**JavaScript Project**

Introduction

There are a lot of theories out there about the best way to learn to program. If there’s one thing almost everyone agrees on, it’s that there’s no substitute for actually building something. Putting together projects from scratch is the single best way to cement your skills (and show them off), and an important early experience for anyone serious about learning to code.

This final project builds off the skills you learned in the Javascript course, and also pre-supposes knowledge of jQuery. If you haven’t worked your way through the jQuery course on Codecademy, do that before starting this project (don’t worry, it’s much quicker than the Javascript track!).

In this project, you’ll build a game from scratch. You’ll leave Codecademy’s learning environment and build locally, on your own computer, testing and running the game in the browser.

You will be walked through the steps required, but not as closely as you’re used to on Codecademy. Our goal is to give you the experience of building without training wheels. The project will be (and should be!) challenging. At times, you might get really frustrated. You’ll also learn a lot about what it means to put together a project from the ground up, and come out with tangible evidence of the skills you’ve learned.



GameBook—Code Your Own Adventure Part 3!

For the JavaScript final project, you’re going to use key JavaScript concepts (string manipulation, conditional flow, looping and object-oriented programming) to build an engaging, multifaceted Code Your Own Adventure game. You’re already familiar with this type of game from JavaScript [Unit 1](https://www.codecademy.com/courses/javascript-beginner-en-x9DnD/0/1?curriculum_id=506324b3a7dffd00020bf661) and [Unit 5](https://www.codecademy.com/courses/javascript-beginner-en-ZA2rb/0/2?curriculum_id=506324b3a7dffd00020bf661). For the final project, the task is to come up with your own and bring it to life in code. The instructions for this project are very open. We ask that you adhere to some general user stories. Other than that, your imagination is the only limit.

**What Are User Stories?**

User stories are descriptions from the perspective of the end-user meant to capture the essence of a program’s features. In the case of your game, the end-user is whoever is playing your game. User stories are usually in the first-person, or in other words, “I” statements. For example, “As a user, I can play the game in Google Chrome and Mozilla Firefox.”

**Code Your Own Adventure User Stories**

As a user:

* I **don’t** want the game to be exactly the same every time I play.
* I want there to be different outcomes based on different choices I make.
* I want to interact with multiple characters.
* I want characters I interact with to remember things about me.
* I want the game to end once I win or lose.

The toolset

*Code Your Own Adventure* will employ the same tools we used for the HTML/CSS final project. If you’ve already completed that project, you’ll be very familiar with these:

**Text Editor.** Software that programmers write code in. Below, you’ll get instructions on how to install Sublime Text, a popular text editor.

**Web Browser.** Google Chrome is the preferred web browser for this project, because it has a great built-in developer tool feature. We’ll learn more about how to use this feature to test your JavaScript later on.

If you’re already set up on Sublime and Google Chrome, feel free to skip to [Step 2.](https://s3.amazonaws.com/codecademy-content/projects/FinalProjects/js/02.html)

Web browser

If you don’t already have Google Chrome on your computer, download it [here](https://www.google.com/chrome/browser/desktop/) Chrome is the preferred web browser for most developers, and a good browser to start getting familiar with (part of what’s made it popular is its Dev Tools, which you can read about [here](https://www.codecademy.com/articles/use-devtools)).

Text Editor

Text editors allow for language-specific syntax highlighting, automatic code indentation and other very useful features for writing code. You’ll use one to write the code files that will eventually become your website.

**Download Sublime Text**

[Sublime Text](http://www.sublimetext.com/2) is the text editor of choice for most developers working to build Codecademy. You can download free trial versions [here](http://www.sublimetext.com/2). If you fall in love with Sublime Text (as millions of other developers have) you can buy it!

Quick Guide to Using Sublime Text (Mac)

**Download**

1. On the Sublime Text download page, click on link for OS X (OS X 10.6 or later is required).
2. After it downloads, open the application either from your browser’s download section or from Finder.
3. Move Sublime Text into your applications folder.
4. Click on the “S” icon to get started on your first code file.

**Writing Code**

1. Explore the Sublime Text navigation pane at the top of the screen. You’ll find the basics there, like File > New, File > Save, File > Open, etc., which you’ll need to get started.
2. Open a new file. When you save it, assign to it the file extension of any programming language you’re familiar with. For HTML, save your file with a `.html` file extension. This will enable syntax-specific code highlighting and auto-indentation, like what you’re used to on Codecademy.

