to JavaScript, so you'll see a few adjustments here and there. The pseu-

docode gives us a good idea of *what* the code needs to do, and now we have to write the JavaScript code that can do the *how*.

Let's start with all the code we have so far and then we'll zero in on just the parts we're adding (to save a few trees here and there, or electrons if you're reading the digital version of the book):

DECLARE variables

```
let location1 = 3;
let location2 = 4;
let location3 = 5;
let guess;
let hits = 0;
let guesses = 0;
let isSunk = false;
```

We've already covered these, but we're including them here for completeness.

Here's the start of the loop. While the ship isn't sunk, we're still in the game, so keep looping.

Remember, while uses a conditional test to determine whether to keep looping. In this case we're testing to make sure that isSunk is still false. We'll set it to true as soon as the ship is sunk.

LOOP: *while the ship is not sunk*

```
while (isSunk == false) {
```

GET the user's guess

```
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");

}
```

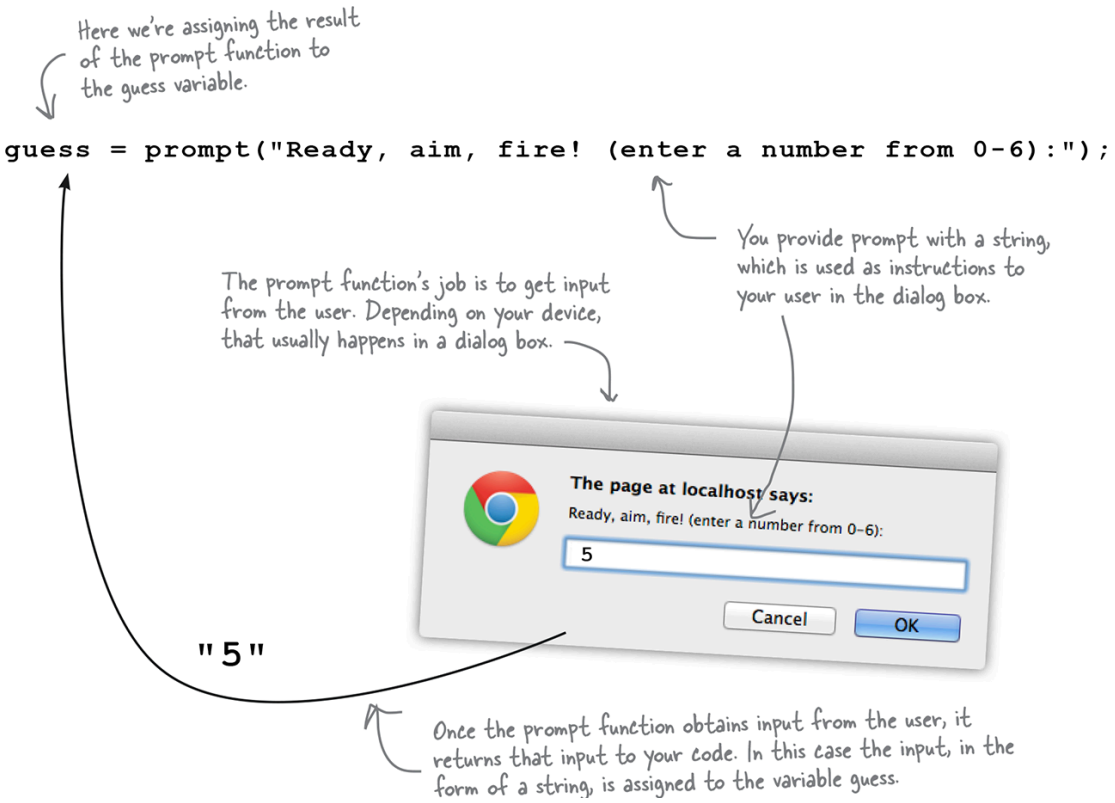
Each time we go through the while loop, we're going to ask the user for a guess. To do that, we use the prompt built-in function. More on that on the next page...

BRAIN POWER

If you ran this code now, would the game ever end?

