



7

CREATING A HANGMAN GAME

In this chapter we'll build a Hangman game! We'll learn how to use dialogs to make the game interactive and take input from someone playing the game.

Hangman is a word-guessing game. One player picks a secret word, and the other player tries to guess it.

For example, if the word were *TEACHER*, the first player would write:

The guessing player tries to guess the letters in the word. Each time they guess a letter correctly, the first player fills in the blanks for each occurrence of that letter. For example, if the guessing player guessed the letter *E*, the first player would fill in the *Es* in the word *TEACHER* like so:

__E____E__

When the guessing player guesses a letter that isn't in the word, they lose a point and the first player draws part of a stickman for each wrong guess. If the first player completes the stickman before the guessing player guesses the word, the guessing player loses.

In our version of Hangman, the JavaScript program will choose the word and the human player will guess letters. We won't be drawing the stickman, because we haven't yet learned how to draw in JavaScript (we'll learn how to do that in Chapter 13).



INTERACTING WITH A PLAYER

To create this game, we have to have some way for the guessing player (human) to enter their choices. One way is to open a pop-up window (which JavaScript calls a *prompt*) that the player can type into.

CREATING A PROMPT

First, let's create a new HTML document. Using **File ▶ Save As**, save your *page.html* file from Chapter 5 as *prompt.html*. To create

a prompt, enter this code between the `<script>` tags of *prompt.html* and refresh the browser:

```
var name = prompt("What's your name?");  
console.log("Hello " + name);
```

Here we create a new variable, called `name`, and assign to it the value returned from calling `prompt("What's your name?")`. When `prompt` is called, a small window (or *dialog*) is opened, which should look like Figure 7-1.

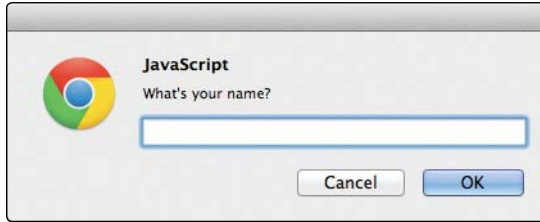


Figure 7-1: A prompt dialog

Calling `prompt("What's your name?")` pops up a window with the text “What’s your name?” along with a text box for input. At the bottom of the dialog are two buttons, Cancel and OK. In Chrome, the dialog has the heading *JavaScript*, to inform you that JavaScript opened the prompt.

When you enter text in the box and click OK, that text becomes the value that is returned by `prompt`. For example, if I were to enter my name into the text box and click OK, JavaScript would print this in the console:

```
Hello Nick
```

Because I entered *Nick* in the text box and clicked OK, the string “Nick” is saved in the variable `name` and `console.log` prints “Hello ” + “Nick”, which gives us “Hello Nick”.

NOTE

The second time you open any kind of dialog in Chrome, it adds an extra line to the dialog with a checkbox saying, “Prevent this page from creating additional dialogs.” This is Chrome’s way of protecting users from web pages with lots of annoying pop-ups. Just leave the box unchecked for the exercises in this chapter.

WHAT HAPPENS IF YOU CLICK CANCEL?

If you click the Cancel button, prompt returns the value `null`. In Chapter 2, we learned that you can use `null` to indicate when something is intentionally empty.

Click Cancel at the dialog, and you should see this:

Hello null

Here, `null` is printed as a string by `console.log`. Normally, `null` isn't a string, but since only strings can be printed to the console and you told JavaScript to print `"Hello " + null`, JavaScript turns the value `null` into the string `"null"` so it can be printed. When JavaScript converts a value into another type, it's called *coercion*.

Coercion is an example of JavaScript trying to be clever. There isn't any way to combine a string and `null` using the `+` operator, so JavaScript does its best with the situation. In this case, it knows it needs two strings. The string version of `null` is `"null"`, which is why you see the string `"Hello null"` printed.

USING CONFIRM TO ASK A YES OR NO QUESTION

The `confirm` function is a way to take user input without a text box by asking for a yes or no (Boolean) answer. For example, here we use `confirm` to ask the user if they like cats (see Figure 7-2). If so, the variable `likesCats` is set to `true`, and we respond with “You're a cool cat!” If they don't like cats, `likesCats` is set to `false`, so we respond with “Yeah, that's fine. You're still cool!”



```
var likesCats = confirm("Do you like cats?");
if (likesCats) {
  console.log("You're a cool cat!");
} else {
  console.log("Yeah, that's fine. You're still cool!");
}
```

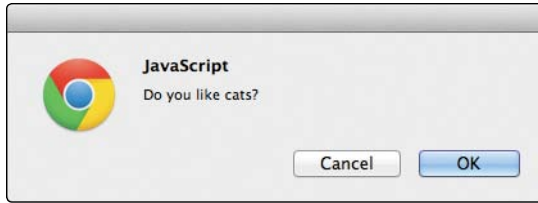


Figure 7-2: A confirm dialog

The answer to the confirm prompt is returned as a Boolean value. If the user clicks OK in the confirm dialog shown in Figure 7-2, `true` is returned. If they click Cancel, `false` is returned.

USING ALERTS TO GIVE A PLAYER INFORMATION

If you want to just give the player some information, you can use an alert dialog to display a message with an OK button. For example, if you think that JavaScript is awesome, you might use this alert function:

```
alert("JavaScript is awesome!");
```

Figure 7-3 shows what this simple alert dialog would look like.

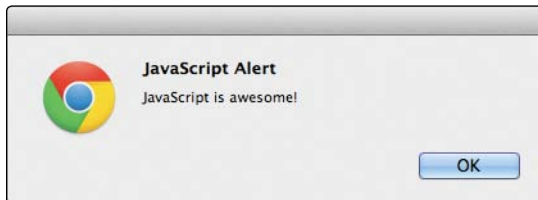


Figure 7-3: An alert dialog

Alert dialogs just display a message and wait until the user clicks OK.