Quick Guide to Using Sublime Text (Windows)

1. On the Sublime Text download page, click on the link for Windows.
2. After it downloads, open the application either from your browser’s download section or from your computer’s Downloads folder.
3. You may get a security alert asking if it’s okay to run this software. Yes, it’s okay. Click “Run.”.
4. Follow the prompts in the Sublime Text II Setup Wizard to install the software.
5. Find Sublime Text II in your file system and open it.
6. See step 2 in **Quick Guide to Using Sublime Text (Mac)** above. It’s the same as Windows from this point.

Come Up With Your Adventure

Before you dive into thinking programmatically about your game, take a little unstructured time to let your mind wander and play. Discover fun plotlines, characters and themes that you think will sustain your interest.

Try to find a game topic you really want to bring to life. As you translate your story into code, you’re likely to have moments of difficulty and frustration. If you continue to be entertained by the spirit of your game, you’ll be much more motivated to power through any technical changes.

Game Pitches and Starter Examples

Hacking your imagination comes very easy to some folks, but it may feel uncomfortable for others. Try your best to come up with a completely unique game idea, but if you’re anxious and just want to get coding, here are a few adventure suggestions to get you started.

* Lord of the Rings theme—the classic epic by J.R.R. Tolkien.
* Star Wars theme—George Lucas' popular space opera.
* Harry Potter theme—J.K. Rowling's modern classic.
* Current Event theme—find something in the news that catches your attention.
* Current Event theme—your favorite sporting event.

Beyond Yes and No

In Codecademy’s JavaScript course, Code Your Own Adventure challenges 1 and 2 took in simple user input, like “Yes” and “No” and responded accordingly. For your GameBook, don’t be constrained by so few choices. At various points in your game, you can prompt your users for multiple inputs. You could try things like the “Browser Prompt Example” below, and keep your code neat and readable with a [switch statement](https://www.codecademy.com/courses/javascript-beginner-en-qDwp0/0/4?curriculum_id=506324b3a7dffd00020bf661).

**Browser Prompt Example**

**JavaScript**

Bilbo: What would you like to eat, my friend?

Your choices: Cream, Honey Cake, Lembas Bread, Petty Dwarf Roots

Cancel

Ok

Create a Program Outline

Great programs begin with strong program design, the logic and organization that inform your code. Now that you have an idea of what your GameBook will be about, translate it into an outline by answering these questions:

* What are the important things (nouns and verbs) in your *game-world?*
* Who are your characters?
* What attributes do your characters have?
* What is your *win condition*?
* What is your *lose condition?*
* What actions lead to a win?
* What actions (or inaction) lead to a loss?

**Outline Example:**

* /\*
* Title: Simpsons Bullies
* -----------------------
* You and Bart Simpson aare trying to stand up to Nelson Muntz before he picks on you and says "Ha ha," in the Springfield Elementary schoolyard.
* ----------
* ----------
* GAME WORLD
* ----------
* ----------
* Characters
* ----------
* Attributes
* ----------
* - You (hero)
* - name, age (grade), ways to stand up to Nelson
* - Bart
* - gives advice, stands up to Nelson (e.g. slingshot, make/trhow water balloons, acorns)
* - Nelson
* - picks on you and Bart (e.g., insults, taunts)
* - schoolyard kids
* - name, grade, ways to help stand up to Nelson (laugh, make/throw water balloons, push)
* Wins and Losses
* ----------------
* - Win: stand up to Nelson
* - Lose: get picked on by Nelson
* \*/

A Brief Look at MVC

Before we dive into game structure, it’ll be helpful to consider how to design your code on a high level. A popular framework for organizing code is MVC. We recommend using MVC here to get familiar with the system and set yourself up well as you start coding *Gamebook*.

What is MVC? MVC is a programming design pattern that applies to a wide variety of program types, including games. It’s short for Model-View-Controller. Let’s briefly walk through each:

* **Model.** Model code typically reflects real-world things. But in the case of your game, the model should consist of key objects and characters in the *game-world*. In *Simpsons Bullies* for example, the classes and objects derived from the outline – e.g., schoolyard kids, Nelson and Bart – are considered part of the model, because they reflect the core constructs of the *game-world*.
* **View.** View code are all the functions that directly interact with the user. In the case of *GameBook*, they are window prompts (e.g., alert, confirm, prompt). The user interacts with the view through those prompts, for example, typing in “YES” or “NO” and pressing “OK.”
* **Controller.** Controller code acts as a liaison between the model and the view, receiving user input and deciding what to do with it. For example, if the user has typed “YES” into a window prompt, it’s up to the controller to determine what model code it should execute in response. Controller logic is implemented via conditional statements, loops and other code constructs that execute different decisions based on varying inputs. We’ll give more detail on the controller as we dive deeper into the building process.