How prompt works

The browser provides a built-in function you can use to get input from the user, named `prompt`. The `prompt` function is a lot like the `alert` function you've already used—`prompt` causes a dialog to be displayed with a string that you provide, just like `alert`—but it also gives the user a place to type a response. That response, in the form of a string, is then returned as a result of calling the function. Now, if the user cancels the dialog, then `null` is returned instead.



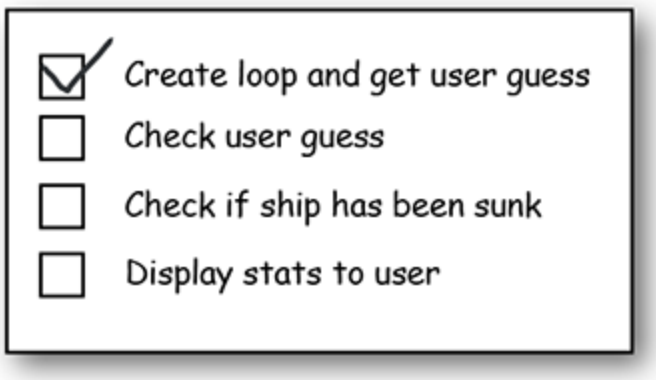
WATCH IT!

You might be tempted to try this code now...

...but don't. If you do, your browser will start an *infinite loop* of asking you for a guess, and then asking you for a guess, and so on, without any means of stopping the loop (other than using your operating system to force the browser process to stop).

Step 2: Checking the user's guess

If you look at the pseudocode, you'll see that to check the user's guess, we need to first make sure the user has entered a valid input. If so, then we also check to see if the guess was a hit or a miss. And, we'll want to make sure we appropriately update the `guesses` and `hits` variables. Let's get started by checking the validity of the user's input, and, if the input is valid, we'll increment the `guesses` variable. After that, we'll write the code to see if the user has a hit or a miss.



```
// Variable declarations go here

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
       alert("Please enter a valid cell number!");
    } else {
       guesses = guesses + 1;
    }
}
```

We check validity by making sure the guess is between zero and six.

If the guess isn't valid, we'll tell the user with an alert.

And if the guess is valid, go ahead and add one to guesses so we can keep track of how many times the user has guessed.

Let's look a little more closely at the validity test. You know we're checking to see that the guess is between zero and six, but how exactly does this conditional test that? Let's break it down:

Try to read this like it's English: this conditional is true if the user's guess is less than zero OR the user's guess is greater than six. If either is true, then the input is invalid.

```
if (guess < 0 || guess > 6) {
```

This is really just two small tests put together. The first test checks if guess is less than zero.

And this one checks to see if guess is greater than six.

And this, which we call the OR operator, combines the two tests so that if either test is true, then the entire conditional is true. If both tests are false, then the statement is false, and the guess is between zero and six, which means it's valid.

THERE ARE NO DUMB QUESTIONS

Q: I noticed there is a Cancel button on the prompt dialog box. What gets returned from the prompt function if the user hits Cancel?

A: If you click Cancel in the prompt dialog box, then prompt returns the value null rather than a string. null means "no value," which is appropriate in this case because you've canceled without entering a value. We can use the fact that the value returned from prompt is null to check to see if the user clicked Cancel, and if they did, then we could, say, end the game. We're not doing that in our code, but keep this idea in the back of your mind, as we might use it later in the book.

Q: You said that prompt always returns a string. So how can we compare a string value, like "0" or "6", to numbers, like 0 and 6?

A: In this situation, JavaScript tries to convert the string in guess to a number in order to do the comparisons, guess < 0 and guess > 6. As long as you enter only a number, like 4, JavaScript knows how to convert the string "4" to the number 4 when it needs to. We'll come back to the topic of type conversion in more detail later.

Q: What happens if the user enters something that isn't a number into the prompt? Like "six" or "quit"?