WHY USE ALERT INSTEAD OF `console.log`?

Why use an alert dialog in a game instead of using `console.log`? First, because if all you want to do is tell the player something, using alert means the player doesn't have to interrupt game play to open the console to see a status message. Second, calling alert (as well as prompt and confirm) pauses the JavaScript interpreter

until the user clicks OK (or Cancel, in the case of prompt and confirm). That means the player has time to read the alert. On the other hand, when you use `console.log`, the text is displayed immediately and the interpreter moves on to the next line in your program.



DESIGNING YOUR GAME

Before we start writing the Hangman game, let's think about its structure. There are a few things we need our program to do:

1. Pick a random word.
2. Take the player's guess.
3. Quit the game if the player wants to.
4. Check that the player's guess is a valid letter.
5. Keep track of letters the player has guessed.
6. Show the player their progress.
7. Finish when the player has guessed the word.

Apart from the first and last tasks (picking a word for the player to guess and finishing the game), these steps all need to happen multiple times, and we don't know how many times (it depends on how well the player guesses). When you need to do the same thing multiple times, you know you'll need a loop.

But this simple list of tasks doesn't really give us any idea of what needs to happen when. To get a better idea of the structure of the code, we can use *pseudocode*.

USING PSEUDOCODE TO DESIGN THE GAME

Pseudocode is a handy tool that programmers often use to design programs. It means “fake code,” and it's a way of describing how a program will work that looks like a cross between written English and code. Pseudocode has loops and conditionals, but other than

that, everything is just plain English. Let's look at a pseudocode version of our game to get an idea:

```
Pick a random word

While the word has not been guessed {
    Show the player their current progress
    Get a guess from the player

    If the player wants to quit the game {
        Quit the game
    }
    Else If the guess is not a single letter {
        Tell the player to pick a single letter
    }
    Else {
        If the guess is in the word {
            Update the player's progress with the guess
        }
    }
}

Congratulate the player on guessing the word
```

As you can see, none of this is real code, and no computer could understand it. But it gives us an idea of how our program will be structured, before we get to actually writing the code and having to deal with the messy details, like *how* we're going to pick a random word.

TRACKING THE STATE OF THE WORD

In the previous pseudocode, one of the first lines says, “Show the player their current progress.” For the Hangman game, this means filling in the letters that the player has guessed correctly and showing which letters in the secret word are still blank. How are we going to do this? We can actually keep track of the player's progress in a similar way to how traditional Hangman works: by keeping a collection of blank spaces and filling them in as the player guesses correct letters.

In our game, we'll do this using an array of blanks for each letter in the word. We'll call this the answer array, and we'll fill it with the player's correct guesses as they're made. We'll represent each blank with the string “_”.

The answer array will start out as a group of these empty entries equal in number to the letters in the secret word. For example, if the secret word is *fish*, the array would look like this:

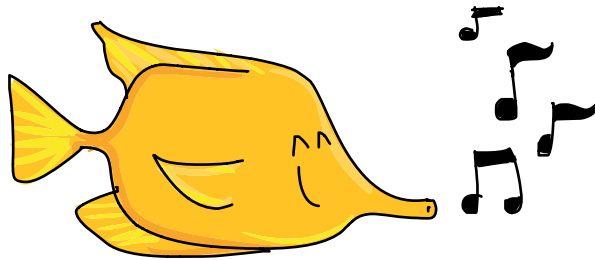
```
["_ ", "_ ", "_ ", "_ "]
```

If the player correctly guessed the letter *i*, we'd change the second blank to an *i*:

```
["_ ", "i", "_ ", "_ "]
```

Once the player guesses all the correct letters, the completed array would look like this:

```
["f", "i", "s", "h"]
```



We'll also use a variable to keep track of the number of remaining letters the player has to guess. For every occurrence of a correctly guessed letter, this variable will decrease by 1. Once it hits 0, we know the player has won.

DESIGNING THE GAME LOOP

The main game takes place inside a while loop (in our pseudo-code, this loop begins with the line “While the word has not been guessed”). In this loop we display the current state of the word being guessed (beginning with all blanks); ask the player for a guess (and make sure it's a valid, single-letter guess); and update the answer array with the chosen letter, if that letter appears in the word.

Almost all computer games are built around a loop of some kind, often with the same basic structure as the loop in our Hangman game. A game loop generally does the following:

1. Takes input from the player
2. Updates the game state
3. Displays the current state of the game to the player

Even games that are constantly changing follow this same kind of loop—they just do it *really* fast. In the case of our Hangman game, the program takes a guess from the player, updates the answer array if the guess is correct, and displays the new state of the answer array.

Once the player guesses all letters in the word, we show the completed word and a congratulatory message telling them that they won.

CODING THE GAME

Now that we know the general structure of our game, we can start to go over how the code will look. The following sections will walk you through all the code in the game. After that, you'll see the whole game code in one listing so you can type it up and play it yourself.

CHOOSING A RANDOM WORD

The first thing we have to do is to choose a random word. Here's how that will look:

```
❶ var words = [  
    "javascript",  
    "monkey",  
    "amazing",  
    "pancake"  
];  
  
❷ var word = words[Math.floor(Math.random() * words.length)];
```

We begin our game at ❶ by creating an array of words (*javascript*, *monkey*, *amazing*, and *pancake*) to be used as the source of our secret word, and we save the array in the `words` variable. The words should be all lowercase. At ❷ we use `Math.random` and `Math.floor` to pick a random word from the array, as we did with the random insult generator in Chapter 3.