Step 3: Build Your Program

Estimated time: 2 hours

For GameBook, you’ll find that create-your-own-ending stories have a few essential elements, like characters, choices for the reader, and a variable ending. There are many ways to translate these ideas into code. The following is one way to think about organizing your project.

Set Up Your Folder Directory

First and foremost, create your folder structure. Try these steps to get it set up:

1. Create index.html inside of your main folder (called “GameBook” in our example below).
2. Create a folder for JavaScript files, and create a main.js file.
3. In main.js, write an alert().
4. Link the main.js file to the head of index.html.
5. Open index.html in the Browser.
6. Confirm that the alert works and you’re in business.

**Sample Folder Structure**

Object-Oriented Thinking: Creating Classes

You learned about JavaScript [constructor notation](https://www.codecademy.com/courses/objects-ii/0/1?curriculum_id=506324b3a7dffd00020bf661) in the Codecademy JavaScript path. Using constructor functions to create your game’s classes may be a good choice, because they organize your code in an easy-to-read, easy-to-change format. Here’s one such example, taken from the game narrative outline in the previous page.

**Simpsons Bullies OO Example: SchoolyardKid Constructor**

1. var SchoolyardKid = function(name) {
2. this.name = name;
3. this.canHelp = false;
4. this.responses = {
5. homework: "Sorry, I'm doing homework.",
6. scared: "No way, Nelson is a beast!",
7. brave: "Sure, I've been meaning to get back at that jerk."
8. };
9. }

In this example, we define a class that you could use to create multiple characters. There are multiple attributes within the class that can help us move the game forward. In your own game, the attributes you use can be anything that makes sense to your plot.

We also want to program for a chance situation. Check out the “responses” attribute on line 4. Structures like this can allow users to interact with your virtual characters. Depending on the user input, one of the “responses” will be sent back to the user, which will help move the story along.

Mapping Your Oultine "Actions" to Functions with the Prototype Pattern

In JavaScript [Introduction to Objects II](https://www.codecademy.com/courses/objects-ii/2/5?curriculum_id=506324b3a7dffd00020bf661), you learned how to “extend” a prototype in order to add functions to your class objects. Here’s an example of the syntax:

**Using 'prototype' Example I**

1. className.prototype.newFunction = function() {
2. // write the function here...
3. };

You can use this pattern to map character attributes from your outline to the functions they belong to in your game code.

For example, in *Simpsons Bullies*, the user will ask schoolyard kids for help to stand up to Nelson. Here’s a function that shows how the schoolyard kid will decide to help the user or not.

**Using 'prototype' Example II**

1. SchoolYardKid.prototype.randomResponse = function() {
2. var randomNum = Math.floor((Math.random() \* 3) + 1);
3. if (randomNum === 1) {
4. return this.responses.homework;
5. } else if (randomNum === 2) {
6. return this.responses.scared;
7. else {
8. return this.responses.brave;
9. }
10. };

Line 12 produces a random number between 1 and 3 using the JavaScript Math object, which you encountered in the [Dragon Slayer](https://www.codecademy.com/courses/javascript-beginner-en-mrTNH-6VIZ9/0/2?curriculum_id=506324b3a7dffd00020bf661) lesson. Based on the random number generated, the if/elsestatement that follows will determine how a schoolyard kid will respond to the user’s plea for help.

Use your discretion about when you need a class and when a simple object will do. In Simpsons Bullies, there will only ever be one bully: Nelson. Therefore, creating a single object with the name Nelson which houses Nelson’s functions and attributes will suffice. Thinking critically about these sorts of program design decisions will result in orderly, readable and maintainable code.

Organize Your Code: Separate Files

Your project is going to have multiple game-world characters interacting with each other. In code-speak, there is a flow to how each interaction might work.

1. First, you’ll have a character that’s defined by a class (this is the Model part in MVC).
2. Second, you’ll have what the user sees and interacts with (this is the View part in MVC).
3. And lastly, you’ll need the code that makes your game-world characters come to life (this is the Controller in MVC).