A: In that case, JavaScript won't be able to convert the string to a number for the comparison. So, you'd be comparing "six" to 6 or "quit" to 6, and that kind of comparison will return false, which will lead to a MISS. In a more robust version of Battleship, we'll check the user input more carefully and make sure they've entered a number first.

Q: With the OR operator, is it true if only one or the other is true, or can both be true?

A: Yes, both can be true. The result of the OR operator (||) is true if either of the tests is true, or if both are true. If both are false, then the result is false.

Q: Is there an AND operator?

A: Yes! The AND operator (&&) works similarly to OR, except that the result of AND is true only if both tests are true.

Q: What's an infinite loop?

A: Great question. An infinite loop is one of the many problems that plague programmers. Remember that a loop requires a conditional test, and the loop will continue as long as that conditional test is true. If your code never does anything to change things so that the conditional test is false at some point, the loop will continue forever. And ever. Until you kill your browser or reboot.

TWO-MINUTE GUIDE TO BOOLEAN OPERATORS

A boolean operator is used in a boolean expression, which results in a true or false value. There are two kinds of boolean operators: comparison operators and logical operators.

Comparison Operators

Comparison operators compare two values. Here are some common comparison operators:

<	means "less than"
>	means "greater than"
==	means "equal to"
===	means "exactly equal to" (we'll come back to this one later)
<=	means "less than or equal to"
>=	means "greater than or equal to"
!=	means "not equal to"

Logical Operators

Logical operators combine two boolean expressions to create one boolean result (true or false). Here are two logical operators:

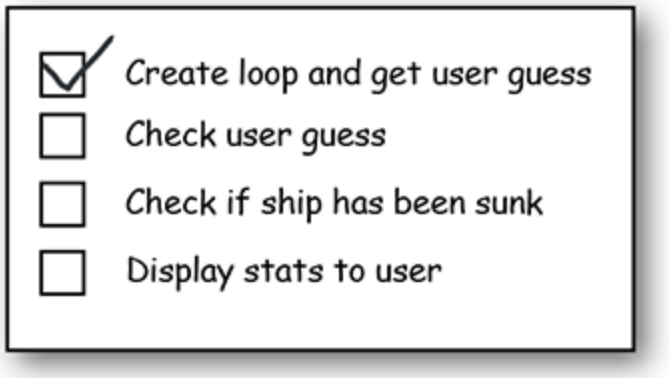
||	means OR. Results in true if *either* of the two expressions is true.
&&	means AND. Results in true if *both* of the two expressions are true.

Another logical operator is NOT, which acts on one boolean expression (rather than two):

!	means NOT. Results in true if the expression evaluates to false.

So, do we have a hit?

This is where things get interesting—the user's taken a guess at the ship's location, and we need to write the code to determine if that guess has hit the ship. More specifically, we need to see if the guess matches one of the locations of the ship. If it does, then we'll increment the `hits` variable.



Here's a first stab at writing the code for the hit detection—let's step through it:

```
if (guess == location1) {
     hits = hits + 1;
} else  if (guess == location2) {
     hits = hits + 1;
} else if (guess == location3) {
     hits = hits + 1;
}
```

If the guess is location1, then we hit the ship, so increment the hits variable by one.

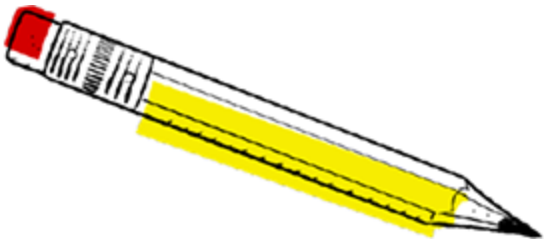
Otherwise, if the guess is location2, then do the same thing.

Finally, if the guess is location3, we increment the hits variable.

And if none of these are true, then the hits variable is never incremented.

Notice we're using indentation for the code in each if/else block. This makes your code easier to read, especially when you've got lots of blocks nested inside blocks.

SHARPEN YOUR PENCIL

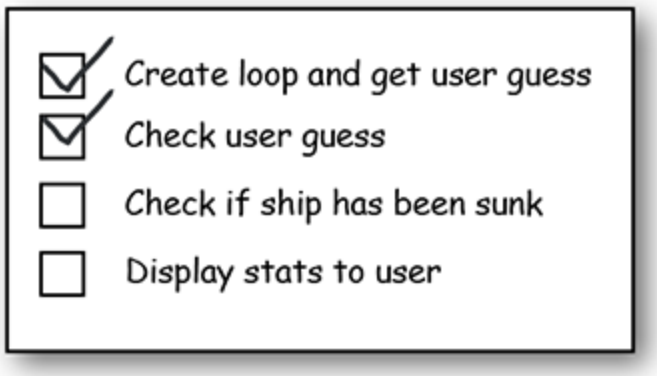


What do you think of this first attempt to write the code to detect when a ship is hit? Does it look more complex than it needs to be? Are we repeating code in a way that seems a bit, well, redundant? Could we simplify it? Using what you know of the || operator (that is, the boolean OR operator), can you simplify this code? Make sure you check your answer at the end of the chapter before moving on.

Solution in "Sharpen your pencil Solution"

Adding the hit detection code

Let's put everything together from the previous couple of pages:




```
// Variable declarations go here

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
      alert("Please enter a valid cell number!");
    } else {
      guesses = guesses + 1;
    }
    if (guess == location1 || guess == location2 || guess == location3) {
        hits = hits + 1;
    }
  }
}
```

LOOP: *while the ship is not sunk*

GET the user's guess

← Check the user's guess...

ADD one to guesses

The user's guess looks valid, so let's increase the number of guesses by one.

IF the user's guess matches a location

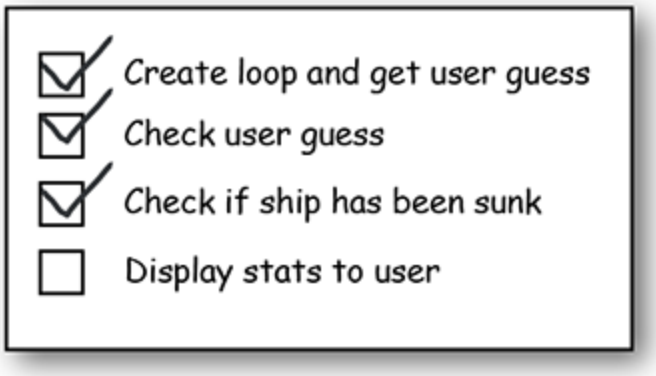
ADD one to the number of hits

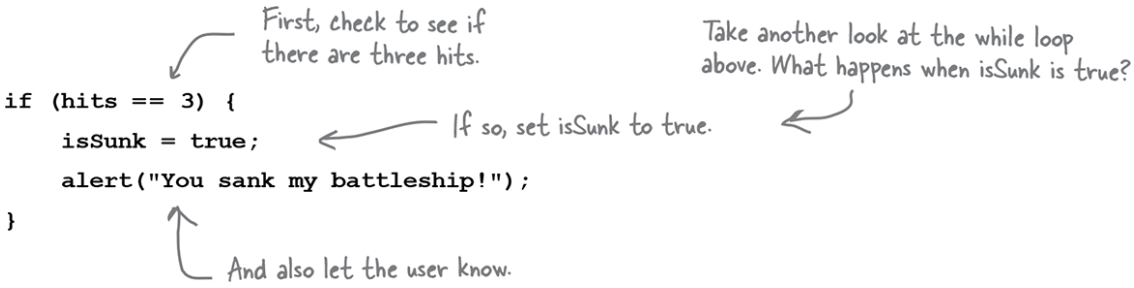
If the guess matches one of the ship's locations, we increment the hits counter.

We've combined the three conditionals into one if statement using || (OR). So read it like this: "If guess is equal to location1 OR guess is equal to location2 OR guess is equal to location3, increment hits."

Step 3: Hey, you sank my battleship!