Since there are three essential functions your code will serve, it can be helpful to separate your code by its purpose. For example, you might have a SchoolyardKid.js, a Bart.js, a controller.js, and an index.html. Use whatever separation of files that makes sense for you. Since this is a relatively small program, it’s also possible to put everything in one file.

One thing to think about: how you organize your files should make sense to you and for someone else who might want to update or modify your program. You can make it easier to understand your code by inserting comments in-line with your code on what your code is doing.

You can make comments within your code by using // for a single line, or using /\* ... \*/ for multiline comments.

**Sample Separated Folder Structure**

Test Along the Way

As you move forward, find a process to build a feature and then test to see if it works. This could mean writing some example code, testing to see if it works, and then writing a little more code to go on top of it, and testing both pieces together. Little by little you’ll make your way to an enthralling, can’t-put-it-down, *GameBook!*

Tying It Together in index.html

If you choose to use multiple files to organize your JavaScript, remember that you’ll need to source each of them in the <head>...</head> section of your index.html file. Once you do, the code in each file will be available to you when you run index.html in the browser. See below for an example.

**Sourcing Javascript Example**

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <title>GameBook</title>
5. <script src="js/model/characters.js"></script>
6. <script src="js/controllers/controller.js"></script>
7. </head>
8. <body>
9. </body>
10. </html>

Step 4: Build Your Program (Continued)

Estimated time: 3-5 hours

A Simple Game Controller

As mentioned earlier, the controller drives the events of the game. It responds directly to user input and weaves together the experience of forward progress. Simpsons Bullies has a JavaScript file called controller.js to keep this code organized.

**Controller Example I**

1. function newGame() {
2. // your game controller code goes here
3. };

As illustrated above in **Controller Example I**, you may choose to create a controller function and place all the code that drives your game here. But remember, you need to invoke the function before testing your game in the browser, or it won’t run. See the example below.

**Controller Example II**

1. newGame();

So what type of code lives in the controller?

Conditional Statements

When someone playing your game inputs data into a window prompt, this event is crucial to pushing your game forward. But how does your code know what to do with the input? Use your knowledge of if/else and switch statements to determine what should happen depending on different user inputs.

**Controller Example III**

1. function newGame() {
2. var userInput = prompt("Please input your data here!");
3. if (userInput === "X") {
4. // code that executes one outcome
5. } else if (userInput === "Y") {
6. // code that executes a second outcome
7. else {
8. // code that executes a third outcome
9. }
10. };

The if/else statement in **Controller Example III** demonstrates how to think about using a controller to determine outcomes based on user input. User input enters the program via the window prompt (line 2). From this point, it’s up to the if/else to determine how the game will progress.

**View Functions.** In large, complex applications, code that determines view logic would live in its own separate files. However, for our simple game, it’s perfectly fine to invoke view functions within the controller. What’s most important is that you learn to distinguish view from controller logic. If you’re unsure, look carefully at **Controller Example III**. One line of code in particular is interacting directly with the user. Which one?

**Game Loops.** In [Dragon Slayer](https://www.codecademy.com/courses/javascript-beginner-en-mrTNH-6VIZ9/0/3?curriculum_id=506324b3a7dffd00020bf661), you built a very simple game loop, code that executes continuously while some condition is true or false. If your game’s plot involves actions that should repeat until the user has submitted the desired input, consider using a game loop.

**Game Loop Example**

1. var backDown = false;
2. function gameLoop() {
3. while (backDown === false) {
4. var userMove = confirm("Nelson won't back down! Try again!");
5. var nelsonStatus = Math.floor((Math.random() \* 5) + 1);
6. if (nelsonStatus === 4) {
7. alert("Nelson backed down -- Nice work!");
8. backDown = true;
9. };
10. };
11. };

**Note: Exit Conditions**

If the user triggers an event that causes her to win or lose the game, how do you tell JavaScript to stop running your code? If your controller is contained within a function, as shown in **Controller Example I** using the return keyword will exit the function. Window prompts below it will not “pop up,” and your user will have in effect exited the game.

Enhancing Your "View" Code

As you get more familiar with using window prompts, you may start to feel constrained by how little this view format is capable of expressing. Within such limitations, you do have a few visual options that can enhance the user’s experience of your game (a little).

**New Lines.** Backspace, \, in many programming languages is an *escape character*. If you use it with a new line character, n, you can break up your window prompt text into different sections. This could be useful for portraying dialogue between two characters. See examples below.