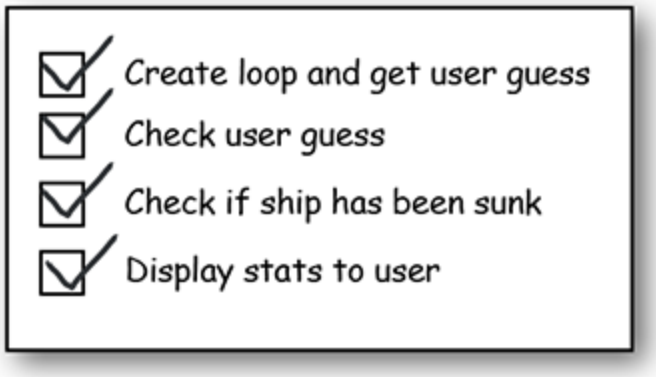
We're nearly there; we've almost got this game logic nailed down. Looking at the pseudocode again, what we need to do now is test to see if we have three hits. If we do, then we've sunk a battleship. And if we've sunk a battleship, then we need to set `isSunk` to true and also tell the user they've destroyed a ship. Let's sketch out the code again before adding it in:





Step 4: Provide some post-game analysis

After `isSunk` is set to true, the while loop is going to stop looping. That's right, this program we've come to know so well is going to stop executing the body of the while loop, and before you know it the game's going to be over. But we still owe the user some stats on how they did. Here's some code that does that:




```
let stats = "You took " + guesses + " guesses to sink the battleship, " +
            "which means your shooting accuracy was " + (3/guesses);
```

```
  alert(stats);
```

NOTE

Here we're creating a string that contains a message to the user including the number of guesses they took, along with the accuracy of their shots. Notice that we're splitting up the string into pieces (to insert the variable guesses, and also to fit the string into multiple lines) using the concatenation operator, +. For now just type this as is, and we'll explain more about this later.

Now let's add this and the sunk ship detection into the rest of the code:

EXERCISE

Remember we said pseudocode often isn't perfect? Well, we actually left something out of our original pseudocode: we're not telling the user if her guess is a HIT or a MISS. Can you insert these pieces of code in the proper place to correct this?

```
// Variable declarations go here

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;
        if (guess == location1 || guess == location2 || guess == location3) {
            hits = hits + 1;
            if (hits == 3) {
              isSunk = true;
              alert("You sank my battleship!");
            }
        }
    }
}
let stats = "You took " + guesses + " guesses to sink the battleship, " +
               "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

NOTE

This is a lot of curly braces to match. If you're having trouble matching them, just draw lines right in the book to match them up.

Solution in “Exercise Solution”

And that completes the logic!

Alright! We’ve now fully translated the pseudocode to actual JavaScript code. We even discovered something we left out of the pseudocode, and we’ve got that accounted for too. Below, you’ll find the code in its entirety. Make sure you have this typed in and saved in “battleship.js”:

```
let location1 = 3;
let location2 = 4;
let location3 = 5;
let guess;
let hits = 0;
let guesses = 0;
let isSunk = false;

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;

        if (guess == location1 || guess == location2 || guess == location3) {
            alert("HIT!");
            hits = hits + 1;
            if (hits == 3) {
                isSunk = true;
                alert("You sank my battleship!");
            }
        } else {
           alert("MISS");
        }
    }
}
let stats = "You took " + guesses + " guesses to sink the battleship, " +
```

```
                "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

Doing a little quality assurance

Quality assurance, or QA, is the process of testing software to find defects. We're going to do a little QA on our code. When you're ready, load "battleship.html" in your browser and start playing. Try some different things. Is it working perfectly? Or did you find some issues? If so, list them here. You can see our test run on this page too.

Boolean operators allow you to write more complex statements of logic.

You've seen enough conditionals to know how to test, say, if the temperature is greater than 32 degrees. Or, that a variable that represents whether an item is inStock is true. But sometimes we need to test more. Sometimes we need to know not only if a value is greater than 32, but also if it's less than 100. Or, if an item is inStock and also onSale. Or that an item is on sale only on Tuesdays when the user is a VIP member. So, you see, these conditionals can get complex.

Let's step through a few to get a better idea of how they work.

Say we need to test that an item is inStock AND onSale. We could do that like this:

We can simplify this code by combining these two conditionals together. Unlike in Simple Battleship, where we tested if guess < 0 OR guess > 6,

here we want to know if inStock is true AND onSale is true. Let's see how to do that...

We don't have to stop there—we can use multiple boolean operators to combine conditionals in a variety of ways:

SHARPEN YOUR PENCIL

We've got a whole bunch of boolean expressions that need evaluating below. Fill in the blanks, and then check your answers at the end of the chapter before you go on.

```
let temp = 81;
let willRain = true;
let humid = (temp > 80 && willRain == true);
```

What's the value of **humid**? _____

```
let guess = 6;
let isValid = (guess >= 0 && guess <= 6);
```

What's the value of **isValid**? _____

```
let kB = 1287;
let tooBig = (kB > 1000);
let urgent = true;
let sendFile =
     (urgent == true || tooBig == false);
```

What's the value of **sendFile**? _____

```
let keyPressed = "N";
let points = 142;
let level;
if (keyPressed == "Y" ||
    (points > 100 && points < 200)) {
   level = 2;
} else {
   level = 1;
}
```

What's the value of **level**? _____

Solution in "Sharpen your pencil Solution"

EXERCISE

Bob and Bill, both from accounting, are working on a new price checker application for their company's website. They've both written if/else statements using boolean expressions. Both are sure they've written the correct code. Which accountant is right? Should these accountants even be writing code? Check your answer at the end of the chapter before you go on.

Solution in "Exercise Solution"

Can we talk about your verbosity...

We don't know how to bring this up, but you've been a little verbose in specifying your conditionals. What do we mean? Take this condition, for instance:

As it turns out, that's a bit of overkill. The whole point of a conditional is that it evaluates to either true or false, but our boolean variable `inStock` already *is* one of those values. So, we don't need to compare the variable to anything; it can just stand on its own. That is, we can just write this instead:

Now, while some might claim our original, verbose version was clearer in its intent, it's more common to see the more succinct version in practice. And you'll find the less verbose version easier to read as well.

EXERCISE

We've got two statements below that use the onSale and inStock variables in conditionals to figure out the value of the variable buyIt. Work through each possible value of inStock and onSale for both statements. Which version is the biggest spender?

Solution in "Exercise Solution"

Finishing the Simple Battleship game

We still have one little matter to take care of, because right now we've hardcoded the location of the ship—no matter how many times you play the game, the ship is always at locations 3, 4, and 5. That actually works out well for testing, but we really need to randomly place the ship to make it a little more interesting for the user.

Let's step back and think about the right way to place a ship on the 1D grid of seven cells. We need a starting location that allows us to place it in three consecutive positions on the grid. That means we need a starting location from zero to four.

How to assign random locations

Once we have a starting location (between zero and four), we can simply use it and the following two locations to hold the ship:

```
let location1 = randomLoc;
let location2 = location1 + 1;
let location3 = location2 + 1;
```

NOTE

Take the random location along with the next two consecutive locations.

Okay, but how do we generate a random number? That's where we turn to JavaScript and its built-in functions. More specifically, JavaScript comes with a bunch of built-in math-related functions, including a couple that can be used to generate random numbers. We're going to get deeper into built-in functions, and functions in general, a little later in the book. For now, we're just going to make use of these functions to get our job done.

The recipe for generating a random number

We're going to start with the `Math.random` function. Calling this function will give us back a random decimal number:

What we need is an integer between 0 and 4—that is, 0, 1, 2, 3 or 4—not a decimal number, like 0.34. How can we get that? To start, we could multiply the number returned by `Math.random` by 5 to get a little closer; here's what we mean...

That’s closer! Now all we need to do is clip off the end of the number to give us an integer number. To do that we can use another built-in Math function, `Math.floor`:

THERE ARE NO DUMB QUESTIONS

Q: If we're trying to generate a number between 0 and 4, why did we multiply the value by 5? Why did we write `Math.floor(Math.random() * 5)`?