**Escape Character / New Line Example I**

1. alert("Look! A single line break:\nAnd two line breaks:\n\nI feel so isolated down here.");

**Escape Character / New Line Example II**

**JavaScript Alert**

Look! A single line break:

And two line breaks:

I feel so isolated down here.

Ok

**Tabs, White-space and Symbols.** Using \t will insert a tab into your string. A few \t ‘s in a row will effectively center text in a window prompt. For more exact positioning of text, you can use white-space (just hit the spacebar inside your string). Finally, the special symbols on your keyboard provide some rudimentary, but effective ways to further decorate your views.

**Escape Character / New Line Example II**

**JavaScript Alert**

\*\*\*YOU LOSE\*\*\*

hahahahahahahaha!

Ok

Building in Error Messages

When giving your user a choice, they might enter a value that you don’t expect. Like in The Matrix, if Morpheus offered you a blue pill and a red pill, how would he react if you insisted on taking taking a yellow pill? After a long philosophical conversation, he would show you the options again, and tell you to answer either, “Red” or “Blue.”

If a user accidentally submits an invalid answer to one of your prompts, you can give them an informative error message, then redirect them back to the original question to try again. This is done easily by utilizing an else statement. Within the else statement, use an alert(); to provide a useful error message, then call the question’s function again to let them try again. It might look something like this:

1. function questionOne() {
2. var userAnswer = prompt("You choose: A or B?");
3. if (userAnswer ==="A") {
4. alert("Glad you picked A.");
5. questionTwoA();
6. } else if (userAnswer === "B") {
7. alert("B was a brave choice.");
8. questionTwoB();
9. else {
10. alert("Make sure to type in the letter A or B only.");
11. questionOne();
12. }
13. };

**JavaScript Alert**

Make sure to type in the letter A or B only.

Ok

Debugging

As your plot line thickens, chances are your game will break or you’ll find that one of your conditional statements keeps on giving you the same result over and over. This is where debugging comes in.

One of the hardest parts of debugging is finding where the issue occurs within your code. A few simple methods to find a bug in your code include:

1. Isolating the faulty code by commenting out lines of new or suspect code.
2. Using console.log() messages to find which pieces of code are being executed, and which ones aren’t. Then you can look inside the Developer Tools Console window to see which messages are being printed to the console when you run your program.
3. If the bug is within a conditional statement (like an if/else or a for/while loop), set the condition to “true” or “false” to make sure each side of the statement is functioning correctly.

There are many more methods of debugging if these don’t work. Debugging can be an arduous and frustrating process, but it’s also one of the core skills of working developers. You can find a more complete guide to debugging JavaScript [here](https://developer.chrome.com/devtools/docs/javascript-debugging). Happy bug hunting!

Step 5: Test Your Project

Estimated time: 2+ hours

Big Picture Time

Go through the checklist on page one of this project. Does your game meet all of the requirements the user is looking for? When playing through your game, does it make sense, and is it easy enough to follow? And most important of all, is it fun?

If so, you’re nearly finished; just a few more steps.

Break your game and find edge cases

Now it’s time to think of yourself as your program’s worst enemy. Try to think of anything a user could possibly do to sabotage your game, and try them. If the game breaks, go back to your code and build in ways to protect against those events.

For example, if there’s a multiple choice question, instead of answering with one of the listed options, answer with something else, like “bananas.” Or, even better, answer with an empty input box.

Non-standard interactions like this are sometimes called **edge cases**. Finding them is hard work and can take a good deal of creativity. One of the best ways to find edge cases is to watch someone else play your game without explaining to them how to do it. Take notes on where they get stuck, then revisit relevant portions of your code.

Iterate and make your *Gamebook* Stronger

When you feel like you’ve found answers for major edge cases, consider what else can be added. Adding features to an existing codebase is a huge part of programming, and good practice to have early on. When you feel your game is complete, challenge yourself to add at least one more feature or plot twist.

Here are some ideas of how you can make your *Gamebook* even better, from a user’s perspective:

* I want to be able to name my character, and to have the game remember my character’s name when responding to me.
* There needs to be a final boss battle, where I can attempt to defeat the main antagonist/villain (you did a similar project in [Unit 4](https://www.codecademy.com/courses/javascript-beginner-en-mrTNH-6VIZ9/0/1?curriculum_id=506324b3a7dffd00020bf661)).
* I want to be able to type in a special cheat code that will make my destiny always end in a certain way.
* I want to have a badge of completion (as an image) if I win/lose, so I can share it on social media.