A: Good question. Math.random generates a number between 0 and 1, but not including 1. The maximum number you can get from Math.random is 0.999... When you multiply that number by 5, the highest number you'll get is 4.999... Math.floor always rounds a number down, so 1.2 becomes 1, but so does 1.9999. If we generate a number from 0 to 4.999... the rounded-down value will always be between 0 and 4. This is not the only way to do it, and in other languages it's often done differently, but this is how you'll see it done in most JavaScript code.

Q: So if I wanted a random number between 0 and 100 (including 100), I'd instead write `Math.floor(Math.random() * 101)`?

A: That's right! Multiplying by 101 and using Math.floor to round down ensures that your result will be at most 100.

Q: What are the parentheses for in Math.random()?

A: We use parentheses whenever we "call" a function. Sometimes we need to hand a value to a function, like we do when we use alert to display a message, and sometimes we don't, like when we use Math.random. But whenever you're calling a function (whether it's built-in or not), you'll need to use parentheses. Don't worry about this right now; we'll get into all these details in the next chapter.

Q: I can't get my Battleship game to work. I'm not seeing anything in my web page except the "Play battleship" heading. How can I figure out what I did wrong?

A: This is where using the console can come in handy. If you've made an error like forgetting a quote in a string, then JavaScript will typically complain about the syntax of your program not being right, and may even show you the line number where your error is. But some errors are more subtle. For instance, if you mistakenly write isSunk = false instead of is-

Sunk == false, you won't see a JavaScript error, but your code won't behave as you expect it to. For this kind of error, try using console.log to display the values of your variables at various points in your code to see if you can track down the problem.

A little more QA

That's all we need. Put this code together (we've already done that below) and replace your existing location code with it. When you're finished, give it a few test runs to see how fast you can sink the enemy battleship.

Here's one of our test sessions. The game's a little more interesting now that we've got random locations for the ship. But we still managed to get a pretty good score...

EXERCISE

Wait a sec, we noticed something that might be a bug. Hint: when we enter 0, 1, 1, 1, things don't look right! Can you figure out what's happening?

It's a cliff-hanger!

Will we ***find*** the bug?

Will we ***fix*** the bug?

Stay tuned for a much improved version of Battleship a little later in the book...

And in the meantime, see if you can come up with ideas for how you might fix the bug.

QA NOTES

Found a bug! Entering the same number that is a hit on a ship results in sinking the ship, when it shouldn't.

Congrats on your first true JavaScript program, and a short word about

reusing code

You've probably noticed that we've made use of a few *built-in functions* like `alert`, `prompt`, `console.log`, and `Math.random`. With very little effort, these functions have given you the ability to pop up dialog boxes, log output to the console, and generate random numbers, almost like magic. But these built-in functions are just packaged-up code that's already been written for you, and as you can see, their power is that you can use and reuse them just by making a call to them when you need them.

There's a lot to learn about functions, how to call them, what kinds of values you can pass them, and so on, and we're going to start getting into all that in the next chapter, where you'll learn to create your own functions.

But before you get there, you've got the bullet points to review, a crossword puzzle to complete...oh, and a good night's sleep to let everything sink in.

BULLET POINTS

- You can use a flowchart to outline the logic of a JavaScript program, showing decision points and actions.
- Before you begin writing a program, it's a good idea to sketch out what your program needs to do with pseudocode.
- **Pseudocode** is an approximation of what your real code should do.
- There are two kinds of boolean operators: comparison operators and logical operators. When used in an expression, boolean operators result in a true or false value.
- **Comparison** operators compare two values and result in true or false. For example, we can use the boolean comparison operator < ("less than") like this: 3 < 6. This expression results in true.
- **Logical** operators combine two boolean values. For example, true || false results in true; true && false results in false.
- You can generate a random number between 0 and 1 (including 0, but not including 1) using the **Math.random** function.
- The **Math.floor** function rounds down a decimal number to the nearest integer.
- Make sure you use Math with an uppercase M, and not m, when using Math.random and Math.floor.
- The JavaScript function **prompt** shows a dialog with a message and a space for the user to enter a value.
- In this chapter, we used prompt to get input from the user and **alert** to display the results of the Battleship game in the browser.

How does a crossword puzzle help you learn JavaScript? The mental twists and turns burn the JavaScript right into your brain!

ACROSS

1. This helps you think about how a program is going to work.

7. Both while and if statements use _____ tests.

8. Boolean operators always result in true or _____.

9. To get a true value from an AND operator (&&), both parts of the conditional must be _____.

10. JavaScript has many built-in _____, like alert and prompt.

11. To randomly choose a position for a ship, use Math._____.

DOWN

1. To get input from a user, you can use the _____ function.

2. == is a _____ operator you can use to test to see if two values are the same.

3. OR (||) and AND (&&) are _____ operators.

4. If you're good at testing programs, you might want to become a _____ assurance specialist.

5. If you don't initialize a variable, the value is _____.

6. We keep track of whether a ship is sunk or not with a _____ variable.

8. To get a false value from an OR operator (||), both parts of the conditional must be _____.

Solution in “JavaScript cross Solution”

From "Sharpen your pencil"

Let's say our virtual grid looks like this:

And we've represented the ship's location using our location variables, like this:

```
location1 = 3;
location2 = 4;
location3 = 5;
```

Use the following sequence as your test user input:

```
1, 4, 2, 3, 5
```

Now, using the pseudocode in "Working through the pseudocode", trace through each step of the code and see how this works given the user input. Put your notes below. Here's our solution.

location1	location2	location 3	guess	guesses	hits	isSunk
3	4	—	5	0	0	false
3	4	5	1	1	0	false
3	4	5	4	2	1	false
3	4	5	2	3	1	false
3	4	5	3	4	2	false
3	4	5	5	5	3	true

EXERCISE SOLUTION

From "Exercise"

We've got two statements below that use the onSale and inStock variables in conditionals to figure out the value of the variable buyIt. Work through each possible value of inStock and onSale for both statements. Which version is the biggest spender? The OR (||) operator!

From "Sharpen your pencil"

We've got a whole bunch of boolean expressions that need evaluating below. Fill in the blanks. Here's our solution.

```
let temp = 81;
let willRain = true;
let humid = (temp > 80 && willRain == true);
```

What's the value of **humid**?

```
let guess = 6;
let isValid = (guess >= 0 && guess <= 6);
```

What's the value of **isValid**?

```
let kB = 1287;
let tooBig = (kB > 1000);
let urgent = true;
let sendFile =
    (urgent == true || tooBig == false);
```

What's the value of **sendFile**?

```
let keyPressed = "N";
let points = 142;
let level;
if (keyPressed == "Y" ||
    (points > 100 && points < 200)) {
   level = 2;
} else {
```

```
    level = 1;
}
```

What's the value of **level**?

EXERCISE SOLUTION

From "Exercise"

Bob and Bill, both from accounting, are working on a new price checker application for their company's website. They've both written if/else statements using boolean expressions. Both are sure they've written the correct code. Which accountant is right? Should these accountants even be writing code? Here's our solution.

EXERCISE SOLUTION

From "Exercise"

Remember we said pseudocode often isn't perfect? Well, we actually left something out of our original pseudocode: we're not telling the user if her guess is a HIT or a MISS. Can you insert the missing pieces of code in the proper place to correct this? Here's our solution.

SHARPEN YOUR PENCIL SOLUTION

From "Sharpen your pencil"

What do you think of this first attempt to write the code to detect when a ship is hit? Does it look more complex than it needs to be, or are we repeating code in a way that seems a bit, well, redundant? Could we simplify it? Using what you know of the || operator (that is, the boolean OR operator), can you simplify this code? Here's our solution.

JAVASCRIPT CROSS SOLUTION

From "JavaScript cross"

How does a crossword puzzle help you learn JavaScript? The mental twists and turns burn the JavaScript right into your brain! Here's our solution.
